# zUpdate: Updating Data Center Networks with Zero Loss

Hongqiang Harry Liu
Yale University
hongqiang.liu@yale.edu

Xin Wu
Duke University
xinwu@cs.duke.edu

Ming Zhang
Microsoft
mzh@microsoft.com

Lihua Yuan
Microsoft
lyuan@microsoft.com

Roger Wattenhofer
Microsoft
wattenhofer@ethz.ch

David A. Maltz
Microsoft
dmaltz@microsoft.com

## ABSTRACT

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.

**Categories and Subject Descriptors:** C.2.1 [**Computer Communication Networks**]: Network Architecture and Design–Network communications. C.2.3 [**Computer Communication Networks**]: Network Operations.

**Keywords:** Data Center Network, Congestion, Network Update.

## 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.* [17] 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). 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 paper 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 con-


*SIGCOMM'13,* August 12–16, 2013, Hong Kong, China.

straints 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 [19], 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 [3] switches and Floodlight controller [1].
- 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.

## 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 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 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. 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 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 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.

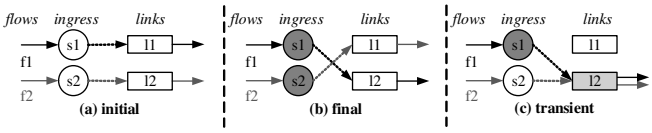



**Figure 1: Transient load increase during traffic migration.**

Such network-wide traffic migration, if not done properly, could lead to severe congestion. 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 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 [17].

A modern DCN hosts many interactive applications such as search and advertisement which require very low latencies. Prior research [5] reported that even small losses and queuing delays could dramatically elevate the flow completion time and impair the user-perceived

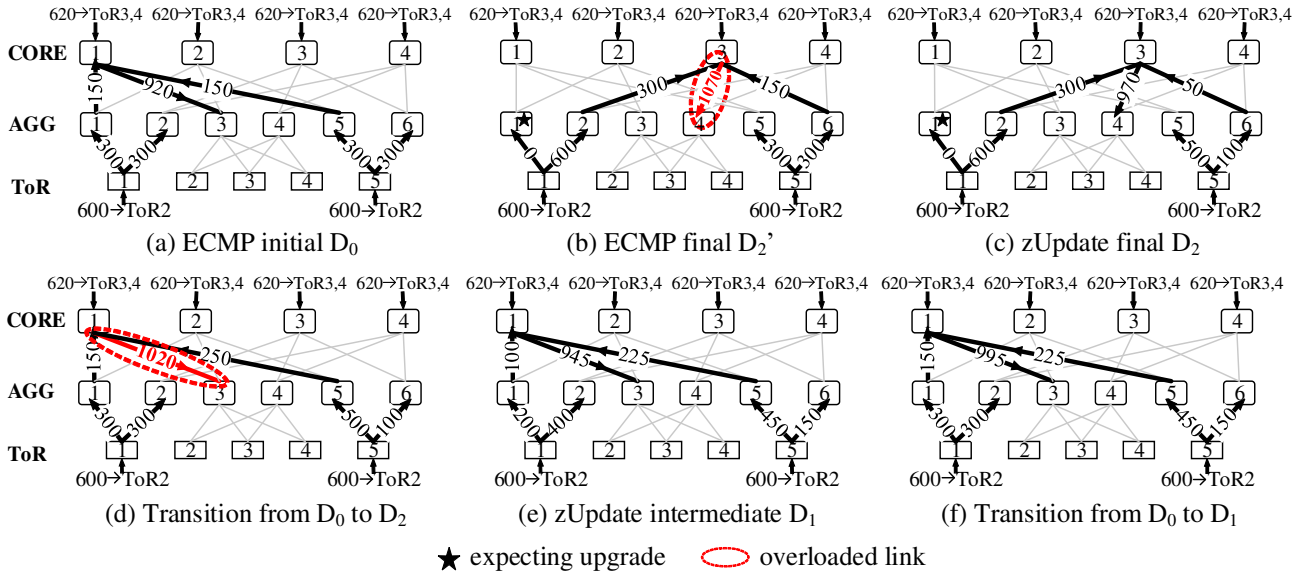



**Figure 2: This example shows how to perform a lossless firmware upgrade through careful traffic distribution transitions.**

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. 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 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.

