A CONGESTION AND MEMORY-AWARE FAILURE RECOVERY IN SD-WAN

by

MEYSAM SHOJAEE

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

at

Dalhousie University
Halifax, Nova Scotia
December 2020

Table of Contents

List of Tables

List of Figures

Abstract

In software-defined wide area networks (SD-WANs), link failure is a common occurrence creating heavy congestion and packet loss and degrading the application performance. Proactive failure recovery, i.e., preinstalling backup tunnels beforehand, is heavily used in SD-WAN to break the dependency on the controller and to enable fast rerouting. However, existing systems either lead to wasting the valuable network resources (such as bandwidth capacity and switch memory) because of vacant link capacity or impose non-realistic assumptions on the network topologies, such as the existence of link-disjoint routes or unlimited switch memory resources. We advocate a balanced network resource utilization instead. We argue that a reliable system must take into account multiple resources while planing a failure recovery and must be adaptable to multi-QoS traffic classes. Thus, in this thesis, we propose a novel multi-resource aware proactive recovery system in SD-WANs. Our system can be used in any network topology and is adaptable to environments with multiple QoS requirements. Moreover, it makes an optimized trade-off among the critical network resources and minimizes route stretch while recovering from a failure. In particular, we formulate the failure recovery problem as a multi-objective MILP optimization problem and evaluate it using CPLEX. Then we develop a heuristic to efficiently compute backup routes as the problem is NP-Hard. We implement a prototype of our system using the Ryu SDN controller and extensively evaluate it in Mininet over four real WAN topologies. The results show significant performance improvement of our failure recovery system compare to the state-of-the-art.

Acknowledgements

First and foremost, I would like to express my deep and sincere gratitude to my research supervisor, Dr. Israat Haque, for giving me the golden opportunity to do research and providing invaluable guidance throughout this research. Her insightful feedback pushed me to sharpen my thinking and brought my work to a higher level. It was a great privilege and honor to work and study under her guidance.

I would also like to express my special thank to my colleague, Dr. Miguel Neves, for his encouragement, insightful comments, and hard questions. He was always willing and enthusiastic to assist in any way he could throughout this research.

I thank all my fellow labmates in the Programmable and Intelligent Network (PINet) research lab for their constructive discussions. My special thanks go to one of my labmates, Dipon, for his friendship, empathy, and for all the fun we have had in the last two years.

I offer my deepest gratitude to my caring, loving, and supportive wife. Without her patience and continuous support, I could not have completed this thesis. Last but not the least, I would like to thank my parents for their love, prayers, and sacrifices for educating and preparing me for my future.

Chapter 1

Introduction

Over the past decade networks have come under increased traffic demands to support an ever-increasing array of applications ranging from financial services to electronic data interchange and big-data analytics. Software-defined networking (SDN) is a crucial technology needed to keep up with these new demands. SDN enables a centralized view of the distributed network for more efficient orchestration and automation of network services through software applications using open APIs. The core idea of SDN is decoupling the control and packet forwarding planes in the network. This way operators can use the software-based control plane to manage the entire network and its devices consistently without being tied to the underlying network infrastructure [1–3].

Large service providers, such as Google, Microsoft, and Facebook [4–6], are turning to SDN technology and deploy software-defined WAN (SD-WAN) to orchestrate data transmission. They usually adopt a centralized traffic engineering (TE) system to improve the performance of their SD-WANs. In these networks, a logically centralized controller maintains a global view of the network, and installs forwarding rules to the data plane, dictating the behavior of the forwarding devices. Usually the controller spread traffic across a number of paths between ingress-egress switch pairs.

In a SDN, a failure may occur in one the three layers: application layer, control layer, infrastructure layer (known as data plane). Application layer failures usually happens because of software bugs, which could result in configuration and network correctness violations. The control plane failure is when the controller fails to reconfigure a node in a timely manner because of control channel disruption or the controller malfunctioning itself. The data plane failure includes link or node failures, which happen when a link(s) is down or a node fails because of a software bugs or hardware malfunction. In this work, we focus on all possible single link failures in SD-WANs as this is the most common failure.

In SD-WANs, link failures are prevalent but undesirable because of the heavy congestion and packet loss, and the resulting application performance degradation. Typically, to deal with link failure, a TE system dynamically adjusts the traffic splitting across backup and alive paths to reroute the affected flows [7]. However, the existing TE systems either lead to wasting the valuable network resources (such as bandwidth capacity and switch memory) because of vacant link capacity or impose non-realistic assumptions on the network topologies, such as the existence of link-disjoint routes or unlimited switch memory resources [8].

In this thesis, we propose a proactive failure recovery system in SD-WANs, which takes into account multiple resources limitations. Our system is not confined to any network topology and can be used in any network topology. It is also adaptable to environments with multiple QoS requirements. Moreover, our system makes an optimized trade-off among the critical network resources and minimizes route stretch while recovering from a failure. We formulate the failure recovery problem as a multi-objective MILP optimization problem and develop a heuristic to efficiently compute backup routes as the problem is NP-Hard. We implement a prototype of our system using the Ryu SDN controller and extensively evaluate it in Mininet over four real WAN topologies. The results show that it can reduce the number packet loss, route stretch, and memory usage by up to 50%, 25%, and 20% than the state-of-the-art failure recovery scheme [8, 9].

1.1 Motivation

This thesis is motivated by the research gap in the existing failure recovery approaches in SD-WANs. In the following we present the crucial points being missed in the prior work.

Failure recovery approaches usually compute backup path for each ingress-egress switch pair and proactively install them in the switches. There exist many route selection algorithms such as k-shortest path, link-disjoint paths and oblivious-routing [9]. However, state-of-the-art failure recovery approaches for SD-WANs usually adopt link-disjoint paths as the primary tunnels [7,10,11]. This is because using link-disjoint paths leads to less number of affected flows in the case of a failure. This approach assumes the existence of multiple link-disjoint routes on a network topology, which is not the case in many real-word applications. For example, B4, Google SD-WAN topology, has two link-disjoint paths between almost all source-destination pairs [8].

Additionally, installing flow rules beforehand can lead to congestion as there is no information about the network load at the time of flow installation. An immediate solution to avoid congestion is to keep a certain portion of network capacity unutilized [10]. This keeps network utilization sufficiently low so that shifts in traffic can be absorbed when failures occur. This approach can provide high availability levels. However, as a down-side, it leads to wasting valuable resources of WAN providers as many links become underutilized, especially when the network is under the normal operation [9, 12].

SDN switches use expensive Ternary content accessible memory (TCAM) chips for processing traffic at line rates [13], which are power-hungry and have limited capacity [14–17]. In this sense, it is crucial to take switch memory constraints into account when allocating backup routes for fast failure recovery in SD-WANs. However, prior works fall short in this issue by commonly assuming switches have infinite memory resources [8, 9].

WAN traffic usually includes flows from different applications with different QoS requirements [18]. In order to meet these requirements, providers typically consider priority classes while routing traffic. However, most of the existing failure recovery approaches in SD-WAN do not consider priorities while rerouting flows through backup paths after a link failure. Consequently, high-priority flows may end up taking longer paths compared to low priority ones after the failover process, which degrades their performance [8, 9].

1.2 Contribution

In this thesis, we propose a failure recovery system for SD-WANs. Our system selects backup routes and allocates traffic rates on them for any single link failure in the network. Our system considers traffic flow with different QoS requirements and is not specialized to any specific network topology. We formulate the failure recovery problem as a multi-objective MILP optimization problem, which takes into account the link capacity, route length and switch memory usage. As our model is NP-hard and does not scale to large networks, we develop a heuristic that selects backup routes and allocated traffic rates to them [8].

To evaluate the performance of our system, we implement a prototype using the Ryu SDN controller and deploy the flow and group tables of OpenFlow protocol. Our open source code is available at [19]. We conduct an extensive evaluation of our system over several topologies with different characteristics. We compare the performance of our system with the state-of-the-art SD-WAN failure recovery scheme. Then, we illustrate that our system can quickly reroute the traffic with low delay, while uniformly distribute the traffic over the network and efficiently use the network resources. In summary, we make the following contributions:

- First, we formulate the failure recovery in SD-WAN as a MILP problem. We consider link capacity, route length and switch memory usage in our model. We evaluate our model in CPLEX and discuss its scalability.
- Next, We develop a heuristic to solve our model in a reasonable time and evaluate its efficiency in terms of the solution accuracy and runtime.
- We design a software-defined failure recovery system, called SafeGuard, which runs our heuristic in SD-WAN environment. Then, we implement a prototype of SafeGuard using the Ryu SDN controller and make our code publicly available at [19].
- We perform an extensive evaluation of SafeGuard over four real WAN topologies. We show that SafeGuard can quickly recover form single link failures and efficiency use the network resource compared to the state-of-the-art.

1.3 Thesis Outline

The rest of this thesis is organized as follows. In chapter 2, we present the necessary background on SDN, and review the related works in the area. In chapter 3, we present our failure recovery system in SD-WAN, called SafeGuard. We model the failure recovery as a MILP problem. We evaluate the model and elaborate on its complexity, which leads us to design a heuristic for solving the model in reasonable time. Then we implement our system using an SDN emulator and show the performance of SafeGuard. In chapter 4, we extend SafeGuard to environments with multiple QoS requirements and non-unitary switch memory demand (i.e. requiring more than one flow rule for a single flow between an ingress-egress switch pair). We also extensively evaluate SafeGuard in different settings and compare it with the state-of-the-art failure recovery approach in SD-WAN. Lastly, we conclude the thesis in chapter 5 and discus the future work directions.

Chapter 2

Background and Related Work

In this chapter, we present the necessary background to better understand this thesis in the first subsection. We start with software-defined networking paradigm and its most notable protocol, OpenFlow. Then we discuss about the data plane reliability in software-defined networking and look at the traffic engineering as a tool for managing network failures. In the second subsection, we review prior literature related to our work.

2.1 Background

2.1.1 Software-Defined Networking

The evolving technology of software-defined networking (SDN) has become increasingly popular since it emerged in the early 2010s. SDN aims to make networks flexible, giving the hope to overcome the limitations faced by legacy networks [3].

Having no standard open interface in legacy networks, most network devices are closed-boxes that run proprietary software. Therefore, to implement network-wide policies, each individual network device need to be separately configured using low-level commands that vary across vendors. This time-consuming configuration risks service disruption, discourages network changes, and increases both the capital and operational costs of running a network. Besides, in legacy networks, the lack of centralization and heterogeneity of network devices imposes constraints to automatic reconfiguration and dynamic response mechanisms to the ever-changing nature of the network. The vertical integration of the legacy networks, on the other hand, hinders innovation and evolution, making it hard to keep pace with modern technology [2].

SDN is revolutionizing the way in which we design and manage networks by decoupling *control plane* (that configures the network and engineers the traffic) from the

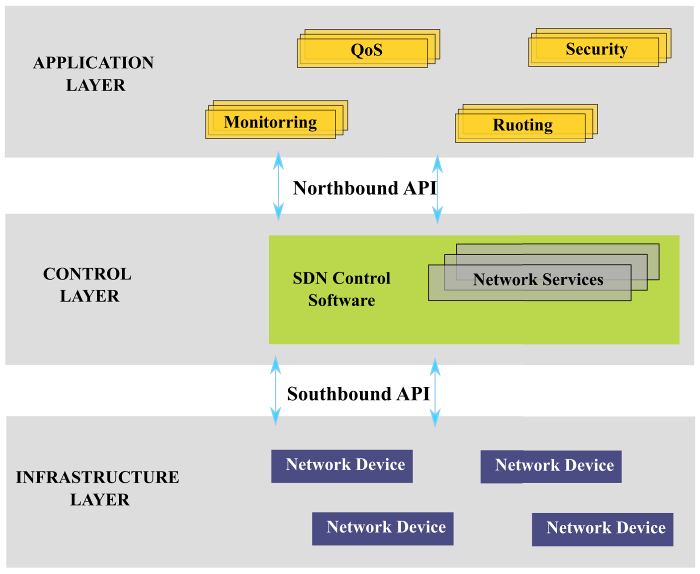


Figure 2.1: SDN architecture.

data plane (the underlying hardware infrastructure). The logically centralized controller uses a standard protocol, such as *de facto* OpenFlow protocol [20], to configure the devices. This enables network programmability that leads to flexible network configuration and rapid innovation. As depicted in Fig. 2.1, SDN architecture comprises three layers (planes) and two APIs:

1. **Infrastructure layer**: Also known as data plane, includes physical and virtual switches, such as OpenFlow switches, that are responsible for data forwarding and statistics storage. Network devices take actions on the incoming packets, such as forwarding to specific ports, drop, forward to the controller, etc., using well-defined instruction sets (e.g., flow rules).

2. **Southbound API (SB API)**: Provides a communication protocol between the SDN Controller and the forwarding devices in the data plane. It directly pushes the instruction set in the forwarding devices.

3. **Control layer**: Consists of one or more SDN controllers that maintains a logically centralized network view and acts as the *brain* of the network. The controller provides core network functions using the global knowledge of the underlying network. According to the network policy, it configures forwarding devises through SB API.

4. **Northbound API (NB API)**: Provides a communication interface, which can be used to configure switches' forwarding policies. It abstracts the low-level instruction that enables a particular component of a network to communicate with a higher-level component.

5. **Application layer**: Consists of set of applications and uses abstractions provided by the control layer to implement network applications.

2.1.1.1 OpenFlow Protocol

OpenFlow protocol enables secure communication between the control and data planes. It also provides the necessary APIs to enable the controller to configure data plane devices. An OpenFlow switch consists of one or more flow tables, a group table and an OpenFlow channel to communicate with the controller [21]. The controller uses OpenFlow channel to add, modify, or delete flow or group entries. Fig. 2.2 represents a sketch of an OpenFlow switch.

Each flow table comprises a set of elements, called flow entries, which is used to match and process packets. Each flow entry comprises a set of *match fields* for matching packets, a *priority* for matching precedence, a set of *counters* to track packets, and a set of *instructions* to apply.

An instruction associated with a flow entry describes the OpenFlow processing that happen when a packet matches the flow entry. It either modifies pipeline processing, such as direct the packet to another flow table, or contains a set of actions that describes packet forwarding, packet modification and group table processing.

Additionally, each flow table must include a table-miss flow entry to process table misses. The table-miss flow processes packets unmatched by other flow entries in the flow table. A table-miss flow may send packets to the controller, drop packets or direct packets to a subsequent table.

The group table consists of group entries that facilitates more complex and specialized packet operations that cannot easily be performed using flow entries. Each group entry (group for short) contains a list of action buckets and a mechanism that enables choosing one or more of those buckets to apply to packets directed to the group. An action bucket is a list of actions and associated parameters. The exact

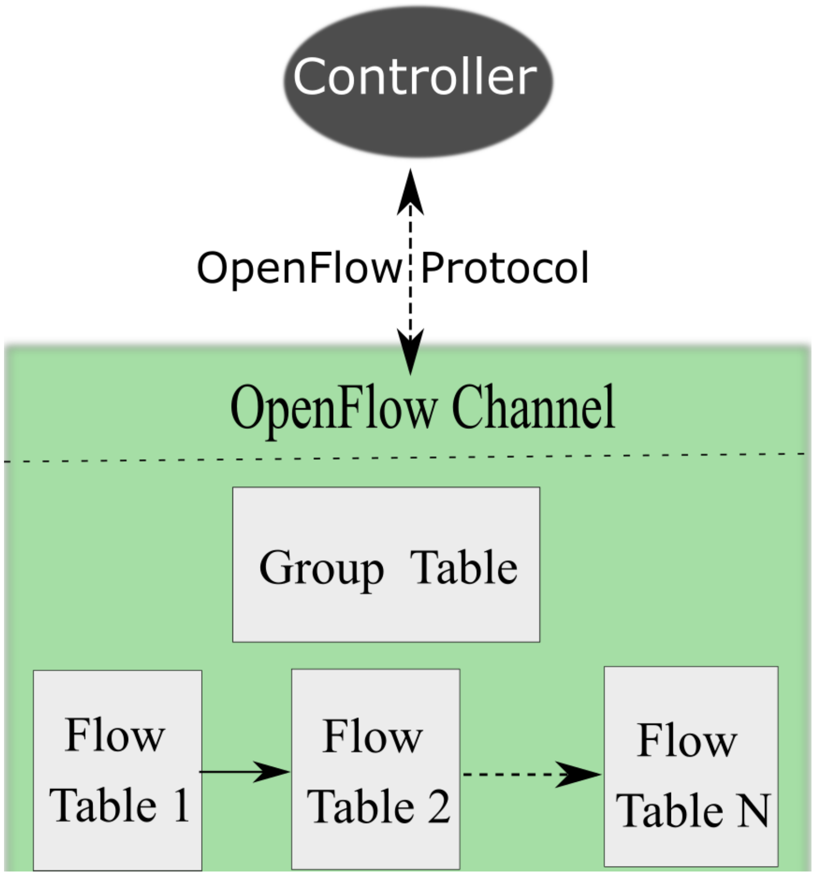


Figure 2.2: OpenFlow-switch

behaviour of an action bucket and its associated parameters depends on the group type.

Two important types of group we make use in this work are: `select` and `fast failover` groups. `select` group is designed for load balancing. As depicted in Fig. 2.3, in a `select` group each bucket has an weight as a special parameter that facilitates the load balancing. Each packet that enters the group is sent to a single bucket. The bucket selection algorithm is external to OpenFlow, and depends on the switch's implementation. Nonetheless, weighted round robin algorithm is the most simplest option for packet distribution to buckets.

