Abstract

Traffic Planning under Network Dynamics

Hongqiang Liu

2014

Networks are constantly dynamic because of various types of faults (*e.g.* link failures) and maintenance (*e.g.* device upgrades). Such network dynamics could result in severe congestion which significantly undermines the performance of latency sensitive applications, such as search engines, online retail, online games, live streaming and so forth.

This dissertation asks a fundamental question – how does one efficiently protect latency sensitive applications against network dynamics? The solution it proposes, reasons out, and realizes is *to plan traffic strategically in networks and applications*.

Specifically, this dissertation provides systematic understandings on the motivations and methodologies of strategic traffic planning. First, for network providers, it shows why they should, and how they could, spread traffic over their networks tactically so that spare capacity in the networks can be efficiently utilized to accommodate rerouted packets and traffic spikes under network dynamics. Second, for application owners, it shows why they should, and how they could, allocate traffic loads among multiple infrastructures (*e.g.* clouds) adaptively to boost their robustness to uncertainties within individual infrastructures. Moreover, this dissertation designs, implements, and evaluates practical algorithms and systems based on the preceding understandings to achieve strategic traffic planning at different levels: intra-infrastructure (intra- and inter-datacenter) and inter-infrastructure.

Around the theme of strategic traffic planning, this dissertation makes the following three main contributions. First, to handle the network dynamics caused by faults, we propose and practically realize a concept – Forward Fault Correction (FFC) – which requires

a traffic engineering (TE) that guarantees no congestion without reconfiguring the network as long as the number of faults is under k. The challenges to realize FFC lie in the overhead in network throughput and the computational complexity to prepare for a huge number of fault cases. We develop an efficient and uniform method to obtain a TE with FFC under diverse kinds of faults on both control- and data-plane.

Next, to make sure the network dynamics aroused by maintenance cause no harm, we introduce a concept – Smooth traffic Distribution Transition (SDT) – which means that a network is configured in a congestion-free way to achieve some traffic distribution that is needed by specific maintenance. For example, before rebooting a switch, an operator will drain the traffic on switch first. SDT provides a common functionality that is needed during various types of network maintenance. The key challenge to realize SDT stems from the inherent difficulty in synchronizing the changes to many devices, which may lead to unforeseen transient link load spikes or even congestion. We present one primitive, `zUpdate`, which performs SDT via a multi-step and progressive network re-configuration scheduling.

While FFC and SDT are designed for avoiding congestion on network links, congestion inside overloaded servers is also crucial especially for applications that are both traffic intensive and latency sensitive, *e.g.* live streaming and video on demand. Therefore, we finally present a framework – Content Multihoming Optimization (CMO) – which uses multiple infrastructures, *e.g.* clouds and/or content delivery networks (CDNs), and adaptive download scheduling among these infrastructures from client-side to protect users' quality of experience (QoE) against performance fluctuations in individual infrastructures. The key challenge to realize CMO resides in the expensive and complex usage prices of infrastructures and the overhead of establishing connections to multiple servers. We show how these applications can use joint optimizations and algorithms with both global and local views to minimize the cost for utilizing multiple infrastructures and reduce overhead.

Traffic Planning under Network Dynamics

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Hongqiang Liu

Dissertation Director: David H. Gelernter

December, 2014

To my parents, my brother and my wife

Acknowledgements

The last four years at Yale were an incredible journey in my life. Along this journey, I encountered dilemmas and breakthroughs, depressions and endeavors, explorations and discoveries, and failures and successes. After all, I did not only become stronger in ability and mind, but also got to know many brilliant people who will be my friends for life.

I could never have been more fortunate to have Prof. David Gelernter as my advisor. David's trust in me and in my research efforts is the spring of my confidence and dedication to accomplish this dissertation. His acknowledgements and suggestions on my work and my capabilities always motivate me to become a better researcher and person. More importantly, from David I learnt that a real scholar should be wise, optimistic, outstanding and brave not only for research but also for the rest of life. I wish I could preserve these qualities for my life and pass them onto my students and children.

I also had a wonderful experience working with Prof. Bryan Ford and Dr. David Wolinsky. They are superb system researchers and hackers from whom I learnt much on how to think about ideas from system and security angles. I also thank them for sitting through countless practice talks and giving me valuable advice. David also spent a lot of time teaching me how to improve my written English.

I will always appreciate the academic and financial support from Prof. Yang Richard Yang in my first two years at Yale. He was the first one who told me I had potential to become a good scholar, even though then I was only a fresh student who did not know

how to proceed. He helped me to get rid of my bad habits in thinking and working and led me through the gate of research. It was an unforgettable experience that I could work with Richard and his students Ye Wang, Chen Tian and Hao Wang side by side, day and night to submit the CMO paper to SIGCOMM'12. Without their help, I had no way to push this project into SIGCOMM in my second year.

I spent two happy and productive internships in Microsoft Research. In the summer of 2012, under the mentoring of Ming Zhang and with the help from Lihua Yuan, Roger Wattenhofer and Dave Maltz, I finished the zUpdate project and got it published in SIGCOMM'13. Ming's high standards on research and presentation quality let me know the direction to become a first-class Ph.D. student. I also spent half a year in 2013 at Microsoft Research with Ratul Mahajan as my mentor. I finished the FFC project which I am proud of the most by now. Ratul and I experienced a long and arduous process from an initial idea which seems to be infeasible to a practical concept and a systematic algorithm design. Ratul often gave me a golden suggestion at each key turning point of this project, and discussions with him were consistently enjoyable and fruitful.

I am so lucky to have Prof Vladimir Rokhlin and Prof Holly Rushmeier as my DGS and Department Head. I had the great pleasure of sharing my journey through Yale graduate school with Dongqu Chen, Ronghui Gu, Jiewen Huang, Michael Nowlan, Ewa Syta, Xueyuan Su, Minghui Tan, Huan Wang, Weiyi Wu, Hongzhi Wu, Xiongnan Wu, Su Xue, and Ennan Zhai. I am also honored to have collaborations with scholars outside Yale: Xin Jin, Peng Sun and Prof. Jenifer Rexford from Princeton, Xin Wu from Duke, Rohan Gandhi and Prof. Y. Charlie Hu from Purdue, Hongyi James Zeng from Stanford.

Finally, I would like to express my earnest gratitude to my parents who made numerous sacrifices to raise and educate me. Also, I could not have my current achievements without my wife Wei's persistent support and love. This thesis is dedicated to them.

Previously Published Material

Chapter 2 revises a previous publication [59]: H. H. Liu, D. Gelernter, S. Kandula, R. Mahajan, M. Zhang. Traffic Engineering with Forward Fault Correction,
in *Proc. ACM SIGCOMM*, 2014.

Chapter 3 revises a previous publication [61]: H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, D. Maltz. zUpdate: Updating Data Center Networks with Zero Loss,
in *Proc. ACM SIGCOMM*, 2013.

Chapter 4 revises a previous publication [60]: H. H. Liu, Y. Wang, Y. R. Yang, H. Wang, C. Tian. Optimizing Cost and Performance for Content Multihoming,
in *Proc. ACM SIGCOMM*, 2012.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"*There is nothing permanent except change.* "

——— Heraclitus

Latency sensitive, cloud-based applications, *e.g.* search engines, online retail, advertisements, online games, live streaming, *etc.*, play an increasingly important role in our lives. Studies show that the traffic from such applications will increase by multiple orders of magnitudes over the next few years [52]. This trend is driven by the IT reforms being undertaken in traditional industries and markets. It is also emphasized by users' ease of accessing Internet with various kinds of devices and networks.

An essential requirement from latency sensitive applications is to guarantee stable and short delays in the interactions between users and backend servers. For instance, web search and online shopping typically require an interaction latency that is within 200-300 ms [75, 83], and missing this soft-deadline could cause substantial loss in users and service revenue [79]. For another example, even though live video streaming is not sensitive to occasional packet delays, large jitters in packet arrival latency could result in a large playing buffer that magnifies the time gap between videos and real events.

Nevertheless, it is a big challenge to satisfy the delay requirement from latency sen-

sitive applications. One of the major reasons is that communication delay is difficult to ensure, since networks are constantly dynamic because of various types of faults (*e.g.* link failures) and maintenance (*e.g.* device upgrades). Such network dynamics can easily cause congestion which ultimately results in significant packet in-network queuing delays or even packet drops. Therefore, nowadays network providers are forced to over-provision their networks, which results a low (30-40%) bandwidth resource utilization on average [42, 44] and still cannot provide any congestion-free guarantee. Despite of some efforts like prioritizing latency sensitive traffic in networks [42, 44], the protection to the application performance is still far from satisfactions [59].

This dissertation conducts a systematic study on how to prevent congestion and application performance degradation caused by network dynamics. It provides two foundational insights:

- If network providers carefully spread traffic over networks, they can efficiently utilize the spare capacity existing in the networks to accommodate rerouted packets and traffic spikes under network dynamics. As a result, congestion can be avoided.

- In applications which are not strictly sensitive but still impacted by long interaction delays (*e.g.* content services), users' quality of experience (QoE) could be further enhanced if the applications adaptively utilize multiple infrastructures. As a result, the impact to QoE from performance fluctuations in individual infrastructures can be minimized.

Overall, this dissertation aims to achieve *strategic traffic planning* which protects latency sensitive applications from network dynamics at different levels – intra-infrastructure (intra- and inter-datacenter) level performed by network providers, and inter-infrastructure level performed by application owners.

1.1 Strategic Traffic Planning

Around the theme of *strategic traffic planning*, this dissertation includes mainly three pieces of work which are summarized in the following of this section:

- Traffic engineering which is robust to network dynamics caused by faults.
- Congestion-free network re-configurations for network maintenance.
- Traffic load allocation and adaptive download scheduling in content services.

1.1.1 Proactive fault handling in traffic engineering

One cause of network dynamics is faults that happen commonly in production networks. Such faults include not only failures on data-plane, such as link failures and switch failures, but also failures on control-plane like long delays or failures to configure some network devices.

These faults significantly limit network providers' ability to protect traffic on networks, because they make both the input and output of traffic engineering (TE) systems inaccurate. On one hand, network providers need current network topology to figure out a TE solution which plans the traffic to avoid congestion and makes the network highly utilized. However, frequent data-plane faults usually lead to mismatch between TE solution and network topology, which creates congestion.

On the other hand, all TE solutions need to be implemented by configuring switches to update routing rules. However, because of the existence of control-plane faults, new routing rules might not be able to installed in all of the switches successfully at once. Such partial TE implementation or inconsistent configurations in network can also trigger severe congestion.

The biggest challenge to handle the preceding faults is that it is difficult to predict

where individual faults will happen, despite there are likely to be any because faults are generally prevalent. A straightforward way to address this challenge is that instead of predicting specific faults, the network only reacts when some faults happen via computing and installing a new TE which fixes existing congestion over networks. Nonetheless, such reactive approach to handling faults is not efficient for mainly three reasons:

- Reactions always happen after faults and congestion. How much a network suffers from congestion depends on how fast a reaction can be finished.
- It could take a long time to finish a reaction. It takes time to detect faults, to compute a new TE and to update switch configurations to realize the new TE. Overall it could take from seconds to minutes to accomplish a reaction in a large-scale production network.
- Control-plane faults could block a reaction. If configuration failures happen on some switches during the installation of the new TE, a reaction could be postponed, and so that the existing congestion could not be eliminated.

Therefore, we argue that it is necessary for network providers to have a proactive approach to handling common network faults.

Network providers typically update the TE in their networks periodically to adapt to changes in traffic demands. *Proactive approach* means that each time network providers compute a new TE for a network, they require the TE is not only congestion-free in current situation, but also robust to fault cases that could happen in the following TE update interval. Hence, there is no need to update TE within a TE update interval due to faults, because current TE can eliminate congestion automatically.

To make a TE robust to most potential fault cases that can happen in a TE update interval, this dissertation proposes and practically realizes the concept **Forward Fault Correction (FFC)** which requires a traffic engineering (TE) that guarantees congestion-free

without reconfiguring the network if only the number of faults is under k. This guarantee should hold for arbitrary combinations of faults. Since there are multiple categories of faults, we have a parameter k for each type of faults. For instance, a FFC-TE should be robust to k_c configuration failures, k_e link failures and k_v switch failures. The definition of FFC is from the intuition that the probability of a single fault is small, so that the total number of faults in a typical TE update interval (2-5 minutes) should be statistically small and stable. By picking a proper parameter k, a TE satisfying FFC can be robust to almost all potential faults that can happen before next TE update.

There are two challenges to realize FFC in TE. First, in a number of fault cases a FFC-TE needs to be prepared for could be huge – with n possible faults, the number of combinations up to k faults is $\sum_{i=1}^{k} \binom{n}{i}$. Including all possible fault cases in TE formulations will make the computation of FFC-TE intractable. Second, the loss of network throughput due to FFC robustness should be minimized. Essentially, the robustness of FFC comes from the spare capacity left in the network which can accommodate re-routed traffic during faults. FFC should consume spare capacity as little as possible for a given robustness level (k), so that FFC's overhead on the network throughput can be insignificant or negligible.

In Chapter 2, we elaborate how we develop and evaluate an efficient and uniform method to obtain a TE with FFC under diverse kinds of faults on both control- and data-plane. The key idea is that we can transfer the original FFC constraints for each type of faults into a uniform "bounded M-sum" problem: the sum of any M out of N variables is bounded. Finally, we develop a method based on sorting networks [19], to efficiently encode this problem as $O(kn)$ linear constraints.

Fundamentally, FFC makes available a novel control knob to network operators. Today, operators must either conservatively over-provision the network to guarantee the absence of congestion when faults occur or aggressively utilize the network [42, 44] at the

risk of severe fault-induced congestion. FFC enables operating points that trade-off provisioning and congestion-risk in an informed manned, based on network characteristics and desired protection levels for traffic.

1.1.2 Congestion avoidance in network maintenance operations

Besides faults, another major cause of network dynamics is maintenance. There are various types of maintenance being performed every day in networks. For instance, in a data center network, network operators usually need to upgrade firmware of all switches. In addition, to on-board a newly built network, network operators need to conduct existing traffic in old network onto the new one. In production datacenter networks, there are even more types of maintenance such as load balancer reconfigurations, virtual machine migrations, and so forth.

Each of such maintenance activities changes the distribution of traffic over the network in its own way. Typically, a sophisticated operation plan is needed if we do not want to corrupt latency sensitive applications running on top of the network during maintenance. For example, before rebooting a switch for firmware upgrade, a network operator wants this switch to carry no traffic. Therefore, beforehand he configures the routing in the network to conduct all traffic flows to go through paths which do not contain the switch. This configuration can involve many switches because moving one flow usually needs the coordination from more flows to make sure at the end the network is congestion-free.

After studying common maintenance activities in datacenter networks, we find that they all need to configure the network beforehand to achieve a traffic distribution which satisfies their specific requirements. According to this observation, we introduce the concept **Smooth traffic Distribution Transition (SDT)** which means that the preceding configuration process is realized quickly without any congestion. SDT provides a common functionality that is needed during diverse types of network maintenances.

The key challenge to realize SDT is from the inherent difficulty in synchronizing the changes to many devices, which may lead to unforeseen transient link load spikes or even congestion. In Chapter 3, we present one primitive, `zUpdate`, which performs SDT via a multi-step and progressive network re-configuration scheduling. To perform a SDT with `zUpdate`, operators only need to describe the requirements to the target traffic distribution. These requirements can easily be converted into a set of input constraints to `zUpdate`. Then `zUpdate` will attempt to compute a sequence of steps to progressively meet the requirements from an initial traffic distribution. When such a sequence is found, `zUpdate` autonatically configures the network step by step to accomplish the traffic distribution transition smoothly and losslessly. Chapter 3 also presents implementation of `zUpdate` with practical limitations (*e.g.* limited switch table size, computation scalability, *etc.*), and extensive evaluations.

1.1.3 Adaptive download scheduling in content services

While FFC and SDT are useful to prevent congestion on network links, they are hardly helpful for handling congestion on overloaded servers. Such congestion in servers is especially crucial for traffic intensive and latency sensitive applications, *e.g.* content services like video on demand and live streaming. Most content services leverage public clouds and/or content delivery networks (CDNs) as their backend infrastructures. Therefore, fluctuations of server performance in the infrastructures directly impact users' quality of experience (QoE) in the content services.

We present a framework **Content Multihoming Optimization (CMO)** to enhance QoE of content services under the server performance dynamics in individual infrastructures. The key idea in CMO is that a content service provider selects multiple servers from one or more than one infrastructures for a single user, and the user performs a client-side adaptive download scheduling algorithm to download different content pieces in parallel

from the servers. This scheduling algorithm adjusts the downloading rates from different servers on-the-fly according to the servers' real-time performance, making sure that the total downloading rate of a user satisfies the service's requirement.

There are two key challenges to achieve CMO. First, how to design the download scheduling algorithm with small overhead. A CMO client needs to learn different servers' real-time capability quickly and schedule downloading according to it. Additionally, given the high overhead to maintain HTTP connections, the client should also use as few servers as possible to satisfy its downloading requirement. Second, how to minimize the cost of infrastructure usage. The charging models of infrastructures are typically non-linear and based on multiple factors (*e.g.* traffic volume and number of HTTP requests). Additionally, an infrastructure also has different price functions in different geographical locations. Such complex infrastructure charging models make it extremely difficult to decide how to choose infrastructures for each user to minimize cost as well as guarantee users' QoE.

Chapter 4 elaborates our design of CMO that addresses the preceding challenges. We leverage a two-level approach to realize the whole framework. In the client-side level, a client assigns a request window for each server, and the number of pieces it requests from the server never exceeds the window size. The window size is adjusted with AIMD (additive increase multiplicative decrease) according to the server's capability. A client requests pieces from different servers with different priorities. Intuitively, a server with a lower price and larger window size should be preferred because it can help to save cost and reduce the number of servers in use.

The ranks of server for cost is given by a global level centralized optimization. Based on the observation that the cost function of most infrastructures are concave (the more usage, the lower price), we design an optimal algorithm which decides which infrastructures to return to each user and the priority of these infrastructures. The client-side algorithm respects the ranking of infrastructures.

With such joint optimizations from both global and local views, we show that CMO can minimize the cost for utilizing the infrastructures while still guaranteeing users' QoE.

1.2 Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we describe FFC in more detail and show how the basic concept and algorithm can be extended in multiple dimensions. Chapter 3 describes how `zUpdate` is formulated, designed and proved. Additionally, we discuss how to handle several practical issues in the implementation of `zUpdate`. Next, Chapter 4 presents CMO and describes in details our two-level joint optimization design. Further, it proves the correctness and optimality of the global optimization algorithm. It leverages real CDNs in the evaluation to demonstrate its effectiveness in reality. Finally, in Chapter 5, we conclude and look ahead to future work.

Chapter 2

Traffic Engineering with Forward Fault Correction

Faults such as link failures and high switch-configuration delays can cause heavy congestion and packet loss in networks with traffic engineering. Because it takes time to detect and react to faults, these conditions can last long—even tens of seconds. We propose forward fault correction (FFC), a proactive approach to handling faults. FFC spreads network traffic such that freedom from congestion is guaranteed under arbitrary combinations of up to k faults. We show how FFC can be practically realized by compactly encoding the constraints that arise from this large number of possible faults and solving them efficiently using sorting networks. Experiments with data from real networks show that, with negligible loss in overall network throughput, FFC can reduce total data loss by a factor of 7$\sim$130 in well-provisioned networks, and reduce the loss of high-priority traffic to almost zero in well-utilized networks.

2.1 Introduction

Centralized traffic engineering reduces network congestion and increases efficiency [13, 21, 31, 42, 44, 47]. In such systems, a (logically centralized) TE controller frequently reconfigures the network to match current traffic demand. This control enables the network to carry more traffic, thus enabling the operators to extract more value from the expensive infrastructure investment.

While centralized TE can be highly effective, it suffers from an inability to quickly react to faults in both the data and control planes. Data plane faults occur when a link or switch fails, which impacts packet forwarding. Control plane faults occur when the controller fails to reconfigure a switch in a timely manner, even though the switch continues to forward packets (as previously configured). They can occur due to a host of factors, such as RPC (remote procedure call) failures, bugs in switch firmware or software, shortage of memory in the switch, and so forth.

Both types of faults are common. Several studies have reported frequent link and switch failures in large networks [40, 63, 74]; in a wide area network that we study, a link fails every 30 minutes on average. Google reports both heavy delays and outright failures in configuring switches [44]. The failure rate is in the range of 0.1-1%.[1] In a network with a hundred switches, this failure rate implies that the controller will commonly fail to configure at least some of them.

These faults can cause heavy congestion and packet loss. When a link fails, switches quickly move traffic to other available paths but, because this movement does not account for link capacity constraints, it can severely congest some links. Similarly, control plane faults cause congestion when a switch continues sending traffic on a link as per old configuration, while it was expected to move the traffic away. Our experiments show that data

[1] Our own experiments confirm this failure rate. We used custom software on commodity switches. For two different vendors, the failure rate was 10 times higher with the default software.

and control plane faults frequently lead to links getting 10-20% more traffic than capacity. The resulting high loss rate will seriously hurt TCP-based applications that are using the network.

Today, relieving congestion due to these faults requires intervention by the TE controller, but these reactions happen *after* congestion has already occurred. Worse, they can take a long time due to the delay inherent in updating a large network, which stems from factors such as RPC delays, control load on switches, and the time to switch forwarding rules. Updating a single switch rule can take a few seconds [44], and network-wide updates typically require updating many rules per switch.

We thus argue for proactively handling faults, i.e., spread traffic in the network such that no congestion occurs as long as the total number of faults is k or fewer. This guarantee should hold for arbitrary combinations of faults. Our view is inspired by forward error correction (FEC), in which a transmitted packet stream is modified such that all original packets can be recovered, without any reaction (e.g., retransmission) as long as the number of losses is up to k. Analogously, we call our approach forward fault correction (FFC).

Practically realizing FFC requires addressing two intertwined challenges—minimizing throughput loss and computational scalability. Just as FEC has overhead in terms of redundant information that is transmitted, FFC will have overhead in terms of link capacity set aside to tolerate faults. This overhead must be low for FFC to be acceptable. The computational challenge is that with n possible faults, the number of combinations of up to k faults is $\sum_{i=1}^{k} \binom{n}{i}$. With n=1000, a plausible number of links in a large network, and k=3, this number over 10^9. Thus, enumerating all possible faults and considering their impact is intractable. Approximations may help reduce computational effort, but unless one is careful, they may heavily impact overhead.

We address these challenges by first formulating FFC requirements as a linear program. This formulation is precise, which minimizes overhead, but has an intractably large

number of constraints because of possible fault combinations. We then observe that these constraints can be transformed, with minimal loss in precision, to a "bounded M-sum" problem: the sum of any M out of N variables is bounded. Finally, we develop a method based on sorting networks [19], to efficiently encode this problem as $O(kn)$ linear constraints. For control plane FFC, our approach is optimal with respect to overhead; for data plane FFC, it is optimal in the case where the multiple paths that carry traffic between two switches are disjoint.

Further, our approach is flexible and applies to many TE scenarios. We show how it accommodates multiple traffic priorities, multi-step network updates [42], different TE objectives (e.g., fairness versus maximizing throughput versus minimizing maximum link utilization), and handles configuration faults in flow rate limiters [18, 67].

We evaluate FFC in a testbed with commodity switches and using simulations based on traffic and fault data from real networks. We find that FFC is valuable in a range of scenarios. In well-provisioned networks, as is common for ISPs today, FFC has negligible throughput overhead and reduces data loss by a factor of 7$\sim$130. In well-utilized networks that use multiple traffic priorities, as is common for inter-datacenter networks [42, 44], FFC protects high-priority traffic from almost all loss, again with negligible loss in total network throughput.

Fundamentally, FFC makes available a novel control knob to network operators. Today, operators must either conservatively overprovision the network to guarantee the absence of congestion when faults occur or aggressively utilize the network [42, 44] at the risk of severe fault-induced congestion. FFC enables operating points that trade-off provisioning and congestion-risk in an informed manned, based on network characteristics and desired protection levels for traffic.

2.2 Motivation

FFC is motivated by the observations that data and control plane faults cause congestion and that reacting to these faults is slow. We illustrate these observations below.

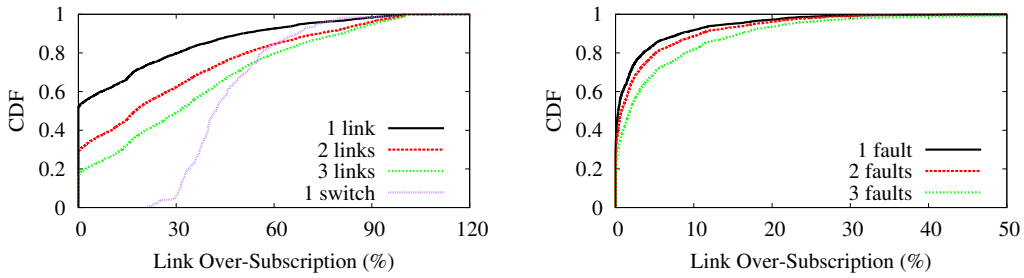
As is prevalent [13, 31, 42, 44] in TE networks, we assume tunnel-based forwarding. One or more tunnels carry traffic between each ingress-egress switch pair; we call this traffic a *flow*. Relative weights configured at the ingress switch determine how the flow's traffic is split across tunnels.

2.2.1 Impact of data plane faults

When a link or switch fails, it impacts all tunnels that traverse it. Upon learning about tunnel failures, ingress switches *rescale* traffic to the remaining tunnels for the flow, in proportion to configured weights. Suppose a flow has three tunnels with splitting weights $(0.5, 0.3, 0.2)$. When the third tunnel fails, weights of $(\frac{0.5}{0.8}, \frac{0.3}{0.8}, 0)$ are used to split traffic. OpenFlow group tables can implement such rescaling [7].

Rescaling quickly restores connectivity but can leave the network in a congested state. For example, Figure 2.2(a) shows an initial traffic distribution with two flows, $\{s2, s3\}$ $\rightarrow s4$. Dashed curves represent tunnels and numbers represent traffic volume they carry. When link $s2$-$s4$ fails and $s2$ rescales, the traffic distribution of Figure 2.2(b) emerges, in which link $s1$-$s4$ is heavily congested. Such congestion will persist until the TE controller can compute a new solution and configure the network, which can take tens of seconds (see below).

While this example was illustrative, Figure 2.1(a) characterizes congestion due to data plane faults for `L-Net`, a real network on which we provide more information in §2.8. This experiment uses topology and traffic data from the network, and it uses a standard TE algorithm (§2.4.1) to spread traffic every interval (5 minutes) across six tunnels per-



(a) Data plane faults (b) Control plane faults

Figure 2.1: Congestion due to faults in L-Net.

flow. We fail randomly selected links or switches in each time interval and measure the maximum link oversubscription rate, i.e., the amount of traffic above capacity that arrives at the link. The graph plots the CDF of the oversubscription rate for the cases of 1–3 link failures and 1 switch failure. Even with a single link failure—which occurs every 30 minutes on average in this network—the link oversubscription rate is over 20% a quarter of the time (75th %-ile). For a 100 Gbps link, 20% oversubscription means that 120 Gbps traffic will arrive soon after a failure. Even with 100 MB buffer capacity, the switch will be unable to buffer even 50 ms of this traffic and (TCP) flows will suffer a burst of high loss rate.

The graph also shows that in the worst case, links receive traffic that is twice their capacity. Switch failures hurt more in general, but have similar worst-case impact.

2.2.2 Impact of control plane faults

To illustrate how a delay or failure in configuring a switch causes congestion, Figure 2.3(a) shows a simple network with 4 flows: $s1 \rightarrow \{s2, s3\}$ and $\{s2, s3\} \rightarrow s4$. Assume that the controller wants to change the configuration to Figure 2.3(b), to accommodate a new flow $s1 \rightarrow s4$. This change requires updating $s2$ and $s3$, to modify traffic split weights for their flows. If $s2$ does not update and continues to split traffic as before, link $s1$-$s4$ will be congested, as in Figure 2.3(c).

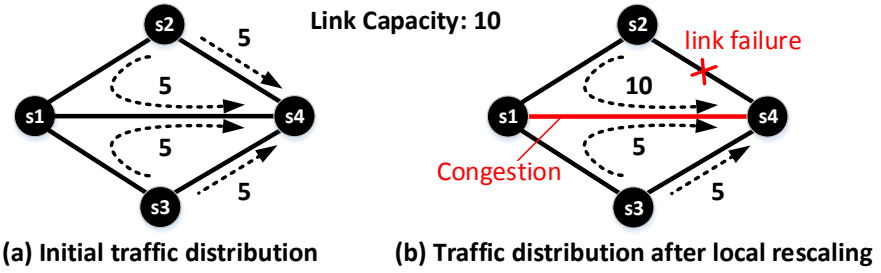


Figure 2.2: Congestion due to a data plane fault.

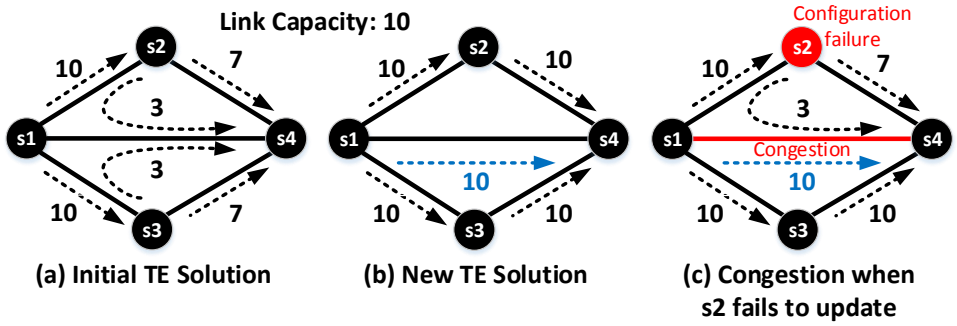


Figure 2.3: Congestion due to a control plane fault.

Figure 2.1(b) characterizes congestion in `L-Net` due to control plane faults. In this experiment, we simulate a network update every interval based on the same TE algorithm as above. For each update, we inject control plane faults at randomly selected switches and measure the maximum link oversubscription rate. We see from the graph that, though control plane faults are less damaging than data plane faults, even a single fault—-which can occur every 5 minutes if the fault rate is 1% [44] and the network has 100 switches—can lead to an oversubscription of 10% a tenth of the time.

While we focus on switch configuration faults above, a similar problem arises for rate limiter configuration as well, in networks that control traffic rate [42, 44]. Congestion will occure if a rate limiter continues to send traffic at the older, higher rate. We show that FFC handles such faults as well.

The example above assumes that all updates are sent to the switches in one shot; in another method [42], updates are sent in multiple steps to avoid transient congestion due

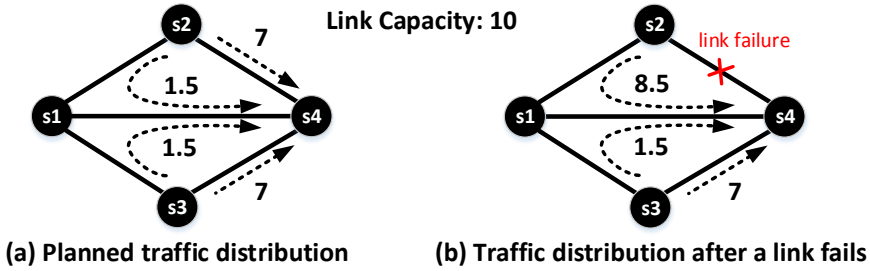


Figure 2.4: FFC for link failures ($k = 1$)

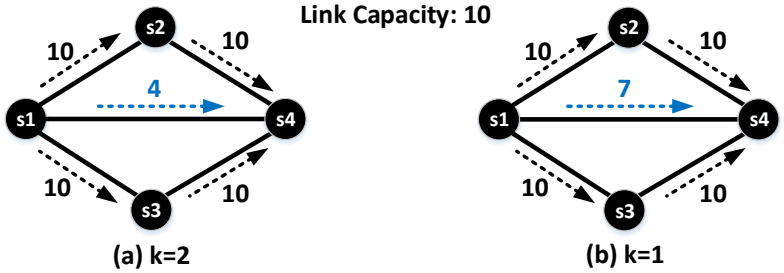
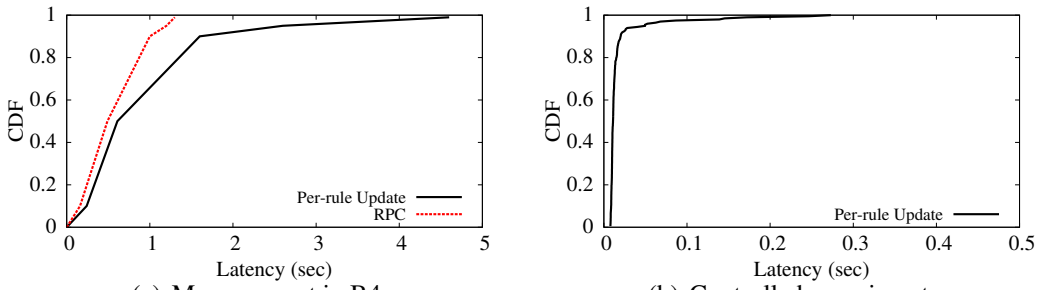


Figure 2.5: FFC for conntrol plane faults.

to switches applying updates at different times. To go from Figure 2.3(a) to 2.3(b), possible steps are: $1)$ update traffic splitting ratios at $s2$ and $s3$; and $2)$ if that succeeds, update the rate of flow $s1 \rightarrow s4$. This way, no congestion will occur if $s2$ (or $s3$) fails to update. However, configuration failures will stall network updates because Step 2 cannot proceed until Step 1 finishes. They will also lower throughput since flow $s1 \rightarrow s4$ cannot start if Step 1 fails. FFC handles control plane faults for multi-step updates as well.

2.2.3 Slow reaction to faults

Reactive approaches suffer from the fact that they start *after* congestion and loss has already started, and they can take a long time in large production networks. Figure 2.6(a) shows the distribution of rule update times in B4, based on Figure 12 and Table 4 of the paper [44]. It excludes switches for which configuration completely fails. It shows both RPC



(a) Measurement in B4 (b) Controlled experiments

Figure 2.6: Switch update latencies.

delays and the time to update a single forwarding rule. Typically, many rules per switch are updated during a network update; this number is commonly over 100 for `L-Net`. For updating R rules, the total update delay will be RPC delay + $R\times$ per-rule delay; we have confirmed this behavior experimentally. Thus, in B4, the median delay for 100 rules will be over 40 seconds and the worst case over 7 minutes.

While B4 is a complex network, we also quantified update delays in a controlled, lab environment using commodity switches. We issued rule update commands, while the switches had moderate background control load such as reading counters and running tunnel liveness detection protocol. (Forwarding load has no visible impact on rule update time.) The number and type of updates were drawn from those issued in `L-Net`. Figure 2.6(b) shows that the rule update time is still substantial. The median is 10 ms and the worst case is over 200 ms. Ignoring RPC delay, for updating 100 rules, the median update delay for a switch will be 1 second and the worst case over 20 seconds. We will show that FFC provides significant benefit even when switch update characteristics mimic this simplified setting.

It may be possible to make reaction times faster, but it is fundamentally limited by factors such as network path latencies, overloaded switch CPUs, time to update forwarding tables,[2] and noise inherent to any production environment. We thus advocate an approach

[2]Being based on TCAMs (ternary content addressable memory), they are optimized for fast lookups, rather than updates [31]

where common faults are handled proactively, and only big, rare faults are handle reactively.