# 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 2(a) shows a FatTree [4] 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 indicates 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 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 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 2(a)) to $D_2$ (Figure 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 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 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 2(a)) to $D_2$ (Figure 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 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 2(a)) to $D_1$, no matter in what order $ToR_1$ and $ToR_5$ are changed, there will be no congestion. Figure 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

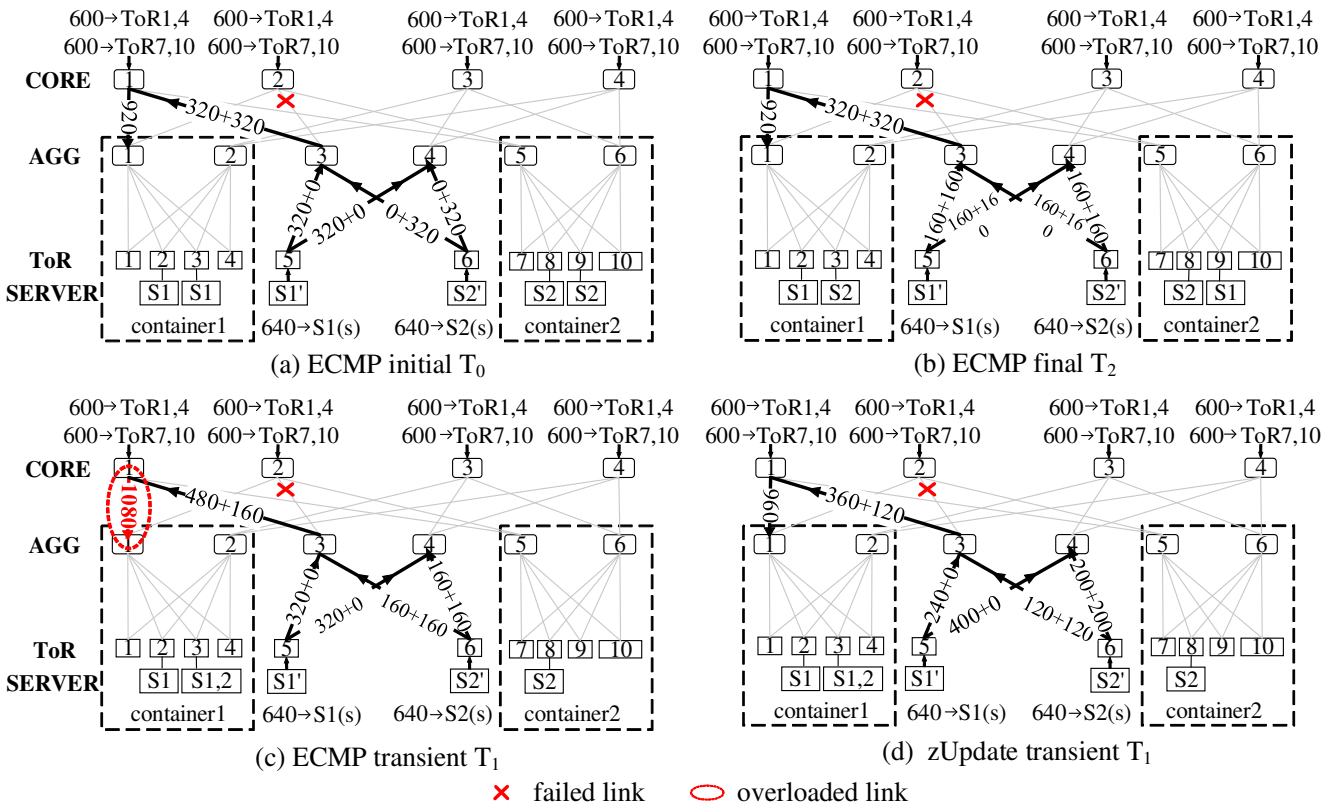



**Figure 3: This example shows how to avoid congestion by choosing the proper traffic split ratios for switches.**

if $ToR_1$ is changed first. Similarly, the transition from $D_1$ (Figure 2(e)) to $D_2$ (Figure 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 2(c)) that satisfies the firmware upgrade requirement (moving traffic away from $AGG_1$) as well as the appropriate $D_1$ (Figure 2(e)) that bridges the lossless transitions from $D_0$ (Figure 2(a)) to $D_2$. We will explain how `zUpdate` computes the intermediate and final traffic distributions in §5.3.

**Load balancer reconfiguration:** Figure 3(a) shows another Fat-Tree network where the link capacity remains 1000. One of the links $CORE_2 \leftrightarrow AGG_3$ 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, $ToR_5 \rightarrow AGG_3$, 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(a) shows the traffic distribution under the initial traffic matrix $T_0$ where the load on the busiest link $CORE_1 \rightarrow AGG_1$ is 920, which is the sum of 600 (the demand $CORE_1 \rightarrow ToR_{1/4}$) and 320 (the traffic on link $AGG_3 \rightarrow CORE_1$ 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(b) shows the traffic distribution under the final traffic matrix $T_2$ after the update. Note that the traffic on link $ToR_5 \rightarrow AGG_3$ 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(c) where $S_2$' LB is reconfigured before $S_1$'s. This causes link $ToR_6 \rightarrow AGG_3$ to carry "160+160" traffic, half of which goes to the $S_2$ in $Contain_1$, and further causes congestion on link $CORE_1 \rightarrow AGG_1$. 