Fig. 2.4 depicts a `fast failover` group (FFG) that is designed to detect and respond to port failures. It enables the switch to change forwarding without requiring to contact to the controller. In a FFG, each bucket has a watch port and/or watch group as a special parameter that monitors the liveness of a port or group being watched. The first live bucket, i.e., the bucket whose specified port/group is live, will

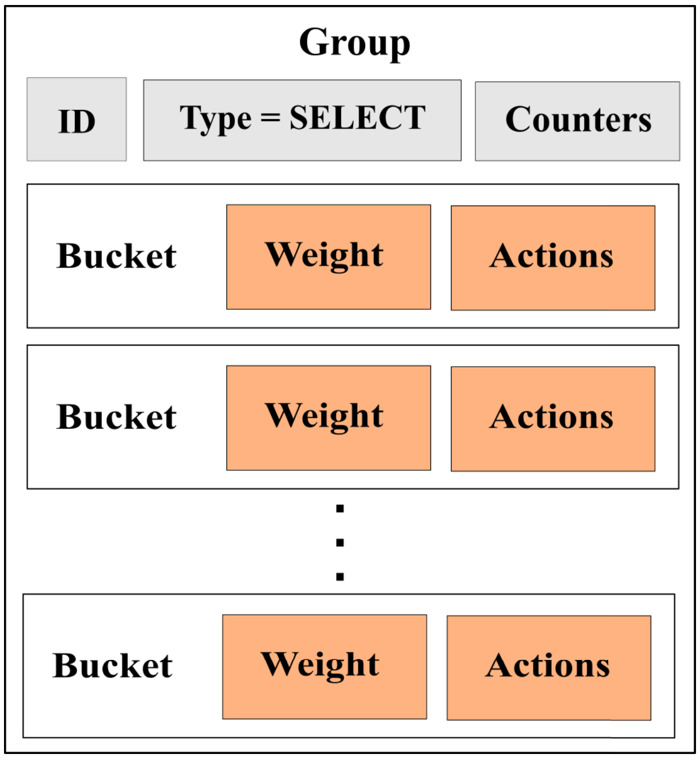


Figure 2.3: Select-group table

be executed.

Figure 2.5 shows the OpenFlow processing pipeline in one snapshot. It starts with the first flow table and may continue to additional flow tables. An incoming packet will be matched against flow entries in the first flow table in the priority order (❶). If no match is found in a flow table, the packet will be forwarded to the controller over the OpenFlow channel (❷) (assuming this instruction is set to table-miss flow entry). Then, controller handles the packet and installs a new flow rule at the switch (❸). If a matching entry is found (❹), the actions associated with it will be executed (❺).

2.1.1.2 Ternary Content Addressable Memory

While SDN allows fine-grained routing policies at the granularity of flows, it can place a huge burden on switch memory. The forwarding rules, i.e., flow and group entries, in an SDN switch are commonly stored in Ternary Content Addressable Memory (TCAM), a specialized type of high-speed memory that searches its entire contents in a single clock cycle [13]. While flexible and efficient in terms of matching capabilities, TCAMs are well-known to be power-hungry and to have limited capacity [14–16]. The

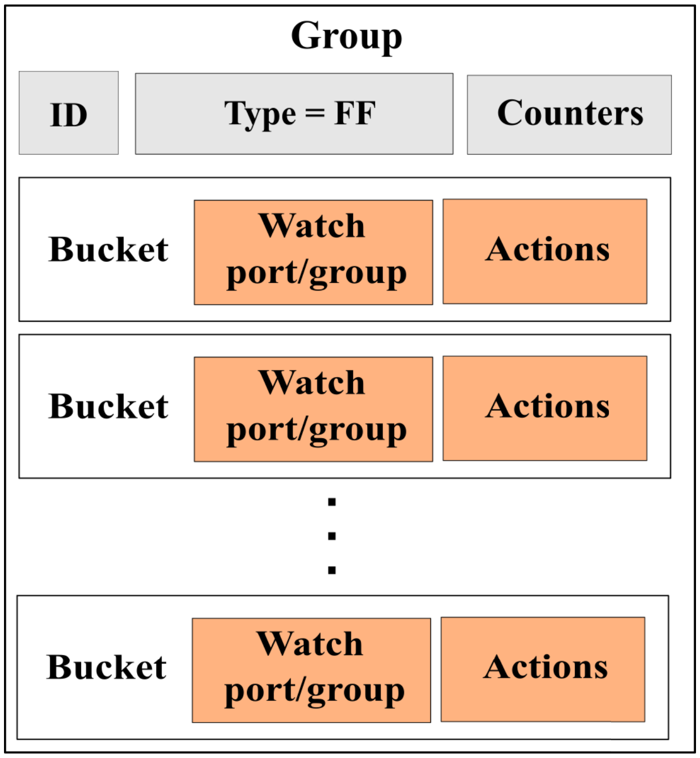


Figure 2.4: FF group table

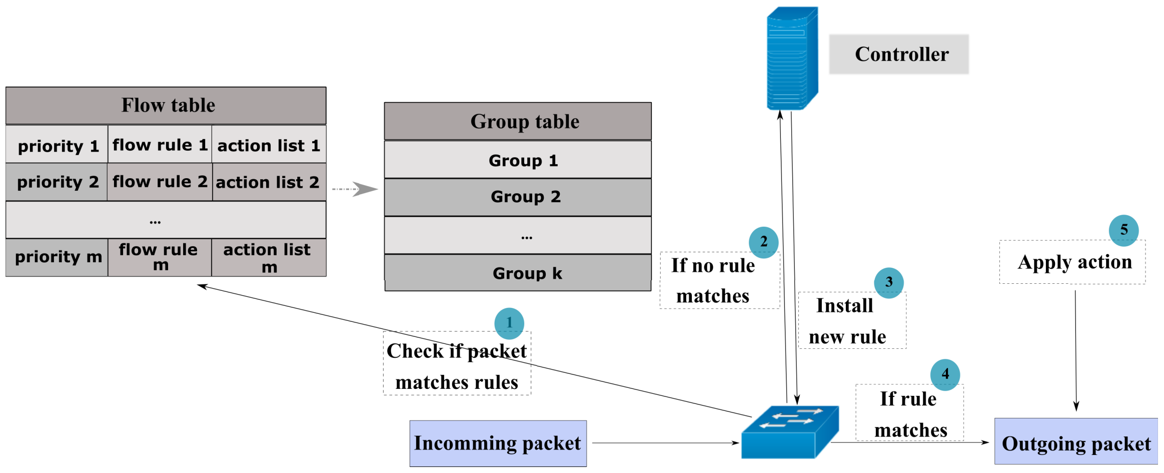


Figure 2.5: SDN switch processing pipeline

capacity of a TCAM chip memory is far less than that of Binary Content Addressable Memory (BCAM). For example, the Broadcom Trident2 SDN switch supports 16K OpenFlow rules, which is not enough in an inter-data center network [5]. In terms of power consumption, TCAM consumes 30 times as much energy as SRAM with an equal number of entries [17].

2.1.1.3 Data Plane Reliability In SDN

In SDN, a failure may occur in the application plane, control plane, or data plane [22]. In this thesis, our focus is on the data plane failures, specifically communication link ones. These type of failures are rather prevalent in WANs, which is unacceptable to meet the strict requirements of the latency-sensitive applications [23–25]. Moreover, 80% of failures are unplanned, while only 20% of failures happen because of scheduled maintenance actions [26]. Also, according to an experiment on Google's data centers and WANs, 80% of the network component failures last from 10 to 100 minutes, which leads to intensive packet loss [27].

Naturally, one may target providing a sufficient level of resiliency to any number of concurrent link failures. However, this approach faces serious scalability and efficiency issues. For example, providing resiliency to up to F link failures in a network with $|E|$ links requires considering $\sum_i^F \binom{|E|}{|i|}$ scenarios. This results in prohibitive computation and configuration costs even for a small number of failures.

Luckily, multiple link failures are less likely to happen than single ones. Driven from a study on Microsoft's data center WAN [7], Table 2.1 shows the probability of having a different number of link failures at 2, 5, and 10 minutes time intervals. As it can be seen, the probability of having more than one link failure within a 5 minute cycle (we will discuss the importance of considering a 5 minute cycle) is around 1%.

Table 2.1: Link failure frequency in Microsoft data center WAN

	Time interval		
Number of link failures	2 min	5 min	10 min
1	10.6%	21.5%	31.2%
2	0.14%	1.1%	4.2%
3	0.14%	0.7%	1.4%

The probability of having one link failure within a 5 minute cycle, on the other hand, is more than 20%, which is not negligible. Only a single link failure can severely impact the performance as the link utilization can approach 100% in an SD-WANs. Assume a 10 Gbps link fails, and its whole traffic load is re-directed to another 20 Gbps link. Even a switch with a 100 MB buffer can withstand for 80 ms, after which a burst of packet drops is inevitable that severely degrades application performance [28].

In general, there are two major failure recovery schemes; namely, *restoration or reactive* and *protection or proactive* in SDN. The controller reactively responds to a link failure in the restoration scheme upon receiving a failure notification from the data plane elements. The controller calculates an alternative route and installs the corresponding flow in the switches along that new route. The communication between the affected switches and the controller, the new route computation, and the device configuration introduces overhead on the controller and delay to recover from a failure [29].

The OpenFlow protocol introduces a local fast failover scheme, called Fast Failover Group (FFG), to eliminate such overhead and delay in the restoration scheme [30]. It enables the quick and local reaction to failures without the need to resort on the controller. In the FFG mechanism, multiple action buckets for the same flow are installed and applied according to the status of links (active or failed). However, FFG can only be used to define a local detour mechanism when alternative routes are available from the node that detects the failure. If a switch does not have an alternative route towards a destination, it can deploy Crankback approach [31]. In Crankback, a switch maintains the packet states and forwards the packet towards the source until the affected packet finds an alternative route.

The protection scheme brings further challenges despite reducing the recovery time. The controller needs to install the available backup routes at the switches that can exhaust switches' TCAM (we already discussed TCAM limitations in section 2.1.1.2). Another crucial challenge faced by a protection scheme is striking a good balance between link utilization and backup path length. These two objectives are inherently at odds; providing short backup paths requires high link utilization over the links along the shortest paths. However, this leads to an unbalanced bandwidth utilization of overall existing links on the network. Fig. 2.6 shows the inherent trade-off between link utilization and backup path length. In this example, S_1, S_2, ..., S_n are sending traffic to S_4. If the red-dashed link fails, it is not possible to optimize the backup path length and link utilization simultaneously. Sending packets over the shortest route can over-utilize the link capacity, whereas distributing packets over longer routes can avoid link congestion.

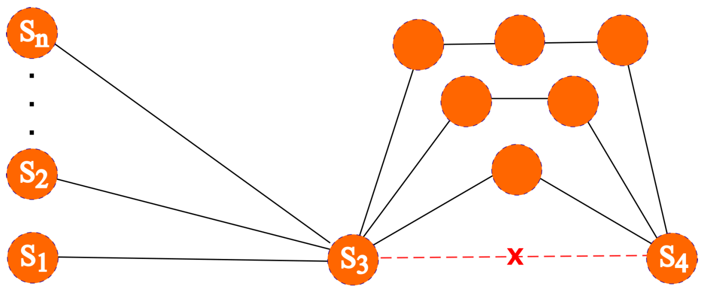


Figure 2.6: The trade-off between the backup path length and the link capacity usage.

2.1.2 Traffic Engineering

To deliver high-quality service, communication networks must distribute traffic efficiently, even in the presence of unexpected traffic spikes and network failures. If a traffic communication network cannot withstand unexpected traffic shifts, it may cause wastage of network resources (e.g., link capacity). For example, a particular link may be unnecessarily overloaded while having underutilized links in other parts of the network. This can leads to a long delay, massive packet loss, and reduced network throughput. These degrade network reliability and availability and may violate increasingly stringent service level agreements (SLAs). Typically, a Traffic Engineering (TE) system improves the network's performance by tuning the routing-protocol parameters. TE decisions are made at fixed TE cycles, practically every 5 minutes [5]. There are two essential phases in the design of a TE system:

- **Path selection:** determines the forwarding routes for carrying traffic. This phase computes a set of forwarding routes between each source-destination pair. Commonly this phase is not executed at each TE cycle unless there is a topology change. This is because updating end-to-end forwarding routes is computationally expensive. In fact, in a network, specifically, a WAN, reconfiguring multiple switches is time-consuming. There are several options for the path selection, such as k-shortest paths and link-disjoint paths (i.e., routes with no common links) [32].

  k-shortest paths work well in simple settings, but one should be careful when using it in topologies with shortcut links because they may become bottlenecks leading to excessive congestion. Link-disjoint paths are also good from the standpoint of throughput. The higher the number of available paths for a flow after a traffic spike or a link failure, the greater the network throughput. However, link-disjoint paths suffer from the same issue as the k-shortest ones: paths between different source-destination pairs can still compete for bandwidth on bottleneck links [33].

- **Rate adaption:** determines how to divide the traffic among selected routes. In the second phase, which is executed at each TE cycle, the system captures the *traffic matrix* and computes the splitting weights that describes how traffic flows should be distributed among the routes. A traffic matrix is a two-dimensional matrix whose ij-th element t_{ij} indicates the volume of traffic sourcing from node i and exiting at node j [34]. As an example of rate adaption phase, the system might adjust weights on some routes as a respond to demand's changes, or set the weight of some routes to zero when there is a network failure [35].

Open Shortest Path First (OSPF) [36] and Multi-Protocol Label Switching (MPLS) [37] are two commonly used routing protocols. OSPF uses the shortest path first (SPF) algorithm to determine routes added to the routing table. Each OSPF router contains a *link state database*, which is the map of the network. This database is synchronized by all OSPF routers, and the information contained in the link-state database is used to compute routing table entries. Each OSPF router forms an adjacency with its neighboring routers. Any time a change occurs in the in the network, information about the change is flooded to the entire network.

OSPF selects the best routes by finding the lowest-cost paths to a destination. All router interfaces (links) are given a cost. The cost of a route is equal to the sum of all the costs configured on all the outbound links between the router and the destination network, plus the cost configured on the interface that OSPF received the Link State Advertisement on [38]. OSPF requires each router to independently determine a packet's next hop by inspecting the packet's destination IP address before consulting its own routing table. However, this approach generally makes the overhead in the packet prohibitively expensive [39].

Instead of hop-by-hop routing, MPLS forwards packets to predetermined routes. In an MPLS network, the first router to receive a packet determines the packet's entire route upfront, the identity of which is quickly conveyed to subsequent routers using a label in the packet header. Thus, MPLS does not entail additional packet header overhead [40].

There have been significant advancements in OSPF- and MPLS-based traffic engineering systems in recent years. They have been optimized to make more efficient use of network resources. However, because of their offline nature, they still have limitations. Instead of balancing the real-time traffic load, they are designed to balance the traffic load given the long term traffic demands averaged over multiple days. But the actual traffic may differ from the long term demands due to external or internal failures, diurnal variations, and security attacks. Neglecting this real-time traffic leads to inefficient load distribution [33].

SDN has facilitated designing centralized TE systems, particularly in wide area networks (WAN), a crucial and expensive infrastructure for large service providers such as Google, Microsoft, and Facebook [4–6]. To better use this invaluable resource, those service providers are investing heavily in centralized TE systems by deploying SDN in their WAN. B4, Google's private software-defined WAN (SD-WAN), is an example of such a deployment. It connects 33 Google's data centers across the globe as of January 2018 [41].

2.2 Related Work

In this work, we propose a software-defined failure recovery system for SD-WANs. Our system takes into account both link and switch memory constraints for proactively allocating backup routes. We formulate the failure recovery problem as a multi-objective MILP optimization problem that selects routes (either primary or backup) and assigns rates to them for all possible single link failures, which is the most common failure scenario in WANs. In this section, we review the literature and present existing works related to this work and point out the gaps in existing solutions.

2.2.1 Reactive and Proactive Failure Recovery Approaches

There has been a large body of work on developing failure reactive and proactive failure recovery approaches in SDN. Not surprisingly, reactive failure recovery approaches are mainly focused on reducing recovery time. Astaneh *et al.* [42] argue that the "operation cost" of adding/removing flow entries to/from the flow tables has an impact on the recovery time, especially in catastrophic scenarios. They study the trade-off between the operation cost and the route stretch and design algorithm to minimize the required operations for end-to-end routes in order to mask a failure.

Ku´ zniar *et al.* [43] propose AFRO, a reactive failure recovery mechanism that generates a new controller instance after a failure occurs. AFRO operation has two phases: record and recovery. During the recording phase, AFRO records all `PacketIn` messages sent to controllers and currently installed rules on switches. Once a failure occurs, AFRO switches to a recovery mode. A copy of the network, excluding the failed elements, is emulated, and the network events are replayed, achieving a valid state that can be used by the actual network. Then, reconfiguration shifts the actual network from the current state to the new one by making modifications to both the controller's internal state and forwarding rules in the switches [22].

To ensure fast failover, Liu *et al.* [44] introduce the notion of Data-Driven Connectivity (DDC). DDC realizes the basic connectivity recovery as a data-plane service, while leaving the optimal path computation task to be handled by the controller. In their model, all possible paths to a destination are modeled as a directed acyclic graph (DAG), where the destination is a node with no outgoing links (i.e., a sink node). Upon failure communication, in parallel with optimal path computation by the controller, the disconnected node implements link-reversal algorithms [45] to find a path toward the destination [22]. Borokhovich *et al.* [46] use graph theory and model the link failure as a graph search problem. They use Modulo algorithm, Depth-First Search (DSF), and Breadth-First Search (BFS) algorithms to ensure connectivity when a failure occurs.