2.3 FFC Overview and Challanges

Our goal is to develop proactive methods to handle data and control plane faults. Inspired by FEC, we develop the concept of FFC, which guarantees that no congestion will occur as long as the number of faults is up to (a configurable bound) k . Explicit fault detection or any reaction from the TE controller is not needed to maintain this guarantee.

Just as the primary controls in FEC are the number of packets sent and their encoding, the primary controls in FFC are the amount of traffic entering the network and its spread. (Assume for now that incoming traffic is rate controlled; in §2.5.4, we apply FFC to other networks.) We illustrate below how these controls, with some overhead in terms of lower throughput, can proactively protect against faults.

2.3.1 FFC for control plane faults

We start with control plane faults because they are unique to centralized TE and have not been studied before. Control plane FFC guarantees that no congestion occurs as long as the number of switches that experience a configuration fault is up to k. To see how this guarantee may be achieved, let us revisit Figure 2.3, in which we wanted to update switches $s2$ and $s3$ to accommodate a new flow. If we try to update the network from Figure 2.3(a) to 2.3(b), in which flow $s1 \rightarrow s4$ sends 10 units of traffic, it is impossible to be robust against configuration failure of $s2$ or $s3$. However, the network configuration of Figure 2.5(a), in which $s1 \rightarrow s4$ sends 4 units of traffic, is robust to either one or both switches failing to configure. Thus, this traffic distribution is an example of FFC with k=2, where no congestion will occur if up to two switches fail to update.

The downside of course is that the network throughput is lower than what would have been in the absence of faults and FFC. However, if $s2$ and $s3$ were successfully configured, the flow $s1 \rightarrow s4$ will be allowed to increase its rate to 10 units in the next period. Even if temporary, lowered throughput is an overhead of robustness provided by FFC.

As with FEC, FFC overhead is lower for lower protection levels. In the example above, if robustness to the configuration failure of up to one switch (k=1) is desired, we can safely install the configuration of Figure 2.5(b), which supports 7 units of flow $s1 \rightarrow s4$. There will be no congestion if $s2$ or $s3$ (but not both) fail to configure.

2.3.2 FFC for data plane faults

Data plane FFC guarantees that no congestion will occur as long as up to k links (or switches) fail. To see how this guarantee may be achieved, let us revisit Figure 2.2, where congestion occurs when link $s2$-$s4$ fails. However, if we spread traffic as in Figure 2.4(a), any single link failure ($k = 1$) will not cause congestion. For example, if link $s2$-$s4$ fails, the traffic distribution after rescaling is shown in Figure 2.4(b).

As for control plane, data plane FFC can lower throughput. When a link fails, network capacity is reduced. To not congest after a failure, we must leave spare capacity to absorb the traffic that was being carried by the link.

2.3.3 Challenges and overview of techniques

Practical realization of FFC for arbitrary topologies and traffic demands poses two challenges. The first is the scalability with which robust traffic distributions can be computed. If there are n network entities, and we want to be robust to up to k of them failing, FFC has to deal with $\sum_{j=1}^{k} \binom{n}{j}$ failure cases. Thus, naive, enumeration-based approaches are computationally intractable for large networks. We must meet the computational challenge

while meeting the second challenge of minimizing the loss in network throughput. After all, if network throughput were not a concern, a trivially robust solution is to not carry any traffic.

We address these challenges by first formulating the conditions on traffic quantity and spread as linear constraints. While this formulation is precise, it has a large number of constraints. We then reduce these large number of constraints to a much smaller number by observing that the constraints can be transformed to what we call the "bounded M-sum" problem and all constraints in such a problem can be reduced to a single constraint on the largest (or smallest) M variables. Finally, we encode these variables using efficient linear expressions with the aid of sorting networks [19]. The result is an FFC formulation with $O(kn)$ constraints.

Our techniques exploit two properties of our setting. First, the impact of a fault is easy to model. If switch configuration fails, it sticks to its old configuration; if a link fails, ingress switches deterministically rescale traffic. This simplicity allows us to capture the conditions imposed by FFC using efficient, linear constraints. Second, while faults are common, the fault ratio (i.e., the fraction of elements that fail) is low. Thus, it suffices to guard against a small number of faults (k). Solving for high values of k would be computationally intensive and impose a high throughput overhead.

2.4 Basic FFC Formulation

We now describe how to formulate and solve FFC for a basic TE setting. Our formulation is highly flexible, and §2.5 shows how it easily extends to a range of TE settings.

TE Input	G	Network graph with switches V and links E.
	F = $\{f\}$	Flows aggregated by ingress-engress switches.
	d_f	The bandwidth demand of f in a TE interval.
	c_e	The bandwidth capacity of link e.
	T_f	The set of tunnels that are set up for flow f.
	$l[t, e]$	1 if tunnel t uses link e and 0 otherwise.
	$s[t, v]$	1 if tunnel t's source is switch v and 0 otherwise.
TE Output	b_f	The granted bandwidth to flow f.
	$a_{f,t}$	The bandwidth allocated for flow f on tunnel t.
TE Others	$\beta_{f,t}$	The upper-bound of flow f's traffic.
	$w_{f,t}$	The traffic splitting weight of flow f on tunnel t.
FFC	p_f & p	The max number of f's tunnels traverse a link.
	q_f & q	The max number of f's tunnels traverse a switch.
	k_c, k_e, k_v	The number of configuration, link and switch failures that FFC protects the network against.
	τ_f	The min number of f's residual tunnels with up to k_e link and k_v switch failures.

Table 2.1: The key notations in FFC formulations.

2.4.1 Basic TE (without FFC)

The basic TE problem can be formulated as follows, with the key notations summarized in Table 4.1. The input is a graph G=(V, E), where V and E are sets of switches and directed links between switches. Each link $e \in E$ has a capacity c_e. The traffic demand is a set of flows, where each flow f is (aggregated) traffic from an ingress to an egress switch. The bandwidth demand of f in a TE interval is d_f and its traffic can be carried on a set of pre-established tunnels T_f.

The output of the TE is bandwidth allocation $\{b_f|\forall f\}$ of each flow and how much of the flow can traverse each tunnel $\{a_{f,t}|\forall f, t \in T_f\}$. In networks where the flow rate cannot be controlled, the TE output is only the latter (and $b_f = d_f$).

The TE problem can be solved based on path-constrained multi-commodity flow problem [35], as follows:

$$\max \quad \sum_{f \in F} b_f \tag{2.1}$$

$$\text{s.t.} \forall e \in E: \quad \sum_{f \in F, t \in T_f} a_{f,t} l[t,e] \leq c_e \tag{2.2}$$

$$\forall f \in F: \quad \sum_{t \in T_f} a_{f,t} \geq b_f \tag{2.3}$$

$$\forall f \in F, t \in T_f: \quad 0 \leq b_f \leq d_f;\ 0 \leq a_{f,t} \tag{2.4}$$

where the binary variable $l[t,e]$ denotes if tunnel t traverses link e. The TE objective in this formulation is to maximize network throughput (Eqn. 2.1). We consider other objectives in §2.5.3. Eqn. 2.2 says that no link should be overloaded, and Eqn. 2.3 says that the sum of the allocation of a flow across all its tunnels should no less than its allocated rate.[3] Eqn. 2.4 says that the bandwidth granted to a flow is no more than the flow's demand, and all variables are nonnegative.

The formulation above captures TE for both wide area networks (WAN) and data center networks (DCN). One difference is that in DCNs, given the larger scale, TE focuses only on large flows (elephants) and link capacity (c_e) refers to what is not used by small flows (mice).

To implement the computed solution, the TE controller updates the flow's rate limiters to $\{b_f\}$ and ingress switches to use traffic splitting weights of $w_{f,t} = a_{f,t} / \sum_{t \in T_f} a_{f,t}$.

2.4.2 Modeling control plane faults

For control plane faults, the goal of FFC is to compute the new configuration ($\{b_f\}$, $\{a_{f,t}\}$) such that no congestion will occur as long as k_c or fewer switches fail to update their old configuration ($\{b'_f\}$, $\{a'_{f,t}\}$). Another type of control plane fault is a failure to configure a rate limiter, which we will consider in §2.5.5. Let λ_v=1 denote a configuration failure for

[3]Using '$\geq$' instead of '$=$' in Eqn. 2.3 simplifies the exposition of FFC for data plane faults. For TE without FFC, given the goal of maximizing b_f, using '$\geq$' is equivalent to using '$=$'.

at least one of the flows with v as the ingress switch; λ_v=0 denotes that configuration for *all* flows starting at v succeeds. An individual case of control plane faults in the network can be represented by a vector $\boldsymbol{\lambda}$=$[\lambda_v|v \in V]$ that indicates the status of each switch. Thus, FFC that is robust to k_c faults requires that the network have no overloaded link under the set of cases $\Lambda_{k_c} = \{\boldsymbol{\lambda}| \sum_{v \in V} \lambda_v \leq k_c\}$.

This requirement can be captured as:

$$\forall e \in E, \boldsymbol{\lambda} \in \Lambda_{k_c} : \sum_{v \in V} \{(1 - \lambda_v)\hat{a}_{v,e} + \lambda_v \hat{\beta}_{v,e}\} \leq c_e \tag{2.5}$$

where $\hat{a}_{v,e}$ is the total traffic that can arrive at link e from flows starting at v if there is no configuration fault. That is:

$$\forall v \in V, e \in E : \quad \hat{a}_{v,e} = \textstyle\sum_{f \in F, t \in T_f} a_{f,t} l[t,e] s[t,v] \tag{2.6}$$

where binary variable $s[t,v]$ denotes if tunnel t's source switch is v.

In Eqn. 2.5, $\hat{\beta}_{v,e}$ is the upper bound on link e 's traffic from flows starting at v when a fault occurs ($\lambda_v = 1$). That is:

$$\forall v \in V, e \in E : \quad \hat{\beta}_{v,e} = \textstyle\sum_{f \in F, t \in T_f} \beta_{f,t} l[t,e] s[t,v] \tag{2.7}$$

where $\beta_{f,t}$ is the upper bound on flow f's traffic on tunnel t when a faults occurs for f. Since we assume the updates in rate limiters are successful, $\beta_{f,t}$ can be modeled as:

$$\forall f \in F, t \in T_f : \quad \beta_{f,t} = \max\{w'_{f,t} b_f, a_{f,t}\} \tag{2.8}$$

where $w'_{f,t}$ is flow f's splitting weight for tunnel t in the old configuration (which is known).

Adding Eqns. 2.5$\sim$2.8 to the basic TE formulation can, in theory, find TE configurations that are robust to k_c control plane faults. However, Eqn. 2.5 contains $|E|\sum_{j=1}^{k_c}\binom{|V|}{j}$ constraints because Λ_{k_c} has $\sum_{j=1}^{k_c}\binom{|V|}{j}$ failures cases in total. Directly solving for these number of constraints is computationally intractable. We outline in §2.4.5 how we reduce this problem to a smaller number of equivalent constraints.

2.4.3 Modeling data plane faults

For data plane faults, the goal of FFC is to compute flow allocations such that no congestion occurs even after up to k_e links fail *and* up to k_v switches fail. The guarantee is for link failures that are *not* incident on the failed switches. Since switch failures imply link failures, one could protect against them by considering only link failures [78]. But we explicitly consider switch failures because switches can have a large number of incident links; protecting against switch failures implicitly (using link failures) would require a high value of k_e. This approach would significantly hurt throughput because it will protect against arbitrary combinations of up to k_e links, a much stronger condition than protecting against k_e incident links on the same switch.

We model data plane FFC as follows. Let μ_e=1 denote that link e has failed, and η_v=1 denote that switch v has failed; the variable values are 0 otherwise. Then, a case of data plane fault can be represented by $(\boldsymbol{\mu}, \boldsymbol{\eta})$ in which vector $\boldsymbol{\mu}$ = $[\mu_e | e \in E]$ and $\boldsymbol{\eta}$ = $[\eta_v | v \in V]$. Thus, TE that is robust to k_e link failures and k_v switch failures requires that there is no overloaded link under the set of hardware failure cases $U_{k_e,k_v} = \{(\boldsymbol{\mu}, \boldsymbol{\eta}) | \sum_e \mu_e \leq k_e, \sum_v \eta_v \leq k_v\}$.

Recall that data plane faults can cause congestion because they alter traffic distribution over the network when ingress switches rescale traffic, that is, move it from the impacted

to the residual tunnels for the flow. Given a fault case $(\boldsymbol{\mu}, \boldsymbol{\eta})$, we know the residual tunnels $T_f^{\boldsymbol{\mu},\boldsymbol{\eta}}$ of each flow f—those that do not traverse any failed link or switch. FFC requires that f's residual tunnels be able to hold its allocated rate.

$$\forall f \in F, (\boldsymbol{\mu}, \boldsymbol{\eta}) \in U_{k_e,k_v} : \quad \sum_{t \in T_f^{\boldsymbol{\mu},\boldsymbol{\eta}}} a_{f,t} \geq b_f \tag{2.9}$$

Further, if a flow f has no residual tunnels ($T_f^{\boldsymbol{\mu},\boldsymbol{\eta}} = \emptyset$) under a failure case $(\boldsymbol{\mu}, \boldsymbol{\eta})$, its flow size b_f should be fixed to 0.

Eqn. (2.9) also guarantees that no link will be overloaded:

Lemma 1 *A TE configuration ($\{a_{f,t}\}$, $\{b_f\}$) which satisfies constraints Eqn. 2.2 ∼Eqn. 2.4 and Eqn. 2.9 under fault case $(\boldsymbol{\mu}, \boldsymbol{\eta})$ causes no link overload after all ingress switches rescale.*

Proof. When a data plane failure case $(\boldsymbol{\mu}, \boldsymbol{\eta})$ happens, the traffic load of a flow f on a residual tunnel $t \in T_f^{\boldsymbol{\mu},\boldsymbol{\eta}}$ is:

$$b_{f,t}^{\boldsymbol{\mu},\boldsymbol{\eta}} = \frac{a_{t,f}}{\sum_{t \in T_f^{\boldsymbol{\mu},\boldsymbol{\eta}}} a_{f,t}} * b_f \leq \frac{a_{t,f}}{b_f} * b_f = a_{f,t} \tag{2.10}$$

which is directly derived from Eqn. 2.9. Therefore, we know the total traffic load on a link e is:

$$\forall e \in E : \quad \sum_{f,t \in T_f^{\boldsymbol{\mu},\boldsymbol{\eta}}} b_{f,t}^{\boldsymbol{\mu},\boldsymbol{\eta}} l[t,e] \leq \sum_{f,t \in T_f} a_{f,t} l[t,e] \leq c_e$$

which finishes the proof. ■

As for control plane faults, adding these constraints to the basic TE will in theory yield a solution that is robust to data plane faults, but directly solving for these constraints is

intractable given the large number of possible failure cases in U_{k_e,k_v}. Before describing how to solve these constraints, we briefly discuss how careful tunnel layout can improve robustness to data plane faults as well as reduce the overhead of data plane FFC.

2.4.4 Robust tunnel layout

One observation from Eqn. 2.9 is: higher the number of residual tunnels, greater the network throughput that FFC supports. Thus, we can help improve throughput by laying out tunnels such that flows lose as few tunnels as possible when faults occur. The ideal case is that each a flow loses at most one tunnel when a fault occurs, but it requires that tunnels be switch-disjoint, which also implies link-disjoint, which limits flows to a small number of tunnels in networks with low path diversity. That would in turn restrict network throughput, as more tunnels are better able to utilize network capacity.

To balance these needs, we recommend (p, q) link-switch disjoint tunnels. For an individual flow, at most p tunnels should traverse a link and at most q tunnels should traverse a switch. The parameters p and q can be flow specific. Algorithms to find link and switch disjoint paths can be extended to find (p, q) link-switch disjoint tunnels. We outline a simple algorithm to find network paths that can support such tunnels as follows.

Finding p-link-q-router-disjoint paths

We can construct p-link-q-router-disjoint tunnels by extending the classic arc-disjoint and vertex-disjoint path search algorithms [24]. Figure 2.7 shows a concrete example to demonstrate how the algorithm works. The original network topology is in Figure 2.7(a). In the classic arc-disjoint path search algorithm, regardless the actual link capacity in (a), the virtual capacity of all links is set to 1 as in (b). The link-disjoint tunnels from $s1$ to $s5$ are the pathes used by the max-flows from $s1$ to $s5$ on the network graph with the virtual link capacity.

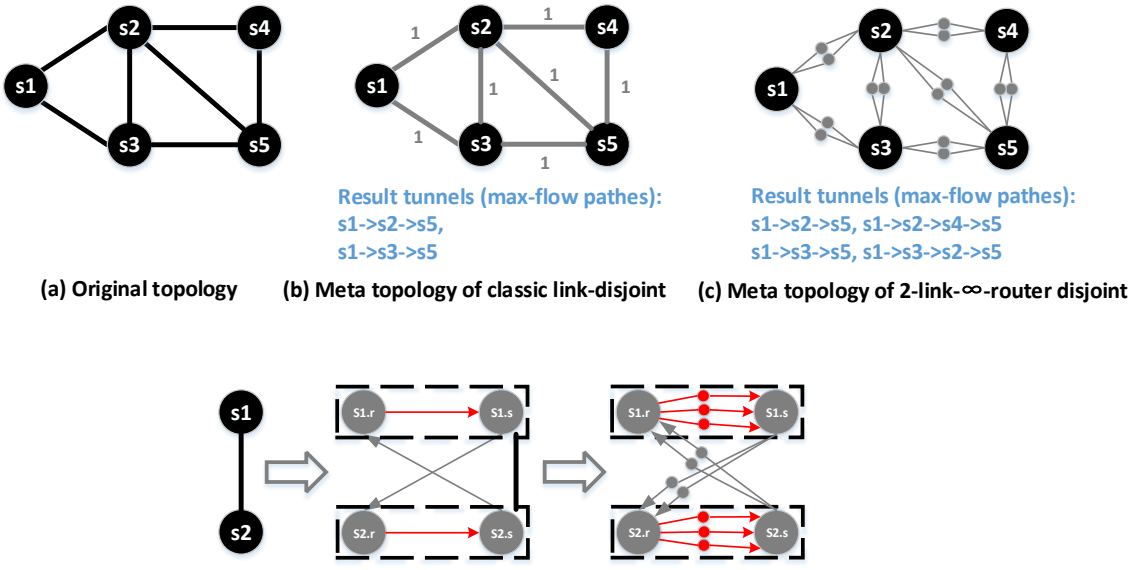


Figure 2.7: A concreate example of the algorithm to construct robust tunnels.

If we want 2-link-∞-router disjoint pathes, we replace each link with 2 meta nodes and the links (with capacity 1) connecting the meta nodes and the start/end node of the original link, as illustrated by (c). On such a meta-graph, we find link-disjoint pathes from $s1$ to $s5$. After we remove all the meta nodes in the pathes, we get the tunnels that are 2-link-∞-router disjoint.

For router's disjointment, we first need to transform nodes on the graph into intra-node links. Figure 2.7 (d) shows the transformation of a single link, For 2-link-3-router disjointment, we first transform each node on the original graph into a directional intra-node link from the ''receiver'' part of the node ($s1.r$ and $s2.r$ in (d)) to the '''sender' part ($s2.s$ and $s2.s$ in (d)). Accordingly, each inter-node link in the original graph becomes two directional links from one end's sender part to the other end's receiver part. Next, we bring in 2 meta-nodes to inter-node links and 3 meta-nodes to intra-node links. Finally, we obtain the 2-link-3-router disjoint tunnels on the original graph after removing all meta-

nodes and transform intra-node links back to nodes from the link-disjoint pathes on the meta-graph (not shown in Figure 2.7).

Note that the robust tunnels are built on physical links rather than logical links. If two routers have multiple physical links that can fail independently, each one of the links can carry p tunnels for a flow in a p-link-q-router disjoint strategy.

2.4.5 Efficiently solving FFC constraints

To tractably solve the large number of FFC constraints, we transform them to a "bounded M-sum" problem and then encode the transformed problem using a sorting network.

Transformation to bounded M-sum

We define the *bounded M-sum* problem as: Given a set of N variables, the sum of any M of those should be less (or more) than a bound B. Formally, if N_M is the set of all possible variable subsets with cardinality $\leq M$, we have:

$$\forall S \in N_M : \sum_{n_i \in S} n_i \leq B \tag{2.11}$$

The interesting aspects of this problem are: $1)$ FFC constraints can be transformed to it; and $2)$ while the original formulation has a large number of constraints, all of them are satisfied as long as one constraint involving the largest M variables is satisfied. If n^j is an expression for the j-th largest variable in N, all constraints above hold if:

$$\sum_{j=1}^{M} n^j \leq B \tag{2.12}$$

Thus, if we can find efficient (linear) expressions for the largest M variables in N, we

can replace the original subset constraints with one constraint. We show below how to find such expressions, but first we show how to transform FFC constraints into bounded M-sum problem.

Control plane faults Eqn. 2.5 of control plane FFC can be equivalently re-written as:

$$\forall e \in E, \boldsymbol{\lambda} \in \Lambda_{k_c} : \sum_{v \in V} \lambda_v (\hat{\beta}_{v,e} - \hat{a}_{v,e}) \leq c_e - \sum_v \hat{a}_{v,e} \tag{2.13}$$

Let $D = \{\hat{\beta}_{v,e} - \hat{a}_{v,e} | v \in V\}$ and d^j be the jth largest element in D. Since $\hat{\beta}_{v,e} - \hat{a}_{v,e} \geq 0$, Eqn. 2.13 is equivalent to

$$\forall e \in E : \sum_{j=1}^{k_c} d^j \leq c_e - \sum_v \hat{a}_{v,e} \tag{2.14}$$

Thus, we have transformed the original $|E| \times |\Lambda_{k_c}|$ constraints into $|E|$ constraints, one for each link.

Data plane faults Assume that the tunnels of flow f are (p_f, q_f) link-switch disjoint. (The values (p_f, q_f) are computable for any given tunnel layout; the layout does not have to use the robust strategy above.) Then, for any data plane fault case $(\boldsymbol{\mu}, \boldsymbol{\eta}) \in U_{k_e,k_v}$, the number of residual tunnels is no less than $\tau_f = |T_f| - k_e p_f - k_v q_f$. Suppose $a^j_{f,t}$ is the jth smallest (not largest) element in $A_f = \{a_{f,t} | t \in T_f\}$, the following guarantees that all constraints in Eqn. 2.9 are satisfied:

$$\forall f : \quad \textstyle\sum_{j=1}^{\tau_f} a^j_{f,t} \geq b_f \tag{2.15}$$

This is because the left-hand side of Eqn. 2.15 is the worst-case bandwidth allocation that flow f can have from its residual tunnels under any case in U_{k_e,k_v}.

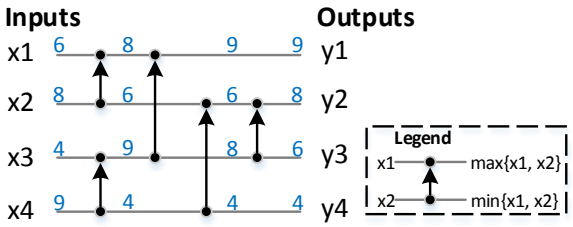


Figure 2.8: An example sorting network.

Unlike control plane faults, the transformation from Eqn. 2.9 to Eqn. 2.15 does not preserve equivalence. Satisfying Eqn. 2.15 satisfies Eqn. 2.9, but not vice versa. In the special cases link failures with link-disjoint tunnels and switch failures with switch-disjoint tunnels, the two are equivalent.

Interestingly, however, the imprecision of Eqn. 2.15 allows it protect against fault cases beyond U_{k_e,k_v}. It essentially protects the network against any data plane fault case where the number of tunnel failures is no more than $k_t = k_e p_f + k_v q_f$. Suppose (p_f, q_f)=$(1, 3)$ and our desired protection level is up to three links failures and no switch failure (k_e=3, k_v=0), with Eqn. 2.15 we also simultaneously protect the network against a single arbitrary switch failure and no link failure (k_e=0, k_v=1). We leverage this effect in our experiments (§2.8).

Encoding for largest (or smallest) M variables

We now explain how we express the largest M variables as linear constraints. When added to other TE constraints, they help efficiently compute FFC traffic distribution.

Our method is based on sorting networks [19], which are networks of compare-swap operators that can sort any array of N values. An example network to sort 4 values is shown in Figure 2.8. This network is based on the merge sort algorithm. Each compare-swap operator takes two inputs from left, and it moves the higher input upwards and the lower downwards.

The characteristic of sorting networks that we exploit is that the sequence of compare-

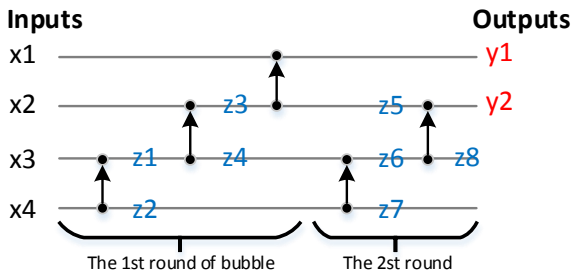


Figure 2.9: A network to find the largest-2 elements.

swap operations are independent of the input, unlike many sorting algorithms (e.g., quick sort) where the next comparison depends on the outcome of previous ones. This characteristic allows us to encode its computation as linear expressions for each of the largest to smallest variable.

We also exploit that we are interested in only the largest M values, rather than sorting them all, which allows us to build a smaller network. Practical sorting networks requires $O(Nlog(N)^2)$ compare-swap operators, while we use a partial network with $O(NM)$ operators. Since M is small in our setting—equivalent to the number of faults we want to guard against—and N is large, the savings are significant.

We base our network on bubble sort; unlike other algorithms, its premature termination after M stages yields the largest M values. Figure 2.9 illustrates our strategy for the case of finding the largest 2 of 4 values. The first stage finds the largest element, and the second finds the second largest.

Algorithm 1 shows the pseudocode for generating expressions for the largest M values. It operates in M steps, and in each step, it builds an expression for the largest of the remaining values. Algorithm 2 shows the pseudocode for building the expression for the largest value.

A similar approach can find the expressions for the smallest M values. We just need to use compare-swap operators that push the lower of the two values upwards.

Algorithm 1: LargestValues(X, M)

[Input] X: an array of variables
[Input] M: the number of largest values to extract
[Output] Y: an array of new variables in which $Y[i]$ $(0 \leq i \leq M)$ is the ith largest element in X
[Output] C: a set of constraints between X and Y

$Y \leftarrow \emptyset$; $C \leftarrow \emptyset$;
// Pop M largest variables from X.
while $|Y| < M$ **do**
    $y^*, X, C' \leftarrow$ BubbleMax(X) ;
    $Y \leftarrow Y + \{y^*\}$; $C \leftarrow C + C'$;
return Y, C ;

Algorithm 2: BubbleMax(X)

[Input] X: an array of variables
[Output] x^*: a variable that represents max$\{X\}$
[Output] Y: an array that represent $X \setminus \{x^*\}$
[Output] C: a set of constrants among X, Y and x^*

$x^* \leftarrow X$.pop(); $Y \leftarrow \emptyset$; $C \leftarrow \emptyset$;
while $X \neq \emptyset$ **do**
    $x \leftarrow X$.pop() ;
    $x_{max}, x_{min} \leftarrow$ new variables ;
    // Make two new constraints.
    $c_1 \leftarrow 2 * x_{max} = x + x^* + |x - x^*|$;
    $c_2 \leftarrow 2 * x_{min} = x + x^* - |x - x^*|$;
    $x^* \leftarrow x_{max}$; $Y \leftarrow Y + \{x_{min}\}$; $C \leftarrow C + \{c_1, c_2\}$;
return x^*, Y, C ;

Throughput and computational overhead

Our methods have the following properties with respect to throughput overhead: 1) Our control plane FFC scheme is optimal; and 2) Our data plane FFC scheme is optimal for the special cases of link failures with link disjoint tunnels and switch failures with switch disjoint tunnels. Optimal means that no other scheme will have lower overhead for the same degree of protection.

The computational overhead of our methods can be characterized using the number of additional variables and constraints they introduce in the LP. The basic (non-FFC) TE

problem has $2|F| + |E|$ constraints and $\sum_f |T_f| + |F|$ variables. Control plane FFC introduces $|E|$ constraints (Eqn. 2.14) plus up to $4k_c|V||E|$ constraints and up to $3k_c|V||E|$ variables to encode the partial sorting network. (Even though we show only 2 new variables and 2 constraints in Algorithm 2, multiplicative factors of 4 and 3 stem from converting the absolute values in Lines 10$\sim$11 into standard linear constraints.) Data plane FFC introduces up to $|F|+4\sum_f |T_f|\min\{|T_f|-\tau_f, \tau_f\}$ constraints and up to $3\sum_f |T_f|\min\{T_f-\tau_f, \tau_f\}$ variables. Recall that $\tau_f = |T_f| - k_e p_f - k_v q_f$.

2.4.6 Combined FFC for both faults types

To simultaneously protect against control and data plane faults, we simply include both types of constraints in TE computations. It will take three parameters (k_c, k_e, k_v) and the guarantee is that no congestion will occur as long as switch configuration failures, link failures, and switch failures are up to k_c, k_e , and k_v , respectively.

A subtle issue can arise in combined protection settings right after data plane faults bigger than the protection level (e.g., number of failed links $> k_e$). Due to such faults, some links may get congested and there may be no way to move traffic away from them while being also robust to control plane faults. For instance, assume that due to a big data plane fault, after rescaling, a link e with capacity 10 gets 7 units of traffic each from three flows that start at different ingress switches. Moving the extra 11 units of traffic away from the link requires updating at least two switches. But if we are protecting against two control plane faults (k_c=2) planning for this movement is impossible; there would be no feasible solution to the FFC constraints. To handle this issue, we allow unprotected moves for overloaded links by setting k_c=0 for such links in Eqn. 2.5. In theory, overloaded links can arise after big control plane faults as well, but in our experience, such faults are unlikely to create congestion that is so severe that traffic cannot be moved in a manner that is robust to (further) control plane faults.

2.5 Extending Basic FFC

Our FFC formulation is not only efficient but also flexible. We now show how it extends to a wide range of TE settings.

2.5.1 Traffic with different priorities

Earlier, we assumed that all traffic has the same priority; some networks may use multiple priorities to differentiate between applications with different performance requirements [42, 44]. FFC can be extended to this setting to offer different levels of protections to different priorities. The TE solution for higher priority traffic is computed first with a custom protection level (k_c^h, k_e^h and k_v^h), and the TE solution of lower priority traffic is computed next with its own protection level (k_c^l, k_e^l and k_v^l). This cascading computation is already done to support multiple priorities [42, 44]; computation for lower priority traffic uses residual link capacity (not actually used higher priority traffic).

A requirement for the extension above is that the protection level for high priority should not be smaller ($k_x^h \geq k_x^l, x \in \{c, e, v\}$), which is a desirable property anyway. Otherwise, the FFC-TE for lower priorities may not have a feasible solution because the configuration for high priority traffic may violate FFC constraints.

2.5.2 Congestion-free updates

Some networks use multi-step updates to preclude transient congestion caused by different switches updating their configuration at different times [42, 61]. The basic idea is to find a chain of intermediate TE configurations A^i=$\{a_{f,t}^i\}(0 < i < n)$ that update the network from current configuration A^0 to the desired configuration A^n. The transition between each pair of adjacent TE configurations is guaranteed to be congestion free irrespective of the order in which the switches apply updates. Such intermediate TEs are found using the following

key constraint:

$$\forall e \in E, i : \sum_{v} \max\{\hat{a}^{i-1}_{v,e}, \hat{a}^{i}_{v,e}\} \leq c_e \tag{2.16}$$

This constraint captures the condition that each link e should be able to accommodate the maximum traffic, across adjacent configurations, it gets from each ingress switch v. After computing the intermediate configurations, the TE controller updates the network step-by-step: $A^0 \rightarrow A^1 \ldots A^n$.

With congestion-free updates, control plane faults will not cause congestion, but will block the update process; the preceding step must complete before the next step can be taken. In this setting, FFC can ensure that the update can proceed from A^{i-1} to A^i as long as the cumulative number of faults (across all steps thus far) is k_c or fewer. We can find such intermediate configurations by replacing Eqn. 2.16 with:

$$\forall e \in E, \lambda \in \Lambda_{k_c}, i :$$
$$\textstyle\sum_v \lambda_v \max\{\hat{\beta}^0_{v,e}, \ldots \hat{\beta}^i_{v,e}\} + (1 - \lambda_v)\max\{\hat{a}^{i-1}_{v,e}, \hat{a}^i_{v,e}\} \leq c_e$$

This is a large number of constraints, but we can solve them efficiently by transforming them to the bounded M-sum problem (§2.4.5). We omit details for space constraints.

2.5.3 Optimizing for fairness

Earlier, we assumed that TE objective was to maximize network throughput; another common objective is fairness among flows [42, 44]. Fairness typically introduces more constraints in the TE problem, and simply including those constraints will yield FFC-TE with fairness. As a concrete example, consider the iterative approximate-max-min fair meth-od

of SWAN [42]. It solves the basic TE (Eqns. 2.1- 2.4) multiple times, each time with an upper-bound on flow allocations (b_f). The bound is iteratively increased by multiplying it by a factor α. The allocation of flows that are unable to reach the bound in a given iteration are frozen for future iterations, and the iterations continue until the bound goes above the maximum flow demand (d_f). This process ensures that flows with large demands cannot get a higher allocation until the allocation of other flows cannot be increased to at least that level. It yields flow allocations that are provably at most α away from true max-min fair allocation (which is computationally hard to compute) [42].

The same process can be used largely unmodified to compute TE solutions that are both fair and provide FFC. The only difference is that in each iteration we include the FFC-related constraints as well.

2.5.4 TE without flow rate control

In some networks, such as ISP backbones, controlling the rates of incoming flows is not possible. Instead of allocating flow rates, the goal of TE tends to be to configure the network such that maximum link utilization (MLU) is minimized while carrying the offered demand ($\{d_f\}$). The network configuration ($\{a_{f,t}\}$) can be computed by using the following objective function and constraint:

$$\begin{aligned} &\text{min.} && \Theta(u) \\ &\text{s.t.}\forall e: && u \geq \textstyle\sum_v \hat{a}_{v,e}/c_e \end{aligned}$$

where u denotes MLU and Θ is a function of MLU that needs to be minimized.

Constraints for data plane FFC stay the same in this setting, but control plane FFC requires changing the objective function and additional constraints, as follows:

$$\text{min.} \qquad \Theta(u) + \sigma\Theta(u_\text{f})$$

$$\forall e \in E, \lambda \in \Lambda_{k_c}: \quad u_\text{f} \geq \sum_v \{\lambda_v \hat{\beta_{v,e}} + (1-\lambda_v)\hat{a}_{v,e}\}/c_e$$

where $\sigma > 0$ is a coefficient that balances the importance of MLU in normal cases u and MLU when faults occur u_f. These constraints can be solved using the method of § 2.4.5.

2.5.5 Control plane faults for rate limiters

Earlier, we assumed that configuration updates for rate limiters always succeed; some networks may experience update failures for rate limiters as well. If ingress switches and rate limiters are updated independently, the traffic load of a flow on a tunnel can be a mix of old or new traffic splitting weights and old or new flow sizes. We can account for this by modifying Eqn 2.8 to:

$$\beta_{f,t} = \max\{a^{'}_{f,t}, b^{'}_f w_{f,t}, b_f w^{'}_{f,t}, a_{f,t}\} \tag{2.17}$$

In some networks, the updates on switches and rate limiters are ordered to ensure that there is no congestion due to transient inconsistencies in flow sizes and tunnel weights [42]. The order is: if f's size is increasing ($b^{'}_f$ ¡ b_f), the traffic splitting weights at ingress switches are updated first and the rate limiter is updated after (and only if) that succeeds; otherwise, the rate limiter is updated first and the splitting weights are updated after that succeeds. Thus, if $b^{'}_f$ ¡ b_f, the combination of new flow size and old weights ($b_f w^{'}_{f,t}$) will not occur and $b^{'}_f w_{f,t}$ ¡ $a_{f,t}$; similarly if $b^{'}_f$ ¿ b_f, the combination of old flow size and new weights ($b^{'}_f w_{f,t}$) will not occur and $b_f w^{'}_{f,t}$ ¡ $a^{'}_{f,t}$. We can then simplify Eqn. 2.17 to:

$$\beta_{f,t} = \max\{a^{'}_{f,t}, a_{f,t}\} \tag{2.18}$$

Besides simplifying the FFC formulation, ordering of updates on switches and rate limiters also helps lower the overhead of FFC. It reduces the number of possible traffic configurations for which we must be prepared.