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 there will be no congestion under the initial, final or any possible transient traffic matrices during the update. Figure 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 $CORE_1 \rightarrow AGG_1$ 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(a)), the final $T_2$ (Figure 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 §6.2.

**The `zUpdate` process:** We provide `zUpdate`($T_0$, $D_0$, $C$) 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

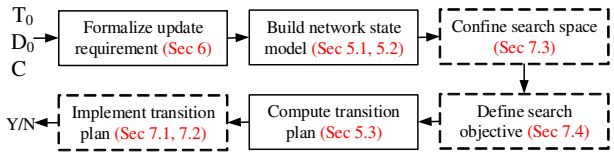



**Figure 4: The high-level working process of `zUpdate`.**

$\mathcal{C}$. $D_n$ would then allow an update, e.g., upgrading switch or reconfiguring LB, to be executed without incurring any loss.

Figure 4 shows the overall workflow of `zUpdate`. In the following, we will first present a network model for describing lossless traffic distribution transition (§5.1, 5.2) and an algorithm for computing a lossless transition plan (§5.3). We will then explain how to represent the constraints in each update scenario (§6). After that, we will show how to implement the transition plan on switches with limited table size (§7.1, 7.2), reduce computational complexity by confining search space (§7.3), and reduce transition overhead by picking a proper search objective function (§7.4).

# 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.

| | |
|---|---|
| $V$ | 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 $D := \{l^f_{v,u} \mid \forall f, e_{v,u} \in E\}$. |
| $r^f_{v,u}$ | A rule for $f$ on $e_{v,u}$: $r^f_{v,u} = l^f_{v,u}/T_f$. |
| $R$ | All rules in network: $R := \{r^f_{v,u} \mid \forall f, e_{v,u} \in E\}$. |
| $R^f_v$ | Rules for $f$ on switch $v$: $\{r^f_{v,u} \mid \forall u : e_{v,u} \in E\}$. |
| $R^f$ | Rules for $f$ in the network: $\{r^f_{v,u} \mid \forall e_{v,u} \in E\}$. |
| $\mathcal{D}(T)$ | All *feasible* traffic distributions which fully deliver $T$. |
| $\mathcal{D}_{\mathcal{C}}(T)$ | All traffic distributions in $\mathcal{D}(T)$ which satisfy constraints $\mathcal{C}$. |
| $\mathcal{P}(T)$ | $\mathcal{P}(T) := \{(D_1, D_2) \mid \forall D_1, D_2 \in \mathcal{D}(T)$, direct transition $D_1$ to $D_2$ is lossless.$\}$ |

