國立清華大學電機資訊學院資訊工程研究所
碩士論文
Department of Computer Science
College of Electrical Engineering and Computer Science
National Tsing Hua University
Master Thesis

在軟體定義網路下利用標籤交換之流量工程系統

Traffic-Engineer in Software Defined Networks using Label Switching

茅辰寧
Chen-Nien Mao

學號：104062551
Student ID:104062551

指導教授：徐正炘 博士
Advisor: Cheng-Hsin Hsu, Ph.D.

中華民國 106 年 05 月
May, 2017

國立清華大學
資訊工程研究所

碩士論文

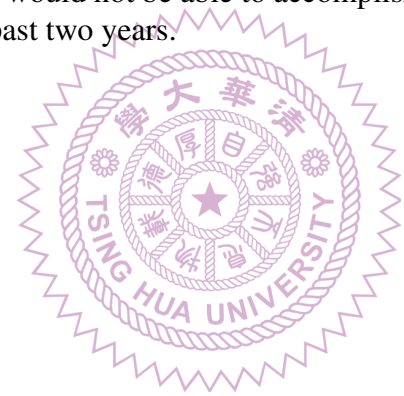
在軟體定義網路下利用標籤交換之流量工程系統

茅辰寧 撰

106
05

Acknowledgments

I would like to express my gratitube toward all the people who helped me in the past two years. I wouldn't be able to finish my thesis were it not for your help along the way. I want to thank my parents specifically, for it is they who provided me with the firm support and stand behind my decisions. I would also like to thank my labmates in Networking and Multimedia Systems Laboratory, especially Tao-Ya Fan Chiang, who helped me a great deal in the course of my research. Lastly, I would like to express my gratitude toward my adviser: Prof Cheng-Hsin Hsu. Without the guidance and the suggestion I received from him, I would not be able to accomplish what I have done and learn so much in the past two years.



致謝

在此我要感謝在過去兩年中所有幫助過我的人， 如果沒有你們的幫助我一定沒有辦法順利完成我的論文。 在此我要特別感謝我的父母，他們提供我堅定不移的支持， 同時也支持我所作的每一個決定。我也要感謝網路與多媒體系統實驗室的同學們，特別是范姜陶亞在過去兩年的研究中幫助我非常的多。 最後，我要感謝我的指導教授：徐正炘教授。如果沒有他的給予我的指導以及建議， 在過去的兩年內我一定沒辦法完成如此多的事情以及學到如此多的東西。

中文摘要

在此論文中，我們探討在軟體定義網路 (SDNs)下網路流量工程的優化。我們將原本網路虛擬化成眾多虛擬管道，每條管道都會由一個標籤代表，而所有的路由都會由標籤進行優化。為了達到更適應性的路由，我們將原先交換機中的路油表(Flow Table)區分為: 1) 儲存管道路由資訊的管道表 2) 儲存流量與管道之間對應關係的流量表，而我們提出四種演算法優化儲存的路由資訊與對應關係，並且施行流量工程以優化系統。我們實作真實測試環境來驗證我們的方法，並且除了實作我們提出的系統架構外，我們另外也實作分段路由(Segment Routing, SR)，也利用Ryu提供的簡單路由(Ryu Simple Routing, RSR)。實驗結果顯示出我們的系統在最高鏈接使用量和初始延遲的兩項指標中都超越SR跟RSR: 1) 與SR相比，我們的系統有效降低20%到46%的最高鏈接使用量；跟RSR相比，我們更進一步降低30%到50%的使用量。2) 跟SR相比，與我們的系統降低了39%的初始延遲；以及與RSR相比，93.2%的初始化延遲被降低。除此之外，我們的方法更為系統帶來了負載平衡和快速錯誤回復的能力。

Abstract

We study the problem of Traffic Engineering in multi-site Software Defined Network (SDN). In our system, we virtualize the physical links with virtual tunnels and each tunnel will be represented by a label. We propose to decouple the flow table into two tables: the more static tunnel tables, and the more dynamic path table, for higher flexibility. The tunnel tables contain a set of pre-built tunnels, while the path table matches each traffic flow to the labels. We formulate two optimization problems for tunnel construction and path assigner. Four algorithms are purposed to solve the Traffic Engineering properly, and a real testbed is implemented to prove the concept. We also compare the proposed solution with Segment Routing (SR) and Ryu Simple Routing (RSR), the experimental results show that our solution outperforms SR and RSR: 1) maximal link utilization reduction between 20% and 46% compared to SR, 30% and 50% compared to RSR. 2) Initial delay reduction by 39% compared to SR, and 93.2% compared to RSR. Furthermore, our proposed solution also brings load balancing and error resilience features to the network system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Currently, the explosive traffic flows which come from different services have brought many challenges to the networks. The Quality of Service (QoS) and Quality of Experience (QoE) become more and more challenging due to the massive data transmissions. The core network components play a crucial role in processing these data and they usually connected to different types of networks, from the wireless network to conventional IP networks. Therefore, the traffic engineering inside the core networks is the key to building a modern network system. The traditional core networks perform Interior Gateway Protocol (IGP) which relies on shortest path algorithms to route the traffic is not sufficient enough for the massive traffic nowadays. The links are usually suffered from congestions in some network conditions while there are still idling resources in the network. The unbalanced resource usage leads to the poor performance and even congestion, which seriously affects the QoS and QoE.

To overcome these issues, Software Defined Network (SDN) [28] with OpenFlow protocols has shown its strength in traffic engineering. The SDN controller acts as a composer of the network system, and it is able to change routing rules by programming the switches under the same SDN domain. The SDN is has been widely deployed in the data center networks. For example, Google has drawn our attention by deploying B4 [20] in their own internal networks. Although SDN solves many problems in conventional networks, however, the speed of real world deployment can not keep up with the evolution speed of SDN and there are some remaining traffic engineering problems : (i) initialization time in large networks, (ii) scalability, (iii) QoS, and (iv) flexibility.

First of all, QoE is seriously impacted by the initialization delay. The problem is due to all the switches along with the traffic path needs to communicate with the controller, the round-trip times between each switch and controller enlarge the initialization or reroute delay. Furthermore, the problem will be amplified by the size of networks, and the distance of the traffic paths.

Second, decoupling the data flow and control flow in SDN brings the scalability issues. The scalability may be constrained by the limited size of flow entries and the deployment cost of SDN supported devices. The size issue may be easily achieved by hardware, but the infrastructure cost of SDN switches is relatively high. On the other side, the scalability of SDN is also constrained by the real network architectures, and we can not assume the network is all under SDN. Thus, how to carefully transmit the packets between SDN networks and the non-SDN networks is a critical challenge when we deploy SDN in the real network.

Third, OpenFlow protocol brings the straightforward QoS support into their standard. However, the OpenFlow provides basic QoS features and the First In First Out policy is not suitable for the large core networks. The routing mechanism needs to be carefully designed to perform traffic engineering and utilize the resources wisely inside the network. More precisely, the crucial task of traffic engineering is to optimize the resources allocation inside the networks.

Last, although SDN decoupling the data flow and control flow which seems to be more flexible for the network administrator to manage the network. However, the cost of changing routing decision is still too high for SDNs. In order to change routing rules, SDN controller needs to program all the affected switches, both the original switches and new switches. To handle different demands of traffic from various service, the core network has to be evolved to provide flexibility. The cost of changing routing behavior need to be optimized.

1.1 Contribution

In this work, we study the above issues in SDNs and purposed a flexible architecture to simplify the core networks, and well-designed routing algorithms to perform traffic engineering in SDNs. To address the flexibility and reliability problem in our system, we forward the traffic along multiple virtual paths. Purposed path finding algorithms and load balancer also brings load balancing and fast recovering to the system. More precisely, our proposed system has the following features:

- Low initialization latency
- Load-balance
- Scalability
- Resilience and QoS support
- Simplified mechanism in core network

1.2 Thesis Organization

In this thesis, we first give general introductions to the background knowledge using in this work in chap. 2. In chap. 3, we show the high-level overview of the proposed system and then the usage scenarios of the proposed system. To perform traffic engineering, we discussing the focusing problems in chap. 4. In chap. 5, we give a detail introduction of the components using in our system and the innovation approaches we used to perform traffic engineering. In chap. 6 and chap. 7, we introduce the design goal of our algorithms and describe the routing behaviors in detail, and then introduce the dynamic routing algorithms in chap. 8. Chap. 9 gives the detail implementation of proposed system and the testbed to proof the concept. Besides, we also implement two different solutions in our testbed as the baselines. We describe the experiment setup and presents the results in chap. 10. Finally, we survey the related work in chap. 11. We conclude our contributions and discuss the future works in chap 12.

Chapter 2

background

2.1 Software Defined Network

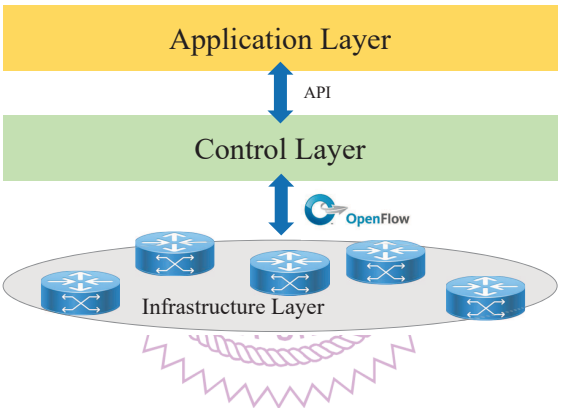


Figure 2.1: Architecture of SDN.

SDN is first presented in 2008 and it has recently drawn tremendous attentions from both academia and industry. Different from the legacy IP network, separating the control plane and data plane shows the strength of SDN and makes it easier to deploy new innovations in real-world networks, which in turn speeds up the evolution of computer networks. In the IP forwarding networks, each switch will handle all the forwarding mechanisms by running its own routing protocols. However, in SDNs, the routing rules are mainly decided by the centralized controllers, controllers will decide the routing rules of each traffic request and then program the network nodes. Thus, the network nodes in SDN are only responsible for forwarding the packets. Fig. 2.1 shows the architecture of SDN. The controller generally has the global view of the infrastructure layer and using specific protocols to communicate with the network nodes under same SDN domain. The most well-known protocol is OpenFlow [28], which is supported by Open Networking Foundation [33]. Besides, the controller can also communicate with the applications to support various services, which is able to achieve Network Function Virtualization.

2.2 Tunneling in the Internet

Tunneling is a network engineering which transmits the packet over virtual tunnels, and these tunnels usually traverse among several physical links. The packets are first archived inside a routing header which has a tunneling information. The original source and destination are encapsulated, and the switch will forward these packets using the tunneling information. The tunneling information can be represented by various types. For example, IP Encapsulation Within IP (IP in IP) [24, 37], and Generic Routing Encapsulation (GRE) [16, 24] are generally used in IP tunneling. In the cellular network, GPRS Tunneling Protocol (GTP) [2] is used to forward the packets.

2.3 Segment Routing

Segment Routing (SR) [45] is a new solution to allow network administrators to perform network engineering. The idea is extended from source routed, which store all the routing decision on each passing switches inside the packet header. The switches follow these routing rules to forward the packet instead of lookup its own forwarding table. Similar to source routed, SR also stores the routing decisions inside the packet header, but the routing information is formed by a sequence of segments. The switches will check segments inside the header and direct the packet to the defined segment. [14] shows the architecture of SR and [6, 12, 43] also proposed their approaches to SR.

2.4 Multi-Protocol Label Switching

Multiprotocol Label Switching (MPLS) [3, 46, 52], which forwards the packets according to the label without looking up the network address. A label stands for a label-switched path (LSP), which is usually set up by the Label Distribution Protocol (LDP) and RSVP-TE under an MPLS domain. The packet will be encapsulated inside an MPLS header which contains the label information, and the switches inside the MPLS domain will forward the packet according to the labels. MPLS also be known as the ability to perform traffic engineering in the IP network [47], and it usually performs with Constrained Shortest Path First (CSPF), Open Shortest Path First (OSPF) or Intermediate System To Intermediate System (IS-IS) protocols.

2.5 Traffic Engineering

According to the definition of Traffic Engineering in RFC [49]. Traffic engineering focuses on the optimization of network performance. The performance metrics includes the 1) traffic management, 2) resource allocation, 3) minimization of delay and packet loss, 4) maximization of throughput. These challenges have to be careful design to achieve Traffic Engineering in the Network. In this thesis, we consider the above metrics and brings 1) load balancing, 2) minimize the delay and 3) error resilience into our proposed system. The load balancing is an important feature in traffic engineering, and we have addressed the issue with two optimization algorithm. In the rest of the thesis, we will go through our proposed system and solve the traffic engineering carefully.

Chapter 3

Usage Scenario

3.1 Network Topology

We give an overview of the proposed system considered in this thesis. We assume all sites run on SDNs and all the switches in the same subnetwork are working with OpenFlow protocols which controlled by a logically centralized OpenFlow controller. To simplify the SDN solution, we classify the switches as core switches and edge switches and leverage the concept proposed in Fabric [7] to design our architecture. More precisely, we design our system to follow the bellowing scheme. The complexity procedures are handled by the edge switches (ingress and egress switches), the core switches are only responsible for forwarding the packets. The edge switches will be programmed by the controller so that the packets can go into the network, and leave to the destination appropriately. The detailed forwarding mechanism will be discussed in ch. 5.

3.2 Label Switching Operation

Label Switching will forward the packet relying on label number, which makes the switching between two nodes faster. The label is generally imposed between the data link layer and network layer. The label header contains a label which can be identified by the local domain and the header length is short and fixed (32-bits for MPLS label) [32]. In our proposed system, we use MPLS to realize the label switching, however, we part with the original protocols using in MPLS system. Instead, we manipulate the features in SDN to magnificently reduce the label distribution protocols. More precisely, we extract the concept of MPLS label in our system and let the SDN controller finish the routing jobs.

The reasons for choosing MPLS label include :(i) stackable, providing higher extensibility, (ii) fixed length, allowing more efficient matching, and (iii) supported in OpenFlow protocol. In the legacy IP networks, MPLS shows its ability to perform traffic engineering

and provide QoS, however, there are many issues need to be considered by the network designers while deploying MPLS [52], e.g., the scope of the whole system, the hierarchy of MPLS system and the path attributes of label switching paths etc. With the benefit of SDN, these issues may be solved in SDNs environment. SDN controllers have the global view of the whole system and network designers can easily deploy MPLS label inside the networks. More precisely, the label distribution protocols can be replaced by SDN controller due to the direct communication between switches and controller.

With the help of SDN, we leverage the MPLS labels and build a label switching system to perform traffic engineering to optimize the network. Most important of all, label switching paths can be designed into flow table, which meets the spirit of SDN.

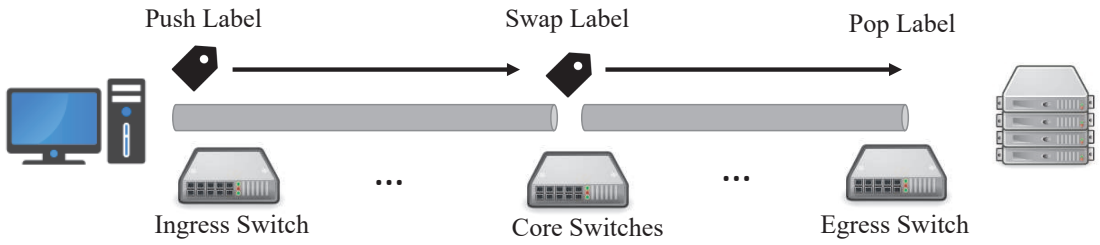


Figure 3.1: Packet forwarding mechanism in proposed solution.

Fig. 3.1 introduces tree different label switching actions which generally used in our proposed system. When a packet first enters the network controlled by the proposed controller, it will be **Push**ed a specific label. Then, the labeled packet is able to travel among multiple physically links without interacting with the SDN controller. When the packet reaches the egress switch, it will decapsulate the labeled packet. The label header will be removed and forward to the destination host according to the address. Finally, the **Swap** action replaces the old label with a new one, which redirects the packet to another path. In our system, it is possible that the label will be changed inside the core network to make the forwarding mechanism more flexible. The swap action comes from a sequence of actions: 1) **Pop** the original label, and 2) **Push** a new assigned label.

To better understand our system, we give two usage scenarios to demonstrate the proposed forwarding mechanism. We take enterprise networks to demonstrate the system, however, the same solution can be used in different scenarios without any modification.

Fig. 3.2 shows a use case in which multiple sites are connected by a network owned by the enterprise. The network comprises of two sites, and each of them represents a label switching domain which is controlled by logically centralized SDN controllers with a global view respectively. The controllers will coordinate with each other to optimize the site-to-site network flows.

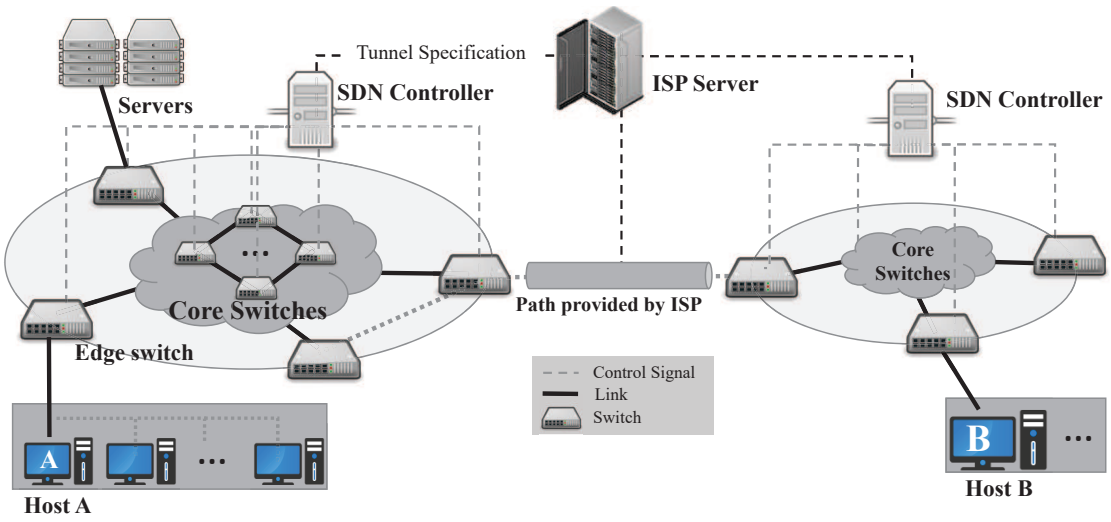


Figure 3.2: Use cases in a multiple sites enterprise network.