2.5.6 Uncertainty in current TE

Earlier, we assumed that while computing the next TE configuration, the controller exactly knows the current configuration $(\{a^{'}_{f,t}\}, \{b^{'}_{f}\})$ of each flow. Sometimes, however, there may be uncertainty in the configuration of some flows, e.g., if update commands were sent in the last round to change configuration from $(\{a^{''}_{f,t}\}, \{b^{''}_{f}\})$ to $(\{a^{'}_{f,t}\}, \{b^{'}_{f}\})$ but the success of some updates could not be confirmed. In such an event, the configuration of the flow can be either $(\{a^{''}_{f,t}\}, \{b^{''}_{f}\})$ or $(\{a^{'}_{f,t}\}, \{b^{'}_{f}\})$.

We can explicitly account for this uncertainty in FFC computations. Suppose $\mathcal{F}$ is a set of flows with uncertain configurations. Instead of computing yet another configuration for them, we try to bring their configuration up-to-date and, in terms of link capacity allocation, we plan for them to be in either of the last two configurations. The following constraints capture this strategy:

$$\forall f \in \mathcal{F}, t \in T_f: \quad b_f = b^{'}_{f}; a_{f,t} = a^{'}_{f,t}$$
$$\forall f \in \mathcal{F}, t \in T_f: \quad \beta_{f,t} = \max\{a^{''}_{f,t}, a^{'}_{f,t}\}$$

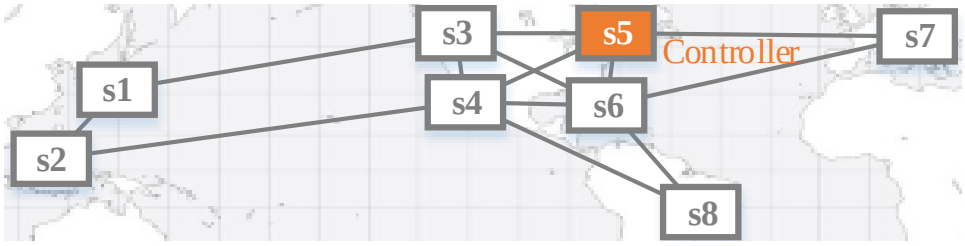


Figure 2.10: The network topology emulated in our testbed.

2.6 Implementation

Our FFC controller is implemented as a drop-in replacement for existing TE controllers. It takes as input traffic demand (per-priority ingress-egress flow), network topology, and current traffic, and produces the allocation of each flow and configuration for each switch. The additional configuration for our controller includes the protection level (k_c, k_e, k_v) for each priority. It implements all the extensions in §2.5. We use Solver Foundation [4] v3.0.2 with CPLEX [1] plugin v12.5.0 as our LP solver.

Our implementation includes a few optimizations that do not practically impact the FFC properties but help reduce the computational burden. For control plane FFC, observe that not all ingress switches have traffic on each link. Thus, in Eqn. 2.14, if a switch v has no load on a link e in the old TE, we ignore v when considering the safety of e. Similarly, we also ignore switches that have little traffic load—less than 0.001% of capacity—on e in the old TE because the impact of such switches not updating is negligible. For data plane FFC, we pick mice flows—those that collectively carry less than 1% of the traffic—and fix their tunnel bandwidth allocations with the constraint $a_{f,t} = \frac{b_f}{\tau_f}$, which suffices to satisfy Eqn. 2.15, rather than using sorting networks.

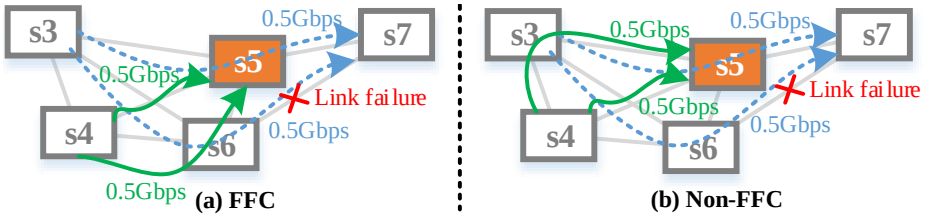


Figure 2.11: Traffic distribution before link s6-s7 fails.

2.7 Testbed Evaluation

We begin our evaluation of FFC by first performing experiments on a testbed. These experiments show the value of FFC with real switches and actual delays in detecting and reacting to failures. The next section shows the value of FFC at scale using simulations with data from a real WAN.

Our testbed emulates a WAN with 8 sites spread across 4 continents, as shown in Figure 2.10. Each site has 5 servers that generate traffic and 1 WAN-facing switch (Arista 7050T). The capacity of every cross-site link is 1 Gbps. The TE controller is in New York ($s5$), and we emulate delays for control messages based on geographic distances. We update switch rules via a custom software agent running on the switches (which we have found to be more performant and reliable than the built-in software). We use iPerf [3] on the servers to generate UDP traffic flows with specified rates and measure packet loss according to both iPerf's server reports and the packet counters in the switches. All switches run link liveness detection protocol, and they report any failures to ingress switches. Upon hearing about a failure, ingress switches rescale traffic away from the impacted tunnels.

We conducted several experiments to study the behavior of FFC and non-FFC TE, we present results from a simple, representative experiment that illustrates their differences for a data plane fault. This experiment has two flows, s3→s7 and $s4 \rightarrow s5$, each with demand 1 Gbps. Figure 2.11 shows how this traffic was spread in the case of FFC and non-FFC. The main difference is that FFC uses tunnel s4-s6-s5, instead of s4-s3-s5, to transmit

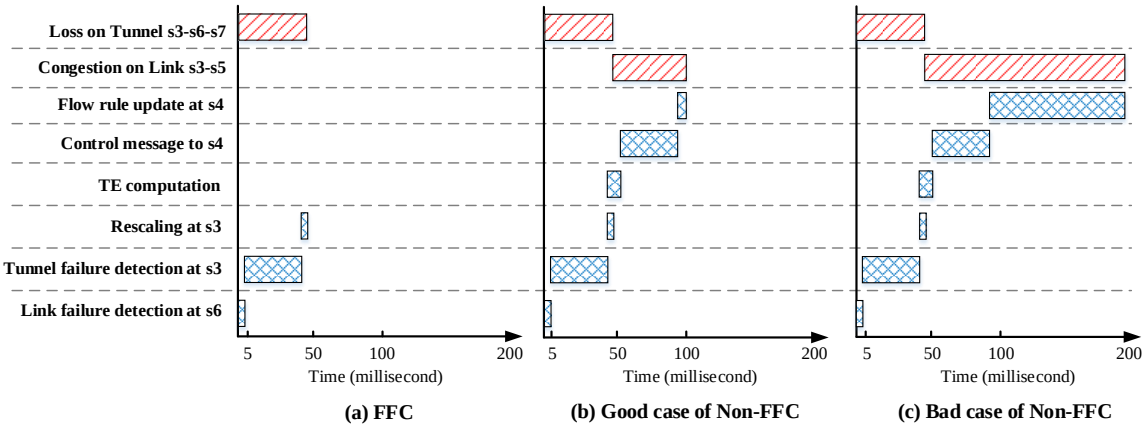


Figure 2.12: Events with and without FFC during link failures.

0.5 Gbps traffic, which provides protection against any single-link failure. We conducted many trials with this setup; in each we failed link s6-s7 and observed the ensuing events.

FFC behavior was consistent across trials. Figure 2.12(a) shows an example. The x-axis is a time line relative to when the link failure was injected and y-axis lists the events that could happen in a failure reaction. Shadowed blocks denote the start and end of the event. We see $s6$ detects link failure within 5 ms, and $s3$ hears about it within 45 ms. $s3$ rescales and moves all traffic to the residual tunnel within 2 ms. Packet loss on tunnel s3-s6-s7 stops immediately after that. We thus see that losses in FFC are a purely a function of the time it takes for fault detection and rescaling delay. We see that modulo propagation delay, which is fundamental, today's switches can detect faults and rescale quickly. FFC ensures that these activities suffice at eliminating congestion, and the TE controller does not need to react.

The situation is more complex in the non-FFC case. After s3 rescales, link s4-s5 will get congested as it starts getting 1.5 Gbps of traffic; see Figure 2.10(b). To remedy this, the TE controller computes a new solution in which s4 moves 0.5 Gbps of traffic to tunnel s4-s6-s5 and updates the switch s4.

Figure 2.12(b) shows the best case for non-FFC in our trials, in which the update

itself was quick (within 5 ms). We see rescaling-related loss on s3-s5 stops within 45 ms, given the propagation delay from the controller to the switch s4. Overall, the network was congested for twice the time compared to FFC. Figure 2.12(c) shows a bad case for non-FFC, in which the switch s4 took a long time to apply the update. Consequently, the congestion lasted for much longer.

2.8 Data-Driven Evaluation

We now evaluate FFC with topology, traffic, and failure data from real networks. We first microbenchmark its throughput and computation cost, then study its end-to-end impact in single- and multi-priority networks, and finally study its impact on update time for multi-step updates. We start by describing our data sources and experimental methodology.

2.8.1 Experimental methodology

Networks: We use two inter-datacenter networks in our experiments. The first, which we call `L-Net`, has $O(50)$ sites globally with $O(100)$ and $O(1000)$ links. We have data on link capacities, traffic flows, and data plane faults for this network. To ensure that our results to are robust to network topology, we also use B4's site-level topology with 12 sites [44], which we call `S-Net`. Not knowing the internal details of `S-Net`, we assume that there are 2 switches per site and site-level connectivity is composed of 16 switch-level 10 Gbps links. Switches for a site connect symmetrically.

Traffic demand: We collect network traffic logs and aggregate it into ingress-egress flows. We partition time into 5-minute bins (the TE interval) and the demand of the flow for an interval is the average bandwidth it consumed. For `L-Net`, we use its traffic logs directly. For `S-Net`, we use traffic logs from another inter-datacenter network (not `L-Net`) and synthesize demand by mapping sites from the other network onto `S-Net`, as in earlier

work [42].

For multi-priority experiments, we partition traffic into three priorities based on their source services. Following SWAN [42], high priority is for interactive services, which are highly sensitive to loss and delay; medium priority is for services that are less sensitive but are still impacted by packet loss (e.g., deadline-driven transfers); and low priority is for background services (e.g., large replication jobs).

To understand the impact of network provisioning level, we study three cases. The first is a well-utilized network where capacity matches demand. To mimic this, we scale the demands of all flows (uniformly) such that 99% of demands per interval are satisfied. The results below refer to this case as traffic scale of 1. The other two cases are a well-provisioned network and an under-provisioned network, and we use traffic scales of 0.5 and 2 to mimic them.

In terms of relationship to practice, ISP networks are typically well-provisioned and use single priority traffic. Inter-datacenter networks are well-utilized and use multiple priorities to protect high priority traffic from short-term demand increases of lower priority traffic [42, 44].

Failures and switch update models: For `L-Net`, we inject data plane faults as per logs from the network. For `S-Net`, we inject faults based on the per-link and per-switch failure probabilities derived from `L-Net` logs. We assume that it takes 5 ms for a switch to detect a link failure, and the time it takes for an ingress switch to hear about the failure and rescale depends on the propagation delay.

We consider two models of switch update behaviors. In the `Realistic` model, we use the update delay distribution reported for B4 (§2.2.3) and a configuration failure rate of 1%. In the `Optimistic` model, we use the update delay distribution that we measured in a controlled environment (§2.2.3) and a configuration failure rate of 0%.

TE approaches: We compare TE with and without FFC. Without FFC, when a link or

switch fails, the TE controller immediately computes a new traffic distribution and updates the network. With FFC, the controller does not react to data plane faults unless it is on the edge of protection level. If the link protection level k_e=2, the controller reacts only after 2 links have failed. reaction logic is per-priority; when a given priority traffic is at the edge of protection level, only that traffic is re-adjusted. Both approaches use the same set of tunnels. We use (1, 3)-link-switch disjoint strategy to establish six tunnels for each flow. Except in the microbenchmark experiments, if a flow's demand is not satisfied in an interval, the remaining bytes add to its demand in the next interval.

We evaluated both max-throughput and max-min fairness as TE objectives but present results only for the former. Results for the latter are qualitatively similar.

Metrics: We use two metrics to capture the behavior of FFC:

i) *Throughput ratio*: Network throughput with FFC versus without FFC. One minus this ratio is the overhead of FFC.

ii) *Data loss ratio*: Bytes lost with FFC versus without FFC. We count bytes lost due to both blackholes and congestion. Blackhole losses occur during the time between when a link fails and when ingress switches rescale; all packets that traverse the failed link are considered lost in this period. Congestion losses are computed based on link capacity and the duration and degree to which the link is oversubscribed. This simple measure overestimates actual losses if congestion control is used at the hosts, as they will reduce sending rates in response to early losses. But it serves as a good proxy for capturing the intensity and duration of congestion [42].

2.8.2 Microbenchmarks

Throughput overhead To microbenchmark the overhead of FFC, we compute traffic distributions with and without it for successive TE intervals. In each interval, we exclude

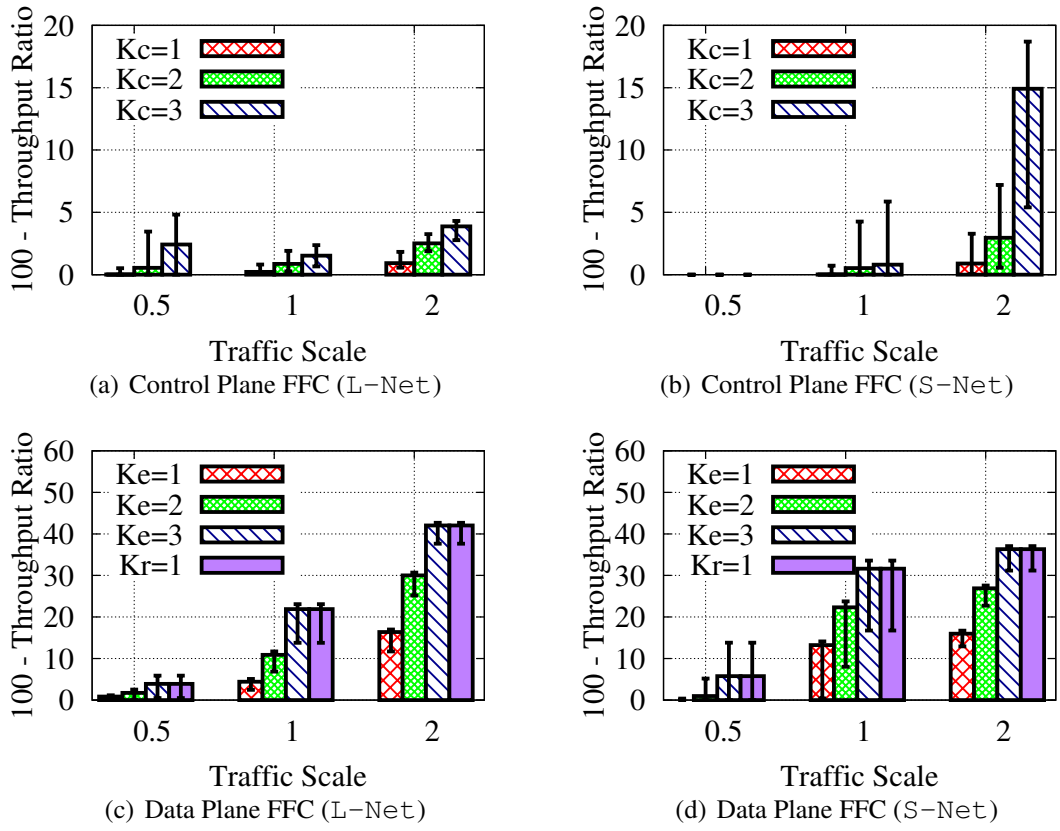


Figure 2.13: Throughput overhead of FFC. Bars show the 90th %-ile value and error bars show the 50th and 99th %-iles.

any unfinished data from previous intervals, so that both approaches operate on the same demand in each interval, independent of preceding allocations. Failure and switch models do not impact these experiments as we do not inject faults but study overhead when no faults happen.

Figure 2.13(a) and 2.13(b) show the overhead (1 - throughput ratio) of control plane FFC (by itself; no data plane FFC) for three protection levels. For each traffic scale, it plots three percentile values. We see that the overhead of control plane FFC is small—under 5% even at 90th percentile level for all but one settings–and, expectedly, increases with the protection level. We also see that the overhead generally increases with the traffic scale. As the network gets busier, it becomes harder to accommodate all traffic in a way

	FFC (3, 3, 0) ∪ (3, 0, 1)	FFC (2, 1, 0)	Without FFC
L-Net	1.2 sec	0.3 sec	0.05 sec
S-Net	0.03 sec	0.02 sec	0.015 sec

Table 2.2: TE computation time with and without FFC.

that modifies existing traffic spread robustly. The results are similar for L-Net and S-Net.

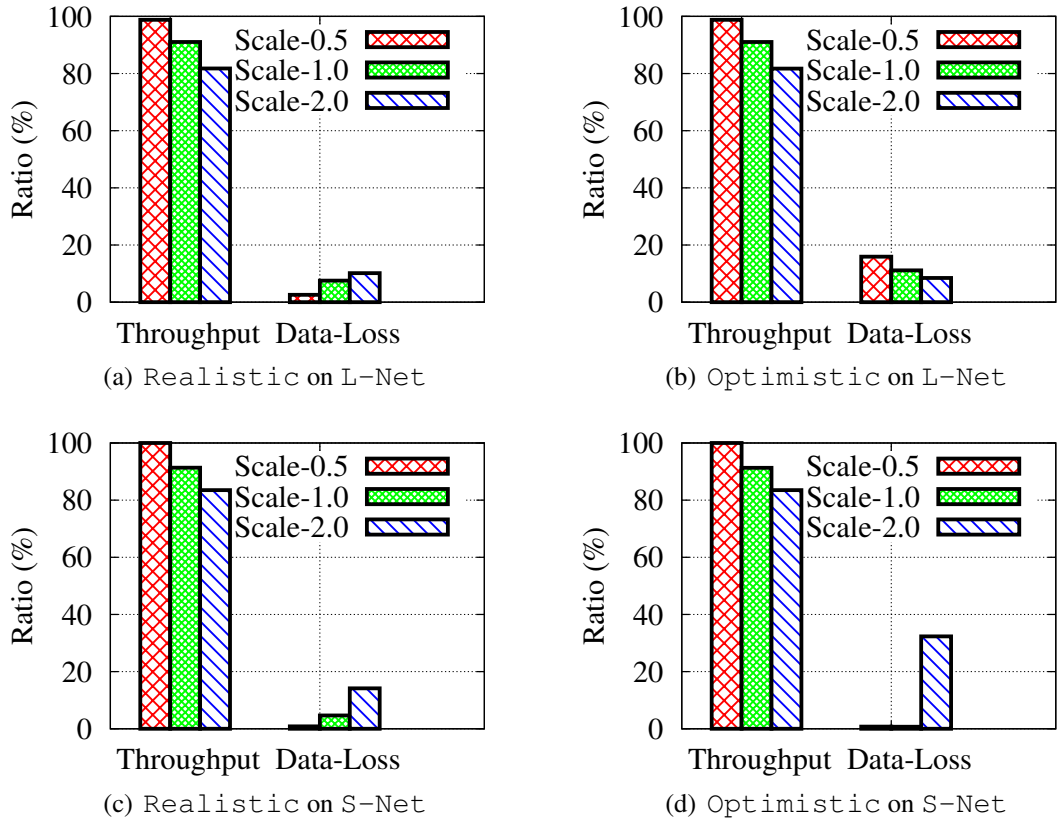
Figure 2.13(c) and 2.13(d) show the overhead of data plane FFC for 1-3 links failures and 1 switch failure. Because we use (1, 3)-link-switch disjoint strategy to construct tunnels, the cases with $k_e = 3$ and $k_v = 1$ are identical. We see that the overhead is low at traffic scale 0.5 (which implies a well-provisioned network), but it grows quickly as traffic scales and protection level increases.

Based on these results and that multiple link failures in a short amount of time and switch failures are uncommon (but they do occur), we recommend using a protection level of $(k_c, k_e, k_v) = (2, 1, 0)$ in single-priority networks. In multi-priority networks, we recommend a higher protection level for high-priority flows. As the relative volume of this traffic is typically under 50%, FFC's overhead will still be small.

Computation time We benchmark computation time on an ordinary PC with Intel i5 M540 2.53 Ghz CPU (2 cores) and 4GB RAM. Table 2.2 lists the average computation time for FFC with different protection levels and without FFC. Because of the imprecision of Eqn. 2.15 (§2.4.5), FFC configuration of $(3, 3, 0)$ simultaneously provides protection for $(3, 0, 1)$. We use $(3, 0, 1) \cup (3, 3, 0)$ as a shorthand for this combined protection level. We see that even at a high protection level, FFC computation takes only 1.2 seconds for large network like L-Net.

2.8.3 Single-priority traffic

We now perform an end-to-end evaluation of FFC with realistic failure and switch models. This section focuses on the single-priority case, and the next on multi-priority case. Per



(a) `Realistic` on `L-Net` (b) `Optimistic` on `L-Net`

(c) `Realistic` on `S-Net` (d) `Optimistic` on `S-Net`

Figure 2.14: Throughput and data loss ratio for FFC with single traffic priority.

above, we configure FFC as (k_c, k_e, k_v) =$(2, 1, 0)$.

Figure 2.14 shows the results for the two networks, the two switch models, and the three traffic scales. Focusing first on the case of well-provisioned networks (traffic scale 0.5), which is the common case for single-priority traffic, we see that throughput difference with and without FFC is negligible. At the same time, FFC offers 10∼20 times reduction in data loss. We do not report absolute amount of lost data to maintain confidentiality for link capacities in `L-Net`. But we note that the loss is substantial (well above $O(100GB)$ per day), and in almost all cases of link oversubscription, the amount of data lost is well above typical buffer capacities.

For the case of well-utilized networks (traffic scale 1), we see that FFC carries over 90% of the traffic carried without FFC and cuts data loss to 0.72∼11.5%.

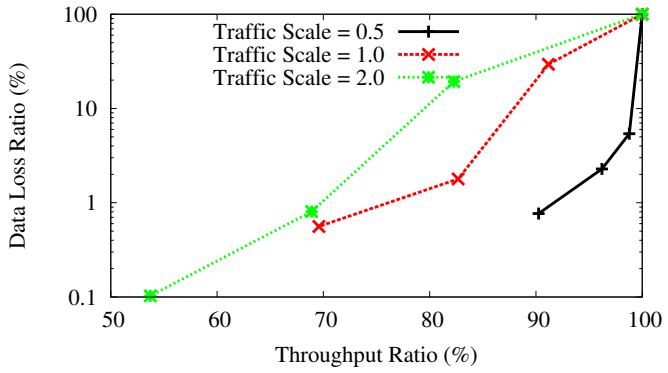
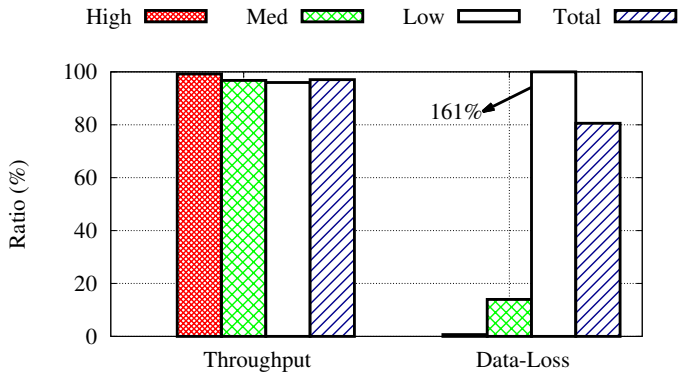


Figure 2.15: The tradeoff of data loss and throughput.

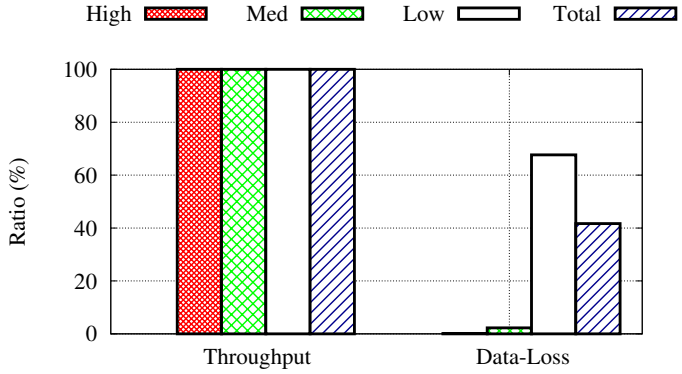
Closer inspection of lost data reveals that both with and without FFC, blackhole losses, due to delay in rescaling after a link failure, are negligible. With FFC, the primary factor behind losses is cases where the number of data plane faults is greater than the protection level. Without FFC, any control and data plane fault leads to congestive losses, and both types of faults contribute roughly equally.

Though the absolute amount of data loss is lower for the `Optimistic` switch model, we observe in Figure 2.14 that the relative gain of FFC is similar for both switch models. Thus, FFC helps even if switch updates times could be improved to those in controlled environments today and all configuration failures could be eliminated.

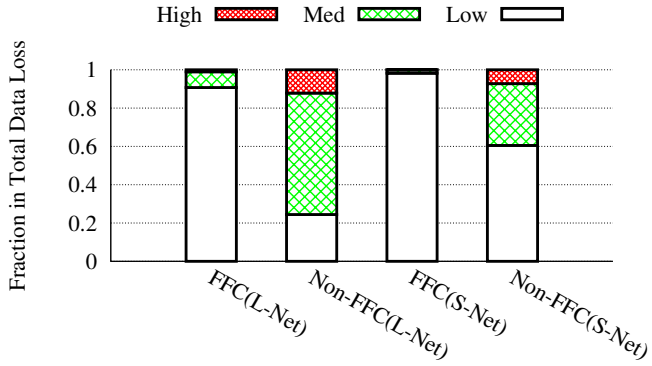
Trade-off between throughput and data loss Using link failures as an example, Figure 2.15 shows the tradeoff between data loss and throughput overhead as we change the protection level. This experiment uses the `Realistic` switch model and no protection from control plane faults or switch failures. For each traffic scale, the point at (100, 100) corresponds to no protection ($k_e = 0$) and successive points to the left correspond to increasing values of link protection. Expectedly, the less data we want to lose, the higher the throughput overhead, though throughput overhead grows slower than loss reduction (linear versus exponential).



(a) Throughput and data Loss in `L-Net`



(b) Throughput and data loss in `S-Net`



(c) The fractions of data loss

Figure 2.16: FFC with multiple priorities.

2.8.4 Multi-priority traffic

We now consider networks with multi-priority traffic (§2.5.1). Here, FFC offers the opportunity to provide greater protection for high-priority traffic with minimal loss in total throughput because low-priority traffic can be safely carried over the capacity that is set

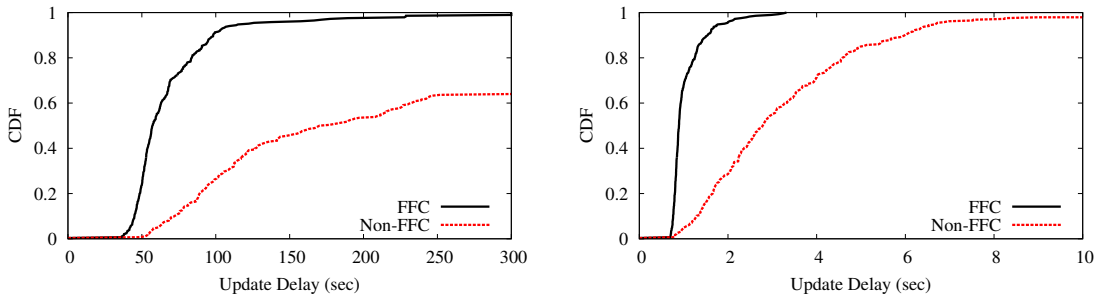
aside to protect high-priority traffic. When congestion occurs, priority queueing, which preferentially drops lower-priority packets, protects high-priority traffic as long as its rate does not exceed link capacity.

We use different protection levels for different priorities: (k_c, k_e, k_v) =$(3,0,1) \cup (3,3,0)$ for high-priority traffic to provide it strong protection, $(2,1,0)$ for medium priority, and $(0,0,0)$ for low priority. Thus, low priority traffic is not protected at all, which lets it use all network capacity.

Figure 2.16 shows the results for both our networks. This experiment uses traffic scale of 1 (well-utilized network). From Figures 2.16(a) and (b), we see that the throughput ratio is close to 100%, for total traffic as well for individual priorities. Recall that in a single-priority network, throughput ratio is 90% and loss ratio is 0.72$\sim$11.5% for this traffic scale. Given the basic FFC trade-off, the increase in total throughput for this multi-priority network must accompany a decrease in protection from congestion. The loss ratio for total traffic bears this out; it is 40$\sim$80%.

What is interesting, however, are the data loss ratios of different priorities. The high-priority traffic suffers almost no loss, and the loss has been concentrated towards low-priority traffic. The effect is extreme for `L-Net`, where low-priority traffic looses more bytes with FFC than without FFC.

Figure 2.16(c) shows the relative fraction of lost bytes for each priority. With FFC, there is negligible loss for high-priority traffic and a small amount (2$\sim$7%) of loss for medium-priority traffic. In contrast, without FFC, 5$\sim$15% of the bytes lost are high-priority and 30$\sim$70% are medium-priority. Thus, the use of priority queueing by itself is not sufficient for preventing congestion losses for high priority traffic. FFC can provide strong protection without loss in throughput.



(a) `Realistic` (b) `Optimistic`

Figure 2.17: Update time for congestion-free updates.

2.8.5 Congestion-free network updates

We now consider the case of congestion-free, multi-step updates. Recall that in this setting, control plane faults do not lead to congestion losses, but network updates can be slow and can even stall. We evaluate the speed of network updates with and without FFC.

Figure 2.17 shows the results for `L-Net` for both switch models. With the `Realistic` model, without FFC, 40% of the updates do not finish within 300 seconds. Since that is the TE interval, it is the maximum time we wait for the update to finish. This poor performance stems from the fact that even a single switch that takes a long time or fails to update altogether hurts the update process. FFC allows for faster updates by being robust to a small number of control plane faults (in this case k_c=2). FFC provides significant reduction in update time even with the `Optimistic` model, in which there are no configuration failures, only occasional delays. The median and 99th percentile speedup is a factor of three.

Faster updates would lead to a more nimble network that can quickly react to demands bursts. They can also improve throughput by enabling a shorter TE interval, which enables the network to handle shorter-term demand variations.

2.9 Related Work

We build upon the rich line of work in TE algorithms and systems. Researchers have studied various aspects of this problem, including $i)$ how to implement close-to-optimal traffic distributions in different settings such as link-state routing [81], MPLS environments [34], and SDNs [25, 42, 44, 71]; $ii)$ how to stably adapt to changing traffic demands [46, 54]; $iii)$ how to reroute traffic after failures such that minimal changes are needed [16]; and $iv)$ how to find efficient backup paths [48, 55].

For brevity, we discuss below only proactive techniques to make TE robust. Our key contributions in this space are that, to our knowledge, we are the first to handle control plane faults and to proactively handle data plane faults in a way that is both robust to any combination of up to k failures and works with today's commodity switches.

Data plane faults: To prevent rescaling-induced congestion after a data plane fault, Suchara *et al.* [73] modify the ingress switch's rescaling behavior. Instead of simple proportional rescaling, tunnel splitting weights are based on the set of residual tunnels. These weights are pre-computed and configured at the switch. Unlike our data plane FFC, which protects against any combination of up to k faults, this approach can handle only a modest number of potential failure cases as there are exponential number of residual tunnel sets.

For a distributed TE setting, R3 [78] proposes an approach for congestion-free fast reroute (FRR) [10], in which adjacent routers route around failed links. The routing behavior is determined by a fast online computation, aided by offline computation that is done ahead of time. Like FFC, R3 protects against any combination of up to k link failures. A key difference is that we focus on rescaling at ingress switches versus fast reroute at link-adjacent routers. This difference induces different types of constraints (and thus require a different solution technique).

Further, both works above require changes to the switch hardware or software; we

build solely on existing primitives.

Demand uncertainty: In networks where incoming traffic rate is not controlled, oblivious routing [17] and COPE [77] aim to find TE configurations that are robust to changes in traffic demand or errors in demand prediction. These works do not consider control and data plane faults which is the focus of our work. An interesting area of future investigation is if our approach for handling control plane faults in rate limiters, which induces uncertainty in traffic entering the network, can be extended to tackle demand uncertainty. That would enable a common framework for handling both faults and demand uncertainty.

2.10 Summary

We developed FFC methods that proactively protect a network from congestion and packet loss due to data and control plane faults. These methods have low overhead in terms of network throughput—even optimal in some cases—and are computationally efficient. Using testbed experiments and data from real networks, we showed how FFC is useful in a variety of settings. For instance, in well-provisioned networks, it can reduce packet loss by a factor of 7$\sim$130; in well-utilized networks that carry traffic with multiple priorities, it can reduce loss for high-priority traffic to almost zero, with negligible reduction in total network throughput.

Chapter 3

zUpdate: Updating Data Center Networks with Zero Loss

Datacenter networks (DCNs) are constantly evolving due to various updates such as switch upgrades and VM migrations. Each update must be carefully planned and executed in order to avoid disrupting many of the mission-critical, interactive applications hosted in DCNs. The key challenge arises from the inherent difficulty in synchronizing the changes to many devices, which may result in unforeseen transient link load spikes or even congestions. We present one primitive, `zUpdate`, to perform congestion-free network updates under asynchronous switch and traffic matrix changes. We formulate the update problem using a network model and apply our model to a variety of representative update scenarios in DCNs. We develop novel techniques to handle several practical challenges in realizing `zUpdate` as well as implement the `zUpdate` prototype on OpenFlow switches and deploy it on a testbed that resembles real DCN topology. Our results, from both real-world experiments and large-scale trace-driven simulations, show that `zUpdate` can effectively perform congestion-free updates in production DCNs.

3.1 Introduction

The rise of cloud computing platform and Internet-scale services has fueled the growth of large datacenter networks (DCNs) with thousands of switches and hundreds of thousands of servers. Due to the sheer number of hosted services and underlying physical devices, *DCN updates* occur frequently, whether triggered by the operators, applications, or sometimes even failures. For example, DCN operators routinely upgrade existing switches or onboard new switches to fix known bugs or to add new capacity. For applications, migrating VMs or reconfiguring load balancers are considered the norm rather than the exception.

Despite their prevalence, DCN updates can be challenging and distressing even for the most experienced operators. One key reason is because of the complex nature of the updates themselves. An update usually must be performed in multiple steps, each of which is well planned to minimize disruptions to the applications. Each step can involve changes to a myriad of switches, which if not properly coordinated may lead to catastrophic incidents. Making matters even worse, there are different types of update with diverse requirements and objectives, forcing operators to develop and follow a unique process for each type of update. Because of these reasons, a DCN update may take hours or days to carefully plan and execute while still running the risk of spiraling into operators' nightmare.