Sharma *et al.* [47] propose a failure recovery framework in OpenFlow that supports different levels of quality of service. Traffic is categorized into the business and best-effort traffic. The framework first checks the type of service field in packet header. If it is enabled, it forwards that traffic using low-delay queues, otherwise, queues with higher delay are chosen [48]. Phemius and Bouet [49] propose a resilient traffic engineering service in WANs. In their method two level of traffic classes are considered: critical and noncritical traffic. The bandwidth utilization is monitored at each link and when a link failure occurs, the traffic engineering service calculates a new path. The alternative paths for the critical flows can be redirected through paths already used by lower priority flows [22].

Paris *et al.* [50] propose a fast failure recovery system that optimizes network usage through network reconfiguration. Two modules are implemented in the controller: a Fast Recovery Setup (FRS) module that is responsible to quickly accept new demands and react to failures, and a Network Garbage Collector (GC) module that enhances the sub-optimal network configurations obtained. They first formulate network reconfiguration as an extension of the NP-complete min-cost Multi-Commodity Flow problem. The GC then applies an iterative algorithm to solve that problem [22].

Nagano and Shinomiya [51] apply the concept of fundamental tie-sets, i.e., minimal cycles in a graph, to failure recovery. Initially, their method produces fundamental tie-sets, and establishes backup paths from them. When the controller receives a notification of a link failure it locates the link failure. Then it applies a tie-set including the a link that will restore connectivity, and installs corresponding rules in the switches [22].

Kim and Gil [52] propose a fault tolerant architecture for hybrid environment consisting of an SDN controller and a Software Defined Network Operations Center (SD-NOC). These two components are orchestrated to cooperate and complement each other. The SDN controller provides an interface to configure network infrastructure and the SD-NOC synchronizes new status information with the SDN controller. When SD-NOC detects a link failure, it relays on the SDN controller to reach SDN devices to recover from the failure using an alternative path [22].

Araújo et al. [53] propose INFLEX, an SDN-based architecture for cross-layer network resilience. In INFLEX architecture, an SDN-enabled routing layer manifests multiple routing planes to the transport layer. This enables shifting from one routing plane to another upon an end-to-end failure detection. The allocation of forwarding planes to flows, is handled using an specialized controller, which resides locally. Despite all above efforts, deploying SDN capabilities could reduce recovery time to the scale of 50ms [54], which can still be an issue in modern data centers [55].

The pioneer proactive works in SDN deploy FFG. Ghannami *et al.* [56] preinstall a set of rooted trees as the primary and backup routes and quickly activate them in case of failure. Cheng *et al.* [57] apply flow aggregation using VLANs to prevent congestion while using FFG.

The FFG is efficient as long as there is an available alternative route from the node that detects the failure. Otherwise the switch that does not have an alternative route towards a destination should deploy Crankback approach [31]. In Crankback, a switch maintains the packet states and forwards the packet towards the source until the affected packet finds an alternative route [58]. Capone *et al.* [59] use Openstate [60] and formulate a constrained optimization problem to compute the backup route, where the formulation considers congestion, reverse route length from the failure detection node to the detour node, and the number of links allocated to the backup routes. Zhu *et al.* [61] propose a backup approach that aggregates non-conflicted backup routes to reduce the number of backup rules.

PIQoS [62] realized a data-driven model for the failure detection and leveraged the FFG for the failure recovery. SD-FAST [63] proposed a packet rerouting architecture that maintains the packets states. SD-FAST invokes the failure recovery and enforces the traffic to traverse through a series of OpenFlow tables only if the traffic has faced a failure.

In [11], Foerster *et al.* propose an algorithm, called *CASA*, a proactive failure recovery mechanism that aims to guarantee resilient networks under multiple link failures, where they consider route stretch and load. CASA defines multiple link-disjoint arborescences (a directed graph in which there is exactly one directed route from any node to a specific node, called root) between each source-destination pair, where packets change their arborescence after facing failure. CASA relies on the network connectivity after multiple link failures, the assumption that we are not considering in this work. Moreover, TCAM usage, an essential design criterion in the backup route computation, is not considered in the above works when constructing the backup routes.

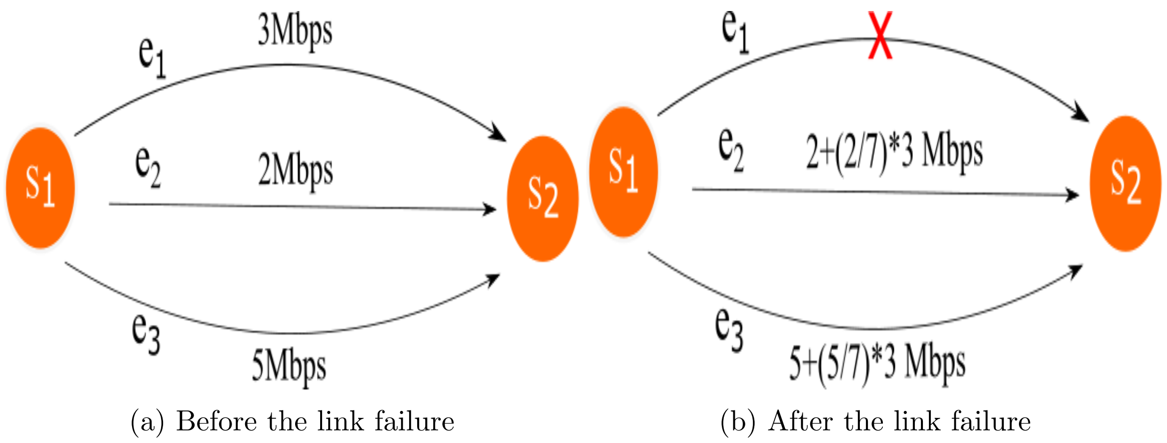
Besides the reactive and proactive approaches, there exit some hybrid works as well. For example, Haque *et al.* [64] propose Revive, a hybrid failure recovery method that takes advantage of both the reactive and proactive capabilities by pre-installing backup rules only on a subset of switches that carry traffic for the ongoing communications. Revive constructs edge-disjoint routing topology using the spanning structure-based topology construction algorithm proposed in [65]. Also, Tilmans *et al.* [66] proposed a hybrid SDN architecture to reduce the control-plane overhead.

Once a failure occurs, the network controller configures routing policies and uses Interior Gateway Protocol to recover the network connectivity.

2.2.2 Failure Recovery Using Traffic Engineering (TE)

One line of work has explored the space of failure recovery as a TE application for SDN. This work falls within this category of studies. We start by the work of Wang *et al.* [67] who propose R3, a proactive failure recovery approach that is resilient to multiple link failures. To cover all link failure scenarios, R3 computes virtual demand for every link in the network. The convex combination of all such virtual demands covers the entire space of rerouted traffic under all possible multiple link failures.

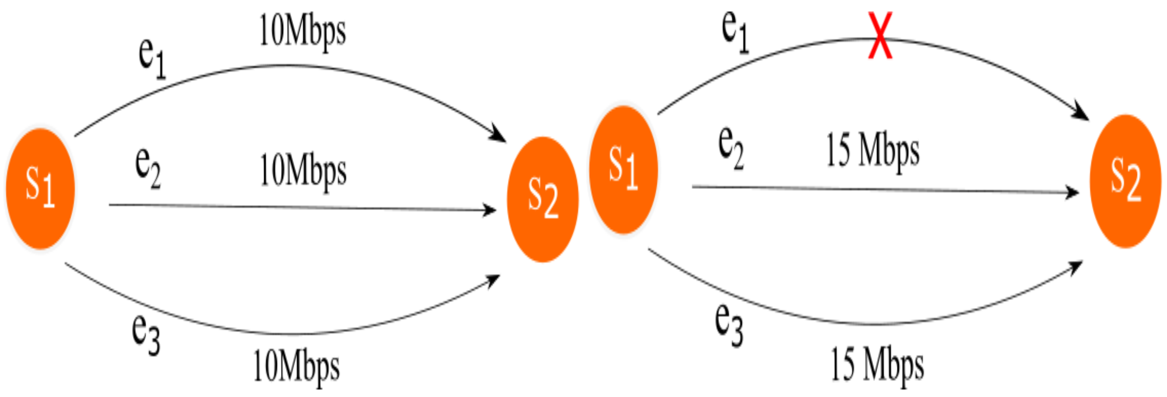
We illustrate the operation of R3 after a link failure occurs in Fig. 2.7, where node S_1 sends traffic S_2 at 3 Mbps, 2 Mbps and 5 Mbps on links e_1, e_2 and e_3 (2.7a), respectively. When link e_1 fails, R3 splits the traffic rate on failed link e_1 between e_2 and e_3 (2.7b). e_3 that had a higher contribution in traffic before the failure, will also carry more traffic of e_1 after the failure.



(a) Before the link failure (b) After the link failure

Figure 2.7: R3's operation: after the link failure the traffic rate on link e_1 is distributed between e_2 and e_3 proportional to their contribution on the traffic rate before the failure happens.

Liu et al. [10] leave off spare capacity to support up to k arbitrary failures. For the simple case of $k = 1$, the operation of FFC in an example is shown in Fig. 2.8. This time, S_1 wants to send 45Mbps traffic to S_2 over links e_1, e_2 and e_3 whose capacities are 15Mbps. However, FFC does not send the whole traffic. Instead, it sends 10Mb to each link (2.8a), leaving 5Mbps spare capacity on each link. Therefore, after the failure, e_2 and e_3 can absorb the traffic that was being carried by failed link e_1 (2.8b). Obviously, FFC cannot satisfy the flow demand requirement of 45Mb from S_1 to S_2 in the first place and leads to 15Mb capacity loss.



(a) Before the link failure (b) Afte the link failure

Figure 2.8: FFC's operation: before the link failure, FFC leaves off 10Mbps capacity on each link, so that the traffic rate on link e_1 can be distributed between e_2 and e_3 after it fails.

Bogle *et al.* [12] propose TEAVAR, a TE framework that optimizes bandwidth assignment subject to meeting a desired availability threshold. Using empirical data, TEAVAR generates a probabilistic model of network failures and then computes bandwidth allocations that guarantees a desired availability threshold. One TE solution of TEAVAR is shown in figure 2.9, which is the same network as in figure 2.8a with added information about some sample link failure probabilities. In such a scenario, a possible traffic allocation of TEAVAR can support 45Mbps traffic 97% of the time.

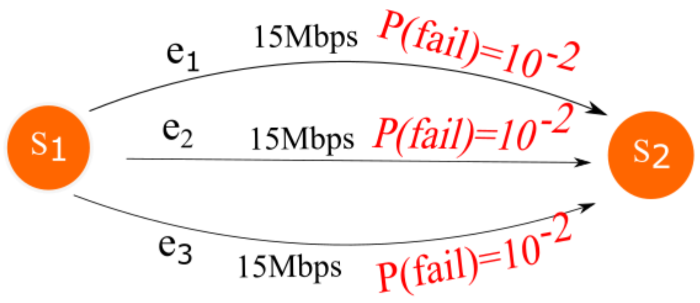


Figure 2.9: A network with link failure probabilities. In such a scenario, a potential traffic allocation of TEAVAR can support 45Mbps traffic which is available 97% of the time.

In [7], Zheng *et al.* propose Sentinel, a failure recovery approach that uses link-disjoint backup tunnels. Sentinel computes backup tunnels and splitting weights, and activates them when a failure occurs. As opposes to the previous TE-based failure recovery solutions, Sentinel does not forwards traffic through the failed tunnels when completing the failover at the ingress switch. Instead, it uses backup tunnels that start from the failing switch and end at the egress switches to re-direct the affected traffic. The operation of Sentinel is shown in figure 2.10. In this example, ingress switch S_1 forwards traffic through tunnels T_1 and T_2 toward egress switch S_6 (figure 2.10a). When S_4 detects the failure, it uses tunnel T_3 to forward the upcoming packets (figure 2.10b) until ingress switch S_1 completes activating backup tunnel T_2 and new splitting weights (figure 2.10c).

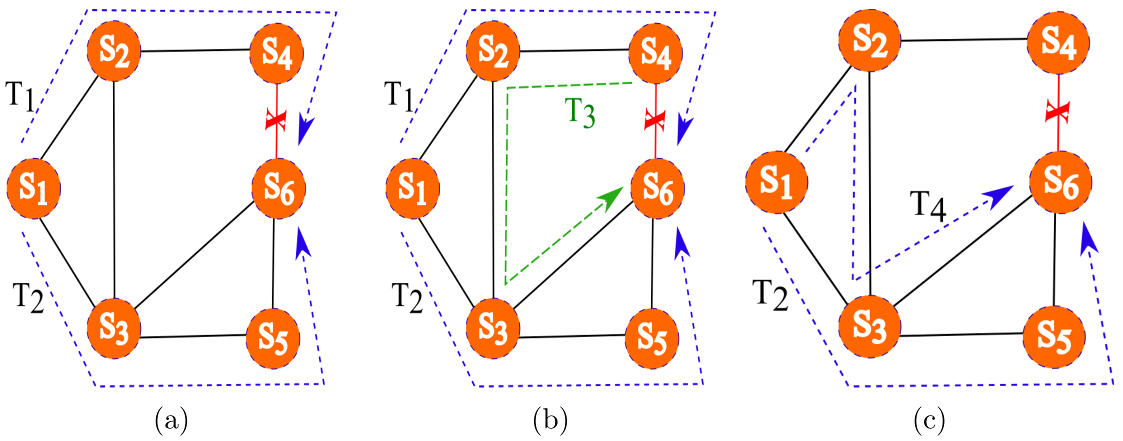


Figure 2.10: Sentinel's operation: a) two link-disjoint tunnels for the flow from S_1 to S_6, b) while completing the failover at the ingress switch, Sentinel uses backup tunnels that start from the failing switch and end at the egress switches to re-direct the affected traffic, c) new established link-disjoint tunnels

The set of paths selected to carry traffic in a TE system, is a key factor that impacts the quality of that system. In this regard, Kumar *et al.* [68] propose Smore, a TE system that leverages the concept of oblivious routing. An oblivious routing algorithm computes a probability distribution on short paths and forwards traffic based on that distribution. That means the paths in oblivious routing are computed without knowledge of the demands (i.e., it is oblivious to the demands). Particularly, an oblivious routing identifies *routing trees* that have a unique path between each-source-destination pair. Then, it defines a *randomized routing tree* (RRT), which is a probability distribution over routing trees. Then a path is computed by first sampling a routing tree T for RRT, then selecting the unique path from the sampled routing tree. In addition to oblivious routing, Smore leverages centralized traffic adaptation to minimizing the maximum link utilization.

In 2013, Jain *et al.* [4] presented their years experience of designing and deploying Google's SD-WAN, B4. In B4, traffic flows are divided into flow groups, each of which represented by a tuple (including source site, destination site, QoS). B4 applies a hash-based ECMP algorithm for load-balanced routing, and remarkably shows 99% availability and near to 100% link utilization. Five years later, Hong *et al.* [41] presented their experience of evolving B4 into a hierarchical topology that improves the availability by two orders of magnitude, from 99% to 99.99%, in the presence of failures.

We summarize the related works reviewed above in table 2.2. We compare them in terms of their optimization model and the metrics they have taken into account in their solution. "NA" in the "model" column implies that the corresponding work has not modeled the failure recovery as a optimization problem.

In the next chapter, we present our failure recovery system, which falls within the TE-based approach category. We model the failure recovery problem as a MILP, where we not only target simultaneously minimizing the backup route length and the link utilization but also we consider the TCAM constraint. None of the prior works considers all these objectives and constraints in their solution. We furthermore take into account the traffic with different QoS requirements, an important point being missed in most of the prior works.

Table 2.2: Failure recovery work comparison

Authors	**method**	**model**	**Backup route Length**	**Link utilization**	**TCAM**
Astaneh *et al.* [42]	reactive	NA	✓	✗	✗
Ku´ zniar *et al.* [43]	reactive	NA	✗	✗	✗
DDC [44]	reactive	NA	✓	✗	✗
Borokhovich [46]	reactive	NA	✓	✗	✗
Sharma *et al.* [47]	reactive	NA	✗	✗	✗
Phemius *et al.* [49]	reactive	NA	✗	✗	✗
Paris *et al.* [50]	reactive	LP	✗	✗	✗
Nagano *et al.* [51]	reactive	NA	✗	✗	✗
Kim *et al.* [52]	reactive	NA	✗	✗	✗
Araújo et al. [53]	reactive	NA	✗	✗	✗
Ghannami *et al.* [56]	proactive	NA	✓	✓	✗
Cheng *et al.* [57]	proactive	ILP	✗	✓	✓
Capone *et al.* [59]	proactive	MILP	✓	✓	✗
Lekhala *et al.* [62]	proactive	NA	✗	✗	✗
SD-fast [63]	proactive	NA	✗	✗	✗
CASA [11]	proactive	NA	✓	✓	✗
Revive [64]	hybrid	NA	✓	✗	✓
Tilmans *et al.* [66]	hybrid	NA	✗	✗	✗
R3 [67]	TE-based	LP	✗	✓	✗
FFC [10]	TE-based	LP	✗	✓	✗
Teavar [12]	TE-based	LP	✗	✓	✗
Sentinel [7]	TE-based	LP	✗	✓	✓
Smore [68]	TE-based	LP	✓	✓	✗
B4 [4]	TE-based	LP	✗	✓	✗
Hong *et al.* [41]	TE-based	LP	✗	✓	✓
SafeGuard [8, 9]	TE-based	MILP	✓	✓	✓

Chapter 3

SafeGuard: Congestion and Memory-aware Failure Recovery in SD-WAN

This chapter is based on [8], where we present our software-defined failure recovery system for SD-WANs called SafeGuard. We formulate the failure recovery problem as a multi-objective MILP optimization problem. This model selects alive primary or backup paths and assigns rates to them for all possible single link failures. We solve this optimization problem and elaborate on its complexity. This leads us to design a heuristic for solving our optimization problem. Then, we present SageGuard's architecture and workflow, and show how it is implemented in an SDN environment. After that, we briefly introduce Mininet, the network emulator that we use throughout this work. Lastly, we evaluate our proposed solution and state its performance.