In the first example, if the host A wants to send data to the server on the same site, the packets will first be forwarded to the ingress switches near host A. If the switch doesn't contain the corresponding rules to forward the packet, the packet-in event will be sent to the controller and controller responses with the label(s) which stand for a dedicated path. Then, the rules will be recorded in the traffic table at the ingress switch, which allows the same traffic request transmit the packet automatically.

On the other hand, if host A wants to communicate with host B in different sites. In this case, the site-to-site path can be provided by Internet Service Providers (ISPs). The traffic flows will be transmitted with the label assigned by SDN controllers inside our own domain, and then transmitted to others domain through paths provided by ISPs. For an enterprise, ISPs often provide pre-configured paths using technologies such as Frame Relay [40] or MPLS [39, 52, 53]. These tunnels are owned by ISPs, and thus are not directly managed by our controller. The controller, however, can acquire the tunnel information, such as QoS guarantees, from portals provided by ISPs. Note that the underneath tunneling technologies used by ISPs may be confidential which are out of the scope of this thesis.

Above cases demonstrate how label switching works in enterprise networks. However, the transmission between two nonadjacent network domains can also be traditional IP forwarding. Leveraging MPLS label also makes our proposed solution suitable for legacy MPLS-enabled or eBGP-enabled network without implementing any additional mechanism. Moreover, edge switches can also be programmed with additional rules to support more general scenarios under a hybrid network architecture. Since we are focusing on SDNs, the detail implementations are not provided in this thesis.

Chapter 4

Problem Statement

4.1 Initialization Delay

High initialization and re-route delay may not be a major concern in data centers, as the switches are in close proximity. This may partly explain the success of SDNs in data centers. However, the core networks are vulnerable to long initialization time, because: (i) sites may be geographically distributed, hence the propagation delay is already high, making the additional flow initialization time intolerable, and (ii) network conditions and flow demands are constantly changing, making the flow highly dynamic and frequent new flows further amplify flow initialization time.

To minimize the overall utilization and initialization delay, we first *pre-build* tunnels among all these switches under the same SDNs domain. These tunnels are virtual paths in our system which goes over one or multiple underneath network links. Our well-designed controller is responsible for computing these tunnels and assigning a specific MPLS label on each of them. The tunnels are essentially *virtual* links with some QoS guarantees, such as bandwidth, latency, and packet loss rate.

To route traffic flows, controller *fit* each of them into a suitable *path* that is composed of one or multiple *tunnels* when the requests first come to the ingress switches. Therefore, by using pre-built tunnels, initialization delay can be significantly minimized, because only edges switches are programed when new traffic requests come to the ingress switches.

4.2 Load Balancing

The idea of load balancing is to offload the traffic to another unused link, which utilize the network resource efficiently. [38] has shown the ability to use multiple paths routing to gain better performance. In our proposed system, we also set up more than one tunnels

for each switch pair. By doing so, our controller is able to utilize the idling resources in the network to transmit the packets.

Perform load balancing wisely also help to avoid the chance to congestion. Adopting MPLS label switching in our proposed system allow us to perform load balancing in an elegant way. The load balancing can be conducted by changing the designated label in any switch inside the system, which makes the system more flexible and efficient. Because we don't need to program all the path when the routing decision is changing, the delay cost of changing rules is relatively reduced. Furthermore, instead of using shortest path algorithm, we choose tunnels by computing the disjoint set of each switches pair. The tunnel is then selected by the length of the disjoint paths and the detail implementation is discussed in chap. 6.

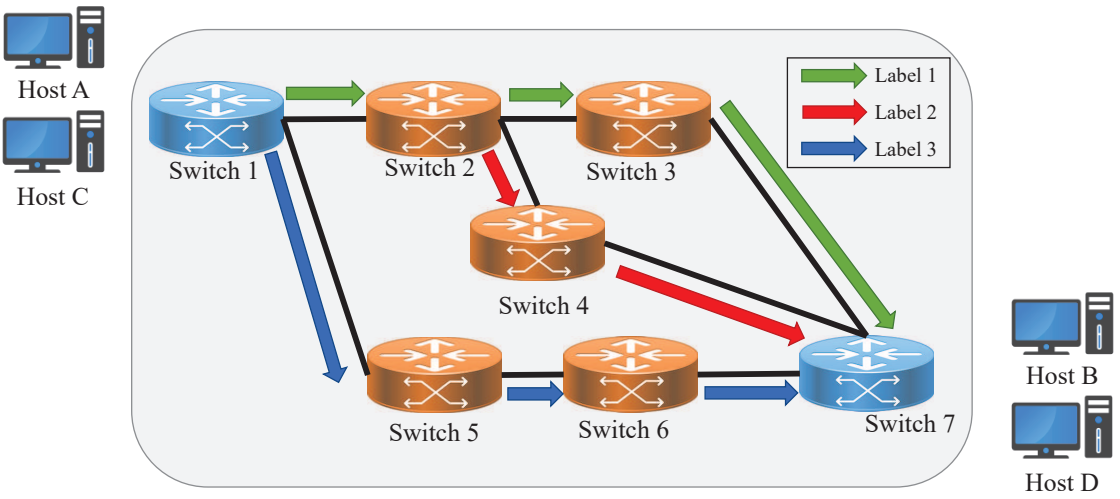


Figure 4.1: Traffic Engineering with MPLS label.

The load balancing features are implemented in our admission algorithm, we give a few examples to demonstrate this features. Fig. 4.1 illustrates the mechanism of load balancing. First of all, traffic flows from H_A to H_B are directed to the tunnel with label 1 and we assume that there is no other traffic on this network. If there is a new traffic from H_C to H_D, the DPA will then choose a path with lower link utilization which may be the tunnel with label 1 or 3 (we choose 3 in this case) to decrease the possibility of congestions. Furthermore, if there is a congestion between S_2 and S_3, the controller can generate a label-swapped event at S_2 and the traffic flows can offload to label 2 to gain better link utilization (To perform load balancing in real time, a network monitor is required, which is out of the scope of this work).

4.3 Error Resilience

Different from the network in the data center, the network in the real world are more dynamic and it is possible that many unforeseen events will occur, which challenges the network system. To handle these events in real time, we adopt redundant tunnels and proposed dynamic algorithms to handle such conditions and bring resilience to the system.

When the system setup, the tunnel finder will patch the network with redundant tunnels to provide more flexible routing choose. On the other hand, the additional tunnels also bring fast-recovery to the system. Network failures can be resolved faster, resulting in shorter downtime experienced by users. We use failure node event to prove the concept, however, the proposed event handlers inside the controller can be further extended to support more various of the failure event, and most of them can be solved by our proposed algorithms (see chap. 8).

Chapter 5

System Architecture

5.1 Controller

5.1.1 Component Diagrams

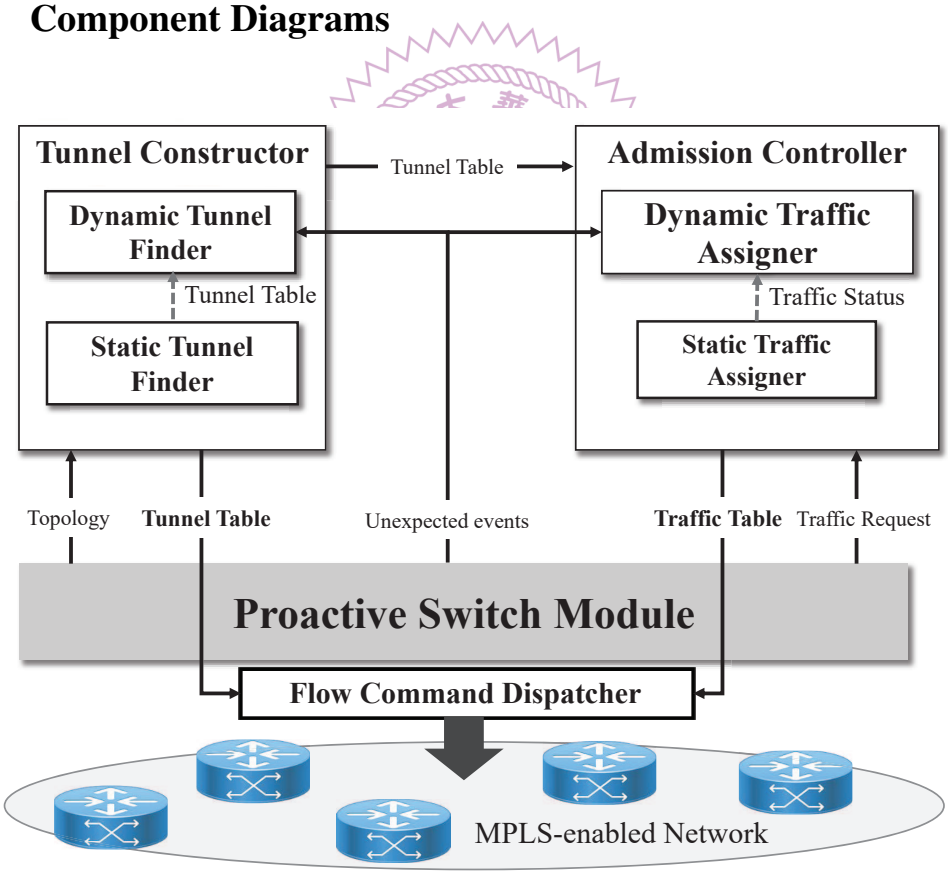


Figure 5.1: Controller components.

We implement five modules inside the controller, and Fig. 5.1 gives the components diagram of proposed controller. The **proactive switch** is the main control logic, which acts as an event handler. The events are first handled by the proactive switch and then

processed by the appropriate modules respectively. Inside our controller, the modules are classified into two group according to their mission: 1) Tunnel Constructor is responsible for building tunnels inside the given network, and 2) Admission Controller is responsible for processing traffic request and perform load balancing. Finally, both the tunnel and path rules are parsed by the **Flow dispatcher** and issues flow-mod commands to the underlying switches with OpenFlow protocol.

5.1.2 Tunnel Constructor

Tunnel Constructor, which composes of two tunnel finder modules, is responsible for finding or recovering tunnels between every two switches. The **Static Tunnel Finder (STF)** module tries to optimize the tunnels among the switches when the system setup or topology change. All the switches inside the network will be programmed with the computed tunnel information, and store in the tunnel tables as mentioned above. The tunnel constructor can also compute backup tunnels in the system, and the network administrators are able to fully configure the parameters to adjust the number of backup tunnels.

On the other hand, if there is any unexpected event occur, the dynamic module, **Dynamic Tunnel Finder (DTF)**, aims to find a possible tunnel in real time. The module is triggered when there are new edge switches connect to the network, or fast recovery mechanism fails to recover the tunnel from the node failure.

5.1.3 Admission Controller

We also implement two modules to perform admission control. **Dynamic Path Assigner (DPA)** acts as an online traffic admission controller which decides whether the system has enough resources to accept more traffic, and assigning a traffic flow to a proper path in real time. Furthermore, we also purposed an offline traffic balancer to achieve load balancing in our system. **Static Path Assigner (SPA)** is our load balancer which aims to balance the traffic periodically in order to avoid the traffic congestions.

5.2 Flow Tables

We propose to decouple an ordinary flow table in a switch into a *tunnel table* and a *path table*, in order to: (i) maintain high interactivity and (ii) support more flexible and dynamic traffic assignments.

In tunnel table, we store pre-built tunnels information, which is computed by the controller when the topology changes. The controller proactively installs the tunnel tables on

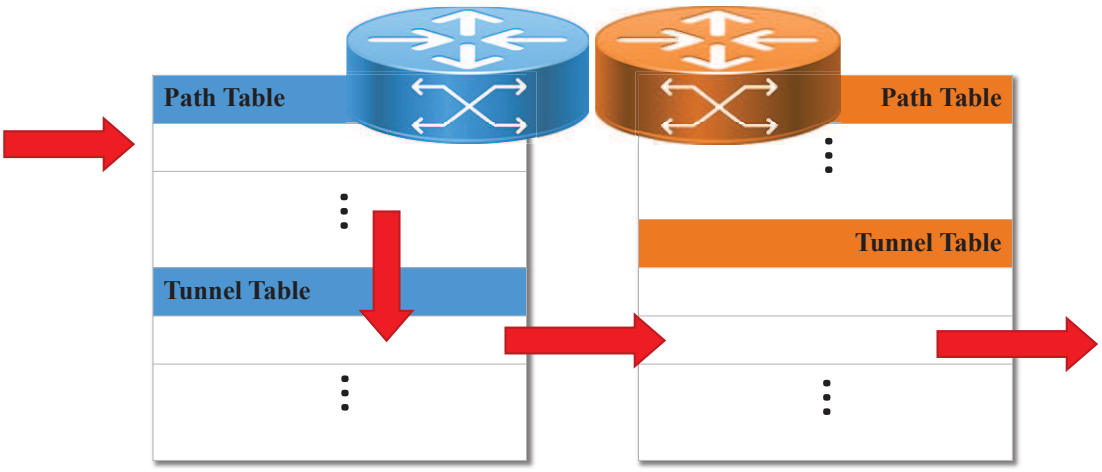


Figure 5.2: Proposed decoupled flow tables.

switches, and these tunnels are represented by MPLS labels respectively. All the information will be translated to flow-mod commands, and store in the flow table inside the corresponding switches. Each flow entries represents a matching relationship between the label and the output port, and the proactive installations of such forwarding information are similar to building routing tables in legacy IP networks.

On the other hand, we store the bindings between labels (tunnels) and the traffic flows in the path table. When a new unlabeled packets visit the network, the controller will compute a path by our path assigner algorithm (DPA), and programs the ingress switch to forward the traffic flow with the label.

Leveraging the novel idea of using two tables, we reduce the complexity of the network architectures. The core switches are simplified and the control plan is mainly separated to the edge switches, leading forwarding mechanism more responsive.

5.2.1 Packet Forwarding

We give an example to illustrate the operations between the controller modules and the decoupled flow tables in Fig. 5.3. First of all, the STF derives all new tunnels inside topology when the system setup. The tunnel information is stored in the controller and then proactive switch module invokes the flow command dispatcher to translate the information into flow-mod commands. The tunnel information will then be stored in the **Tunnel Table** among these switches before any traffic flow arrive in the system. In this example, part of the information about the tunnel table computed by controller are: 1) tunnel 1 from S_1 to S_3 via S_2, and 2) tunnel 2 from S_3 to S_4.

Next, when the traffic request from H_1 to H_2 newly arrives at the ingress switch S_1, the

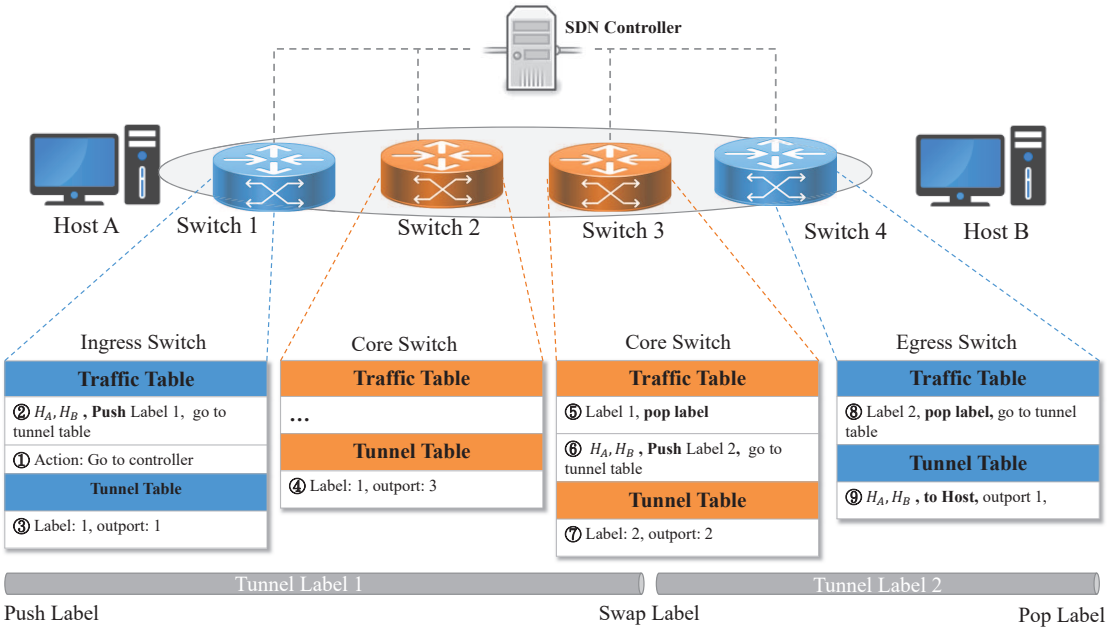


Figure 5.3: Interaction between tables in switches and controller.

packet_in event will be sent to the controller. The **Proactive switch** will invoke the **DTA** to determine if the system has enough resource to accept more traffic into the system, without affecting the ongoing traffic in the network. If the admission is granted by the admission controller, the assigner will response with the labels, which determine the tunnel(s) to use for packets between H_1 and H_2. The proactive switch module then invokes the flow command dispatcher to translate the path table into flow-mod commands.