This stark reality calls for a simple yet powerful abstraction for DCN updates, which can relieve the operators from the nitty-gritty, such as deciding which devices to change or in what order, while offering seamless update experience to the applications. We identify three essential properties of such an abstraction. First, it should provide a simple interface for operators to use. Second, it should handle a wide range of common update scenarios. Third, it should provide certain levels of guarantee which are relevant to the applications.

The seminal work by Reitblatt *et al.* [69] introduces two abstractions for network updates: *per-packet* and *per-flow* consistency. These two abstractions guarantee that a packet

or a flow is handled either by the old configuration before an update or by the new configuration after an update, but never by both. To implement such abstractions, they proposed a two-phase commit mechanism which first populates the new configuration to the middle of the network and then flips the packet version numbers at the ingress switches. These abstractions can preserve certain useful trace properties, e.g., loop-free during the update, as long as these properties hold before and after the update.

While the two abstractions are immensely useful, they are not the panacea for all the problems during DCN updates. In fact, a DCN update may trigger network-wide traffic migrations, in which case many flows' configurations have to be changed. Because of the inherent difficulty in synchronizing the changes to the flows from different ingress switches, the link load during an update could get significantly higher than that before or after the update (see example in §3.3). This problem may further exacerbate when the application traffic is also fluctuating independently from the changes to switches. As a result, nowadays operators are completely in the dark about how badly links could be congested during an update, not to mention how to come up with a feasible workaround.

This chapter introduces one key primitive, `zUpdate`, to perform congestion-free network-wide traffic migration during DCN updates. The letter "z" means zero loss and zero human effort. With `zUpdate`, operators simply need to describe the end requirements of the update, which can easily be converted into a set of input constraints to `zUpdate`. Then `zUpdate` will attempt to compute and execute a sequence of steps to progressively meet the end requirements from an initial traffic matrix and traffic distribution. When such a sequence is found, `zUpdate` guarantees that there will be no congestion throughout the update process. We demonstrate the power and simplicity of `zUpdate` by applying it to several realistic, complex update scenarios in large DCNs.

To formalize the traffic migration problem, we present a network model that can precisely describe the relevant state of a network — specifically the traffic matrix and traffic

distribution. This model enables us to derive the sufficient and necessary conditions under which the transition between two network states will not incur any congestion. Based on that, we propose an algorithm to find a sequence of lossless transitions from an initial state to an end state which satisfies the end constraints of an update. We also illustrate by examples how to translate the high-level human-understandable update requirements into the corresponding mathematical constraints which are compliant with our model.

`zUpdate` can be readily implemented on existing commodity OpenFlow switches. One major challenge in realizing `zUpdate` is the limited flow and group table sizes on those switches. Based on the observation that ECMP works sufficiently well for most of the flows in a DCN [80], we present an algorithm that greedily consolidates such flows to make efficient use of the limited table space. Furthermore, we devise heuristics to reduce the computation time and the switch update overhead.

We summarize our contributions as follows:

- We introduce the `zUpdate` primitive to perform congestion-free network updates under asynchronous switch and traffic matrix changes.
- We formalize the network-wide traffic migration problem using a network model and propose a novel algorithm to solve it.
- We illustrate the power of `zUpdate` by applying it to several representative update scenarios in DCNs.
- We handle several practical challenges, *e.g.* switch table size limit and computation complexity, in implementing `zUpdate`.
- We build a `zUpdate` prototype on top of OpenFlow [6] switches and Floodlight controller [2].

- We extensively evaluate `zUpdate` both on a real network testbed and in large-scale simulations driven by the topology and traffic demand from a large production DCN.

3.2 Datacenter Network

Topology: A state-of-the-art DCN typically adopts a FatTree or Clos topology to attain high bisection bandwidth between servers. Figure 3.2(a) shows an example in which the switches are organized into three layers from the top to the bottom: *Core*, *Aggregation (Agg)* and *Top-of-Rack (ToR)*. Servers are connected to the ToRs.

Forwarding and routing: In such a hierarchical network, traffic traverses a valley-free path from one ToR to another: first go upwards and then downwards. To limit the forwarding table size, the servers under the same ToR may share one IP prefix and forwarding is performed based on each ToR's prefix. To fully exploit the redundant paths, each switch uses ECMP to evenly split traffic among multiple next hops. The emergence of Software Defined Networks (SDN) allows the forwarding tables of each switch to be directly controlled by a logically centralized controller, *e.g.*, via the OpenFlow APIs, dramatically simplifying the routing in DCNs.

Flow and group tables on commodity switches: An (OpenFlow) switch forwards packets by matching packet headers, *e.g.*, source and destination IP addresses, against entries in the so called *flow table* (Figure 3.6). A flow entry specifies a pattern used for matching and actions taken on matching packets. To perform multipath forwarding, a flow entry can direct a matching packet to an entry in a *group table*, which can further direct the packet to one of its multiple next hops by hashing on the packet header. Various hash functions may be used to implement different load balancing schemes, such as ECMP or Weighted-Cost-Multi-Path (WCMP). To perform pattern matching, the flow table is made of TCAM (Ternary Content Addressable Memory) which is expensive and power-hungry. Thus, the

commodity switches have limited flow table size, usually between 1K to 4K entries. The group table typically has 1K entries.

3.3 Network Update Problem

Scenario	Description
VM migration	Moving VMs among physical servers.
Load balancer reconfiguration	Changing the mapping between a load balancer and its backend servers.
Switch firmware upgrade	Rebooting a switch to install new version of firmware.
Switch failure repair	Shutting down a faulty switch to prevent failure propagation.
New switch onboarding	Moving traffic to a new switch to test its functionality and compatibility.

Table 3.1: The common update scenarios in production DCNs.

We surveyed the operators of several production DCNs about the typical update scenarios and listed them in Table 3.1. One common problem that makes these DCN updates hard is they all have to deal with so called *network-wide traffic migration* where the forwarding rules of many flows have to be changed. For example in switch firmware upgrade, in order to avoid impacting the applications, operators would move all the traffic away from a target switch before performing the upgrade. Taking VM migration as another example, to relocate a group of VM's, all the traffic associated with the VM's will be migrated as well.

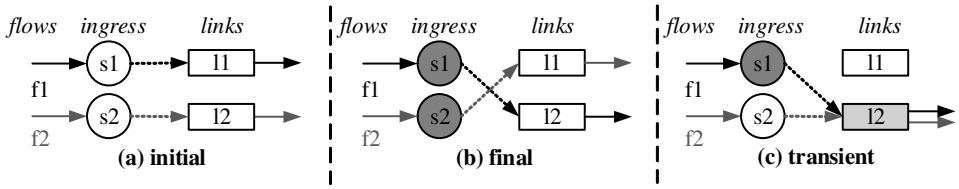


Figure 3.1: Transient load increase during traffic migration.

Such network-wide traffic migration, if not done properly, could lead to severe conges-

tion. The fundamental reason is because of the difficulty in synchronizing the changes to the flows from different ingress switches, causing certain links to carry significantly more traffic during the migration than before or after the migration. We illustrate this problem using a simple example in Figure 3.1. Flows f_1 and f_2 enter the network from ingress switches s_1 and s_2 respectively. To move the traffic distribution from the initial one in (a) to the final one in (b), we need to change the forwarding rules in both s_1 and s_2. As shown in (c), link l_2 will carry the aggregate traffic of f_1 and f_2 if s_1 is changed before s_2. Similar problem will occur if s_2 is changed first. In fact, this problem cannot be solved by the two-phase commit mechanism proposed in [69].

A modern DCN hosts many interactive applications such as search and advertisement which require very low latencies. Prior research [15] reported that even small losses and queuing delays could dramatically elevate the flow completion time and impair the user-perceived performance. Thus, it is critical to avoid congestion during DCN updates.

Performing lossless network-wide traffic migration can be highly tricky in DCNs, because it often involves changes to many switches and its impact can ripple throughout the network. To avoid congestion, operators have to develop a thoughtful migration plan in which changes are made step-by-step and in an appropriate order. Furthermore, certain update (*e.g.*, VM migration) may require coordination between servers and switches. Operators, thus, have to carefully calibrate the impact of server changes along with that of switch changes. Finally, because each update scenario has its distinctive requirements, operators today have to create a customized migration plan for each scenario.

Due to the reasons above, network-wide traffic migration is an arduous and complicated process which could take weeks for operators to plan and execute while some of the subtle yet important corner cases might still be overlooked. Thus, risk-averse operators sometimes deliberately defer an update, *e.g.*, leaving switches running an out-of-date, buggy firmware, because the potential damages from the update may outweigh the gains.

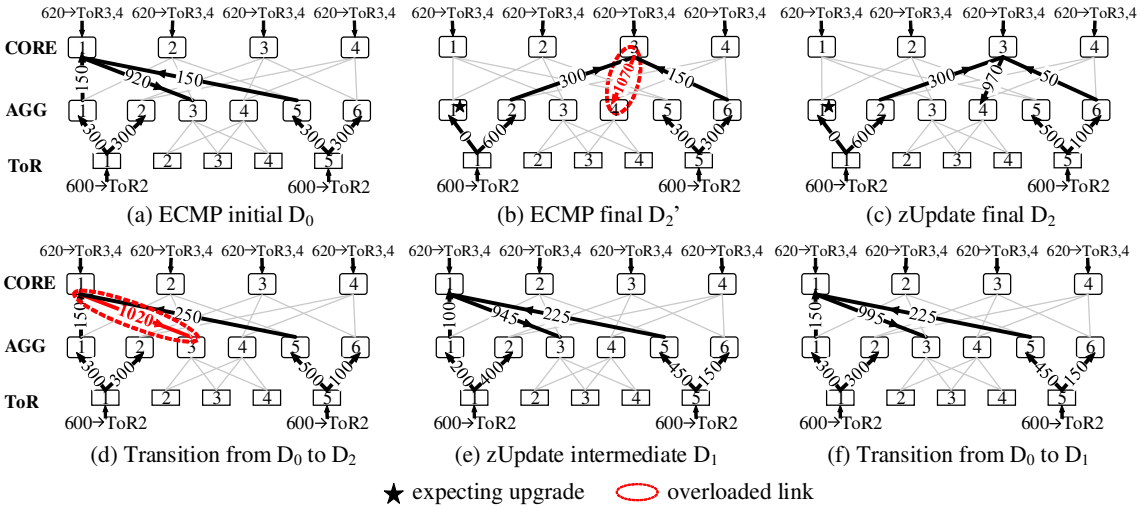


Figure 3.2: This example shows how to perform a lossless firmware upgrade through careful traffic distribution transitions.

Such tendency would severely hurt the efficiency and agility of the whole DCN.

Our goal: is to provide a primitive called `zUpdate` to manage the network-wide traffic migration for all the DCN updates shown in Table 3.1. In our approach, operators only need to provide the end requirements of a specific DCN update, and then `zUpdate` will automatically handle all the details, including computing a lossless (perhaps multi-step) migration plan and coordinating the changes to different switches. This would dramatically simplify the migration process and minimize the burden placed on operators.

3.4 Overview

In this section, we illustrate by two examples how asynchronous switch and traffic matrix changes lead to congestion during traffic migration in DCN and how to prevent the congestion through a carefully-designed migration plan.

Switch firmware upgrade: Figure 3.2(a) shows a FatTree [14] network where the capacity of each link is 1000. The numbers above the core switches and those below the ToRs are traffic demands. The number on each link is the traffic load and the arrow indi-

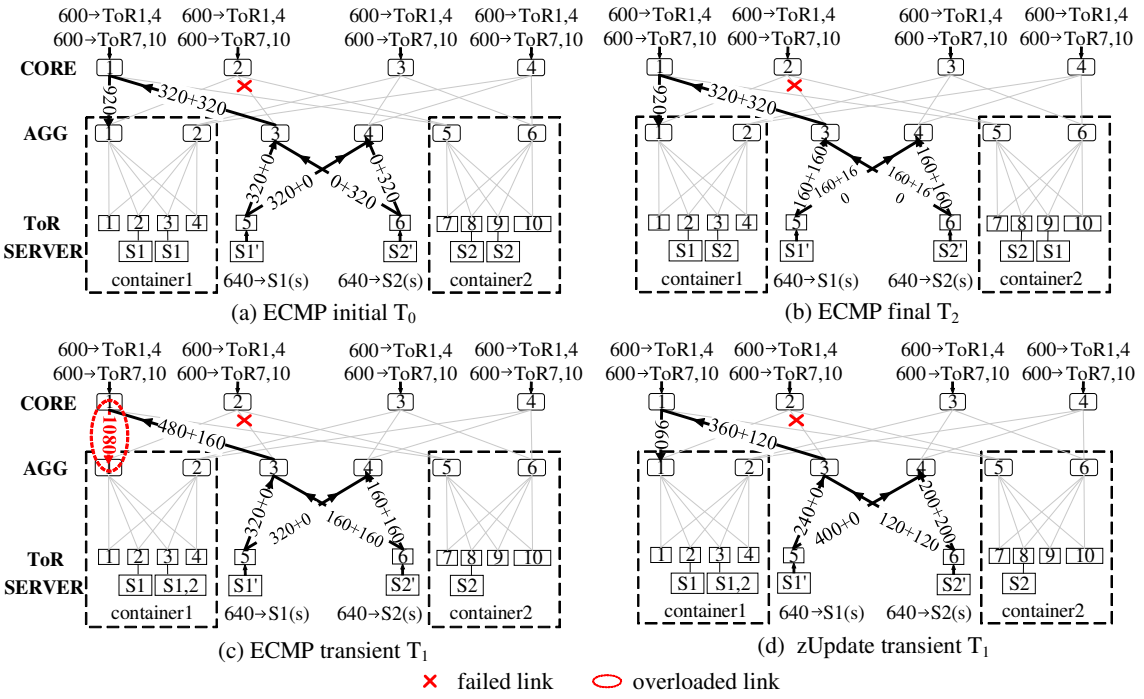


Figure 3.3: This example shows how to avoid congestion by choosing the proper traffic split ratios for switches.

cates the traffic direction. This figure shows the initial traffic distribution D_0 where each switch uses ECMP to evenly split the traffic among the next hops. For example, the load on link $ToR_1 \rightarrow AGG_1$ is 300, half of the traffic demand $ToR_1 \rightarrow ToR_2$. The busiest link $CORE_1 \rightarrow AGG_3$ has a load of 920, which is the sum of 620 (demand $CORE_1 \rightarrow ToR_{3/4}$), 150 (traffic $AGG_1 \rightarrow CORE_1$), and 150 (traffic $AGG_5 \rightarrow CORE_1$). No link is congested.

Suppose we want to move the traffic away from AGG_1 before taking it down for firmware upgrade. A naive way is to disable link $ToR_1 \rightarrow AGG_1$ so that all the demand $ToR_1 \rightarrow ToR_2$ shifts to link $ToR_1 \rightarrow AGG_2$ whose load becomes 600 (as shown in Figure 3.2(b)). As a result, link $CORE_3 \rightarrow AGG_4$ will have a load of 1070 by combining 300 (traffic $AGG_2 \rightarrow CORE_3$), 150 (traffic $AGG_6 \rightarrow CORE_3$), and 620 (demand $CORE_3 \rightarrow ToR_{3/4}$), exceeding its capacity.

Figure 3.2(c) shows the preceding congestion can be prevented through proper traffic distribution D_2, where ToR_5 forwards 500 traffic on link $ToR_5 \rightarrow AGG_5$ and 100 traffic on

link $ToR_5 \rightarrow AGG_6$ instead of using ECMP. This reduces the load on link $CORE_3 \rightarrow AGG_4$ to 970, right below its capacity.

However, to transition from the initial D_0 (Figure 3.2(a)) to D_2 (Figure 3.2(c)), we need to change the traffic split ratio on both ToR_1 and ToR_5. Since it is hard to change two switches simultaneously, we may end up with a traffic distribution shown in Figure 3.2(d) where link $CORE_1 \rightarrow AGG_3$ is congested, when ToR_5 is changed before ToR_1. Conversely, if ToR_1 is changed first, we will have the traffic distribution $D_2^{'}$ in Figure 3.2(b) where link $CORE_3 \rightarrow AGG_4$ is congested.

Given the asynchronous changes to different switches, it seems impossible to transit from D_0 (Figure 3.2(a)) to D_2 (Figure 3.2(c)) without causing any loss. Our basic idea is to introduce an intermediate traffic distribution D_1 as a stepping stone, such that the transitions $D_0 \rightarrow D_1$ and $D_1 \rightarrow D_2$ are both lossless. Figure 3.2(e) is such an intermediate D_1 where ToR_1 splits traffic by 200:400 and ToR_5 splits traffic by 450:150. It is easy to verify that no link is congested in D_1 since the busiest link $CORE_1 \rightarrow AGG_3$ has a load of 945.

Furthermore, when transitioning from D_0 (Figure 3.2(a)) to D_1, no matter in what order ToR_1 and ToR_5 are changed, there will be no congestion. Figure 3.2(f) gives an example where ToR_5 is changed before ToR_1. The busiest link $CORE_1 \rightarrow AGG_3$ has a load of 995. Although not shown here, we verified that there is no congestion if ToR_1 is changed first. Similarly, the transition from D_1 (Figure 3.2(e)) to D_2 (Figure 3.2(c)) is lossless regardless of the change order of ToR_1 and ToR_5.

In this example, the key challenge is to find the appropriate D_2 (Figure 3.2(c)) that satisfies the firmware upgrade requirement (moving traffic away from AGG_1) as well as the appropriate D_1 (Figure 3.2(e)) that bridges the lossless transitions from D_0 (Figure 3.2(a)) to D_2. We will explain how `zUpdate` computes the intermediate and final traffic distributions in §3.5.3.

Load balancer reconfiguration: Figure 3.3(a) shows another FatTree network where the link capacity remains 1000. One of the links $\text{CORE}_2 \leftrightarrow \text{AGG}_3$ is down (a common incident in DCN). Two servers $S_1^{'}$ and $S_2^{'}$ are sending traffic to two services S_1 and S_2, located in Container_1 and Container_2 respectively. The labels on some links, for example, $\text{ToR}_5 \rightarrow \text{AGG}_3$, are in the form of "$l_1 + l_2$", which indicate the traffic load towards Container_1 and Container_2 respectively.

Suppose all the switches use ECMP to forward traffic, Figure 3.3(a) shows the traffic distribution under the initial traffic matrix T_0 where the load on the busiest link $\text{CORE}_1 \rightarrow \text{AGG}_1$ is 920, which is the sum of 600 (the demand $\text{CORE}_1 \rightarrow \text{ToR}_{1/4}$) and 320 (the traffic on link $\text{AGG}_3 \rightarrow \text{CORE}_1$ towards Container_1). No link is congested.

To be resilient to the failure of a single container, we now want S_1 and S_2 to run in both Container_1 and Container_2. For S_2, we will instantiate a new server under ToR_3 and reconfigure its *load balancer (LB)* (not shown in the figure) to shift half of its load from ToR_9 to ToR_3. For S_1, we will take similar steps to shift half of its load from ToR_3 to ToR_9. Figure 3.3(b) shows the traffic distribution under the final traffic matrix T_2 after the update. Note that the traffic on link $\text{ToR}_5 \rightarrow \text{AGG}_3$ is "160+160" because half of it goes to the S_1 under ToR_2 in Container_1 and the other half goes to the S_1 under ToR_9 in Container_2. It is easy to verify that there is no congestion.

However, the reconfiguration of the LBs of S_1 and S_2 usually cannot be done simultaneously because they reside on different devices. Such asynchrony may lead to a transient traffic matrix T_1 shown in Figure 3.3(c) where S_2' LB is reconfigured before S_1's. This causes link $\text{ToR}_6 \rightarrow \text{AGG}_3$ to carry "160+160" traffic, half of which goes to the S_2 in Contain_1, and further causes congestion on link $\text{CORE}_1 \rightarrow \text{AGG}_1$. Although not shown here, we have verified that congestion will happen if S_1's LB is reconfigured first.

The congestion above is caused by asynchronous traffic matrix changes. Our basic idea to solve this problem is to find the proper traffic split ratios for the switches such that

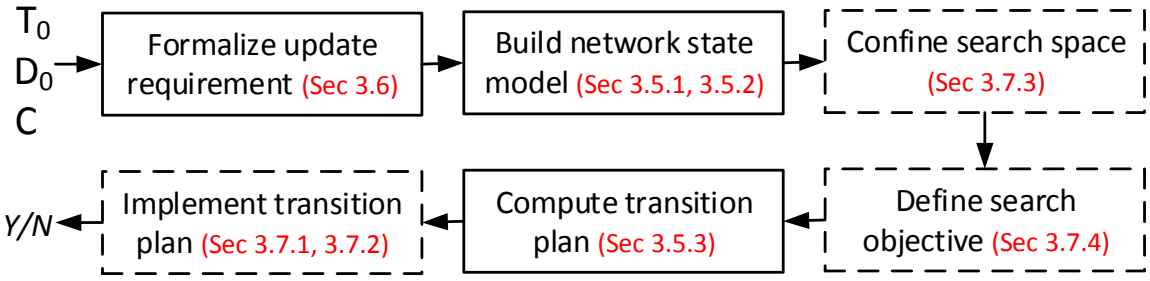


Figure 3.4: The high-level working process of `zUpdate`.

there will be no congestion under the initial, final or any possible transient traffic matrices during the update. Figure 3.3(d) shows one such solution where ToR_5 and ToR_6 send 240 traffic to AGG_3 and 400 traffic to AGG_4, and other switches still use ECMP. The load on the busiest link $\text{CORE}_1 \rightarrow \text{AGG}_1$ now becomes 960 and hence no link is congested under T_1. Although not shown here, we have verified that, given such traffic split ratios, the network is congestion-free under the initial T_0 (Figure 3.3(a)), the final T_2 (Figure 3.3(b)), and the transient traffic matrix where S_1's LB is reconfigured first.

Generally, the asynchronous reconfigurations of multiple LBs could result in a large number of possible transient traffic matrices, making it hard to find the proper traffic split ratios for all the switches. We will explain how to solve this problem with `zUpdate` in §3.6.2.

The `zUpdate` process: We provide `zUpdate`(T_0, D_0, $\mathcal{C}$) to perform lossless traffic migration for DCN updates. Given an initial traffic matrix T_0, `zUpdate` will attempt to compute a sequence of lossless transitions from the initial traffic distribution D_0 to the final traffic distribution D_n which satisfies the update requirements $\mathcal{C}$. D_n would then allow an update, e.g., upgrading switch or reconfiguring LB, to be executed without incurring any loss.

Figure 3.4 shows the overall workflow of `zUpdate`. In the following, we will first present a network model for describing lossless traffic distribution transition (§3.5.1, 3.5.2)

and an algorithm for computing a lossless transition plan (§3.5.3). We will then explain how to represent the constraints in each update scenario (§3.6). After that, we will show how to implement the transition plan on switches with limited table size (§3.7.1, 3.7.2), reduce computational complexity by confining search space (§3.7.3), and reduce transition overhead by picking a proper search objective function (§3.7.4).

3.5 Network Model

This section describes a network model under which we formally define the traffic matrix and traffic distribution as the inputs to `zUpdate`. We use this model to derive the sufficient and necessary conditions for a lossless transition between two traffic distributions. In the end, we present an algorithm for computing a lossless transition plan using an optimization programming model.

3.5.1 Abstraction of traffic distribution

Network, flow and traffic matrix: A network is a directed graph $G = (V, E)$, where V is the set of switches, and E is the set of links between switches. A flow f enters the network from an ingress switch (s_f) and exits at an egress switch (d_f) through one or multiple paths (p_f). Let G_f be the subgraph formed by all the p_f's. For instance, in Figure 3.5, suppose f takes the two paths (1, 2, 4, 6, 8) and (1, 2, 4, 7, 8) from switch$_1$ to switch$_8$, then G_f is comprised of switches $\{1, 2, 4, 6, 7, 8\}$ and the links between them. A traffic matrix T defines the size of each flow T_f.

Traffic distribution: Let $l^f_{v,u}$ be f's traffic load on link $e_{v,u}$. We define $D = \{l^f_{v,u} | \forall f, e_{v,u} \in E\}$ as a traffic distribution, which represents each flow's load on each link. Given a T, we call D *feasible* if it satisfies:

V	The set of all switches.
E	The set of all links between switches.
G	The directed network graph $G = (V, E)$.
$e_{v,u}$	A directed link from switch v to u.
$c_{v,u}$	The link capacity of $e_{v,u}$.
f	A flow from an ingress to an egress switch.
s_f	The ingress switch of f.
d_f	The egress switch of f.
p_f	A path taken by f from s_f to d_f.
G_f	The subgraph formed by all the p_f's.
T	The traffic matrix of the network.
T_f	The flow size of f in T.
$l^f_{v,u}$	The traffic load placed on $e_{v,u}$ by f.
D	A traffic distribution $D := \{l^f_{v,u} \vert \forall f, e_{v,u} \in E\}$.
$r^f_{v,u}$	A rule for f on $e_{v,u}$: $r^f_{v,u} = l^f_{v,u}/T_f$.
R	All rules in network: $R := \{r^f_{v,u} \vert \forall f, e_{v,u} \in E\}$.
R^f_v	Rules for f on switch v: $\{r^f_{v,u} \vert \forall u : e_{v,u} \in E\}$.
R^f	Rules for f in the network: $\{r^f_{v,u} \vert \forall e_{v,u} \in E\}$.
$\mathcal{D}(T)$	All *feasible* traffic distributions which fully deliver T.
$\mathcal{D}_{\mathcal{C}}(T)$	All traffic distributions in $\mathcal{D}(T)$ which satisfy constraints $\mathcal{C}$.
$\mathcal{P}(T)$	$\mathcal{P}(T) := \{(D_1, D_2) \vert \forall D_1, D_2 \in \mathcal{D}(T)$, direct transition D_1 to D_2 is lossless.$\}$

Table 3.2: The key notations of the network model.

$$\forall f : \sum_{u \in V} l^f_{s_f,u} = \sum_{v \in V} l^f_{v,d_f} = T_f \tag{3.1}$$

$$\forall f, v \in V \setminus \{s_f, d_f\} : \sum_{u \in V} l^f_{v,u} = \sum_{u \in V} l^f_{u,v} \tag{3.2}$$

$$\forall f, e_{v,u} \notin G_f : l^f_{v,u} = 0 \tag{3.3}$$

$$\forall e_{v,u} \in E : \sum_{\forall f} l^f_{v,u} \leq c_{v,u} \tag{3.4}$$

Equations (3.1) and (3.2) guarantee that all the traffic is fully delivered, (3.3) means a link should not carry f's traffic if it is not on the paths from s_f to d_f, and (3.4) means no link is congested. We denote $\mathcal{D}(T)$ as the set of all *feasible* traffic distributions under T.

Flow rule: We define a *rule* for a flow f on link $e_{v,u}$ as $r^f_{v,u} = l^f_{v,u}/T_f$, which is essentially

a normalized value of $l_{v,u}$ by the flow size T_f. We also define the set of rules in the whole network as $R = \{r^f_{v,u} | \forall f, e_{v,u} \in E\}$, the set of rules of a flow f as $R^f = \{r^f_{v,u} | \forall e_{v,u} \in E\}$, and the set of rules for a flow f on a switch v as $R^f_v = \{r^f_{v,u} | \forall u : e_{v,u} \in E\}$.

Given T, we can compute the rule set R for the traffic distribution D and vice versa. We use $D = T \times R$ to denote the correspondence between D and R. We call a R *feasible* if its corresponding D is *feasible*. In practice, we will install R into the switches to realize the corresponding D under T because R is independent from the flow sizes and can be directly implemented with the existing switch functions. We will discuss the implementation details in §3.8.

3.5.2 Lossless transition between traffic distributions

To transition from D^1 to D^2 under given T, we need to change the corresponding rules from R^1 to R^2 on all the switches. A basic requirement for a lossless transition is that both D^1 and D^2 are feasible: $D^1 \in \mathcal{D}(T) \wedge D^2 \in \mathcal{D}(T)$. However, this requirement is insufficient due to asynchronous switch changes as shown in §3.4.

We explain this problem in more detail using an example in Figure 3.5, which is the subgraph G_f of f from ToR_1 to ToR_8 in a small Clos network. Each of switches 1-5 has two next hops towards ToR_8. Thus $l^f_{7,8}$ depends on f's rules on switches 1-5. When the switches are changed asynchronously, each of them could be using either old or new rules, resulting in 2^5 potential values of $l^f_{7,8}$.

Generally, the number of potential values of $l^f_{v,u}$ grows exponentially with the number of switches which may influence $l^f_{v,u}$. To guarantee a lossless transition under an arbitrary switch change order, Equation (3.4) must hold for any potential value of $l^f_{v,u}$, which is computationally infeasible to check in a large network.

To solve the state explosion problem, we leverage the *two-phase commit* mechanism [69] to change the rules of each flow. In the first phase, the new rules of f ($R^{f,2}$) are added to all

the switches while f's packets tagged with an old version number are still processed with the old rules ($R^{f,1}$). In the second phase, s_f tags f's packets with a new version number, causing all the switches to process the packets with the new version number using $R^{f,2}$.

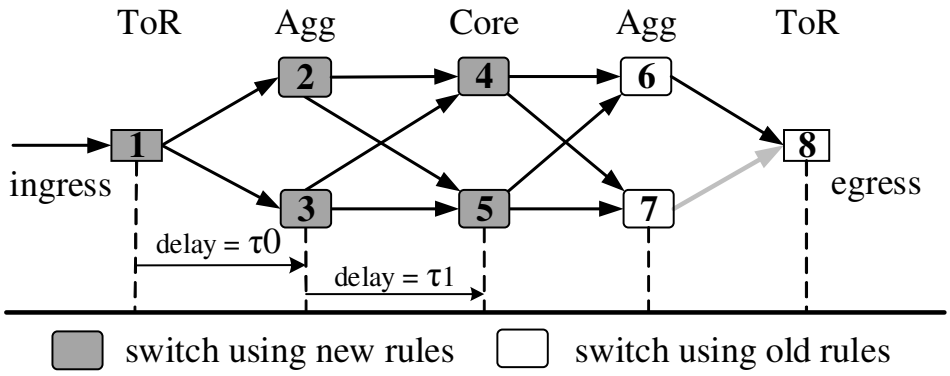


Figure 3.5: Two-phase commit simplifies link load calculations.

To see how two-phase commit helps solve the state explosion problem, we observe that the subgraph G_f of a flow f (Figure 3.5) has multiple layers and the propagation delay between two adjacent layers is almost a constant. When there is no congestion, the queuing and processing delays on switches are negligibly small. Suppose switch$_1$ flips to the new rules at time 0, switch$_4$ will receive the packets with the new version number on both of its incoming interfaces at $\tau_0 + \tau_1$ and flip to the new rules at the same time. It will never receive a mix of packets with two different version numbers. Moreover, all the switches in the same layer will flip to the new rules simultaneously. This is illustrated in Figure 3.5 where switch$_4$ and switch$_5$ (in shaded boxes) just flipped to the new rules while switch$_6$ and switch$_7$ (in unshaded boxes) are still using the old rules. Formally, we can prove

Lemma 2 *Suppose a network uses two-phase commit to transition the traffic distribution of a flow f from D^1 to D^2. If G_f satisfies the following three conditions:*

i) Layered structure: *All switches in G_f can be partitioned into sets $L_0, \ldots, L_m$, where $L_0 = \{s_f\}$, $L_m = \{d_f\}$ and $\forall e_{v,u} \in G_f$, if $v \in L_k$, then $u \in L_{k+1}$.*

ii) Constant delay between adjacent layers: *$\forall e_{v,u} \in G_f$, let $s_{v,u}$ and $r_{v,u}$ be the sending rate of v and the receiving rate of u, $\delta_{v,u}$ be the delay from the time when $s_{v,u}$ changes to the time when $r_{v,u}$ changes. Suppose $\forall v_1, v_2 \in V_k, e_{v_1,u_1}, e_{v_2,u_2} \in G_f$: $\delta_{v_1,u_1} = \delta_{v_2,u_2} = \delta_k$.*

iii) No switch queuing or processing delay: *Given a switch u and $\forall e_{v,u}, e_{u,w} \in G_f$: if $r_{v,u}$ changes from $l^1_{v,u}$ to $l^2_{v,u}$ simultaneously, then $s_{u,w}$ changes from $l^1_{u,w}$ to $l^2_{u,w}$ immediately at the same time.*

then we have $\forall e_{v,u} \in G_f$, $s_{v,u}$ and $r_{v,u}$ are either $l^1_{v,u}$ or $l^2_{v,u}$ during the transition.

Proof. Assume at time $t = \tau_k$, flow f's traffic distribution is:

- $\forall v \in L_i (i < k), e_{v,u} \in G_f$: $s_{v,u} = r_{v,u} = l^2_{v,u}$.
- $\forall v \in L_i (i > k), e_{v,u} \in G_f$: $s_{v,u} = r_{v,u} = l^1_{v,u}$.
- $\forall v \in L_k, e_{v,u} \in G_f$: all the $s_{v,u}$'s are changing from $l^1_{v,u}$ to $l^2_{v,u}$ simultaneously, but all the $r_{v,u}$'s are $l^1_{v,u}$.

Consider $\forall u \in L_{k+1}, e_{v,u}, e_{u,w} \in G_f$: According to Condition **ii)**, in the duration $\tau_k \leq t < \tau_{k+1} = \tau_k + \delta_k$, all the $r_{v,u}$'s remain $l^1_{v,u}$. Therefore, f's traffic distribution is:

- $\forall v \in L_i (i < k), e_{v,u} \in G_f$: $s_{v,u} = r_{v,u} = l^2_{v,u}$, because nothing has changed on these links.
- $\forall v \in L_i (i > k), e_{v,u} \in G_f$: $s_{v,u} = r_{v,u} = l^1_{v,u}$, because nothing has changed on these links.
- $\forall v \in L_k, e_{v,u} \in G_f$: $s_{v,u} = l^2_{v,u}$ and $r_{v,u} = l^1_{v,u}$, since the rate change on the sending end has not reached the receiving end due to the link delay.

At $t = \tau_{k+1}$, $\forall u \in L_{k+1}$, all the $r_{v,u}$'s change from $l^1_{v,u}$ to $l^2_{v,u}$ simultaneously. According to Condition **iii)**, all the $s_{u,w}$'s also change from $l^1_{u,w}$ to $l^2_{u,w}$ at the same time. Thus, at $t = \tau_{k+1}$:

- $\forall v \in L_i (i < k+1), e_{v,u} \in G_f$: $s_{v,u} = r_{v,u} = l^2_{v,u}$
- $\forall v \in L_i (i > k+1), e_{v,u} \in G_f$: $s_{v,u} = r_{v,u} = l^1_{v,u}$
- $\forall v \in L_{k+1}, e_{v,u} \in G_f$: all the $s_{v,u}$'s are changing from $l^1_{v,u}$ to $l^2_{v,u}$ simultaneously, but all the $r_{v,u}$'s are $l^1_{v,u}$.

At the beginning of the transition $t = \tau_0$, s_f, the only switch in L_0, starts to tag f's packets with a new version number, causing all of its outgoing links to change from the old sending rates to the new sending rates simultaneously. Hence, we have:

- $\forall v \in L_i (i > 0), e_{v,u} \in G_f$: $s_{v,u} = r_{v,u} = l^1_{v,u}$.
- $\forall v \in L_0, e_{v,u} \in G_f$: all the $s_{v,u}$'s are changing from $l^1_{v,u}$ to $l^2_{v,u}$ simultaneously, but all the $r_{v,u}$'s are $l^1_{v,u}$.

which matches our preceding assumption at $t = \tau_k$. Since we have derived that if the assumption holds for $t = \tau_k$, it also holds for $t = \tau_{k+1}$. Hence it holds for the whole transition process. Because in each duration $[\tau_k, \tau_{k+1})$, $\forall e_{v,u} \in G_f$, $s_{v,u}$ and $r_{v,u}$ are either $l^1_{v,u}$ or $l^2_{v,u}$, proof completes.

