Doctoral Thesis

Dynamic Traffic Engineering for Improving Energy Efficiency and Network Utilization of Data Center Networks

Sin-seok Seo (서신석)

Department of Computer Science and Engineering

Pohang University of Science and Technology

2014

데이터 센터 네트워크의 에너지 소모 절감 및 네트워크 자원 이용 효율 향상을 위한 동적 트래픽 엔지니어링

Dynamic Traffic Engineering for Improving Energy Efficiency and Network Utilization of Data Center Networks

Dynamic Traffic Engineering for Improving Energy Efficiency and Network Utilization of Data Center Networks

by

Sin-seok Seo
Department of Computer Science and Engineering
Pohang University of Science and Technology

A dissertation submitted to the faculty of the Pohang University of Science and Technology in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Computer Science and Engineering.

Pohang, Korea
Dec. 20, 2013
Approved by

Prof. James Won-Ki Hong

Academic Advisor

Dynamic Traffic Engineering for Improving Energy Efficiency and Network Utilization of Data Center Networks

Sin-seok Seo

The undersigned have examined this dissertation and hereby certify that it is worthy of acceptance for a doctoral degree from POSTECH.

Dec. 20, 2013

Committee Chair	James Won-Ki Hong	(Seal)
Member	Jae-Hyoung Yoo	(Seal)
Member	Jong Kim	(Seal)
Member	Jangwoo Kim	(Seal)
Member	Choong Seon Hong	(Seal)

DCSE 20080302 서신석, Sin-seok Seo, Dynamic Traffic Engineering for Improving Energy Efficiency and Network Utilization of Data Center Networks, 데이터 센터 네트워크의 에너지 소모 절감 및 네트워크 자원 이용 효율 향상을 위한 동적 트래픽 엔지니어링, Department of Computer Science and Engineering, 2014, 109P, Advisor: James Won-Ki Hong. Text in English.

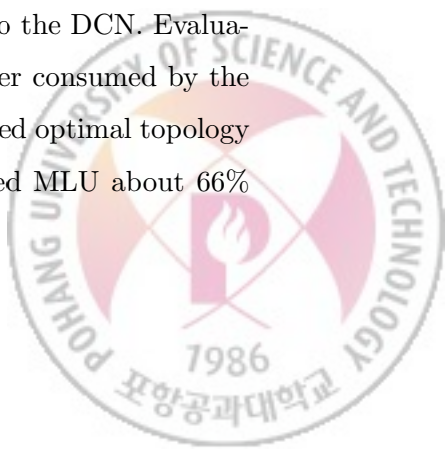
ABSTRACT

Today's Data Center Networks (DCNs) contain tens of thousands of hosts with significant bandwidth requirements as the needs for cloud computing, multimedia contents, and big data analysis are increasing. In addition, the traffic patterns of DCNs are changing from communications between a DCN server and an outside client to communications between servers within the same DCN. A conventional hierarchical tree-based DCN topology, however, has several disadvantages, including limited capacity, high capital and operational expenditures, and limited use of multi-path diversity, for building and maintaining large-scale DCNs.

The problems of existing DCN technologies and research motivation of this thesis can be summarized as follows. First, the amount of power consumed by a DCN is constant regardless of the utilization of network resources, such as links and switches, while the network utilization fluctuates depending on the time of day. As a result, operating costs of DCNs are higher than what are actually required. Second, a few specific links are experiencing congestions, especially the links connected to core switches, while other majority links are being underutilized due to a static routing path selection scheme. The last problem is that current network failure recovery mechanisms of a DCN do not consider dynamic network traffic status to avoid possible congestions.

To overcome these limitations of current DCN technologies, we propose a dynamic Traffic Engineering (TE) system for DCNs that exploits Software Defined Networking (SDN) technologies. SDN is an emerging paradigm in the networking research and industry area that separates a control plane from a data plane of network devices. The proposed TE system consists of three major procedures: optimal topology composition, traffic load balancing, and failure recovery. Optimal topology composition, the first procedure, finds a minimum subset of links and switches that can accommodate expected traffic demands at the moment and it adds extra links and switches to the minimum subset topology to diminish possible congestions. We can reduce the amount of power consumed by the DCN by turning off links and switches that are not included in the constructed subset topology. The next procedure is traffic load balancing that distributes ever-changing traffic demands over the found optimal topology to minimize Maximum Link Utilization (MLU). This MLU minimizing traffic distribution approach makes it possible to accommodate more traffic demands without adding extra network resources, i.e. it improves efficiency of DCN resource utilization. Failure recovery, the last procedure, rapidly restores from network failures by setting detour paths to DCN switches impacted by the failures. Our failure recovery proposal avoids possible congestions by considering dynamic network traffic status when selecting the detour paths and it also minimizes the number of switches to be changed.

We have implemented a prototype of the proposed TE system for a DCN; Mininet network emulation tool was used for constructing a virtual DCN and Floodlight was used for managing the DCN as a centralized SDN controller. The APIs provided by Floodlight were used for applying outputs of the TE manager to the DCN. Evaluation results using simulations revealed that the amount of power consumed by the DCN was reduced about 41% on average by applying the proposed optimal topology composition algorithms. The dynamic TE proposal also reduced MLU about 66% on average in comparison with a static routing scheme.




POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

Contents

List of Figures

List of Tables

List of Algorithms

Chapter I

INTRODUCTION

This chapter provides a brief introduction to a Data Center Network (DCN) and Software Defined Networking (SDN)-based traffic engineering. Then, the problems of current DCN technologies are described. Lastly, the research goals of this thesis and a solution approach that this thesis takes to resolve those problems are outlined.

1.1 Background

A Data Center (DC) is a facility used to house computer servers or hosts, and a Data Center Network (DCN) interconnects these hosts using dedicated links and switches. Today's DCs may contain tens of thousands of hosts with significant bandwidth requirements as the needs for cloud computing, multimedia contents, and big data analysis are increasing [1]. In addition, the traffic patterns of DCNs are changing from communications between a DCN server and an outside client to communications between servers within the same DCN.

A conventional hierarchical tree-based DCN topology, however, has several disadvantages, including limited capacity, high capital and operational expenditures, and limited use of multi-path diversity, for building and maintaining the large-scale DCN. Recently, to overcome these limitations, several new DCN topologies have been suggested including Fat-Tree [1], VL2 [2], PortLand [3], DCell [4], BCube [5], and Jellyfish [6]. These newly proposed DCN topologies provide higher link capacity via path diversity, better fault tolerance, and better scalability than the conventional DCN topology.

Software Defined Networking (SDN) is an emerging paradigm in the networking research and industry areas to cope with the recently appearing demands for flexible and agile network control. It is defined as "the physical separation of the network control plane from the forwarding plane, and where a control plane controls several devices" by Open Networking Foundation (ONF) [7]. This migration of control into programmable computing devices enables the underlying network infrastructures to be abstracted for applications and network services. The programmable control plane also enables flexible and rapid modifications of network behavior.

With SDN, a network administrator can vendor independently control the entire network from a single logical point (i.e. SDN controller) and this simplifies the network design and operation. In addition, SDN simplifies the network devices as well because the devices no longer need to implement and process many network protocol standards; instead they just need to receive instructions from the SDN controller and execute the instructions. For example, using SDN technologies, we can construct a very complex routing path, which is not possible in the current shortest path-based routing, to efficiently handle dynamically changing traffic pattern, or we

can quickly create or modify virtual networks in a cloud computing environment.

Traffic Engineering (TE) is defined as "network engineering dealing with the issue of performance evaluation and performance optimization of operational IP networks" [8]. Typical objectives of TE include balancing network load and minimizing network utilization. A lot of studies have been proposed in the area of TE and they can be classified into three categories according to routing enforcement mechanisms: MultiProtocol Label Switching (MPLS)-, Internet Protocol (IP)-, or SDN-based.

At first, TE was introduced in MPLS-based environments [9, 10] by setting dedicated Label Switched Paths (LSPs) for delivering encapsulated IP packets [11, 12]. Later, IP-based TE approaches were proposed by Fortz *et al.* [13, 14, 15, 16]. The basic idea of the IP-based TE is to set the link weights of Interior Gateway Protocols (IGPs) to control routing behavior for intra-domain traffic. Lastly, SDN-based TE is a relatively new research area that provides many advantages over the MPLS- or IP-based TE. The advantages include flexible routing management using software programs and fine-grained flow control using packet header fields match. Moreover, this SDN-based approach is relatively resilient from failures because the SDN controller can handle the failures rapidly and robustly by using global knowledge of networks. As a consequence, many SDN-based TE approaches were proposed recently including [17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27]. Our dynamic TE proposal for a DCN takes the SDN-based approach as well.

More specific benefits of SDN-based TE approaches for a DCN are summarized as follows.

- It is relatively easier to obtain traffic and failure information, which is basic input for a valid TE decision, via a (logically) centralized SDN controller.
- Any flow format with arbitrary granularity can be exploited for TE (e.g., an aggregated flow between a Top-of-Rack switch pair, an aggregated flow between two hosts, a flow defined by a 5-tuple[1], etc.)
- It is easy to apply TE results to switches in a DCN by modifying flow tables within the switches using APIs provided by an SDN controller.

1.2 Motivation and Problem Statement

Current DCNs are suffering from high operational expenditure, frequent link congestions, and delayed recovery from link or switch failures. Fundamental causes of these problems can be summarized as follows. First, the amount of power consumed by a current DCN is constant regardless of the utilization of network resources, such as links and switches, while the network utilization fluctuates depending on the time of day. However, the varying traffic demands in the DCN can be satisfied by a subset of the network resources most of the time [17]. As a result, operating costs of the DCN are higher than what are actually required.

Second, due to a static routing path selection scheme, a few specific links are frequently experiencing congestions while other majority links are being underuti-

[1](Source IP, Destination IP, Protocol Type, Source Port, Destination Port)

lized [28]. Currently, most enterprise core routers support static Equal-Cost Multi-Path (ECMP) [29], which splits traffic among a number of equal-cost paths by applying a hash function to a packet, to exploit multi-path diversity of a DCN. The static ECMP, however, does not consider dynamic nature of DCN traffic characteristics and this leads to a congestion of a specific link even in a situation where almost every other links are underutilized.

Lastly, efficient and fast network failure recovery mechanisms for a DCN, which consider network traffic status to avoid possible congestions, are missing in the current DCN technologies.

Pertaining to these problems of current DCN technologies, this thesis tries to answer the following key questions.

- What are the limitations of current DCN technologies for improving energy efficiency and network utilization?
- How can we make use of SDN technologies to realize a dynamic Traffic Engineering (TE) system for a DCN?
- How can we find an optimal subset DCN topology that satisfies varying traffic demands at the moment to reduce power consumed by the DCN?
- How can we dynamically allocate flows to a path among multiple equal cost paths in a DCN to minimize Maximum Link Utilization (MLU)?
- How can we rapidly restores from link or switch failures avoiding possible traffic congestions?

- How does the dynamic TE system for a DCN help reduce power consumption, minimize MLU, and restore quickly from network failures?

1.3 Research Goal and Approach

Considering the problems of current DCN technologies, as aforementioned in the previous section (1.2), this thesis proposes a dynamic Traffic Engineering (TE) system for a DCN based on SDN technologies. In this section, we enumerate our research goals of the TE system to resolve the current problems and outline our solution approach to achieve the goals.

The research goals of our dynamic TE system for a DCN is as follows.

- To investigate both the current and state-of-the-art DCN technologies and identify the limitations of them for improving energy efficiency and network utilization.
- To study characteristics and advantages of SDN technologies for dynamic TE and design an SDN-based dynamic TE system architecture for a DCN.
- To develop efficient algorithms for constructing an optimal subset DCN topology to reduce power consumed by the DCN.
- To develop efficient algorithms for dynamically allocating flows to minimize Maximum Link Utilization (MLU).
- To develop efficient algorithms for rapidly restoring from network failures while avoiding possible traffic congestions.

- To implement a prototype of the dynamic TE system and investigate its performance benefits.

The proposed SDN-based dynamic TE system architecture consists of three components: a DCN, an SDN Controller, and a Traffic Engineering Manager. The DCN is a target network of our traffic engineering system. The SDN controller collects traffic and failure status of the DCN from SDN switches in a centralized manner, then it aggregates and summarizes the collected data. The SDN controller also changes switching behavior of SDN switches by updating their flow tables, and turns on/off switches and links in the DCN to apply the traffic engineering decision that minimizes power consumptions and link congestions. The traffic engineering manager periodically takes traffic and failure information from the SDN controller, make a traffic engineering decision using the information, and notifies the decision results to the SDN controller.

The traffic engineering manager consists of three major procedures: optimal topology composition, traffic load balancing, and failure recovery. Optimal topology composition, the first procedure, finds a minimum subset of links and switches that can accommodate expected traffic demands at the moment. The next procedure is traffic load balancing that distributes ever-changing traffic demands over the found optimal topology to minimize MLU. Failure recovery, the last procedure, rapidly restores from network failures by setting detour paths to impacted DCN switches.

To validate the proposed dynamic TE system, we have implemented a prototype of the proposed TE system for a DCN. *Mininet* [30] network emulation tool was

used for constructing a virtual DCN and *Floodlight* [31] SDN controller was used for managing the DCN. The APIs provided by Floodlight were used for applying outputs of the TE manager to the DCN. We also have evaluated performance of the proposed TE system using simulations in terms of power saving ratio, MLU, failure restoration delay, and algorithm computation time.

1.4 Organization

The organization of this thesis is as follows. Chapter II introduces DCN topologies, an SDN concept, an OpenFlow protocol, and the basics of TE as related work. In Chapter III, we present our dynamic TE system for a DCN in detail. Thereafter, Chapter IV describes how we implemented a prototype of the proposed TE system, and then Chapter V evaluates the performance of the proposal in terms of power saving ratio, MLU, failure restoration delay, and computation time. Finally, Chapter VI concludes this thesis with a summary and contributions. It also discusses several research topics as possible future work.

Chapter II

RELATED WORK

In this chapter, we introduce several state-of-the-art Data Center Network (DCN) topologies as well as a typical tree-based one. We then summarize characteristics of DCN traffic. After that, we introduce Software Defined Networking (SDN) and an OpenFlow protocol, a *de facto* standard for realizing core concept of SDN. Thereafter, we introduce definitions and classifications of Traffic Engineering (TE). Finally, we summarize existing TE techniques for a DCN and compare them with our dynamic TE proposal.

2.1 Data Center Network

A Data Center (DC) is a facility used to house computer servers or hosts, and a DCN interconnects the hosts using dedicated links and switches. Today's DCs may contain tens of thousands of hosts with significant bandwidth requirements [1]. This section introduces a typical tree-based DCN topology as well as the latest DCN topologies

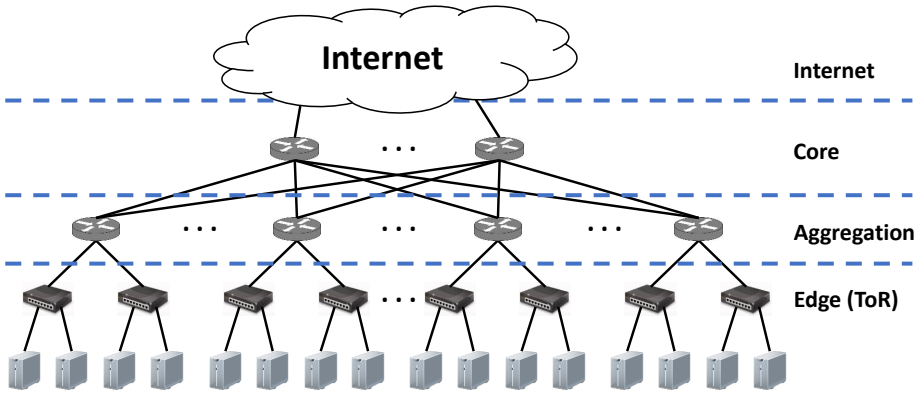


Figure II.1: A typical three-tiered DCN topology.

proposed to meet the requirements from a large-scale DC. Review of several studies about DCN traffic characteristics follows.

2.1.1 Current Data Center Network Topology

A multi-rooted hierarchical tree topology has been widely adopted for current DCNs and, typically, it consists of three tiers: core, aggregation, and edge (or Top-of-Rack, ToR) as illustrated in Figure II.1[1]. The core tier connects aggregation tier routers with each other, and it also connects the aggregation routers to the Internet. The aggregation tier exists to overcome scalability issue; the core tier alone cannot support all the edge tier switches in a large-scale DCN. Finally, the edge tier connects hosts to the aggregation tier. In general, the edge tier supports 20–40 hosts per a switch. Unfortunately, this widely adopted conventional DCN topology has several limitations for building and managing large-scale DCs as follows.

- **Limited Capacity**: Most of current DCNs exploit over-subscription to sup-

[1]Adapted from a figure by Cisco [32].

port large number of hosts using a restricted range of budget. The over-subscription is defined as "the ratio of the worst-case achievable aggregate bandwidth among the end hosts to the total bisection bandwidth of a particular communication topology" by Al-Fares *et al.* [1]. For example, an over-subscription of 2:1 means that all hosts in a DCN may potentially communicate with other arbitrary hosts at half of the full bandwidth of their network interfaces. Accordingly, if we introduce more over-subscription to a DCN, the DCN may suffer more from congestions. While the recommended over-subscription ratio is 2.5:1 to 8:1 [33], Greenberg *et al.* found that an aggregation layer is typically 5:1 to 20:1 over-subscribed, and a core layer can be 240:1 over-subscribed [2]. This large over-subscription ratio for CAPEX saving limits data communication capacity between a host to another host located in a different rack.

- **High CAPEX and OPEX**: In a hierarchical tree-based DCN topology, a core layer router has to provide many ports and handle a huge amount of traffic; these requirements make the core router very expensive. The cost of the expensive core routers dramatically increases CAPEX for building a DCN as the size of the DCN grows [1]. Moreover, the current DCN topology is not flexible enough to adapt its energy consumption according to varying traffic demands (*e.g.*, daily, weekly, or monthly). This limitation makes the OPEX of a DCN remain at a high cost [17].

- **Limited Use of Multi-Path Diversity**: Currently, most enterprise core routers support static Equal-Cost Multi-Path (ECMP) [29] to exploit multi-

path diversity of a DCN. The static ECMP, which splits traffic among a number of equal-cost paths by applying a hash function to a packet, does not consider dynamic nature of DCN traffic characteristics and this may lead to a congestion of a specific link even in a situation where almost every other links are underutilized.

2.1.2 State-of-the-Art Data Center Network Topology

Recently, several new DCN topologies have been suggested to overcome the aforementioned limitations of a conventional DCN topology: Fat-Tree [1], VL2 [2], and PortLand [3] are based on a Clos network topology [34, 35], which provides extensive path diversity using smaller commodity switches; DCell [4] and BCube [5] take a server centric approach; and Jellyfish [6] forms a DCN incrementally by selecting a random pair of switches.

Al-Fares *et al.* [1] used a special instance of a Clos network topology, called a Fat-Tree [36], to provide scalable interconnection bandwidth, economies of scale, and backward compatibility. The Fat-Tree-based DCN has regular addressing scheme, which simplifies building processes of routing table, and it provides static ECMP to distribute traffic among multiple equal-cost paths by using two-level routing tables. The realization of the two-level routing scheme requires switch modifications, which can be easily implemented using OpenFlow [37, 38] switches.

We used this Fat-Tree as a baseline DCN topology for prototype implementation and performance evaluation of our traffic engineering approach because it provides several advantages over other DCN topologies, including low CAPEX and OPEX, good scalability, high bisection bandwidth, and wide-range of path diversity. We will

further describe these advantages and implementation details in Chapter IV.

VL2 [2] is a network architecture that is built using low-cost switches arranged into a Clos topology [34, 35], which provides extensive path diversity between arbitrary two hosts. Its design goal is to provide scalable DCNs with uniform high capacity between hosts, performance isolation between services, and layer-2 semantics for Plug-and-Play. VL2 adopted Valiant Load Balancing (VLB) [39, 40, 41] to cope with volatility of a DCN in terms of workload, traffic, and failure patterns. The VLB is a method to spread traffic uniformly across network paths by randomly selecting a path without centralized coordination or traffic engineering. Experiment results showed that the VLB achieves both the uniform capacity and performance isolation objectives.

PortLand [3] is a scalable and fault-tolerant layer 2 routing protocol for DCNs working on Fat-Tree topology. The main characteristic of PortLand is that it distinguishes Actual MAC (AMAC) and Pseudo MAC (PMAC) of end hosts. The PMAC contains position information of an end host in the Fat-Tree topology, and all packet forwarding is carried out using this PMAC; it enables Plug-and-Play and small forwarding tables. Another notable advantage of PortLand is that it can autonmously detect and recover from a failure by using a fault matrix.

DCell [4] takes a server centric approach, which means that each end host has an important role for interconnecting hosts and routing traffic between two hosts. It uses a recursively-defined structure and each host connects to different levels of DCells via multiple links. Its recursive structure makes it possible to use only mini-switches to scale up instead of high-end switches. DCell is fault tolerant and supports high network capacity, because it provides rich physical connectivity be-

tween servers. However, its design requires higher wiring cost to interconnect the servers and additional functionalities on servers to route traffic.

Similarly, BCube [5], which is designed for shipping-container based modular DCs, takes a server centric approach using a recursively-defined structure. It supports various bandwidth-intensive applications including one-to-one, one-to-several, one-to-all, or all-to-all traffic patterns. It also supports graceful performance degradation as a host or switch failure rate increases. These two advantages stem from the design feature of BCube that it has multiple parallel short paths between any pair of hosts.

Jellyfish [6] exploits a degree-bounded[2] random graph topology among ToR switches to incrementally expand a DCN. The randomness of the topology allows great flexibility, which leads to easy addition of new components, support of heterogeneity, and construction of arbitrary-size networks. Surprisingly, Jellyfish supports more hosts, has lower mean path length, and is more resilient to failures than a Fat-Tree [1].

In summary, these newly proposed DCN topologies provide higher link capacity via path diversity, better fault tolerance, and better scalability than the conventional tree-based DCN topology.

2.1.3 Data Center Network Traffic Characteristic

Traffic characteristics of DCNs are known a little by a few studies despite the tremendous interest in designing improved DCNs. Here, we review some of the recent studies

[2]Degree-bounded means that the number of connections per node is limited by the number of switch ports

that have analyzed DCN traffic and revealed its characteristics.