In this case, controller designates tunnel 1 and tunnel 2 for this request. In ingress switch S_1, the packet will be encapsulated with label 1 and direct the packet to the tunnel 1. All the switches along the tunnel will match and forward the packet according to their tunnel table. When the packet reaches the end of the tunnel, S_3 removes the label from the packet and matches the packet against its path table again, in order to determine if the packet needs to be sent to another tunnel. In this example, S_3 will swap the label and redirect the packet to tunnel 2. When the packet reaches its egress switch S_4, the label will be removed and send to the destination H_B. The detailed packet forwarding procedures are shown below:

1 Packet_in event is sent to the controller due to there is no matching rule in the path table.

2 Controller set up the traffic flow on both ends of the tunnel, and S_1 switch pushes label 1 to the packet.

3 The packet with label 1 will hit the tunnel table which is already set up by the

controller.

4 The packet will transmit among the core switches (only S_2 in this case) with the label.

5 In order to change switch, the label will be swapped in S_3.

6 New label 2 will be pushed to the packet.

7 The label 2 will hit the tunnel rule in S_3 tunnel table.

8 When the packet comes to S_4, it will pop the label.

9 The packet will be sent to the H_B.

Note that for simplicity, in the above description, we assume the traffic flow exist in host-to-host level, not socket-to-socket. Nonetheless, our design and system are flexible and can support socket-level forwarding.

5.2.2 Dynamic Adjustment

If a new switch joins the system, a switch state change event will be sent to the controller. If the switches is a core switch, which means that the state change will not affect the current traffic immediately and we will leave the problems to the STF for better optimization. Thus, the controller will trigger the static algorithm to find tunnels. On the other hand, if the switch is an edge switch, and it's highly possible that there will be traffic requests on the switch. The same state change event will then be sent to the dynamic module to construct the temporary tunnels among the others switches. The dynamic tunnel finder will try to find tunnels without affecting the others in-used tunnels. However, the tunnel will still be further optimized by static algorithm periodically.

Chapter 6

Tunnel Table Construction

In this chapter, we will introduce the proposed tunnel finder algorithm. For constructing tunnels among the given network topology, we first design Static Tunnel Finder (STF) to find disjoint set on each switches pair. Then, the tunnel is picked up in shortest order from the disjoint set. To avoid overlong end-to-end tunnels, which leads to inefficiency, stretch factor [8] is used to constraint the maximum length.

6.1 Problem Formulation

Although shortest path routing can reduce the transmission time of propagation delay, however, we design our system to be more reliable and flexible. In this algorithm, our goal is to identify the largest number of mutually disjoint tunnels between each switch pairs, so that we can maximize the available bandwidth among each switch pairs.

In this problem we model the underlying network as a directed graph. Let N be the set of nodes in the network, which corresponds to switches in the network. Let P be the set of all the possible non-trival path without repeating node between each switch pair (Notice that the word path, which is a way from source to destination, is different from the **Path**, which compose of tunnels with QoS guarantee). Let L be the set of all the links in the network. For each $p \in P$, we also define $BottleNeck_p = min(c(l))$, where $c(l)$ stands for the capacity of each l, $1 \leq l \leq L$. Define $S_{n,p}$ and $D_{n,p}$, which are the source node and the destination mode in each P respectively. Define y_p, which act as a decision variable. We define the decision variable $B(p)$ for $1 \leq p \leq P$, which corresponds to the allocated bandwidth for each p in P, the range of $B(p)$ is $[0, max(c(l))], 1 \leq l \leq L$. We also define $m_{p,l}$ ($1 \leq p \leq P$, $1 \leq l \leq L$), where $m_{p,l} = 1$ if path p includes link l; $m_{p,l} = 0$ otherwise. Since we are not using shortest path as our routing scheme, we define k, which is the maximum length of selected tunnels between two nodes, to avoid overlong tunnel in the system. The k is determine by the stretch factor, which is the quotient of hop

Table 6.1: Symbol Table used in Tunnel Finder Problem

Symbol	DefinitionDefinition
N	Set of nodes in the network
P	Set of all the possible non-trival path
L,L_i	Set of all the links in the network
$BottleNeck_p$	The bottle capacity of path t
$c(l)$	capacity of link l
$S_{n,p}$	The source node of path p
$D_{n,p}$	The destination node of path p
$B(p)$	The allocated bandwidth for path p
$m_{p,l}$	Whether path p includes link l
k	Maximum length of selected tunnels

number of selected tunnel and the hop number of shortest path of the given switch pair.

$stretch\_factor = \frac{\sum_{l=1}^{L} m_{p,l}}{shortest\_path\_hop}, \forall p \in P$

Thus, the k can be written as: $k = stretch\_factor \times shortest\_path\_hop$. With the defined notations, we write the tunnel finder problem as:

$$\text{maximize} \sum_{p=1}^{P} B(p) \tag{6.1a}$$

$$\text{s.t.} B(p) m_{p,l} \leq \frac{c(l)}{(\sum_{p'=1}^{P} y_{p'} m_{p',l})}, \forall l \in L, \forall p \in P; \tag{6.1b}$$

$$B(p) \leq BottleNeck_p, \forall p \in P; \tag{6.1c}$$

$$\sum_{l=1}^{L} B(p) m_{t,l} \leq k, \forall p \in P \tag{6.1d}$$

The objective function in Eq. (6.1a) aims to find the maximum bandwidth of each paths. If the bandwidth of a path equals to zeros, which means that the path is not selected by the solver. The constraint in Eq. (6.1b), each link is shared by several paths ,and thus the path capacity is less than or equal to the minimum link share among the links used by the path. In Eq. (6.1c), we define the bottle neck of each path, p, by the minimal link capacity among all the links in the given topology. In Eq. (6.1d), we make sure the selected tunnels will never exceed the constraint k to gain better efficiency.

6.2 Static Tunnel Finder algorithm

To solve the disjoint set problem, we extend D. Torrieri's heuristic solution [50]. Algorithm 1 shows the pseudo code of modified fast-disjoint tunnel finder. The algorithm

Algorithm 1 Static Tunnel Finder (STF)

Src,**Dst** //source node and destination node
SP_hop $= shortest\_path(Src, Dst)$
k //Path limit which constrained by Eq. 6.1
Adj_matrix //adjacency matrix of the given topology
function DISJOINTFINDER(Src,Dst,Adj_matrix)
    Prefix_tunnels //List of sub-tunnel list from source node
    Suffix_tunnels //List of sub-tunnel list to destination node
    length //current target length
    disjoint_ans //Final ans which is the optimal set
    while **length** $\leq$ **k** **do**
        Break if neither **Prefix_tunnels** nor **Suffix_tunnels** are not able to increase
        //skip the checking until shortest path reach
        if **length** $\geq$ **SP_hop** **then** //Check interaction
            for each sub_tunnel_A in **Prefix_tunnels** **do**
                for each sub_tunnel_B in **Suffix_tunnels** **do**
                    if sub_tunnel_A[-1] == sub_tunnel_B[0] **then**
                        disjoint_ans.append$(sub\_tunnel_A + sub\_tunnel_B)$
                        Adj_matrix$(sub\_tunnel_A[-1], n) = 0, \forall n \in N$
                        Adj_matrix$(n, sub\_tunnel_B[0]) = 0, \forall n \in N$
        //Need to prepare $Prefix\_tunnels$ for next round
        if $ceil((\textbf{length}+1)/2) > (\textbf{length}+1)/2$ **then**
            for each sub_tunnel in **Prefix_tunnels** **do**
                if **Adj_matrix**$(sub\_tunnel[-1], n), \forall n \in N$ **then**
                    sub_tunnel.**append**(n)
        //Need to prepare $Suffix\_tunnels$ for next round
        if $floor((\textbf{length}+1)/2) == (\textbf{length}+1)/2$ **then**
            for each sub_tunnel in **Suffix_tunnels** **do**
                if **Adj_matrix**$(n, sub\_tunnel[-1]), \forall n \in N$ **then**
                    sub_tunnel.**prepend**(n)
    return(**disjoint_ans**)

first compute the shortest hop $\mathbf{SP\_hop}$ using breadth first search. The algorithm always find all the possible tunnels start from source node and tunnels go to the destination node (store the tunnels info in $\mathbf{Prefix\_tunnels}$ or $\mathbf{Suffix\_tunnels}$ respectively). In each loop, either $\mathbf{Prefix\_tunnels}$ or $\mathbf{Suffix\_tunnels}$ is increased by 1. If the last nodes in $\mathbf{Prefix\_tunnels}$ intersect with the first nodes of $\mathbf{Suffix\_tunnels}$, which means that there is a tunnel from source to destination, the two sub-tunnels will be merged and appended to the disjoint set. To avoid choosing the same node, the all the adjacency information about the interaction node will be wiped out. The process continue until there is no more node can be added to the $\mathbf{Prefix\_tunnels}$ or $\mathbf{Suffix\_tunnels}$, or length limit exceeds.

6.3 Analysis

The optimal disjoint path algorithm is similar to multi-commodity flow problem [13], which is a NP-complete problem. We add some constraints to solve this problem in polynomial time. However, the maximal length constraint will not affected our system, but brings efficiency to the network.

Algorithm 1 runs in polynomial time.

Proof. We first adopt BFS to find the shorest path hop, which generally takes $O(N + L)$, where $O(1) < O(L) < O(N^2)$ [48]. The main loop runs in $O(k)$ rounds. In each rounds, we may need to update the sub-tunnels list (line 21-29), which runs in $O(TN)$ and T is the numbers of sub-tunnels. For checking the intersection of each sub-tunnels (line 14-19), it takes $O(T^2N)$ to find the intersection. Therefore, the complexity of algorithm 1 can be expressed as $O(KT^2N)$. Since $0 < T < N$ and k is a constant value, $O(KT^2N) \cong O(N^3)$.

Chapter 7

Path Table Construction

On the other hands, we design another two algorithms to assign traffic. The first one is Static Path Assigner (SPA) which balance the load to minimize the link utilization. To handle new traffic requests and unforeseen events, we design an Dynamic Path Assigner (DPA) to determine the paths for the traffic in real time (see chap: 8).

7.1 Problem Formulation

We consider a network with S sites denoted as S_i ($1 \leq i \leq S$) connected by an underlying network. We model our system as a directed graph with S nodes and T edges, each vertex denotes a site in the network and each edge denotes selected tunnels between two sites. Each tunnel is written as T_i ($1 \leq i \leq T$) with σ_i and δ_i are the source and the destination of selected tunnel i. Note that these S sites are connected by T directed site-to-site tunnels, each tunnel in here are composited of one or several link(s).

There are L links between switches in the system, each link is written as L_i ($1 \leq i \leq L$), and $c(l)$ ($1 \leq l \leq L$) denotes the capacity of the link. Each traffic flow is defined as F_i ($1 \leq i \leq F$). θ_i and η_i denote the switches connected to the source and the destination of traffic flow i. b_i denotes the bandwidth requirement of traffic flow i.

In this problem, we consider the problem of (i) optimizing the utilization of tunnels between sites (ii) balancing the load among tunnels. That is, given a set of matching traffic flows with tunnels so that we can minimize the maximum link utilization. Noted that in this problem we consider the situation after the admission control, that is to say we only consider the case that all the requirement of the traffics can be fulfilled by the tunnels we have in the system in some way.

We define 0-1 decision variable $x_{t,f}$ for $1 \leq t \leq T$, $1 \leq f \leq F$, where $x_{t,f} = 1$ if tunnel t from S_i to S_j is on the path of this traffic flow; $x_{t,f} = 0$ otherwise. We define a 0-1 variable as $m_{t,l}$ ($1 \leq l \leq L$, $1 \leq t \leq T$), if tunnel t contains link l , $m_{t,l} = 1$; $m_{t,l} = 0$ otherwise. We also define another 0-1 variable as $w_{i,j,t}$ for $1 \leq i, j \leq S$, $1 \leq t \leq T$, where $w_{i,j,t} = 1$ if σ_t = i and δ_t = j; $w_{i,j,t} = 0$ otherwise. We define $S'(x)$ as S - $\{S_x\}$,

Table 7.1: Symbol used in Path Assigner Problem

Symbol	**DefinitionDefinition**
T_i	Set of selected non-trival tunnel with label i
S,S_i	Network sites
$m_{p,l}$	Whether path p includes link l
L,L_i	Set of all the links in the network
$c(l)$	capacity of link l
F_i	Traffic flow
σ_i	The source of selected tunnel i
δ_i	The destination of selected tunnel i
θ_i	Ingress switch of traffic flow i
η_i	Egress switch of traffic flow i
b_i	bandwidth requirement of traffic flow i
C_l	Current used bandwidth of link l
$x_{t,f}$	Whether tunnel t is on the Path of Traffic f
$m_{t,l}$	Whether selected t includes link l
$w_{i,j,t}$	Whether S_i and S_j are the source node and end node of tunnel t
$S'(x)$	The set consists of all the vertices in S other than x
R	Limitation of Binary Search

that is the set consists of all the vertices in S other than x. With the notations defined above, we mathematically formulate the traffic assigner problem as:

$$\text{minimize} \max_{1\le l\le L} \sum_{t=1}^{T}\sum_{f=1}^{F} x_{t,f} m_{t,l} b_f / c(l) \tag{7.1a}$$

$$\text{s.t.} \sum_{n\in S'(\theta_f)}\sum_{t\in T} w_{\theta_f,n,t} x_{t,f} - \sum_{n\in S'(\theta_f)}\sum_{t\in T} w_{n,\theta_f,t} x_{t,f} = 1, \forall f \in F; \tag{7.1b}$$

$$\sum_{n\in S'(e)}\sum_{t\in T} w_{e,n,t} x_{t,f} - \sum_{n\in S'(e)}\sum_{t\in T} w_{n,e,t} x_{t,f} = 0,$$

$$\forall e \in S'(\theta_f, \eta_f), \forall f \in F; \tag{7.1c}$$

$$\sum_{n\in S'(\eta_f)}\sum_{t\in T} w_{\eta_f,n,t} x_{t,f} - \sum_{n\in S'(\eta_f)}\sum_{t\in T} w_{n,\eta_f,t} x_{t,f} = -1, \forall f \in F; \tag{7.1d}$$

$$\sum_{t=1}^{T}\sum_{f=1}^{F} x_{t,f} m_{t,l} b_f \le c(l)\ \forall l \in L; \tag{7.1e}$$

$$x_{t,f} \in \{0,1\},\ 1 \le t \le T, 1 \le f \le F. \tag{7.1f}$$

$$m_{t,l} \in \{0,1\},\ 1 \le l \le L, 1 \le t \le T. \tag{7.1g}$$

$$w_{i,j,t} \in \{0,1\},\ 1 \le i,j \le S, 1 \le t \le T. \tag{7.1h}$$

The objective function in Eq. (7.1a) aims to minimize the maximal tunnel utilization in our system. Eq. (7.1b) ensures that the traffic flow never returns to the source of the traffic. Eq. (7.1c) ensures that none of the intermediate sites consume the traffic flow. Eq. (7.1d) ensures that the traffic flow enters the destination and never leaves it. The constraint in Eq. (7.1e) ensures that the traffic over a link does not exceed its capacity. Since our decision variable $x_{i,j,f}$ is a 0-1 decision variable, it can only take value either 0 or 1 according to Eq. (7.1f). In the formulation in Eq. (7.1), we only take the bandwidth as a QoS metric (Eq. (7.1e)), but administrators can add more QoS metrics, such as packet loss rate and latency.

7.2 Static Path Assigner Algorithm

To solve this problem, we develop a static algorithm called SPA (Static Path Assigner). Using the same objective function as in Sec. 7.1, our algorithm leverage constrained BFS, along with Binary Search. To maintain a balanced load among links in the system, we define a value α denoting the maximal allowed link utilization. A flow f can only flow through a tunnel t if every link l on that tunnel satisfies the constraint: $C_l + b_f \le c(l)\alpha,\ \forall l$, where C_l denotes the currently used bandwidth. The constrained BFS guarantees: $\sum_{t=1}^{T}\sum_{f=1}^{F} x_{t,f} m_{t,l} b_f \le c(l)\alpha,\ \forall l \in L$.

Our algorithm leverage binary search to find best α value based on the current available bandwidth. Algorithm 2 gives the pseudo code of our algorithm. We define a variable R as the limitation on the rounds of binary search. We also define L and U as the lower and upper bounds of binary search.

Algorithm 2 Static Traffic Assigner (SPA).

$Upper = 1.0$, $Lower = 0$ //The upper/lower bounds
$\mathbf{Final\_Assignment} = \phi$ //The final answer
Sort traffic flow $\mathbf{F}$ by bandwidth b_f in desc. order
$\alpha = \phi$ // Utilization
while $Upper - Lower < threshold$ **do**
    $\alpha' = (Upper + Lower)/2$
    $\mathbf{CurrentAssignment} = \phi$
    for each flow f in $\mathbf{F}$ **do**
        $path = \mathbf{ConstrainedBFS}(\theta_f, \eta_f, b_f, \alpha')$
        if $\mathbf{path} = \phi$ **then**
            $\mathbf{A}' \leftarrow \phi$
            Break
        else
            update available link bandwidth
            $\mathbf{CurrentAssignment}.append(\mathbf{path})$
    if $\mathbf{CurrentAssignment} = \phi$ **then**
        $Lower = \alpha'$
    else
        $Upper = \alpha'$
        $\alpha = \alpha'$
        $\mathbf{A} \leftarrow \mathbf{A}'$
if α is not defined, return no answer

7.3 Analysis

The SPA algorithm runs in polynomial time.