**Table 2: The key notations of the network model.**

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

$$\forall f : \sum_{u \in V} l^f_{s_f,u} = \sum_{v \in V} l^f_{v,d_f} = T_f \quad (1)$$

$$\forall f, v \in V \setminus \{s_f, d_f\} : \sum_{u \in V} l^f_{v,u} = \sum_{u \in V} l^f_{u,v} \quad (2)$$

$$\forall f, e_{v,u} \notin G_f : l^f_{v,u} = 0 \quad (3)$$

$$\forall e_{v,u} \in E : \sum_{\forall f} l^f_{v,u} \le c_{v,u} \quad (4)$$

Equations (1) and (2) guarantee that all the traffic is fully delivered, (3) means a link should not carry $f$'s traffic if it is not on the paths from $s_f$ to $d_f$, and (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 §8.

## 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 §4.

We explain this problem in more detail using an example in Figure 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 (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 [17] 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 $R^{f,2}$.

To see how two-phase commit helps solve the state explosion problem, we observe that the subgraph $G_f$ of a flow $f$ (Figure 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. Sup-

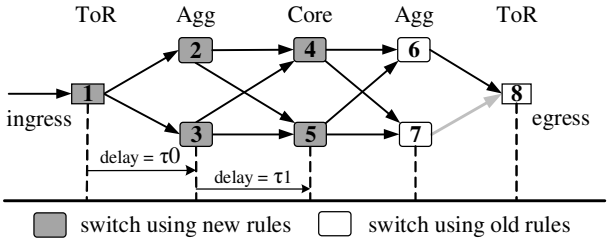



**Figure 5: Two-phase commit simplifies link load calculations.**

pose 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 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 1. *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. See appendix. □

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 (4) will be $2^n$ where $n$ is the number of flows. To further reduce the complexity in checking (4), we introduce the following:

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

$$\forall e_{v,u} \in E : \sum_{\forall f} max\ \{l^{f,1}_{v,u}, l^{f,2}_{v,u}\} \le c_{v,u} \tag{5}$$

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

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

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

Hence, (5) holds.

$\Leftarrow$: When (5) holds, we have:

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

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

Lemma 2 means we only need to check Equation (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, (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 (5) under traffic matrix $T$.

## 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 (1)$\sim$(5). We defer the discussion of constraints $\mathcal{C}$ in §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 2) and a lossless transition can be performed in one step.

**Algorithm 1**: `zUpdate`($T_0$, $D_0$, $\mathcal{C}$)

```
1  // D_0 is the initial traffic distribution
2  // If D_0 satisfies the constraints C, return D_0 directly
3  if D_0 ∈ D_C(T_0) then
4      return [D_0] ;
5  // The initial # of steps is 1, N is the max # of steps
6  n ← 1 ;
7  while n ≤ N do
8      M ← new optimization model ;
9      D[0] ← D_0 ;
10     for k ← 1, 2, ..., n do
11         D[k] ← new traffic distribution variable ;
12         M.addVariable(D[k]) ;
13         // D[k] should be feasible under T_0
           M.addConstraint(D[k] ∈ D(T_0)) ;
14     for k ← 1, 2, ..., n do
15         // Transition D[k-1] → D[k] is lossless ;
16         M.addConstraint((D[k-1], D[k]) ∈ P(T_0)) ;
17     // D[n] should satisfy the constraints C
       M.addConstraint(D[n] ∈ D_C(T_0)) ;
18     // An objective is optional
19     M.addObjective(objective) ;
20     if M.solve() = Successful then
21         return D[1 → n] ;
22     n ← n + 1 ;
23 return [] // no solution is found ;
```

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) $D_n \in \mathcal{D_C}(T_0)$; (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 1 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 §7.4.