Benson *et al.* [42, 28] analyzed traffic data (in the forms of both SNMP logs and packet traces) obtained from several DCNs including universities, private enterprises, and commercial clouds. They found out that packet losses occur primarily at the edge of the DCNs while links in the core are heavily utilized, and the DCN traffic follows a heavy tailed distribution and is bursty. In addition, they observed that flows in DCNs are generally small in size and several of these flows last only a few milliseconds. The implication of those findings in the perspective of traffic engineering for DCNs is that it is hard to directly adopt existing traffic engineering techniques designed for the Internet backbone networks, which operate at a coarse time scale of several hours and assume relatively smoother traffic variations [18].

Kandula *et al.* [43] collected network related events from 1,500 servers in a DCN for over two months by using lightweight monitoring program installed at the servers, rather than switches. They found that about 80% of flows last less than 10 s, less than 0.1% last longer than 200 s, and more than 50% of the bytes are in flows lasting less than 25 s. These findings imply that centralized traffic engineering is quite challenging. They also found that traffic volume changes quite quickly and at several times during a typical day all the network links are used close to their maximum capacity.

In an aspect of a DCN resource usage, Heller *et al.* [17] found that DCNs are typically provisioned for peak workload, but traffic can be satisfied by a subset of the network resources (*i.e.*, links and switches) most of the time because the total traffic volume in the DCN varies daily, weekly, monthly, and yearly. To save energy consumed by the DCN resources, they proposed a strategy that turns off unnecessary

links and switches considering the dynamic amount of traffic volume.

Besides the aforementioned studies about traffic characteristics of a single DCN, Chen *et al.* [44] studied characteristics of inter-DC traffic by analyzing datasets collected at the border routers of five major Yahoo! DCs. Their analysis results revealed that Yahoo! uses a hierarchical way of deploying multiple DCs—several satellite DCs distributed outside US and backbone DCs distributed inside US. Interestingly, the results showed that background inter-DC traffic has no significant trends and smaller variance compared with DC to customer traffic.

The characteristics of network failures in DCs were studied by Gill *et al.* [45]. The main findings of the study is that current DCNs are highly reliable, commodity switches exhibit high reliability, load balancers exhibit high failure rates, and network redundancy is effective only 40% in reducing failure impact.

2.2 Software Defined Networking

Software Defined Networking (SDN) is an emerging paradigm in the networking research and industry areas to cope with the recently appearing demands for flexible and agile network control. The main driving factors of the new demands are the explosion of mobile devices and content, server virtualization, advent of cloud services, and massive parallel processing of big data. However, current conventional networking technologies are not suitable for meeting the current market demands because they are too complex and static, unable to scale, and dependent on network device vendors [46]. This discrepancy between the market demands and the current networking technologies has brought the research and industry to a tipping point. In

response, the SDN concept has been introduced by starting from academia [37, 38] and it is now being developed and standardized by industry [47] in cooperation with the academia.

2.2.1 SDN Concept

SDN is defined as "the physical separation of the network control plane from the forwarding plane, and where a control plane controls several devices" by Open Networking Foundation (ONF) [7]. This migration of control into programmable computing devices enables the underlying network infrastructures to be abstracted for applications and network services. The programmable control plane also enables flexible and rapid modifications of network behavior.

With SDN, we can construct a very complex routing path, which is not possible in the current shortest path-based routing, to efficiently handle dynamically changing traffic pattern, or we can quickly create or modify virtual networks in a cloud computing environment. In addition, we can deploy a high capacity network, which is necessary for big data analysis using such as MapReduce [48], at a low cost, or we can implement complex network policies, which govern security, access control, or billing, to network infrastructures immediately.

Figure II.2 [46] illustrates a conceptual architecture of SDN, which has three layers: Infrastructure, Control, and Application. The infrastructure layer contains various network devices, and the devices communicate with (logically) centralized SDN control software in the control layer using a control data plane interface. The SDN controller, the core of SDN concept, maintains a global view of the network and controls network devices following the logic from the application layer. As a result,

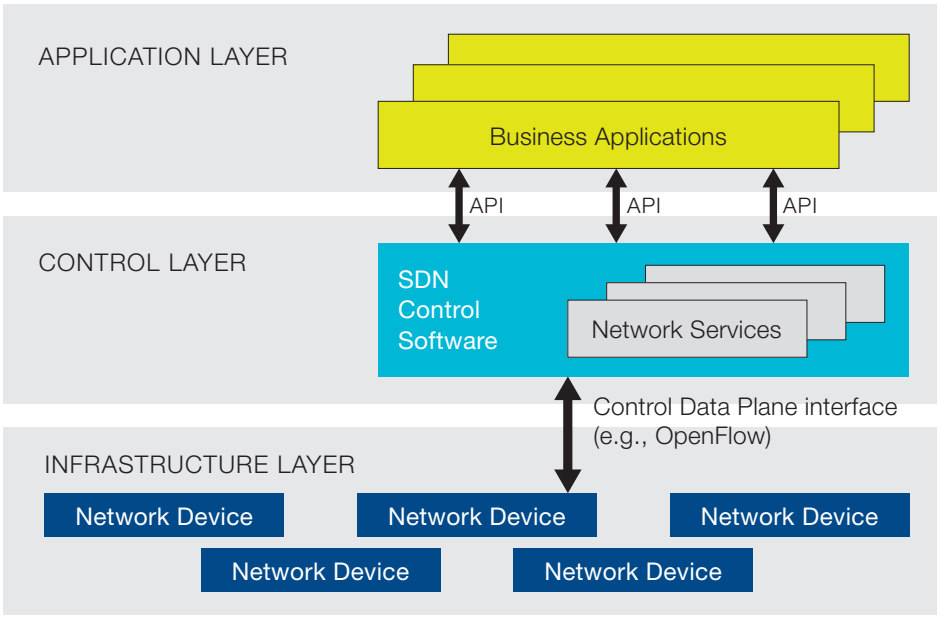


Figure II.2: Conceptual architecture of Software Defined Networking.

the network infrastructure appears to the applications as a single logical switch. Thanks to these features, a network administrator can vendor independently control the entire network from a single logical point and this simplifies the network design and operation. In addition, SDN simplifies the network devices as well because the devices no longer need to implement and process many network protocol standards, instead they just need to receive instructions from the SDN controller and execute the instructions.

2.2.2 OpenFlow

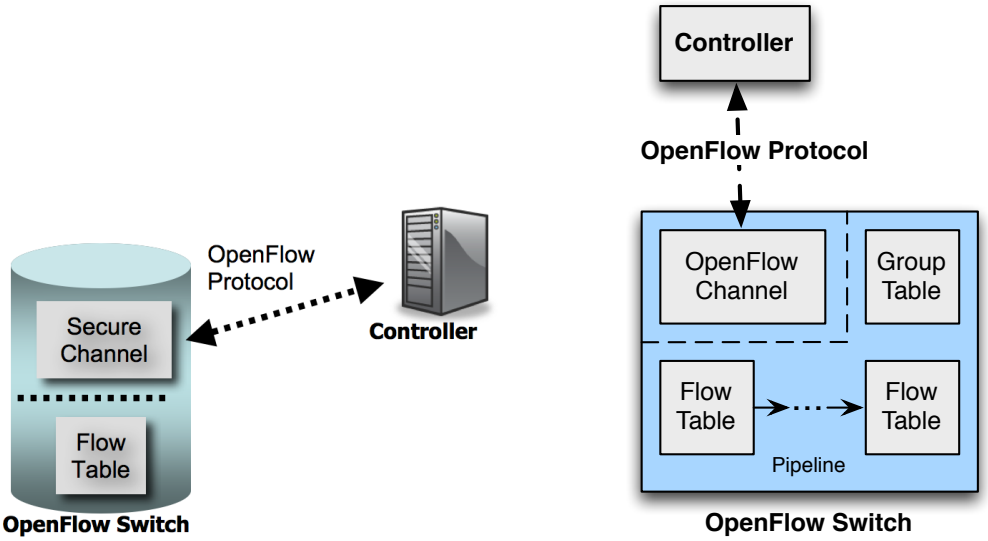
OpenFlow, which is one of the standards for communication interfaces between SDN controllers and network devices, is considered as the most appropriate technology for realizing SDN. OpenFlow allows direct access to and manipulation of the for-

Table II.1: OpenFlow-based SDN Controllers.

Name	Language	Developer
MuL [49]	C	NEC
Trema [50]	C/Ruby	NEC
NOX [51]	C++/Python	Nicira
Lithium [52]	C++	Georgia Tech.
POX [53]	Python	Nicira
Ryu [54]	Python	NTT
Frenetic [55]	Python	Cornell/Princeton
Floodlight [31]	Java	Big Switch Net.
Beacon [56]	Java	Stanford
Maestro [57]	Java	Rice Univ.

warding plane of network devices [37, 38, 46]. OpenFlow 1.0 [58] is the first standard release and currently it is the most widely adopted version by the OpenFlow switch vendors and SDN controller developers among itself and its later releases (version 1.1–1.4) [59, 60, 61, 62]. As of Oct. 2013, the latest version of OpenFlow is 1.4 [62] released at Oct. 14, 2013.

OpenFlow uses flows to control network traffic based on pre-defined match rules that can be specified by a SDN controller (state-of-the-art OpenFlow-based SDN controllers are listed in Table II.1). The SDN controller can define how traffic should be handled by OpenFlow switches using OpenFlow protocol based on parameters such as usage patterns, applications, and security policy. An OpenFlow-based SDN architecture provides highly granular control because it allows the network to be controlled on a per-flow basis, which current IP-based routing does not provide.



(a) OpenFlow 1.0 [58] (b) After OpenFlow 1.1 [59]

Figure II.3: Components of an OpenFlow switch.

The first release of an OpenFlow switch specification (version 1.0) [58] defines that an OpenFlow switch consists of a *flow table* and a *secure channel* (Figure II.3a). The controller manages the flow table by adding or removing flow entries over the secure channel using the OpenFlow protocol. The flow table contains a set of flow entries, which consists of a set of header fields, activity counters, and a set of actions, to perform packet lookup and forwarding. All packets processed by the OpenFlow switch are compared against the header fields of flow entries. If a matching entry is found, any actions for that entry are applied to the matched packet and the corresponding activity counter is incremented. If no match is found, the packet is forwarded to the controller over the secure channel, then the controller determines how to handle the unmatched packets.

Group table and pipeline processing have been added to OpenFlow switch spec-

ifications after version 1.1 [59, 60, 61, 62] (Figure II.3b). The group table contains group entries and each group entry contains a list of action buckets. The actions in one or more action buckets are applied to packets sent to the group. The pipeline processing allows packets to be sent to subsequent tables for further processing and allows meta-data to be communicated between tables. The processing stops when the instruction set associated with a matching flow entry does not specify a next table.

A meter table has been added to OpenFlow switch specifications after version 1.3 [61, 62] to enable OpenFlow to implement various QoS operations, such as rate-limiting. It consists of meter entries that defines per-flow meters, which contains meter identifier, meter bands, and counters. The meter band specifies the rate of the band and the way to process the packet. Each meter band is identified by its rate and contains band type, rate, counters, and type specific arguments.

Table II.2 shows several examples of various operation modes of OpenFlow[3]. The wild card (*) in the match fields means that all packets are matched with the fields. The simplest operation mode is default routing that sends all the incoming packets to a specified out port (port 1 in this example). Layer-2 switching can be carried out by specifying a destination MAC address in the match fields and set the action to send the matched packets to a designated port. Other operation modes including flow switching, VLAN switching, and firewall are also possible by specifying appropriate match fields and actions.

[3]Here, we use a subset (10 tuples) of entire match fields for simplicity.

Table II.2: Various operation modes of OpenFlow.

Mode	Match Fields										Action	Cnt.
	SW Port	Src MAC	Dst MAC	Ether. Type	VLAN ID	Src IP	Dst IP	Proto. No.	Src Port	Dst Port		
Def. Route	*	*	*	*	*	*	*	*	*	*	Port 1	2131
Switching	*	*	00:1f..	*	*	*	*	*	*	*	Port 2	878
Routing	*	*	*	*	*	*	1.2.3.4	*	*	*	Port 3	342
Flow Swit.	*	*	*	*	*	1.2.3.4	5.6.7.8	4	4452	80	Port 4	283
VLAN Swit.	*	*	00:3f..	*	vlan2	*	*	*	*	*	Port 6, 7	5432
Firewall	*	*	*	*	*	*	*	*	*	22	Drop	4325

2.3 Traffic Engineering

Traffic Engineering (TE) is defined as "network engineering dealing with the issue of performance evaluation and performance optimization of operational IP networks" [8]. A more explicit definition is given by Lee and Mukherjee [63]: "to put the traffic where the network bandwidth is available." Therefore, Wang *et al.* [64] mentioned that the nature of TE is "a routing optimization for enhancing network service capability without causing network congestion." Accordingly, typical TE objectives include balancing network load and minimizing network utilization. After all, to achieve these objectives, TE encompasses all the processes of traffic measurement, characterization, modeling, and, most importantly, control [11, 12].

2.3.1 Traffic Engineering Classification

A lot of papers have been published in the area of TE and Wang *et al.* [64] classified those various TE approaches according to four orthogonal criteria: traffic type (unicast or multicast), traffic optimization scope (intra- or inter-domain), timescale of operations (online or offline), and routing enforcement mechanisms (MPLS-, IP-, or SDN-based). Figure II.4[4] illustrates those taxonomy of TE.

Traffic Type (Unicast or Multicast)

In general, multicast TE is more complicated than unicast TE because it is associated with point-to-multipoint tree construction. Resource optimization of multicast TE is normally formulated as a Steiner tree problem [65] with the objective of minimizing

[4]Adapted from a figure by Wang *et al.* [64]. The SDN-based traffic engineering was included to the classifications later by us to represent our dynamic TE approach for a DCN.

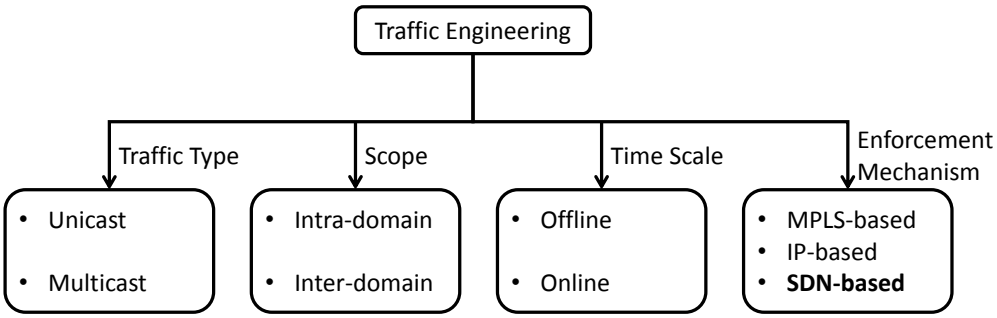


Figure II.4: Classifications of traffic engineering.

bandwidth consumption. TE for the traffic of both unicast and multicast should not be done independently without an awareness of each other since unicast and multicast traffic can be simultaneously injected into the same physical network. Our TE approach for a DCN is targeting unicast traffic.

Traffic Optimization Scope (Intra-domain or Inter-domain)

The purpose of intra-domain TE is to optimize traffic routing within a single domain (*e.g.*, ASs or DCs). On the contrary, inter-domain TE optimizes routing for inter-domain traffic that travels across multiple domains. The main focus of inter-domain TE is how to select border routers optimally as the ingress/egress points for inter-domain traffic, hence inter-domain TE can be further classified into inbound TE and outbound TE. In spite of their clear difference, intra- and inter-domain TE should not be considered independent of each other, since the network configuration of one could potentially impact another [66]. Our TE approach for a DCN is targeting intra-domain traffic within a DCN.

Timescale of Operations (Offline or Online)

The fundamental differences between offline and online TE are the use of a traffic matrix (TM) and the timescales of TE execution. The TM represents the overall traffic demand on the network with each element being the total traffic demand from a specific source to a specific destination. In some situations, it is possible to forecast the TM before TE is performed; offline TE requires estimation of the TM whereas online TE does not require any information about the future traffic demands. The average duration between two consecutive TE cycles is known as the Resource Provisioning Cycle (RPC) [67]. Normally, the RPC of offline TE for ISPs is weekly or monthly, thus the offline TE lacks of adaptive traffic manipulation according to dynamics of traffic demands and network changes. On the contrary, online TE is typically performed on a timescale of hours or even minutes in order to rapidly respond to dynamic traffic variations. Our TE approach for DCNs is both offline and online because it utilizes estimated information in the form of a TM with online timescales of TE cycles (*e.g.*, several hours or minutes).

Routing Enforcement Mechanisms (MPLS-, IP-, or SDN-based)

At first, TE was introduced in MultiProtocol Label Switching (MPLS)-based environments [9, 10] by setting dedicated Label Switched Paths (LSPs) for delivering encapsulated IP packets [11, 12]. The MPLS-based TE can explicitly set routing path and arbitrarily split traffic, thus it provides very flexible and granular controls for TE. However, it lacks of scalability and robustness because the overhead of setting up dedicated LPSs for required pairs of sources and destinations can be very

high in large-scale networks, such as DCNs, and path protection mechanisms are necessary to automatically route traffic through alternative paths in case of any link or switch failures.

Fortz *et al.* [13, 14, 15, 16] proposed the first IP-based TE approaches. The basic idea of the IP-based TE is to set the link weights of Interior Gateway Protocols (IGPs) to control routing behavior for intra-domain traffic. Unfortunately, this IP-based TE cannot provide fine-grained path selection unlike MPLS-based TE because the changes of IGP link weights may affect the routing patterns of the entire traffic. For inter-domain TE, Quoitin *et al.* [68] proposed an approach that manipulates BGP routing attributes. These IP-based TE approaches lack flexibility in path selection in comparison with the MPLS-based TE, since explicit routing and uneven traffic splitting are not possible. Nevertheless, the IP-based TE has better scalability than MPLS-based TE because no overhead for dedicated LSPs is required, and has better availability resilience because traffic can be automatically delivered via alternative shortest paths in case of link failures without explicit settings of backup paths. As a trade-off, however, this type of shortest path-based failover in the IP-based TE may severely affect optimized TE routing settings of other intact paths.

SDN-based TE is a relatively new research area that provides many advantages, including flexible routing management using software programs and fine-grained flow control using packet header fields match, over the MPLS- or IP-based TE. Moreover, this approach is relatively resilient from failures because the SDN controller can handle the failures rapidly and robustly by using global knowledge of networks. As a result, many SDN-based TE approaches were proposed recently including [17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27]. Our TE proposal for a DCN also takes the SDN-based

approach. Table II.3[5] summarizes the characteristics of MPLS-, IP-, and SDN-based TE.

2.3.2 Traffic Engineering for a Data Center Network

Many TE techniques for the traditional Internet have been proposed including [29, 11, 12, 67, 9, 10, 13, 14, 15, 16, 69, 70], but TE for DCNs is still in a preliminary stage [71]. In the meantime, the needs for large-scale DCNs are increasing sharply as well as the needs for efficient utilization of DCN resources as cloud computing and big data analysis gain attentions from IT companies and customers. Recently, for that reason, several studies about TE for DCNs have been published including [17, 18, 20, 23, 26, 27, 71, 72]. We will introduce five major approaches (Hedera [20], microTE [18], PEFT [71], DLB [23], and ElasticTree [17]) among those proposals and compare them with our TE approach.

Hedera [20], a dynamic flow routing system for a multi-stage switching fabric, utilizes a centralized approach to route *elephant* flows exceeding 10 percent of the host Network Interface Card (NIC) bandwidth while it utilizes static ECMP for the rest short lived *mice* flows. The purpose of Hedera is maximizing bisection bandwidth of a DCN by appropriately placing the elephant flows among multiple alternative paths; estimated flow demands of the elephant flows are used for the placement. The main limitation of Hedera is that it let the static ECMP routes the mice flows, which comprise more than 80% of the actual DCN traffic [43, 28]. In other words, the TE approach in Hedera can deal with only 20% of DCN traffic.

Benson *et al.* proposed microTE [18], a centralized system that adapts to traffic

[5] Adapted from a table by Wang *et al.* [64] and supplemented the comparison of SDN-based TE.

Table II.3: MPLS-, IP-, and SDN-based TE comparison.

Category	MPLS	IP	SDN
Routing Mechanism	Explicit routing with packet encapsulation	Plain IGP/BGP-based routing	Explicit routing with header fields match
Routing Optimization	Constraint-based routing	IGP link weight adustment, BGP route attribute adjustment	Flow table adjustment
Multipath Forwarding	Arbitrary traffic splitting	Even traffic splitting only	Arbitrary traffic splitting
Hardware Requirement	MPLS capable routers required	Conventional IP routers	SDN capable routers and controllers required
Route Selection Flexibility	More flexible (arbitrary path)	Less flexible (shortest path only)	More flexible (arbitrary path)
Scalability	Less scalable	More scalable	Less scalable
Failure Impact on Traffic Delivery (Availability)	High	Low	Low
Failure Impact on TE Performance	Low	High	Low

variations by leveraging the short term predictability of the DCN traffic, to achieve fine grained TE. To do so, it constantly monitors traffic variations, determines which ToR pairs have predictable traffic, and assigns the predicted traffic to the optimal path, which minimizes Maximum Link Utilization (MLU). Similar to Hedera, the remaining unpredictable traffic is then routed using weighted ECMP, where the weights reflect the available capacity after the predictable traffic has been assigned. The major shortcomings of microTE is twofold: a) it lacks of scalability due to its extremely short execution cycle, i.e. every 2 s, and b) it requires host modifications to collect fine-grained traffic data.

Penalizing Exponential Flow-spliTting (PEFT) was originally proposed by Xu *et al.* [70] to achieve optimal TE for wide-area ISP networks, where traffic demands are rather static and predictable. Switches running PEFT make forwarding and traffic splitting decisions locally and independently with each other, i.e. the TE in PEFT is done by a hop-by-hop manner. In addition, packets, even in a same flow, can be forwarded through a set of unequal cost paths by exponentially penalized higher cost paths. Later, Tso *et al.* [71] modified the PEFT for DCN TE as a reactive online version to cope with dynamic and unpredictable nature of DCN traffic. However, PEFT imposes heavy load on switches because each switch in PEFT has to measure traffic volume incoming to and outgoing from its ports and it has to calculate optimal routing paths using the measured traffic data. Another limitation of PEFT is that routing decisions of PEFT switches are not global optimal solutions because each PEFT switch carries out TE locally and independently. Finally, PEFT delivers packets in the same flow through a set of unequal cost paths by splitting them and this causes a packet reordering problem.