Proof. We first sort the flow in the system in line 3, which has a complexity of $O(F \log(F))$. For the binary search loop starting from line 4, it is executed for R times. The time complexity can be expressed as $O(R)$ where $O(1) \leq O(R) \leq O(\log(S))$ [48]. In each iteration it take turns to find a path using ConstrainedBFS for each flow $f \in F$. Each ConstrainedBFS in line 8 takes at most $O(S + L)$, where $O(1) < O(L) < O(S^2)$ [48]. Each update of link bandwidth in line 13 takes $O(L)$. Combined with the iteration mentioned above, these two complexity will be $O(FR(S + L))$ and $O(FRL)$ with $O(FR(S+L))$ being the dominant one. In the worest case, if the nodes is fully connected and the limitation is infinite, $O(FR(S+L))$ can be expressed as $O(F \log(S) S^2)$. This yield the lemma.

Chapter 8

Error Resilience and Network Dynamics

8.1 Switch Dynamic

To handle unexpected failure situations, we also design a Dynamic Tunnel Finder (DTF) algorithm to find an additional tunnels in real time. We want to find an alternative tunnel to reconstruct the affected tunnels under the condition that the impact of building new tunnels should be minimized. More precisely, our goal is to find a tunnel with lowest utilization so that the load balancing module can still utilize the tunnels.

Figure 8.1: Scenario of Dynamic Tunnel Finder.

Figure 8.1 gives an example of a failure event. The failed switch splits the tunnels into two sub-tunnels. Our algorithm aims to find a tunnel which connects to the two sub-tunnels. Node S and Node E is two hypothetical nodes which connect to all the switches among the two split tunnels respectively. To recover two switches set, we want to find a node-to-node shortest tunnel from node S to node E. Once we find the tunnel, we can create a temporal tunnels (which will further be optimized by STF) by the overlapping nodes of the tunnel and original sub-tunnels. To solve this problem, we adopt modified Dijkistra algorithm to find a tunnel with lowest utilization to avoid affecting QoS of other working tunnels.

8.2 Dynamic Tunnel Finder Algorithm

Algorithm 3 gives the pseudo code for solving the tunnel problem. Once the tunnel has been found, we can determine the best conjunction points to put a temporal tunnel. Rather than tagging a new label on the temporal tunnel, we will first try to build the tunnel with existed tunnels. The same algorithm is used to find tunnels when new edge switches connect to the network. In this case, instead of finding shortest path between two hypothetical nodes, we directly find tunnels

If the given algorithm fail to find a alternative tunnel or the tunnel increase the utilization dramatically, the static tunnel finder will be triggered immediately to recomputed the tunnels.

Algorithm 3 Dynamic Tunnel Finder (DTF)

1: $\mathbf{M}$ //the map of the given topology
2: $\mathbf{S}$ //set of start nodes
3: $\mathbf{E}$ //set of end nodes
4: $\mathbf{U}$ //utilization of each links as edge weight
5: **function** DISJOINTFINDER(S,E,U)
6: $\mathbf{s}$ //A node which connect to all the nodes in S
7: $\mathbf{e}$ //A node which connect to all the nodes in E
8: **for each** v in M **do**
9: $\mathbf{Util}[\mathbf{v}] \leftarrow \infty$
10: $\mathbf{Prev}[\mathbf{v}] \leftarrow \phi$
11: $\mathbf{queue}(\mathbf{v})$
12: $\mathbf{Util}[\mathbf{s}] = 0$
13: **while** $queue \neq \phi$ **do**
14: $n \leftarrow$ node with lowest utilization
15: $\mathbf{dequeue}(\mathbf{n})$
16: **for each** neighbor v of n **do**
17: **if** lower utilization is found **then**
18: $\mathbf{Util}[\mathbf{v}] = \mathbf{Util}[\mathbf{v}] + \mathbf{utilization}(\mathbf{n}, \mathbf{v})$
19: $\mathbf{Prev}[\mathbf{v}] = n$
20: **if** n is e **then**
21: $path = \mathbf{convertToPath}(\mathbf{Prev})$
22: **return** $path$
23: **break**

Lemma 1. *DTF will assign the traffic in polynomial time.*

Proof. In line 9, the point to point shortest path will have the time complexity $O(NP)$. Notice that we adopt point-to-point algorithm, the run time is expected to lower than

shortest path algorithm since the complexity of the both methods are the sane, Thus, combining with loop, the time complexity will be dominated by $O(TNP)$.

8.3 Path Dynamic

To handle the new flow requests and the nodes failure, we develop a dynamic algorithm to assign the traffic flow in polynomial time. Before we introduce our solution, we first discuss all the possible events: i) A new traffic request is arrived at the ingress switch. ii) Switch fails unexpectedly. If a new traffic request trigger the proactive switch module, controller will first check it's traffic table and response with the assigned label directly. If the controller doesn't have the record of the request, DPA will be trigger to assign tunnels.

For failure events, if the failed switch is a edge switch, we will ignore the flows since the the flows can not be recovered by controller. Thus, we simply remove these flows from our system. On the other hand, if the failed switch is central switch. Controller will first recover the traffic using backup tunnels. Since we choose tunnels from disjoint set, the traffic usually can be recovered by backup tunnels. If the backup tunnels failed to support the affected flows, dynamic traffic assigner will allocate those flows by calculating an alternative path. In this problem, our goal is also trying to find a possible path in real time. Our algorithm will allocate the flows according to its bandwidth requirement, the demanding flow will be first assigned. To perform path selecting in real time, we adopt BFS to find a path among possible tunnels.

8.4 Dynamic Path Assigner algorithm

Algorithm 4 Dynamic Path Assigner (DPA)

```
1:  F //set of flows need to be assign
2:  Sort traffic flow F by bandwidth b_f in desc. order
3:  while F is not empty do
4:      A ← φ //The final answer
5:      A = ConstrainedBFS(θ_f, η_f, b_f)
6:      if A = φ then
7:          //Needs to reallocate traffic
8:          TriggerSPA();
9:      else
10:         // Update utilization
11:         UpdateFlow(); //OpenFlow commands
```

Lemma 2. *DPA will assign the traffic in polynomial time.*

Proof. We first sort the flow in line 3,which has a complexity of $O(F \log(F))$. In line 7, ConstrainedBFS takes at most $O((S+L)L)$. Thus, the overall complexity will be dominated by $O(F(S+L)L)$.

Chapter 9

Implementation

9.1 RYU Controller

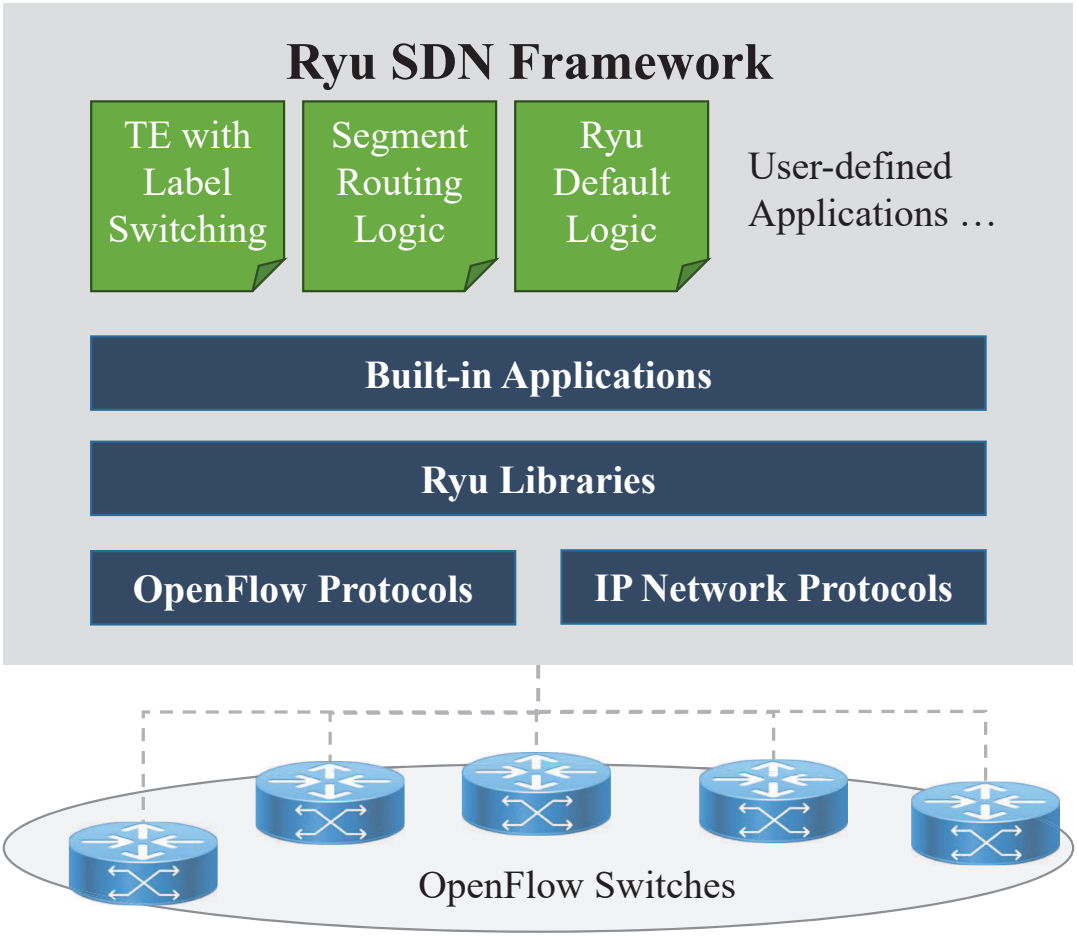


Figure 9.1: Ryu Framework

The proposed routing controller is implemented in SDN controller on top of Ryu [41].

Ryu is an SDN framework written in Python. In Ryu, control logics are designed as Ryu applications. Ryu provides the component-based framework with well-defined API, and developers are able to combine/modify the existing components or add new components to build up their own applications. Ryu supports various protocols, both SDN protocols and IP network protocols for managing network infrastructures includes OpenFlow, Netconf and OF-config. Fig. 9.1 shows the overview of Ryu SDN framework. There are some built-in applications and libraries in Ryu such as tenant isolation, L2 switch, OF REST, topology discovery, firewall, etc. For OpenFlow protocols, Ryu supports fully 1.0 to 1.5 and Nicira Extensions.

9.2 Mininet Emulator

Mininet [30] is a network emulator system, and it creates network nodes and links inside a single Linux Kernel. Mininet virtualizes the network topology into a system, which has the real host (you can control host via ssh), switches with real network interfaces, network link with configurable parameters. Inside the emulator, packets are processed with the same behavior as the real world system. To achieve the precise measurement requirement of the proposed system, we have patched mininet to support multiple interfaces between two nodes.

9.3 Algorithms and Utilizations

We have implemented a proof-of-concept testbed that uses mininet 2.1.0 and Ryu 3.17. The Ryu controller uses OpenFlow 1.3 protocols which fully support MPLS actions, to communicate with the underlying OpenVSwitches 2.4.0 [35]. To verify our the feasibility of proposed system, a few Ryu applications, and network tools are implemented. In each control applications for Ryu, we implement several handler functions defined by Ryu framework, including (i) switch connection (`EventOFPStateChange`) (ii) switch feature reply (`EventOFPSwitchFeature`), and (iii) packet-in hander (`EventOFPPacketIn`), etc.

9.3.1 Traffic Engineering with Label switching

In the proposed Proactive Switch application, we implement all the 4 algorithms. When `EventOFPStateChange` event send to the controller, the controller will proactively install all the path information which computed by STF inside each switch. We assume the required bandwidth of each traffic requests is known by the controller. Thus, the

SPA will allocate the tunnel resources for each edge switches pair into a table for better resource distribution. When the packet_in events are sent to the controller, the controller will first check the above traffic table inside the controller and program the edge switches. If the table miss the request, DPA will be activated and assign a possible path to the request.

9.3.2 Segment Routing Control Logic

We also implement the segment routing from Cisco Pathman-SR [36]. The pathman-sr is an application for another SDN controller, OpenDaylight [34], and we transplant the segment routing logic from OpenDaylight to Ryu. We implement the Interior Gateway Protocol (IGP) for collecting path information and adopting shortest path algorithm for MPLS tunnel creation. To compare our system with segment routing, we also adopt MPLS label for indicating each network segments and proactively install these segment information inside all the networks nodes as we did in our proposed system. For comparing the traffic engineering features, different from the pathman-sr which leaves the path selection to the network administrators, we design a feasible path assigned logic to select the segments for each traffic requests. The network administrator always chooses a path with the lowest utilization, and the total length of each assigned path will also be constraint by the stretch factor for fair and precise comparison.

9.3.3 Ryu Default Control Logic

To demonstrate the ability of proposed system, we adopt RYU Default Controller as one of our baselines. The Ryu default applications run the spanning tree protocol to prevent cycle and response the packet_in event with corresponding routing results. Different from TEL and SRI, the routing decisions are determined without considering traffic engineering when the traffic reaches the switches inside the network.

9.3.4 Network Monitor

To monitor our network, we have implemented a network monitor for mininet and provide REST API to RYU. The monitor records the state of each switch created by mininet, and the information includes flow tables and link utilization. We also record the packet information with tcpdump for analysis. To perform traffic engineering in real time, the monitor is able to use REST API to communicate with Ryu controller, so that the controller is able to make load balancing decision in real time. However, in this work, the network monitor are designed for simulated switches. To perform load balancing in runtime, a dedicated network monitor is required.

Chapter 10

Evaluations

10.1 Setup

We conduct our experiments on modified Mininet 2.1.0p and Open vSwitch 2.5.1. We build our control applications (see 9.3) on Ryu 4.11, and we record the statistic using our network monitor.

10.1.1 Topology Generator

Fig. 10.1 is an example of the topology using in our experiments. We use topology generator BRITE [29] to generate the topologies for our experiments. BRITE generates the network topologies at router- and AS-level, reflects the interconnectivity characteristics of the real world internetwork. We use the top-down model in BRITE to generate hierarchical topology, and the latency between switches is proportional to the distance. We add 16 ms delay as the mean value of the latency between all the switches, and 20 ms as a constant value of the latency between controller and switches. For the capacity of the links, we set the bandwidth to be a constant equal to 1 Gbps.

To ensure the connection between each node, we also add links between each isolated nodes in the network. The host is randomly connected to the edge switches in each the topologies. To demonstrate our system in different scales, we generate the topology from 8 nodes to 32 nodes. For each size, we also randomly generates five different topologies, because the results not only affected by sizes and also the connection complexity of each node. We defined our topology as $Topo_i^n$, n is the total switches nodes in the topology, i is the topology index. The detail information can be found in Table 10.1.

Table 10.1: Topology Information

Topo	Edge sw.	Core sw.	Links	Host
$Topo_1^8$	4	4	25	16
$Topo_2^8$	4	4	25	16
$Topo_3^8$	4	4	25	16
$Topo_4^8$	4	4	26	16
$Topo_5^8$	4	4	25	16
$Topo_1^{16}$	7	9	54	28
$Topo_2^{16}$	7	9	55	28
$Topo_3^{16}$	7	9	55	28
$Topo_4^{16}$	7	9	55	28
$Topo_5^{16}$	7	9	55	28
$Topo_1^{24}$	9	15	86	36
$Topo_2^{24}$	9	15	86	36
$Topo_3^{24}$	9	15	85	36
$Topo_4^{24}$	9	15	87	36
$Topo_5^{24}$	9	15	84	36
$Topo_1^{32}$	12	20	108	48
$Topo_2^{32}$	12	20	105	48
$Topo_3^{32}$	12	20	105	48
$Topo_4^{32}$	12	20	104	48
$Topo_5^{32}$	12	20	101	48

Table 10.2: Bandwidth Range of each topologies

Topo	Min. Bandwidth	Max. Bandwidth	Avg. Request/sec	Bandwidth (Mbps/#req.)
$Topo_i^8$	75 Mbps	100 Mbps	22.33	53.56
$Topo_i^{16}$	10 Mbps	100 Mbps	27.33	103.49
$Topo_i^{24}$	40 Mbps	100 Mbps	50.33	73.70
$Topo_i^{32}$	10 Mbps	100 Mbps	67.00	64.77

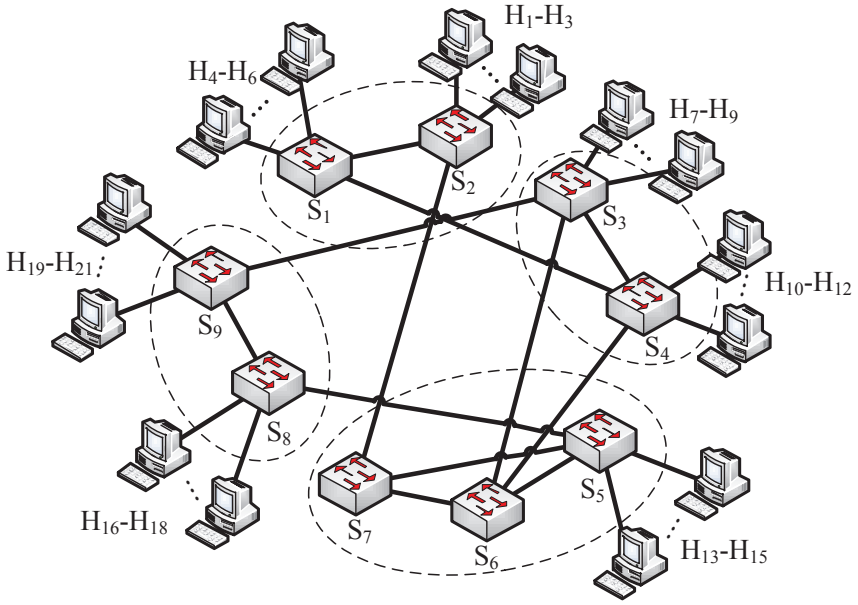


Figure 10.1: An example of simplified topology generated with BRITE.