# 6. HANDLING UPDATE SCENARIOS

In this section, we apply zUpdate to various update scenarios listed in Table 1. Specifically, we will explain how to formulate the requirements of each scenario as zUpdate's input constraints $\mathcal{C}$.

## 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 $D^*$:

$$\forall f, u \in U, e_{v,u} \in E : l^{f,*}_{v,u} = 0 \tag{6}$$

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 $D^*$:

$$\forall f \in \mathcal{F}_{test}, e_{v,u} \notin G_f(u_0) : l^{f,*}_{v,u} = 0 \tag{7}$$

$$\forall f \notin \mathcal{F}_{test}, e_{v,u_0} \in E : l^{f,*}_{v,u_0} = 0 \tag{8}$$

where (7) forces all the test flows to only use the paths through $u_0$, while (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 $D^*$:

$$\forall f, v \in V, e_{v,u_1}, e_{v,u_2} \in G_f : l^{f,*}_{v,u_1} = l^{f,*}_{v,u_2} \tag{9}$$

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

## 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 §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 3. *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:

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

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

Lemma 3 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 $D^*$:

$$D^* = T_0 \times R^* \tag{10}$$

$$T_{max} \times R^* \in \mathcal{D}(T_{max}) \tag{11}$$

Here $T_{max}$ is specified by the applications owners who are going to perform the update. In essence, lemma 3 enables the operators to migrate multiple VMs in parallel, saving the overall migration time, while not incurring any congestion.

# 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.

## 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 [17], the version number used by two-phase commit can be encoded in the VLAN tag.

Figure 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.

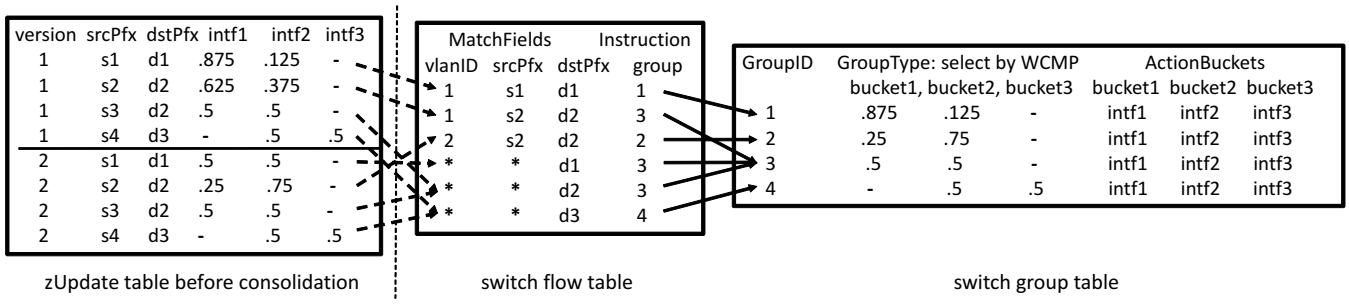



**Figure 6: Implementing `zUpdate` on an OpenFlow switch.**

## 7.2 Limited Flow and Group Table Size

As described in §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 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 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 $\delta_v^f = \sum_u |l_{v,u}^f - \bar{l}_{v,u}^f|$ 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 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 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.

## 7.3 Reducing Computational Complexity

In §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 (§6.1), the final traffic distribution $D^*$ must satisfy Equations (6)∼(8), each of which is in the form of $l_{v,u}^{f,*} = 0$. To meet each constraint $l_{v,u}^{f,*} = 0$, 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 (5)); ii) the congested links under $D^*$ after the transition is done (violating Equation (4)).

For traffic matrix update (§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 $D_{max} = T_{max} \times R_{ecmp}$.

## 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 $\mathcal{M}$:

$$\sum_{i=1}^{n} \sum_{f,v,u,w} |l_{v,u}^{f,i} - l_{v,w}^{f,i}|, \text{where } e_{v,u}, e_{v,w} \in E \qquad (12)$$

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(12) is its linearity. In fact, because Equations(1) $\sim$ (12) are all linear, $\mathcal{M}$ becomes a linear programming (LP) problem which can be solved efficiently.