Dynamic Load Balancing (DLB) [23] is a centralized algorithm that distributes traffic of upcoming network flows and makes each alternative path receive equal amounts of traffic. It is designed for an OpenFlow based DCN that follows a Fat-Tree topology. The algorithm utilizes the hierarchical feature of the Fat-Tree to recursively search for paths between a given source and a destination. It then makes decisions based on real-time traffic statistics obtained every 5 s from OpenFlow switches. This DLB algorithm is specifically dependent on Fat-Tree topology, so it cannot be applied to other DCN topologies. Moreover, the algorithm selects the path by considering only local conditions, so its path allocation is a local optimal solution. Finally and most importantly, DLB lacks of scalability and responsiveness for handling large number of flows, because it has to be executed in a centralized controller each time a new flow has appeared in the DCN.

ElasticTree [17] is a centralized system for dynamically adapting the energy consumption of a DCN by turning off the links and switches that do not essentially necessary to meet the varying traffic demands at the time. The fact that traffic can be satisfied by a subset of the entire network links and switches most of the time makes ElasticTree feasible. To find the minimum subset topology, i.e. essential links and switches, ElasticTree proposed three algorithms with different optimality and scalability: Linear Programming (LP)-based formal model, greedy bin packing heuristic, and topology-aware heuristic. The main problem of ElasticTree is that its flow allocation algorithms probably cause severe link congestions because they allocate flows to links while maximizing the utilization of link capacity.

In summary, Table II.4 compares these existing TE techniques for DCNs with our TE approach.

Table II.4: Comparison of various TE approaches for DCNs.

Category	Our	Hedera[20]	microTE[18]	PEFT[71]	DLB[23]	E.Tree[17]
Objective	Min. MLU and Energy Cost	Max. Bisection BW	Min. MLU	Min. MLU	Balancing Load	Min. Energy Cost
Failure Recovery	Yes	Adopt PortLand [3]	No	No	No	No
Approach	Centralized	Centralized	Centralized	Distributed	Centralized	Centralized
Granularity	Per-flow	Per-elephant flow	Per-flow	Per-packet	Per-flow	Per-flow
Global Optimization	Yes	No	Yes	No	No	Yes
Scalability	Good	Good	Bad	Good	Bad	Good
TM Estimation	Necessary	Necessary	Necessary	Not Necessary	Not Necessary	Necessary
Switch Load	Low	Low	Medium	High	Low	Low
Host Modification	No	No	Yes	No	No	No
Experiment Topology	Fat-Tree	Fat-Tree	Typical Tree	Typical & Fat-Tree	Fat-Tree	Fat-Tree

Chapter III

DYNAMIC TRAFFIC ENGINEERING for DCNs

In this chapter, we present our dynamic Traffic Engineering (TE) system for a Data Center Network (DCN) in detail. First, we explain the overall system architecture of the proposed TE system. We then introduce two different TE algorithmic approaches, which are linear programming and a heuristic. Thereafter, detailed algorithms for optimal topology composition, traffic load balancing, and failure recovery are described.

3.1 Traffic Engineering System Architecture

Current DCNs are suffering from high operational expenditure, frequent link congestions, and delayed recovery from link or switch failures due to rigid routing scheme and inefficient network management. To overcome these limitations of current DCN

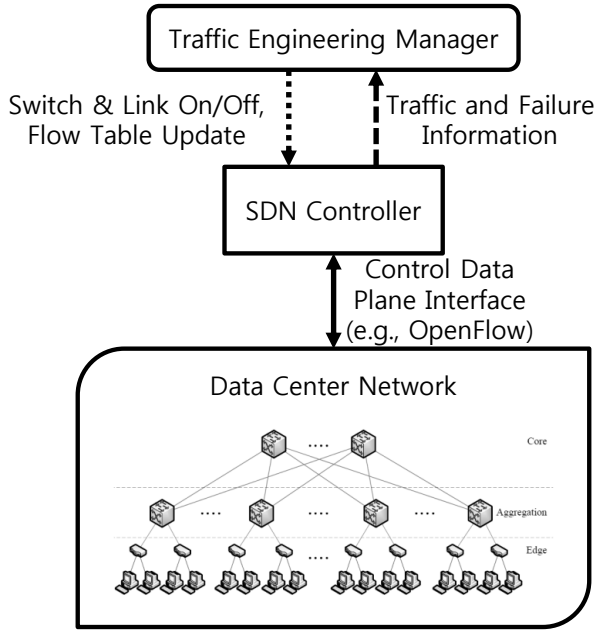


Figure III.1: Traffic engineering system architecture.

technologies, in this thesis, we propose a dynamic Traffic Engineering (TE) system for a DCN that exploits Software Defined Networking (SDN) technologies. The proposed TE system was designed to improve energy efficiency and network utilization of a DCN. In addition, it can quickly recover from network failures avoiding possible congestions by considering dynamic network traffic status.

Figure III.1 shows an overall architecture of the proposed dynamic TE system that has three components: a Data Center Network (DCN), an SDN Controller, and a TE Manager. The DCN, which contains many servers and SDN switches, is a target network of our TE system. The SDN switches in the DCN report their traffic and failure status to the SDN controller through the control data plane interface (e.g., OpenFlow). They also update their flow tables according to commands from

the SDN controller.

The SDN controller collects traffic and failure status of the DCN from SDN switches in a centralized manner, then it aggregates and summarizes the collected data. The TE manager takes the summarized traffic and failure information to dynamically make an appropriate TE decision at the time. The SDN controller also changes switching behavior of SDN switches by updating their flow tables, and turns on/off switches and links in the DCN to apply the TE decision that minimizes power consumptions and link congestions.

The TE manager, a core component of our dynamic TE system, periodically takes traffic and failure information from the SDN controller, makes a TE decision using the information, and notifies the decision results to the SDN controller. The TE manager consists of three major procedures: optimal topology composition, traffic load balancing, and failure recovery (see Figure III.2).

Optimal topology composition, the first procedure, periodically finds a minimum subset of links and switches that can accommodate hourly estimated traffic demands at the moment within the whole DCN topology. The estimated traffic demands are provided by the SDN controller as the form of traffic matrix that contains a set of (*source*, *target*, *demand*) tuples. After finding the minimum subset toptology, we add extra switches and links to the minimum topology to diminish possible congestions; we call this augmented subset topology as an *optimal* topology. This procedure also considers link or switch failures when constructing the optimal topology. We can reduce the amount of power consumed by the DCN by turning off links and switches that are not included in the constructed optimal DCN topology.

The next procedure is traffic load balancing that periodically distributes ever-

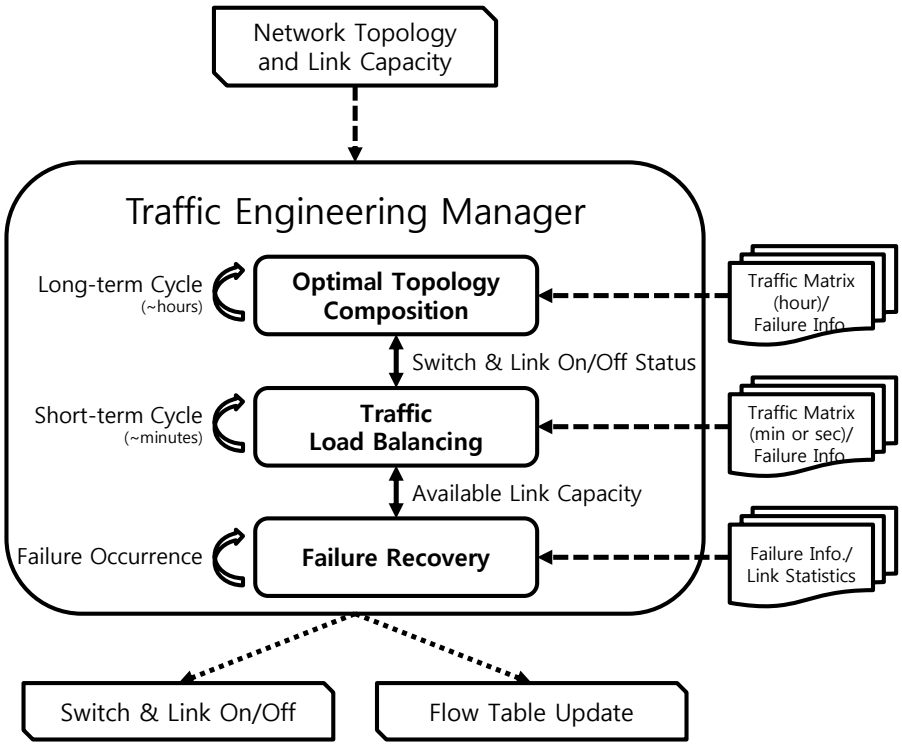


Figure III.2: Traffic engineering manager.

changing traffic demands over the found optimal topology to minimize Maximum Link Utilization (MLU). To distribute the traffic demands within only the optimal topology, this procedure takes switch and link power status as input from the optimal topology composition procedure. This procedure also considers link or switch failures when distributing the traffic demands. For this procedure, the traffic matrix is estimated in the time scale of a few minutes or seconds to reflect the ever-changing traffic demands. Note that traffic load balancing is repeatedly executed in the time scale of minutes whereas the optimal topology composition is executed in the time scale of hours. This traffic load balancing procedure makes it possible to accommo-

date more traffic demands without installing extra network resources, i.e. it improves efficiency of resource utilization in a DCN. This procedure may dynamically trigger an execution of the optimal topology composition step to secure sufficient network resources.

Failure recovery, the last procedure, rapidly restores from network failures by setting detour paths to impacted DCN switches. Unlike the former two procedures, this procedure is executed whenever a failure occurs and finds a local optimal solution. Our failure recovery proposal avoids possible congestions by considering dynamic network traffic status when selecting the detour paths and it minimizes the number of switches to be changed. After successfully recovering from failures promptly by this module, the optimal topology compostion and traffic load balancing procedures are exectued sequentially to find and apply a globally optimal solution considering the failures.

3.2 Basics of Traffic Engineering

In this section, we introduce two TE algorithmic approaches, which are Linear Programming (LP) and heuristics. Then we introduce Multi Commodity Flow (MCF) problem, the fundamental LP-based traffic engineering algorithm, and propose path-based MCF that improves the basic MCF.

3.2.1 Traffic Engineering Approach

Generally, traffic engineering problems can be solved by two different methods: linear programming and heuristics. Linear Programming (LP, or linear optimization) is a

mathematical approach for finding an optimal solution (such as maximum profit or minimum cost) in a given mathematical model [73]. More formally, LP is a technique for the optimization of a linear objective function subject to linear equality and/or linear inequality constraints. An LP problem can be expressed in a canonical form as follows:

$$\begin{aligned} &\text{maximize } \mathbf{c}^T\mathbf{x} \\ &\text{subject to } A\mathbf{x} \leq \mathbf{b} \\ &\text{and } \mathbf{x} \geq 0 \end{aligned} \tag{III.1}$$

where $\mathbf{x}$ represents the vector of variables to be determined, $\mathbf{c}$ and $\mathbf{b}$ are vectors of known coefficients, and A is a matrix of known coefficients. The expression to be maximized or minimized is called the *objective function* ($\mathbf{c}^T\mathbf{x}$ in the equation). LP has proved useful for modeling diverse types of problems in planning, routing, scheduling, assignment, and design.

A heuristic, on the other hand, is a technique designed for solving a large scale problem more quickly when classic methods (such as LP) take too long time to solve the problem. The objective of a heuristic is to produce a near-optimal solution in a short time period. This heuristic solution probably not be the best of all the possible solutions, or it may simply approximate the exact solution. A heuristic achieves speed for solving a problem by trading optimality, completeness, accuracy, or precision [74].

In this thesis, we suggest both LP- and heuristic-based approaches for solving TE problems in a Data Center Network (DCN). More specifically, we suggest using

LP-based TE for a small scale DCN to achieve the best performance while using heuristic-based TE for a large scale DCN to finish the TE computation within a practical time period. Moreover, the optimal solutions calculated using LP-based TE can be used as criteria for evaluating the performance of heuristic-based TE.

3.2.2 Linear Programming for Traffic Engineering

A mathematical model for Linear Programming (LP) consists of input, decision variables, constraints, and an objective. A TE problem can be represented as the LP mathematical model known as Multi Commodity Flow (MCF) problem [75]. We can find an optimal DCN topology or distribute traffic load over the DCN by defining an appropriate objective function and additional constraints to the basic MCF problem. In this subsection, we introduce MCF problem and propose path-based MCF, which significantly reduces computation time in comparison with MCF. We also introduce the way to prevent flow splitting [17] that may diminish network performance when we use the basic MCF-based approaches.

Multi Commodity Flow (MCF) Problem

Multi Commodity Flow (MCF) problem is a network flow problem with multiple flow demands between different source and target nodes [75, 76]. MCF is defined as follows.

- **Input**
  - Network topology: $G(V, E)$
  - Traffic matrix: T

  - Link capacity: $\forall(u,v) \in E, c(u,v)$

- **Decision Variables**

  - Flows along each link: $\forall i, \forall(u,v) \in E, f_i(u,v)$

- **Constraints**

  - Capacity limitation: $\forall(u,v) \in E, \sum_{i=1}^{k} f_i(u,v) \leq c(u,v)$
  - Flow conservation:

    $\forall i, \forall v \in V - \{s_i, t_i\}, \sum_{(u,v)\in E} f_i(u,v) = \sum_{(v,w)\in E} f_i(v,w)$
  - Demand satisfaction: $\forall i, \sum_{w\in V} f_i(s_i,w) = \sum_{w\in V} f_i(w,t_i) = d_i$

MCF takes the followings as input: a network topology graph $G(V,E)$ that has vertices $u \in V$ and edges $(u,v) \in E$, a traffic matrix T that contains k flows where i-th flow $T_i = (s_i, t_i, d_i)$ (s_i: source, t_i: target, and d_i: traffic demand), and a capacity of each link $c(u,v)$. The decision variables $f_i(u,v)$ represent allocation of flow T_i to link (u,v). In order to allocate flows in T to the network topology $G(V,E)$, MCF finds values of $f_i(u,v)$ that satisfy the three constraints: capacity limitation, flow conservation, and demand satisfaction.

The capacity limitation constraints force the sum of flows along each link (u,v) does not exceed the link capacity $c(u,v)$. The flow conservation constraints restrict flows are neither created nor destroyed at intermediate nodes. Finally, the demand satisfaction constraints ensure each source s_i or target t_i in a flow T_i sends or receives an equal amount of traffic to its demand d_i.

This MCF problem of producing integer flow allocations satisfying all demands was proved to be *NP-complete* even for only two flows and unit capacities [77]. If

fractional split of flows is allowed, the problem can be solved in polynomial time through linear programming [78].

Path-based MCF Problem

The basic MCF problem finds a solution that allocates each flow to each link. As a result, the number of decision variables $f_i(u, v)$ significantly increases as the size of a network topology grows and as the number of flows in the topology increases. The large number of decision variables makes the computation time of the MCF problem unpractical for large scale DCNs. On the other hand, our proposed path-based MCF problem finds a solution that allocates each flow to each *path* where the flow can pass through. Consequently, our path-based MCF problem dramatically reduces the computation time for solving the problem in comparison with the basic MCF. The path-based MCF problem is defined as follows.

- **Input**
  - Network topology: $G(V, E)$
  - Traffic matrix: T
  - Link capacity: $\forall (u, v) \in E, c(u, v)$
  - Set of considered paths of flows: $\forall i, P_{T_i} = \{p_{i,0}, \cdots, p_{i,j}, \cdots, p_{i,l}\}$
  - Subset of considered paths that contain a link (u, v): $\forall i, P_{T_i}^{(u,v)} \subseteq P_{T_i}$
- **Decision Variables**
  - Flows along each path: $\forall i, \forall p \in P_{T_i}, f_i(p)$

- **Constraints**
  - Capacity limitation: $\forall(u,v) \in E, \sum_{i=1}^{k} \sum_{p \in P_{T_i}^{(u,v)}} f_i(p) \leq c(u,v)$
  - Demand satisfaction: $\forall i, \sum_{p \in P_{T_i}} f_i(p) = d_i$

Similar to basic MCF, our path-based MCF takes a network topology graph $G(V,E)$, a traffic matrix T, and a capacity of each link $c(u,v)$ as input. In addition, it requires input of both a set of considered paths $P_{T_i} = \{p_{i,0}, \cdots, p_{i,j}, \cdots, p_{i,l}\}$ where $p_{i,j}$ represents a j-th path of flow T_i, and a subset of the considered paths that contain a link (u,v). The decision variables $f_i(p)$ represent allocation of flow T_i to each path p. In order to allocate flows in T to the network topology $G(V,E)$, path-based MCF finds values of $f_i(p)$ that satisfy both capacity limitation and demand satisfaction constraints. Unlike the basic MCF, our path-based MCF does not require flow conservation constraints for accurately allocating flows because each considered path itself ensures a flow to pass through only predetermined links.

Our path-based MCF dramatically reduces the computation time for solving the flow allocation problem due to its small the number of decision variables and constraints in comparison with MCF. Table III.1 compares the number of required decision variables and constraints between basic MCF and proposed path-based MCF. Let us assume *32-ary* Fat-Tree topology where $|V| = 9,472$ and $|E| = 49,152$ $(24,576 \times 2)$, the number of flows in the topology is 16,384 (#hosts $\times$ 2), and the number of alternative paths per each flow is 4. According to these assumptions, the number of decision variables of path-based MCF is 65,536 while the number of decision variables of MCF is 805,306,368. Similarly, the number of constraints of path-based MCF is 65,536 while the number of constraints of MCF is 155,238,400.

Table III.1: Comparison of the number of decision variables and constraints between MCF and proposed path-based MCF.

Category	Basic MCF	Path-based MCF
Decision Variables	$k \times \|E\|$	$k \times \|P\|$
Capacity Limitation Const.	$\|E\|$	$\|E\|$
Flow Conservation Const.	$k \times (\|V\| - 2)$	0
Demand Satisfaction Const.	$2 \times k$	k

Flow Split Prevention Constraints

Either the basic MCF or the proposed Path-based MCF generates solutions for flow allocation that probably split flows along multiple paths. This is typically undesirable due to performance degradations originated from TCP packet reordering [79]. In addition, it is very hard to install routing rules that split flows on OpenFlow switches. Fortunately, we can prevent flow splitting by adding the following decision variables and constraints to the basic MCF or the Path-based MCF.

- **Decision Variables**
  - Binary decision variable indicating whether flow i uses link (u, v): $r_i(u, v)$
- **Constraints**
  - Flow split prevention: $\forall i, \forall (u, v) \in E, f_i(u, v) = d_i \times r_i(u, v)$

The *flow split prevention* constraints ensure the traffic on link (u, v) of flow i is equal to either the full demand d_i or zero [17].

3.3 Optimal Topology Composition

The amount of power consumed by a current Data Center Network (DCN) is constant regardless of the utilization of network resources, such as links and switches, while the network utilization fluctuates depending on the time of day. However, traffic demands in the DCN can be satisfied by a subset of the network resources most of the time [17]. As a result, power consumption of the DCN are higher than what are actually required.

To reduce power consumption of a DCN, Heller *et al.* [17] proposed *ElasticTree*, which dynamically adjusts the set of active network elements that can sufficiently accommodate the fluctuating traffic demands in the DCN. Their proposal contains both an MCF-based formal model, which produces an *optimal* solution, and a greedy bin packing heuristic, which quickly produces a *near-optimal* solution. In this thesis, we propose a way to find the minimum subset topology using path-based MCF, which improves the ElasticTree using MCF. We also propose a refined greedy bin packing heuristic for minimum subset topology composition. Furthermore, our optimal topology composition proposal includes a procedure that adds extra switches and links to the minimum subset topology to reduce the possible traffic congestions.

3.3.1 Subset Topology Composition using MCF

Heller *et al.* [17] proposed a minimum subset topology composition model, which augments MCF with binary decision variables that indicate power state of links or switches. The model minimizes the total network power cost by solving a mixed-integer linear program. The minimum subset topology composition model using

MCF is defined as follows.

- **Input**
  - Network topology: $G(V, E)$
  - Traffic matrix: T
  - Link capacity: $\forall(u, v) \in E, c(u, v)$
  - Set of all switches: $S \subset V$
  - Set of nodes connected to a switch: $\forall u \in S, V_u$
  - Power cost of links: $\forall(u, v) \in E, a(u, v)$
  - Power cost of switches: $\forall u \in S, b(u)$
- **Decision Variables**
  - Flows along each link: $\forall i, \forall(u, v) \in E, f_i(u, v)$
  - Binary decision variable indicating whether a link (u, v) is powered on: $\forall(u, v) \in E, X_{u,v}$
  - Binary decision variable indicating whether a switch u is powered on: $\forall u \in S, Y_u$
- **Constraints**
  - Capacity limitation: $\forall(u, v) \in E, \sum_{i=1}^{k} f_i(u, v) \leq X_{u,v} \times c(u, v)$
  - Flow conservation:

    $\forall i, \forall v \in V - \{s_i, t_i\}, \sum_{(u,v)\in E} f_i(u, v) = \sum_{(v,w)\in E} f_i(v, w)$
  - Demand satisfaction: $\forall i, \sum_{w\in V} f_i(s_i, w) = \sum_{w\in V} f_i(w, t_i) = d_i$

  - Bidirectional link power: $\forall(u,v) \in E, X_{u,v} = X_{v,u}$
  - Switch-to-link correlation: $\forall u \in S, \forall w \in V_u, X_{u,w} = X_{w,u} \leq Y_u$
  - Link-to-switch correlation: $\forall u \in S, Y_u \leq \sum_{w \in V_u} X_{w,u} = \sum_{w \in V_u} X_{u,w}$

- **Objective**: Minimize $\sum_{(u,v) \in E} X_{u,v} \times a(u,v) + \sum_{u \in S} Y_u \times b(u)$

In addition to input of basic MCF, this minimum subset topology composition model requires extra input including a set of all switches S, a set of nodes connected to a switch V_u, and power costs of links and switches. The augmented binary decision variables $X_{u,v}$ or Y_u indicate whether a link (u,v) or a switch u is powered on respectively. With these binary decision variables, we can define an *objective function* that represents minimization of power costs of a DCN.