10.1.2 Traffic Generator

We also implemented a traffic generator, which generates traffic flows with the arrival rate of Poisson distribution with λ_A minutes. The duration of each generated traffic flow follows the Poisson distribution with λ_D minutes. The destination of each traffic flow is randomly selected from all the hosts in other ASes that follows the uniform distribution. In table 10.2, we fix random seeds so that all the configurations (total bandwidth, the number of request and the source and destination hosts) are the same in the same size.

The bandwidth requirement of each traffic flow is selected between $10Mbps$ and $b_{\max}$ $100Mbps$ that follows the uniform distribution. For comparing our system with others, we use different bandwidth configurations for each topology size. The configuration is shown in table 10.2. We then use Iperf [19] to generate the real traffic in Mininet.

10.2 Scenario and Metrics

With the control applications and configuration mentioned above, we conduct our experiments on a linux machine with Intel i7-4790 CPU and 16 Gb ram. The duration of our experiment is set to 3 minutes. We run each experiment for 10 rounds and report the aggregate results. We consider three controlling logics and shows the results respectively:

- **TEL** Our proposed Traffic Engineering system with Label switching.
- **SRI** Segment Routing with IGP.

- **RSR** Ryu Spanning tree Routing.

In TEL and SRI, we also set the same stretch factor to avoid overlong paths, and the tunnel information is proactively installed inside the network. To perform error resilience in the proposed TEL, we find at most two possible tunnels from the disjoint set ($k = 2$) for each switch pairs.

We consider the following performance metrics to validate our system:

- **Link utilization.** The traffic load of links in the system.
- **Flow initialization delay.** The time it takes for the first packet of a traffic flow from source to destination.
- **Running time.** The runtime of our proposed algorithms.

10.3 Results

10.3.1 Link Utilization

Fig. 10.2 shows the link distributions in 24 nodes topologies. Two observations can be seen in the results: 1) Since our proposed TEL balances the traffic by overloading the traffic to lower utilization links, TEL use more links compare to SRI and RSR. 2) There are a few links which utilization exceeds 50% compare to the other two control logics which consume all the capacity of the links. Fig. 10.3 also indicates the maximum utilization along with the time, TEL achieves the lowest maximum utilization since TEL always tries to optimal the link utilization. The bounces utilization in every minute is due to the new traffic requests. The controller will inject the control messages after each packet_in requests, however, this issues can be minimized by separating the control signals to a dedicated line.

However, it's possible that more links are used by TEL compared to SRI and RSR since TEL uses the non-shortest paths to determine the routing, and offload the traffic to idle links. Fig. 10.4(a) shows that TEL reduce the max link utilization with 55% compared to SRI, and 54.9% compared to RSR. In fig. 10.4(b), The average link utilization is increased by 32% compared to SRI, and compared to RSR 9.5% respectively. However, fig. 10.4(d) shows that TEL balances the links efficiently, and 90% of the links have the utilization lower than 50%. The congestion can be prevented by TEL which gains better using experience on the client sides. We also show the comparison under light traffic in fig. 10.4(c), TEL also achieves the lowest utilization compare the others control logics.

Our algorithms outperforms SRI, RSR in terms of load balancing.

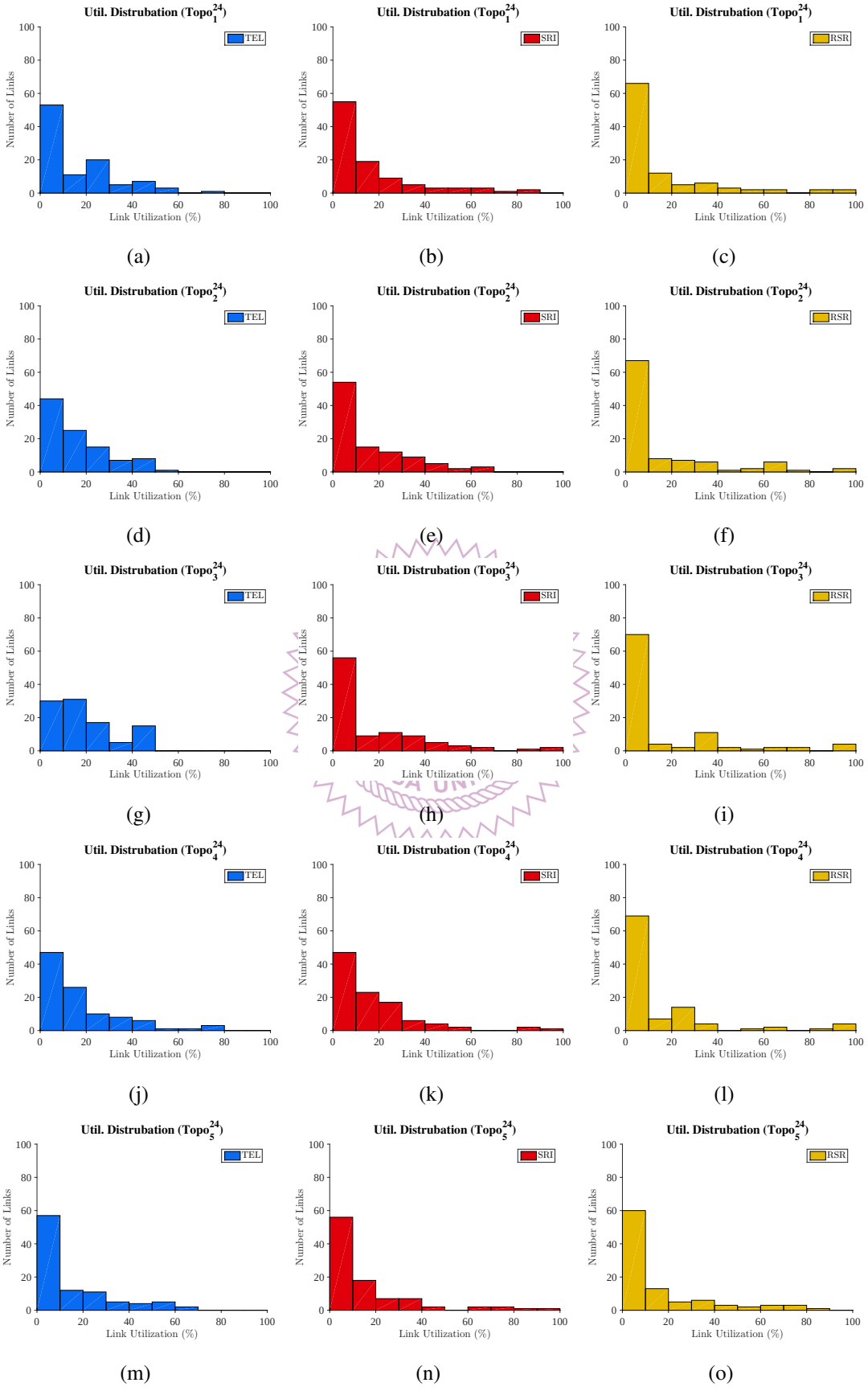


Figure 10.2: Utilization distribution of T_i^{24}

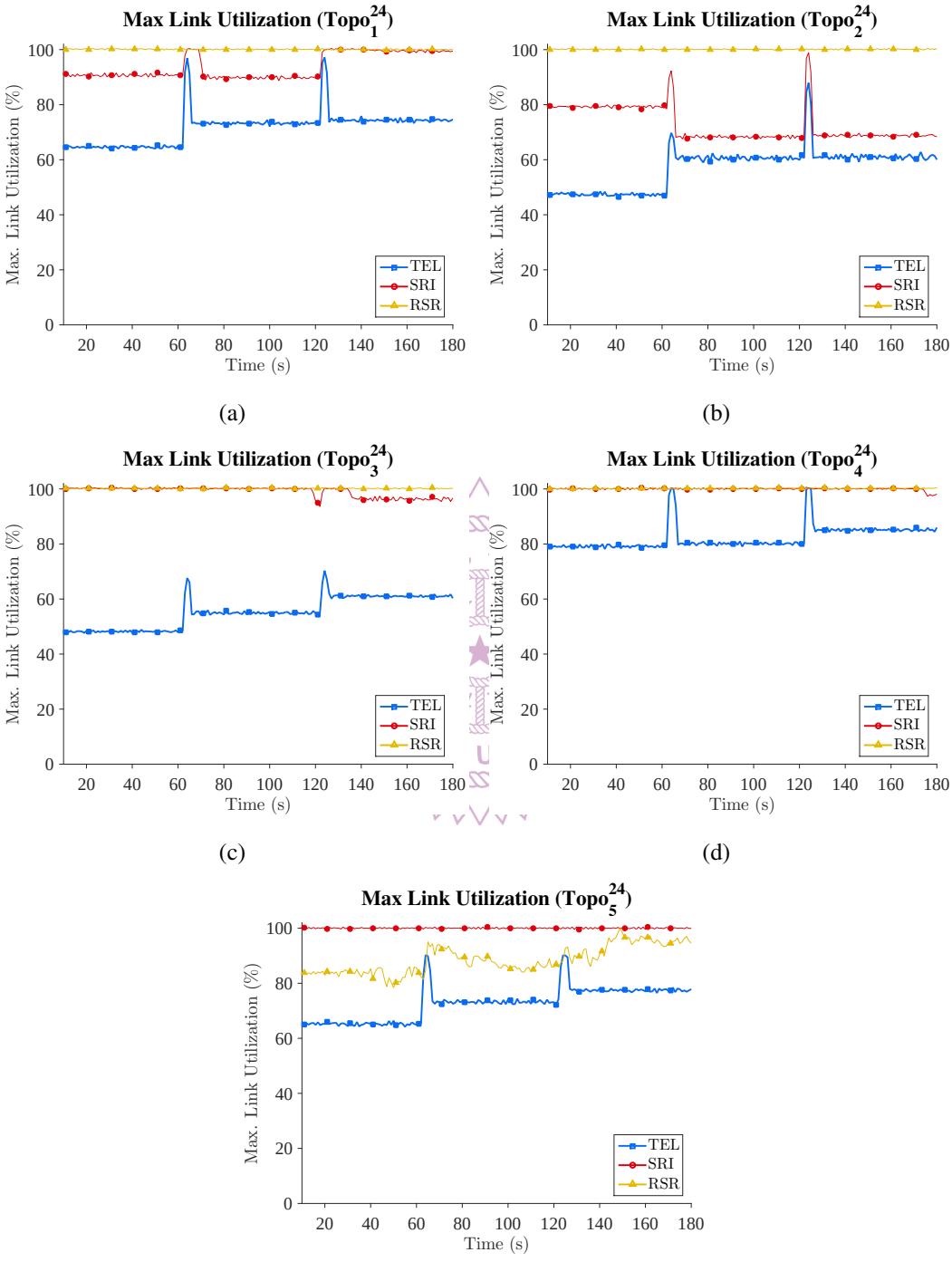


Figure 10.3: Max Utilization of T_i^{24}

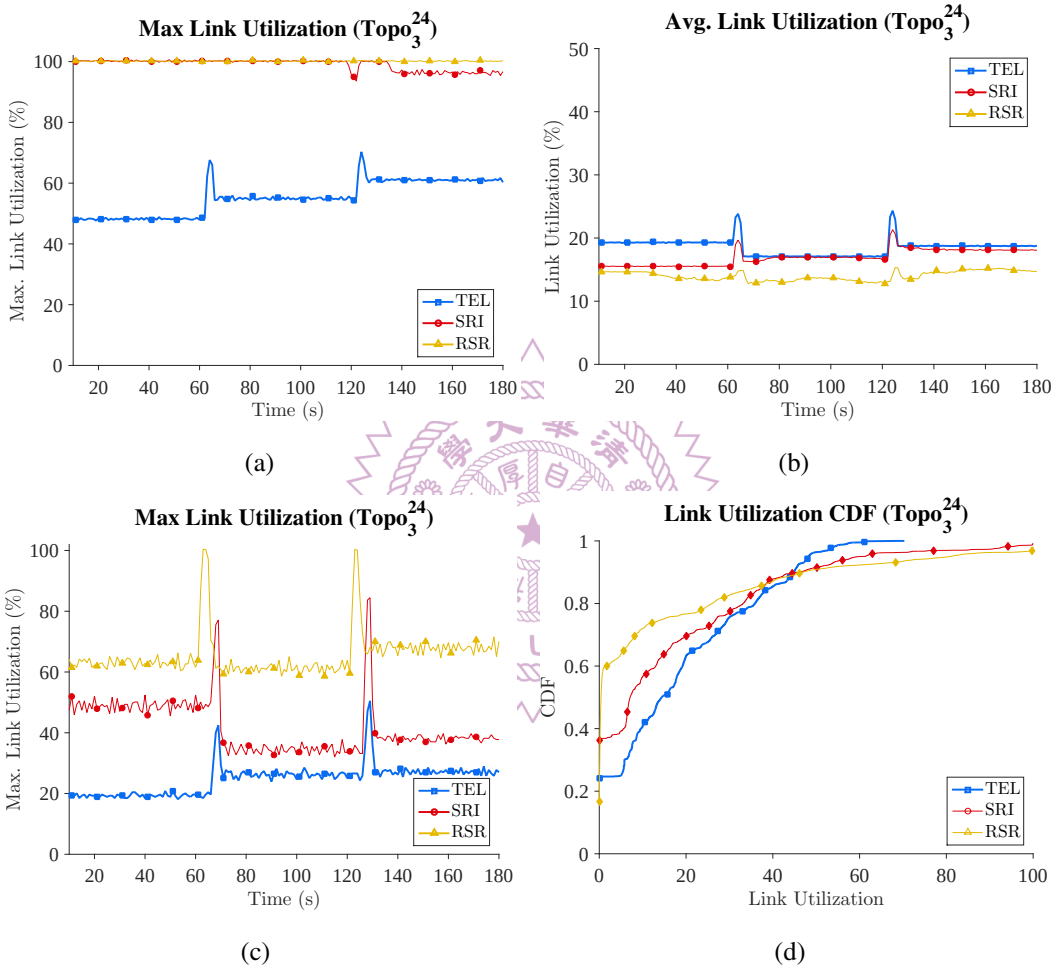


Figure 10.4: Link Usage in $Topo_3^{24}$ (a) Max utilization along time (Heavy traffic), (b) Avg. utilization along time (Heavy traffic), (c) Max utilization along time (Light traffic), (d) Utilization CDF (Heavy traffic)

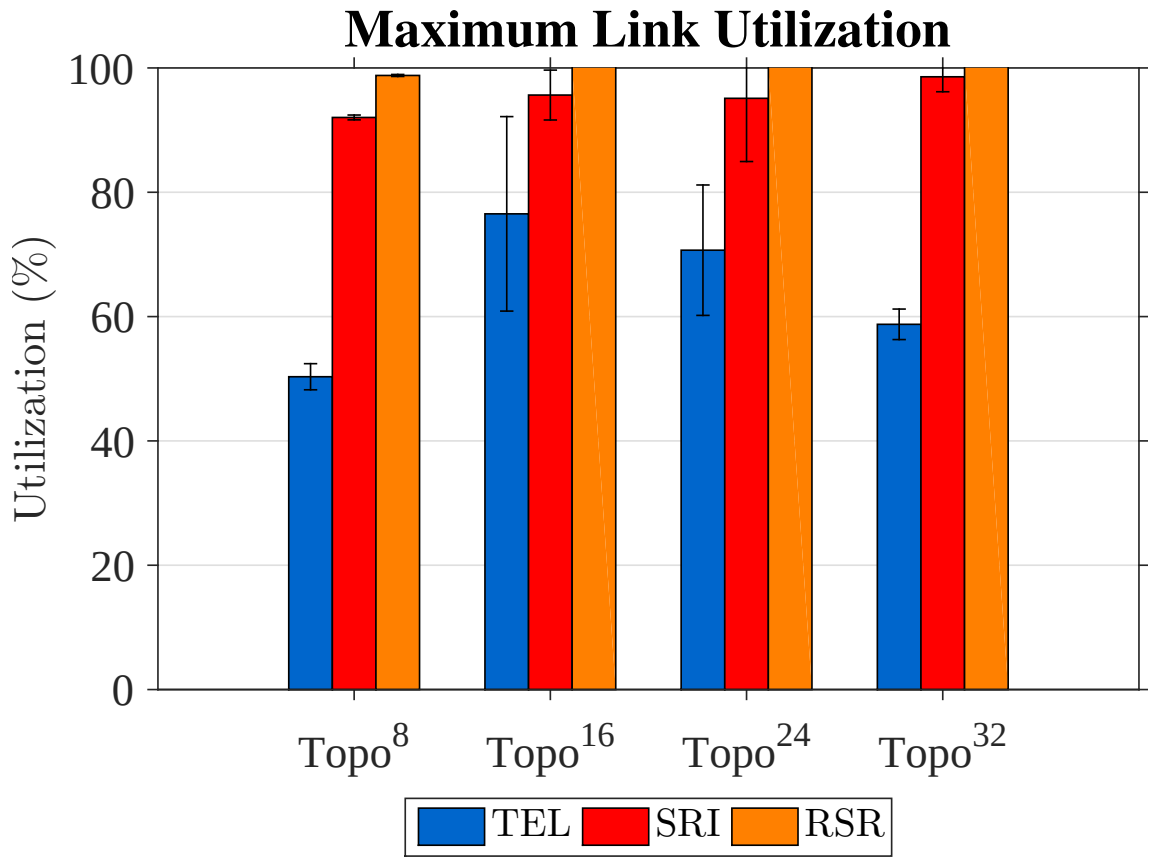


Figure 10.5: Max Link Utilization.