## 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 §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 $\eta$ ($\eta > 1$). Lemma 3 guarantees that the transitions are lossless so long as the real traffic matrix $T \leq \eta T_{max}$.

# 8. IMPLEMENTATION

Figure 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 (§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 (§5, 7.3 & 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 (§7.1 & 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 [2] as the linear programming solver. The transition plan translator is written in 1500+ lines of Python code. We use Floodlight 0.9 [1] as the OpenFlow controller and commodity switches which support OpenFlow 1.0 [3]. Given that WCMP 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 $\xi$ in $f$, we first compute the next hop $u$ of $\xi$ according to WCMP hashing and then insert a rule for $\xi$ with $u$ as the next hop into $v$.

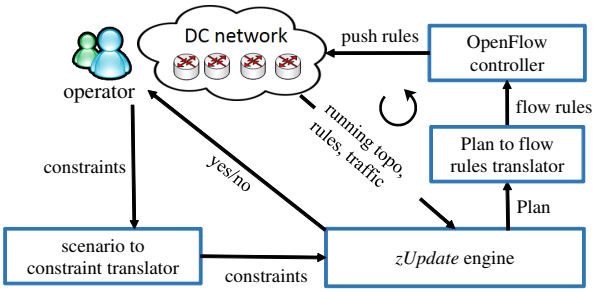



**Figure 7: `zUpdate`'s prototype implementation.**

# 9. EVALUATIONS

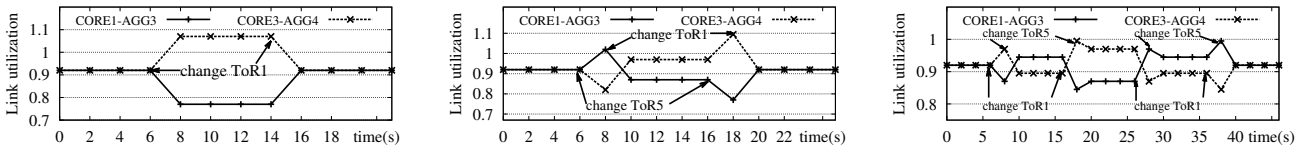
In this section, we show `zUpdate` can effectively perform congestion-free traffic migration using both testbed experiments and large-scale simulations. Compared to alternative traffic migration approaches, `zUpdate` can not only prevent loss but also reduce the transition time and transition overhead.

## 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. (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.

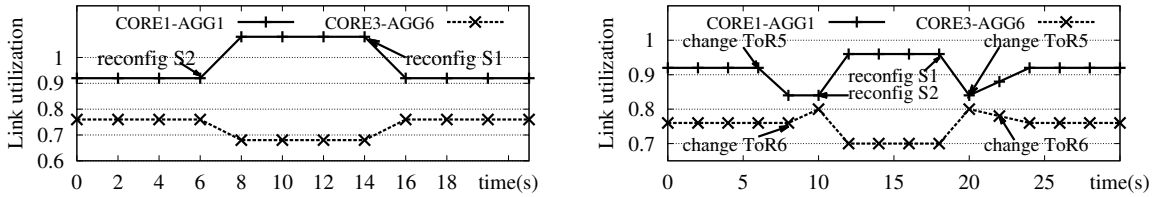
**Large-scale simulations**: Our simulations are based on a production DCN with hundreds of switches and tens of thousands of servers. The flow and group table sizes are 750 and 1,000 entries respectively, matching the numbers of the commodity switches used in the DCN. To obtain the traffic matrices, 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 §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




(a) ECMP-O (b) zUpdate-O (c) zUpdate

**Figure 8: The link utilization of the two busiest links in the switch upgrade example.**




(a) ECMP (b) zUpdate

**Figure 9: The link utilization of the two busiest links in LB reconfiguration example.**

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%. (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.