3.1 Problem formulation

3.1.1 Network Design

Let $G = (V, E)$ be a digraph, where $V = \{1, 2, .., n\}$ is the set of network switches (or nodes) and E is the set of links among the switches. A node can be a source or a destination of flows that are routed through the network. Each link $(i, j) \in E$ has a capacity c_{ij}. Each flow f in this network has a required bandwidth, b_f. The demands represent the *aggregate* traffic from an ingress to an egress switch which are divided among a set of primary routes P_f. We denote the traffic rate of flow f on each route $p \in P_f$ as t_{fp}[1]. When a link e fails, all flows traversing that link will be affected. For each affected flow f, there exist a set Q_f of backup paths available. Each path $p \in Q_f$ consists of two parts: a portion of the impacted primary route (i.e., from ingress switch to failing switch (i.e. the source switch of a failing link), and an alternative path from the failing switch to the egress switch of the respective

[1]for the sake of simplicity, any route in our model, either primary or backup, is referred to as p.

flow. Note that a link failure could impact one or multiple primary routes of a flow, as primary routes are not necessarily link-disjoint. We denote the set of non-affected primary routes as $P_f^{'}$. In either case, we select both backup and non-affected primary routes to compute a new routing for flow f after a failure. Table 3.1 summarizes these notations.

Table 3.1: Key notations used in the model.

	Notation	Description
Input	$G(V,E)$	Network graph with switches V and Links E;
	P_f	Set of primary routes for flow f;
	Q_f	Set of backup routes for flow f;
	t_{fp}	Traffic rate of flow f on route p;
	b_f	Bandwidth required for flow f;
	c_{ij}	Total capacity of link $(i,j) \in E$;
	Tc_i	TCAM space capacity of node $i \in V$;
Auxiliary variables	e	A failed link;
	F_e	Set of impacted flows by link failure e;
	F	Set of flows not impacted by link failure e;
	$P_f^{'}$	Set of primary routes for flow $f \in F_e$ not impacted by link failure e;
	l_p	Length of route $p \in P_f^{'} \bigcup Q_f$;
	$r_{ij,p}$	1 if link $(i,j) \in p$, 0 otherwise;
	$s_{i,p}$	1 if node $i \in p$, 0 otherwise;
Output	x_{fp}	Allocation of affected flow f on route $p \in P_f^{'} \bigcup Q_f$;
	y_{fp}	Traffic rate of affected flow f assigned to route p;

3.1.2 Variables and Parameters

After the link failure e, we need to assign traffic rates on backup and unaffected primary routes for each affected flow. As such, we define two sets of decision variables: x_{fp}, which is a binary decision variable denoting the routes used for impacted flow f after recovery, as follows:

$$x_{fp} = \begin{cases} 1 & \text{if } route\ p \in P'_f \bigcup Q_f\ is\ used\ to\ forward \\ & traffic\ from\ flow\ f, \\ 0 & otherwise. \end{cases} \tag{3.1}$$

We also define the decision variable $y_{fp} \geq 0$ to denote the traffic rate of impacted flow f assigned to path p after the failure. Lastly, we define binary parameters, $r_{ij,p}$ and $s_{i,p}$, to identify the links and nodes belonging to path p, respectively:

$$r_{ij,p} = \begin{cases} 1 & \text{if } \ (i,j) \in p, \\ 0 & otherwise. \end{cases} \tag{3.2}$$

$$s_{i,p} = \begin{cases} 1 & \text{if } \ i \in p, \\ 0 & otherwise. \end{cases} \tag{3.3}$$

3.1.3 Constraints

Link and node capacity: constraints in (3.4) prevent the link utilization on a link from exceeding the bandwidth capacity of that link. The total utilization in the left hand side of the inequality includes the traffic rates assigned to both affected ($f \in F_e$) and unaffected ($f \in F$) flows, either for primary or backup routes. Constraints in (3.5), on the other hand, prevent switch memory usage from exceeding its capacity. Similar to link capacity constraints, the total switch memory usage includes the demands from primary and backup routes of both affected and unaffected flows.

$$\sum_{f \in F_e} \sum_{p \in P'_f \bigcup Q_f} y_{fp} r_{ij,p} + \sum_{f \in F} \sum_{p \in P_f} r_{ij,p} t_{fp} \leq c_{ij} \ \ \forall (i,j) \in E \tag{3.4}$$

$$\sum_{f \in F_e} \sum_{p \in P'_f \bigcup Q_f} x_{fp} s_{i,p} + \sum_{f \in F} \sum_{p \in P_f} s_{i,p} \leq Tc_i \ \ \forall i \in V \tag{3.5}$$

Traffic rate: constraints in (3.6) correlate variables x_{fp} and y_{fp}. They ensure that traffic rate of flow f on path p is zero when p is not allocated to flow f (i.e., $x_{fp} = 0$). Otherwise, it must be a positive number.

$$x_{fp} \leq y_{fp} \leq x_{fp} b_f, \ \ \forall f \in F_e, p \in P'_f \bigcup Q_f \tag{3.6}$$

Flow satisfaction To ensure flow demands are satisfied, constraints in (3.7) enforces the total traffic rate of flow on its primary and backup routes to be equal to its demand. We only need to take care of the affected flows (i.e., $f \in F_e$) as non-affected ones are satisfied by default.

$$\sum_{p \in P'_f \bigcup Q_f} y_{fp} = b_f \quad \forall f \in F_e \tag{3.7}$$

3.1.4 Objective Function

The objective function (3.8) includes two terms: the first term takes care of the backup route length, and the second one accounts for the overall link utilization in the network. α and β parameters are to favor a specific objective depending on the operator needs. For example, favoring the shorter routes due to application latency constraints or lower link utilization due to network congestion.

$$\min \sum_{f \in F_e} \sum_{p \in P'_f \bigcup Q_f} (\alpha x_{fp} l_p + \beta \sum_{(i,j) \in E} \frac{r_{ijp} y_{fp}}{c_{ij}}) \tag{3.8}$$

3.1.5 Parameter Sensitivity

We incorporated α and β parameters in the objective function. By varying these parameters we can reflect the importance of each term in the objective function. As such, a larger α value implies a higher importance for the backup path length (first term of the objective function), whereas a larger β value implies a higher importance for the link utilization (second term of the objective function). To gain further insight, we perform a parameter sensitivity analysis using CPLEX to empirically determine how α and β impact the objective function and what values of α and β implies the same importance for both objectives.

We use IBM ILOG CPLEX Optimization Studio (commonly refereed to as CPLEX for short), which is a commercial optimization software package developed by IBM [69]. CPLEX Optimizer is the best known and most widely used large-scale solver capable of solving wide range of optimization problems incluing:

- Linear Programming (LP),

- Mixed Integer Linear Programming (MILP),
- Quadratic programming, and
- Quadratically constrained programming.

We consider ATT network topology [70] (with 25 nodes and 112 links), and 100Mbps traffic size. We set the $\frac{\alpha}{\beta}$ ratio to 0.01, 0.1, 1, 10, and 100 and calculate the the optimal value of objective function using CPLEX.

Fig. 3.1 shows how the average optimal backup route length varies with vary in $\frac{\alpha}{\beta}$ ratio. Note that we use the *log* scale of $\frac{\alpha}{\beta}$ ratio to make the figure more informative. As it can be seen, when we give the higher importance to the backup route length by setting the $\frac{\alpha}{\beta}$ to a larger value, the average backup route length decreases. Intuitively, our objective function is optimal with a small value of the backup route length when it has a large coefficient compare to the second term in the objective function (link utilization).

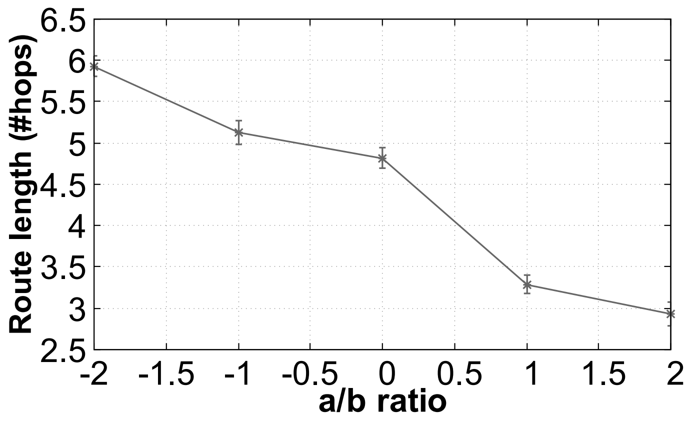


Figure 3.1: Parameter sensitivity (effect on route length)

As expected, we observe an opposite trend in figure 3.2, where the average optimal link utilization increases with the increase in the $\frac{\alpha}{\beta}$ ratio. When the link utilization is given a low importance (large $\frac{\alpha}{\beta}$), its value does not have a significant impact on the optimal objective function.

Although, solving the model gives us the optimal solution of the problem, it does not in large networks:

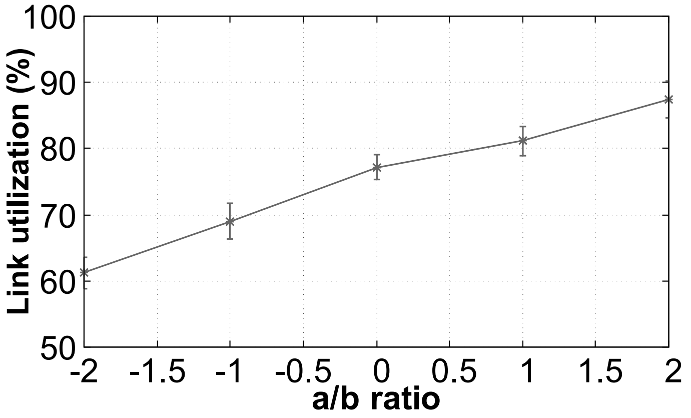


Figure 3.2: Parameter sensitivity (effect on link utilization)

Definition 1. *A mixed integer linear program (MILP) is of the form:*

$$\begin{aligned} &\min c^T x \\ &\quad Ax = b \\ &\quad x \geq 0, \quad x_i \in \mathbb{Z} \quad i \in I \end{aligned}$$

where c and b are vectors and A is a matrix, where all entries are integers. Also, only some of the variables, are constrained to be integers, while other variables are allowed to be non-integers [71].

Our model is known to be NP-hard [72], thus it takes a long time to be solved in practice. For example, it took us more than an hour to solve our model for Cogent[2] topology on CPLEX. As traffic engineering intervals are usually below 5 minutes, we develop a heuristic to solve the SD-WAN failure recovery problem in a reasonable amount of time. We describe our heuristic in the next section.

3.2 Heuristic Design

Now, we present our proposed heuristic, and use a toy example to show how it works in action. Algorithm 2 illustrates the algorithm's operation. The inputs of the algorithm are: a) the network topology after the link failure e, b) the set of impacted flows F_e, and c) a set of candidate paths P, which includes both backup paths and non-impacted primary ones for all impacted flows. We want to reallocate impacted flows

[2]https:/cogentco.com/en/network/network-map

through the candidate paths to satisfy their demands d_f while taking into account the link and switch memory budget.

Algorithm 1: Heuristic for rerouting traffic

```
Input  : Network topology $G' = (V, E\backslash e)$; impacted flows $F_e$; set of available
         paths $P = P'_f \bigcup Q_f, \forall f \in F_e$
Output: Traffic rates assigned to $y_{pf}$

1  Sort $F_e$ in descending order according to flow demands
2  Sort $P$ in ascending order according to path length
3  for $f \in F_e$ do
4      for $p \in P'_f \bigcup Q_f$ do
5          if $\forall s \in p, u_s < Tc_s - 1$ then
6              $y_{pf} = \min\left(\min_{(i,j)\in p} a_{ij}, d_f\right)$
7          else
8              if $d_f \leq \min_{(i,j)\in p} a_{ij}$ then
9                  $y_{pf} = d_f$
10             end
11         end
12         if $y_{pf} > 0$ then
13             Update $u_s, a_{ij}\ \forall s, (i,j) \in p$
14             $d_f \leftarrow d_f - y_{pf}$
15         end
16     end
17 end
```

First, we sort the impacted flows in descending ascending order in terms of their demands (line 3). We also sort the available paths in terms of their path length (line2). This way, we allocate the flow demands, from biggest to smallest, to the shortest paths as an attempt to optimize the link utilization. For a given flow, We iterate over its candidate paths, and inspect to see whether all switches along that path have enough *memory* capacity (u_s) to accommodate new forwarding rules (lines 4-7). If enough memory is available, the traffic rate allocation is equal to the minimum between the flow demand and the remaining *bandwidth* capacity (a_{ij}) of the bottleneck link (line 8).

We give the higher priority to the the bigger demands until we hit a given threshold in terms of switch memory utilization on a path ($Tc_s - 1$ for the bottleneck switch in our case). After that, we apply a “best-fit” strategy, i.e., we allocate the demands that can be fully satisfied. This way we guarantee that smaller flows have access to shorter paths too (lines 10-12). Finally, we update the remaining capacities and demand to be allocated whenever we select a path for a flow (lines 14-17). This approach can be adverse for some shorter flows; however, it is consistent with our multi-objective optimization model, where we try to make a good balance between *path length and link utilization.* Particularly, exclusive allocation of smaller demands first would lead to bigger demands being assigned to longer paths which in turn leads to higher link utilization.

The time complexity of Algorithm 2 is correlated with the number of flows impacted by a link failure and the corresponding set of available paths. The time complexity of the algorithm is $O(MN(|V| + |V|log|V|))$, where $|F_e| = M$ and $|P| = N$. Note the we consider the worst case complexity which is given by the operations at lines 4-8.

We illustrate the operation of Algorithm 2 in Fig. 3.3 [8]. There are two flows $\{f_1, f_2\}$: f_1 from switch S_1 to S_5 has a demand of 25Mbps and flow f_2 from switch S_2 to S_5 has a demand of 20Mbps. Their primary routing tunnels and corresponding rates are shown in Fig. 3.3a. For simplicity, we assume that all switches have enough memory to accommodate these two flow rules. We also assume that the available capacity (after traffic rate allocation of f_1 and f_2) of all links is 15Mbps except link $S_3 - S_5$ whose capacity is 5Mbps. The failure of link $S_4 - S_5$ impacts both $\{f_1, f_2\}$. In the following, we illustrate the operation of the heuristic.

Algorithm 2 allocates flow f_1 (black flow) first as it has the higher demand. This flow has access to two tunnels to send the traffic through after the failure: $S_1 - S_3 - S_5$ and $S_1 - S_4 - S_6 - S_5$. It sends 20Mbps over the former tunnel. The remaining 5Mbps is allocated over the second tunnel. On the other hand, the available tunnels for the second flow f_2 are: $S_2 - S_1 - S_3 - S_5$ and $S_2 - S_4 - S_6 - S_5$. The traffic allocation on the former one (10Mbps) does not change as it has no more capacity.

The remaining 10Mbps can be forwarded over the second tunnel. Note that for both flows, one of the original tunnels remains unchanged except the corresponding rate that depends on traffic allocation.

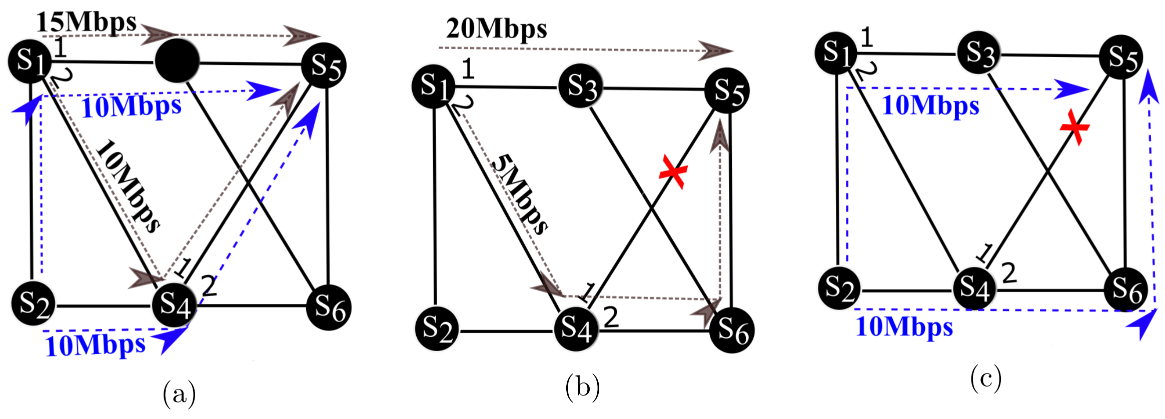


Figure 3.3: a) Traffic rates on the primary tunnels of two flows (blue and black). The available capacity (after allocating two flows) of all links is 20Mbps except link $S_3 - S_5$ whose capacity is 5Mbps; b) the available tunnels for the impacted flow with the highest demand and corresponding traffic rates assigned to them; c) the available tunnels for the second impacted flow after the link failure with the traffic rates assigned to them.