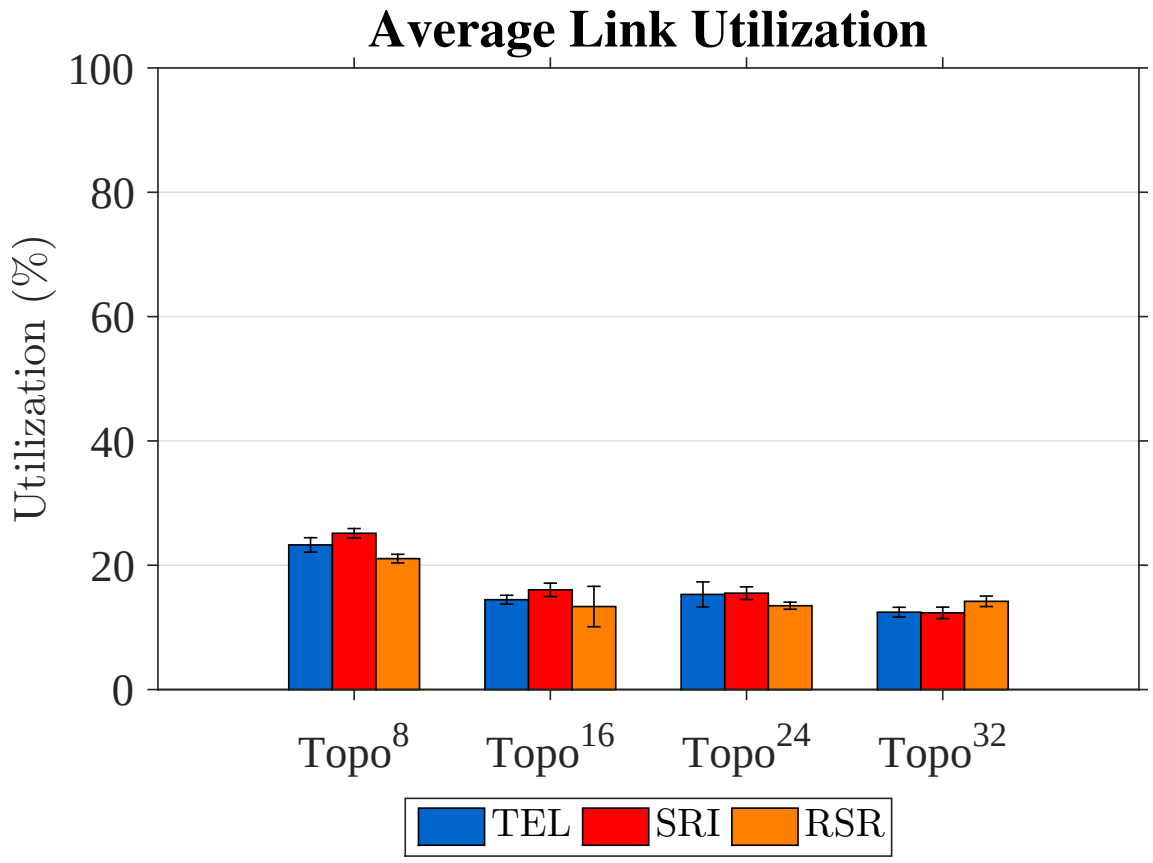


Figure 10.6: Average Link Utilization.

We conduct the experiments on different size of topologies, fig. 10.5 shows that TEL is able to reduce the utilization in different topologies. TEL always achieves the lowest maximum link utilization. On the other hand, in fig 10.6, the increased average utilization is negligible since the average link utilizations of TEL are close to the other solutions. Therefore, we can conclude that TEL is bandwidth-aware since fewer links are neither useless nor congestion as shown in the results. TEL leverages the proposed load balancing algorithms to optimal the network resources distribution to achieve load balancing. The max utilizations are reduced by TEL which decreases the possibility of network congestion.

10.3.2 Delay

We plot the CDF of normalized initial delay of T_i^{24}, which is the average delay time per hop in Fig. 10.7. TEL are able to transmit the packet within 100 ms. However, we find that around 20% of the requests do not receive the first packets within 100 ms if we using SRI, and around 50% if we using RSR. Compare to SRI which also use pre-built tunnels to minimize the delay, TEL achieves better results since SRI takes more time on segments selection and TEL adopt load balancing algorithm to allocate the traffic which avoids the congestion. For SRI and RSR control logics, some requests were queued in the switches, which leads to significantly higher initial delay: up to 2000 ms.

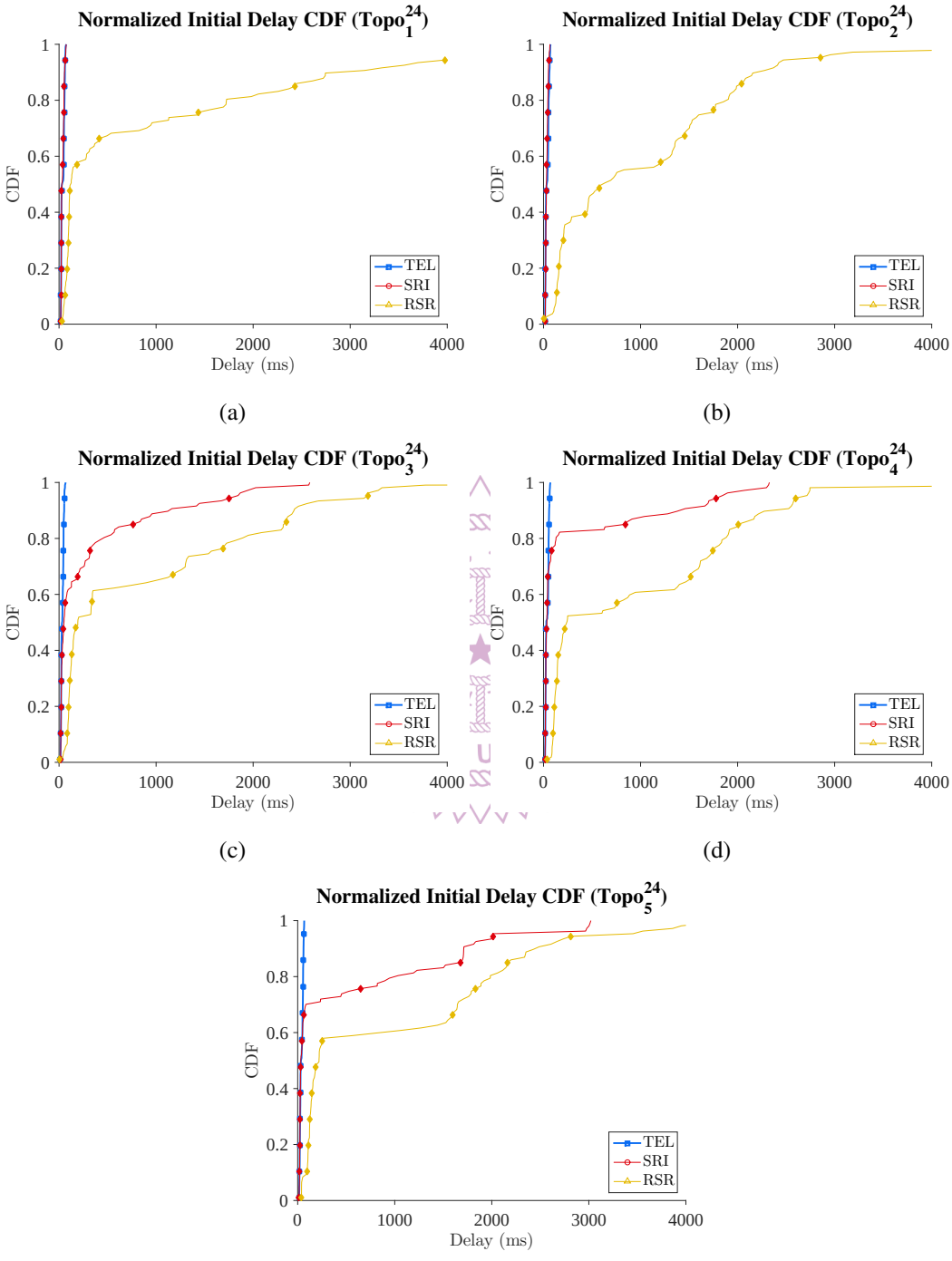
Shorter initialization delay. We give the average number of initial delay of all the topologies in Fig. 10.8 and the normalized delay in 10.9. TEL achieves the lowest initial delay compared to SRI and SRS. Our solution reduces the initial delay since we setup all pre-build tunnels in the system and our algorithms also avoid congestion. In average, 93.22% of initial delay is reduced by TEL compared to RSR. Since SRI also use the pre-defined paths to route packets, 39.02% of initial delay is reduced.

The initialization delay is increased with the number of hops along the traffic path. However, in fig. 10.8, we observe that the delay of $Topo^{32}$ is shorter than $Topo^{24}$ in both SRI and RSR. The reason may due to that more traffic is either lost or congestion in the network. Fig. 10.10 gives the CDF comparison of T_4^{24} and T_4^{32}, we can see that more links are overloading in T_4^{24} compare to the links in T_4^{32} (9% of links are overloading in T_4^{24}, 3-5% in others). The same issue doesn't appear in TEL since TEL always avoid congestion in networks.

10.3.3 Running Time and Flow Entries

Running Time

We show the running time of tunnel finder for TEL and SRI in fig. 10.11. The run-



(a) (b) (c) (d) (e)

Figure 10.7: Initialization Delay CDF of T_i^{24}

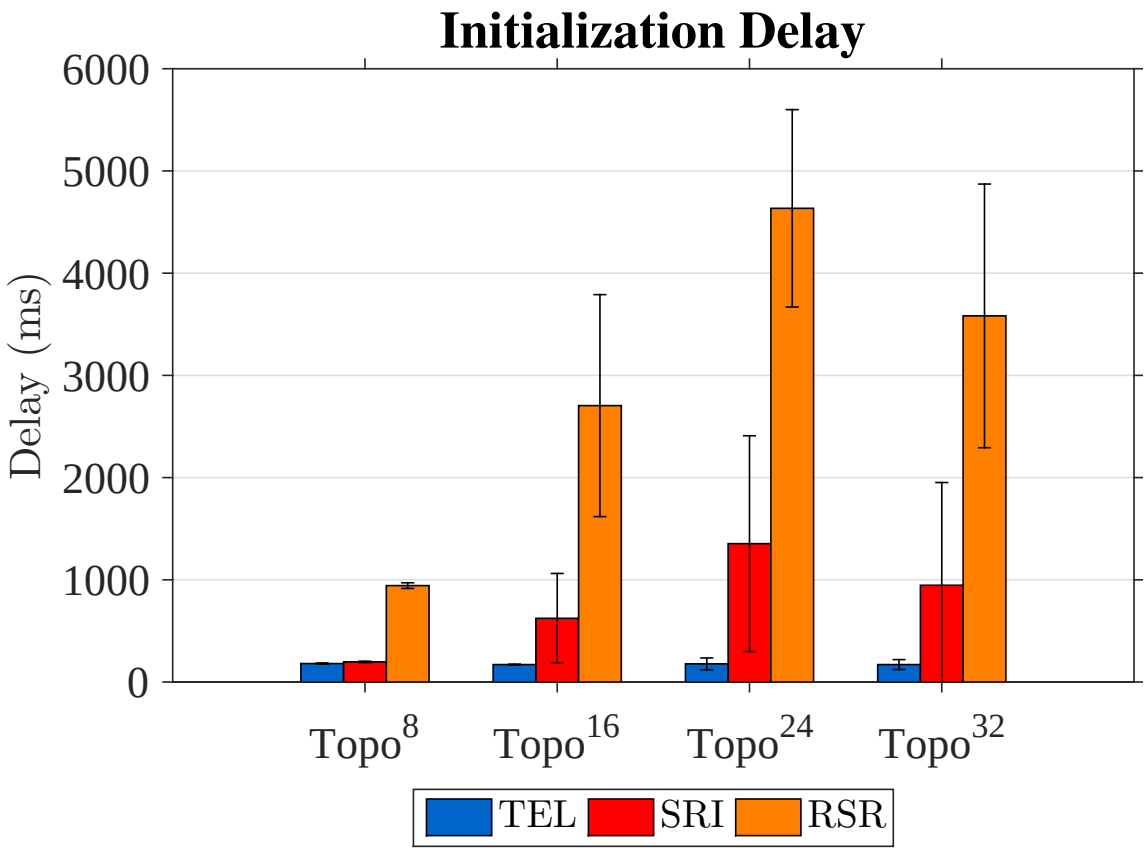


Figure 10.8: First Packet Initialization Delay

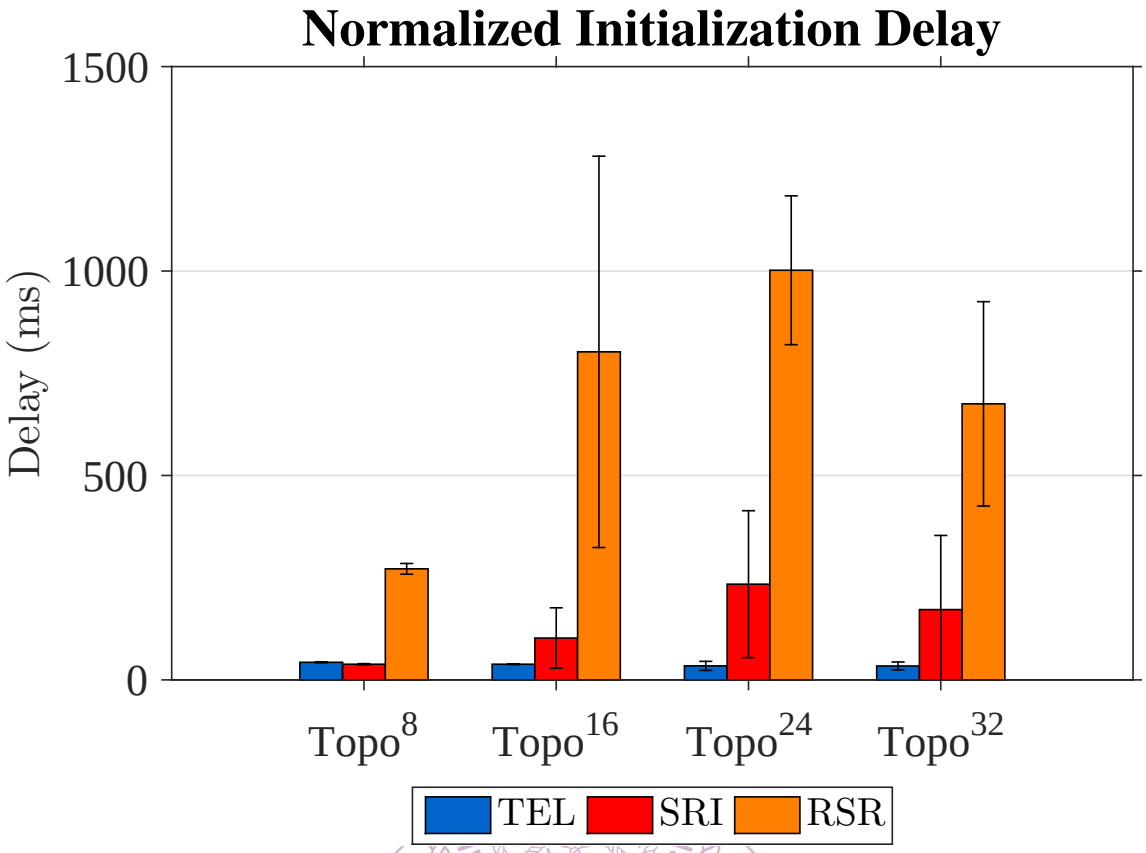
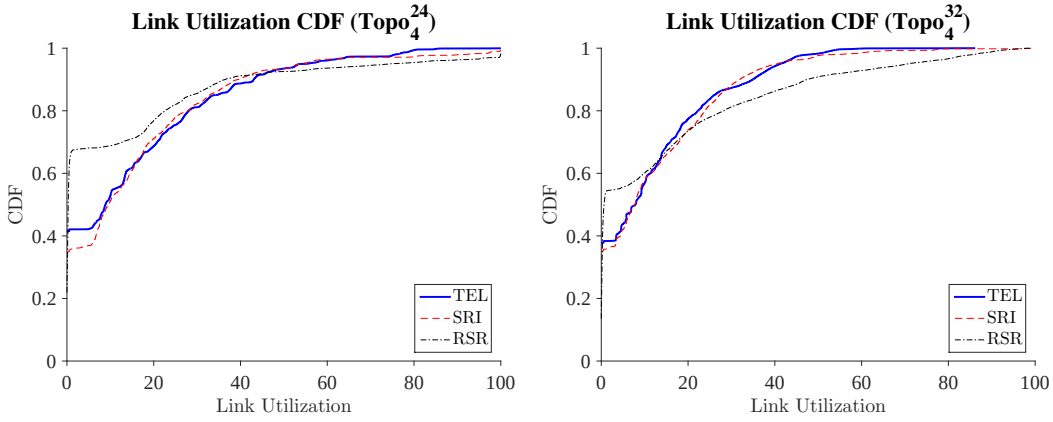


Figure 10.9: Normalized First Packet Delay



(a) (b)