This subset topology composition model also requires a modification of capacity limitation constraints and extra constraints, which are bidirectional link power, switch to link correlation, and link to switch correlation. The modified capacity limitation constraints ensure flows are allocated to only those links that are powered on. The bidirectional link power constraints make the power statuses of both a link (u,v) and (v,u) consistent. The switch-to-link correlation constraints ensure that when a switch u is powered off, all links connected to this switch are also powered off. Similarly, the link-to-switch correlation constraints ensure that when all links connected to a switch u are powered off, the switch is also powered off.

3.3.2 Subset Topology Composition using Path-based MCF

As we have seen in the previous subsection, the minimum subset topology composition model using MCF is even more complicated than the basic MCF due to several

new decision variables and constraints. Therefore, the MCF-based model takes too much time for solving the problem of finding a minimum subset topology even for a small sized DCN with less than a hundred servers. To reduce the computation time, in this thesis, we propose a minimum subset topology composition model based on path-based MCF, which requires significantly less decision variables and constraints than the MCF-based model. The minimum subset topology composition model using path-based MCF is defined as follows.

- **Input**
  - Network topology: $G(V, E)$
  - Traffic matrix: T
  - Link capacity: $\forall (u, v) \in E, c(u, v)$
  - Set of considered paths of flows: $\forall i, P_{T_i} = \{p_{i,0}, \cdots, p_{i,j}, \cdots, p_{i,l}\}$
  - Subset of considered paths that contain a link (u, v): $\forall i, P_{T_i}^{(u,v)} \subseteq P_{T_i}$
  - Set of all switches: $S \subset V$
  - Set of nodes connected to a switch: $\forall u \in S, V_u$
  - Power cost of links: $\forall (u, v) \in E, a(u, v)$
  - Power cost of switches: $\forall u \in S, b(u)$
- **Decision Variables**
  - Flows along each path: $\forall i, \forall p \in P_{T_i}, f_i(p)$
  - Binary decision variable indicating whether a link (u, v) is powered on: $\forall (u, v) \in E, X_{u,v}$

  - Binary decision variable indicating whether a switch u is powered on: $\forall u \in S, Y_u$

- **Constraints**

  - Capacity limitation: $\forall (u,v) \in E, \sum_{i=1}^{k} \sum_{p \in P_{T_i}^{(u,v)}} f_i(p) \leq X_{u,v} \times c(u,v)$
  - Demand satisfaction: $\forall i, \sum_{p \in P_{T_i}} f_i(p) = d_i$
  - Bidirectional link power: $\forall (u,v) \in E, X_{u,v} = X_{v,u}$
  - Switch-to-link correlation: $\forall u \in S, \forall w \in V_u, X_{u,w} = X_{w,u} \leq Y_u$
  - Link-to-switch correlation: $\forall u \in S, Y_u \leq \sum_{w \in V_u} X_{w,u} = \sum_{w \in V_u} X_{u,w}$

- **Objective**: Minimize $\sum_{(u,v) \in E} X_{u,v} \times a(u,v) + \sum_{u \in S} Y_u \times b(u)$

3.3.3 Subset Topology Composition using Heuristic

We can find a subset topology using a heuristic within a short period of time even for a large scale DCN. While this heuristic does not guarantee a solution within a bound of optimal, it produces a high-quality subset topology in practice. In brief, this heuristic, which is called greedy bin packing [17], evaluates possible paths between a source and a target of each flow, and allocates the flow to the leftmost or the rightmost path with sufficient capacity. Similar to the linear programming-based models (MCF or path-based MCF), this approach requires knowledge of the traffic matrix in advance, but it can compute the solution incrementally; we can process the traffic matrix on-line with this heuristic.

Algorithm III.1 describes the heuristic for subset topology composition in detail. It takes a traffic matrix T, a DCN topology, and a capacity of each link as input. The

Algorithm III.1: Heuristic for Subset Topology Composition

Input : T (Traffic Matrix), DCN Topology, Link Capacity
Output: On/Off Status of *switches* and *links*

```
 1: for all flow f in T do
 2:   list_p ← possible paths from f_src to f_dst
 3:   sort list_p by position from left to right
 4:   for all path p in list_p do
 5:     if All links in p satisfy cap_avail > f_dmd then
 6:       Assign f to p
 7:       for all link l in p do
 8:         cap_avail[l] = cap_avail[l] − f_dmd
 9:       end for
10:       break
11:     end if
12:   end for
13: end for
14: for all link l in DCN do
15:   if cap_avail[l] < cap_max[l] then
16:     Set l and connected switches to turn on
17:   end if
18: end for
```

traffic matrix T is a set of flows specified by (source, target, demand) tuples. The output of this algorithm is on/off status of switches and links that constitute the entire physical DCN topology. By using only turned on switches and links specified by this algorithm, we can construct the near-minimum subset topology that satisfies all the traffic demands in T. This subset topology composition algorithm consists of two parts: 1) allocation of each flows in T to the left most path with sufficient capacity (line 1–13) and 2) determination of switches and links to be turned on (line 14–18).

3.3.4 Extra Switch and Link Augmentation

The three subset topology composition algorithms (using MCF, path-based MCF, and a heuristic) try to find the subset topology with the minimum number of links and switches to accommodate all the traffic demands in a DCN. Thus, the probability that these subset links would experience traffic congestions is very high. We can diminish this link congestion probability by adding extra switches and links on the found minimum topology.

A procedure for re-distributing the ever-changing traffic demands over the augmented optimal topology is necessary to fully utilize the reinforced extra links and to reduce possible link congestions. We will propose this re-distribution procedure (traffic load balancing) in the following section.

3.4 Traffic Load Balancing

In a current Data Center Network (DCN), a few specific links are experiencing congestions, especially the links connected to core switches, while other majority links are being underutilized due to a static routing path selection scheme [28]. In this thesis, we propose a dynamic traffic load balancing algorithms to minimize possible link congestions by distributing traffic demands over the entire DCN topology or over the subset topology found by the optimal topology composition algorithms.

Similar to the optimal topology composition algorithms, we propose three different traffic load balancing algorithms for predicted traffic demands using MCF, path-based MCF, and a heuristic. These algorithms periodically distribute the ever-changing estimated traffic demands, which are predicted in the form of traffic matrix

and in the time scale of a few minutes or seconds, over the found optimal topology to minimize Maximum Link Utilization (MLU). By doing so, they can make it possible to accommodate more traffic demands without installing extra network resources. Furthermore, our proposal includes a heuristic algorithm that can take care of *unpredicted* traffic demands as well.

3.4.1 Predicted Traffic Load Balancing using MCF

We can allocate flows in a traffic matrix T to network resources using MCF while distributing traffic load to minimize MLU. For that, it is required to define additional decision variable m, which represents MLU, and to modify capacity limitation constraints with this variable m. The predicted traffic load balancing model using MCF is defined as follows.

- **Input**
  - Network topology: $G(V, E)$
  - Traffic matrix: T
  - Link capacity: $\forall(u, v) \in E, c(u, v)$
- **Decision Variables**
  - Maximum Link Utilization (MLU): m
  - Flows along each link: $\forall i, \forall(u, v) \in E, f_i(u, v)$
- **Constraints**
  - Capacity limitation: $\forall(u, v) \in E, \sum_{i=1}^{k} f_i(u, v) \leq m \times c(u, v)$

  - Flow conservation:

    $\forall i, \forall v \in V - \{s_i, t_i\}, \sum_{(u,v)\in E} f_i(u,v) = \sum_{(v,w)\in E} f_i(v,w)$

  - Demand satisfaction: $\forall i, \sum_{w\in V} f_i(s_i,w) = \sum_{w\in V} f_i(w,t_i) = d_i$

- **Objective**: Minimize m

The modified capacity limitation constraints ensure the sum of flows along each link (u,v) does not exceed the adjusted link capacity $m \times c(u,v)$. This predicted traffic load balancing model exploits the modified capacity limitation constraints to minimize the MLU of all the links in the network by setting the objective function to minimize m.

3.4.2 Predicted Traffic Load Balancing using Path-based MCF

We can significantly reduce the computation time for solving the problem of predicted traffic load balancing by using path-based MCF instead of the basic MCF. The predicted traffic load balancing model using path-based MCF is defined as follows.

- **Input**

  - Network topology: $G(V,E)$
  - Traffic matrix: T
  - Link capacity: $\forall (u,v) \in E, c(u,v)$
  - Set of considered paths of flows: $\forall i, P_{T_i} = \{p_{i,0}, \cdots, p_{i,j}, \cdots, p_{i,l}\}$
  - Subset of considered paths that contain a link (u,v): $\forall i, P_{T_i}^{(u,v)} \subseteq P_{T_i}$

- **Decision Variables**
  - Maximum Link Utilization (MLU): m
  - Flows along each path: $\forall i, \forall p \in P_{T_i}, f_i(p)$
- **Constraints**
  - Capacity limitation: $\forall (u,v) \in E, \sum_{i=1}^{k} \sum_{p \in P_{T_i}^{(u,v)}} f_i(p) \leq m \times c(u,v)$
  - Demand satisfaction: $\forall i, \sum_{p \in P_{T_i}} f_i(p) = d_i$
- **Objective**: Minimize m

3.4.3 Predicted Traffic Load Balancing using Heuristic

We can allocate flows to minimize MLU using a heuristic within a short period of time even for a large scale DCN. While this heuristic does not guarantee a solution within a bound of optimal, it produces a high-quality traffic allocation that minimizes MLU in practice. Similar to the optimal topology composition heuristic, this heuristic takes greedy approach for allocating flows. In brief, the predicted traffic load balancing heuristic evaluates possible paths between a source and a target of each flow, and allocates the flow to a path that has minimum MLU.

Algorithm III.2 describes the heuristic for predicted traffic load balancing in detail. It takes a traffic matrix T, a DCN topology, and a capacity of each link as input. The traffic matrix T is a set of flows specified by (source, target, demand) tuples. The output of this algorithm is allocation, which minimizes MLU, of flows in T to paths. This heuristic sorts T in descending order of traffic demands to allocate large flows first (line 1). Then, for each flow in T, it evaluates possible paths from

Algorithm III.2: Heuristic for Predicted Traffic Load Balancing

Input : T (Traffic Matrix), DCN Topology, Link Capacity
Output: Minimizing MLU path allocation of flows in T

```
1:  sort T in descending order of demands
2:  for all flow f in T do
3:      list_p ← possible paths from f_src to f_dst
4:      list_MLU ← null
5:      for all path p in list_p do
6:          list_MLU[p] = MLU of p
7:      end for
8:      p_sel = list_p[index of minimum list_MLU]
9:      Assign f to p_sel
10:     for all link l in p_sel do
11:         cap_avail[l] = cap_avail[l] − f_dmd
12:     end for
13: end for
```

a source f_{src} to a target f_{dst} (line 3). Among these considered paths, this heuristic allocates the flow f to a path p_{sel} with the minimum MLU (line 4–9). Finally, it decreases the available capacities cap_{avail} of links consisting the selected path p_{sel} as many as the amount of traffic demand f_{dmd} of the flow f (line 10–12).

3.4.4 Unpredicted Traffic Load Balancing

The aforementioned *predicted* traffic load balancing algorithms allocate flows in an estimated traffic matrix, but they cannot deal with *unpredicted* flows that are not specified in the traffic matrix. Therefore, we need a way to manage those unpredicted flows as well. We can think of two different ways to do that: *reactive* and *proactive* approaches.

The former, reactive approach, follows the primitive flow processing procedure of a OpenFlow protocol [58]; whenever a new unknown flow is generated, the SDN

Algorithm III.3: Heuristic for Unpredicted Traffic Load Balancing

Input : DCN Topology, α (Average Demand), Available Link Capacity
Output: Minimizing MLU path allocation of all ToR pairs

1: **for all** flow f in ToR pairs **do**
2: $list_p \leftarrow$ possible paths from f_{src} to f_{dst}
3: $list_{MLU} \leftarrow null$
4: **for all** path p in $list_p$ **do**
5: $list_{MLU}[p] =$ MLU of p
6: **end for**
7: $p_{sel} = list_p$[index of minimum $list_{MLU}$]
8: Assign f to p_{sel}
9: **for all** link l in p_{sel} **do**
10: $cap_{avail}[l] = cap_{avail}[l] - \alpha$
11: **end for**
12: **end for**

controller decides which path to allocate the new flow with a consideration of network status at the moment. However, this approach does not scale to process hundreds of thousands of flows in a large scale DCN.

On the other hand, the latter, proactive approach, decides a path of an unpredicted flow in advance before it would be actually generated. This means that it needs to consider all the possible flows that can be generated between a source and a target in a DCN. Accordingly, this reactive approach does not scale too because it has to consider all the possible flows between all the host-to-host pairs in a large scale DCN. In this thesis, we resolve this scalability issue of the reactive approach by considering all the possible flows between all the *ToR-to-ToR* pairs instead of host-to-host pairs.

Algorithm III.3 describes the heuristic for unpredicted traffic load balancing in detail. It takes a DCN topology, average traffic demand α, and an available capacity of each link after allocating flows in a traffic matrix using the predicted traffic load

balancing procedure as input. The output of this algorithm is allocation, which minimizes MLU, of all the possible flows between all the ToR-to-ToR pairs to paths. For each flow between ToR pairs, this heuristic evaluates possible paths from a source f_{src} to a target f_{dst} (line 2). Among these considered paths, it allocates the flow f to a path p_{sel} with the minimum MLU (line 3–8). Finally, it decreases the available capacities cap_{avail} of links consisting the selected path p_{sel} as many as the amount of average traffic demand α (line 9–11).

3.5 Failure Recovery

The design goal of our failure recovery method is to provide a scalable, load balanced, and fast recovery from link failures. The proposed approach is relying on OpenFlow, with which we can separate path calculation function from link failure handling. Failure detection, path calculation, and flow setup time contribute to the total failure recovery time. Failure detection can be implemented in either a controller-based or a data plane-based manner (e.g., OSPF Hello or Bidirectional Forwarding Direction (BFD)); notice that there will be a vast discrepancy in terms of failure detection time when we choose different types of protocols to implement the failure detection. Therefore, we mainly focus on minimizing the total path calculation time and flow setup time for rapid failure recovery in this thesis.

3.5.1 Overview of Failure Recovery Procedure

Figure III.3 shows the overall working procedure of the proposed failure recovery approach. At the opening stage, we initialize the flow tables in the switch to real-

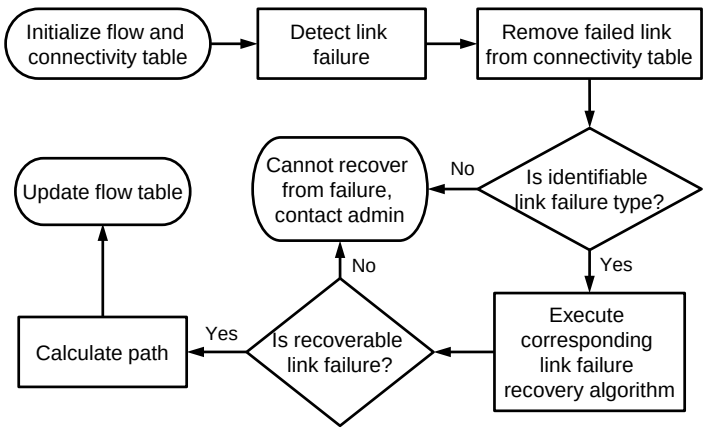
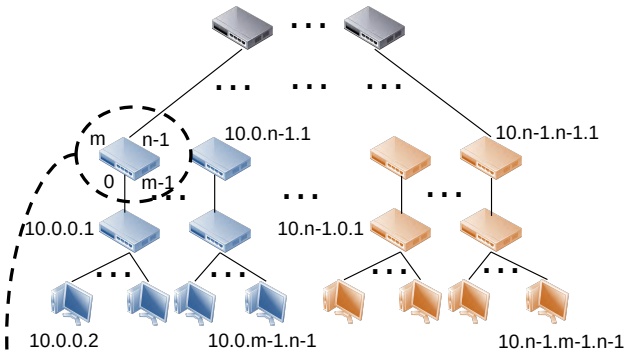


Figure III.3: Flow chart of failure recovery for a DCN.

ize the routing scheme; we also initialize the connectivity table to track down the link status of the target network topology. When a link failure occurs, we detect it by using a topology discovery protocol, and then, we identify the type of the link failure. Once the failure type is identified, we execute the corresponding link failure recovery algorithm. To preserve high scalability, we designed our failure recovery algorithm in a local optimal manner. For instance, if the topology level is three, there will be three different types of failure recovery algorithms. All the failure recovery algorithms work in the similar manner, except there is a minor difference caused by different addressing schemes and cabling patterns. First, the algorithms check the recoverability of the link failure; if the link failure can be recovered, algorithms compute new paths by referring to the traffic statistic table and the connectivity table. Then, they convert the new paths to a form of flow table entries, and invoke the OpenFlow API to insert or update the entries into corresponding switches (referred to as flow setup). As long as the flow tables are updated, buffered and newly arrived packets will be transferred through the new path.



...	priority	destination	action	
...	1	10.0.0.0/24	output=0	Prefix entries for matching downlink flow
...	1	10.0.1.0/24	output=1	
...	...	...	...	
...	1	10.0.m-1.0/24	output=m-1	
...	10	0.0.0.2/8	output=m	Suffix entries for matching uplink flow
...	10	0.0.0.3/8	output=m+1	
...	...	...	...	
...	10	0.0.0.n-1/8	output=n-1	

Figure III.4: Example of a prioritized flow table.

3.5.2 Routing Scheme and Connectivity Status Tracking

To 1) reduce the flow table size; 2) perform fast flow entry lookup; and 3) simplify the packet routing while preserving a certain level of load balancing, we adopt and port the static routing scheme introduced in Fat-Tree [1]. The static routing scheme of Fat-Tree is originally designed for load balancing and to work on Fat-Tree, but it can be easily applied to any types of DCN topologies. This static routing scheme is realized by exploiting two-level flow tables, which are comprised of a set of prefix entries (matched first and stored in the prefix table) and suffix entries (matched last and stored in the suffix table). Each entry has destination and action match field. The prefix entries are responsible for forwarding the packets from higher-layer

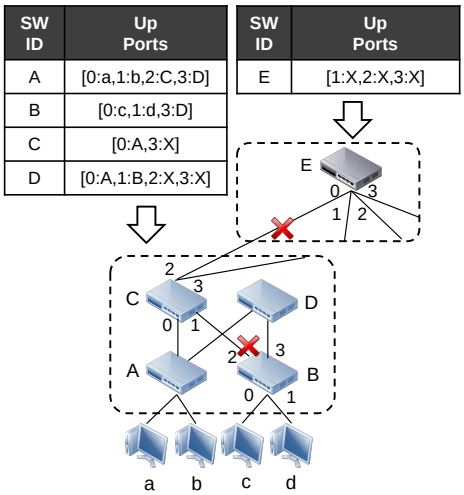


Figure III.5: Example of a connectivity table.

switches to lower-layer switches, while the suffix entries are responsible for forwarding the packets from lower-layer switches to higher-layer switches. We port the static routing scheme by exploiting the priority field provided by the OpenFlow protocol (see Figure III.4). With the help of the priority field, we can mimic the behavior of two-level flow tables by assigning higher priority value to all prefix entries, while assigning lower priority values to all suffix entries.

To track down the connectivity status of the network resources, we introduce a connectivity table. This connectivity table is responsible for storing a set of connectivity entries. A connectivity entry has a key and value pair data structure. The key stores a switch identifier (e.g., IP or MAC), while the value stores a set of activated port entries. A port entry can be further divided into a pair of active port number and the identifier of a switch that is connected to the active port (see Figure III.5). As long as a link failure has occurred, the corresponding port entry is removed from a connectivity entry.

3.5.3 Link Failure Recovery Algorithms

The underlying idea behind link failure recovery is the migration of affected flow entries bound to a failed port to another active port. The flow entry migration can be implemented by modifying the output port number of affected flow entries to another port number. In OpenFlow networks, a port number modification is typically performed by sequentially sending a flow setup message from a controller to a switch. Because the message transport is performed in a sequential manner, a larger number of messages induce more recovery time. Intuitively, to reduce such recovery time, we must minimize the number of flow setup messages generated by the OpenFlow controller. Migrating a flow entry requires one flow setup message; from this we can infer that, if we reduce the number of switches involved in the failure recovery procedure, we can achieve the least failure recovery time.

Considering this goal, we designed our algorithm in a local optimal manner. Most of the existing failure recovery methods are designed in a global optimal manner, which is for finding the shortest path from the link-failed switch to the destination host. However, those solutions suffer from relatively long recovery time caused by high computation complexity. At the other side of the spectrum, there are local optimal solutions, which are for finding the paths using a minimum amount of recovery time. The paths found by the local optimal solutions do not always obtain the shortest path length because the recovery time is regarded as the most important factor to minimize the failure recovery time. In the proposed algorithm, we assume that each switch strictly obeys its own role; therefore, the newly selected switch can somehow forward the incoming packet to an outgoing link. This assumption removes

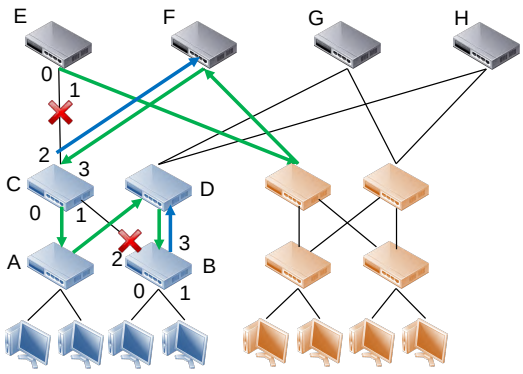


Figure III.6: Up and down link failure recovery illustration. Link failure recovery for the uplink routing path is denoted using a blue arrow line; for the downlink routing path, it is denoted using the green arrow line.

a vast amount of efforts for finding an available path between the link failed switch and all destination hosts, without checking the status of the entire paths. If any link failures occur, the OpenFlow controller enforces the link failed switch to find an alternative path in order to transfer the packet to the next hop switch by modifying the flow table entries.