■

Two-phase commit reduces the number of potential values of $l^f_{v,u}$ to just two, but it does not completely solve the problem. In fact, when each f is changed asynchronously via two-phase commit, the number of potential values of $\sum_{\forall f} l^f_{v,u}$ in Equation (3.4) will be 2^n where n is the number of flows. To further reduce the complexity in checking (3.4), we introduce the following:

Lemma 3 *When each flow is changed independently, a transition from D^1 to D^2 is lossless if and only if:*

$$\forall e_{v,u} \in E : \sum_{\forall f} max\ \{l_{v,u}^{f,1}, l_{v,u}^{f,2}\} \leq c_{v,u} \tag{3.5}$$

Proof. At any snapshot during the transition, $\forall e_{v,u} \in E$, let $\mathcal{F}_{v,u}^1$/$\mathcal{F}_{v,u}^2$ be the set of flows with the old/new load values. Due to two-phase commit, $\mathcal{F}_{v,u}^1 \cup \mathcal{F}_{v,u}^2$ contains all the flows on $e_{v,u}$.

$\Rightarrow$: Construct $\mathcal{F}_{v,u}^1$ and $\mathcal{F}_{v,u}^2$ as follows: f is put into $\mathcal{F}_{v,u}^1$ if $l_{v,u}^{f,1} \geq l_{v,u}^{f,2}$, otherwise it is put into $\mathcal{F}_{v,u}^2$. Because the transition is congestion-free, we have:

$$\sum_{f \in \mathcal{F}_{v,u}^1} l_{v,u}^{f,1} + \sum_{f \in \mathcal{F}_{v,u}^2} l_{v,u}^{f,2} = \sum_{\forall f} \text{max } \{l_{v,u}^{f,1}, l_{v,u}^{f,2}\} \leq c_{v,u}$$

Hence, (3.5) holds.

$\Leftarrow$: When (3.5) holds, we have:

$$\sum_{f \in \mathcal{F}_{v,u}^1} l_{v,u}^{f,1} + \sum_{f \in \mathcal{F}_{v,u}^2} l_{v,u}^{f,2} \leq \sum_{\forall f} \text{max } \{l_{v,u}^{f,1}, l_{v,u}^{f,2}\} \leq c_{v,u}$$

Thus no link is congested at any snapshot during the transition. ■

Lemma 3 means we only need to check Equation (3.5) to ensure a lossless transition, which is now computationally feasible. Note that when the flow changes are dependent, *e.g.*, the flows on the same ingress switch are tagged with a new version number simultaneously, (3.5) will be a sufficient condition. We define $\mathcal{P}(T)$ as the set of all pairs of feasible traffic distributions (D^1, D^2) which satisfy (3.5) under traffic matrix T.

3.5.3 Computing transition plan

Given T_0 and D_0, `zUpdate` tries to find a feasible D_n which satisfies constraints $\mathcal{C}$ and can be transitioned from D_0 without loss. The search is done by constructing an optimization programming model $\mathcal{M}$.

In the simplest form, $\mathcal{M}$ is comprised of D_n as the variable and two constraints: (i) $D_n \in \mathcal{D}_{\mathcal{C}}(T_0)$; (ii) $(D_0, D_n) \in \mathcal{P}(T_0)$. Note that (i) & (ii) can be represented with equations (3.1)$\sim$(3.5). We defer the discussion of constraints $\mathcal{C}$ in §3.6. If such a D_n is found, the problem is solved (by the definitions of $\mathcal{D}_{\mathcal{C}}(T_0)$ and $\mathcal{P}(T_0)$ in Table 4.1) and a lossless transition can be performed in one step.

However, sometimes we cannot find a D_n which satisfies the two constraints above. When this happens, our key idea is to introduce a sequence of intermediate traffic distributions $(D_1, \ldots, D_{n-1})$ to bridge the transition from D_0 to D_n via n steps. Specifically, `zUpdate` will attempt to find $D_k(k = 1, \ldots, n)$ which satisfy: (I) $D_n \in \mathcal{D}_{\mathcal{C}}(T_0)$; (II) $(D_{k-1}, D_k) \in \mathcal{P}(T_0)$. If such a sequence is found, it means a lossless transition from D_0 to D_n can be performed in n steps. In this general form of $\mathcal{M}$, $D_k(k = 1, \ldots, n)$ are the variables and (I) & (II) are the constraints.

Algorithm 3 shows the pseudocode of `zUpdate`(T_0, D_0, $\mathcal{C}$). Since we do not know how many steps are needed in advance, we will search from $n = 1$ and increment n by 1 until a solution is found or n reaches a predefined limit N. In essence, we aim to minimize the number of transition steps to save the overall transition time. Note that there may exist many solutions to $\mathcal{M}$, we will show how to pick a proper objective function to reduce the transition overhead in §3.7.4.

Algorithm 3: `zUpdate`(T_0, D_0, $\mathcal{C}$)

// D_0 is the initial traffic distribution
// If D_0 satisfies the constraints $\mathcal{C}$, return D_0 directly
if $D_0 \in \mathcal{D}_{\mathcal{C}}(T_0)$ **then**
    return $[D_0]$;
// The initial # of steps is 1, N is the max # of steps
$n \leftarrow 1$;
while $n \leq N$ **do**
    $\mathcal{M} \leftarrow$ new optimization model ;
    $D[0] \leftarrow D_0$;
    for $k \leftarrow 1, 2, \ldots, n$ **do**
        $D[k] \leftarrow$ new traffic distribution variable ;
        $\mathcal{M}$.addVariable($D[k]$) ;
        // $D[k]$ should be feasible under T_0 $\mathcal{M}$.addConstraint($D[k] \in \mathcal{D}(T_0)$) ;
    for $k \leftarrow 1, 2, \ldots, n$ **do**
        // Transition $D[k-1] \rightarrow D[k]$ is lossless ;
        $\mathcal{M}$.addConstraint($(D[k-1], D[k]) \in \mathcal{P}(T_0)$) ;
    // $D[n]$ should satisfy the constraints $\mathcal{C}$ $\mathcal{M}$.addConstraint($D[n] \in \mathcal{D}_{\mathcal{C}}(T_0)$) ;
    // An objective is optional
    $\mathcal{M}$.addObjective(objective) ;
    if $\mathcal{M}$.*solve()* = *Successful* **then**
        return $D[1 \rightarrow n]$;
    $n \leftarrow n + 1$;
return [] // no solution is found ;

3.6 Handling Update Scenarios

In this section, we apply `zUpdate` to various update scenarios listed in Table 3.1. Specifically, we will explain how to formulate the requirements of each scenario as `zUpdate`'s input constraints $\mathcal{C}$.

3.6.1 Network topology updates

Certain update scenarios, *e.g.*, switch firmware upgrade, switch failure repair, and new switch on-boarding, involve network topology changes but no traffic matrix change. We

may use `zUpdate` to transition from the initial traffic distribution D_0 to a new traffic distribution D^* which satisfies the following requirements.

Switch firmware upgrade & switch failure repair: Before the operators shutdown or reboot switches for firmware upgrade or failure repair, they want to move all the traffic away from those switches to avoid disrupting the applications. Let U be the set of candidate switches. The preceding requirement can be represented as the following constraints $\mathcal{C}$ on the traffic distribution D^*:

$$\forall f, u \in U, e_{v,u} \in E : l_{v,u}^{f,*} = 0 \tag{3.6}$$

which forces all the neighbor switches to stop forwarding traffic to switch u before the update.

New device on-boarding: Before the operators add a new switch to the network, they want to test the functionality and performance of the new switch with some non-critical production traffic. Let u_0 be the new switch, $\mathcal{F}_{test}$ be the test flows, and $G_f(u_0)$ be the subgraph formed by all the p_f's which traverse u_0. The preceding requirement can be represented as the following constraints $\mathcal{C}$ on the traffic distribution D^*:

$$\forall f \in \mathcal{F}_{test}, e_{v,u} \notin G_f(u_0) : l_{v,u}^{f,*} = 0 \tag{3.7}$$

$$\forall f \notin \mathcal{F}_{test}, e_{v,u_0} \in E : l_{v,u_0}^{f,*} = 0 \tag{3.8}$$

where (3.7) forces all the test flows to only use the paths through u_0, while (3.8) forces all the non-test flows not to traverse u_0.

Restoring ECMP: A DCN often uses ECMP in normal condition, but WCMP during

updates. After an upgrade or testing is completed, operators may want to restore ECMP in the network. This can simply be represented as the following constraints $\mathcal{C}$ on D^*:

$$\forall f, v \in V, e_{v,u_1}, e_{v,u_2} \in G_f : l^{f,*}_{v,u_1} = l^{f,*}_{v,u_2} \tag{3.9}$$

which forces switches to evenly split f's traffic among next hops.

3.6.2 Traffic matrix updates

Certain update scenarios, *e.g.*, VM migration and LB reconfiguration, will trigger traffic matrix changes. Let T_0 and T_1 be the initial and final traffic matrices, we may use `zUpdate` to transition from the initial traffic distribution D_0 to a new D^* whose corresponding rule set R^* is feasible under T_0, T_1, and any possible transient traffic matrices during the update.

As explained in §3.4, the number of possible transient traffic matrices can be enormous when many LBs (or VMs) are being updated. It is thus computationally infeasible even to enumerate all of them. Our key idea to solve this problem is to introduce a *maximum traffic matrix* T_{max} that is "larger" than T_0, T_1 and any possible transient traffic matrices and only search for a D^* whose corresponding R^* is feasible under T_{max}.

Suppose during the update process, the real traffic matrix $T(t)$ is a function of time t. We define $\forall f : T_{max,f} := \sup(T_f(t))$ where sup means the upper bound over time t. We derive the following:

Lemma 4 *Given a rule set R^*, if $T_{max} \times R^* \in \mathcal{D}(T_{max})$, we have $T(t) \times R^* \in \mathcal{D}(T(t))$.*

Proof. Because $T_f(t) \leq T_{f,max}$ and $T_{max} \times R^* \in \mathcal{D}(T_{max})$, $\forall e_{v,u} \in E$, we have:

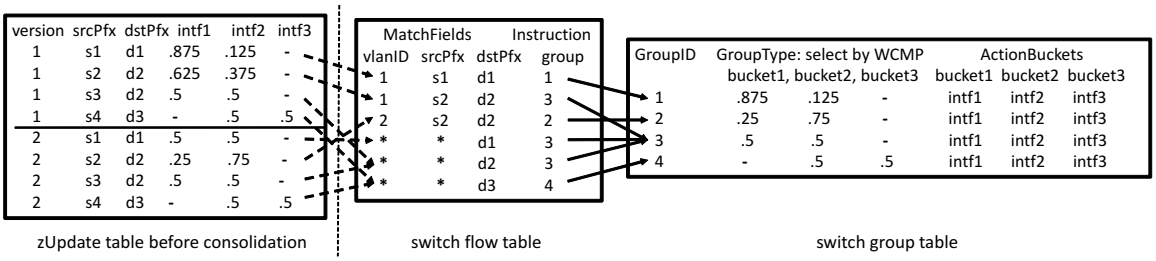


Figure 3.6: Implementing `zUpdate` on an OpenFlow switch.

$$\sum_{\forall f} T_f(t) \times r_{v,u}^{f,*} \leq \sum_{\forall f} T_{f,max} \times r_{v,u}^{f,*} \leq c_{v,u}$$

Hence, $T(t) \times R^* \in \mathcal{D}(T(t))$. ■

Lemma 4 says if R^* is feasible under T_{max}, it is feasible throughout the update process. This means, before updating the traffic matrix from T_0 to T_1, we may use `zUpdate` to transition from D_0 into D^* whose corresponding R^* is feasible under T_{max}. This leads to the following constraints $\mathcal{C}$ on D^*:

$$D^* = T_0 \times R^* \tag{3.10}$$

$$T_{max} \times R^* \in \mathcal{D}(T_{max}) \tag{3.11}$$

Here T_{max} is specified by the applications owners who are going to perform the update. In essence, lemma 4 enables the operators to migrate multiple VMs in parallel, saving the overall migration time, while not incurring any congestion.

3.7 Practical Issues

In this section, we discuss several practical issues in implementing `zUpdate` including switch table size, computational complexity, transition overhead, and unplanned failures and traffic matrix variations.

3.7.1 Implementing zUpdate on switches

The output of `zUpdate` is a sequence of traffic distributions, each of which can be implemented by installing its corresponding flow table entries into the switches. Given a flow f's traffic distribution on a switch v: $\{l^f_{v,u}\}$, we compute a weight set W^f_v in which $w^f_{v,u} = l^f_{v,u} / \sum_{u_i} l^f_{v,u_i}$. In practice, a `zUpdate` flow f is the collection of all the 5-tuple flows from the same ingress to the same egress switches, since all these 5-tuple flows share the same set of paths. W^f_v can then be implemented on switch v by hashing each of f's 5-tuple flows into one next hop in f's next hop set using WCMP. As in [69], the version number used by two-phase commit can be encoded in the VLAN tag.

Figure 3.6 shows an example of how to map the `zUpdate` flows to the flow and group table entries on an OpenFlow switch. The `zUpdate` flow (ver_2, s_2, d_2) maps to the switch flow table entry ($vlan_2$, s_2, d_2) which further points to $group_2$ in the switch group table. $group_2$ implements this flow's weight set $\{0.25, 0.75, -\}$ using the `SELECT` group type and the WCMP hash function.

3.7.2 Limited flow and group table size

As described in §3.2, a commodity switch has limited flow table size, usually between 1K and 4K entries. However, a large DCN may have several hundreds ToR switches, elevating the number of `zUpdate` flows beyond 100K, far exceeding the flow table size. Making things even worse, because two-phase commit is used, a flow table may hold two versions

of the entry for each flow, potentially doubling the number of entries. Finally, the group table size on commodity switches also poses a challenge, since it is around 1K (sometimes smaller than the flow table size).

Our solution to this problem is motivated by one key observation: ECMP works reasonably well for most of the flows in a DCN. During transition, there usually exist only several *bottleneck* links on which congestion may arise. Such congestion can be avoided by adjusting the traffic distribution of a small number of *critical* flows. This allows us to significantly cut down the number of flow table entries by keeping most of the flows in ECMP.

Consolidating flow table entries: Let S be the flow table size and n be the number of ToR switches. In a flow table, we will always have one *wildcard* entry for the destination prefix of each ToR switch, resulting in n wildcard entries in the table. Any flow that matches a wildcard entry will simply use ECMP. Figure 3.6 shows an example where the switch flow table has three wildcard entries for destinations d_1, d_2 and d_3. Since the weight set of `zUpdate` flows (ver_1, s_4, d_3) and (ver_2, s_4, d_3) is $\{-, 0.5, 0.5\}$, they both map to one wildcard entry (*, *, d_3) and use ECMP.

Suppose we need to consolidate k `zUpdate` flows into the switch flow table. Excluding the wildcard entries, the flow table still has $S - n$ free entries (note that S is almost certainly larger than n). Therefore, we will select $S - n$ *critical* flows and install a *specific* entry for each of them while forcing the remaining non-critical flows to use the wildcard entries (ECMP). This is illustrated in Figure 3.6 where the `zUpdate` flows (ver_1, s_1, d_1) and (ver_1, s_3, d_2) map to specific and wildcard entries in the switch flow table respectively. To resolve matching ambiguity in the switch flow table, a specific entry, *e.g.*, ($vlan_1$, s_1, d_1), always has higher priority than a wildcard entry, *e.g.*, (*, *, d_1).

The remaining question is how to select the critical flows. Suppose D_v^f is the traffic distribution of a `zUpdate` flow f on switch v, we calculate the corresponding $\bar{D}_v^f$ which

is f's traffic distribution if it uses ECMP. We use $\delta_v^f = \sum_u |l_{v,u}^f - \overline{l}_{v,u}^f|$ to quantify the "penalty" we pay if f is forced to use ECMP. To minimize the penalty caused by the flow consolidation, we pick the top $S - n$ flows with the largest penalty as the critical flows. In Figure 3.6, there are 3 critical flows whose penalty is greater than 0 and 5 non-critical flows whose penalty is 0.

Because of two-phase commit, each `zUpdate` flow has two versions. We follow the preceding process to consolidate both versions of the flows into the switch flow table. As shown in Figure 3.6, `zUpdate` flows (ver_1, s_4, d_3) and (ver_2, s_4, d_3) share the same wildcard entry in the switch flow table. In contrast, `zUpdate` flows (ver_1, s_1, d_1) and (ver_2, s_1, d_1) map to one specific entry and one wildcard entry in the switch flow table separately.

On some switches, the group table size may not be large enough to hold the weight sets of all the critical flows. Let T be group table size and m be the number of ECMP group entries. Because a group table must at least hold all the ECMP entries, T is almost always greater than m. After excluding the ECMP entries, the group table still has $T - m$ free entries. If $S - n > T - m$, we follow the preceding process to select $T - m$ critical flows with the largest penalty and install a group entry for each of them while forcing the remaining non-critical flows to use ECMP.

After flow consolidation, the real traffic distribution $\tilde{D}$ may deviate from the ideal traffic distribution D computed by `zUpdate`. Thus, an ideal lossless transition from D_0 to D_n may not be feasible due to the table size limits. To keep the no loss guarantee, `zUpdate` will check the real loss of transitioning from $\tilde{D_0}$ to $\tilde{D_n}$ after flow consolidation and return an empty list if loss does occur.

3.7.3 Reducing computational complexity

In §3.5.3, we construct an optimization programming model $\mathcal{M}$ to compute a lossless transition plan. Let $|F|$, $|V|$, and $|E|$ be the number of flows, switches and links in the network and n be the number of transition steps. The total number of variables and constraints in $\mathcal{M}$ is $O(n|F||E|)$ and $O(n|F|(|V|+|E|))$. In a large DCN, it could take a long time to solve $\mathcal{M}$.

Given a network, $|V|$ and $|E|$ are fixed and n is usually very small, the key to shortening the computation time is to reduce $|F|$. Fortunately in DCNs, congestion usually occurs only on a small number of *bottleneck links* during traffic migration, and such congestion may be avoided by just manipulating the traffic distribution of the *bottleneck flows* that traverse those bottleneck links. Thus, our basic idea is to treat only the bottleneck flows as variables while fixing all the non-bottleneck flows as constants in $\mathcal{M}$. This effectively reduces $|F|$ to be the number of bottleneck flows, which is far smaller than the total number of flows, dramatically improving the scalability of `zUpdate`.

Generally, without solving the (potentially expensive) $\mathcal{M}$, it is difficult to precisely know the bottleneck links. To circumvent this problem, we use a simple heuristic called ECMP-Only (or ECMP-O) to roughly estimate the bottleneck links. In essence, ECMP-O mimics how operators perform traffic migration today by solely relying on ECMP.

For network topology update (§3.6.1), the final traffic distribution D^* must satisfy Equations (3.6)$\sim$(3.8), each of which is in the form of $l^{f,*}_{v,u} = 0$. To meet each constraint $l^{f,*}_{v,u} = 0$, we simply remove the corresponding u from f's next hop set on switch v. After that, we compute D^* by splitting each flow's traffic among its remaining next hops using ECMP. Finally, we identify the bottleneck links as: i) the congested links during the one-step transition from D_0 to D^* (violating Equation (3.5)); ii) the congested links under D^* after the transition is done (violating Equation (3.4)).

For traffic matrix update (§3.6.2), ECMP-O does not perform any traffic distribution transition, and thus congestion can arise only during traffic matrix changes. Let T_{max} be the maximum traffic matrix and R_{ecmp} be ECMP, we simply identify the bottleneck links as the congested links under $D_{max} = T_{max} \times R_{ecmp}$.

3.7.4 Transition overhead

To perform traffic distribution transitions, `zUpdate` needs to change the flow and group tables on switches. Besides guaranteeing a lossless transition, we would also like to minimize the number of table changes. Remember that under the optimization model $\mathcal{M}$, there may exist many possible solutions. We could favor solutions with low transition overhead by picking a proper objective function. As just discussed, the ECMP-related entries (*e.g.*, wildcard entries) will remain static in the flow and group tables. In contrast, the non-ECMP entries (*e.g.*, specific entries) are more dynamic since they are directly influenced by the transition plan computed by `zUpdate`. Hence, a simple way to reduce the number of table changes is to "nudge" more flows towards ECMP. This prompts us to minimize the following objective function in $\mathcal{M}$:

$$\sum_{i=1}^{n} \sum_{f,v,u,w} |l_{v,u}^{f,i} - l_{v,w}^{f,i}|, \text{where } e_{v,u}, e_{v,w} \in E \tag{3.12}$$

in which n is the number of transition steps. Clearly, the objective value is 0 when all the flows use ECMP. One nice property of Equation(4.1) is its linearity. In fact, because Equations(3.1) $\sim$ (4.1) are all linear, $\mathcal{M}$ becomes a linear programming (LP) problem which can be solved efficiently.

3.7.5 Failures and traffic matrix variations

It is trivial for `zUpdate` to handle unplanned failures during transitions. In fact, failures can be treated in the same way as switch upgrades (see §3.6.1) by adding the failed switches/links to the update requirements, *e.g.*, those failed switches/links should not carry any traffic in the future. `zUpdate` will then attempt to re-compute a transition plan from the current traffic matrix and traffic distribution to meet the new update requirements.

Handling traffic matrix variations is also quite simple. When estimating T_{max}, we may multiply it by an error margin η ($\eta > 1$). Lemma 4 guarantees that the transitions are lossless so long as the real traffic matrix $T \leq \eta T_{max}$.

3.8 Implementation

Figure 3.7 shows the key components and workflow of `zUpdate`. When an operator wants to perform a DCN update, she will submit a request containing the update requirements to the *update scenario translator*. The latter converts the operator's request into the formal update constraints (§3.6). The `zUpdate` *engine* takes the update constraints together with the current network topology, traffic matrix, and flow rules and attempts to produce a lossless transition plan (§3.5, 3.7.3 & 3.7.4). If such a plan cannot be found, it will notify the operator who may decide to revise or postpone the update. Otherwise, the *transition plan translator* will convert the transition plan into the corresponding flow rules (§3.7.1 & 3.7.2). Finally, the OpenFlow controller will push the flow rules into the switches.

The `zUpdate` engine and the update scenario translator consists of 3000+ lines of C# code with Mosek [5] as the linear programming solver. The transition plan translator is written in 1500+ lines of Python code. We use Floodlight 0.9 [2] as the OpenFlow controller and commodity switches which support OpenFlow 1.0 [6]. Given that WCMP

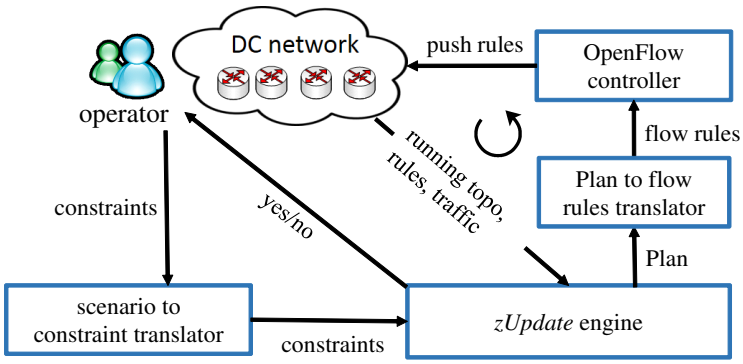


Figure 3.7: `zUpdate`'s prototype implementation.

is not available in OpenFlow 1.0, we emulate WCMP as follows: given the weight set of a `zUpdate` flow f at switch v, for each constituent 5-tuple flow ξ in f, we first compute the next hop u of ξ according to WCMP hashing and then insert a rule for ξ with u as the next hop into v.

3.9 Evaluations

In this section, we show `zUpdate` can effectively perform cong-estion-free traffic migration using both testbed experiments and large-scale simulations. Compared to alternative traffic migration approaches, `zUpdate` can not only prevent loss but also reduce the transition time and transition overhead.

3.9.1 Experimental methodology

Testbed experiments: Our testbed experiments run on a FatTree network with 4 CORE switches and 3 containers as illustrated in Figure 3.3. (Note that there are 2 additional ToRs connected to $AGG_{3,4}$ which are not shown in the figure because they do not send or receive any traffic). All the switches support OpenFlow 1.0 with 10Gbps link speed. A commercial traffic generator is connected to all the ToRs and CORE's to inject 5-tuple

flows at pre-configured constant bit rate.

Large-scale simulations: Our simulations are based on a production DCN with hundreds of switches and tens of thousands of servers. The flow and group table sizes are 750 and 1,000 entries respectively, matching the numbers of the commodity switches used in the DCN. To obtain the traffic matrices, we log the socket events on all the servers and aggregate the logs into ingress-to-egress flows over 10-minute intervals. A traffic matrix is comprised of all the ingress-to-egress flows in one interval. From the 144 traffic matrices in a typical working day, we pick 3 traffic matrices that correspond to the minimum, median and maximum network-wide traffic loads respectively. The simulations run on a commodity server with 1 quad-core Intel Xeon 2.13GHz CPU and 48GB RAM.

Alternative approaches: We compare `zUpdate` with three alternative approaches: (1) *`zUpdate`-One-Step (`zUpdate`-O)*: It uses `zUpdate` to compute the final traffic distribution and then jumps from the initial traffic distribution to the final one directly, omitting all the intermediate steps. (2) *ECMP-O* (defined in §3.7.3). (3) *ECMP-Planned (ECMP-P)*: For traffic matrix update, ECMP-P does not perform any traffic distribution transition (like ECMP-O). For network topology update, ECMP-P has the same final traffic distribution as ECMP-O. Their only difference is, when there are k ingress-to-egress flows to be migrated from the initial traffic distribution to the final traffic distribution, ECMP-O migrates all the k flows in one step while ECMP-P migrates only one flow in each step, resulting in $k!$ candidate migration sequences. In our simulations, ECMP-P will evaluate 1,000 randomly-chosen candidate migration sequences and use the one with the minimum losses. In essence, ECMP-P mimics how today's operators sequentially migrate multiple flows in DCN.

Performance metrics: We use the following metrics to compare different approaches. (1) *Link utilization*: the ratio between the link load and the link capacity. For the ease of presentation, we represent link congestion as link utilization value higher than 100%.

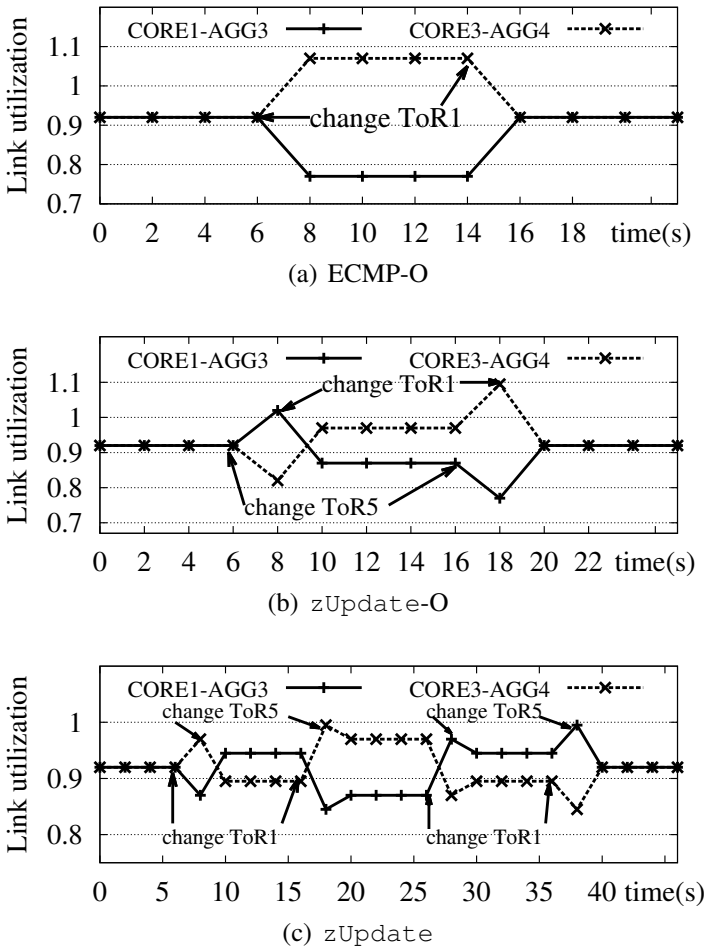


Figure 3.8: The link utilization of the two busiest links in the switch upgrade example.

(2) *Post-transition loss (Post-TrLoss)*: the maximum link loss rate after reaching the final traffic distribution. (3) *Transition loss (TrLoss)*: the maximum link loss rate under all the possible ingress-to-egress flow migration sequences during traffic distribution transitions. (4) *Number of steps*: the whole traffic migration process can be divided into multiple steps. The flow migrations within the same step are done in parallel while the flow migrations of the next step cannot start until the flow migrations of the current step complete. This metric reflects how long the traffic migration process will take. (5) *Switch touch times (STT)*: the total number of times the switches are reconfigured during a traffic migration. This metrics reflects the transition overhead.

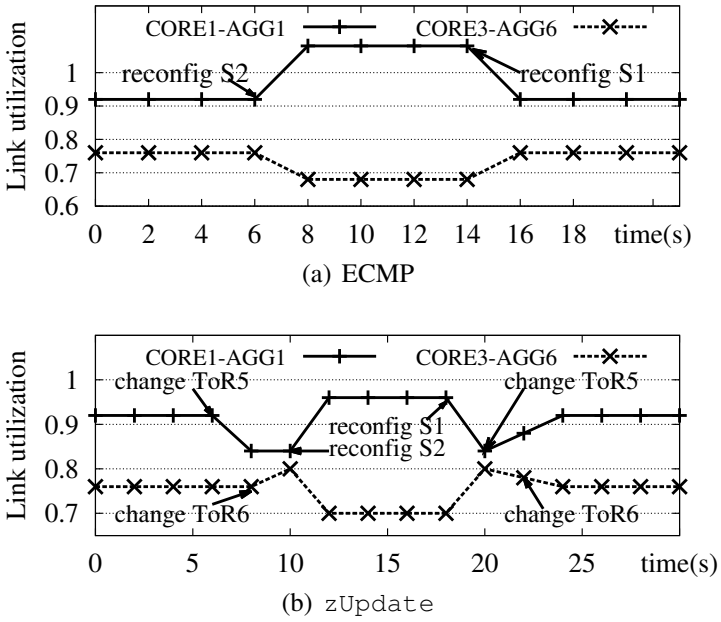


Figure 3.9: The link utilization of the two busiest links in LB reconfiguration example.

Calculating T_{max}: T_{max} includes two components: T_b and T_{app}^{max}. T_b is the background traffic which is independent from the application being updated. T_{app}^{max} is the maximum traffic matrix comprised of only the ingress-to-egress flows (f_{app}'s) related to the applications being updated. We calculate T_{app}^{max} as follows: for each f_{app}, the size of f_{app} in in T_{app}^{max} is the largest size that f_{app} can possibly get during the entire traffic matrix update process.

3.9.2 Testbed experiments

We now conduct testbed experiments to reproduce the two traffic migration examples described in §3.4.

Switch upgrade: Figure 3.8 shows the real-time utilization of the two busiest links, $\text{CORE}_1 \rightarrow \text{AGG}_3$ and $\text{CORE}_3 \rightarrow \text{AGG}_4$, in the switch upgrade example (Figure 3.2). Figure 3.8(a) shows the transition process from Figure 3.2a (0s $\sim$ 6s) to Figure 3.2b (6s $\sim$ 14s) under ECMP-O. The two links initially carry the same amount of traffic. At 6s,

$\text{ToR}_1 \rightarrow \text{AGG}_1$ is deactivated, triggering traffic loss on $\text{CORE}_3 \rightarrow \text{AGG}_4$ (Figure 3.2b). The congestion lasts until 14s when $\text{ToR}_1 \rightarrow \text{AGG}_1$ is restored. Note that we deliberately shorten the switch upgrade period for illustration purpose. In addition, because only one ingress-to-egress flow ($\text{ToR}_1 \rightarrow \text{ToR}_2$) needs to be migrated, ECMP-P is the same as ECMP-O.

Figure 3.8(b) shows the transition process from Figure 3.2a (0s $\sim$ 6s) to Figure 3.2d (6s $\sim$ 8s) to Figure 3.2c (8s $\sim$ 16s) under `zUpdate`-O. At 6s $\sim$ 8s, ToR_5 and ToR_1 are changed asynchronously, leading to a transient congestion on $\text{CORE}_1 \rightarrow \text{AGG}_3$ (Figure 3.2d). After $\text{ToR}_{1,5}$ are changed, the upgrading of AGG_1 is congestion-free at 8s $\sim$ 16s (Figure 3.2c). Once the upgrading of AGG_1 completes at 16s, the network is restored back to ECMP. Again because of the asynchronous changes to ToR_5 and ToR_1, another transient congestion happens on $\text{CORE}_3 \rightarrow \text{AGG}_4$ at 16s $\sim$ 18s.

Figure 3.8(c) shows the transition process from Figure 3.2a (0s $\sim$ 6s) to Figure 3.2e (8s $\sim$ 16s) to Figure 3.2c (18s $\sim$ 26s) under `zUpdate`. Due to the introduction of an intermediate traffic distribution between 8s $\sim$ 16s (Figure 3.2e), the transition process is lossless despite of asynchronous switch changes at 6s $\sim$ 8s and 16s $\sim$ 18s.

LB reconfiguration: Figure 3.9 shows the real-time utilization of the two busiest links, $\text{CORE}_1 \rightarrow \text{AGG}_1$ and $\text{CORE}_3 \rightarrow \text{AGG}_6$ in the LB reconfiguration example (Figure 3.3). Figure 3.9(a) shows the migration process from Figure 3.3a (0s $\sim$ 6s) to Figure 3.3c (6s $\sim$ 14s) to Figure 3.3b (after 14s) under ECMP. At 6s $\sim$ 14s, S_2's LB and S_1's LB are reconfigured asynchronously, causing congestion on $\text{CORE}_1 \rightarrow \text{AGG}_1$ (Figure 3.3c). After both LB's are reconfigured at 14s, the network is congestion-free (Figure 3.3b).

Figure 3.9(b) shows the migration process from Figure 3.3a (0s $\sim$ 6s) to Figure 3.3d (10s $\sim$ 18s) to Figure 3.3b (after 22s) under `zUpdate`. By changing the traffic split ratio on ToR_5 and ToR_6 at 6s $\sim$ 8s, `zUpdate` ensures the network is congestion-free even though S_2's LB and S_1's LB are reconfigured asynchronously at 10s $\sim$ 18s. Once the LB

reconfiguration completes at 18s, the traffic split ratio on ToR_5 and ToR_6 is restored to ECMP at 20s $\sim$ 22s. Note that `zUpdate` is the same as `zUpdate`-O in this experiment, because there is no intermediate step in the traffic distribution transition.

3.9.3 Large-scale simulations

We run large-scale simulations to study how `zUpdate` enables lossless switch onboarding and VM migration in production DCN.