Figure 10.10: Link utilization CDF of T_4^{24} and T_4^{32}

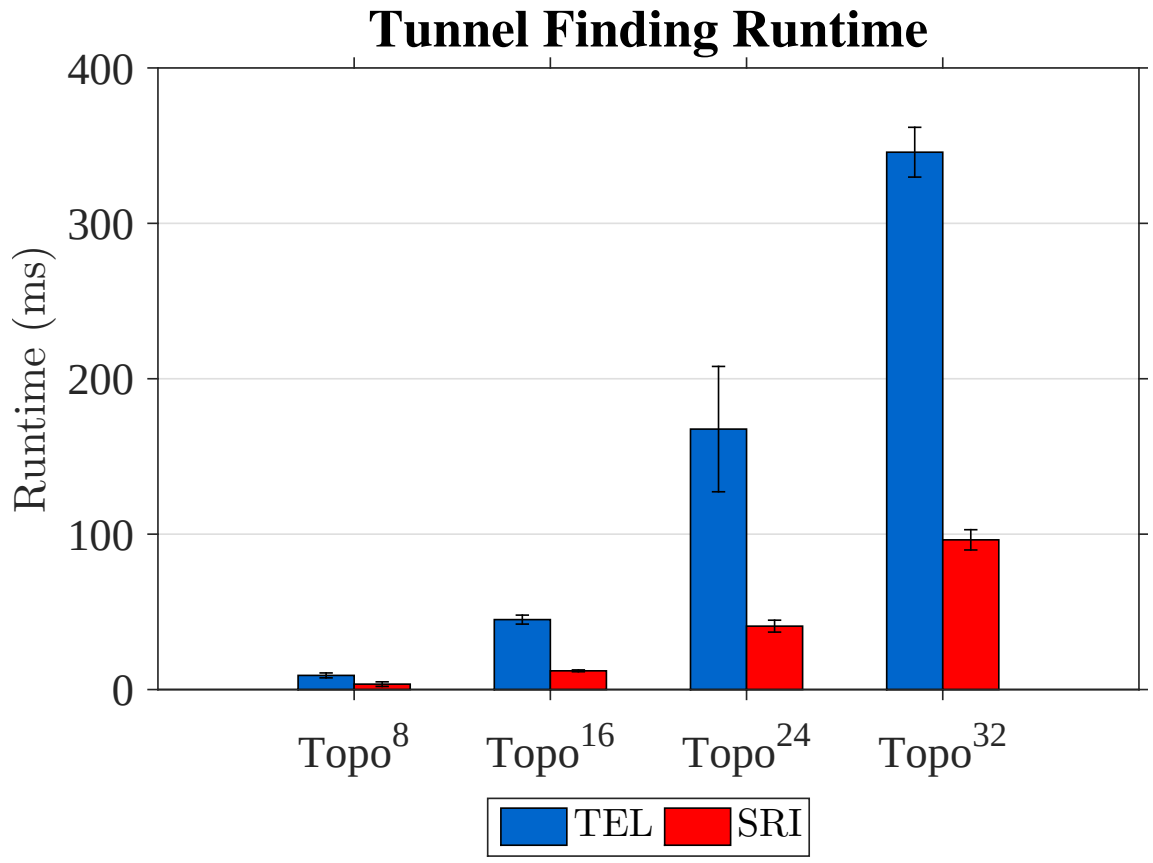


Figure 10.11: Runtime of Tunnel Finder Algorithm.

ning time is directly proportional to the size of topologies. SRI has lower running time than TEL since SRI used shortest path approach and didn't take utilization into consideration while building the tunnels. However, compared to the optimal solution solver (CPLEX [9]) which takes 7 hours to solve the 16 nodes topology, our proposed heuristic algorithm is still able to scale up.

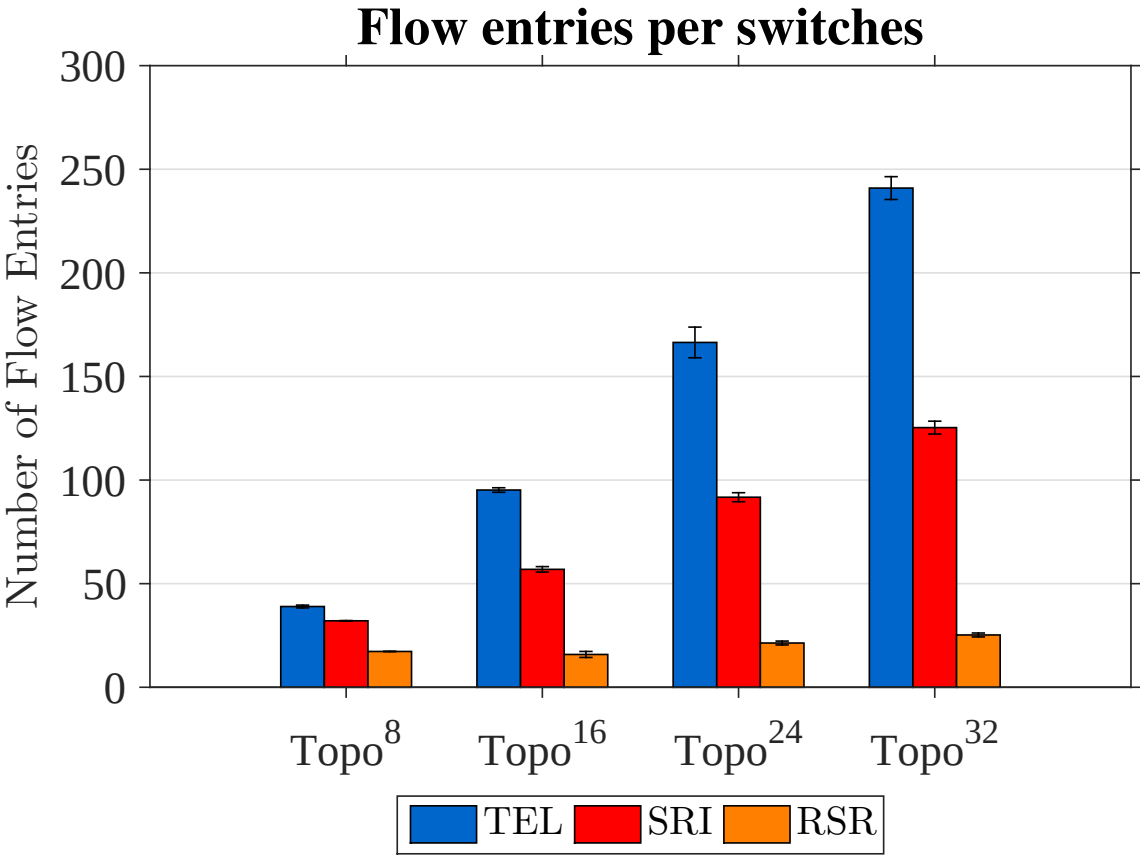


Figure 10.12: Number of flow entries per switches.

Flow entries

Fig. 10.12 gives the average entries numbers of three different approaches. Since both TEL and SRI are the proactive approach, the flow entries for each path are installed in each switch before any traffic. Thus, both TEL and SRI used more entries than RSR. TEL consumes most flow entries on the backup path which aims for providing more rapidly error recovery features. However, the numbers of the backup paths can be fully adjusted by the network administrator.

The number of flows is increasing along with the nodes number of topologies since we have to compute paths between any of two switches pair. In fig. 10.13 shows the activated flow entries which used to forward the packets in each experiment. The number of activated number is mainly dominated by the traffic request number and it is irrelevant

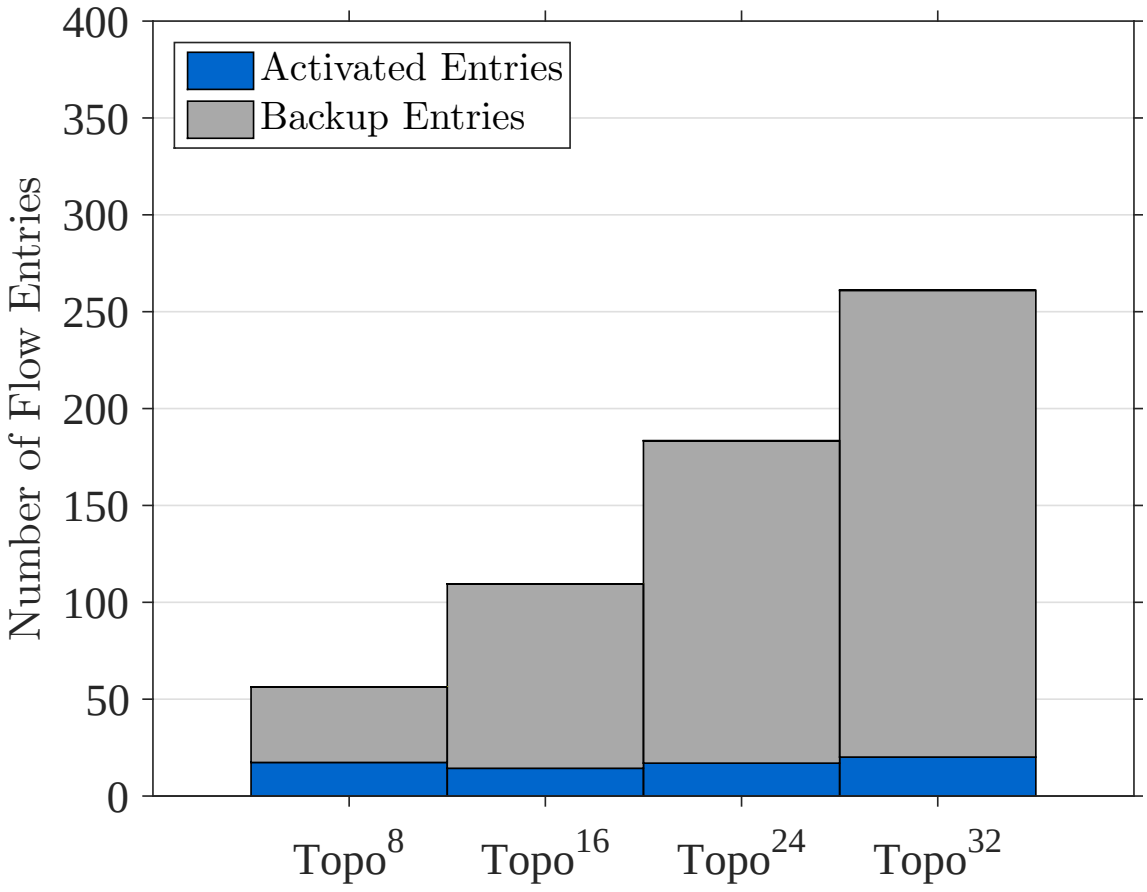


Figure 10.13: Number of activated flow entries per switches.

to the sizes of topologies. However, we can still observe the usage of flow entries is low. Therefore, the proactive path can be wisely decreased, and how to minimize the path without affecting the routing performance is in our future works.

10.3.4 Error Resilience

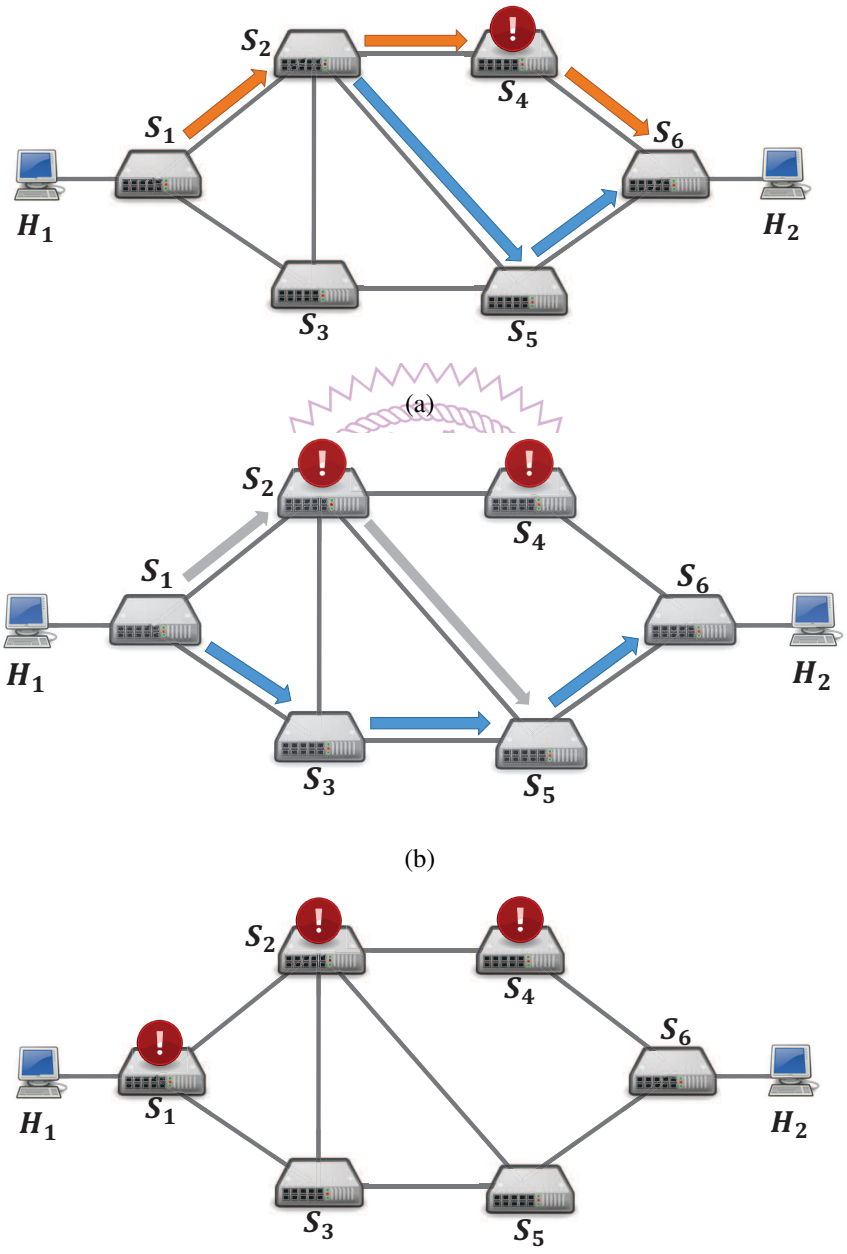


Figure 10.14: Scenarios of Error resilience.

To demonstrate the error resilience feature, we use the topologies in fig. 10.14 to demonstrate the TEL handling three unexpected failures. We assume there are only two

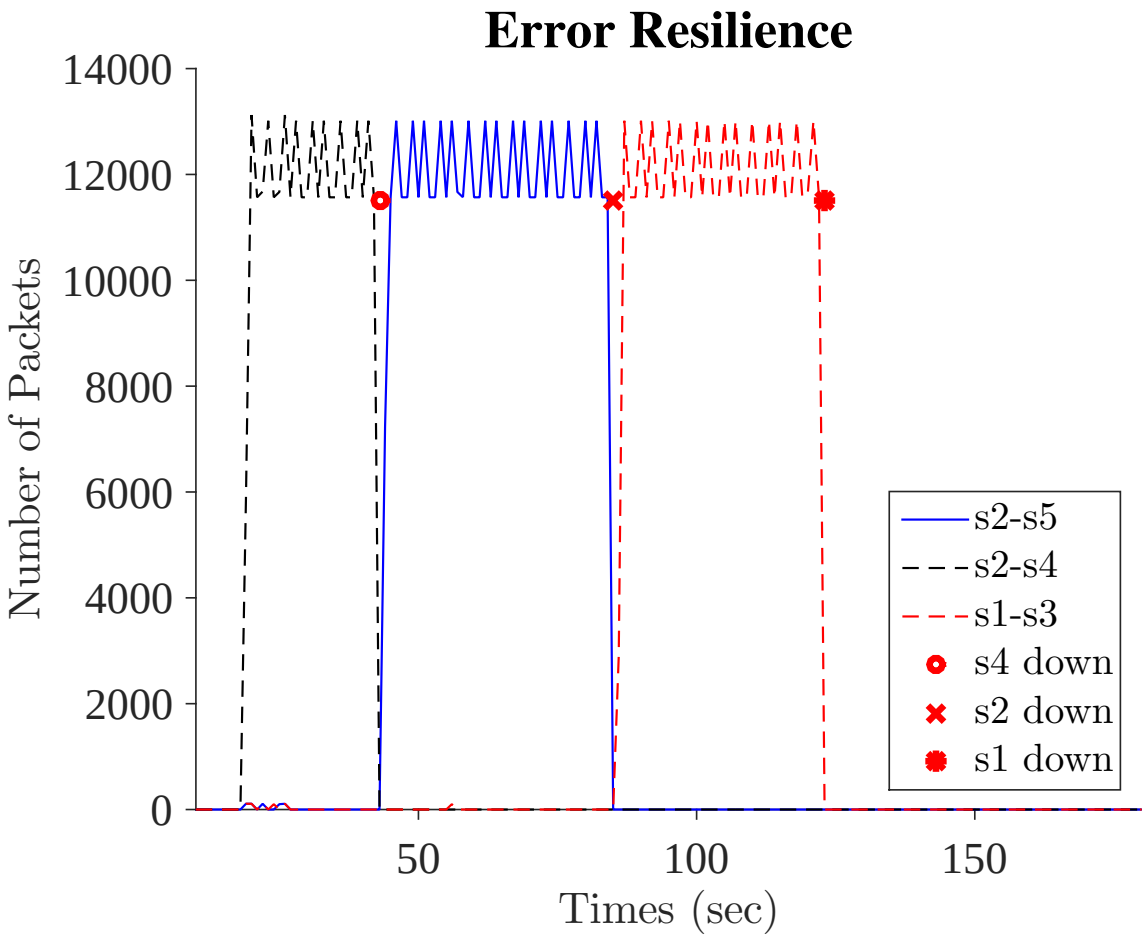
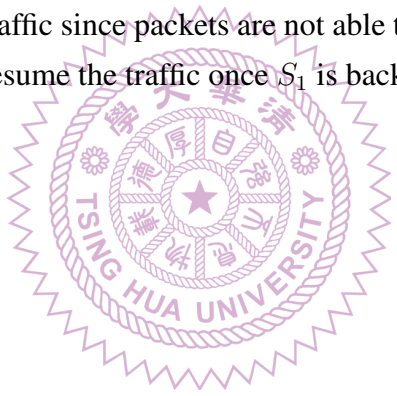


Figure 10.15: Traffic recovered by alternative links. (a) S_4 down (b) S_2 down (c) S_1 down

hosts in the network, and H_1 are sending packets to H_2 through the path: $S_1 \rightarrow S_2 \rightarrow S_4 \rightarrow S_6$. We demonstrate the three events to verify the error resilience of our system: 1) In fig. 10.14(a), the S_4 fails unexpectedly, and the controller is able to find a path from the backup paths. 2) In fig. 10.14(b), the S_2 downs, and there is no backup path. The path is computed by DPA. 3) In fig. 10.14(c), if the edge switch failed, and our system is not going to handle these flows.