In a DCN topology, all links are bidirectional (both uplink and downlink). Therefore, for a single link failure, we must fix both up and down link paths. According to the routing scheme introduced in the previous subsection, the uplink routing path from a host to a switch in the highest level is not strictly specified, and the uplink traffic is forwarded by the referring suffix flow table, which is responsible for distributing packets through different paths. By contrast, the prefix flow table strictly specifies the downlink routing path from the switch in the highest level to the destination host. The role of the prefix flow table is to gather the spread packets across higher level switches and direct them to a certain end host. This means that as long as the destination field is specified in the packet header, there is only one

unique downlink path from the highest level switch to the end host. Therefore, in the downlink case, we cannot simply migrate the flow entry by changing the bound port number. Instead, we can take the detour path by including two intermediate switches for packet forwarding as illustrated in Figure III.6. For example, if the link between switch B and C fails, in order to reach switch B from switch C, we must go through the detour path such as $C \rightarrow A \rightarrow D \rightarrow B$. The detour path always includes one lower level switch (e.g., switch A), and one higher level switch (e.g., switch D). Because we must redirect to port 0 all flows bound for port 1, we must update the flow table of switch C. Originally, there is no flow entry for switch A to direct the flows from port 2 to port 3. Therefore, a new flow entry must be inserted to realize this. Consequently, to fix both the uplink and downlink traffic, we must manipulate the flow tables of the three switches.

The pseudo code for the proposed scalable failure recovery algorithm is given in Algorithm III.4. This algorithm takes four arguments as input:

- **Connectivity Matrix Table (*conn_matrix*)**: stores the link status of the entire topology. The detailed data structured of the *conn_matrix* can be inferred from Figure III.5. Note that an entry of the connectivity matrix table can be referred to by providing a switch identifier (i.e. $conn\_matrix(switch\_id)$).

- **Traffic Statistic Table (*traffic_stat*)**: stores the incoming and outgoing packet statistics of each port. The traffic statistic table has a similar data structure as the connectivity table, except it stores the traffic statistics instead of the switch identifiers. Note that we only maintain traffic statistics collected within the last five minutes.

- **High Level Link Failed Switch (S_h):** a link failed switch instance located in a relatively higher level of network topology. For example, if a fault occurs between switch A and switch D as illustrated in Figure III.6, switch A can be denoted as S_h, and switch D can be denoted as S_l, which will be introduced in the following category. A link failed switch instance includes two properties: 1) switch identifier ($S_h.id$), and 2) de-activated port number ($S_h.port$) by a link failure.

- **Low Level Link Failed Switch (S_l)**: a link failed switch instance located in a lower level of the network topology. S_l has an identical data structure as S_h, except it stores the lower level link failed switch information.

The algorithm is comprised of four parts: 1) Remove the downed port number from the connectivity table (line 1–2); 2) Check recoverability after excluding the downed port number (line 3–6); 3) Fix the uplink failure (line 8–12); and 4) Fix the downlink failure (line 14–19). In the recoverability check phase, we check whether the algorithm can perform failure recover from the current link failure. If it can, we proceed with the failure recovery procedure; otherwise, we terminate the program and inform the administrator that the link failure is unrecoverable. In the uplink failure fixing phase, we refer to the traffic statistic table and sort the port in descending order with respect to traffic load. We choose the least loaded port of low-level switch, and migrate the flow to that port. In the downlink failure fixing phase, we go through a similar procedure as the one we performed in the uplink failure fixing phase, but include two intermediate low and high switches in the new forwarding path.

Algorithm III.4: Recovery from an Aggregate Link Failure

```
1  Remove (port : sid) pair from conn_matrix(S_h.id) entry by specifying
   downed port number S_h.port;
2  Remove (port : sid) pair from conn_matrix(S_l.id) entry by specifying downed
   port number S_l.port;
3  for (port : sid) ∈ conn_matrix(S_h.id) do
4      if conn_matrix(S_l.id) ⋂ conn_matrix(sid) ≠ φ then
5          isRecoverable ← TRUE; break;
6  if isRecoverable = FALSE then Exit(); ;
7  traffic_stat' ← traffic_stat;
   /* Fix uplink failure                                                  */
8  while obtain the least loaded (port_up : sid) from traffic_stat'(S_l.id) do
9      if conn_matrix(sid) contains S_l.id then
10         UpdateFlowTable(S_l.id, S_l.port, port_up);
11         break;
12     Remove (port : sid) from traffic_stat'(S_l.id);
13 traffic_stat' ← traffic_stat;
   /* Fix downlink failure                                                */
14 while obtain the least loaded (port_down : sid_inter_l) from traffic_stat'(S_h.id)
   do
15     if (port_up : sid_inter_h) ← conn_matrix(sid) ⋂ conn_matrix(S_l.id) ≠ φ
       then
16         AddFlowEntry(sid_inter_l, S_l.id, port_up, prefix);
17         UpdateFlowTable(S_h.id, S_h.port, port_down);
18         break;
19     Remove (port_down : sid_inter_l) from traffic_stat'(S_h.id);
```

Chapter IV

PROTOTYPE IMPLEMENTATION

In this chapter, we describe how we implemented a prototype of the proposed dynamic Traffic Engineering (TE) system for a Data Center Network (DCN) using Software Defined Networking (SDN) technologies. For that, we introduce Fat-Tree, the baseline DCN topology of the prototype implementation, and then explain implementation details of each component in the prototype. Thereafter, we explain Fat-Tree specific implementation algorithms. Lastly, we provide demonstrations of the prototype with several scenarios.

4.1 Baseline Topology: Fat-Tree

Fat-Tree, a special instance of a Clos network topology, was introduced by Al-Fares *et al.* [1] to provide scalable interconnection bandwidth, economies of scale, and backward compatibility for a DCN. In this thesis, we used Fat-Tree as a baseline DCN topology for prototype implementation and performance evaluation of our TE

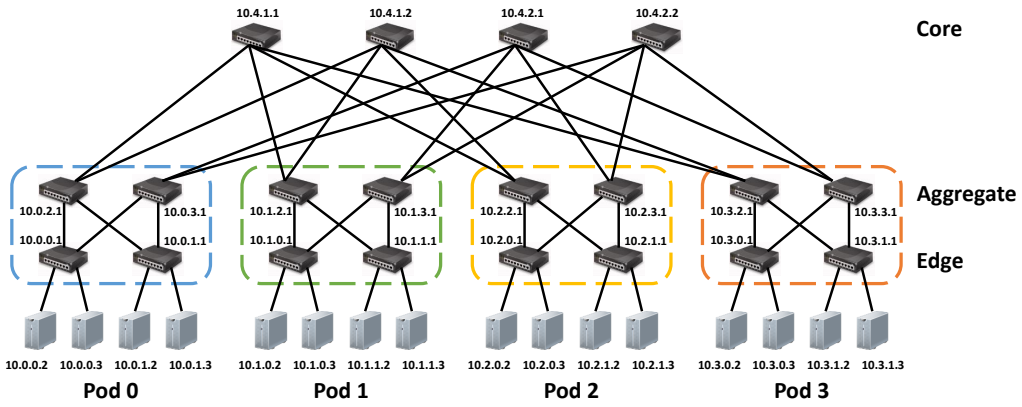


Figure IV.1: Fat-Tree topology ($k = 4$).

system because of its superior characteristics (will be explained in Section 4.1.3) to other state-of-the-art DCN topologies. Note that our dynamic TE approach can be applied to any types of DCN topologies in general, even though we validated it using a Fat-Tree topology.

4.1.1 Characteristic

A k-ary Fat-Tree topology can be organized using only k-port commodity switches (see Figure IV.1 as an example Fat-Tree with $k = 4$). In the k-ary Fat-Tree, there are k pods and each of them contains two layers (edge and aggregate) of $k/2$ switches. Half of ports in each k-port edge switch are directly connected to $k/2$ hosts and the remaining $k/2$ ports are directly connected to $k/2$ aggregate switches. These $k/2$ aggregate switches are connected to $(k/2)^2$ k-port core switches. The role of these core switches is connecting all the pods in the topology; the i-th port of a core switch is connected to pod i [1].

According to this Fat-Tree organization manner, characteristics of a Fat-Tree

Table IV.1: The characteristics of k-ary Fat-Tree.

k	*	**4**	**8**	**16**	**32**	**64**	**128**
#Hosts	$k^3/4$	16	128	1,024	8,192	65,536	524,288
#Switches	$5k^2/4$	20	80	320	1,280	5,120	20,480
#Links	$3k^3/4$	48	384	3,072	24,576	196,608	1,572,864
#Intra-Pod Paths	$k/2$	2	4	8	16	32	64
#Inter-Pod Paths	$k^2/4$	4	16	64	256	1,024	4,096

are dependent on the k value. A k-ary Fat-Tree can support $k^3/4$ hosts with $5k^2/4$ switches ($k^2/4$ core, $k^2/2$ aggregate, and $k^2/2$ edge switches). These hosts and switches are connected with each other using $3k^3/4$ links ($k^3/4$ links for each edge, aggregate, and core layer). Using these switches and links, a source and a destination hosts communicate with each other via three different types of paths: intra-rack, intra-pod, and inter-pod. A pair of hosts connected to the same edge (or Top-of-Rack, ToR) switch communicates with each other via an intra-rack path; there is only one path for this intra-rack communications. A pair of hosts in the same pod but not connected to the same edge switch communicates with each other via intra-pod paths; there are $k/2$ paths for this intra-pod communications. Lastly, communications between a source and a destination hosts located in different pods are carried out via inter-pod paths; there are $k^2/4$ paths for this type of communications. As a summary, Table IV.1 illustrates the number of supported hosts, required switches and links, and provided paths between a pair of hosts in k-ary Fat-Tree.

Table IV.2: Prefix and suffix tables example of 10.0.2.1 switch in a Fat-Tree ($k = 4$).

Prefix	Out Port	Suffix	Out Port
10.0.0.0/24	0	0.0.0.2/8	2
10.0.1.0/24	1	0.0.0.3/8	3

4.1.2 Addressing and Routing

IP addresses of switches and hosts in a Fat-Tree are allocated within the 10.0.0.0/8 private block. Core switches have IP addresses in the form of $10.k.j.i$, where j and i denote the switch's coordinates in the $k^2/4$ core switch grid (each in $[1, k/2]$, starting from top-left). Pod switches (both edge and aggregate) have IP addresses in the form of $10.pod.switch.1$, where *pod* denotes the pod number and *switch* denotes the position of the switch in the pod (starting from left-bottom). Lastly, hosts are given IP addresses in the form of $10.pod.switch.ID$, where ID denotes the host's position in a subnet (each in $[2, k/2+1]$, starting from left to right). This regular addressing scheme of a Fat-Tree is somewhat inefficient in terms of utilization of IP address space, but it helps simplify a building process of routing tables [1].

There are prefix and suffix routing tables in a Fat-Tree to deliver traffic between a pair of hosts [1]. The prefix table, which contains subnet addresses and their corresponding out port numbers, is matched first against a prefix of a destination IP address. This table is used for downlink traffic delivery from top to bottom, hence all the switches in a Fat-Tree topology should maintain this table. The suffix table, on the other hand, is used for uplink traffic delivery from bottom to top, hence this table is matched later when there is no matching entry in the prefix table; otherwise, this suffix table is not used. Accordingly, only edge and aggregate switches maintain

this table, and core switches does not. Entries of the suffix table are matched against a suffix of a destination IP address to distribute traffic load over multiple equal-cost shortest paths. Table IV.2 gives an example of these prefix and suffix tables within 10.0.2.1 switch in a simple Fat-Tree shown in Figure IV.1; a detailed flow table setup procedure will be described in Section 4.3.2.

4.1.3 Advantage

A Fat-Tree DCN topology provides many advantages over other DCN topologies as follows. First, it requires much less capital and operational expenditures for building and maintaining a DCN thanks to its use of cheap off-the-shelf commodity switches [1]. Second, a Fat-Tree topology provides good scalability for building a large-scale DCN; we have shown that we can build the large-scale DCN containing more than 10,000 hosts using only 36-port commodity switches. Third, it provides high bisection bandwidth; this means that there is some set of paths that will saturate all the bandwidth available to the end hosts in the topology for arbitrary communication patterns [1]. Fourth, there are a number of equal-cost shortest paths between a source and a destination hosts within a Fat-Tree: $k/2$ paths for intra-pod and $k^2/4$ paths for inter-pod traffic; this multi path diversity contributes a good chance to improve TE performance. Finally, a Fat-Tree topology and its routing scheme can be easily realized using OpenFlow switches (we will further explain how we did this in Section 4.3.2). Considering these advantages, many existing studies also used a Fat-Tree as a baseline topology including VL2 [2], PortLand [3], ElasticTree [17], and Hedera [20]. We also have chosen to use a Fat-Tree topology for prototype implementation and validation of our dynamic TE approach.

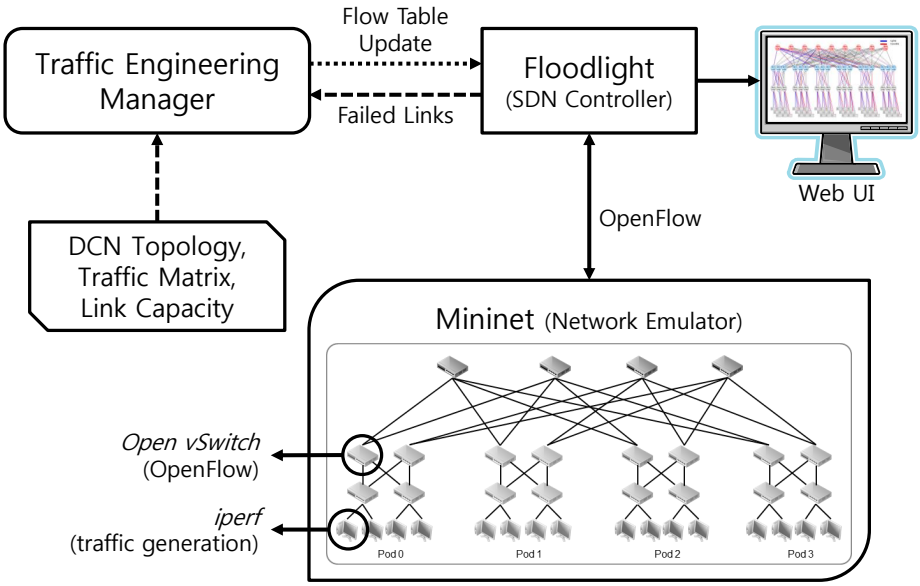


Figure IV.2: Components of prototype implementation.

4.2 Prototype Implementation Detail

Figure IV.2 illustrates each component used in the prototype implementation of our dynamic TE system. We employed *Mininet* [30] network emulation tool for constructing a virtual DCN topology. With Mininet, we can create a realistic virtual network, which executes real Linux kernel, switch, and application code, on a single machine. It also provides a useful way to develop and experiment SDN applications using OpenFlow protocol. Each switch in the virtual topology emulated by Mininet is a virtual instance of *Open vSwitch* [80]. This Open vSwitch is a multilayer virtual switch, which is designed to enable massive network automation, while supporting standard management interfaces and protocols including NetFlow, sFlow, SPAN, RSPAN, CLI, LACP, and 802.1ag. For the automated network control, this virtual switch supports OpenFlow protocol as well. To generate traffic among hosts, we

used *iperf* [81], which is a network testing tool that can create TCP or UDP data streams. It also provides measurement logs of the data streams.

Among various SDN controllers introduced in Section 2.2, we have chosen to use *Floodlight* [31], a Java-based OpenFlow controller, as a centralized manager of the DCN. It collects network status data using OpenFlow protocol and visualizes the collected data through the Web User Interface (UI). It also updates flow tables of switches using OpenFlow protocol to apply TE results. Floodlight offers a module loading system that makes it simple to extend and enhance; we exploited this beneficial feature to integrate failure recovery module of our TE proposal to Floodlight and to enhance visualization of a Fat-Tree topology.

The traffic engineering manager, a core component of our TE system, takes a DCN topology, a traffic matrix, and a capacity of each link as input. It then makes a TE decision using the input, and notifies the decision results to the SDN controller using APIs provided by Floodlight. We implemented prototype of this TE manager using a Python 2.7.4 programming language. TE algorithms based on Linear Programming (LP) were modeled using puLP [82], a Python-based LP modeler, which can call various LP solvers including GLPK [83], COIN [84], CPLEX [85], and Gurobi [86]. Both GLPK and COIN are open source projects, and both CPLEX and Gurobi are commercial products. Among these solvers, we used Gurobi with an academic license and it outperformed the open source solvers (GLPK and COIN).

4.3 Fat-Tree Specific Algorithm

In this Section, we explain an implementation algorithm that computes a set of equal-cost shortest paths between a source and a destination hosts in a Fat-Tree topology. We then describe flow table setup procedures for both default Fat-Tree static routing and TE-based routing.

4.3.1 Path Acquisition

Algorithm IV.1 describes a path acquisition procedure between a pair of hosts in a Fat-Tree topology. This path acquisition algorithm provides essential data for both path-based MCF and heuristic algorithms proposed in Chapter III. It takes k value of Fat-Tree (to know the organization of a target topology), source IP src, and destination IP dst as input. The output of this algorithm is a set of equal-cost shortest paths in a given Fat-Tree topology between a given src and dst.

First of all, this algorithm obtains IP addresses of both a source and a destination edge switches ($edgeSW_{src}$ and $edgeSW_{dst}$). They can be easily derived by substituting the last field of src or dst IP addresses to 1 (line 1–2). After that, the source pod number src_{pod} and the destination pod number dst_{pod} are compared (line 3). If they are not equal, this flow is inter-pod flow, which has $k^2/4$ equal-cost shortest paths (line 4–11). This inter-pod flow is required to pass through a core switch $coreSW$ as well as aggregate switches $aggSW_{src}$ and $aggSW_{dst}$ to communicate with an end host located in a different pod (line 9). If the src_{pod} and the dst_{pod} are equal, then the source subnet src_{subnet} and the destination subnet dst_{subnet} are compared (line 12). If they are different, this flow is intra-pod flow, which has $k/2$ equal-cost short-

Algorithm IV.1: Path Acquisition in a Fat-Tree Topology

Input : k Value of Fat-Tree Topology, Source IP src, Destination IP dst
Output: A Set of Equal Cost Shortest Paths P between src and dst

1: $edgeSW_{src} = 10.src_{pod}.src_{subnet}.1$
2: $edgeSW_{dst} = 10.dst_{pod}.dst_{subnet}.1$
3: **if** $src_{pod} \neq dst_{pod}$ **then** {Inter-Pod flow ($k^2/4$ paths}
4: **for** $i = k/2$ **to** $k - 1$ **do**
5: $aggSW_{src} = 10.src_{pod}.i.1$
6: $aggSW_{dst} = 10.dst_{pod}.i.1$
7: **for** $j = 0$ **to** $k/2 - 1$ **do**
8: $coreSW = 10.k.(i \texttt{mod} (k/2) + 1).(j + 1)$
9: $P = P \cup \{(src \rightarrow edgeSW_{src} \rightarrow aggSW_{src} \rightarrow coreSW \rightarrow aggSW_{dst} \rightarrow edgeSW_{dst} \rightarrow dst)\}$
10: **end for**
11: **end for**
12: **else if** $src_{subnet} \neq dst_{subnet}$ **then** {Intra-Pod flow ($k/2$ paths)}
13: **for** $i = k/2$ **to** $k - 1$ **do**
14: $aggSW = 10.src_{pod}.i.1$
15: $P = P \cup \{(src \rightarrow edgeSW_{src} \rightarrow aggSW \rightarrow edgeSW_{dst} \rightarrow dst)\}$
16: **end for**
17: **else** {Intra-Rack flow (1 path)}
18: $P = \{(src \rightarrow edgeSW_{src} \rightarrow dst)\}$
19: **end if**

est paths (line 13–16). For this intra-pod flow, it is not necessary to pass through a core switch, but it is required to pass through an aggregate switch $aggSW$ (line 15). Lastly, if both pod numbers and subnets of src and dst are identical, then this flow is intra-rack flow that can communicate with each other through the same edge switch ($edgeSW_{src}$ or $edgeSW_{dst}$); there is only one path (line 17–18).

Algorithm IV.2: Flow Table Initialization for Fat-Tree Static Routing.

Input : k Value of Fat-Tree Topology
Output: Flow Table for each Switch in the Fat-Tree

for $x = 0$ **to** $k - 1$ **do**
  for $z = 0$ **to** $k/2 - 1$ **do** {Edge switch}
    for $h = 2$ **to** $k/2 + 1$ **do**
      `AddPrefix`$(10.x.z.1, 10.x.z.h/32, h - 1, P_{prefix})$
      `AddSuffix`$(10.x.z.1, 0.0.0.h/8, (h - 2 + z)\texttt{mod}(k/2) + (k/2), P_{suffix})$
    end for
  end for
  for $z = k/2$ **to** $k - 1$ **do** {Aggregation switch}
    for $i = 0$ **to** $k/2 - 1$ **do**
      `AddPrefix`$(10.x.z.1, 10.x.i.0/24, i, P_{prefix})$
    end for
    for $h = 2$ **to** $k/2 + 1$ **do**
      `AddSuffix`$(10.x.z.1, 0.0.0.h/8, (h - 2 + z)\texttt{mod}(k/2) + (k/2), P_{suffix})$
    end for
  end for
end for
for $j = 1$ **to** $k/2$ **do** {Core switch}
  for $i = 1$ **to** $k/2$ **do**
    for $x = 0$ **to** $k - 1$ **do**
      `AddPrefix`$(10.k.j.i, 10.x.0.0/16, x, P_{prefix})$
    end for
  end for
end for

4.3.2 Flow Table Setup

Algorithm IV.2[1] describes a flow table setup process for default Fat-Tree static routing. It initializes prefix and suffix tables (explained in Section 4.1.2) of each OpenFlow switch in a Fat-Tree topology. Note that we exploit a *priority* field provided by OpenFlow protocol to realize two-level routing of a Fat-Tree; prefix table entries are given a high priority P_{prefix} to be matched first while suffix table

[1] Adapted from an algorithm by Al-Fares *et al.* [1] and modified to apply on OpenFlow switches.

Table IV.3: Flow table setup example of 10.0.2.1 switch in a Fat-Tree ($k = 4$).