3.3 SafeGuard's Architecture

From a high level perspective, SafeGuard is an SDN failure recovery system that enables fast recovery while taking the network traffic load and scarce switch memory resources into account. SafeGuard runs as a SDN controller application that applies our proposed heuristic to derive resilient routing configurations to network switches at every TE interval. In the following we describe SafeGuard's component:

1) Control plane: Fig. 3.4 depicts SafeGuard's architecture [8]. A traffic matrix, the network topology and a set of forwarding routes (primary routes) are given to it as the input. It then produces a primary allocation that will be used for forwarding traffic under normal conditions (i.e., when there is no failure in the network). After that, SafeGuard deploys our proposed heuristic to produce a new allocation using backup routes for all flows for each possible single link failure. Ultimately, it outputs a resilient routing allocation as a set of forwarding rules to be configured by the SDN switches.

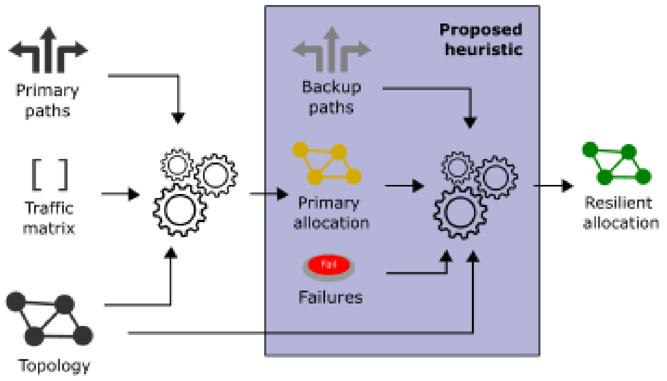


Figure 3.4: The architecture of SafeGUard

2) Data plane: SafeGuard uses the OpenFlow protocol (version 1.3 [21]) to communicate with the network switches. Each switch contains at least one Flow Table and a Group Table. The flow table includes a set of flow rules, each comprising of match fields and corresponding action (or instruction). The group table, on the other hand, consists of group entries (group for short) consisting one or more action buckets. An example is shown in table 3.2. SafeGuard make use of two types of groups: `select` and `fast failover` groups. The former groups performs the traffic rates allocation and the load balancing among multiple paths. This group type executes any one action bucket in the group according to a weight set as a special parameter. The `fast failover` groups overcomes the failure by rerouting traffic upon detecting a link failure (a down port) and executing the first live bucket.

Now, we revisit the example in Fig. 4.1. We consider the first flow (f1: from S_1 to S_5) to show how SafeGuard configures the flow and group tables at switches S_1 and S_3 to recover from the failure. Table 3.2 illustrates the flow and group tables at the ingress switch S_1. There exists one flow entry that points to one of the groups. Before the link failure, the flow entry points to group 1 (G1.1) of type `select` in the group table, which assigns traffic rates to the two primary routes (see Fig. 3.3a). Group entry 1 then points to groups 2 and 3, which are of type `fast failover`. Because both ports of S_1 are up in this example, G 1.2 and G 1.3 forward packets over ports 1 and 2, respectively.

Table 3.3 depicts the flow and group tables at switch S_4, which detects the failure. Before the link failure, it sends packets from S_1 to S_5 through port 1 over path $S_1 - S_4 - S_5$). After the failure detection, S_4 starts sending packets from S_1 to S_5

using port 2 applying the `fast failover` group (path $S_1 - S_4 - S_6 - S_5$) and at the same time it informs the controller about the failure. When SafeGuard receives the failure notification from S_4, it modifies the instruction in the flow table of ingress switch S_1 to point to group 4 (G1.4) to update the allocated rates for each route used by the affected flow.

Table 3.2: Flow and group tables at S_1 for the example in Fig. 4.1

(a) Flow table at S_1

.

Match Field		
SrcAdrr	***DstAdrr***	***Instruction***
S_1	S_5	$G1.1$

(b) Group table at S_1

Group ID	**Group Type**	**Action Buckets**
G 1.1	select	weight:$\frac{3}{5}$, action: G 1.2 weight:$\frac{2}{5}$, action: G 1.3
G 1.2	fast failover	outport: 1 outport: 2
G 1.3	fast failover	outport: 2 outport: 1
G 1.4	select	weight:$\frac{4}{5}$, outport: 1 weight:$\frac{1}{5}$, outport: 2

Table 3.3: Flow and group tables at S_4 for the example in Fig. 4.1

(a) Flow table at S_3

.

Match Field		
SrcAdrr	***DstAdrr***	***Instruction***
S_1	S_5	G 3.1

(b) Group table at S_3

Group ID	**Group Type**	**Action Buckets**
G 3.1	fast failover	outport: 1 outport: 2

3.4 SafeGuard's Workflow

Fig. 3.5 shows the SafeGuard's workflow [8]. The same process repeats every traffic engineering interval. At first, SafeGuard computes all primary paths and splitting weights and installs the corresponding forwarding rules for every flow (step A). Then, it proactively installs backup routes and computes the splitting weights for allocated flows which is resilient to any single link failure (step B). When a failure occurs, the failing switch activates the corresponding backup paths, which connect the failing switch to the egress switches of the corresponding affected flows, and simultaneously sends a message to inform the network about the failure (step C). Lastly, the network controller, which runs the SafeGuard heuristic, adjusts splitting weights for all affected flows at their respective ingress switches (step D).

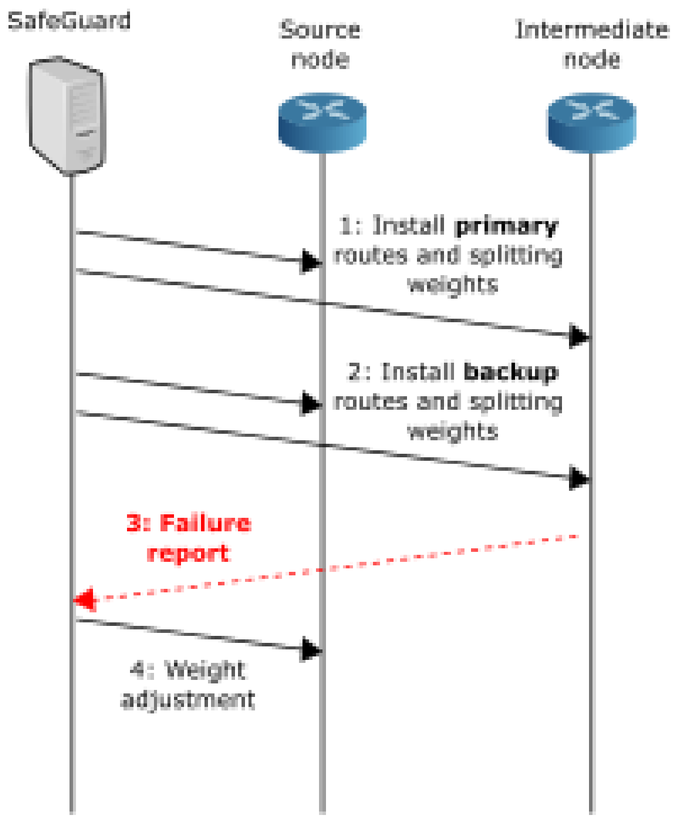

SafeGuard
Source node
Intermediate node
1: Install **primary** routes and splitting weights
2: Install **backup** routes and splitting weights
3: Failure report
4: Weight adjustment

Figure 3.5: The workload of SafeGuard

3.5 Evaluation

In this section, we describe SafeGuard's implementation and discuss the results. We implement SafeGuard as a Python application on top of the Ryu SDN controller (version 4.30). The source code of our implementation is available at [19]. Before jumping to the evaluation discussion, we introduce Mininet [73], the network emulator that we used for the evaluating SafeGuard.

Mininet. It is a network emulator that runs a network of virtual hosts, switches, controllers, and links on a single Linux kernel. Mininet enables SDN development using lightweight virtualization, in which a single system acts like a complete network, running the same kernel, system, and user code. Mininet enables:

- Quick SDN prototyping
- Custom SDN design
- Share and replicate results

Setup. We run our experiments on a machine with 2.66GHz 12 core CPU and 44GB RAM equipped with Mininet (version 2.2.2). We consider two network topologies: Google B4 [4] (12 nodes and 38 links) and ATT [70] (25 nodes and 112 links). We implement each node as a CPqD[3] switch instance [74]. Link capacities are set to 1 Gbps with 1ms delay. We benchmark SafeGuard against Sentinel [7], the state-of-the-art failure recovery approach in SD-WAN. When a failure occurs, Sentinel uses the link-disjoint tunnels to distribute the traffic rate of affected flows. Sentinel tries to minimize the maximum link utilization. However, it considers neither the length of the backup tunnels nor the switch memory usage. We use Iperf[4] tool to generate UDP traffic. We fix the flow rate to 50 Mbps and vary the number of flows in each experiment.

We consider the following criteria in our evaluation:

- **Link utilization:** it measures the the impact of failures on the overall network utilization. Our interest here is to assess how SafeGuard and Sentinel distribute the traffic load in the network after a failure. We use byte counters in the switches to compute link utilization.
- **Route stretch:** it measures the backup route length (number of hops) in terms of *route stretch*, which is defined as the ratio of the length of the longest path

[3]CPqD is a software switch that runs OPenFlow 1.3 and is intended for fast experimentation purposes.

[4]Iperf is a commonly used network testing tool that can create TCP and UDP data streams and measure the throughput of a network that is carrying them.

used for traversing a flow to the length of the shortest possible route for that flow [75]. Route stretch can affect end-to-end delay and link utilization.

- **Memory usage:** it measures how much switch memory a failure recovery approach uses to make the network resilient to any possible single link failure. We count the total number of OpenFlow rules being install at the flow table and group table of the switches.

Link utilization. To measure the utilization of the remaining links, we fix the number of flows at 60 for B4 and 200 for ATT, and randomly fail a link. Fig 3.6 shows the CDF of link utilization over all active links in B4 after a failure. We see a more balanced load distribution in SafeGuard than Sentinel. Moreover, neither approach led to congested congestion, i.e., link utilization equal to one. However, with Sentinel 28% of the links are close to congestion with utilization higher than 80%. In SafeGuard only for 20% of links the link utilization is higher than 80%.

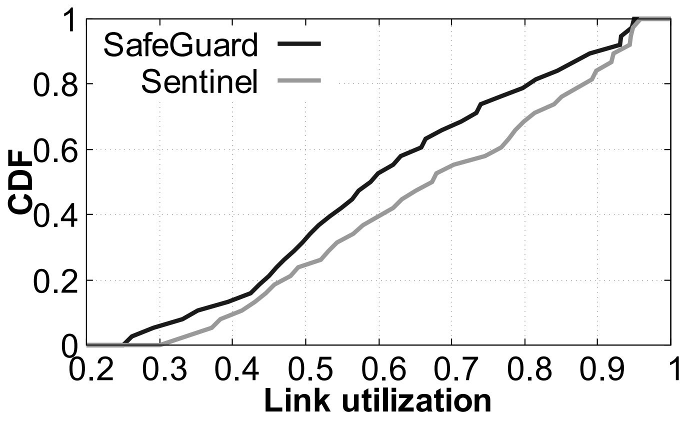


Figure 3.6: CDF of link utilization in B4

Fig 3.7 shows the CDF of link utilization over all active links in B4 after a failure. We see a more balanced load distribution in SafeGuard than Sentinel. Moreover, neither approach led to congested congestion, i.e., link utilization equal to one. However, with Sentinel 28% of the links are close to congestion with utilization higher than 80%. In SafeGuard only for 20% of links the link utilization is higher than 80%.

In ATT, we see have an overall higher link utilization compared to B4 because of the higher number of flows. Nevertheless, SafeGuard outperforms Sentinel with a bigger margin. particularly, the load imposed by SafeGuard is above 80% for around

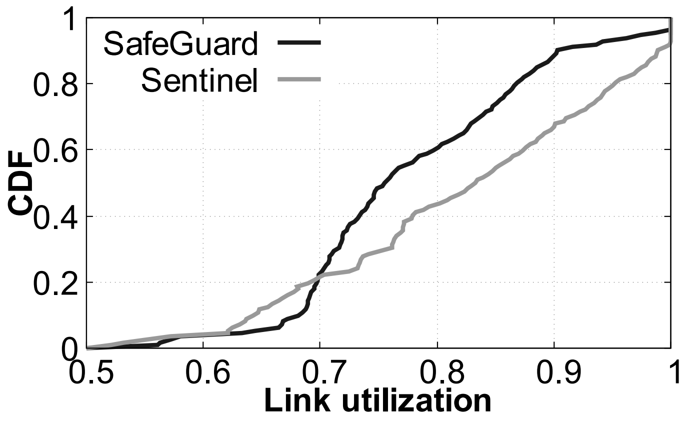


Figure 3.7: CDF of link utilization in ATT

40% of the links while in Sentinel this number increases to 57%. That is because Sentinel is confined to link-disjoint paths and, thus has less options to balance traffic, while SafeGuard can use any link to reroute traffic, even along the affected primary route. Interestingly, both approaches yield at some level of congestion after the failure. However, the number of congested links is 50% smaller in SafeGuard compared to Sentinel.

Memory usage. Here we generate traffic between 20-60 and 40-200 randomly selected source-destination pairs in B4 and ATT networks, respectively. Then we count the total number of forwarding rules (primary and backup) installed at all switches to forward packets. As oppose to Sentinel, SafeGuard takes into account the memory usage and allows non-disjoint routes. Accordingly, we expect that SafeGuard requires less memory as it enjoys the resource sharing. That means in SafeGuard a backup path can include some links of the primary ones, and use the same rules for forwarding traffic on common links.

Fig. 3.8 depicts the total number of rules required to forward traffic for different number of flows in the B4 topology. Error bars indicate the 95% confidence interval. SafeGuard occupies less memory space than Sentinel in all cases, offering up to 17% lower memory usage for 60 flows. Fig. 3.9 shows the number of rules required for ATT topology. Not surprisingly, we see a higher number of rules than the B4 scenario as there are more flows. In this case, SafeGuard entails around 10% fewer rules than Sentinel.

Route stretch. Here again, we consider 20-60 and 40-200 randomly selected

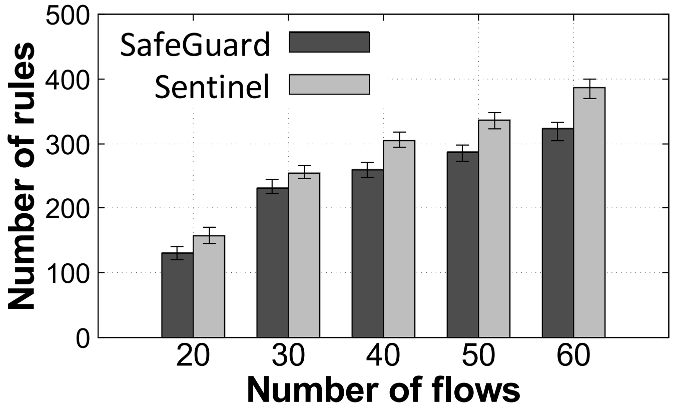


Figure 3.8: Average route stretch in B4

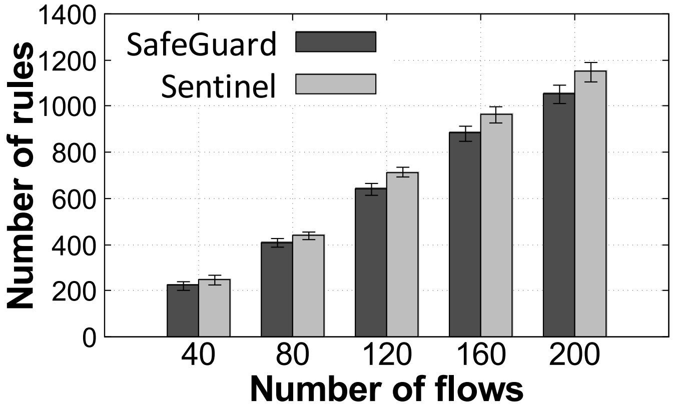


Figure 3.9: Average route stretch in ATT

flows in B4 and ATT networks, respectively. Then we randomly fail links and calculate the average route stretch for all flows. Fig. 3.10 shows the average route stretch of SafeGUard and Sentinel in B4, which represent the superiority of SafeGuard over Sentinel in for B4 topology. With 60 flows, on average, the longest route in SafeGuard and Sentinel after failures is 1.38x and 1.64x of the primary (shortest) route, respectively. That implies SafeGuard deploys shorter backup routes compared to Sentinel, which are 15% shorter on average. We also see a similar performance trend is similar for ATT topology 3.11. Computed route stretch in SafeGUard reflects our design's objectives for reducing the backup route length. Sentinel, on the other hand, does not take into account the route length and requires link disjoint paths. This results in following longer routes compared to SafeGuard.