Fig. 10.15 shows the links usage of the above scenario, we pick up three non-overlapping links which stand for the three different paths respectively. At 43 sec., the S_4 fails to forward the packets, and controller is able to recover the traffic by paths: $S_1 \rightarrow S_2$ and $S_2 \rightarrow S_5 \rightarrow S_6$. We can observe that the traffic is redirected from link $S_2 - S_4$ to $S_2 - S_5$. At 85 sec., another switch S_2 down, however, controller fails to recover the path using backup path. The DPA is triggered and the traffic is reassign to paths: $S_1 \rightarrow S_3$ and $S_3 \rightarrow S_5 \rightarrow S_6$. We can verify the results on the same figure, which shows the traffic is transferred to link $S_1 - S_3$. Finally, edge switch S_1 is down at 122 sec, and the controller will not need to handle the traffic since packets are not able to reach the host anyway. We simply store the status and resume the traffic once S_1 is back online.



Chapter 11

Related Work

11.1 Proactive Flow Set-up

In this section, we survey the studies on traffic engineering in both legacy and software-defined networks. First of all, we want to go through the proactive flow set-up, since our proposed tunnel is very similar to proactive flow setup in the previous work [26, 31, 54].

In [31], Lin et al. proposed to use a centralized Border Gateway Protocol (BGP) implemented in the controller to communicate with legacy Autonomous Systems (AS). The controller then uses the information from BGP update message to proactively set up flows in the SDN AS. Their system is complementary to our system in the sense that we can use their system to communicate with ISPs that run the legacy network in their system. Moshref et al. proposed FAST [31], which is a new abstraction to design switches with different state machines. Instead of installing flow entries to switches, the controller installs state machine that represents different applications (e.g., stateful firewall, TCP connection) to switches. The state machine will proactively install flow entries to switches when necessary. Curtis et al. proposed DevoFlow [10], which leverages the wildcard in OpenFlow protocol to deal with small traffic flows without involving the controller. The switch only sends the packet-in event to the controller when a flow grows beyond a certain threshold

11.2 Traffic Engineering using MPLS in legacy IP Networks

There is a rich body of literature about load balancing and other traffic engineering applications in the domain of legacy IP networks [11, 22, 42, 51]. While these works achieve excellent performance in simulation using either real-world network or synthetic traces,

due to the complexity, only a handful of these work get the chance to be implemented [55] or even evaluated in real world network [23].

In this paper, we leverage the spirit of MPLS to achieve load balance in SDNs and we also want to mention previous work of achieving load balance in IP networks. [46] gives an overview of the benefit using MPLS for traffic engineering. For example, MPLS integrating traffic from multiple paths to a single tunnel and a simple routing scheme make MPLS easy to deploy on the routers and also increase the scalability. In their opinion, MPLS is an effective and manageable solution for traffic engineering. [52] implements MPLS in Internet Service Providers' (ISP) networks and they also discuss the general MPLS issues in IP network. In their experiments, MPLS enhances the performance and also provide QoS in the ISP networks. SecondNet [15] built a virtual data center network using port-switching based source routing. They leverage MPLS to implement the source routing, using the stack of MPLS labels consumed by each hop. Their approach achieves high bandwidth utilization and also provide bandwidth guaranteed.

11.3 Traffic Engineering in SDNs

There are several traffic engineering related studies [5, 17, 21] in SDNs. In [5], Benson et al. leverage the predictable nature of data center network to mitigate the impact of congestion caused by unpredictable traffic. This work focuses on Intra-DC traffic engineering. Both [17] and [21] consider the traffic engineering of inter-DC WAN. These two work achieve high link utilization by dynamically setting up tunnels according to the demands of services that are assumed to be known, controllable, or predictable. Our work considers the case of the multi-site enterprise networks, which can not make powerful assumptions on data plane traffic. In this work, we design and implement a system to load balance unpredictable traffic through pre-build tunnels (to avoid data path setup delay).

11.4 Label Switching

SDNs were designed to lower the reduce the complexity of switch and separate the control flow from the switches to the central controller. M. Casado et al. proposed Fabric [7], they persuade using MPLS label to simplify the core network. In their proposed system, forwarding mechanisms are handled by the ingress/egress switches and the SDN controller. Leveraging MPLS simplify the forwarding procedure and also allow core network and edge devices evolve independently.

Reducing Initialization time with the pre-built tunnels is one of the features of our proposed system, M. Soliman et al. also proposed another approach to solving this prob-

lem in SDN [44]. They proposed implementing source routed in SDN. Each packet is assigned dedicated path by added a sequence of port number, which stand for the destination port number of switches along the path. on the packet header and the switches inside the core network will directly forward the packet according to the port. Their solution lowers the convergence time by 47%, which is pretty close to our solution. However, source routed doesn't follow the spirit of SDNs and even the latest OpenFlow protocols don't contain the source routed. Furthermore, they also need to program the switches to let core network switches recognized the special header, which will bring up deployment concerns and additional overhead.

Similar to our proposed architecture, the SR is also based on the MPLS, but using its own SPRING protocol which is also able to simplify the implementation compared to the traditional MPLS. The emerging SR architecture is proposed by C. Filsfils et al. from Cisco [14]. SR is respected to perform traffic engineering, service function chaining, and network resiliency. A. Sgambelluri et al. validated the SR architecture in the SDNs [43]. In their implementation, the controller running SR protocols is designed and configures the label on the edge nodes to forward the packets. M. Lee and J. Sheu proposed a routing algorithm based on the SR in SDNs [25], which consider the link bandwidth and criticality to optimize the network throughput. However, they only proposed an algorithm based on the bellmen-ford algorithm. In their work, they didn't consider how to create paths in the real network systems, and the algorithm fails to cover the resiliency issues.

J. Bellessa proposed a label switching in a hybrid-OpeFlow network environment [4]. They also strive for releasing the memory from the intensive flow rules and improve the flexibility in SDN, especially when the scalability is large enough. In their system, they proposed using the layer-2 address as the MPLS label and each packet's MAC destination field was swapped with the label. The flow was transmitted according to these labels. However, using MAC address as label cause additional overhead while the flow matching procedures. Furthermore, they leveraged the MAC address table in commodity switches to store the flow rules to support non-SDN switches, but the SDN controller can not assign the rules into these tables which lead to additional administration issues and exploit the flexibility of label switching. In their implementation, they used legacy MPLS protocols to build an MPLS network which is also unnecessary, because in SDN network, the controller has a global view of the network topology and we can wisely define the MPLS paths and also conduct traffic engineering to achieve better performance.

Chapter 12

Conclusion

12.1 Migrating Purposed System on Segment Routing

We purposed our idea of using label switching in SDN in [18]. SR has recently purposed and been standardized in a very fast speed (2016). Although our proposed system shares similar concepts with SR, our proposed system is more general than SR. There are still some issues in SR which can be solved in proposed system. For example, the tunnel finding problem (or path selection problem) is seldom discussing in previous SR works [12,25]. In this work, we had purposed unique solutions which can also be adopted in SR. The idea of decoupling flow entries and the proposed optimizing algorithm can also work with SR protocols to brings flexibility and reliability to solve the tunnel finding problem.

SR has drawn a lot of attention in SDNs, and Cisco is also trying to promote the SR [1]. Therefore, we also working on migrating the current label switching to the source routing version and brings the Traffic Engineering abilities to SR.

12.2 Future Work

In this section, we discuss several future works in our system. *In Tunnel Finder*, the routes for each pre-built tunnel need to be carefully decided, how to find the best tunnels is an important question. The limitation of flow entry in switches has always been a concern, how to pick sufficient tunnels to minimize the number of flow entry used in our system without affecting the routing flexibility is also an interesting question. *In Traffic Assigner*, we assume that controller knows the required bandwidth of each traffic request, however, it is not the case in the real world system. Determine if the system has the resource to accept the incoming traffic flow is an important question.

As mentioned in sec. 3.2, our solution adopting MPLS to route the packets which

allow us to scale the system to non-SDN AS. One possible solution is adopting Border Gateway Protocol (BGP) to interconnect with the SDN Autonomous System (AS) and non-SDN AS [27]. The routing decisions are still made by centralize controllers and use the MPLS protocols to flood the routing information.

12.3 Conclusion

In this work, we demonstrate the potential of using label-switching in SDNs to efficiently perform traffic engineering and drastically reduce the initialization overhead experienced by the first packet. We replace the packeting-based routing with label switching so that the load balancing and error resilience can be easily performed by our controller. Furthermore, decoupling the flow tables into tunnel table and path table brings the flexibility to the system, the latency sensitive traffic assigning task can be processed individually.

We mathematically formulate one model for each table and solve them by proposed heuristic algorithm. For better interactivity and latency issues, we also implemented a dynamically algorithms for tunnel construction and admission control to provide quick response online. We show in our experiments that using our proposed system could reduce maximal link utilization over SRI between 20% and 46%, and RSR between 30% and 50%. The normalized initialization delay is reduced by 39.02% and 93.22% compared to the SRI and RSR.

Bibliography

[1] Segment Routing. http://www.segment-routing.net/.

[2] Transition scenarios for 3gpp networks. RFC 3574, 2003. https://tools.ietf.org/html/rfc3574.

[3] D. Awduche. MPLS and traffic engineering in IP networks. *IEEE Communications Magazine*, 37(12):42–47, 1999.

[4] J. BELLESSA. Implementing MPLS with label switching in software-defined networks. Master's thesis, University of Illinois at Urbana-Champaign, 2015.

[5] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: fine grained traffic engineering for data centers. In *Proceedings of ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT'11)*. ACM, 2011.

[6] S. Bidkar, A. Gumaste, and A. Somani. A scalable framework for segment routing in service provider networks: The omnipresent Ethernet approach. In *In Proc. of IEEE 15th International Conference on High Performance Switching and Routing (HPSR)*, pages 76–83, 2014.

[7] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: a retrospective on evolving SDN. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 85–90. ACM, 2012.

[8] L. Cowen. Compact routing with minimum stretch. *Journal of Algorithms*, 38(1):170–183, 2001.

[9] IBM ILOG CPLEX optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/.

[10] A. Curtis, J. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: scaling flow management for high-performance networks. In *Proc. of ACM Conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM'11)*, 2011.

[11] E. Danna, S. Mandal, and A. Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *Proc of IEEE International Conference on Computer Communications (INFOCOM'12)*, 2012.

[12] L. Davoli, L. Veltri, P. Ventre, G. Siracusano, and S. Salsano. Traffic engineering with segment routing: Sdn-based architectural design and open source implementation. In *In Proc. of Fourth European Workshop on Software Defined Networks*, pages 111–112, 2015.

[13] S. Even, A. Itai, and A. Shamir. On the complexity of time table and multi-commodity flow problems. In *Foundations of Computer Science, 1975., 16th Annual Symposium on*, pages 184–193. IEEE, 1975.

[14] C. Filsfils, N. Nainar, C. Pignataro, J. Cardona, and P. Francois. The segment routing architecture. In *in prec. of 2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2015.

[15] C. Guo, G. Lu, H. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference*, page 15. ACM, 2010.

[16] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic routing encapsulation (GRE). RFC 1701, 1994.

[17] C. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *Proc. of ACM Conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM'13)*. ACM, 2013.

[18] Y. Huang, M. Lee, T. Fan-Chiang, X. Huang, and C. Hsu. Minimizing flow initialization latency in software defined networks. In *in Proc. of Network Operations and Management Symposium (APNOMS), 2015*, pages 303–308. IEEE, 2015.

[19] Iperf home page. `https://iperf.fr/`.

[20] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[21] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined WAN. In *Proc. of ACM Conference on*

Applications, technologies, architectures, and protocols for computer communications (SIGCOMM'13). ACM, 2013.

[22] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the tightrope: responsive yet stable traffic engineering. In *Proc. of ACM Conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM'05)*, 2005.

[23] S. Kandula, I. Menache, R. Schwartz, and S. Babbula. Calendaring for wide area networks. In *Proc. of ACM Conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM'14)*, 2014.

[24] C. Kozierok. *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference*. No Starch Press, 1th edition, 2005.

[25] M. Lee and J. Sheu. An efficient routing algorithm based on segment routing in software-defined networking. *Computer Networks*, 103:44–55, 2016.

[26] P. Lin, J. Hart, U. Krishnaswamy, T. Murakami, M. Kobayashi, A. Al-Shabibi, K. Wang, and J. Bi. Seamless interworking of SDN and IP. In *Proc. of ACM SIGCOMM'13*, 2013.

[27] P. Lin, J. Hart, U. Krishnaswamy, T. Murakami, M. Kobayashi, A. Al-Shabibi, K. Wang, and J. Bi. Seamless interworking of sdn and ip. In *ACM SIGCOMM computer communication review*, volume 43, pages 475–476. ACM, 2013.

[28] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. In *Proc. of ACM SIGCOMM'08*, 2008.

[29] A. Medina, A. Lakhina, I. Matta, and J. Byers. Brite: An approach to universal topology generation. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001. Proceedings. Ninth International Symposium on*, pages 346–353. IEEE, 2001.

[30] Mininet home page. `http://mininet.org/`.

[31] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan. Flow-level state transition as a new switch primitive for SDN. In *Proc. of ACM Workshop on Hot Topics in Software Defined Networking (HotSDN'14)*, 2014.

[32] RFC3031: Multiprotocol Label Switching Architecture. `https://tools.ietf.org/html/rfc3031#section-3.1`.

[33] Open Networking Foundation. https://www.opennetworking.org/.

[34] OpenDaylight home page. http://http://www.opendaylight.org/.

[35] Open vSwitch home page. http://openvswitch.org/.

[36] OpenDaylight Pathman SR App from cisco. https://github.com/CiscoDevNet/pathman-sr.

[37] C. Perkins. IP encapsulation within IP. RFC 2003, 1996.

[38] P. Pham and S. Perreau. Increasing the network performance using multi-path routing mechanism with load balance. *Ad Hoc Networks*, 2(4):433–459, 2004.

[39] R. Prabagaran and J. Evans. Experiences with class of service (CoS) translations in IP/MPLS networks. In *Proc. of IEEE Conference on Local Computer Networks (LCN'01)*, 2001.

[40] V. Rawat, R. Tio, S. Nanji, and R. Verma. Layer two tunneling protocol (L2TP) over frame relay. RFC 3070, 2001.

[41] Ryu SDN Framework Home Page. http://osrg.github.io/ryu/.

[42] C. Scoglio, T. Anjali, J. C. de Oliveira, I. Akyildiz, and G. Uhl. TEAM: A traffic engineering automated manager for DiffServ-based MPLS networks. *IEEE Communications Magazine*, 42(10):134–145, 2004.

[43] A. Sgambelluri, A. Giorgetti, F. Cugini, G. Bruno, F. Lazzeri, and P. Castoldi. First demonstration of sdn-based segment routing in multi-layer networks. In *Optical Fiber Communication Conference*, pages Th1A–5. Optical Society of America, 2015.

[44] M. Soliman, B. Nandy, T. Lambadaris, and P. Ashwood-Smith. Source routed forwarding with software defined control, considerations and implications. In *in Proceedings of the 2012 ACM conference on CoNEXT student workshop*, pages 43–44. ACM, 2012.

[45] Source packet routing in networking (SPRING), 2016. https://tools.ietf.org/html/rfc7855.

[46] G. Swallow. MPLS advantages for traffic engineering. *IEEE Communications Magazine*, 37(12):54–57, 1999.

[47] G. Swallow. MPLS advantages for traffic engineering. *IEEE communications magazine*, 37(12):54–57, 1999.

[48] R. R. T. Cormen, C. Leiserson and C. Stein. *Introduction to Algorithms*. The MIT Press, 3th edition, 2009.

[49] RFC2702: Requirements for Traffic Engineering Over MPLS. `https://tools.ietf.org/html/rfc2702#section-2.0`.

[50] D. Torrieri. Algorithms for finding an optimal set of short disjoint paths in a communication network. *IEEE Transactions on Communications*, 40(11):1698–1702, 1992.

[51] H. Wang, H. Xe, L. Qiu, Y. Yang, Y. Zhang, and A. Greenberg. Cope: Traffic engineering in dynamic networks. In *Proc. of ACM Conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM'06)*, 2006.

[52] X. Xiao, A. Hannan, B. Bailey, and L. Ni. Traffic engineering with MPLS in the internet. *IEEE Network Magazine*, 14(2):28–33, 2000.

[53] X. Xiao and L. Ni. Internet QoS: a big picture. *IEEE Network Magazine*, 13(2):8–18, 1999.

[54] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali. On scalability of Software-Defined Networking. *IEEE Communications Magazine*, 51(2):136–141, 2013.

[55] M. Zhu, J. Li, Y. Liu, D. Li, and J. Wu. TED: Inter-domain traffic engineering via deflection. In *Proc of IEEE International Symposium on Quality of Service (IWQoS'14)*, 2014.