Category	Priority	Match Fields		Action
		Src IP	Dst IP	
Prefix	4000 (high)	*	10.0.0.0/24	Port 0
	4000	*	10.0.1.0/24	Port 1
TE	3000 (mid)	10.0.1.2/32	10.2.0.3/32	Port 3
	3000	10.0.0.2/32	10.2.0.3/32	Port 3
	...	...	...	...
Suffix	2000 (low)	*	0.0.0.2/8	Port 2
	2000	*	0.0.0.3/8	Port 3

entries are given a low priority P_{suffix} to be matched later. To achieve this, an `AddPrefix`(sw, IP_{prefix}, p, P_{prefix}) function installs a prefix entry to the switch sw. This prefix entry has the destination IP match field IP_{prefix} with the priority P_{prefix} and all the matched packets will be delivered to a port p. Similarly, an `AddSuffix`(sw, IP_{suffix}, p, P_{suffix}) function installs a suffix entry.

After initializing the basic Fat-Tree prefix and suffix flow entries, we can apply TE results using flow entries with a medium priority. As we have described above, prefix entries are matched first with a high priority while suffix entries are matched last with a low priority; by giving a medium priority to the TE flow entries, we can change the uplink (bottom to top) traffic delivery path of a flow before suffix flow entries decide it. The prefix flow entries still will be in charge of downlink (top to bottom) traffic delivery with the highest priority. Table IV.3 shows these prefix, suffix, and TE flow entries of a OpenFlow switch as an example.

4.4 Demonstration

Figure IV.3 and IV.4 show screen captures of Floodlight Web UI executed using the implemented prototype of our dynamic TE system. We have captured these figures while we were emulating a 6-ary Fat-Tree using Mininet [30] where each host generated two flows (for Figure IV.3) or one flow (for Figure IV.4). These figures intuitively illustrate the effects of our dynamic TE system with visualized link utilization updated in real-time; the thicker a link is, the more it is utilized.

Figure IV.3a shows a case where default Fat-Tree static routing was applied. As you can see, the entire network resources were used for delivering traffic and the most of the links were underutilized. On the contrary, Figure IV.3b, a case where the proposed optimal topology composition algorithm was applied, shows that only an optimal subset of network resources was used for accommodating the same traffic demands. The unused switches and links can be turned off to reduce power consumptions of a DCN. In this case, however, some of links (especially the core links) were heavily utilized (illustrated as thick lines in the Figure). This means that link congestions would occur with a high probability. To minimize the occurrences of link congestions, we applied our traffic load balancing algorithm to the augmented optimal topology (seven switches—one aggregate per pod and one core—were added) as shown in Figure IV.3c. We can identify that our optimal topology composition and traffic load balancing algorithms reduced power consumed by a DCN as well as minimized a maximum link utilization.

Figure IV.4 demonstrates recovery from link failures. A normal case without failures where each flow was delivered according to default Fat-Tree routing scheme

is shown in Figure IV.4a. In this situation, we artificially injected two failures to the leftmost aggregate and core links, respectively. As we have shown in Figure IV.4b, our failure recovery algorithm proposed in Section 3.5 has immediately restored from these failures using detour paths. Moreover, our traffic load balancing algorithm, which is supposed to be periodically executed every few minutes, has restored from failures using alternative equal-cost shortest paths after the immediate recovery using detour paths (Figure IV.4c).

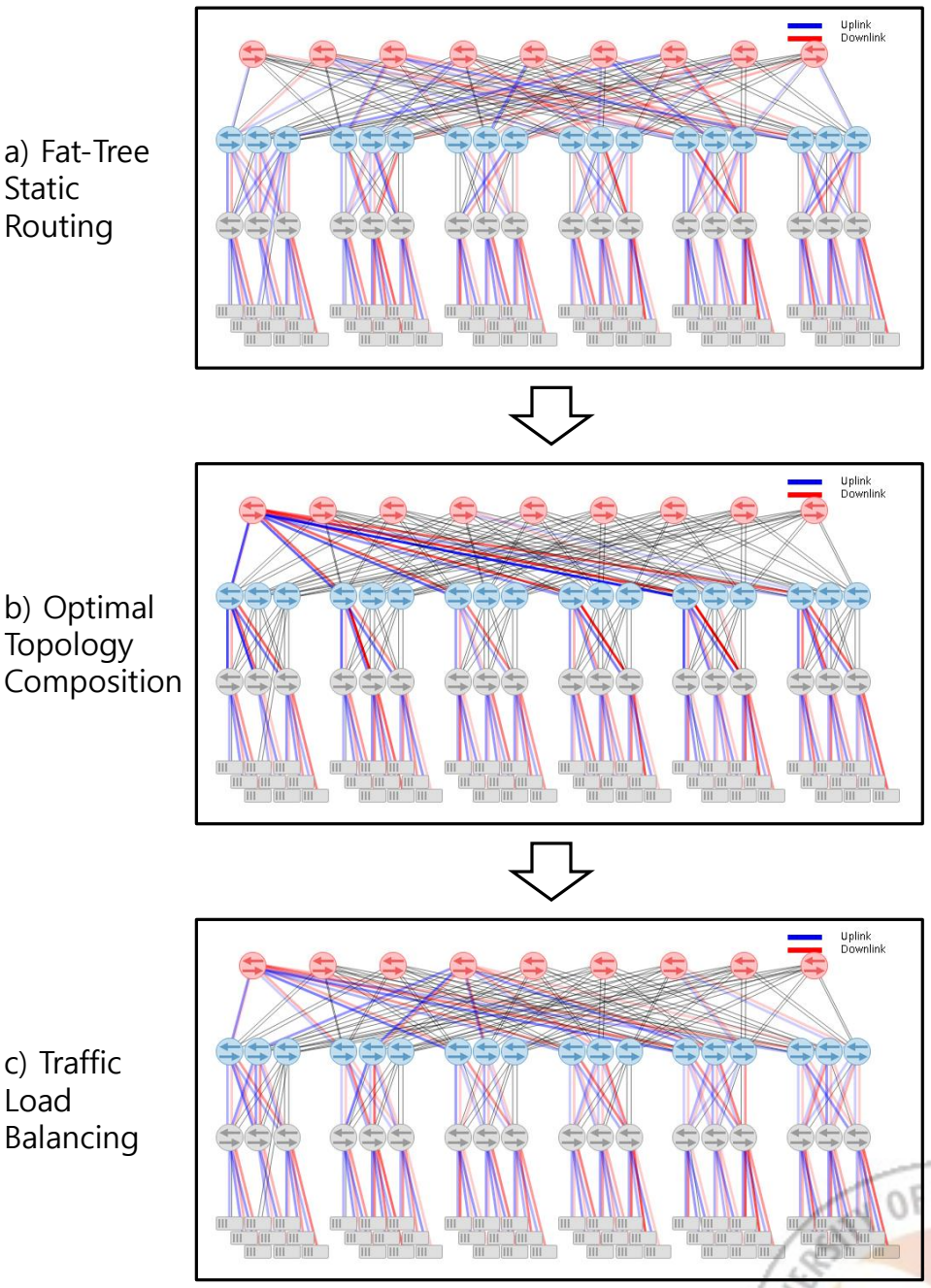


Figure IV.3: Demonstration of a) Fat-Tree static routing, b) optimal topology composition, and c) traffic load balancing on a Fat-Tree topology ($k = 6$).

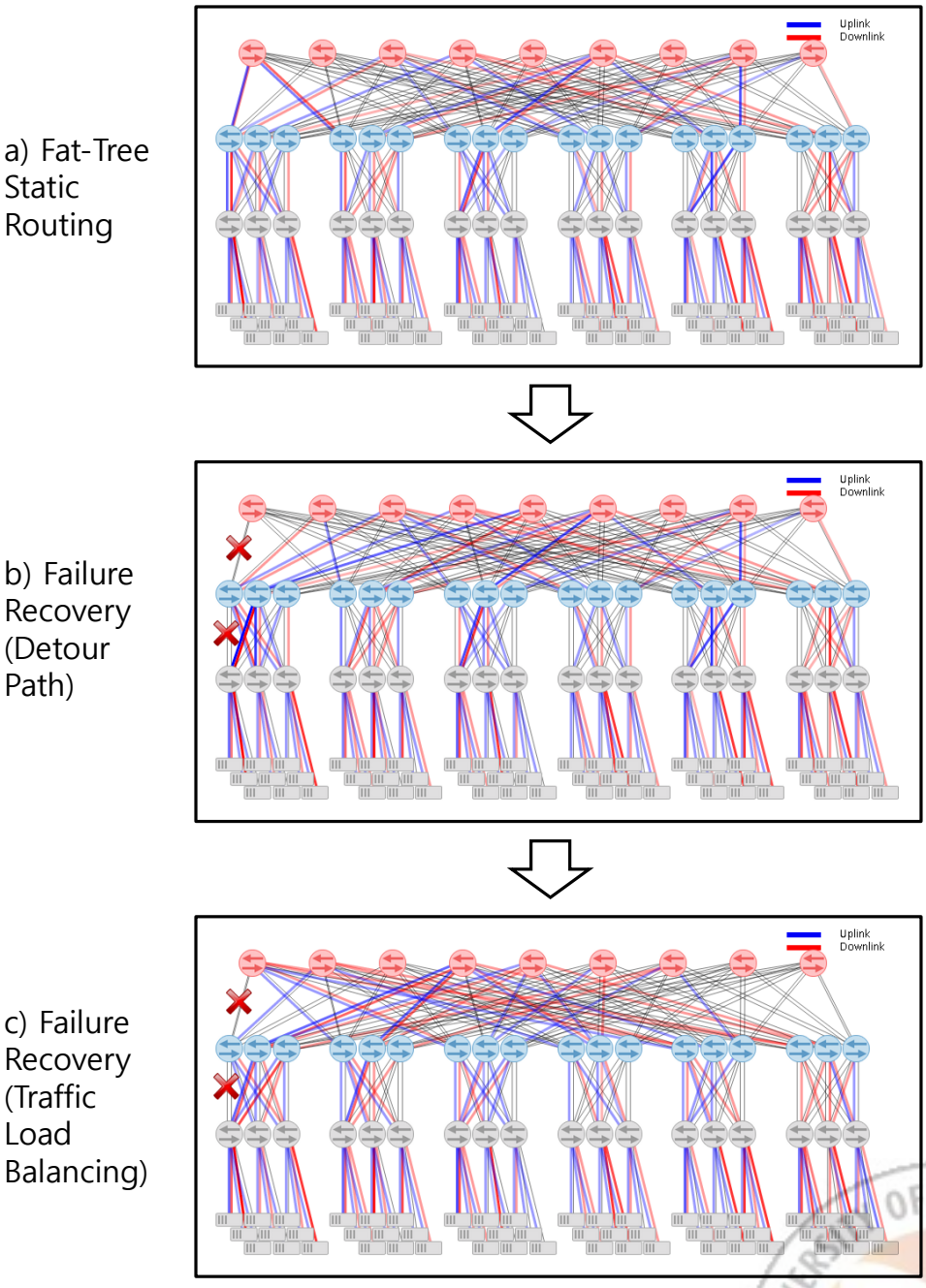


Figure IV.4: Demonstration of a) Fat-Tree static routing, b) failure recovery using detour paths, and c) failure recovery with traffic load balancing on a Fat-Tree topology ($k = 6$).

Chapter V

PERFORMANCE EVALUATION

This chapter provides performance evaluation results of the proposed dynamic Traffic Engineering (TE) system for a Data Center Network (DCN) by means of simulations. It is intended to complement the prototype implementation described in Chapter IV for validating our TE proposal. Using simulations, we can easily evaluate our TE algorithms for large-scale DCN topologies with various traffic demands. Note that it is not possible to build a large-scale DCN topology using the Mininet-based prototype implementation because it emulates all the switches and hosts of a virtual DCN within a single computing machine.

5.1 Experimental Environment

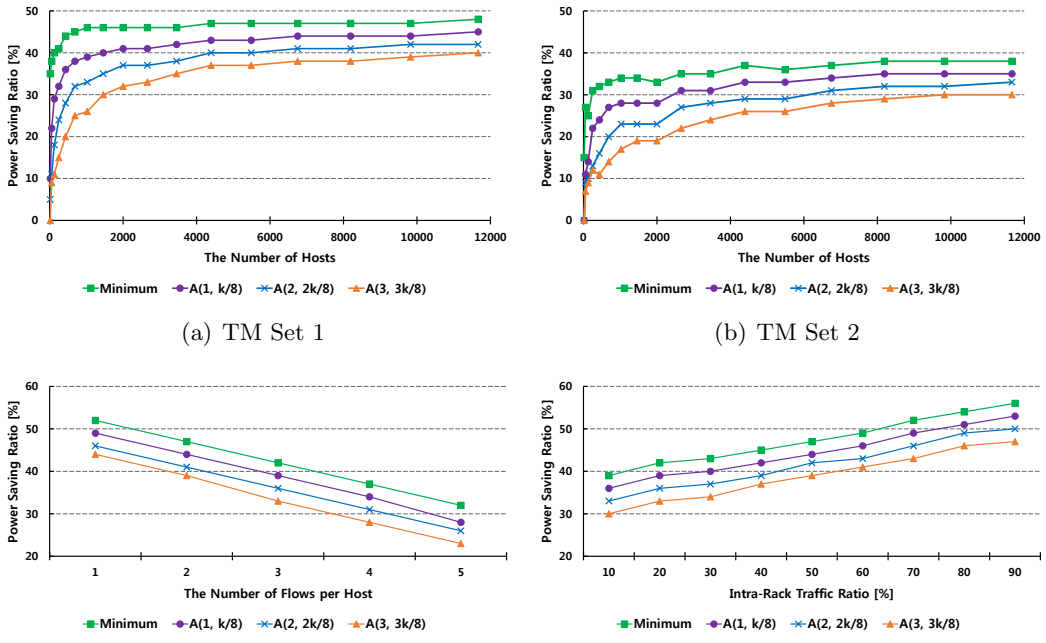
We used a *Dell PowerEdge R710* computer, which was equipped with 3.47-GHz *Intel Xeon X5690* CPUs and 48-GB memories, for running simulations evaluating our DCN TE system. The power costs of each switch and link were assumed as 150

Table V.1: Traffic matrix data sets.

Category	**Set 1**	**Set 2**	**Set 3**	**Set 4**
***k* of Fat-Tree (#Hosts)**	**4–36** (16–11,664)	**4–36** (16–11,664)	32 (8,192)	32 (8,192)
#Flows per Host	2	4	**1–5**	2
Intra-Rack Traffic %	50	50	50	**10–90**
Traffic Demands	10–20% of a maximum link capacity			
Unpredicted Traffic	◦ #Flows: 5 per host ◦ Traffic Demands: 1–5% of a maximum link capacity			

and 1, respectively, referring to a power consumption analysis [17].

We used four different data sets of traffic matrix summarized in Table V.1 for experiments. These data sets were designed to cover as many cases as possible by varying related parameters including the k value of Fat-Tree (to change the size of a DCN), the number of flows, and intra-rack traffic ratio. Each flow in these four traffic matrices has a demand that was randomly selected in between 10–20% of a maximum link capacity in common. Both the data set 1 and 2 were generated by increasing the k value of Fat-Tree from 4 to 36 by the unit of 2 to examine influences of the size of a DCN. The only difference between them was the number of flows generated by each host: 2 for data set 1 and 4 for set 2. The data set 3 was generated by increasing the number of flows per host from 1 to 5 to identify influences of a traffic load in a DCN. Lastly, the data set 4 was generated by changing the intra-rack traffic ratio from 10 to 90%. Moreover, these four data sets also contain unpredicted traffic in common; the unpredicted traffic was assumed to contain 5 flows per host



(a) TM Set 1

(b) TM Set 2

(c) TM Set 3

(d) TM Set 4

Figure V.1: Power saving ratios that were obtained by applying the proposed optimal topology composition algorithms (heuristic was used) for the four traffic matrix data sets. A(i, j) means an augmented optimal topology with i aggregate switches per pod and j core switches. The power costs of each switch and link were assumed as 150 and 1.

with traffic demands that were randomly selected in between 1 to 5% of a maximum link capacity.

5.2 Power Saving Ratio

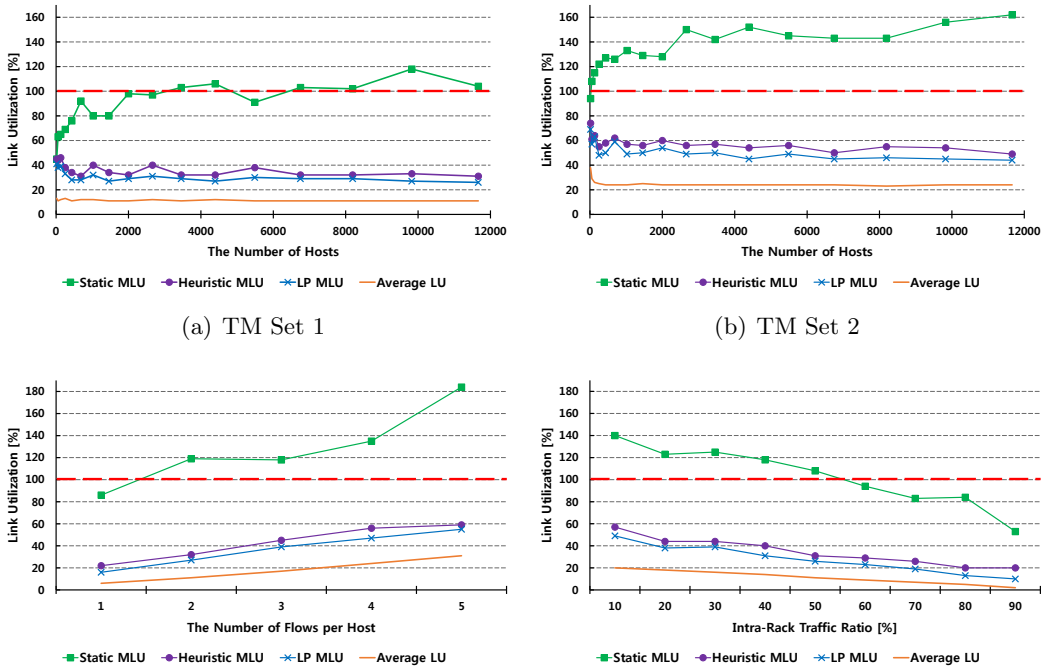
We calculated power saving ratios when we applied the proposed optimal topology composition approach for reducing power consumptions of a DCN (see Figure V.1). We used the heuristic algorithm (Algorithm III.1) for this figure and assumed that

Table V.2: Average power saving ratios [%] of the proposed optimal topology composition algorithms calculated using the data from Figure V.1.

Category		Set 1	Set 2	Set 3	Set 4	All
Minimum	**mean**	44.5	32.8	42.0	47.4	**40.6**
	s.d.	3.7	5.9	7.9	5.8	**11.0**
A(1, k/8)	**mean**	37.2	26.4	38.8	44.4	**34.9**
	s.d.	9.3	9.8	8.2	5.8	**8.7**
A(2, 2k/8)	**mean**	32.0	22.2	36.0	41.7	**30.8**
	s.d.	11.3	9.6	7.9	5.9	**7.9**
A(3, 3k/8)	**mean**	27.4	19.0	33.4	38.9	**27.2**
	s.d.	12.2	9.0	8.4	5.9	**7.9**

the power costs of each switch and link were 150 and 1, respectively. We also calculated power saving ratios after adding extra i aggregate switches per pod and j core switches to the found optimal subset topology (represented as A(i, j)).

We can identify four trends of the power saving ratio of an optimal DCN topology from the Figure V.1. First, the power saving ratio was increased as the size of the DCN grows. Second, the saving ratio was decreased as the number of flows per host increases. Third, the ratio was increased as the intra-rack traffic ratio increases. Fourth and the last, the gap of the power saving ratio between an optimal and an augmented topology was decreased as the size of the DCN grows. Table V.2 summarizes the power saving ratios shown in Figure V.1.



(a) TM Set 1

(b) TM Set 2

(c) TM Set 3

(d) TM Set 4

Figure V.2: Maximum link utilization comparisons of a static routing scheme (default Fat-Tree) and the proposed traffic load balancing algorithms (both Linear Programming (LP)- and heuristic-based) against average link utilization for the four traffic matrix data sets applied on an entire DCN topology.

5.3 Traffic Load Balancing on Entire DCN Topology

We measured Maximum Link Utilization (MLU) of both aggregate and core links[1] to compare performances of the default Fat-Tree static routing and our dynamic traffic load balancing scheme applied on an entire DCN topology (see Figure V.2). Note that the static routing scheme of Fat-Tree cannot be applied on an optimal subset topology because it was designed to make use of an entire Fat-Tree topology to distribute traffic load. We also plotted an average Link Utilization (LU) of aggregate

[1]We excluded edge links because there is only one link between an end host and an edge switch.

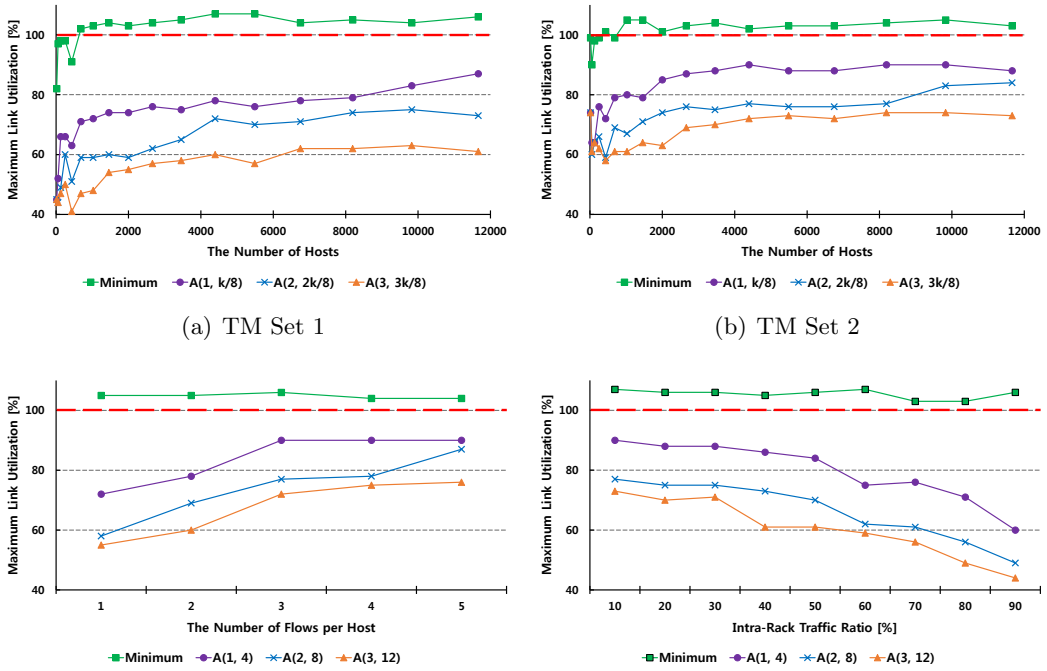
Table V.3: Average maximum link utilizations [%] of the proposed traffic load balancing algorithms calculated using the data from Figure V.2.