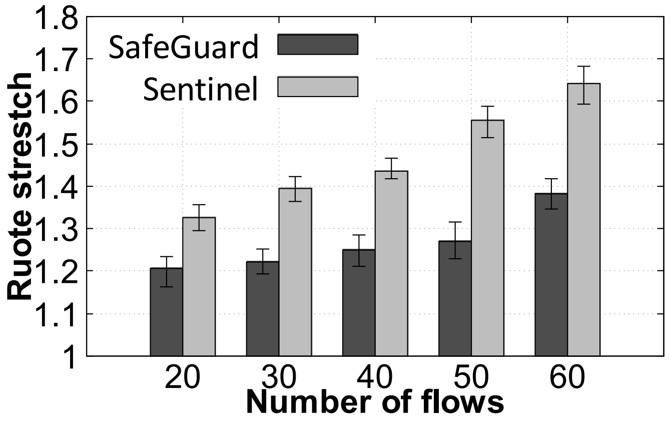


Figure 3.10: Average number of forwarding rules in B4

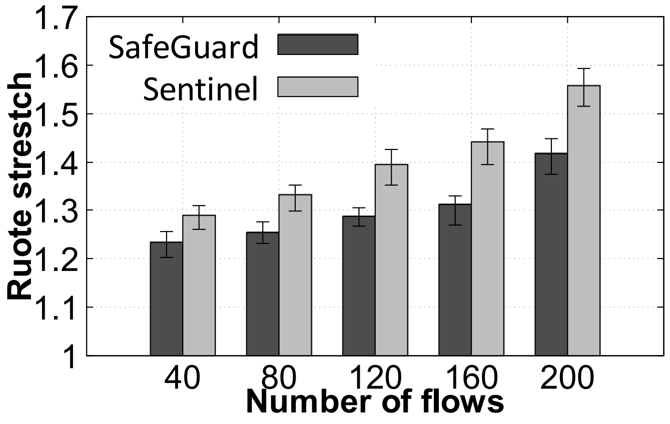


Figure 3.11: Average number of forwarding rules in ATT

Chapter 4

Extending SafeGuard to Multi-Priority Traffic with Non-unitary Switch Memory Demand

In this chapter we extend our failure recovery framework in SD-WAN, SafeGuard (chapter 3), in two important directions:

- **Traffic with different priorities:** in SafeGuad, we assumed all traffic flows have the same priority. However, WAN traffic usually includes flows from different applications with different QoS requirements. In order to meet these requirements, providers typically consider priority classes while routing traffic.
- **Non-unitary switch memory demand:** in SafeGuad, we adopted unitary switch memory demand, where we required one forwarding rule for each flow, which is actually the aggregate flows, between a ingress-egress switch pair. However, there are usually hundreds of active connections between a source-destination pair every TE interval.

Accordingly, we extend SafeGuard to incorporate the above specifications. We also perform significant new evaluation over several metrics and real topologies [9].

4.1 Problem Formulation

We present SafeGuard's new model in this subsection. Although this new model shares some common characteristics with the previous one, we restate those characteristics in this chapter to be more consistent.

4.1.1 Network Design

Similar to the previous chapter, we present a network as a digraph $G = (V, E)$, where $V = \{1, 2, .., n\}$ is the set of network switches (or nodes) and E is the set of links among the switches. Each link $(i, j) \in E$ has a bandwidth capacity c_{ij}, and each node $i \in V$ has memory capacity Tc_i. We consider traffic with different priorities in our model. To this end, each flow demand f in this network represents the *aggregate* traffic from an ingress to an egress switch at a certain priority class. Therefore, we presume three attributes for each flow demand: i) a priority class h_f; ii) a switch memory demand m_f; and iii) a bandwidth demand b_f. Also, each flow demand is divided among a set of primary routes P_f. We use t_{fp} to denote the traffic rate of flow f on each route $p \in P_f$.

With the above settings, a link failure impacts all flows traversing that link. Q_f denotes the the set of backup routes for impacted flow f. Each route $p \in Q_f$ concatenates two routes: a portion of the impacted primary route from ingress switch to failing switch, and an alternative route from the failing switch to the egress switch of the respective flow. Since the primary routes are not necessarily link-disjoint, a link failure could impact one or multiple primary routes of a flow. $P_f^{'}$ denotes the set of non-affected primary routes. We use both backup and non-affected primary routes to forward flow f after a failure. Table 4.1 summarizes this notation.

4.1.2 Variables and Parameters

Each traffic flow impacted by link failure e needs to forwarded on its backup and unaffected primary routes. Accordingly, we define binary decision variables x_{fp}, which denotes the routes used for impacted flow f after recovery, such that

$$x_{fp} = \begin{cases} 1 & \text{if } route\ p \in P_f^{'} \bigcup Q_f\ is\ used\ to\ forward \\ & traffic\ from\ flow\ f, \\ 0 & otherwise. \end{cases} \tag{4.1}$$

Also, we define decision variable $y_{fp} \geq 0$ to denote the traffic rate of impacted flow f on route p after the failure.

Lastly, we define binary parameters, $r_{ij,p}$ and $s_{i,p}$, to identify the links and nodes belonging to route p, respectively:

$$r_{ij,p} = \begin{cases} 1 & \text{if } \ (i,j) \in p, \\ 0 & otherwise. \end{cases} \tag{4.2}$$

Table 4.1: Key notations used in the model.

	Notation	Description
Input	$G(V,E)$	Network graph with switches V and Links E;
	P_f	Set of primary routes for flow f;
	Q_f	Set of backup routes for flow f;
	t_{fp}	Traffic rate of flow f on route p;
	h_f	Priority class of flow f;
	m_f	Switch memory demand of flow f;
	b_f	Bandwidth required for flow f;
	c_{ij}	Total capacity of link $(i,j) \in E$;
	Tc_i	TCAM space capacity of node $i \in V$;
Auxiliary variables	e	a failed link;
	F_e	set of impacted flows by link failure e;
	F	set of flows not impacted by link failure e;
	$P_f^{'}$	set of primary routes for flow $f \in F_e$ not impacted by link failure e;
	l_p	length of route $p \in P_f^{'} \bigcup Q_f$;
	$r_{ij,p}$	1 if link $(i,j) \in p$, 0 otherwise;
	$s_{i,p}$	1 if node $i \in p$, 0 otherwise;
Output	x_{fp}	Allocation of affected flow f on route $p \in P_f^{'} \bigcup Q_f$;
	y_{fp}	Traffic rate of affected flow f assigned to route p;

$$s_{i,p} = \begin{cases} 1 & \text{if} \quad i \in p, \\ 0 & otherwise. \end{cases} \tag{4.3}$$

4.1.3 Constraints

Link and node capacity. constraints in (4.4) ensures that link utilization from does not exceed the maximum capacity of that link. The overall utilization of a link is total traffic rates assigned to both affected ($f \in F_e$) and unaffected ($f \in F$) flows, either for primary or backup routes. Constraints in (4.5), on the other hand, ensures that switch memory usage does not exceed the witch capacity. As in link capacity constraints, the total switch memory usage is the summation demands from primary and backup routes of both affected and unaffected flows.

$$\sum_{f \in F_e} \sum_{p \in P'_f \bigcup Q_f} y_{fp} r_{ij,p} + \sum_{f \in F} \sum_{p \in P_f} r_{ij,p} t_{fp} \leq c_{ij} \ \ \forall (i,j) \in E \tag{4.4}$$

$$\sum_{f \in F_e} \sum_{p \in P'_f \bigcup Q_f} m_f x_{fp} s_{i,p} + \sum_{f \in F} \sum_{p \in P_f} m_f s_{i,p} \leq Tc_i \ \ \forall i \in V \tag{4.5}$$

Traffic rate. Variables x_{fp} and y_{fp} are correlated using constraints (4.6). If p is not used by f then traffic rate of flow f on route p is zero (i.e., $x_{fp} = 0$). Otherwise, it must be a positive number.

$$x_{fp} \leq y_{fp} \leq x_{fp} b_f, \quad \forall f \in F_e, p \in P'_f \bigcup Q_f \tag{4.6}$$

Flow satisfaction. constraints in (4.7) are flow demand satisfaction, i.e., they ensure that the total traffic rate a flow on its primary and backup routes is equal to its demand.

$$\sum_{p \in P'_f \bigcup Q_f} y_{fp} = b_f \ \ \forall f \in F_e \tag{4.7}$$

4.1.4 Objective Function

The objective function (4.8) targets optimizing two criterion: the first term takes care of the backup route length, while the second term accounts for the link capacity usage. We represent link capacity usage as the ratio of the traffic load on a route to the capacity of the link included in the route. Here again we useα and β parameters to bring to parts of the objective function to the same scale.

$$\min \sum_{f \in F_e} \sum_{p \in P'_f \bigcup Q_f} (\alpha h_f l_p x_{fp} + \beta \sum_{(i,j) \in E} \frac{r_{ijp} y_{fp}}{c_{ij}}) \tag{4.8}$$

4.2 Heuristic

In this section, we present our proposed heuristic. Algorithm 2 shows the procedure. The inputs to the algorithm are: the network topology after removing the failed link e, the set of affected flows F_e, and a set of candidate routes P containing both backup routes and non-affected primary ones for all affected flows. Here we adjust the traffic rates of those affected flows over the their candidate routes. The traffic rate assigned to a route is related to the length and the bottleneck capacity of that route.

We iterate over the flows with the highest to the lowest priority (line 1, where H denotes the set of traffic classes sorted in descending order). Then we sort the flows in the same priority class in the descending order (lines 3-2). For a given flow, at first, we sort its available routes in ascending order in an attempt to forward that with as shortest routes as possible (line 5). Then we iterate over all the candidate routes and check whether the *memory* usage of all switches along that route are within a capacity threshold (lines 6-7). If the memory usage is within the threshold, we allocate the minimum between the flow demand and the remaining bandwidth capacity (a_{ij}) of the bottleneck link (line 8). Otherwise we apply best-fit strategy, i.e. we allocate the flow only if it can be fully satisfied (line 9-12). This way we ensure that smaller flows can also have access to the shorter routes. If the traffic rate allocation is not zero, we update the remaining capacity (a_{ij}) and remaining memory of the links and switches, respectively, along that route (lines 14-17). We also update the remaining flow demand to be allocated (line 14-17).

The time complexity of Algorithm 2 is proportional to the number of affected flows from a link failure and corresponding set of routes, and also the number of traffic classes we considered. The complexity of the algorithm is $O(KMN(n + r \log r))$, where $|H| = K$, $|F_e| = M$, $|N| = P$, $|V| = n$, and $|E| = r$.

We illustrate the operation of Algorithm 2 in Fig. 4.1 [9]. There are three flows $\{f_1, f_2, f_3\}$: f_1 with traffic class 3 from switch S_1 to S_6 has a demand of 15Mbps, f_2 with traffic class 3 from switch S_2 to S_6 has a demand of 10Mbps, and f_3 with traffic class 2 from switch S_3 to S_6 has a demand of 5Mbps. Their routing tunnels and corresponding rates are shown in Fig. 4.1(a). For operational simplicity, we assume that all switches have enough memory to accommodate these these flow rules. We also assume that the bandwidth capacity of all links is 30Mbps except link $S_1 - S_5$ and $S_5 - S_6$. whose bandwidth capacity is 10Mbps. The failure of link $S_4 - S_6$ impacts all flows $\{f_1, f_{2,f_3}\}$. In the following, we illustrate the operation of the heuristic.

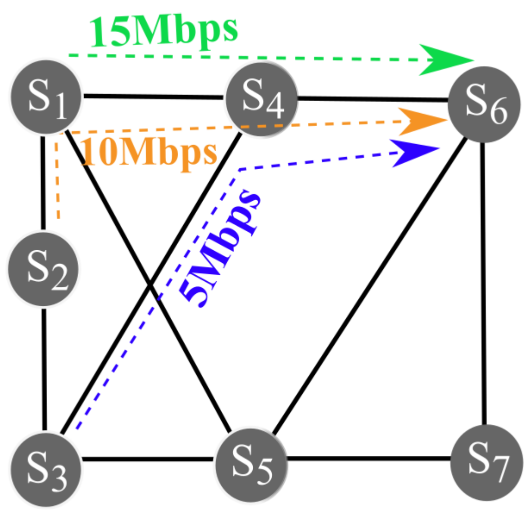
Algorithm 2 starts with flow f_1 (green flow) and f_2 (blue flow) as they have the higher priority. Because f_1 has the higher demand than f_2, algorithm 2 forwards f_1

Algorithm 2: Heuristic for rerouting traffic

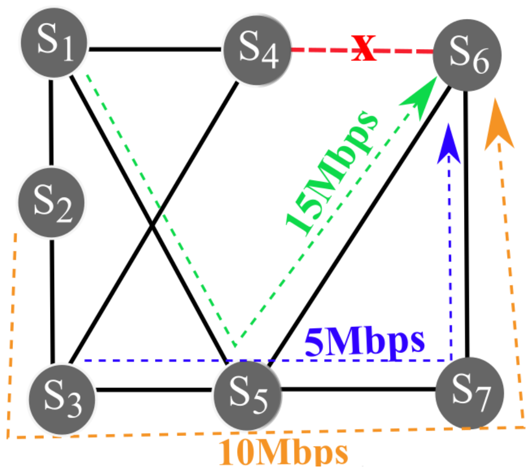
```
Input  : Network topology $G' = (V, E \backslash e)$; affected flows $F_e$; priority classes
         $H$; available routes $P = P_f^{'} \bigcup Q_f, \forall f \in F_e$
Output: Traffic rates assigned to $F_e$

1  for $h \in H$ do
2      $F_h = \{f \in F_e | h_f == h\}$
3      Sort $F_h$ in descending order according to flow demands
4      for $f \in F_h$ do
5          Sort $P = P_f^{'} \bigcup Q_f$ in ascending order in terms of route length
6          for $p \in P$ do
7              if $\forall s \in p, u_s + m_f < \lfloor Tc_s \times \mu \rfloor$ then
8                  $y_{pf} = \min\left(\min_{(i,j)\in p} a_{ij}, d_f\right)$
9              else if $\forall s \in p, u_s + m_f < Tc_s$ then
10                 if $d_f \leq \min_{(i,j)\in p} a_{ij}$ then
11                     $y_{pf} = d_f$
12                 end
13             end
14             if $y_{pf} > 0$ then
15                 Update $u_s, a_{ij}$ $\forall s, (i,j) \in p$
16                 $d_f \leftarrow d_f - y_{pf}$
17             end
18         end
19     end
20 end
```

over its shortest backup tunnel first: $S_1 - S_5 - S_6$. Then, flow f_2 will be forwarded over tunnel $S_2 - S_3 - S_5 - S_7 - S_6$ (note that the shorter tunnel $S_2 - S_1 - S_5 - S_6$ has not more capacity to accommodate flow f_2). Lastly, flow f_3 with the lowest priority, will be forwarded over its shortest tunnel with available capacity: $S_3 - S_5 - S_7 - S_6$.



(a)



(b)

Figure 4.1: a) Traffic rates on the primary routes of two flows (blue and black). The available capacity of all links is 20Mbps except link $S_4 - S_5$ whose capacity is 5Mbps; b) the available tunnels for the affected flow with the highest demand and corresponding traffic rates assigned to them; c) the available tunnels for the second affected flow after the link failure with the traffic rates assigned to them.

4.3 Evaluation

4.3.1 Setup

Our experiments run on a machine with 2.66GHz 12 core CPU and 44GB RAM equipped with Mininet (version 2.2.2). We consider four real network topologies: Google B4 [4], ATT [70], Dial Telecom [76] and Cogent [77]. Their main characteristics is summarized in table 4.2. In particular, we selected networks with distinct number of nodes, diameters (i.e., length of the longest shortest paths), and amount of link-disjoint routes among source-destination pairs. Each node is deployed as a CPqD switch instance [74]. Link capacities are set to 1 Gbps. We benchmark SafeGuard against Sentinel, the state-of-the-art SD-WAN failure recovery approach in SD-WAN. Sentinel relies on the link-disjoint tunnels to distribute the traffic rate of affected flows. Sentinel targets minimizing the maximum link utilization while not considering the length of the backup tunnels or switch memory usage.

We assume the switch memory demand of for a flow is a random number selected from interval $[20, 200]$. This is because, each flow consists of 100s TCP flows [5] which are made of 20% elephant flows and 80% mice flows. Elephant flows usually

Table 4.2: Network topologies used in our evaluation. LD= link-disjoint.

Topology	**#Nodes**	**#Links**	**Diameter**	**#LD paths**		
				Min	**Average**	**Max**
B4	12	38	5	2	2.35	4
ATT	25	112	5	2	3.2	9
Dial Telecom	192	302	30	2	2.6	4
Cogent	197	478	28	2	2.7	6

are treated with exact matching and mice flow are usually treated with wildcard rules [78]. Accordingly, for each flow, we require one flow rule for each elephant TCP flow and one flow rule for the remaining mice TCP flows.

Traffic matrix generation: We use TMgen [79] for the traffic matrix generation based of the gravity model. In Newton's law of gravitation, the force is proportional to the product of the masses of the two objects divided by the distance squared. Inspired by the this concept, gravity model assigns a weight w_i to each node i, and assumes that the traffic demand from i to j is proportional to $w_i w_j$ [80]. We generate three TMs using gravity model: the first, second and third TMs will contribute 50%, 30% and 20% to the traffic load. Lastly, we use iperf tool to generate UDP traffic representing TMs.