Switch onboarding: In this experiment, a new CORE switch is initially connected to each container but carries no traffic. We then randomly select 1% of the ingress-to-egress flows, as test flows, to traverse the new CORE switch for testing. Figure 3.10(a) compares different migration approaches under the median network-wide traffic load. The y-axis on the left is the traffic loss rate and the y-axis on the right is the number of steps. `zUpdate` attains zero loss by taking 2 transition steps. Although not shown in the figure, our flow consolidation heuristic (§3.7.2) successfully fits a large number of ingress-to-egress flows into the limited switch flow and group tables. `zUpdate`-O has no post-transition loss but 8% transition loss because it takes just one transition step.

ECMP-O incurs 7% transition loss and 13.5% post-transition loss. This is a bit counterintuitive because the overall network capacity actually increases with the new switch. We explain this phenomenon with a simple example in Figure 3.11. Suppose there are 7 ingress-to-egress flows to ToR_1, each of which is 2Gbps, and the link capacity is 10Gbps. Figure 3.11(a) shows the initial traffic distribution under ECMP where each downward link to ToR_1 carries 7Gbps traffic. In Figure 3.11(b), 4 out of the 7 flows are selected as the test flows and are moved to the new CORE_3. Thus, $\text{CORE}_3 \rightarrow \text{AGG}_2$ has 8Gbps traffic (the 4 test flows) and $\text{CORE}_2 \rightarrow \text{AGG}_2$ has 3Gbps traffic (half of the 3 non-test flows due to ECMP). This in turn overloads $\text{AGG}_2 \rightarrow \text{ToR}_1$ with 11Gbps traffic. Figure 3.11(c)

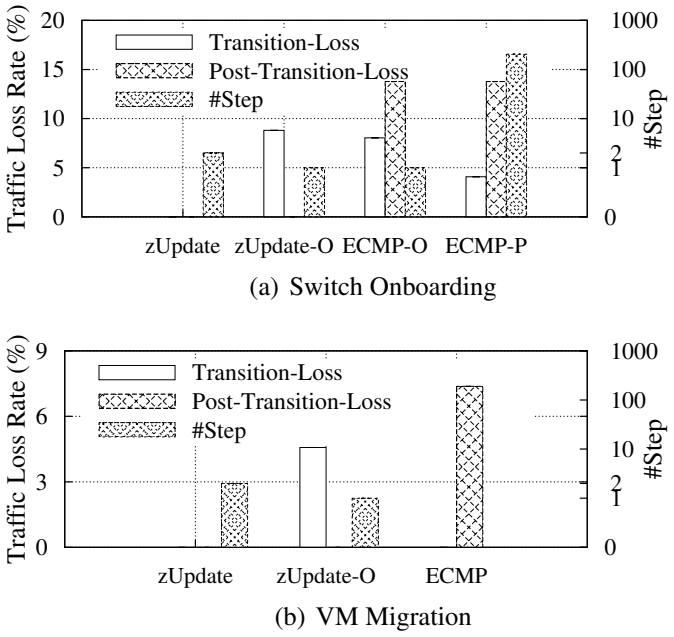


Figure 3.10: Comparison of different migration approaches.

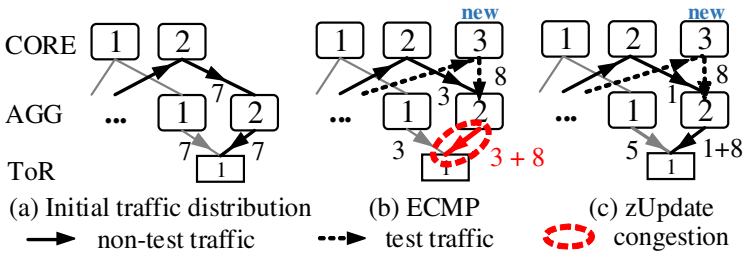


Figure 3.11: Why congestion occurs in switch onboarding.

shows `zUpdate` avoids the congestion by moving 2 non-test flows away from $\text{CORE}_2 \rightarrow \text{AGG}_2$ to $\text{AGG}_1 \rightarrow \text{ToR}_1$. This leaves only 1Gbps traffic (half of the remaining 1 non-test flow) on $\text{CORE}_2 \rightarrow \text{AGG}_2$ and reduces the load on $\text{AGG}_2 \rightarrow \text{ToR}_1$ to 9Gbps.

ECMP-P has smaller transition loss (4%) than ECMP-O because ECMP-P attempts to use a flow migration sequence that incurs the minimum loss. They have the same post-transition loss because their final traffic distribution is the same. Compared to `zUpdate`, ECMP-P has significantly higher loss although it takes hundreds of transition steps (which also implies much longer transition period).

VM migration: In this experiment, we migrate a group of VMs from one ToR to another ToR in two different containers. During the live migration, the old and new VMs establish tunnels to synchronize data and running states [28, 51]. The total traffic rate of the tunnels is 6Gbps.

Figure 3.10(b) compares different migration approaches under the median network-wide traffic load. `zUpdate` takes 2 steps to reach a traffic distribution that can accommodate the large volume of tunneling traffic and the varying traffic matrices during the live migration. Hence, it does not have any loss. In contrast, `zUpdate`-O has 4.5% transition loss because it skips the intermediate step taken by `zUpdate`. We combine ECMP-O and ECMP-P into ECMP because they are the same for traffic matrix updates (§3.9.1). ECMP's post-transition loss is large (7.4%) because it cannot handle the large volume of tunneling traffic during the live migration.

Impact of traffic load: We re-run the switch onboarding experiment under the minimum, median, and maximum network-wide traffic loads. In Figure 3.12, we omit the loss of of `zUpdate` and the post-transition loss of `zUpdate`-O, since all of them are 0.

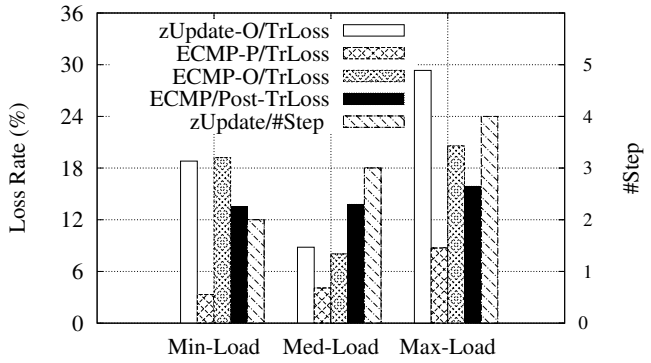


Figure 3.12: Comparison under different traffic loads.

We observe that only `zUpdate` can attain zero loss under different levels of traffic load. Surprisingly, the transition loss of `zUpdate`-O and ECMP-O is actually higher under the minimum load than under the median load. This is because the traffic loss is

determined to a large extent by a few *bottleneck* links. Hence, without careful planning, it is risky to perform network-wide traffic migration even during off-peak hours. Figure 3.12 also shows `zUpdate` takes more transition steps as the network-wide traffic load grows. This is because when the traffic load is higher, it is more difficult for `zUpdate` to find the spare bandwidth to accommodate the temporary link load increase during transitions.

Transition overhead: Table 3.3 shows the number of switch touch times (STT) of different migration approaches in the switch onboarding experiment. Compared to the STT of `zUpdate`-O and ECMP-O, the STT of `zUpdate` is doubled because it takes two steps instead of one. However, this also indicates `zUpdate` touches at most 68 switches which represent a small fraction of the several hundreds switches in the DCN. This can be attributed to the heuristics in §3.7.3 and §3.7.4 which restrict the number of flows to be migrated. ECMP-P has much larger STT than the other approaches because it takes a lot more transition steps.

	zUpdate	zUpdate-O	ECMP-O	ECMP-P
STT	68	34	34	410

Table 3.3: Comparison of transition overhead.

The computation time of `zUpdate` is reasonably small for performing traffic migration in large DCNs. In fact, the running time is below 1 minute for all the experiments except the maximum traffic load case in Figure 3.12, where it takes 2.5 minutes to compute a 4-step transition plan. This is because of the heuristic in §3.7.3 which ties the computation complexity to the number of bottleneck flows rather than the total number of flows, effectively reducing the number of variables by at least two orders of magnitude.

3.10 Related Work

Congestion during update: Several recent papers focus on preventing congestion during a specific type of update. Raza *et al.* [68] study the problem of how to schedule link weight changes during IGP migrations. Ghorbani *et al.* [39] attempt to find a congestion-free VM migration sequence. In contrast, our work provides one primitive for a variety of update scenarios. Another key difference is they do not consider the transient congestion caused by asynchronous traffic matrix or switch changes since they assume there is only one link weight change or one VM being migrated at a time.

Routing consistency: There is a rich body of work on preventing transient misbehaviors during routing protocol updates. Vanbever *et al.* [76] and Francois *et al.* [37] seek to guarantee no forwarding loop during IGP migrations and link metric reconfigurations. Consensus routing [45] is a policy routing protocol aiming at eliminating transient problems during BGP convergence times. The work above emphasizes on routing consistency rather than congestion.

Several tools have been created to statically check the correctness of network configurations. Feamster *et al.* [36] built a tool to detect errors in BGP configurations. Header Space Analysis (HSA) [50] and Anteater [62] can check a few useful network invariants, such as reachability and no loop, in the forwarding plane. Built on the earlier work, VeriFlow [53] and realtime HSA [49] have been developed to check network invariants on-the-fly.

3.11 Summary

We have introduced `zUpdate` for performing congestion-free traffic migration in DCNs given the presence of asynchronous switch and traffic matrix changes. The core of `zUpdate`

is an optimization programming model that enables lossless transitions from an initial traffic distribution to a final traffic distribution to meet the predefined update requirements. We have built a `zUpdate` prototype on top of OpenFlow switches and Floodlight controller and demonstrated its capability in handling a variety of representative DCN update scenarios using both testbed experiments and large-scale simulations. `zUpdate`, as in its current form, works only for hierarchical DCN topology such as FatTree and Clos. We plan to extend `zUpdate` to support a wider range of network topologies in the future.

Chapter 4

Optimizing Cost and Performance in Content Multihoming

Many large content publishers use multiple content distribution networks to deliver their content, and many industrial systems have become available to help a broader set of content publishers to benefit from using multiple distribution networks, which we refer to as content multihoming. In this chapter, we conduct the first systematic study on optimizing content multihoming, by introducing a series of novel algorithms to optimize both performance and cost for content multihoming. In particular, we design a novel, efficient algorithm to compute assignments of content objects to content distribution networks for content publishers, considering both cost and performance. We also design a novel, lightweight client adaptation algorithm executing at individual content viewers to achieve scalable, fine-grained, fast online adaptation to optimize the quality of experience (QoE) for individual viewers. We provide proof on the optimality of our optimization algorithms and conduct systematic, extensive evaluations using real charging data, content viewer demands, and performance data, to demonstrate the effectiveness of our algorithms. We show that our content multihoming algorithms reduce publishing cost by up to 40%. Our

client algorithm executing in browsers reduces viewer QoE degradation by 51%.

4.1 Introduction

Many content publishers on the Internet use multiple content distribution networks (CDNs) to distribute and cache their digital content. For example, both Netflix [11] and Hulu use CDNs including Level 3, Limelight, and Akamai to distribute their content. We refer to content publishing using multiple content distribution networks as *content multihoming*. In our recent survey, major content publishers such as Netflix, Hulu, Microsoft, Apple, Facebook, and MSNBC all use content multihoming.

Content publishers adopt content multihoming to aggregate the diversity of individual CDN providers on features, performance and commitment [26]. For example, one CDN may provide good coverage for locations 1 and 2, whereas another CDN provides good coverage for locations 2 and 3. To deliver content to viewers from all three locations, a content publisher may need to use both CDNs.

Given the wide usage and potential benefits of content multihoming, many commercial systems supporting content multihoming have recently been developed and deployed (*e.g.*, [9, 29, 30, 33, 57, 58, 66]), so that more content publishers can benefit from content multihoming. However, these commercial products either use ad hoc algorithms or do not provide details on their designs. There are no previous known studies on how to effectively utilize content multihoming.

In this chapter, we attempt to provide a framework and a set of novel algorithms to optimize the benefits of content multihoming. We ask a simple question: Given that content multihoming allows a content object to be delivered from multiple CDNs, which CDN(s) should a content publisher use to deliver each object to each content viewer requesting this object, so that the publisher optimizes its benefits from content multihoming? Since

modern content delivery infrastructures provide flexible request routing mechanisms (*e.g.*, DNS CNAME and HTTP Redirect from the server side, and client scripts from the client side), the key to effectively utilizing content multihoming is to address this question.

An answer to this simple question, however, is not immediately obvious. Consider the current common approach of choosing, for each content viewer, the best performing CDN among all candidate CDNs. This approach, despite its simplicity, has multiple issues. First, although the chosen CDN may provide the highest level of performance, for example, satisfying that 99% viewers do not see quality of experience (QoE) degradation, the cost of the chosen CDN can be much higher than another CDN with a slightly lower, but still high enough level of 95%. Second, as we shall see, there are often multiple CDNs with comparable and sufficient levels of performance at a given region, *e.g.*, in US. One common approach to break ties in such cases is to pick the CDN with the lowest cost. However, the costs of CDNs, in particular, pay-as-you-go CDNs such as Amazon CloudFront, are volume based and non-linear. The cost of one object assignment depends on the other assignments. Third, there are locations where even the best performing CDN falls short. For example, a content publisher may have a QoE target of 95%, but the best performing CDN at some location achieves only 90%.

In this chapter, we provide an answer to the above question from two perspectives: (1) an efficient optimization algorithm executing at content publisher server(s) to compute content distribution guidance, and (2) a simple algorithm executing at individual content viewers to follow the guidance with local adaptation. Either algorithm can be deployed alone, but together they provide the most benefits.

Specifically, the local viewer algorithm provides a capability for a content viewer to make efficient usage of multiple servers from multiple CDNs, with a preference ordering on the usage of CDN edge servers provided by the content publisher. Inspired by TCP AIMD and using a simple priority task assignment mechanism, the algorithm adapts the

usage of multiple CDNs, achieving a performance level that no single CDN/server can achieve alone.

The publisher server optimization algorithm, named *CMO*, computes CDN assignments considering many real factors including non-linear, multi-region CDN traffic charging, per-request charging, storage charging, content licensing restrictions, CDN feature availability, and CDN performance variations. The CMO algorithm is novel and highly efficient. It reduces the computational complexity from exponential when using simple enumeration to be independent of the number of content objects when considering traffic costs.

We implement both algorithms and conduct systematic, extensive evaluations using real charging data, content viewer demands, and CDN performance to demonstrate the effectiveness of our algorithms. We show that our content multihoming algorithm reduces publishing cost by up to 40%. Our implementation of the client algorithm running in real Firefox browsers reduces viewer QoE degradation by 51%.

4.2 Background and Notations

We start by introducing background and notations. Table 4.1 provides a reference for the list of notations.

There are three key types of entities being managed in content multihoming: (1) content objects; (2) viewers of contents; (3) distribution networks that cache contents from origin networks to serve content viewers.

Content object: A content publisher can have a large number of content objects such as video and image objects. Let N denote the total number of content objects. An object has many properties. In the context of content distribution, the performance requirement, the size, and the popularity of an object are its key properties [20]. Let s_i be the size of object

i. We introduce object popularity when we next introduce content viewers.

N	Number of content objects.
K	Number of CDNs.
A	Set of fine-grained location areas.
s_i	Size of object i.
n_i^a	Number of requests for object i from area a.
i^a	Object i being requested by viewers from location area a.
α_k^r	Set of location areas served by charging region r of CDN k.
t_k^r	Charging volume of CDN k at its charging region α_k^r.
$C_k^r(t_k^r)$	Charging function of CDN k for charging region α_k^r.
F_k	Set of location objects that CDN k can serve.
$p_{i,k}^a$	CDN k's performance for object i at location area a: fraction of times CDN k can deliver i^a with sufficient QoE.
$x_{i,k}^a$	CDN guidance: fraction of n_i^a requests directed to CDN k.
π_k	CDN assignment: set of location objects assigned to CDN k.
T_k	The maximum volume that can be assigned to CDN k.
b_k	The boundary of the capacity constraint of CDN k.

Table 4.1: Summary of key notations.

Content viewer (client): There can be a large number of content viewers requesting the content objects. These content viewers can be distributed across multiple geographical areas. The specific geographical areas depend on the particular requirements of a content publisher. For generality and conceptual clarity, let A be the set of all geographical areas, say all cities. Note that in this challenging general case the size of A can be large, on the order of thousands. Let $a \in A$ denote a specific location area.

As the popularity of an object among content viewers is location dependent [38], let n_i^a denote the number of times that object i will be requested during a time interval (say a month), from content viewers located at location area a.

We also use n_i^a to encode *licensing restrictions* that a content publisher often needs to enforce in practice. Specifically, if content viewers from a location area a should not receive a content object i, n_i^a should be 0.

Content distribution network (CDN): A key reason of content multihoming is to aggregate the *capability-geography expertise* of different CDNs, as different CDNs can have

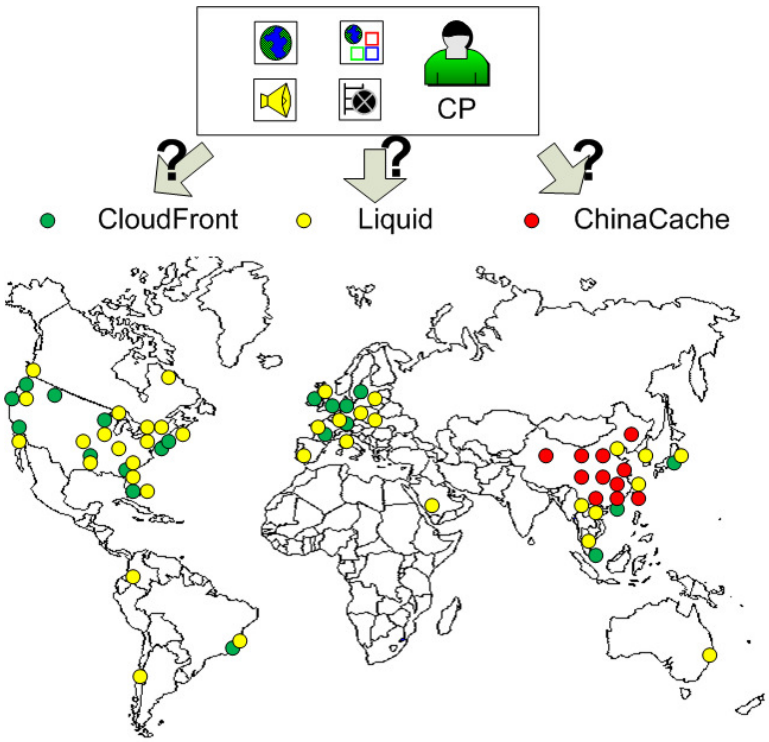


Figure 4.1: Edge server distributions of three CDNs.

quite different performance and cost characteristics, at different geographic regions. On the other hand, as we will see, such differences are a major source of intrinsic complexity when optimizing content multihoming. In this chapter, we assume that the set of CDNs is given. Let K be the number of CDNs, and we use k and j to index individual CDNs.

First consider *performance*. Figure 4.1 shows the edge server footprints of three real CDNs: Amazon CloudFront, MaxCDN and ChinaCache. When a content viewer from a location area a requests a content object through CDN k, a well designed CDN will choose an edge server (or several servers) that is close to a to serve the request, since a short latency from edge servers to end users is typically needed to achieve good content delivery performance [56]. Comparing the geographical footprints of the three CDNs shown in Figure 4.1, one can anticipate that CloudFront and MaxCDN are more likely to provide better performance in US and Europe, while ChinaCache may perform well in China. None of the three covers regions such as Russia and Africa.

To quantify the performance, we conduct measurements on the performance of three CDNs (CloudFront, MaxCDN, and Liquid Web) delivering to different locations. Since performance metrics are dependent on the content type, and streaming media is a major content type [27], we focus on the delivery of streaming media. Table 4.2 shows the success rates of the CDNs to deliver streaming objects encoded at three different streaming rates (1Mbps, 2Mbps, and 3Mbps) to some representative locations. For example, the entry for Liquid Web/Spain shows the success rates when viewers from Spain request from Liquid Web: if the object is encoded at 1 Mbps, 99.4% of the viewers can receive at the encoding rate; for a 2 Mbps object, only 47.3% of the viewers can receive at the encoding rate; for a 3 Mbps object, almost no viewers can receive at the encoding rate. The measurement results clearly show that the usability of a CDN depends on both the object (*e.g.*, a video encoded at 1 Mpbs or higher) and the location of the viewer. We refer to object i being requested by viewers at a specific location area a as a *location object*, denoted as location object i^a.

	CloudFront			MaxCDN			Liquid Web		
US	99.9	99.9	99.9	99.2	98.4	97.8	99.3	96.1	92.1
Brazil	100	100	99.9	98.6	70.5	24.4	99.6	0	0
Austria	99.9	99.9	99.8	97.6	96.7	95.3	97.0	42.2	0
Spain	99.9	99.9	99.9	98.7	96.6	95.1	99.4	47.3	0.2
Japan	99.9	99.9	99.9	97.5	95.8	77.1	99.7	0	0
China	99.9	99.9	99.8	91.1	24.7	0	1.6	0	0
Australia	100	100	99.9	94.7	89.5	0	99.7	0	0

Table 4.2: Measured CDN performance $p^a_{i,k}$ (3 content objects at streaming rates 1/2/3 Mbps).

To precisely characterize the performance of CDN k, in this chapter, we define $p^a_{i,k}$ as the fraction of times (*e.g.*, 90%) that CDN k can deliver i^a at the encoding rate of the object i. One may define $p^a_{i,k}$ for other contexts (*e.g.*, for images) and use other metrics (*e.g.*, 95-percentile latency).

Next consider *costs*. Different regions may have different resource (*e.g.*, bandwidth,

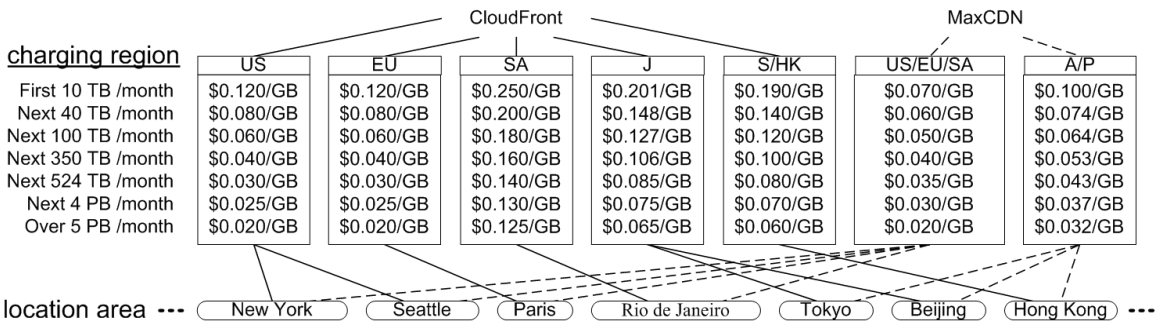


Figure 4.2: Charging structures of CloudFront and MaxCDN.

electricity) costs. Different CDNs may operate at different scales at different regions to negotiate different volume discounts. Hence, different CDNs may charge different costs to content publishers, and one CDN may charge differently at different regions.

Figure 4.2 shows the real charging structures of two CDNs: Amazon CloudFront and MaxCDN. We show these two structures because they are public and represent typical CDN charging structures. We make three observations. First, each CDN groups the locations of its edge servers into multiple regions and each region may have a different pricing model. We refer to each such region as a *charging region*. For example, CloudFront divides into 5 charging regions: US, EU, South America (SA), Japan (J), and Singapore/Hong Kong (SHK). MaxCDN divides into 2 charging regions: US/EU/SA, and Asia Pacific (AP). The total charge of a CDN to a content publisher is the sum of the charges at all of the charging regions. Second, denote the total traffic originated from the edge servers of a CDN located at a charging region during a billing period as the *charging volume* of the charging region; then the charging function of each charging region is a nonlinear concave function of the charging volume. Third, there can be large price diversity within a CDN as well as across CDNs. For example, CloudFront's charge for South America for next 100 TB is \$0.18, which is 3 times that for the US at the same traffic volume.

To precisely express the charging of CDNs, we let α_k^r be the charging region r of CDN k. For viewers from a location area a, each CDN has its own strategy to select servers

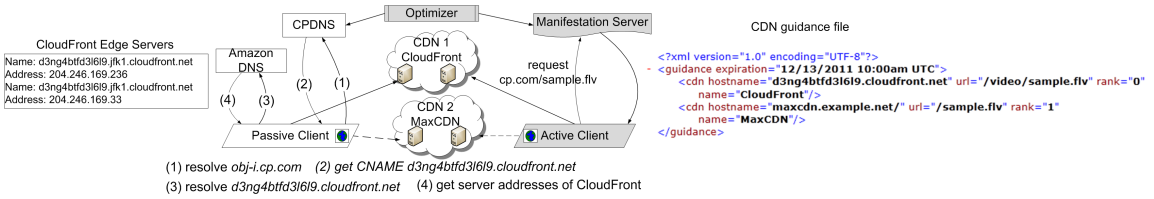


Figure 4.3: Content multihoming control framework (shaded components include our contributions).

in some α_k^r to serve them. This strategy is controlled by CDN k but can be observed by content publishers [70]. For instance, in our measurements, requests from Beijing are redirected to CloudFront's J region but to MaxCDN's US/EU/SA region. Let t_k^r denote the charging volume of CDN k at its charging region α_k^r, during a billing period. Specifically, the value of t_k^r is computed as the total traffic delivered during the billing period to content viewers at location areas who are pointed to α_k^r by CDN k. Then the total charge of CDN k to the content publisher is a sum of the charges at individual charging regions. Let $C_k^r(t_k^r)$ denote the charging function of CDN k for charging region α_k^r.

4.3 Control Framework

We adopt a general, practical content publishing control framework shown in Figure 4.3. A central Optimizer computes the configuration of directing viewer requests to CDNs. The configuration is sent to a DNS system, HTTP redirector, or a manifest-file server system to implement direction for specific viewer requests.

In particular, we distinguish two types of clients according to their capabilities, due to their effects on our problem formulation. The first type is passive clients. A key characteristics of passive clients is that they use one CDN edge server at a time. Although multiple CDNs and/or multiple servers from one CDN are available in content multihoming, such traditional clients at content viewers use only a single CDN server to serve a particular content object request [11]. For such clients, we assume that the results of con-

tent multihoming optimization are implemented by only server side mechanisms such as DNS redirection.

Specifically, suppose that content publisher `cp.com` uses DNS-based re-direction. For simplicity, the publisher assigns content object i URL: `http://obj-i.cp.com`. When a content viewer visits the URL, a DNS query from the client of the content viewer or the local DNS server of the content viewer to resolve `obj-i.cp.com` will be sent to the DNS server of the content publisher, labeled `CPDNS` in Figure 4.3. Looking up the IP address of the content viewer or the local DNS resolver of the content viewer, `CPDNS` obtains the location area a. Using the output from the Optimizer, `CPDNS` returns to the client the CNAME of a chosen CDN k. In the example shown in Figure 4.3, CloudFront is chosen. Note that in real implementation, as we will see in our computation, there is no need to assign per-object DNS name. Alternative implementations (*e.g.*, using HTTP-based re-direction) will be similar.

The second type is active clients. Such clients include an adaptation algorithm (*e.g.*, an Adobe Flash Actionscript) to utilize multiple CDN servers when retrieving a single content object. In particular, when the service rate of one CDN server is insufficient, an active client can use additional servers (from the same CDN or backup CDNs) to make up the deficit. The additional CDNs/servers are provided to the adaptation algorithm through a manifest file from the content publisher. Such manifest files are already used by some clients such as the Netflix clients. In the example shown in Figure 4.3, the CNAMEs of two CDNs are returned to an active client.

4.4 Problem Statements

With the preceding background and control framework, our problems are easy to state. Note that there is much flexibility in the deployment of our control framework. There can

be settings with only passive clients, or only active clients, or a hybrid. We first state the problem in the passive client only setting. An extension to active client only setting is straightforward and follows. Combining them together is also straightforward and hence omitted.

4.4.1 Passive client

Specifically, we formulate the content multihoming optimization problem as consisting two objectives:

QoE guarantee: First and foremost, for every object i and location area a, if $n_i^a > 0$, content multihoming should assign one or more CDNs for viewers from location area a requesting object i to achieve a QoE target. Each assigned CDN k should satisfy two requirements: (1) CDN k be capable of providing the required features (*e.g.*, streaming vs download) to deliver object i; (2) $p_{i,k}^a$ exceed the performance target. For concreteness, we use a major streaming media content publisher's QoE target format that $\bar{p}$ fraction of viewer requests can be satisfied without any QoE degradation. In this chapter, we use rate based $p_{i,k}^a$ for concreteness, and extension to latency-based is straightforward. Define F_k=$\{i^a$: CDN k can provide the features to deliver object i and $p_{i,k}^a \geq \bar{p}.\}$. In other words, F_k is the set of location objects that CDN k can serve.

Cost optimization: Under the QoE guarantee constraint, content multihoming may balance the load to multiple CDNs, in particular to minimize the total cost. Let $x_{i,k}^a$ denote the fraction of the n_i^a requests that is directed to CDN k. Hence, each $x_{i,k}^a$ is an optimization variable with a valid value range between 0 and 1. We state the problem **Q** as:

$$\underset{\{x^a_{i,k}\}}{\text{minimize}} \quad C(\{x^a_{i,k}\}) = \sum_k \sum_{\alpha^r_k} C^r_k(\sum_{a \in \alpha^r_k} \sum_i x^a_{i,k} n^a_i s_i)$$

$$\text{subject to} \quad \forall i, a, n^a_i > 0 : \sum_k x^a_{i,k} = 1;$$

$$\forall i, a, i^a \notin F_k : x^a_{i,k} = 0.$$

The first constraint states that each demand for an object i at a location a should be served. The second constraint states the QoE constraint. In other words, if CDN k cannot provide sufficient performance or feature for a location object i^a, no content viewers from location area a for object i should be directed to CDN k. Note that the QoE constraint may lead to infeasibility. Feasibility can be checked efficiently. In this chapter, we assume feasibility.

After computing a solution (*i.e.*, $\{x^a_{i,k}\}$), the Optimizer sends the solution to `CPDNS` in the control framework of Section 4 to implement it.

4.4.2 Active client

An active client allows the usage of multiple CDNs to serve the same request. One might think that this will add substantial complexity to the preceding formulation. But as we will see, it is a simple extension of the preceding problem definition.

Without loss of generality, consider that each active client uses two CDNs: one primary and one backup. Consider the following set of CDNs: each CDN is a "virtual CDN" that consists of a pair of CDNs: $k^{'} = (k, j)$, where k is the primary CDN and j is the backup. Then we will have a similar problem formulation as the preceding formulation.

First consider the QoE Guarantee. Define $F_{k'}$=$\{i^a$: both CDN k and CDN j can provide the features to deliver object i and $p^a_{i,k} \cup p^a_{i,j} \geq \bar{p}.\}$, where $\cup$ denotes the joint reliability of the two CDNs.

Next consider the objective function. Assume that each primary CDN k still delivers

the same amount of traffic. The backup CDN j incurs an additional traffic $1 - p^a_{i,k}$ fraction of the time. One may verify that $C(\{x^a_{i,k'}\})$ and $C(\{x^a_{i,k}\})$ have similar structure and can be solved with the same method.

After computing a solution for this setting (*i.e.*, $\{x^a_{i,k'}\}$), where each k' is a pair of CDNs. The Optimizer sends the solution to manifest file servers to return two CDNs for each active client request.

4.5 Computing Optimization

We now develop techniques to solve the problem defined in the preceding section. Since the problems for the passive clients and active clients have the same format, we use the passive client formulation. Our strategy is to first transform the problem to a combinatorial partitioning problem in Section 4.5.1. Then, in Section 4.5.2 we develop a novel, efficient algorithm that computes an optimal partition without enumerating all of the exponentially many partitions. We discuss extensions in Section 4.5.3.

4.5.1 Optimal content multihoming as object partitioning

At a first glance of the problem **Q**, one might think about using convex programming (*e.g.*, [23]) to solve the problem. Unfortunately, the objective function $C(\{x^a_{i,k}\})$, which we target to minimize, is a *concave* function, as it is a sum of multiple concave functions. Hence, traditional, efficient convex programming does not apply.

On the other hand, the concavity of the objective does lead to one observation: there is a minimizer of (4.1) such that each location-object is put into only one CDN. Precisely, we have the following result:

Lemma 5 *There exists an minimizer of $C(\{x^a_{i,k}\})$ for problem* **Q***, such that each location object is assigned to one and only one CDN. Formally, for any object i and location $a \in A$,*

there exists some $k^* \in K$ *such that* $(i, a) \in F_k$*, and*

$$\begin{cases} x^a_{i,k} = 1; & \textit{if } k = k^* \\ x^a_{i,k} = 0; & \textit{if } k \neq k^*. \end{cases}$$

Consider one such solution, and let π denote the mapping, according to the solution, from each location object i^a to its assigned single CDN k: $\pi(i^a) = k$. We make two observations. First, we can also write the mapping π in an equivalent partition format: $\pi := \{\pi_1, \ldots, \pi_K\}$, where π_k denote the set of location-objects that are assigned to CDN k. We use both formats in this chapter. Second, since π is derived from a valid solution to problem **Q**, it satisfies the QoE constraint: if $\pi(i^a) = k$, then $i^a \in F_k$. We refer to a partition satisfying the QoE constraint as a *feasible partition*.

The partition-based interpretation of the solution allows us to change problem **Q** into a partition-based formulation. Figure 4.4 illustrates the partition based formulation. The figure includes only the location objects i^a whose demand n^a_i is greater than 0. For each location object, it shows the candidate CDNs that the location object can be assigned to. CDN k is a candidate for location object i^a only if $i^a \in F_k$. The problem then is to assign each location object to one and only one CDN to minimize the cost.

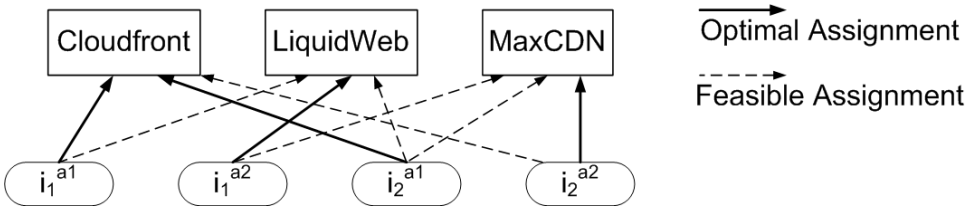


Figure 4.4: Example illustration: **Q** can be formulated as a partition problem.

An advantage of the discrete partition formulation is that it allows enumeration. A straightforward approach to finding an optimal partition is to enumerate all partitions, and select the best one among the feasible partitions . We know that each location object i^a can

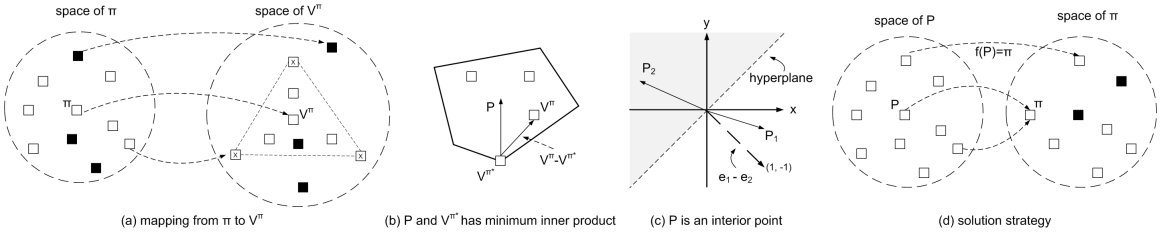


Figure 4.5: An example illustrating the basic idea to solve problem **Q**.

be assigned to one of K CDNs. Hence, the total number of partitions is $K^{|A|N}$. For $K = 2$ or 3, $|A|$ on the order of thousands and N hundreds of thousands, direct enumeration is clearly infeasible. In other words, the partition formulation allows enumeration, but naive enumeration does not work.