Category		Set 1	Set 2	Set 3	Set 4	All
Static Routing	**mean**	87.8	133.8	128.4	103.1	**111.2**
	s.d.	19.3	17.9	35.8	27.0	**22.7**
Heuristic TLB	**mean**	36.1	57.5	42.8	34.6	**44.1**
	s.d.	5.2	5.7	15.7	12.5	**8.1**
LP TLB	**mean**	30.8	51.3	36.8	27.6	**38.1**
	s.d.	4.5	7.0	15.6	12.9	**6.5**

and core links in the Figure as a base line.

From the Figure V.2, we can identify that the MLU of the static routing scheme was increased 1) as the size of the DCN grows, 2) as the number of flows per host increases, and 3) as the intra-rack traffic ratio decreases. On the other hand, our traffic load balancing schemes (using both path-based MCF, i.e. Linear Programming (LP), and a heuristic) significantly reduced the MLU in comparison with the static scheme. Moreover, if we see the gaps between average LUs and MLUs, we can identify that our traffic load balancing schemes stably maintained the MLUs as low as possible against the average LUs, whereas the static scheme failed to do that.

Table V.3 summarizes the measurement results of MLU shown in Figure V.2. According to this Table, the MLU reduction ratios of our traffic load balancing algorithms using a heuristic and LP against the static routing scheme were about 60.3% and 65.7% on average, respectively. The performance gap between the LP-based optimal solutions and the heuristic solutions was about 6.0% on average.



(a) TM Set 1

(b) TM Set 2

(c) TM Set 3

(d) TM Set 4

Figure V.3: Maximum link utilizations that were measured by applying the proposed traffic load balancing algorithms (heuristic was used) for the four traffic matrix data sets applied on an optimal topology obtained by executing the optimal topology composition algorithms (heuristic was used). A(i, j) means an augmented optimal topology with i aggregate switches per pod and j core switches.

5.4 Traffic Load Balancing on Optimal DCN Topology

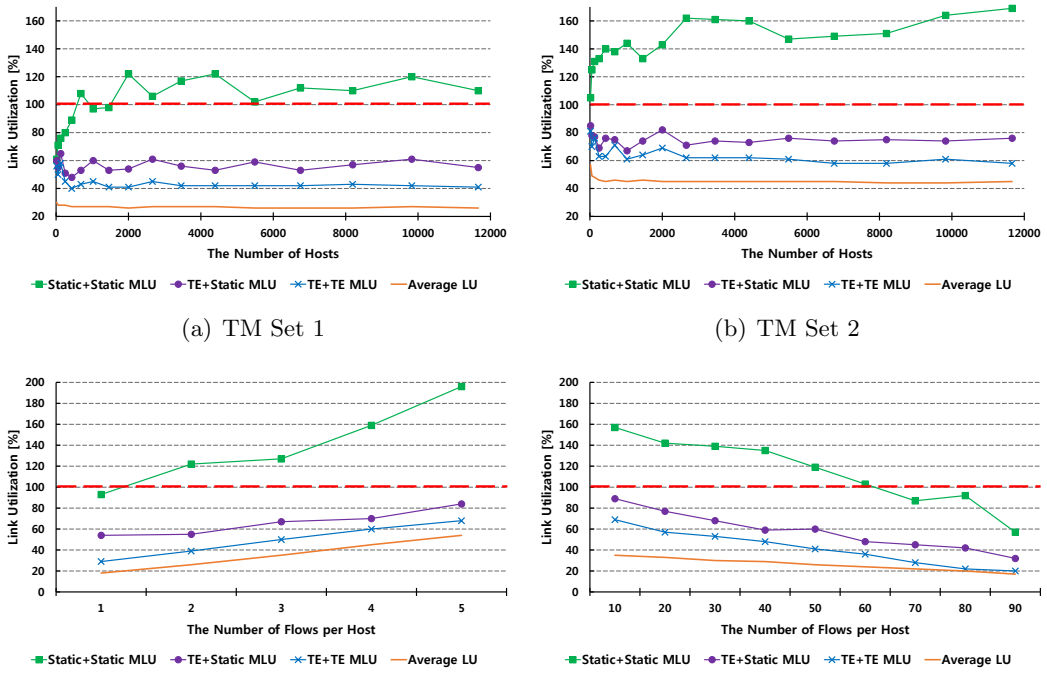
The proposed minimum subset topology composition algorithms try to find the subset topology with the minimum number of links and switches. Therefore, the probability that these subset links would experience traffic congestions is very high. We can diminish this probability of congestions by augmenting the found minimum topology with extra switches and links, and then re-distributing the traffic demands over the augmented optimal topology using our traffic load balancing algorithms.

Table V.4: Average maximum link utilizations [%] of the proposed traffic load balancing algorithms calculated using the data from Figure V.3.

Category		Set 1	Set 2	Set 3	Set 4	All
Minimum	**mean**	101.2	101.4	104.8	105.4	**102.4**
	s.d.	6.5	3.7	0.8	1.5	**25.3**
A(1, k/8)	**mean**	71.3	81.3	84.0	79.8	**77.8**
	s.d.	10.6	8.8	8.5	10.0	**17.6**
A(2, 2k/8)	**mean**	61.6	72.2	73.8	66.4	**67.6**
	s.d.	10.1	7.2	10.9	9.9	**15.9**
A(3, 3k/8)	**mean**	53.6	67.3	67.6	60.4	**61.2**
	s.d.	7.2	5.7	9.5	9.9	**14.0**

We measured MLU of both aggregate and core links to evaluate performances of the proposed traffic load balancing algorithms applied on both the minimum and the augmented optimal topologies (see Figure V.3). We can identify that the MLUs were higher than 100% in general when we applied the traffic load balancing algorithms on the minimum topology, whereas the MLUs on the augmented topologies were lower than 90% in most cases.

Table V.4 summarizes the measurement results of MLU shown in Figure V.3. According to this Table, the proposed traffic load balancing algorithms reduced the MLUs about 23.6% on average (from 102.4% to 78.8%) by adding only 1 aggregate per pod and $k/8$ core switches ($\texttt{A}(1, k/8)$). Moreover, they further reduced the MLUs about 34.8% ($\texttt{A}(2, 2k/8)$) and 41.2% ($\texttt{A}(3, 3k/8)$) by augmenting more switches. Considering this MLU and the power saving ratio of the augmented optimal topology are in a trade-off relationship, a network administrator should decide an appropriate



(a) TM Set 1

(b) TM Set 2

(c) TM Set 3

(d) TM Set 4

Figure V.4: Maximum link utilization comparisons among 1) a static routing scheme (Static+Static), 2) the proposed traffic load balancing algorithm without unpredicted traffic load balancing (TE+Static), and 3) the proposed traffic load balancing algorithm with unpredicted traffic load balancing (TE+TE) against average link utilization for the four traffic matrix data sets applied on an entire DCN topology.

level of augmentation.

5.5 Traffic Load Balancing with Unpredicted Traffic

The proposed *predicted* traffic load balancing algorithms allocate flows in an estimated traffic matrix, but they cannot deal with *unpredicted* flows that are not specified in the traffic matrix. Therefore, we proposed a heuristic algorithm to allocate those unpredicted flows in Section 3.4.4.

Table V.5: Average maximum link utilizations [%] of the proposed traffic load balancing algorithms calculated using the data with unpredicted traffic from Figure V.4.

Category		Set 1	Set 2	Set 3	Set 4	All
Static+Static	**mean**	100.1	144.4	139.4	114.6	**122.6**
	s.d.	18.7	16.4	39.4	32.2	**25.1**
TE+Static	**mean**	56.1	75.1	66.0	57.8	**64.2**
	s.d.	4.4	4.2	12.3	18.1	**13.7**
TE+TE	**mean**	44.6	64.7	49.2	41.6	**51.6**
	s.d.	5.2	6.5	15.7	16.7	**9.7**

Figure V.4 shows measurement results of MLUs to compare the dynamic unpredicted traffic load balancing (TE+TE) against static approaches (both Static+Static and TE+Static). The *Static+Static* represents a case where we applied the default Fat-Tree routing scheme for allocating both predicted and unpredicted flows. For the *TE+Static* case, we applied our dynamic traffic load balancing algorithms for predicted flows and the Fat-Tree static routing scheme for unpredicted flows. Lastly, the *TE+TE* represents a case where we applied our dynamic traffic load balancing algorithms for both predicted and unpredicted flows. Note that those static approaches (Static+Static and TE+Static) cannot be applied on a subset (minimum or augmented) topology because they were designed to make use of an entire DCN topology. We also plotted an average LU of aggregate and core links in the Figure as a base line.

Table V.5 summarizes the measurement results of MLU shown in Figure V.4. According to this Table, the MLU reduction ratio of our traffic load balancing algorithms with a static unpredicted traffic processing (TE+Static) against the pure

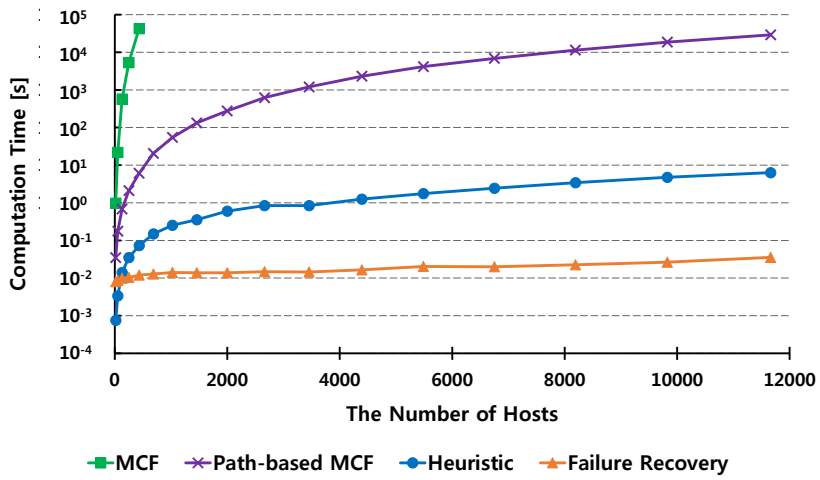


Figure V.5: Computation time comparisons of the three traffic load balancing algorithms (MCF, Path-based MCF, and a heuristic) for the data set 1 and path computation time of the failure recovery algorithm.

static routing scheme (Static+Static) was about 47.6% on average. The MLU reduction ratio of our traffic load balancing algorithms with a dynamic unpredicted traffic processing (TE+TE) against the pure static routing scheme (Static+Static) was about 57.9% on average. The performance gap between the hybrid (TE+Static) solutions and the dynamic (TE+TE) solutions was about 12.6% on average.

5.6 Computation Time

Figure V.5 compares computation times of the three traffic load balancing algorithms using Multi Commodity Flow (MCF), path-based MCF, and a heuristic applied to the data set 1. It also compares failure recovery time according to the increasing number of hosts. The traffic load balancing algorithm using MCF took more than 20 s even for a small DCN with 54 end hosts and took more than 40,000 s (about

11 h) for a DCN with 432 hosts. Accordingly, we can say that this MCF-based approach is not suitable for applying dynamic TE to a DCN.

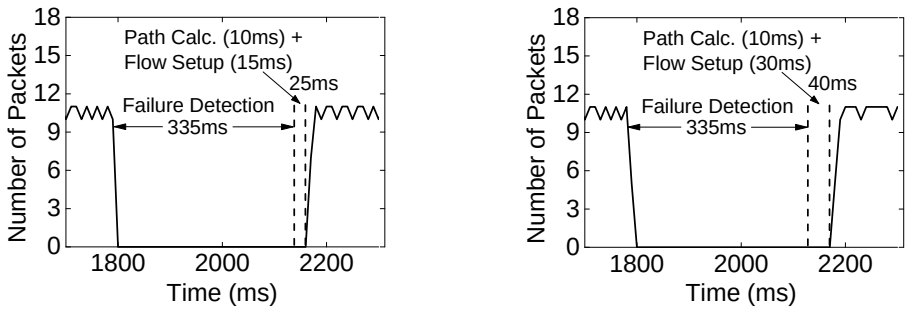
The proposed traffic load balancing algorithm using path-based MCF, on the contrary, took about 0.2 s for a small DCN with 54 hosts and took about 6 s for a DCN with 432 hosts. Therefore, this approach, which provides an optimal traffic load balancing solution, can be applied to a somewhat small DCN with less than 432 hosts.

The heuristic traffic load balancing algorithm took less than 1 s for a DCN with less than 4,394 hosts and took less than 7 s even for a large-scale DCN with more than 10,000 hosts. Even though this heuristic approach provides near-optimal solution (about 6% lower than the optimal) in terms of MLU, it provides good scalability; we can apply this heuristic for a large-scale DCN. Moreover, the computation time of the heuristic can be further reduced through optimization of the implementation code and employment of a high performance computing machine.

Lastly, this figure shows that our failure recovery method scales well and requires only about 36 ms to calculate alternative paths even for a large-scale DCN with 11,664 hosts

5.7 Failure Recovery Time

To measure the total recovery time for handling a single link failure, we artificially transferred UDP traffic from a sender to a receiver. Note that an agent was installed into the receiver to count the number of received packets, and the measurement granularity was configured at 10 ms. The current implementation of Mininet [30]



(a) Uplink traffic (b) Downlink traffic

Figure V.6: Recovery time for a single link failure of Fat-Tree with $k = 4$

can only run in a single machine; consequently, it does not scale well. Moreover, regardless of the number of hosts, the total amount of flow setup time is identical. Furthermore, the total recovery time is mainly affected by the path calculation time as the number of hosts increase. Thus we built a Fat-Tree topology with 16 virtual hosts and performed the experiment. Figure V.6 shows the total recovery time for a single link failure case with 16 hosts ($k = 4$). We exploited an existing topology discovery application of Floodlight to detect a link failure, and found that the time for detecting a link failure is relatively long (around 335 ms). With 16 hosts, we observed that the total path calculation time was around 10 ms, and each flow setup time contributed around 15 ms. As one more flow setup message was sent from the controller for resolving the link failure of downlink traffic compared to uplink traffic, 15 ms of additional time is required for the downlink traffic case.

Chapter VI

CONCLUSION

This chapter summarizes the overall contents of the thesis and lists a set of contributions this thesis has achieved. Furthermore, it discusses several research topics as future work.

6.1 Summary

In the Chapter I (introduction), we have provided a brief introduction to a Data Center Network (DCN), Software Defined Networking (SDN), and Traffic Engineering (TE). SDN-based TE provides many advantages including flexible routing management using software programs, fine-grained flow control using packet header fields match, and resilient failure recovery using global knowledge of networks. More specific benefits of SDN-based TE approaches for a DCN are that: 1) it is relatively easier to obtain traffic and failure information via a centralized SDN controller, 2) any flow format with arbitrary granularity can be exploited for TE, and 3) it is easy

to apply TE results to switches by modifying flow tables in the switches using APIs provided by the SDN controller.

We also have described the problems of current DCN technologies and outlined our solution approach that this thesis has taken to overcome the problems in the Chapter I. Current DCNs are suffering from high operational expenditure, frequent link congestions, and delayed recovery from link or switch failures. Considering these problems, this thesis has proposed a dynamic TE system for a DCN based on SDN technologies to improve energy efficiency and network utilization of the DCN.

In the Chapter II (related work), we have introduced a typical tree-based DCN topology as well as several state-of-the-art DCN topologies including Fat-Tree [1], VL2 [2], PortLand [3], DCell [4], BCube [5], and Jellyfish [6]. These newly proposed DCN topologies provide higher link capacity via path diversity, better fault tolerance, and better scalability than the conventional tree-based DCN topology.

After that, we have introduced SDN and an OpenFlow protocol. SDN is defined as "the physical separation of the network control plane from the forwarding plane, and where a control plane controls several devices" [7] and the OpenFlow protocol is a *de facto* standard for communications interface between SDN controllers and network devices. This migration of control into programmable computing devices enables the underlying network infrastructures to be abstracted for applications and network services. The programmable control plane also enables flexible and rapid modifications of network behavior.

Thereafter, we have introduced definitions and classifications of TE. TE is network engineering that deals with the issue of performance evaluation and performance optimization of operational IP networks [8]. It encompasses all the pro-

cesses of traffic measurement, characterization, modeling, and, most importantly, control [11, 12]. Various TE approaches can be classified according to four orthogonal categories: traffic type, traffic optimization scope, timescale of operations, and routing enforcement mechanisms. Our TE proposal has taken a unicast, intra-domain, online, and SDN-based approach in terms of those categories, respectively.

Finally, we have summarized existing TE techniques for a DCN, such as Hedera [20], microTE [18], PEFT [71], DLB [23], and ElasticTree [17]. We also have compared these existing approaches with our dynamic TE proposal: our approach tries to minimize both MLU and energy consumption at the same time, and provides a fast failure recovery mechanism that the others do not.

In the Chapter III (dynamic traffic engineering for DCN), we have presented our dynamic TE system for a DCN in detail. First, we have explained the overall system architecture of the TE system, which has three components: a DCN, an SDN controller, and a TE manager. The DCN is a target network of our traffic engineering system. The SDN controller collects traffic and failure status of the DCN from SDN switches in a centralized manner, then it aggregates and summarizes the collected data. The SDN controller also changes switching behavior of SDN switches by updating their flow tables, and turns on/off switches and links in the DCN to apply the traffic engineering decision that minimizes power consumptions and link congestions. The traffic engineering manager periodically takes traffic and failure information from the SDN controller, makes a traffic engineering decision using the information, and notifies the decision results to the SDN controller.

Thereafter, we have described detailed algorithms of the TE manager for optimal topology composition, traffic load balancing, and failure recovery using both linear

programming and a heuristic approaches. Optimal topology composition, the first procedure, finds a minimum subset of links and switches that can accommodate expected traffic demands at the moment. The next procedure is traffic load balancing that distributes ever-changing traffic demands over the found optimal topology to minimize MLU. Failure recovery, the last procedure, rapidly restores from network failures by setting detour paths to impacted DCN switches.

In the Chapter IV (prototype implementation), we have described how we implemented a prototype of the proposed dynamic TE system for a DCN using SDN technologies. For that, we have introduced Fat-Tree, the baseline DCN topology of the prototype implementation, and then explained implementation details of each component in the prototype. *Mininet* [30] network emulation tool was used for constructing a virtual DCN and *Floodlight* [31] SDN controller was used for managing the DCN. The APIs provided by Floodlight were used for applying outputs of the TE manager to the DCN.

Thereafter, we have explained Fat-Tree specific implementation algorithms: path acquisition and flow table setup. The path acquisition algorithm computes a set of equal cost shortest paths between a source and a destination hosts in a Fat-Tree topology. The flow table setup algorithm installs prefix and suffix flow entries into switches in a DCN for default Fat-Tree static routing. Lastly, we have provided demonstrations of the prototype using Floodlight Web UI with several useful scenarios including default Fat-Tree static routing, optimal topology composition, traffic load balancing, and failure recovery.

In the Chapter V (performance evaluation), we have provided performance evaluation results of the proposed TE system by means of simulations. It was intended

to complement the prototype implementation, which was emulated a virtual DCN using Mininet network emulator, for validating our proposal against a large-scale DCN. In this evaluation, we have used four different data sets of traffic matrix that were designed to cover as many cases as possible by varying related parameters including the k value of Fat-Tree, the number of flows, and intra-rack traffic ratio.

The evaluation results were summarized in terms of power saving ratio, Maximum Link Utilization (MLU), failure restoration delay, and algorithm computation time. According to the results, our optimal topology composition algorithms reduced power consumptions of a DCN about 41% on average. In addition, the proposed traffic load balancing algorithms reduced MLU about 66% on average in comparison with a static routing scheme.

6.2 Contribution

The followings are the key contributions of this thesis.

- An introduction of both the current and state-of-the-art DCN technologies and a diagnosis of their limitations
- A study on characteristics and advantages of SDN technologies for TE
- A dynamic TE system architecture for a DCN utilizing SDN technologies
- Algorithms for constructing an optimal subset DCN topology to reduce power consumptions of a DCN
- Algorithms for dynamically allocating flows to minimize Maximum Link Utilization (MLU)

- Algorithms for rapid restoration from network failures while avoiding possible traffic congestions
- A prototype implementation and performance evaluations of the proposed dynamic TE system architecture and algorithms

6.3 Discussion and Future Work

The proposed TE algorithms (especially, for optimal topology composition and traffic load balancing) in this thesis require an estimated Traffic Matrix (TM) as input. Performances of those algorithms are bound to the accuracy of the estimated TM. However, it is not a trivial task to obtain a TM with good accuracy. Fortunately, there exist several studies, of which we can take advantage, trying to exactly estimate a TM within a network.

For example, Medina *et al.* [87] compared three existing TM estimation techniques, which are linear programming, bayesian estimation, and expectation maximization. They also introduced a new TM estimation approach based on choice models that model Point Of Presence (POP) fanouts. Recently, Tootoonchian *et al.* [88] proposed an SDN-based TM estimation approach, which uses built-in features provided in OpenFlow switches to directly and accurately measure the TM with low overheads.

As future work, we will further improve those TM estimation techniques considering characteristics of a DCN. Possible research directions to that include a) exploiting data obtained from end hosts as well as data from switches for estimating TM [18] and b) estimating only elephant flows to reduce overheads [20].

Another issue is a scalability problem of a centralized SDN controller for collecting traffic and failure data and for applying TE results in a large-scale DCN. Even though research about this problem of SDN is at a preliminary stage, we can find some hints for mitigating this scalability issue from Onix [89] and Kandoo [90]. We will find an efficient way to collect traffic and failure data and to apply TE results from/to a large-scale DCN as future work.

Finally, we will deploy the proposed dynamic TE approach on a large-scale testbed, such as GENI [91] or SAVI [92], to investigate a real impact of our proposal on performances of a DCN.