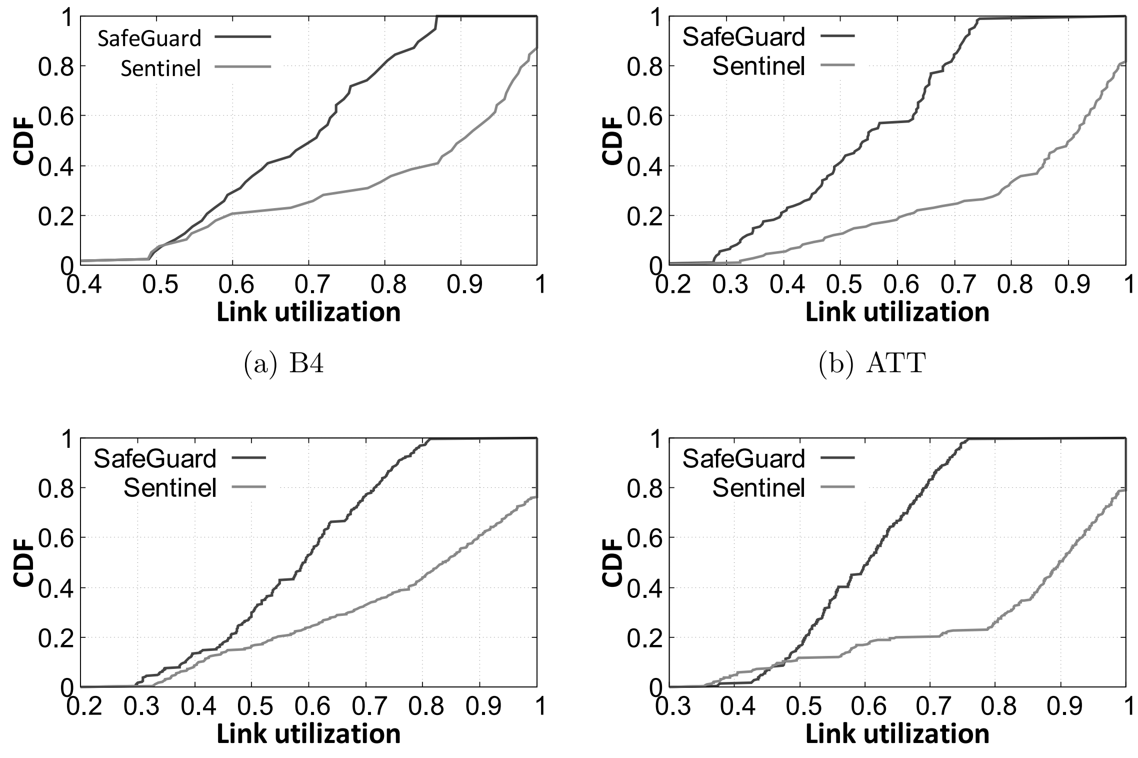
4.3.2 Disscussion on the result

To learn how SafeGuard performs in different settings, we run two sets of experiments: considering different topologies while fixing the traffic load, and taking one network topologies and vary the traffic load. These will reveal where SafeGuard can perform well. We compare SafeGuard against Sentinel in terms of different performance metrics: link utilization, route stretch, memory usage, packet loss and round-trip time. In all experiments, we randomly fail a link and calculate the desired metric.

4.3.2.1 The impact of topology

in this subsection, we fix the traffic load to 100Mbps and measure the link utilization and route stretch for four topologies (presented in table 4.2).

Link utilization. Fig. 4.2 shows the CDF of link utilization across all active links for four topologies after a failure. As it can be seen in the figure, after the link failure: SafeGuard creates no congestion in all topologies, while Sentinel can tolerate the failure in none of the topologies. In all topologies SafeGuard is far from congestion with link peak utilization below 90%. At the same time, Sentinel creates severe congestion to more than 20% of the links. As oppose to Sentinel that requires link-disjoint paths, SafeGuard enjoys using any active link to reroute traffic providing them with more option to balance the load than. Moreover, SafeGuard distributes the affected flows at the granularity of flow priorities. This enables SafeGuard to handle affected flows with higher demands before consuming the available links' capacity by the lower demand flows.



(a) B4

(b) ATT

(c) Dial Telecom

(d) Cogent

Figure 4.2: CDF of link utilization for different topologies

Route stretch. Fig. 4.3 shows the route stretch in four topologies. In all topologies SafeGuard takes shorter backup paths compare to Sentinel. Sentinel needs to take link-disjoint backup paths which are commonly much longer than the primary ones (twice longer in some cases). SafeGuard, on the other hand, can circumvent the failing link with two extra links to reach the destination in many cases. Also, the uniform load distributing by SafeGuard leaves free capacity on the short paths to be used by the flows whose demands fit those short paths. Additionally, the nodes in ATT and B4 are closer than nodes in Cogent and Dial Telecom. this means a slightly longer backup route in ATT and B4 induces a large route stretch value. For example, in the case of SafeGuard: in Cogent, the longest path is 1.09x of the primary (shortest) path while this value increases to 1.32x in B4.

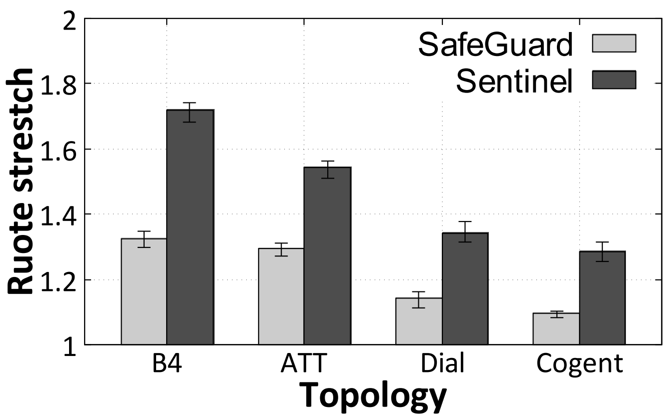
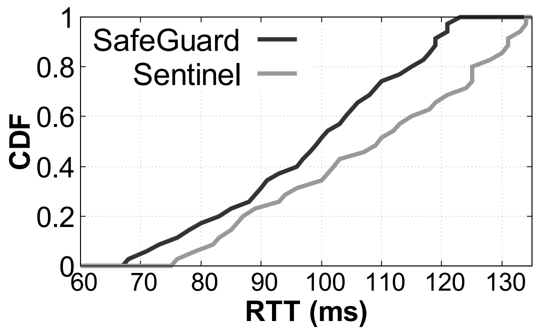


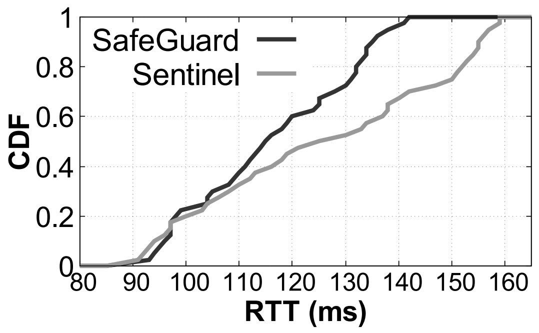
Figure 4.3: Route stretch for different topologies.

Latency. Now we look at the latency as the immediate consequence of either congestion or longer backup routes. Figure 4.4 depicts the CDF of the round-trip times (RTT) for every source-destination pair in the evaluated topologies. The RTT is higher in bigger topologies, i.e. Dial Telecom and Cogent, as there are more hops between each source-destination pair in those topologies. Nonetheless, SafeGuard outperforms Sentinel in all scenarios. In particular, it reduces tail latencies by 12, 13, 15 and 18% for B4, ATT, Dial Telecom and Cogent topologies, respectively. This is because SafeGuard offers lower congestion and shorter routes.

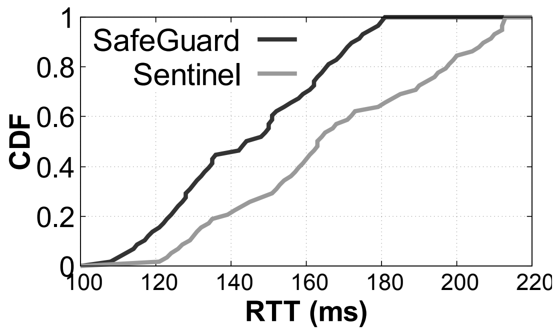
Switch memory usage. Both TE schemes install backup forwarding rules proactively at the switches memory. Fig. 4.5 represents the switch memory usage for each TE scheme. With larger networks typologies, the switch memory usage is larger for



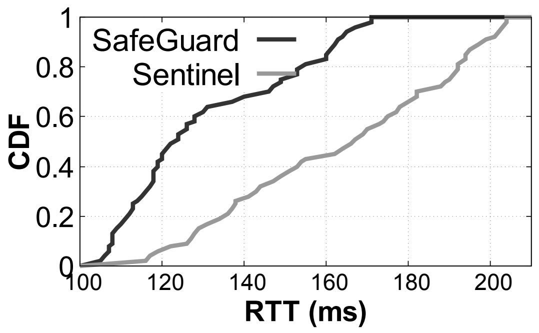
(a) B4



(b) ATT



(c) Dial Telecom



(d) Cogent

Figure 4.4: CDF of round trip time (RTT) for different topologies.

both TE schemes.

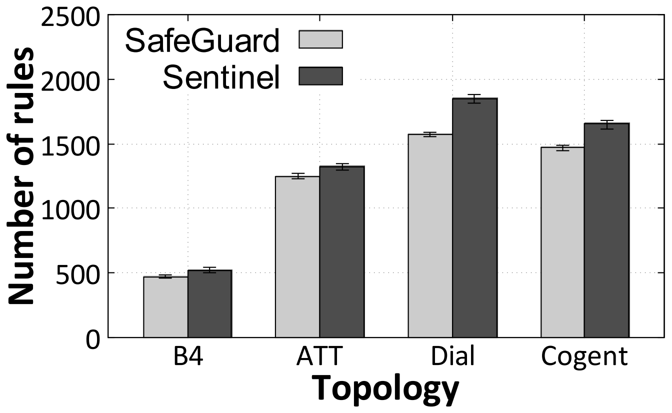


Figure 4.5: Switch memory usage for different topologies.

Also, in Cogent the switch memory usage is slightly lower than than in Dial Telecom, as it is denser than Dial Telecom. This allows using the TE approaches to use the non-affected part of a failed path leading to lower stitch memory usage. Nonetheless, Sentinel occupies the switch memory around 15% more than SafeGuard.

This is because SafeGuard allows using non-disjoint path, which is not the case in Sentinel. Thus, they offer sharing resources, i.e., two paths can take advantage of the same rules for forwarding traffic on overlapping links.

4.3.2.2 The Impact of Load

To examine the effects of traffic load, we consider Cogent topology with different traffic loads: 50, 100, 150, 200, and 250 Mbps.

Link utilization Fig. 4.6 illustrates the box plots of of the link utilization. In all cases Sentinel creates an unbalanced link utilization (long boxes) while creating some congestion in the network (upper whiskers). In SafeGuard 75th percentile is always lower than 90%, although the maximum link utilization is 1 under 250Mbps.

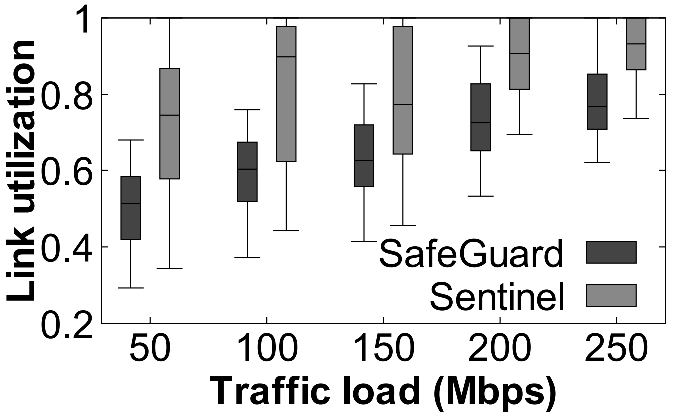


Figure 4.6: CDF of link utilization for different loads.

Route stretch. Fig 4.7 illustrates the route stretch of and Sentinel for different traffic loads. The route stretch increases with the increase in traffic load. This is because the higher load on the links leaves a smaller scope for three approaches when selecting the backup paths for the flows. Here SafeGuard outperforms Sentinel by up to 18% better performance, specially under high loads.

Latency. Figure 4.8 shows the RTT of packets after a link failure as we vary the traffic load. The median RTT is $7 - 26\%$ lower in SafeGuard compared to Sentinel. The superiority of SafeGuard decreases slightly with the increase in the traffic load. This is because SafeGuard has a higher scope for selecting backup routes under the lower traffic loads.

Switch memory usage. Fig. 4.9 plots the switch memory usage for each TE

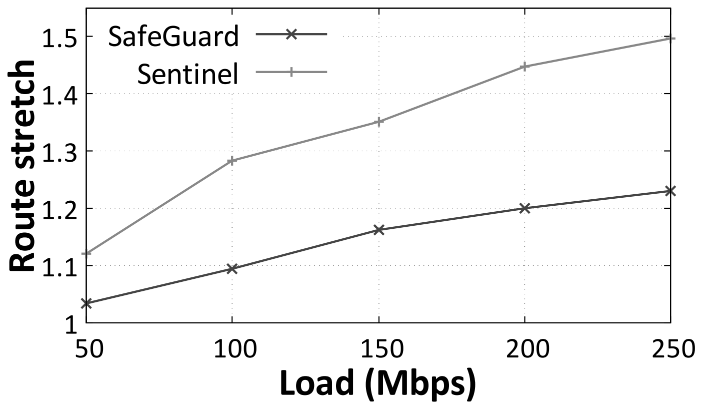


Figure 4.7: The route stretch for different loads.

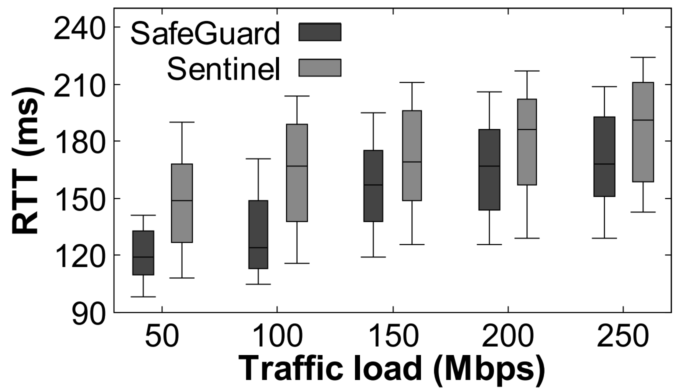


Figure 4.8: RTT for different traffic loads

scheme for different loads. As the loads increases, the memory usage increases as well. This is because both TE schemes have access to less number of paths after the failure enforcing them to allocate flows to longer paths. However, Sentinel occupies more switch memory than SafeGuard, as it requires the link-disjoint tunnels needing to install forwarding rules on new switches along the backup tunnels.

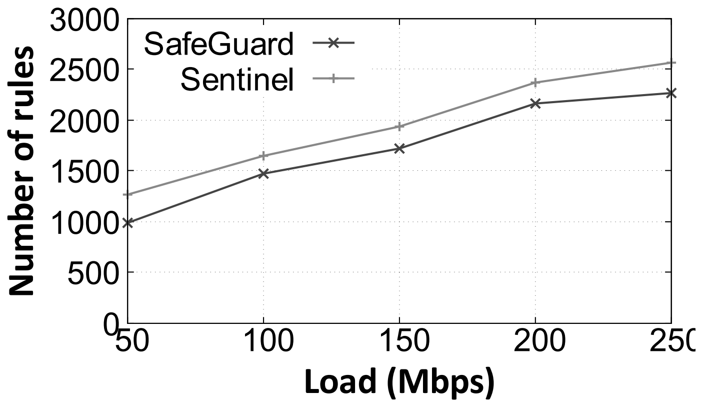


Figure 4.9: The memory usage for different loads.

Chapter 5

Conclusion and Future Works

5.1 Conclusions

In this thesis, we proposed a novel failure recovery system, called SafeGuard, for SD-WANs. We modeled the failure recovery problem as a multi-objective MILP optimization problem, which incorporates link bandwidth capacity, route length, and switch memory constraints for proactively allocating backup routes. We discussed the NP-hardness of our model and evaluated it using CPLEX showing that it does not scale to the large networks. Then, we designed a heuristic to solve the model. We demonstrated the efficiency of our heuristic by analyzing its solution accuracy and runtime.

To evaluate the performance of SafeGuard, we implemented a prototype of it using the Ryu SDN controller. We used Mininet, an SDN emulator, and deployed the flow and group tables of OpenFlow protocol to implement SafeGuard. We made our code publicly available to allow the community to reproduce our results and extend the proposed system. We performed an extensive evaluation of SafeGuard over several topologies with different sizes and densities. We compared the performance of our system with the state-of-the-art SD-WAN failure recovery scheme. We showed that SafeGuard can quickly reroute the traffic with low delay, while uniformly distribute the traffic over the network and efficiently use the network resources.

5.2 Future Works

In future work, we plan to bring this work one step further by extending our SD-WAN failure recovery system in the following aspects:

- In this work we considered single link failure according to the recent studies on the probability of link failures [81]. We plan to extend our model and heuristic to include multiple failures.

- We also plan to consider SDN controller failure in future work. In a recent controller performance evaluation study, the authors compared two distributed controllers ONOS and OpenDaylight, where the former outperforms the latter [82]. Thus, we plan to deploy ONOS instances in our design and extend our model and heuristic to incorporate controller failures.

- Although we considered the switch TCAM limitation on our system, another idea that can further improve our solution to optimize network resource utilization is to further minimize the TCAM usage using wild-card rules. We plan to enhance our system by this complementary idea.

- Our work is an offline failure recovery system, i.e., it needs to select paths between each ingress-egress switch pair and assign corresponding traffic rates for a given set of demands at each TE cycle. However, this may not be efficient if some flows last for multiple TE cycles. As our future work, we plan to adopt an online approach, where at every TE cycle, we monitor the demands from previous TE cycles and allocate rate over the chosen paths for the newly arrived flows within the resource budget.