4.5.2 Efficient optimal partitioning

Our key insight to substantially reduce the complexity is that the naive enumeration of all of the exponential number of partitions is unnecessary. Instead, we need to consider only a polynomial number of partitions.

Basic idea: Specifically, consider the space of all possible partitions illustrated by the space on the left of Figure 4.5(a), where each partition is shown as a point. A feasible partition is shown as an white box while an infeasible partition is shown as a black box. Naive enumeration evaluates every partition, ignores a partition if it is infeasible, and picks a feasible partition that gives the best outcome.

Now instead of looking at the space of partitions, we look at the space of the *outcomes* of the partitions, illustrated by the right space in Figure 4.5(a). Each partition point in the left space has a corresponding outcome point in the right outcome space. Specifically, the outcome of a partition π is a vector, with each element of the vector representing the charging volume at charging region of CDN. Let V^π denote the multi-dimensional charging volume vector representing the outcome of a partition π. We will develop the

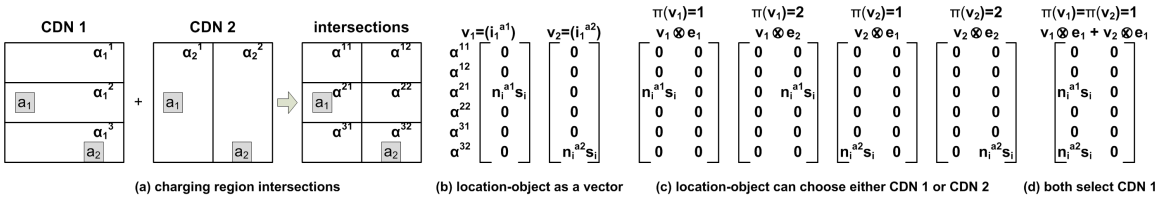


Figure 4.6: An example illustrating the charging-intersections.

exact representation shortly.

Since the objective function of our problem **Q** is a concave function of the charging-volume vector, we know from concave optimization theory that we need to evaluate the objective function only over the extremal points of the convex hull of the charging volume vectors produced by feasible partitions. In other words, if the charging volume vector V^π resulted from a feasible partition π is not an extremal point, the vector can be expressed as a convex combination of those resulted from some other feasible partitions, and hence there is no need to evaluate π. Figure 4.5(a) illustrates that we need to consider those partitions marked with an "x". As we will see, the number of extremal points is polynomial and we can identify them efficiently. Below, we develop the details of our algorithm.

Representing each location-object as a vector: The foundation of our basic idea is based on considering the resulting charging volumes of a partition π as a vector V^π. We now introduce a representation of each location object $v = i^a$ as a vector to allow easy aggregation on the outcome of a partition. This representation is quite simple but involves some notation complexity at the beginning. The benefit of the representation is that it provides essential insight and simplification during our development.

We first introduce charging region intersections. Recall that each CDN k defines a mapping from a location area a to one of its serving charging regions $\alpha_k^1, \cdots, \alpha_k^{R_k}$, where R_k is the number of charging regions of CDN k. Note that $\alpha_k^1, \cdots, \alpha_k^{R_k}$ provides a partition of all location regions A. An intrinsic complexity of multiple CDNs is the heterogeneity of their charging regions. Define the "intersections" of the charging regions of the K

CDNs. Let $\alpha^{r_1 r_2 \cdots r_K}$ denote the intersection of the charging regions r_1 of CDN 1, r_2 of CDN 2, and r_K of CDN K. Then a total of $R = R_1 * R_2 \cdots R_K$ intersections are defined. Figure 4.6(a) illustrates a setting of two CDNs with 3 and 2 charging regions respectively. At most $R = 6$ non-empty intersections may be defined.

With charging region intersections, we can represent each location object as a vector. Specifically, given the set of charging region intersections, one can observe that each location area a belongs to one and only one of the charging region intersections. Fix one ordering of the intersections. Then we can convert the traffic of each location object $v = i^a$ as an R-dimension vector with all elements except one being 0. The position of the non-zero element is the intersection that the location area a belongs to, and the value at the position is $n_i^a s_i$. When it is clear from the context, we use v to either represent the name of a location object i^a or the vector. Figure 4.6(b) shows the vector representations of two location objects.

By representing each location object $v = i^a$ as a R dimensional vector, we introduce a simple, linear outer-production operator to reflect the effect of assigning v to CDN k. Let e_k be a unit K-dimensional vector whose only non-zero element is at the k-th position and the value at the k-th position is 1. Define $v \otimes e_k$, which reflects the effect of assigning v to CDN k, as producing a $R \times K$ matrix such that v is at the k-th column and the other columns are zero. Figure 4.6(c) shows four examples, when we assign two location objects to two CDNs. For example, the first example shows $v_1 \otimes e_1$; that is, assigning v to CDN 1.

Given this definition of the outer-product and a partition π, we can calculate the effect of applying π to a location object as v^π, which is $v \otimes e_{\pi(v)}$, where $\pi(v)$ indicates the CDN that v is assigned to under partition π.

Aggregating the effects of a partition π on all location objects, we define

$$V^\pi = [\sum_{v \in \pi_1} v, \ldots, \sum_{v \in \pi_K} v] = \sum_v v \otimes e_{\pi(v)} \in \mathbb{R}^{R \times K}. \tag{4.1}$$

We now go back to identifying extremal partitions.

Identifying extremal partitions by identifying separation vector: The basis of our technique to identify the extremal points of a set of points is through the fundamental Separation Lemma [43]. Specifically, define $V^{\Pi} = \{V^{\pi}\}$, where π is a feasible partition. Then a sufficient and necessary condition for a specific $V^{\pi^*} \in V^{\Pi}$ to be an extremal point of V^{Π} is that there exists a vector P in the same dimension space such that the inner product $\langle P, V^{\pi} - V^{\pi^*} \rangle > 0$ for any other $V^{\pi} \in V^{\Pi}$. In other words, the inner product of P and V^{π^*} is smaller than the inner product between P and any other point $V^{\pi} \in V^{\Pi}$. For instance, in Figure 4.5(b) , $\langle P, V^{\pi} - V^{\pi^*} \rangle > 0, \forall \pi \neq \pi^*$.

Hence, a strategy to identify the set of all extremal points is to compute a set of Ps and from each P we compute an extremal feasible partition. Denote the computation from P to a partition as $f(P)$. Figure 4.5(d) illustrates a set of Ps and a mapping function $f()$. We want the following on conditions on the set of Ps and $f()$:

1. computationally efficient: both the set of Ps and $f(P)$ are easy to compute;
2. valid: each $f(P)$ is an extremal feasible partition;
3. exhaustive: for each extremal feasible partition π^*, there exists one P in the set of Ps, such that $f(P) = \pi^*$.

Below, we show a set of Ps and a function $f(P)$ satisfying the preceding conditions. **Function** $f(P)$: We start by developing $f(P)$. We have the following CDN Identification Lemma.

Lemma 6 (CDN Identification Lemma) *A feasible partition π^* is an extremal partition among the set of all feasible partitions if and only if $\exists P \in \mathbb{R}^{R \times K}$ such that $\forall v, k,\ k \neq \pi^*(v) \wedge v \in F_k$: $\langle P, v \otimes e_{\pi^*(v)} \rangle < \langle P, v \otimes e_k \rangle$.*

We name the lemma CDN Identification Lemma because it is the foundation to develop $f(P)$. Given a P in the lemma, we can compute a corresponding extremal partition π efficiently: for each location object $v = i^a$, iterate among all feasible CDNs k for the object (*i.e.*, $i^a \in F_k$), we compute $\langle P, v \otimes e_k \rangle$. We assign v to the (unique) CDN k attaining the minimal value: $\pi(v) = k$.

Set of Ps: We consider the following set of Ps: a P satisfies that all of the elements in $\{\langle P, v \otimes e_k \rangle | \forall k : v \in F_k\}$ are distinct. Formally:

$$\{P : \forall k, j, v, k \neq j \wedge v \in F_k \cap F_j : \langle P, v \otimes (e_k - e_j) \rangle \neq 0\}. \tag{4.2}$$

Since the conditions are stronger than those from the CDN Identification Lemma, we know that each such P can compute an extremal partition.

Geometrically, the condition that P satisfies $\langle P, v \otimes (e_k - e_j) \rangle \neq 0$ is equivalent to that P is not on the hyperplane that is orthogonal to $v \otimes (e_k - e_j)$. Denote this hyperplane as $[v \otimes (e_k - e_j)]^\perp$ and let $\mathbb{H}$ be the set of all these hyperplanes. Hence, a P satisfying all conditions in (4.2) is not on any of the hyperplanes in $\mathbb{H}$. In other words, P should be an interior point in a cell created with hyperplanes in $\mathbb{H}$ as boundaries. Efficient algorithms (*e.g.*, [72]) exist to enumerate one interior point from each cell.

Exhaustiveness: From the preceding development, it should be clear that we have developed a set of Ps and the function f which is computationally efficient, and $f(P)$ is valid. The only remaining issue is whether we satisfy the exhaustive requirement by enumerating an arbitrary interior P from each cell. First, we have

Proposition 7 *If π^* is an extremal feasible partition, then $\exists P$ which makes $f(P) = \pi^*$ and P is an interior point of a cell in $\mathbb{H}$.*

Proof. Suppose we find an extremal partition π^* from a point Q that is not an interior point of any cell in $\mathbb{H}$, we shall construct from Q another point Q' that is indeed an interior point

of some cell in $\mathbb{H}$ and that $f(Q') = \pi^*$. Since Q is not interior, there exists $v_0, k_0 \neq j_0$ such that $\langle Q, v_0 \otimes (e_{k_0} - e_{j_0})\rangle = 0$. Consider $P(\delta) = Q + \delta \cdot v_0 \otimes (e_{k_0} - e_{j_0})$, where $\delta \in \mathbb{R}^1$. We have that, for any $v, k \neq j$, $\langle P(\delta), v \otimes (e_k - e_j)\rangle = \langle Q, v \otimes (e_k - e_j)\rangle + \delta \cdot \langle v_0 \otimes (e_{k_0} - e_{j_0}), v \otimes (e_k - e_j)\rangle$, which is a continuous function of δ. So there exists some small enough $|\delta'| \neq 0$ such that:

(1) $\langle P(\delta'), v \otimes (e_k - e_j)\rangle < 0(\text{or} > 0)$ for all $v, k \neq j$ such that $\langle Q, v \otimes (e_k - e_j)\rangle < 0(\text{or} > 0)$; and

(2) $\langle P(\delta'), v_0 \otimes (e_{k_0} - e_{j_0})\rangle = \delta' \cdot \|v_0 \otimes (e_{k_0} - e_{j_0})\|^2 \neq 0$.

This means that $P(\delta')$ is on one less hyperplanes in $\mathbb{H}$ than Q. Moreover, since π^* is extremal, we have $\langle Q, v \otimes (e_{\pi^*(v)} - e_k)\rangle < 0$ for all $v, k, k \neq \pi^*(v) \wedge v \in F_k$, so it follows from (1) that $\langle P(\delta'), v \otimes (e_{\pi^*(v)} - e_k)\rangle < 0$ as well, hence $f(P(\delta')) = \pi^*$ by Lemma 6. This process can be repeated to yield a Q' that is not on any hyperplanes in $\mathbb{H}$ and that $f(Q') = \pi^*$. ■

A potential issue is that the interior P from the preceding lemma may not be the one that our algorithm uses. However, we have the following result, and establish the exhaustiveness of our approach.

Lemma 8 *Interior points from the same cell find the same extremal feasible partition.*

Proof. Let P_1 and P_2 are two interior points from the same cell. Suppose their corresponding extremal partitions $\pi_1^* \neq \pi_2^*$, then $\exists v_0$ which has $\pi_1^*(v_0) \neq \pi_2^*(v_0)$. Therefore, according to Lemma 6

$$\langle P_1, v_0 \otimes (e_{\pi_1^*(v_0)} - e_{\pi_2^*(v_0)})\rangle < 0$$

$$\langle P_2, v_0 \otimes (e_{\pi_1^*(v_0)} - e_{\pi_2^*(v_0)})\rangle > 0$$

which contradicts with the pre-condition that P_1 and P_2 are from the same cell. ■

Algorithm 4: CMO(V, $\{F_k\}$)

Input: V: location objects to be assigned.
Input: $\{F_k\}$: K CDNs and their feasibility sets.
Output: optPi: optimal partition

```
// Step 1: Identify hyperplanes;
HPs ← ∅ ;
foreach v ∈ V do
    vVec = vAsVector(v) ;
    foreach distinct (k, j) pairs do
        if (v ∈ F_k ∧ v ∈ F_j) then
            hpCandidate = norm(vVec ⊗ (e_k − e_j)) ;
            if (hpCandidate ∉ HPs) then
                HPs += hpCandidate
// Step 2: Compute interior Ps from hyperplanes;
Ps ← computePs(HPs);
// Step 3: Evaluate extremal partitions identified by Ps;
optPi ← null;
foreach P ∈ Ps do
    // compute extremal partition π identified by P;
    π ← null ;
    foreach v ∈ V do
        optOuter ← +∞;
        foreach CDN k do
            if v ∈ F_k ∧ ⟨P, v ⊗ e_k⟩ < optOuter then
                π(v) ← k ;
                optOuter ← ⟨P, v ⊗ e_k⟩
    // compare new extremal π with current optPi;
    if (π > currentOptPi) then
        optPi ← π
return optPi;
```

Redundancy elimination: On the surface, we need to enumerate all cells created by a total of $|A|NK(K-1)$ hyperplanes, where $|A|N$ is due to the number of possibilities for v, which is the number of location objects, and $K(K-1)$ is due to the number of possibilities for $(e_k - e_j)$. However, some of the $|A|NK(K-1)$ hyperplanes are redundant. Specifically, if one vector is just the scaling of another vector, they define the same hyperplane. For example, $v \otimes (e_k - e_j)$ and $v \otimes (e_j - e_k)$ give the same hyperplane. Hence,

we need to consider only distinct pairs of k and j. Also, consider two location objects $v_1 = i_1^{a_1}$ and $v_2 = i_2^{a_2}$. If their vector representations satisfy $v_1 = \lambda v_2$, where λ is a scalar, then they define the same hyperplane, for each pair of k and j. In other words, all of the location objects who are mapped to the same charging region intersection define the same hyperplane for each pair of k and j. Hence, the number of unique hyperplanes is at most $R\binom{K}{2}$.

Algorithm: For completeness, we specify the content multihoming optimization (CMO) algorithm in Algorithm 4. Note there are many ways to implement `computePs()`. We choose [72] because it can easily be parallelized running on multiple CPU cores and computers.

Example: To help readers understand the algorithm, we apply it to the simplest setting of two CDNs and both use one global charging region. This is a setting where one can solve problem **Q** using intuition. Specifically, in this setting, we can divide the location objects into 3 categories: V_1: the location objects that can be assigned to only CDN 1; V_2: the location objects that can be assigned to only CDN 2; and V_3: the location objects that can be assigned to either CDN 1 and CDN 2. Then the only remaining issue is to determine the assignments of objects in V_3. One can verify that the correct strategy is that we compare the objective function values of two alternatives: (1) assigning all objects in V_3 to CDN 1, with objects in V_1 and V_2 preassigned to their respective CDNs; and (2) assigning all objects in V_3 to CDN 2, with objects in V_1 and V_2 preassigned to their respective CDNs.

Now, we see how `CMO` solves the setting. In Step 1 (Lines 2 to 9), the algorithm computes that there is only one hyperplane defined by [1, -1]. In Step 2, `ComputePs` computes two interior Ps P_1 and P_2, where P_1 is any one vector in the lower right half-space (x-coordinate is larger than the y-coordinate) of Figure 4.5(c), and P_2 is any one in the upper left half-space (x-coordinate is smaller than the y-coordinate).

In Step 3, the algorithm first evaluates $P = P_1$ to compute an extremal partition. For

each location object $v = i^a$, if it is in V_1 or V_2, the algorithm assigns it to the only feasible CDN. If $v \in V_3$, $\langle P_1, v \otimes e_1 \rangle$ is the x-coordinate of P_1 times the traffic volume of object v, and $\langle P_1, v \otimes e_2 \rangle$ is the y-coordinate of P_1 times the traffic volume of v. We know that P_1 is chosen such that the x-coordinate is larger than the y-coordinate. Hence, P_1 produces the extremal assignment of assigning all objects in V_3 to CDN 1. The algorithm next evaluates $P = P_2$. It produces the the extremal assignment of assigning all objects in V_3 to CDN 2. At Line 24, the algorithm compares the two cases and picks the better one. Hence, it produces the intuitive results. For general settings where we can no longer appeal to intuition, the algorithm computes the optimal assignment efficiently.

4.5.3 Extensions

The basic algorithm developed in the preceding section can be extended to handle diverse practical issues including storage costs, per-request costs, capacity constraints (so that we can handle CDNs with subscription levels as each CDN with a capacity constraint), and dynamic streaming when the streaming rates can adapt. Below we present simple extension to handle per-transaction cost and CDN capacity constraints.

Per-request cost: Besides charging for traffic, many CDNs also include charges for the number of requests. For instance, CloudFront charges $0.0075 per 10,000 HTTP requests in US. Consider that CloudFront charges $0.12/GB for the first 10TB traffic as shown in Figure 4.2. One can calculate that if the object sizes are less than 6.25KB, then the per-request charge can be higher than the traffic charge. Hence, the per-request charge can be the major cost for content publishers providing small objects *e.g.*, small images).

Extending Algorithm 4 to consider both traffic and per-request charge is relatively straightforward. Specifically, in the preceding section, each location object is represented as a R-dimension vector with one non-zero element at the charging region of the object.

An extension to include per-request charge is to represent each location object as a $R+1$ dimension vector, with the one added dimension representing the number of requests for the object.

Hyperplane aggregation in this extended setting is also relatively straightforward. Since each location object i^a has two non-zero dimensions, traffic $n_i^a s_i$ and the number of requests n_i^a, we have that $v(i_1^{a_1}) = \lambda v(i_2^{a_2}), \lambda \neq 0$ if and only if (1) a_1 and a_2 are in the same charging region, and (2) $s_{i_1} = s_{i_2}$. One can further aggregate hyperplanes by discretizing the sizes of objects into ranges.

We use a simple example to demonstrate the per-request cost extension of CMO. Suppose there are two CDNs. CDN1 charges \$1/MB for traffic and \$0/request, and CDN2 charges \$0.1/MB for traffic and \$0.1/request. There are two 10KB objects with 30 and 49 requests respectively, , one object 25KB object with 20 requests, and one 1MB object with 1 request.

CMO goes through following steps to find out the optimal object assignment efficiently:

1. CMO represents the objects as four 2D-vectors: v_1 = [0.3, 30], v_2 = [0.49, 49], v_3 = [0.5, 20] and v_4 = [1, 1], where the first dimension is traffic/MB and the second is number of requests.

2. Line 3 $\sim$ 9: CMO constructs four hyperplanes h_1 = [0.3, -0.3, 30, -30], h_2 = [0.49, -0.49, 49, -49], h_3 = [0.5, -0.5, 20, -20] and h_4 = [1, -1, 1, -1]. After nomalization and de-duplication (Line 7, 8), three hyperplanes left: [1, -1, 100, -100], [1, -1, 40, -40] and [1, -1, 1, -1].

3. Line 11: It finds 6 interior points P_1 = [1, 0, 1, 0], P_2 = [-1, 0, -1, 0], P_3 = [70, 0, -1, 0], P_4 = [-70, 0, 1, 0] , P_5 = [20, 0, -1, 0] and P_6 = [-20, 0, 1, 0].

4. Line 14 $\sim$ 22: Six extremal object assignments are found (π := $\{$CDN1's ob-

jects}{CDN2's objects}): π_1 = $\{v_1, v_2, v_3, v_4\}\{\}$ π_2 = $\{\}\{v_1, v_2, v_3, v_4\}$, π_3 = $\{v_1, v_2\}$, $\{v_3, v_4\}$, π_4 = $\{v_3, v_4\}$, $\{v_1, v_2\}$, π_5 = $\{v_1, v_2, v_3\}$, $\{v_4\}$, π_6 = $\{v_4\}$, $\{v_1, v_2, v_3\}$.

5. Line 23 $\sim$ 25: By comparing the costs of the extremal object assignments, we get π_5 is optimal.

CDN Capacity Constraint: In preceding sections, we developed our algorithms based on the "pay as you go" price model in which there is no usage limit for any CDN. However, in reality some CDNs make a commitment with their content providers and they charge a one-time fee for a particular monthly resource usage upper bound (*e.g.*monthly total traffic volume). Also, they punish the traffic volume exceeded the upper bound with a much higher price. An interesting question is that for a content provider which is considering make commitments with such monthly-plan based CDNs, which data plan (usage upper bound) should it choose and how to assign traffic to CDNs to optimize its cost and performance.

To answer the question above, we introduce CDN capacity constraints for problem **Q**. Define T_k as the maximum traffic volume in a charging period of CDN k, the original cost optimization problem **Q** is adapted as problem **Qc**:

$$\begin{aligned}
\underset{\{x^a_{i,k}\}}{\text{minimize}} \quad & C(\{x^a_{i,k}\}) = \sum_k \sum_{\alpha^r_k} C^r_k\Big(\sum_{a\in\alpha^r_k}\sum_i x^a_{i,k} n^a_i s_i\Big) \\
\text{subject to} \quad & \forall i, a, n^a_i > 0 : \sum_k x^a_{i,k} = 1; \\
& \forall i, a, i^a \notin F_k : x^a_{i,k} = 0; \\
& \forall k : \sum_{i^a\in F_k} x^a_{i,k} n^a_i s_i \le T_k.
\end{aligned}$$

Because of the capacity constraints, Lemma 5 will no longer be true, which means a location object might be split into multiple CDNs in the optimal assignment.

To develop an efficient algorithm to find the optimal object assignments in problem **Qc**, we firstly claim two facts:

Lemma 9 *The feasible set of problem* **Qc** *is convex.*

Lemma 10 *For a given CDN k, the feasible set without k's capacity constraint is F and the one with k's capacity constraint is F_k. All the vertices of F who satisfy k's capacity constraint are vertices of F_k. Moreover, if a vertex of F_k is not a vertex of F, it is on the boundary of k's capacity constraint (b_k).*

Define an "edge" of a convex set as a line segment whose end points are two vertices and each point on it can only be a convex combination of points on itself.

Proposition 11 *If a vertex of F_k is not a vertex of F, it is on an edge of F. One of the edge's end points is outside F_k and the other is inside F_k.*

Proof. Suppose x^* is a vertex of F_k which is not a vertex of F, it is a convex combination of two points $x_0 \in F \wedge x_0 \notin F_k$ and $x_1 \in F_k$.

Assume x_1 is not on any edge of F, there exists two points $x_1 \in F_k$ and $x_2 \in F_k$ satisfying $x_1 = \lambda x_2 + (1-\lambda)x_3$, $(0 < \lambda < 1)$. The line segment between x_0 and x_2 (x_3) intersects b_k on point $x_2^{'}$ $(x_3^{'})$. Without losing generality, let x_0 be the zero point, we have: $x^* = \mu_1 x_1$, $x_2^{'} = \mu_2 x_2$ and $x_3^{'} = \mu_3 x_3$, where $0 < \mu_{1,2,3} < 1$. Hence we have:

$$\frac{x^*}{\mu_1} = \lambda \frac{x_2^{'}}{\mu_2} + (1-\lambda)\frac{x_3^{'}}{\mu_3}$$

Because x^*, $x_2^{'}$ and $x_3^{'}$ are all on b_k, we derive:

$$\frac{1}{\mu_1} = \lambda \frac{1}{\mu_2} + (1-\lambda)\frac{1}{\mu_3}$$

In summary we get:

$$x^* = \frac{1}{1+\mu}x_2^{'} + \frac{\mu}{1+\mu}x_3^{'}$$

where $\mu = \frac{(1-\lambda)\mu_2}{\lambda\mu_3}$. It is contradictory with the pre-condition that x^* is a vertex of F_k. Hence, x_1 should be on an edge of F. ■

With Proposition 11, we can find the vertices of **Qc**'s feasible set starting from the vertices of **Q** from Algorithm 4. In each step from feasible set F we add a new CDN's (k) capacity constraint and figure out the vertices of F_k on b_k by enumerating the edges of F intersecting b_k.

4.6 Active Clients

An active client may be provided with an ordered list of CDNs to use in serving a single content object to improve performance. The list may come from the result of our optimization algorithm in the preceding section or another source of guidance. Even though our optimization algorithm considers performance constraints, the filtering is based on statistics. Hence, an active client still uses multiple CDNs to adapt to specific real-time CDN performance dynamics, in particular, to improve QoE during CDN server failures.

4.6.1 Adaptation problem statement

An active client receives guidance on its adaptation behaviors. First, it is provided with a *prioritized* list of CDNs. Without loss of generality, we consider the case of two CDNs: the first *primary* CDN, and the second *backup* CDN. Second, each individual CDN on the list provides a small number (typically 2) preferred edge servers through its request routing mechanism (*e.g.*, DNS resolution or HTTP redirect). We consider servers from the same CDN having the same priority.

We define the problem statement of active client adaptation as implementing a simple

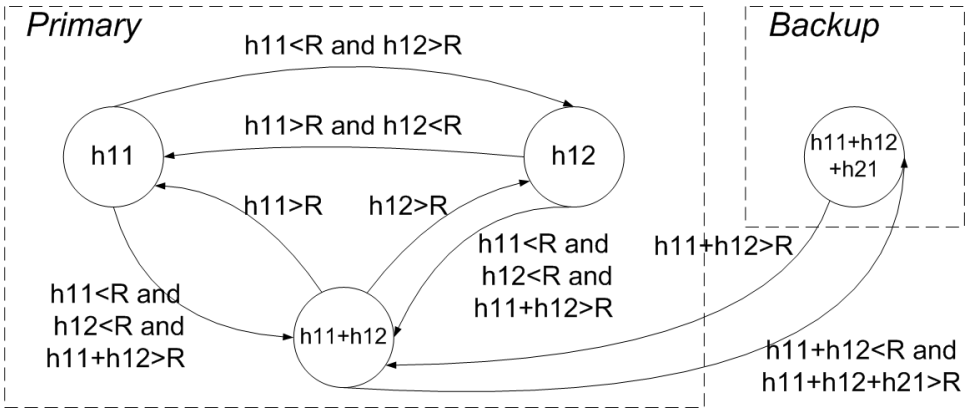


Figure 4.7: A downloading state transition diagram.

control state transition diagram. Figure 4.7 shows the diagram for a setting of two servers (`h11` and `h12`) from the primary CDN and one server (`h21`) from the backup CDN. An active client starts at a state of using one primary CDN server. The state is shown as `h11` in the figure, indicating that the client starts from downloading from `h11`. We construct the control diagram to achieve protection, priority guidance and stability:

- *Protection*: At a given state, if the existing servers together cannot provide the required QoE, the client adds one more server next in rank;
- *Priority guidance*: At a state using multiple servers, if sufficient capacity at a higher ranked server has become available, increase the usage of the new capacity and try to remove the usage of other servers;
- *Stability*: There is no switching overhead among servers from the same CDN (or same priority in general).

Note that one can introduce other transitions using our control state diagram based approach.

4.6.2 Adaptation algorithm: window AIMD and priority assignment

We implement the control diagram using the classic window-based AIMD (Additive Increase Multiplicative Decrease) scheme, based on a key observation that there is an analogy between the traditional congestion control and active client adaptation. In particular, if we consider the flow from each server to the client as a link, then we are solving a rate allocation problem among the links, where two essential mechanisms are needed: (1) the rate on a link should be reduced if the link is overloaded; and (2) a probing scheme is needed to utilize newly available capacity.

Specifically, for each sever h, the client maintains a *request window* size w_h to control the number of bytes that the client will request from h per T seconds, where T is a configuration parameter. We use the classic AIMD algorithm as a base to adjust the window sizes. In particular, if server h is capable of finishing its allocated request load, its window size w_h should be linearly increased; otherwise, w_h should be exponentially decreased.

However, our design problem is also different from the traditional congestion control problem, and hence we need to introduce novel and interesting techniques.

Total workload control: Naive usage of traditional AIMD will imply that all servers can be fully utilized to achieve a download rate that is as high as possible. However, this may not be necessary for the content viewer. Specifically, to achieve QoE of streaming content, the client only needs to sustain a sufficient downloading rate (*e.g.*, the video encoding rate R KB/s).

Based on this observation, we apply *total workload control* to appropriately limit the usage of the CDN servers. In our design, every T seconds, a new request task is added to the request queue of the client. The new task consists of only the next $R \cdot T$ KB content to be downloaded. Note that there can be remaining tasks in the queue to be completed when the new batch of task is added.

Priority task assignment: Total workload control does not yet achieve our adaptation goal. In particular, the load may still spread unnecessarily to too many servers. For example, a single primary server h11 has enough capacity to serve the client, but a backup server h21 may be also used unnecessarily for downloading, if h21 has equal opportunity fetching task from the request queue.

Our solution to this issue is *priority task assignment*. Specifically, whenever a request needs to be sent out, it always goes to the server at the highest rank, if its window allows. The complete algorithm is described in Algorithms 5 and 6. Our real implementation is a simple event loop.

Algorithm 5: WorkloadControl(newdata, server_list)

```
1 /* request_queue is a shared lock-free queue */
2 request_queue ← request_queue + newdata;
3 /* sort server_list by first h.rank then w_h */
4 foreach h ∈ server_list do
5     if request_queue = ∅ then
6         break;
7     if h.free then
8         PriorityAssign(h);
```

Discussions: Our window-based downloading adaptation algorithm is different from the traditional TCP congestion control. (1) To maintain client QoE, it requires (at application-level, during T seconds) that the total downloading rate across all servers to be at least video encoding rate. In particular, the sum window size should satisfy $\sum_i w_i > R \cdot T$. The adaptation algorithm enforces this by setting initial window size for a primary server (*e.g.*, w_{h11}) to be $R \cdot T$, and for each backup server to be 1. (2) Different from TCP's per-segment window update, we apply AI on the window size after all requests of the entire window are completed. In steady state (client streaming smoothly), this allows the client to slowly probe higher-ranked servers. Upon primary server failure or congestion, our AI strategy increases the backup server's window size; due to self clocking and before

Algorithm 6: PriorityAssign(h)

```
1  h.free ← false;
2  len ← min(w_h, request_queue.size);
3  assign ← request_queue[0 : len-1];
4  request_queue ← request_queue - assign;
5  (response, finished) ← HTTP_GET(h, assign) ;
   // Upon HTTP_RESPONSE:
6  /* put finished into content buffer */
7  if response = SUCCEED and w_h = len then
8      w_h ← w_h + 1;
9  else
10     w_h ← max(1, w_h/2);
11     request_queue ← request_queue + assign - finished;
12 if request_queue ≠ ∅ then
13     PriorityAssign(h);
14 else
15     h.free ← true;
16     return;
```

reaching the streaming rate, the increase behavior is similar to TCP-slow start, which is fast to allow request queue cleaning. (3) Although the algorithm maintains a window size for each server, it does not open a (TCP) connection to a lower-ranked server until necessary. Also, when the higher-ranked servers have sufficient capacity, the adaptation algorithm disconnects the lower-ranked servers.

We evaluate both the QoE protection effectiveness and the cost overhead of our algorithm by running real experiments (see Section 4.7).

4.7 Evaluations

In this section, we evaluate the cost and performance of our system design for content multihoming. In specific, we implement and test our optimization algorithm (CMO), the client adaptation algorithm, and the interactions between the two system components. We use real data to drive the run of our optimizer, and instrument clients on PlanetLab to

conduct experiments.

4.7.1 Evaluation methodology

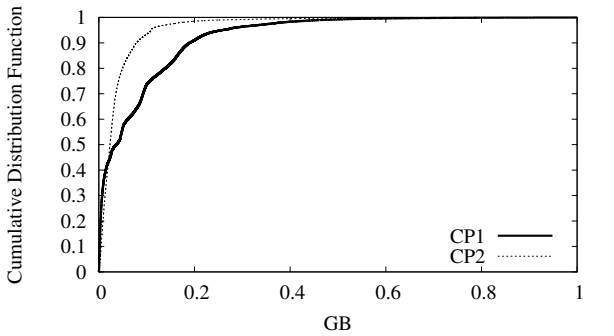
Content publishers: We evaluate our algorithms using real traces of content requests to two production Video-on-Demand (VoD) publishers. We name the content publishers CP1 and CP2 respectively. We consider each video as a content object, and collect the following information about each video: its size, and the number of times that it is requested from each city per month for a 6-month duration. Table 4.3 shows the summary statistics of the content objects. Figure 4.8 plots detailed statistical distributions of object sizes (s_i), number of requests (n_i) to each object, and traffic volume ($s_i n_i$) of each object. We notice that these distributions are long-tail distributions, which is consistent with other measurements [82].

We use MaxMind GeoIP database to map a client IP address in the trace to a location area. Our evaluations use the following definition of location areas: we start with each country as a location area; for a country with a large geographical span, we refine it to a next level; for example, we define each state in USA as a location area.

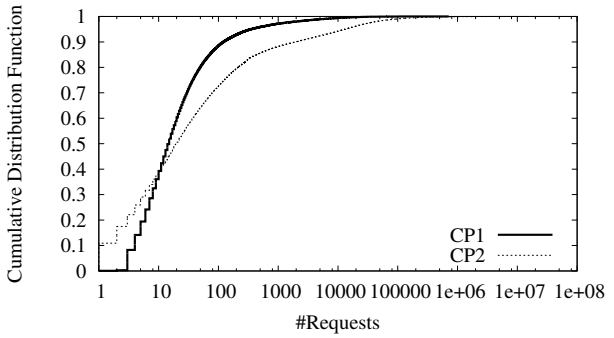
	# Objects	Sum of Obj Size	Total Traffic	#Request
CP1	529,411	40 TB	12,114 TB	153,129,348
CP2	667,856	71 TB	27,307 TB	390,235,440

Table 4.3: Summary statistics of content objects.

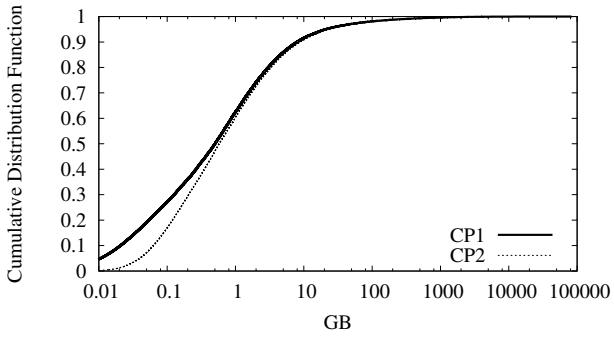
Content distribution networks: Our evaluation is based on three commercial CDNs: Amazon CloudFront, MaxCDN, and an anonymous private CDN which we refer to as CDN3. The geographic footprints of CloudFront and MaxCDN are shown in Figure 4.1, and the real charging structures and parameters of CloudFront and MaxCDN are shown in Figure 4.2. The server distribution and detailed price information for CDN3 are not shown due to privacy.



(a) object size



(b) number of requests



(c) total traffic volume

Figure 4.8: Statistics of object size, number of requests to each object, and total traffic for each object.

In our evaluations, we require that $p^a_{i,k} \geq 90\%$ for each CDN k and each location object i^a.