요 약 문

클라우드 컴퓨팅 기술 발전, 대용량 멀티미디어 컨텐츠 증가, 빅 데이터 분석 기술 도입 등의 추세에 따라 기업, 정부 기관, 대학 등이 구축하고 있는 데이터 센터의 규모가 급격히 커지고 있으며, 데이터 센터 내부에 흐르는 트래픽 패턴도 데이터 센터 외부와의 통신이 주를 이루는 것에서 데이터 센터 내부 호스트들 간의 통신이 주를 이루는 것으로 양상이 변화하고 있다. 하지만, 현재 구축되어 있는 대부분의 데이터 센터 네트워크가 따르고 있는 계층적 트리 구조는 지속적으로 규모가 커지고 있는 데이터 센터를 효율적으로 구축하고 관리하기에는 비용 및 자원 사용 효율 측면에서 적합하지 않다.

기존 데이터 센터 네트워킹 기술들의 가장 큰 문제점은 세 가지로 요약할 수 있다. 첫째, 데이터 센터 네트워크를 구성하는 링크 및 스위치들의 사용률은 시간대에 따라서 변동이 심한 패턴을 보이고 있는데, 데이터 센터 네트워크가 소모하는 전력량은 네트워크 자원의 사용률에 관계없이 일정하여 데이터 센터의 운영 비용이 실제로 필요한 것보다 많이 요구되는 것이다. 둘째, 정적인 경로 선정 방식으로 인해 대다수의 네트워크 자원이 저조한 사용률을 보이는 동시에 일부 특정 링크 자원에는 트래픽이 몰려 혼잡이 발생하는 것이다. 마지막 문제점은 네트워크 장애가 발생했을 때 트래픽 상황을 고려한 신속한 장애 복구 방법이 부족하다는 것이다.

본 논문에서는 이러한 문제점들을 해결하기 위하여 최근 통신 시장의 이슈로 급부상한 소프트웨어 정의 네트워킹(Software Defined Networking, SDN) 기술을 활용한 동적인 트래픽 엔지니어링 시스템을 제안한다. SDN은 제어 평면과 데이터 평면의 분리 및 소프트웨어 프로그래밍을 통해 기존 네트워킹 기술로는 매우 어렵거나 불가능했던 유연한 네트워크 설정 및 관리를 가능하게 한다.

제안하는 트래픽 엔지니어링 시스템은 크게 세 단계로 구성되어 있다. 첫 단계는 예상되는 트래픽 요구 사항을 만족시킬 수 있는 최적의 데이터 센터 네트워크 부분 토폴로지를 구성하는 것이다. 구성된 부분 토폴로지에 포함되지 않는 링크 및 스위치들의 전원을 차단함으로써 데이터 센터 네트워크가 소모하는 전력을 절감할 수 있다. 다음 단계는 구성된 전력 절감 최적 토폴로지 내에서 수시로 변화하는 동적인 트래픽 요구 사항을 만족시키는 동시에 최대 링크 이용률(Maximum Link Utilization, MLU)을 최소화하도록 트래픽을 분배하여 각 링크에 할당하는 것이다. 이를 통해 추가적인 네트워크 자원 할당 없이도 더 많은 트래픽을 수용할 수 있기 때문에 네트워크 자원 이용 효율이 향상된다. 마지막 단계는 발생한 네트워크 장애를 동적인 트래픽 상황을 고려하여 신속히 복구하는 것이다. 신속한 장애 복구를 위해서는 변경해야하는 스위치의 수를 최소화해야 하며 이를 위해 본 논문에서는 최단 경로가 아닌 우회 경로를 통해 장애를 복구할 수 있는 방법을 제안한다.

제안한 방법을 검증하기 위해 Mininet 네트워크 에뮬레이션 툴을 이용해 Open vSwitch로 구성된 가상의 데이터 센터 네트워크를 구축하였고, 구축된 가상의 스위치들은 Floodlight SDN 컨트롤러를 통해 관리하도록 구성했다. 제안한 트래픽 엔지니어링 시스템은 트래픽 엔지니어링 결과를 반영하기 위하여 Floodlight의 API를 통해 링크 및 스위치들의 전원을 제어하고 스위치들의 플로우 테이블을 수정한다. 시뮬레이션 실험 결과 제안한 트래픽 엔지니어링 시스템을 적용하면 평균적으로 데이터 센터 네트워크가 소모하는 전력의 41% 정도를 절감할 수 있었으며, 최대 링크 이용률은 정적인 경로 선정 방식에 비해 66% 까지 낮출 수 있었다.

REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM '08*, (Seattle, USA), pp. 63–74, Aug. 17–22, 2008.

[2] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandular, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM '09*, (Barcelona, Spain), pp. 51–62, Aug. 17–21, 2009.

[3] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "PortLand: A scalable fault-tolerant layer 2 data center network fabric," in *Proc. ACM SIGCOMM '09*, (Barcelona, Spain), pp. 39–50, Aug. 17–21, 2009.

[4] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "DCell: A scalable and fault-tolerant network structure for data centers," in *Proc. ACM SIGCOMM '08*, (Seattle, USA), pp. 75–86, Aug. 17–22, 2008.

[5] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "BCube: A high performance, server-centric network architecture for modular data centers," in *Proc. ACM SIGCOMM '09*, (Barcelona, Spain), pp. 63–74, Aug. 17–21, 2009.

[6] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers randomly," in *Proc. 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, (San Jose, USA), pp. 1–14, Apr. 25–27, 2012.

[7] "Software-defined networking (SDN) definition." `https://www.opennetworking.org/sdn-resources/sdn-definition`. [Online; accessed Oct. 21, 2013].

[8] D. Awduche, A. Chiu, A. Elwalid, I. Widjaja, and X. Xiao, "Overview and principles of internet traffic engineering." RFC 3272, May 2002.

[9] E. Rosen, V. Viswanathan, and R. Callon, "Multiprotocol label switching architecture." RFC 3031, Jan. 2001.

[10] A. Elalid, C. Jin, S. Low, and I. Widjaja, "MATE: MPLS adaptive traffic engineering," in *Proc. 20th IEEE International Conference on Computer Communications (INFOCOM '01)*, (Anchorage, USA), pp. 1300–1309, Apr. 22–26, 2001.

[11] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. MacManus, "Requirements for traffic engineering over MPLS." RFC 2702, Sept. 1999.

[12] D. Awduche, "MPLS and traffic engineering in ip networks," *IEEE Communications Magazine*, vol. 37, no. 12, pp. 42–47, 1999.

[13] B. Fortz and M. Thorup, "Internet traffic engineering by optimizing OSPF weights," in *Proc. 19th IEEE International Conference on Computer Communications (INFOCOM '00)*, (Tel Aviv, Israel), pp. 519–528, Mar. 26–30, 2000.

[14] B. Fortz and M. Thorup, "Optimizng OSPF/IS-IS weights in a changing world," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 4, pp. 756–767, 2002.

[15] B. Fortz, J. Rexford, and M. Thorup, "Traffic engineering with traditional IP routing protocols," *IEEE Communications Magazine*, vol. 40, no. 10, pp. 118–124, 2002.

[16] B. Fortz and M. Thorup, "Increasing internet capacity using local search," *Computational Optimization and Applications*, vol. 29, no. 1, pp. 13–48, 2004.

[17] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "ElasticTree: Saving energy in data center networks," in *Proc. 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI '10)*, (San Jose, USA), pp. 1–16, Apr. 28–30, 2010.

[18] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *Proc. 7th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '11)*, (Tokyo, Japan), pp. 1–12, Dec. 6–9, 2011.

[19] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-Service: Load-balancing web traffic using OpenFlow," in *Proc. ACM SIGCOMM '09 Demo*, (Barcelona, Spain), Aug. 17–21, 2009.

[20] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI '10)*, (San Jose, USA), pp. 1–15, Apr. 28–30, 2010.

[21] R. Wang, D. Butnariu, and J. Rexford, "OpenFlow-based server load balancing gone wild," in *Proc. 1st USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE '11)*, (Boston, USA), pp. 1–6, Mar. 29, 2011.

[22] M. Koerner and O. Kao, "Multiple service load-balancing with OpenFlow," in *Proc. 13th IEEE Conference on High Performance Switching and Routing (HPSR '12)*, (Belgrade, Serbia), pp. 210–214, June 24–27, 2012.

[23] Y. Li and D. Pan, "OpenFlow based load balancing for Fat-Tree networks with multipath support," in *Proc. 12th IEEE International Conference on Communications (ICC '13)*, (Budapest, Hungary), pp. 1–5, June 9–13, 2013.

[24] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, and J. Zolla, "B4: Experience with a globally-deployed software defined wan," in *Proc. ACM SIGCOMM '13*, (Hong Kong, China), pp. 3–14, Aug. 12–16, 2013.

[25] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in *Proc. ACM SIGCOMM '13*, (Hong Kong, China), pp. 15–26, Aug. 12–16, 2013.

[26] R. Trestian, G.-M. Muntean, and K. Katrinis, "MiceTrap: Scalable traffic engineering of datacenter mice flows using OpenFlow," in *Proc. IFIP/IEEE International Symposium on Integrated Network Management (IM '13)*, (Ghent, Belgium), pp. 904–907, May 27–31, 2013.

[27] H. Long, Y. Shen, M. Guo, and F. Tang, "LABERIO: Dynamic load-balanced routing in OpenFlow-enabled networks," in *Proc. 27th IEEE International Conference on Advanced Information Networking and Applications (AINA '13)*, (Barcelona, Spain), pp. 291–297, Mar. 25–28, 2013.

[28] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. ACM Internet Measurement Conference 2010 (IMC '10)*, (Melbourne, Australia), pp. 267–280, Nov. 1–3, 2010.

[29] C. Hopps, "Analysis of an Equal-Cost Multi-Path algorithm." RFC 2992, Nov. 2000.

[30] "Mininet: An instant virtual network on your laptop (or other pc)." `http://mininet.org/`. [Online; accessed Nov. 27, 2013].

[31] "Floodlight OpenFlow controller." `http://www.projectfloodlight.org/floodlight/`. [Online; accessed Oct. 23, 2013].

[32] "Data center: Load balancing data center services SRND," tech. rep., Cisco Systems, Inc., San Jose, CA, USA, Mar. 2004.

[33] "Cisco data center infrastructure 2.5 design guide." `http://www.cisco.com/univercd/cc/td/doc/solution/dcidg21.pdf`. [Online; accessed Oct. 18, 2013].

[34] C. Clos, "A study of non-blocking switching networks," *Bell System Technical Journal*, vol. 32, no. 2, 1953.

[35] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, 2004.

[36] C. E. Leiserson, "Fat-Trees: Universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. 34, no. 10, pp. 892–901, 1985.

[37] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," in *Proc. ACM SIGCOMM '07*, (Kyoto, Japan), pp. 1–12, Aug. 27–31, 2007.

[38] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communications Review*, vol. 38, no. 2, pp. 69–119, 2008.

[39] R. Zhang-Shen and N. McKeown, "Designing a predictable internet backbone network," in *Proc. 3rd ACM Workshop on Hot Topics in Networks (HotNets '04)*, (San Diego, USA), pp. 1–6, Nov. 15–16, 2010.

[40] R. Zhang-Shen and N. McKeown, "Designing a predictable internet backbone network with valiant load-balancing," in *Proc. 13th International Conference on Quality of Service (IWQoS '05)*, vol. 3552 of *LNCS*, (Passau, Germany), pp. 178–192, June 21–23 2005.

[41] M. Kodialam, T. V. Lakshman, and S. Sengupta, "Efficient and robust routing of highly variable traffic," in *Proc. 3rd ACM Workshop on Hot Topics in Networks (HotNets '04)*, (San Diego, USA), pp. 1–6, Nov. 15–16, 2010.

[42] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," in *Proc. ACM SIGCOMM Workshop: Research on Enterprise Networking (WREN '09)*, (Barcelona, Spain), pp. 92–99, Aug. 21, 2009.

[43] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of datacenter traffic: Measurement & analysis," in *Proc. ACM Internet Measurement Conference 2009 (IMC '09)*, (Chicago, USA), pp. 202–208, Nov. 4–6, 2009.

[44] Y. Chen, S. Jain, V. K. Adhikari, Z.-L. Zhang, and K. Xu, "A first look at inter-data center traffic characteristics via Yahoo! datasets," in *Proc. 30th IEEE International Conference on Computer Communications (INFOCOM '11)*, (Shanghai, China), pp. 1620–1628, Apr. 10–15, 2011.

[45] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proc. ACM SIGCOMM '11*, (Toronto, Canada), pp. 350–361, Aug. 15–19, 2011.

[46] "Software-Defined Networking: The new norm for networks." White Paper, Open Networking Foundation, Apr. 13, 2012.

[47] "Open Networking Foundation." `https://www.opennetworking.org/`. [Online; accessed Oct. 22, 2013].

[48] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th USEIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, (San Francisco, USA), pp. 137–149, Dec. 6–8, 2004.

[49] "MuL - OpenFlow controller." `http://sourceforge.net/projects/mul/`. [Online; accessed Oct. 23, 2013].

[50] "Trema: Full-stack OpenFlow framework in ruby and c." `http://trema.github.io/trema/`. [Online; accessed Oct. 23, 2013].

[51] "About NOX." `http://www.noxrepo.org/nox/about-nox/`. [Online; accessed Oct. 23, 2013].

[52] "Lithium: Event-driven control for software-defined networks." `http://projectbismark.net/lithium/`. [Online; accessed Oct. 23, 2013].

[53] "About POX." http://www.noxrepo.org/pox/about-pox/. [Online; accessed Oct. 23, 2013].

[54] "Build SDN agilely: Ryu, a component-based software-defined networking framework." http://osrg.github.io/ryu/. [Online; accessed Oct. 23, 2013].

[55] "frenetic - a network programming language." http://www.frenetic-lang.org/index.php. [Online; accessed Oct. 23, 2013].

[56] "Beacon." https://openflow.stanford.edu/display/Beacon/Home. [Online; accessed Oct. 23, 2013].

[57] "maestro-platform - a scalable control platform written in java which supports OpenFlow switches." https://code.google.com/p/maestro-platform/. [Online; accessed Oct. 23, 2013].

[58] "OpenFlow switch specification version 1.0.0." Open Networking Foundation, Dec. 31, 2009.

[59] "OpenFlow switch specification version 1.1.0." Open Networking Foundation, Feb. 28, 2011.

[60] "OpenFlow switch specification version 1.2.0." Open Networking Foundation, Dec. 5, 2011.

[61] "OpenFlow switch specification version 1.3.0." Open Networking Foundation, June 25, 2012.

[62] "OpenFlow switch specification version 1.4.0." Open Networking Foundation, Oct. 14, 2013.

[63] Y. Lee and B. Mukherjee, "Traffic engineering in next-generation optical networks," *IEEE Communications Surveys and Tutorials*, vol. 6, no. 3, pp. 16–33, 2004.

[64] N. Wang, K. H. Ho, G. Pavlou, and M. Howarth, "An overview of routing optimization for Internet traffic engineering," *IEEE Communications Surveys and Tutorials*, vol. 10, no. 1, pp. 36–56, 2008.

[65] F. K. Hwang, D. S. Richards, and P. Winter, *The Steiner Tree Problem (Annals of Discrete Mathematics 53)*. North-Holland, 1992.

[66] R. Teixeira, A. Shaikh, T. Griffin, and G. M. Voelker, "Network sensitivity to hot-potato disruptions," in *Proc. ACM SIGCOMM '04*, (Portland, USA), pp. 231–244, Aug. 30–Sep. 3, 2004.

[67] P. Trimintzios, I. Andrikopoulos, G. Pavlous, P. Flegkas, D. Griffin, P. Georgatsos, D. Goderis, Y. T'Joens, L. Georgiadis, C. Jacquenet, and R. Egan, "A management and control architecture for providing IP differentiated services in MPLS-based networks," *IEEE Communications Magazine*, vol. 39, pp. 80–88, May 2001.

[68] B. Quoitin, C. Pelsser, L. Swinnen, O. Bonavernture, and S. Uhlig, "Interdomain traffic engineering with BGP," *IEEE Communications Magazine*, vol. 41, no. 5, pp. 122–128, 2003.

[69] H. Wang, H. Xie, L. Qiu, Y. R. Yang, Y. Zhang, and A. Greenberg, "COPE: Traffic engineering in dynamic networks," in *Proc. ACM SIGCOMM '06*, (Pisa, Italy), pp. 99–110, Sept. 11–15, 2006.

[70] D. Xu, M. Chiang, and J. Rexford, "Link-state routing with hop-by-hop forwarding can achieve optimal traffic engineering," *IEEE/ACM Transactions on Networking*, vol. 19, pp. 1717–1730, Dec. 2011.

[71] F. P. Tso and D. P. Pezaros, "Improving data center network utilization using near-optimal traffic engineering," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, pp. 1139–1148, June 2013.

[72] H. T. Viet, Y. Deville, O. Bonaventure, and P. Fancois, "Traffic engineering for multiple spanning tree protocol in large data centers," in *Proc. 23rd International Teletraffic Congress (ITC '11)*, (San Francisco, USA), pp. 23–30, Sept. 6–9, 2011.

[73] "Linear programming." `http://en.wikipedia.org/wiki/Linear_programming`. [Online; accessed Nov. 19, 2013].

[74] "Heuristic (computer science)." `http://en.wikipedia.org/wiki/Heuristic_(computer_science)`. [Online; accessed Nov. 19, 2013].

[75] T. C. Hu, "Multi-commodity network flows," *Operations Research*, vol. 11, no. 3, pp. 344–360, 1963.

[76] "Multi-commodity flow problem." `http://en.wikipedia.org/wiki/Multi-commodity_flow_problem`. [Online; accessed Nov. 19, 2013].

[77] S. Even, A. Itai, and A. Shamir, "On the complexity of time table and multi-commodity flow problems," in *Proc. 16th Annual Symposium on Foundations of Computer Science*, (USA), pp. 184–193, Oct. 13–15, 1975.

[78] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, pp. 788–789. MIT Press and McGraw-Hill, second ed., 2001.

[79] S. Kandula, D. Katabi, S. Sinha, and A. Berger, "Dynamic load balancing without packet reordering," *ACM SIGCOMM Computer Communications Review*, vol. 37, no. 2, pp. 51–62, 2007.

[80] "Open vSwitch: An open virtual switch." `http://openvswitch.org/`. [Online; accessed Dec. 2, 2013].

[81] "Iperf." `http://en.wikipedia.org/wiki/Iperf`. [Online; accessed Dec. 2, 2013].

[82] "puLP: An LP modeler in python." `https://code.google.com/p/pulp-or/`. [Online; accessed Dec. 2, 2013].

[83] "GLPK (GNU linear programming kit)." `http://www.gnu.org/software/glpk/glpk.html`. [Online; accessed Dec. 2, 2013].

[84] "COIN-OR: Computational infrastructure for operations research." `http://www.coin-or.org/`. [Online; accessed Dec. 2, 2013].

[85] "CPLEX optimizer: High-performance mathematical programming solver for linear programming, mixed integer programming, and quadratic programming." `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/index.html`. [Online; accessed Dec. 2, 2013].

[86] "Gurobi - the overall fastest and best supported solver available." `http://www.gurobi.com/`. [Online; accessed Dec. 2, 2013].

[87] A. Medina, N. Taft, S. Bhattacharyya, and C. Diot, "Traffic matrix estimation: Existing techniques and new directions," in *Proc. ACM SIGCOMM '03*, (Pittsburgh, USA), pp. 161–174, Aug. 19–23, 2003.

[88] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "OpenTM: Traffic matrix estimator for OpenFlow networks," in *Proc. Passive and Active Measurement Conference (PAM '10)*, vol. 6302 of *LNCS*, (Zurich, Switzerland), pp. 201–210, Apr. 7–9 2010.

[89] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A distributed control platform for large-scale production networks," in *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, (Vancouver, Canada), pp. 1–14, Oct. 4–6, 2010.

[90] S. H. Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proc. 1st Workshop on Hot Topics in*

Software Defined Networking (HotSDN '12), (Helsinki, Finland), pp. 19–24, Aug. 13, 2012.

[91] "Geni - exploring networks of the future." `http://www.geni.net/`. [Online; accessed Dec. 6, 2013].

[92] "Smart applications on virtual infrastructure." `http://www.savinetwork.ca/`. [Online; accessed Dec. 6, 2013].

Acknowledgements
감사의 글

박사 학위 졸업 논문을 무사히 완성할 수 있도록 많은 도움을 주신 홍원기 교수님, 유재형 교수님 및 연구실 선후배님들께 깊은 감사의 말씀을 전하고 싶습니다. 홍원기 교수님께서 마련해 주신 좋은 연구 환경 덕택에 다양한 연구 주제를 접할 수 있었고, 연구를 위한 기본적인 소양을 갖출 수 있었습니다. 그 후 실무 경험이 풍부하신 유재형 교수님의 지도하에 본 학위 논문의 주제를 잡고 세부 내용을 완성할 수 있었습니다. 다시 한 번 진심으로 감사 드립니다.

또한, 연구실 선배님들이신 강준명, 원영준, 홍성철 박사님들로 부터 많은 가르침을 받았습니다. 특히, 준명이 형은 CMEST 과제를 같이 하면서 제 대학원 생활의 대부분을 함께했고 연구는 물론이고 삶을 대하는 태도에 대해서도 많이 배울 수 있었습니다. 영준이 형은 연구는 많이 함께하지 못했지만, 연구 외적인 생활에서 정말 모범이 되어 주셨고 항상 의지가되고 도움이 되는 말씀을 많이 해주셨습니다. 연구실 선배이자 친구인 병철이와 성수 덕분에 연구실에 잘 적응할 수 있었고 즐거운 연구실 생활을 이어나갈 수 있었습니다. 바로 밑 후배로 들어와서 궂은 일을 도맡아 해준 재윤이에게도 고마운 마음을 전하고 싶습니다. 그리고, 이 학위 논문을 완성하는 데 있어서 너무나도 큰 도움을 준 건, 윤선, 태열, 종환이에게 큰 신세를 졌습니다. 도움 주신 모든 분들 잊지 않겠습니다. 감사합니다.

마지막으로 캐나다에서 운명적으로 만나 지금까지 사랑을 키워왔고 앞으로도 함께 할 여자 친구 혜진이와 지금의 제가 존재할 수 있도록 낳고 길러 주신 사랑하는 저의 할아버지, 할머니, 부모님께 이 논문을 바칩니다.


POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교