- In this work, the link bandwidth capacity is the only cost occurred when a traffic flow traverses a link. However, other metrics cloud also be considered for the link cost. In the future, we want to extend this work to IPv6 routing, where we consider the default OSPF cost, delay and L2 factor [83].

- We also plan to deploy randomized or weighted randomized route selection schemes [84–86] and extend our model and heuristic.

Appendix A

A.1 How to run SafeGuard?

In this section we explain how one can run our failure recovery system in SD-WAN and how it works. Our open source code is publicly available in [19].

A.1.1 Setting up the environment:

We need the following software packages and tools to run SafeGuard:

- Download and install Mininet from [73].
- Install Ryu controller in Mininet using `pip install Ryu` command or from the source code in [87].
- Install CPqD software switch from [74].
- install TMgen tool from [88].

We also need three Python scripts from our open sourcce code: `topology.py`, `initialization.py` and `SafeGuard.py`. The first two scripts are for setting up a virtual SDN. The `SafeGuard.py` script is a Ryu controller application that runs the SafeGuard's failure recovery system. To run SafeGuard, we need to have two Mininet terminals open. In the first terminal we lunch the network, and in the other one we run our controller application. In the following we present the most important modules we used at each part.

A. Lunching the SDN topology: We launch the network by `topology.py` script in one Mininet terminal using `sudo python topology.py` command.

How it works? `topology.py` script calls the `initialization.py` script, which defines the network typologies and creates the TMs. At first, `topology.py` has a `network` class, which sets up the network switches and the links among them.

After setting up the network, multiple threading is used for concurrent traffic generation. `myTraffic` method that calls the helper method `doIperf` to generate the traffic between each source destination pair. `myTraffic` also injects a random link failure in the network.

Lastly, to lunch the network topology, the `runner` method instantiates a network object and creates a remote controller at port number 6653. Switches are set to wait to be connected to the controller.

B. Running the controller application:

Now, that the network is set up, we run the controller application in the second Mininet terminal by `ryu-manager --observe-links SafeGuard.py` command. After lunching the controller, the port statistics will be printed at each 30 second interval on the controller window.

Now, we use Mininet CLI to interact with the network for the traffic generation. We run `pingall` command at the topology window. This leads to installing all primary and backup forwarding rules at the switches. Also, for the traffic generation, we run `myTraffic()` command at the topology window. This will automatically fail a random link in the network and generate UDP traffic using iperf between the list of pairs.

How it works? At first, the controller runs a topology discovery module by listening to events fired by Ryu, which handles the actual communication. It will triggers the a 'switch handler' module any time a switch is added to the controller. It also installs a table-miss flow entry at the switch allowing them to send packets to the controller.

When all switches are registered, the a module computes the primary and backup routes between each source-destination pair for any single link failure in the network. For each source-destination pair, it removes a given link from the network, and computes the corresponding backup routes.

Now, the heuristic' module deploys the computed primary and backup routes, and runs the SafeGuard's heuristic to configure the traffic rate allocation, given any single link failure in the network. After configuring the primary and backup routes and installing the flow and group entries, a 'link failure handler' module handles the link failures. Whenever a failure happens, the switch that detects the failure will send a message to the controller, which triggers the 'link failure handler' module. It will then updates the flow table at the ingress switch of the affected flows to adjusts the tariff rate.

A.2 How to run our model implementation?

We implemented our model using Python API of CLPEX. To run it, we need to download the `MILP mode` script from our open source code at [19]. The fallowing software steps should be taken to run the model implementation:

- Install Python
- Install CPLEX from [69]
- Set up the Python API of CPLEX using the instruction presented in [89]

Now, one can run the model by running the `MILP mode` in any Pythom editor. We ran our model in Python version 3.6.0. We recommend using the same Python version.

Bibliography

[1] D. Kreutz, F. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” *IEEE*, 2015.

[2] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” vol. 103, no. 1, 2014, pp. 14–76.

[3] N. Feamster, J. Rexford, and E. Zegura, “The road to sdn: an intellectual history of programmable networks,” vol. 44, no. 2, 2014, pp. 87–98.

[4] S. Jain and et al., “B4: Experience with a globally-deployed software defined wan,” vol. 43, no. 4, 2013, pp. 3–14.

[5] C. Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, “Achieving high utilization with software-driven wan,” 2013, pp. 15–26.

[6] “George leopold. 2017. building express backbone: Facebook’s new long-haul network.” 2017. [Online]. Available: http://code.facebook.com/posts/1782709872057497/

[7] J. Zheng, H. Xu, X. Zhu, G. Chen, and Y. Geng, “Sentinel: failure recovery in centralized traffic engineering,” vol. 27, no. 5, 2019, pp. 1859–1872.

[8] M. Shojaee, M. Neves, and I. Haque, “Safeguard: Congestion and memory-aware failure recovery in sd-wan,” 2020, pp. 1–7.

[9] M. Neves, M. Shojaee, and I. Haque, “Multi-resource aware failure recovery in sd-wan,” 2020.

[10] H. H. Liu, S. Kandula, R. Mahajan, and D. Zhang, M.and Gelernter, “Traffic engineering with forward fault correction,” 2014, pp. 527–538.

[11] K. T. Foerster, Y. A. Pignolet, S. Schmid, and G. Tredan, “Casa: congestion and stretch aware static fast rerouting,” 2019, pp. 469–477.

[12] J. Bogle, N. Bhatia, M. Ghobadi, I. Menache, N. Bjørner, A. Valadarsky, and M. Schapira, “Teavar: striking the right utilization-availability balance in wan traffic engineering,” 2019, pp. 29–43.

[13] S. Q. Zhang, Q. Zhang, A. Tizghadam, B. Park, H. Bannazadeh, R. Boutaba, and A. Leon-Garcia, “Tcam space-efficient routing in a software defined network,” no. 125, 2017, pp. 26–40.

[14] T.-H. T. . G. M. Mohan, P. M., “Tcam-aware local rerouting for fast and efficient failure recovery in software defined networks,” 2015, pp. 1–6.

[15] J. Li, J. Hyun, J. H. Yoo, S. Baik, and J. W. K. Hong, “Scalable failover method for data center networks using openflow,” 2014, pp. 1–6.

[16] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, “Infinite cacheflow in software-defined networks. in proceedings of the third workshop on hot topics in software defined networking,” 2014, pp. 175–180.

[17] P. C. Lekkas, “Network processors: Architectures,” 2013.

[18] Y. Chen, S. Jain, Z. L. Adhikari, V. K.vand Zhang, and K. Xu, “A first look at inter-data center traffic characteristics via yahoo! datasets,” 2011, pp. 1620–1628.

[19] “Safeguard,” https://github.com/Meysam-Sh/SafeGuard.git, (accessed Dec. 16, 2020).

[20] e. a. McKeown, N., “Openflow: enabling innovation in campus networks,” in *ACM queue*, vol. 38, no. 2, 2008, pp. 69–74.

[21] “Openflow switch specification version 1.3.0,” https://www.cs.princeton.edu/courses/archive/fall13/cos597E/papers/openflow-spec-v1.3.2.pdf, (accessed Sep. 8, 2020).

[22] P. C. da Rocha Fonseca and E. S. Mota, “A survey on fault management in software-defined networks,” vol. 19, no. 4, 2017, pp. 2284–2321.

[23] C. Doerr and F. Kuipers, “All quiet on the internet front?” vol. 52, no. 10, 2014, pp. 46–51.

[24] M. a. Chiesa, “The quest for resilient (static) forwarding table,” 2016, pp. 1–91.

[25] A. Shaikh, C. Isett, A. Greenberg, M. Roughan, and J. Gottlieb, “A case study of ospf behavior in a large enterprise network,” 2002, pp. 217–230.

[26] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C. N. Chuah, and C. Diot, “Characterization of failures in an ip backbone,” vol. 4, 2004, pp. 2307–2317.

[27] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, “Evolve or die: High-availability design principles drawn from googles network infrastructure,” 2016, pp. 58–72.

[28] G. Liang and K. Kökten, “On diagnosis of forwarding plane via static forwarding rules in software defined networks,” 2014, pp. 1716–1724.

[29] V. Adrichem, B. J. N. L., Van Asten, and F. A. Kuipers, “Fast recovery in software-defined networks,” 2014, pp. 61–66.

[30] O. N. Foundation, “Openflow switch specification ver 1.4,” 2013.

[31] A. Farrel, A. Satyanarayana, A. Iwata, and . A. G. Fujita, N., “Crankback signaling extensions for mpls and gmpls rsvp-te,” 2007.

[32] S. Kandula, D. Katabi, B. Davie, and A. Charny, “Walking the tightrope: Responsive yet stable traffic engineering,” vol. 35, no. 4, 2005, pp. 253–264.

[33] X. D. D. R. J. D. Suchara, M. and J. Rexford, “Network architecture for joint failure recovery and traffic engineering,” vol. 39, no. 1, 2011, pp. 97–108.

[34] G. M. . G. Y. Tootoonchian, A., “Opentm: traffic matrix estimator for openflow networks,” 2010, pp. 201–210.

[35] X. H. Q. L. Y. Y. R. Z. Y. Wang, H. and A. Greenberg, “Cope: traffic engineering in dynamic networks,” 2006, pp. 99–110.

[36] R. J. Fortz, B. and M. Thorup, “Traffic engineering with traditional ip routing protocols,” vol. 40, no. 10, 2002, pp. 118–124.

[37] J. C. L. S. Elwalid, A. and I. Widjaja, “Mate: Mpls adaptive traffic engineering,” vol. 3, 2001, pp. 1300–1309.

[38] B. Fortz and M. Thorup, “Internet traffic engineering by optimizing ospf weights,” vol. 2, 2000, pp. 519–528.

[39] . T. M. Fortz, B., “Optimizing ospf/is-is weights in a changing world,” vol. 20, no. 4, 2002, pp. 756–767.

[40] M. J. A. J. O. M. Awduche, D. and J. McManus, “Requirements for traffic engineering over mpls.”

[41] C. Y. Hong and et al., “B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google’s software-defined wan,” 2018, pp. 74–87.

[42] S. A. Astaneh and S. S. Heydari, “Optimization of sdn flow operations in multi-failure restoration scenarios,” vol. 13, no. 3, 2016, pp. 421–432.

[43] M. Kuźniar, P. Perešíni, N. Vasić, and D. Canini, M.and Kostić, “Automatic failure recovery for software-defined networks,” 2013, pp. 159–160.

[44] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker, “Ensuring connectivity via data plane mechanisms,” vol. 13, 2013, pp. 113–126.

[45] B. Charron-Bost and J. Welch, J. L. ad Widder, “Link reversal: How to play better to work less,” 2009, pp. 88–101.

[46] M. Borokhovich, L. Schiff, and S. Schmid, “Provable data plane connectivity with local fast failover: Introducing openflow graph algorithms,” 2014, pp. 121–126.

[47] S. Sharma and et. al, “Demonstrating resilient quality of service in software defined networking,” 2014, pp. 133–134.

[48] E. Lakiotakis, C. Liaskos, and X. Dimitropoulos, “Improving networked music performance systems using application-network collaboration,” vol. 31, no. 24, 2019, p. e4730.

[49] K. Phemius and M. Bouet, “Implementing openflow-based resilient network services,” 2012, pp. 212–214.

[50] S. Paris, G. S. Paschos, and J. Leguay, “Dynamic control for failure recovery and flow reconfiguration in sdn,” 2016, pp. 152–159.

[51] J. Nagano and N. Shinomiya, “A failure recovery method based on cycle structure and its verification by openflow,” 2013, pp. 298–303.

[52] D. Kim and J. M. Gil, “Reliable and fault-tolerant software-defined network operations scheme for remote 3d printing,” vol. 44, no. 3, 2015, pp. 804–814.

[53] J. T. Araújo, R. Landa, R. G. Clegg, and G. Pavlou, “Software-defined network support for transport resilience,” 2014, pp. 1–8.

[54] D. Staessens, S. Sharma, D. Colle, M. Pickavet, and P. Demeester, “Software defined networking: Meeting carrier grade requirements,” 2011, pp. 1–6.

[55] M. e. a. Alizadeh, “Software defined networking: Meeting carrier grade requirements,” 2010.

[56] A. Ghannami and C. Shao, “Efficient fast recovery mechanism in software-defined networks: multipath routing approach,” 2016, pp. 432–435.

[57] Z. Cheng, X. Zhang, Y. Li, S. Yu, R. Lin, and L. He, “Congestion-aware local reroute for fast failure recovery in software-defined networks,” vol. 9, no. 11, 2017, pp. 943–944.

[58] F. Tang and I. Haque, “Remon: A resilient flow monitoring framework,” in *In 2019 Network Traffic Measurement and Analysis Conference (TMA)*, 2019, pp. 137–144.

[59] A. Capone, C. Cascone, A. Q. Nguyen, and B. Sanso, “Detour planning for fast and reliable failure recovery in sdn with openstate,” 2015, pp. 25–32.

[60] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, “Openstate: programming platform-independent stateful openflow applications inside the switch,” vol. 44, no. 2, 2014, pp. 44–51.

[61] Z. Zhu, Q. Li, M. Xu, Z. Song, and S. Xia, “A customized and cost-efficient backup scheme in software-defined networks,” 2017, pp. 1–6.

[62] U. Lekhala and I. Haque, “Piqos: A programmable and intelligent qos framework,” in *Proceedings of the 2019 IEEE INFOCOM workshop on Network Intelligence*, 2019.

[63] M. Moyeen, , F. Tang, D. Saha, and I. Haque, “Sd-fast: A packet rerouting architecture in sdn,” in *15th International Conference on Network and Service Management (CNSM)*. IEEE, 2019.

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

[65] I. Haque, S. Islam, and J. Harms, “On selecting a reliable topology in wireless sensor networks,” in *Proceedings of the 2015 IEEE International Conference on Communications*, ser. ICC ’15, 2015.

[66] O. Tilmans and S. Vissicchio, “Igp-as-a-backup for robust sdn networks,” in *10th International Conference on Network and Service Management (CNSM) and Workshop*. IEEE, 2014, pp. 127–135.

[67] Y. Wang, H. Wang, A. Mahimkar, R. Alimi, Y. Zhang, L. Qiu, and Y. R. Yang, “R3: resilient routing reconfiguration,” 2010, pp. 291–302.

[68] P. Kumar and et al., “Semi-oblivious traffic engineering: The road not taken,” 2018, pp. 157–170.

[69] “Ibm cplex optimizer,” https://www.ibm.com/ca-en/products/ilog-cplex-optimization-studio, (accessed Sep. 11, 2020).

[70] “Att network topology,” http://www.topology-zoo.org/maps/AttMpls.jpg, (accessed Agu. 16, 2020).

[71] G. L. Wolsey, L. A. abd Nemhauser, “Integer and combinatorial optimization,” vol. 55, 1999.

[72] M. R. Garey and D. S. johnson, “Computers and intractability: a guide to the theory of np-completeness,” vol. 24, no. 1, 1982.

[73] “Mininet,” http://mininet.org/, (accessed Nov. 7, 2020).

[74] “Cpqd software switch,” https://github.com/CPqD/ofsoftswitch13, (accessed Oct. 7, 2020).

[75] L. J. Cowen, “Compact routing with minimum stretch,” vol. 38, no. 1, 2001, pp. 170–183.

[76] “Dial telecom topology,” http://www.topology-zoo.org/maps/DialtelecomCz.jpg, (accessed Aug. 18, 2020).

[77] "Cogent topology," http://www.topology-zoo.org/maps/Cogentco.jpg, (accessed Agu. 16, 2020).

[78] G. Zhao, H. Xu, S. Chen, L. Huang, and P. Wang, "Joint optimization of flow table and group table for default paths in sdns," vol. 26, no. 4, 2018, pp. 1837–1850.

[79] P. Tune and M. Roughan, "Spatiotemporal traffic matrix synthesis," in *In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 579–592.

[80] Y. Zhang, M. Roughan, N. Duffield, and A. Greenberg, "Fast accurate computation of large-scale ip traffic matrices from link loads," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, 2003, pp. 206–217.

[81] "Private conversation with researchers," 2015.

[82] M. Darianian, C. Williamson, and I. Haque, "Experimental evaluation of two openflow controllers," in *in the proceeding of IEEE ICNP workshop on PVE-SDN*, Oct 2017.

[83] "Ospfv3," https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_ospf/configuration/15-1sg/ip6-route-ospfv3.html, (accessed Dec. 14, 2020).

[84] I. Haque, I. Nikolaidis, and P. Gburzynski, "On the benefits of nondeterminism in location-based forwarding," in *International Conference on Communications (ICC)*, 2009.

[85] T. Fevens, I. Haque, and L. Narayanan, "A class of randomized routing algorithms in mobile ad hoc networks," in *AlgorithmS for Wireless and mobile Networks (A SWAN 2004), Boston*, 2004.

[86] T. Fevens, I. T. Haque, and L. Narayanan, "Randomized routing algorithms in mobile ad hoc networks," in *IFIP International Conference on Mobile and Wireless Communication Networks*, 2004.

[87] "Ryu sdn controller," https://github.com/faucetsdn/ryu, (accessed Sep. 21, 2020).

[88] "Tmgen tool," https://github.com/progwriter/TMgen, (accessed Nov. 5, 2020).

[89] "set up python api of cplex," https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.cplex.help/CPLEX/GettingStarted/topics/set_up/Python_setup.html, (accessed Nov. 27, 2020).