To study how each CDN maps a location area to its charging region, we deploy a measurement client on each one of 536 available PlanetLab machines to request objects from each CDN. We use `traceroute` to determine the locations of the CDN servers, as the GeoIP database can be inaccurate, *e.g.*, all CloudFront servers are always classified

as in Seattle, WA, US. After computing the charging region intersections of the three CDNs, we pick the top 5 intersections that contain the most traffic. Table 4.4 shows the percentages of traffic to major geographical regions.

	US	EU	SA	Asia & Pacific	Japan
CP1	77 %	11 %	6 %	5 %	1 %
CP2	19 %	7 %	1 %	71 %	2 %

Table 4.4: Traffic distribution across major geo regions.

Optimizer, Client, and Content Deployment: Our publishing optimizer is implemented in C++ and runs on a commodity PC with 2 quad-core Intel Xeon 2.33 GHz CPUs and 3 GB of memory.

We integrate our client adaptation algorithm into a production-based Adobe Flash video player. We leverage `NetStream.appendBytes` in Adobe Flash (supported by version 10.1 and above) to integrate multiple CDN servers for one video streaming session. To collect the performance metrics, our player periodically reports logs to a logging server through HTTP. Our player is deployed on CDN3 and can be accessed by any web browser on the Internet, including PlanetLab nodes. We then install Mozilla Firefox with Adobe Flash Player (using Xvnc as XServer) on 412 PlanetLab nodes and instrument these clients to conduct video streaming experiments. Based on our traces, we select the PlanetLab nodes according to their availability and geographical locations. We also generate client request load according to the video request patterns of CP1. Note we use PlanetLab for experiments to avoid compromising real users' experience and privacy.

We use sampled videos from CP1 for evaluation. We deploy the video content on CloudFront and CDN3. CloudFront has the best performance according to our PlanetLab measurement (see Table 4.2). We are able to run Adobe Flash Socket Policy service on CDN3, thus the clients can use customized and optimized TCP sockets for requests to this

private CDN. We also run multiple pre-tests (before the evaluation) to warm up both CDNs *i.e.*, the edge servers prefetched the video content before the experiments start.

CDN server capacity failure models: We evaluate the effectiveness of our client adaptation algorithm under both controlled stress tests and real server congestions. In the real experiments using two CDNs and PlanetLab clients, we do *not* inject any failures, but the CDN servers can get temporarily congested due to the bursty nature of client request load.

Performance metrics: We evaluate both the *CDN cost* and *client QoE* of our content multihoming optimization. CDN cost is simply the total charge (in USD) by all CDNs given a CDN assignment, *i.e.*, the value of function $C(\{x^a_{i,k}\})$ defined in (4.1). For client QoE, we use three performance metrics: (1) *freezes*, the frequency (number of times per view) a viewer encounters rebuffering during the video view, excluding the cases due to initial start or user drag (seek). As observed in [32], freezes are a major factor reducing viewer's QoE. (2) *smoothness ratio*, the percentage of the clients that never encounter freeze. This is a statistical performance metric. (3) *buffering time*, viewer visible buffering time (in seconds) per video view, including startup delay and seek delay.

Algorithms: We evaluate three algorithms to assign content objects to CDNs:

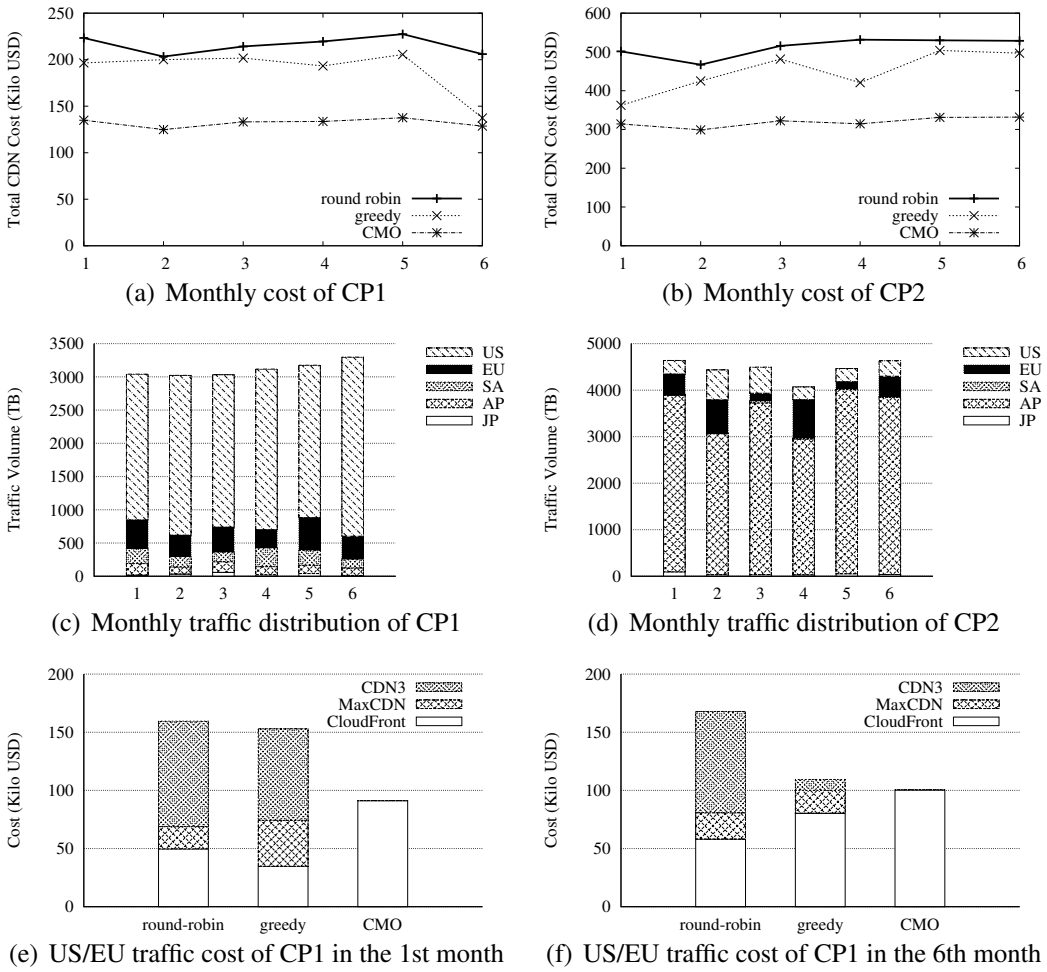
- CMO: This is the `CMO` algorithm defined Section 4.5.2. We also report results for CMO extensions defined in Section 4.5.3.
- greedy: This algorithm assigns location objects sequentially in a uniformly random order. When assigning the next object, the algorithm computes the cost to be reached when the object is put in each feasible CDN. The object is assigned to the CDN resulting the lowest cost among the alternatives.
- round-robin: This algorithm also assigns location objects sequentially in a uniformly random order. An CDN index is maintained. When assigning the next object, the algorithm uses round-robin, starting from the current CDN index, to assign the object to a

feasible CDN.

4.7.2 Publishing cost optimization

We start by evaluating the CDN cost savings of our CMO algorithm. At the beginning of a month, for each location area a, we use the content traffic to a in last month as the traffic prediction in this month. We leverage the three algorithms listed in Section 4.7.1 to decide how to redirect locational requests to the three CDNs. We use the real traffic in this month to calculate the total monthly cost of CP1 and CP2. Figure 4.9 shows the monthly cost of the two content publishers with different CDN assignment algorithms. From Figure 4.9(a) and 4.9(b) we make following observations. First, CMO saves sufficient cost for both CP1 and CP2 every month, compared with both round-robin and greedy, except that in the last month of CP1 greedy does almost as well as CMO. Second, CMO is efficient even though we use the traffic distribution in last month to predict the one in the coming month. Figure 4.9(c) and 4.9(d) shows the traffic distribution among each region in each month, from which we find that the traffic distribution is relatively stable and CMO is not very sensitive to the traffic prediction errors.

To better understand why CMO saves more cost than greedy, we look into US/EU traffic cost of CP1 in the 1st month and the 6th month in Figure 4.9(e) and 4.9(f). In US and EU, CDN3 has a constant price ($0.10 per GB). Also, according to Figure 4.2, when the traffic volume is lower than 10 TB, both CDN3 and MaxCDN are cheaper than CloudFront. Hence, greedy will choose CDN3 (for location objects cannot be served by MaxCDN) rather than CloudFront until it meets a location object which has a large enough traffic volume (¿ 10TB) to make greedy find CloudFront cheaper. Therefore, the reason that greedy computes a cost that is close to CMO in the 6th month is that greedy meets a location object whose traffic volume is larger than 10TB in the beginning of the process.



(a) Monthly cost of CP1 (b) Monthly cost of CP2

(c) Monthly traffic distribution of CP1 (d) Monthly traffic distribution of CP2

(e) US/EU traffic cost of CP1 in the 1st month (f) US/EU traffic cost of CP1 in the 6th month

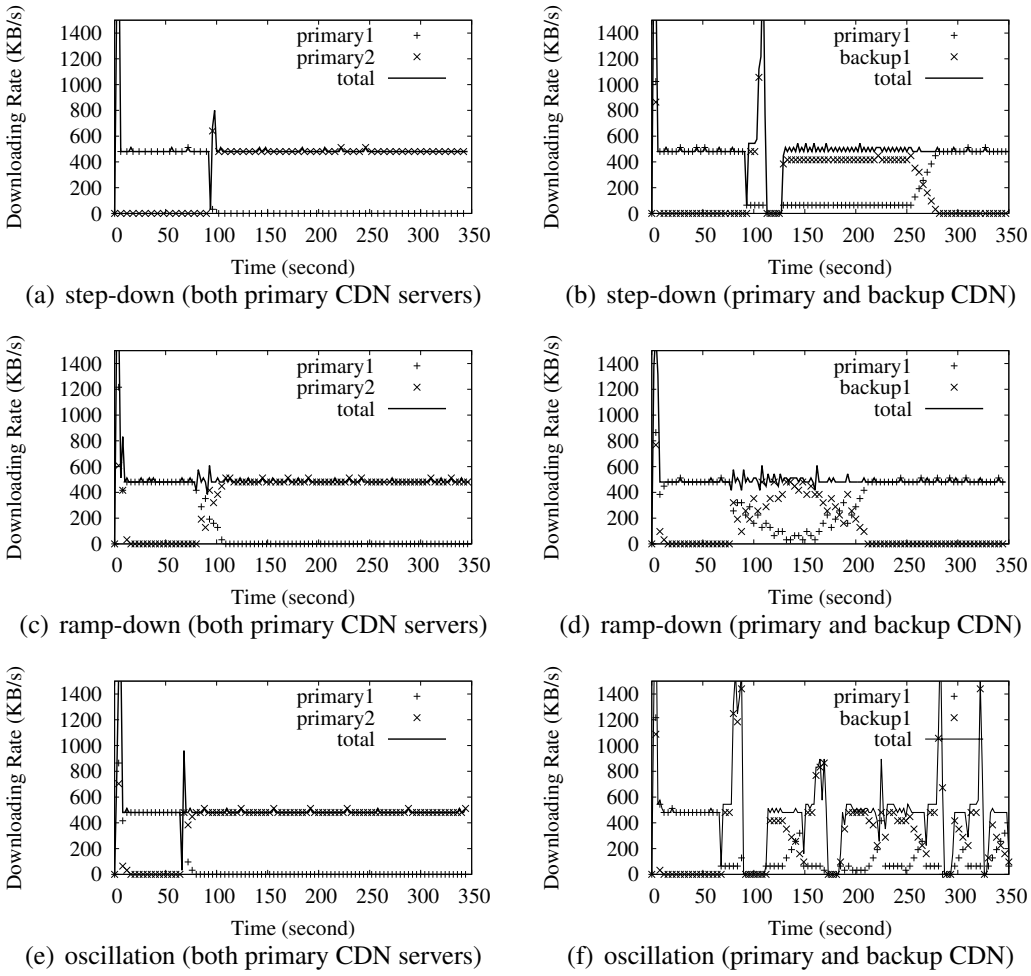
Figure 4.9: Cost and traffic distributions in the 6 months with different CDN assignment algorithms.

4.7.3 Client QoE adaptation

Given the optimized CDN assignment, passive clients can achieve high performance, as we will show in our PlanetLab experiments. However, individual passive client cannot handle the failures or congestions of primary CDN capacity, and may encounter QoE degradation at times. We demonstrate the effectiveness of our adaptation algorithm in the active clients, which achieves per-view QoE.

Stress tests: We start with stress tests when delivering a 1080p HD video object encoded at 480 KB/s. We run two sets of experiments: (1) only one CDN (primary), which has two

servers named `primary1` and `primary2`; (2) two CDNs (one primary+one backup), each with one server, named `primary1` and `backup1` respectively. In each set, we vary the capacity of `primary1` in the following three cases: (1) step-down, in which the capacity of `primary1` is reduced down to only 10% of video encoding rate and then recovers after 2 min; (2) ramp-down: in which the capacity of `primary1` is linearly decreased (to 10% of the video encoding rate) in one minute and then linearly increased back; (3) oscillation, in which the capacity of `primary1` periodically falls down (to 10% of the encoding rate) and then recovers after 20 seconds. We plot detailed downloading rates to observe more behaviors. Figure 4.10 plots the results.



(a) step-down (both primary CDN servers)
(b) step-down (primary and backup CDN)
(c) ramp-down (both primary CDN servers)
(d) ramp-down (primary and backup CDN)
(e) oscillation (both primary CDN servers)
(f) oscillation (primary and backup CDN)

Figure 4.10: Stress tests of client adaptation in CDN server failure cases.

We make multiple observations on client behaviors.

First, in all 6 cases, the client downloads at full speed at the beginning to build the required 4-second video buffer before playback can start. After starts, the active client continues downloading to maintain a 16-second video buffer (total workload control).

Second, despite the fluctuations of `primary1`, the active client achieves protection by downloading from the alternative server. In (b) and (e), under gradual `primary1` capacity changes, the aggregated downloading rate (labeled `total`) never falls below the streaming rate at any instance of time. In the other 4 cases, the instantaneous total rate may drop below the streaming rate, when the client need to finish the failed request task and catch up with the new scheduling. Note most instantaneous dips do *not* lead to viewer visible freezes because of the streaming buffer. Actually, in all 6 cases, we can view the entire video without seeing the rebuffering wheel.

Third, the active client prefers `primary1` over `backup1`. For example, in (d), `primary1` recovers at 250s and its utilization starts to increase, and at time 280s all requests have shifted from `backup1` back to `primary1`. One can also observe this "shifting-back" in (e) at time 160s-210s.

Fourth, as a contrast to the previous observation, our active client achieves "stickiness". For example, comparing (a)(b) with (d)(e), we observe that in (a) and (b), there is no shifting-back to `primary1`, as `primary2` can handle the load alone and belongs to the same CDN as `primary1`.

Fifth, the active client performs relatively the worst in (f), when there is a single primary CDN server and the capacity of the server fluctuates widely. We observe multiple downloading spikes. Detailed logs show the reason is the recovery from low rates resulted from HTTP request timeouts. In practical deployment, it is recommended that a content publisher uses a primary CDN which offers client *multiple* edge servers. The fluctuation of client downloading rates reflects server capacity fluctuations.

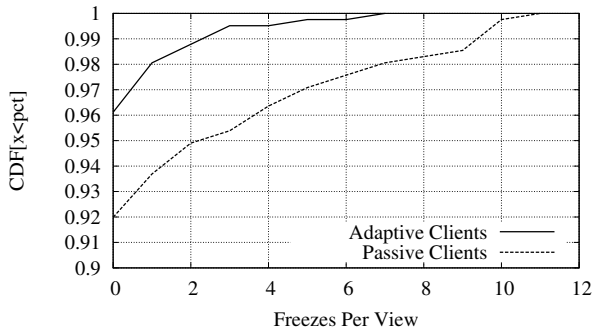
PlanetLab experiments: We next evaluate the statistical performance for both passive and active clients with real PlanetLab experiments. First, we measure the freezes and the smoothness ratio. Figure 4.11(a) shows the results. We observe that with passive clients, 8% clients experience at least one freeze. Thus, the smoothness ratio is 92%, which is considered as high performance in industry. Our adaptive clients further improve the client QoE and reduce the percentage of freezing clients to only 3.8%. Thus, our active client algorithm reduces QoE degradation by 51%. We also calculate from the figure that the active clients reduce the average number of freezes from 4.78 (for passive clients) to 2.19. Second, Figure 4.11(b) shows the buffering time performance. Active clients reduce the average buffering time from 9.6 seconds to less than half at 4.5 seconds.

Cost impact of client adaptation: The active clients might increase publisher cost for two reasons. First, the optimization algorithm considers the impact of active clients by predicting the amount of traffic shifted to backup. Reality may be different from the prediction. Second, for simplicity, during evaluation, our optimization algorithm does not consider the number of requests increase due to backup protection.

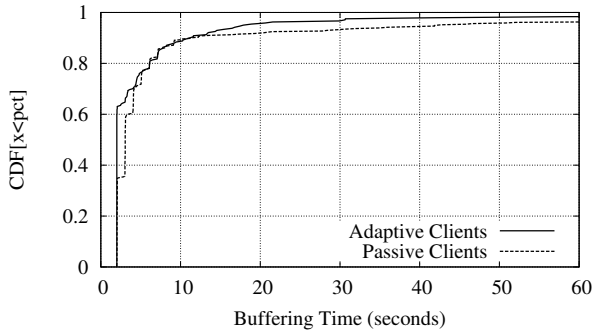
We evaluate the cost impact of client adaptation by comparing the computed optimal cost, the real cost during our PlanetLab experiments (after scaling up the traffic), and the cost of using round-robin CDN assignment. To better understand this impact, we further break down the cost into "traffic cost" and "per-request cost". Figure 4.11(c) shows the result. We observe the cost impact in total is less than 5.6% ($\sim 8,000$ USD added to $\sim 142,000$). Traffic deviation is less than 2.1% from prediction and contributes to 1.4% of the total 5.6% difference. The addition number of requests accounts for 4.2% of the total cost. Hence, it does not change the overall effectiveness in terms of cost savings.

Comments: in summary, we observe the client adaptation algorithm can achieve the following :

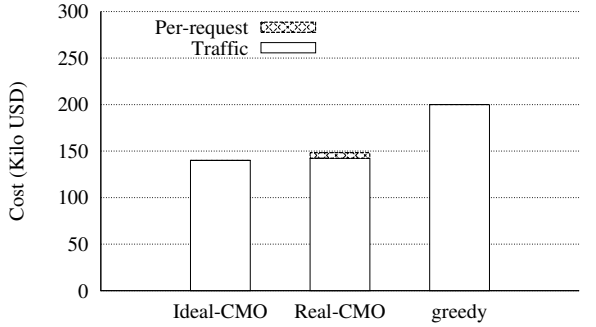
- Protected per-view QoE: for an individual viewer, if at any time, all the CDN servers (in



(a) Freeze per view statistics



(b) Buffering time per view statistics



(c) Cost impact of client adaptation

Figure 4.11: Per-view QoE in PlanetLab experiments.

the guidance list) together can serve all the requests (from this viewer and other clients), the viewer can achieve the target QoE, even if the primary CDN has insufficient capacity.

- Guided traffic distribution: if each CDN has sufficient capacity to serve all the assigned (by optimizer) clients at any time, the traffic distribution will follow the optimization guidance.

  If some CDN server's capacity is insufficient to serve all assigned clients at a given time,

then the clients try to fully utilize the server's capacity.

- Stable persistent downloading: if there exists a single primary CDN server who can serve all requests at any time, then a client tries to download only from one primary CDN server and maintains only one persistent connection.

4.8 Related Work

The importance of content multihoming has led to substantial industrial related work. We divide these industrial systems and efforts into three categories [26].

A first category is software systems which we name CDN switchers (*e.g.*, [8, 66]) and integrators (*e.g.*, [9, 64]. For example, the One Pica Image CDN extension [66] of the Magento Commerce platform provides an API to support the integration with multiple CDNs, including Amazon S3, Coral CDN, Mosso/Rackspace Cloud Files, and any CDN server or service that supports FTP, FTPS, or SFTP. Their objective, however, is not on the algorithms to effectively use multiple CDNs, but rather on usability issues such as seamless switching from one CDN provider to another. Commercial systems such as [9,64] provide CDN services based on aggregation of multiple CDNs. They can benefit from using our algorithms.

Going beyond the CDN switchers and integrators is a category of systems named CDN Load balancers. There are many CDN load balancers commercially available, including Cotendo CDN balancer [30], LimeLight traffic load balancer [58], Level 3 intelligent traffic management [57], and Dyn CDN manager [33]. Some of these systems offer quite flexible specification of rules to split CDN traffic among multiple CDNs. For example, Cotendo CDN balancer allows a content publisher to specify balancing rules including weighted allocation, geographic location, geographic distance, time of day, and any combination. In particular, the weighted allocation rule allows a publisher to specify: x percent

to CDN one, y percent to CDN two, and so on. A key missing component of the existing systems, however, is the key algorithms to compute the allocation, for example, the percentage. Hence, the output of our optimizer can use the existing systems as implementation.

There are also client based CDN load balancers. One interesting example is Conviva [29], whose video player plug-in performs continuous video quality monitoring, and could perform automatic CDN and/or source server switching during video playback. The exact details of their algorithm, however, is unknown.

A third category of related industrial efforts is CDN interconnect. In [65], a CDN interconnect (CDNi) architecture has been proposed so that a content publisher contracts with a few upstream CDNs, who may delegate some requests to downstream CDNs. The delegation relationship can be recursive to form a directional delegation graph, and all of the involved CDNs are said to form a CDN federation [22]. Our algorithms can be extended to the CDNi setting by considering a set of connected CDNs as a single logical CDN.

So far content multihoming has not been a focus of academic research. A related recent academic work is a measurement study of NetFlix [11]. The paper shows that similar to many content publishers, Netflix uses content multihoming. The paper conducts a measurement study and shows that there are indeed potential performance benefits of using content multihoming.

We refer to our system as content multihoming to draw an analogy with traditional Internet multihoming (*e.g.*, [12, 41]). However, content multihoming is quite different from traditional ISP multihoming. For example, while ISPs typically have a uniform price based on traffic, CDNs charge customers by regions.

4.9 Summary

In this chapter, we have conducted the first systematic study on content multihoming, by introducing the CMO algorithm and the client adaptation algorithm to optimize both the cost and the performance for content multihoming. Our realistic evaluations show that our content multihoming algorithms reduce publishing cost by up to 40%, and reduce viewer QoE degradation by 51%.

Chapter 5

Conclusions

In conclusion, this dissertation is about network dynamics. Traditionally, people usually treat dynamics in networks as exceptions and only handle them after they occur. In this dissertation, we introduce an alternate approach and show that it is better to understand the nature of traffic behaviors under network dynamics and protect traffic proactively in traffic planning. By doing so, we were able to design novel and efficient models and algorithms, and build practical systems that manage traffic intra-datacenter, inter-datacenter and inter-infrastructure in a smooth and safe way. Specifically, we make the following contributions:

- We propose and realize Forward Fault Correction (FFC) which is the first proactive approach to handling both data-plane and control-plane faults in traffic engineering.
- We propose and realize Smooth traffic Distribution Transition (SDT) which is the first network maintenance primitive which avoids congestion during network routing updates when routing rules in network devices are inconsistent.
- We propose and realize Content Multihoming Optimization (CMO) which is the first framework that optimizes users' quality of experience and content publishers' infrastructure usage cost jointly for latency sensitive content services.

5.1 Future Work

Latency sensitive applications are becoming more and more critical in people's daily life. Therefore, how we can protect the performance of latency sensitive applications under limitations in reality will be continuously a hot and crucial topic in the future. In this dissertation, we focus on exploring the strategic traffic planning that helps to avoid congestion on network and server links inside infrastructures, because most of the applications are using clouds as their backends, and congestion in clouds is one of the major cause of performance degradation.

However, latency sensitive applications could also suffer from other factors, such as delays and congestion on edge networks. Edge networks are the networks from the edges of clouds to users' access networks (*e.g.* ADSL, WiFi, 3/4G) and devices. The difficulty for directly using the approaches presented in this dissertation to avoid congestion on edge networks lies on the fact that clouds and edge networks are controlled by different providers. Therefore, it is not easy to make these providers coordinate together to optimize their networks for one type of applications. One potential solution is that the providers of edge networks open their network controls via some APIs to cloud providers or application providers. Another way is that cloud providers deploy edge servers that are "closer in network " to end users and move appropriate backend processing into the edge servers.

The approaches presented in this dissertation can also be improved in several ways. For instance, FFC currently operates on a given network topology, while an interesting research problem is whether we could improve the network topology to increase network throughput with robustness via FFC. In addition, zUpdate is using linear programming (LP), which is computational intensive, to find an update plan. How to replace the large scale LP with some smaller scale LPs or discrete algorithms to accelerate and computation is also important for zUpdate's practicality.

Bibliography

[1] CPLEX. http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[2] Floodlight. http://floodlight.openflowhub.org/.

[3] iPerf. http://sourceforge.net/projects/iperf/.

[4] Microsoft Solver Foundation. http://msdn.microsoft.com/en-us/devlabs/hh145003.aspx.

[5] MOSEK. http://mosek.com/.

[6] OpenFlow 1.0. http://www.openflow.org/documents/openflow-spec-v1.0.0.pdf.

[7] OpenFlow 1.1. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.

[8] 01box. http://cdn.01box.net/plug-ins/wordpress/index.php.

[9] 3crowd. http://www.3crowd.com/solutions/multi_cdn_manager/.

[10] Ed A. Atlas and Ed A. Zinin. Basic Specification for IP Fast Reroute: Loop-Free Alternates.

[11] Vijay Kumar Adhikari, Yang Guo, Fang Hao, Matteo Varvello, Volker Hilt, Moritz Steiner, and Zhi-Li Zhang. Unreeling netflix: Understanding and improving multi-CDN movie delivery. In *IEEE INFOCOM'12*.

[12] Aditya Akella, Bruce Maggs, Srinivasan Seshan, Anees Shaikh, and Ramesh Sitaraman. A measurement-based analysis of multihoming. In *ACM SIGCOMM'03*.

[13] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI'10*.

[14] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference*.

[15] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP DCTCP. In *Proceedings of the ACM SIGCOMM 2010 Conference*.

[16] David Applegate, Lee Breslau, and Edith Cohen. Coping with Network Failures: Routing Strategies for Optimal Demand Oblivious Restoration. In *SIGMETRICS'04*.

[17] David Applegate and Edith Cohen. Making Intra-domain Routing Robust to Changing and Uncertain Traffic Demands: Understanding Fundamental Tradeoffs. In *Proceedings of the ACM SIGCOMM 2003 Conference*.

[18] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards Predictable Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*.

[19] K. E. Batcher. Sorting Networks and Their Applications. In *AFIPS'68 (Spring)*.

[20] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *USENIX OSDI'10*.

[21] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *CoNext'11*.

[22] G. Bertrand, E. Stephan, G. Watson, T. Burbridge, P. Eardley, and K. Ma. Use cases for content delivery network interconnection. `http://tools.ietf.org/html/draft-ietf-cdni-use-cases-02`, January 2012.

[23] Dimitri Bertsekas. *Convex Analysis and Optimization*. Athena Scientific, 2003.

[24] Ramesh Bhandari. *Survivable Networks: Algorithms for Diverse Routing*. Springer Press, 1999.

[25] Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *HotSDN'12*.

[26] CDN expert. `http://cdnexpertonline.com/node/45`.

[27] Cisco Systems. Cisco Visual Networking Index: Forecast and Methodology, 2009-2014. `http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360_ns827_Networking_Solutions_White_Paper.html`.

[28] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *NSDI'05*.

[29] Conviva. `http://www.conviva.com`.

[30] Cotendo CDN Load Balancer. http://www.cotendo.com/services/CDN-Balancer/.

[31] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalag, Puneet Sharma, and Sujata Banerjee. Devoflow: Scaling Flow Management for High-Performance Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*.

[32] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. In *ACM SIGCOMM'11*.

[33] Dyn CDN manager. http://dyn.com/dns/dynect-managed-dns/cdn-manager/.

[34] Anwar Elwalid, Cheng Jin, Steven Low, and Indra Widjaja. MATE: MPLS Adaptive Traffic Engineering. In *INFOCOM'01*.

[35] S. Even, A. Itai, and A. Shamir. On the Complexity of Timetable and Multicommodity Flow Problems. *SIAM Journal on Computing*, 1976.

[36] Nick Feamster and Hari Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *NSDI'05*.

[37] P. Francois, O. Bonaventure, B. Decraene, and P. A. Coste. Avoiding Disruptions During Maintenance Operations on BGP Sessions. *IEEE Trans. on Netw. and Serv. Manag.*, 2007.

[38] Geo best-of YouTube. http://geobestofyoutube.gmapify.fr/.

[39] Soudeh Ghorbani and Matthew Caesar. Walk the Line: Consistent Network Updates with Bandwidth Guarantees. In *HotSDN'12*.

[40] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*.

[41] David Goldenberg, Lili Qiu, Haiyong Xie, Yang Richard Yang, and Yin Zhang. Optimizing cost and performance for multihoming. In *ACM SIGCOMM'04*.

[42] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving High Utilization with Software-driven WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference*.

[43] A.D. Ioffe and V.M. Tihomirov. *Theory of Extremal Problems*. Elsevier Science Ltd, 1979.

[44] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined Wan. In *Proceedings of the ACM SIGCOMM 2013 Conference*.

[45] John P. John, Ethan Katz-Bassett, Arvind Krishnamurthy, Thomas Anderson, and Arun Venkataramani. Consensus Routing: the Internet as a Distributed System. In *NSDI'08*.

[46] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. Walking the Tightrope: Responsive Yet Stable Traffic Engineering. In *Proceedings of the ACM SIGCOMM 2005 Conference*.

[47] Srikanth Kandula, Jitendra Padhye, and Paramvir Bahl. Flyways to De-Congest Data Center Networks. In *NotNets'09*.

[48] Koushik Kar, Murali Kodialam, and T. V. Lakshman. Routing Restorable Bandwidth Guaranteed Connections Using Maximum 2-route Flows. *IEEE/ACM Transactions on Networking*, 2003.

[49] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI'13*.

[50] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *NSDI'12*.

[51] Eric Keller, Soudeh Ghorbani, Matt Caesar, and Jennifer Rexford. Live Migration of an Entire Network (and its hosts). In *HotNets'12*.

[52] Michael Kende. Internet Global Growth: Lessons for the Future, September 2012.

[53] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying Network-Wide Invariants in Real Time. In *HotSDN'12*.

[54] Murali Kodialam, T. V. Lakshman, and Sudipta Sengupta. Efficient and Robust Routing of Highly Variable Traffic. In *HotNets'04*.

[55] Murali Kodialam, Associate Member, T. V. Lakshman, and Senior Member. Dynamic Routing of Restorable Bandwidth-guaranteed Tunnels using Aggregated Network Resource Usage Information. *IEEE/ACM Transactions on Networking*, 2003.

[56] Rupa Krishnan, Harsha V. Madhyastha, Sridhar Srinivasan, Sushant Jain, Arvind Krishnamurthy, Thomas Anderson, and Jie Gao. Moving beyond end-to-end path information to optimize CDN performance. In *ACM IMC'09*.

[57] Level 3 Intelligent Traffic Management. http://www.level3.com/~/media/Assets/brochures/brochure_intelligent_traffic_management.pdf.

[58] Limelight Traffic Load Balancer. http://www.limelight.com/traffic-load-balancer/.

[59] Hongqiang Harry Liu, David Gelernter, Srikanth Kandula, Ratul Mahajan, and Ming Zhang. Traffic Engineering with Forward Fault Correction. In *Proceedings of the ACM SIGCOMM 2014 Conference*.

[60] Hongqiang Harry Liu, Ye Wang, Yang Richard Yang, Hao Wang, and Chen Tian. Optimizing Cost and Performance for Content Multihoming. In *Proceedings of the ACM SIGCOMM 2012 Conference*.

[61] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zUpdate: Updating Data Center Networks with Zero Loss. In *Proceedings of the ACM SIGCOMM 2013 Conference*.

[62] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the Data Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*.

[63] Athina Markopoulou, Gianluca Iannaccone, Supratik Bhattacharyya, Chen-Nee Chuah, Yashar Ganjali, and Christophe Diot. Characterization of Failures in an Operational IP Backbone Network. *IEEE/ACM Transactions Networking*, 2008.

[64] MetaCDN. http://www.metacdn.com/.

[65] B. Niven-Jenkins, F. Le Faucheur, and N. Bitar. Content distribution network interconnection (CDNI) problem statement. `http://datatracker.ietf.org/doc/draft-ietf-cdni-problem-statement/`, January 2012.

[66] OnePica. `http://www.magentocommerce.com/magento-connect/One+Pica/extension/1279/one-pica-image-cdn`.

[67] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud Control with Distributed Rate Limiting. In *Proceedings of the ACM SIGCOMM 2007 Conference*.

[68] S. Raza, Yuanbo Zhu, and Chen-Nee Chuah. Graceful Network State Migrations. *Networking, IEEE/ACM Transactions on*, 2011.

[69] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for Network Update. In *Proceedings of the ACM SIGCOMM 2012 Conference*.

[70] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *ACM CCS'09*.

[71] Ali Reza Sharafat, Saurav Das, Guru Parulkar, and Nick McKeown. MPLS-TE and MPLS VPNS with Openflow. In *Proceedings of the ACM SIGCOMM 2011 Conference*.

[72] Nora H. Sleumer. Output-sensitive cell enumeration in hyperplane arrangements. *Nordic J. of Computing*, 6:137–147, June 1999.

[73] Martin Suchara, Dahai Xu, Robert Doverspike, David Johnson, and Jennifer Rexford. Network Architecture for Joint Failure Recovery and Traffic Engineering. In *SIGMETRICS'11*.

[74] Daniel Turner, Kirill Levchenko, Alex C. Snoeren, and Stefan Savage. California Fault Lines: Understanding the Causes and Impact of Network Failures. In *Proceedings of the ACM SIGCOMM 2010 Conference*.

[75] Balajee Vamanan, Jahangir Hasan, and T. N. Vijaykumar. Deadline-aware Datacenter TCP (D2TCP). In *Proceedings of the ACM SIGCOMM 2012 Conference*.

[76] Laurent Vanbever, Stefano Vissicchio, Cristel Pelsser, Pierre Francois, and Olivier Bonaventure. Seamless Network-Wide IGP Migrations. In *Proceedings of the ACM SIGCOMM 2011 Conference*.

[77] Hao Wang, Haiyong Xie, Lili Qiu, Yang Richard Yang, Yin Zhang, and Albert Greenberg. COPE: Traffic Engineering in Dynamic Networks. In *Proceedings of the ACM SIGCOMM 2006 Conference*.

[78] Ye Wang, Hao Wang, Ajay Mahimkar, Richard Alimi, Yin Zhang, Lili Qiu, and Yang Richard Yang. R3: Resilient Routing Reconfiguration. In *Proceedings of the ACM SIGCOMM 2010 Conference*.

[79] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*.

[80] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. NetPilot: Automating Datacenter Network Failure Mitigation. In *Proceedings of the ACM SIGCOMM 2012 Conference*.

[81] Dahai Xu, Mung Chiang, and Jennifer Rexford. Link-state Routing with Hop-by-hop Forwarding Can Achieve Optimal Traffic Engineering. *IEEE/ACM Transactions on Networking*, 2011.

[82] Hao Yin, Xuening Liu, Feng Qiu, Ning Xia, Chuang Lin, Hui Zhang, Vyas Sekar, and Geyong Min. Inside the bird's nest: measurements of large-scale live VoD from the 2008 olympics. In *ACM IMC'09*.

[83] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. Detail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2012 Conference*.