**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.

## 9.2 Testbed Experiments

We now conduct testbed experiments to reproduce the two traffic migration examples described in §4.

**Switch upgrade**: Figure 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 2). Figure 8(a) shows the transition process from Figure 2a (0s $\sim$ 6s) to Figure 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 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 8(b) shows the transition process from Figure 2a (0s $\sim$ 6s) to Figure 2d (6s $\sim$ 8s) to Figure 2c (8s $\sim$ 16s) under `zUpdate`-O. At 6s $\sim$ 8s, $\text{ToR}_5$ and $\text{ToR}_1$ are changed asynchronously, leading to a transient congestion on $\text{CORE}_1 \rightarrow \text{AGG}_3$ (Figure 2d). After $\text{ToR}_{1,5}$ are changed, the upgrading of $\text{AGG}_1$ is congestion-free at 8s $\sim$ 16s (Figure 2c). Once the upgrading of $\text{AGG}_1$ completes at 16s, the network is restored back to ECMP. Again because of the asynchronous changes to $\text{ToR}_5$ and $\text{ToR}_1$, another transient congestion happens on $\text{CORE}_3 \rightarrow \text{AGG}_4$ at 16s $\sim$ 18s.

Figure 8(c) shows the transition process from Figure 2a (0s $\sim$ 6s) to Figure 2e (8s $\sim$ 16s) to Figure 2c (18s $\sim$ 26s) under `zUpdate`. Due to the introduction of an intermediate traffic distribution between 8s $\sim$ 16s (Figure 2e), the transition process is lossless despite of asynchronous switch changes at 6s $\sim$ 8s and 16s $\sim$ 18s.

**LB reconfiguration**: Figure 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). Figure 9(a) shows the migration process from Figure 3a (0s $\sim$ 6s) to Figure 3c (6s $\sim$ 14s) to Figure 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 3c). After both LB's are reconfigured at 14s, the network is congestion-free (Figure 3b).

Figure 9(b) shows the migration process from Figure 3a (0s $\sim$ 6s) to Figure 3d (10s $\sim$ 18s) to Figure 3b (after 22s) under `zUpdate`. By changing the traffic split ratio on $\text{ToR}_5$ and $\text{ToR}_6$ 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 $\text{ToR}_5$ and $\text{ToR}_6$ 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.

## 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 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

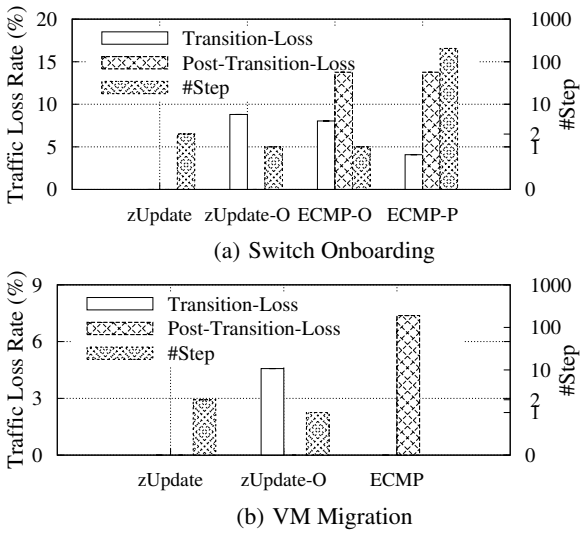



**Figure 10: Comparison of different migration approaches.**

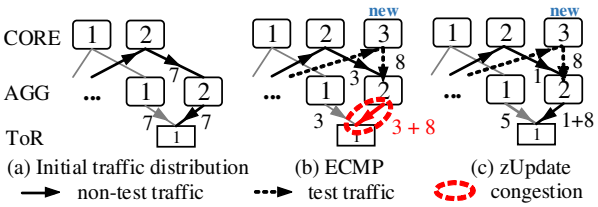



**Figure 11: Why congestion occurs in switch onboarding.**

figure, our flow consolidation heuristic (§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 11. Suppose there are 7 ingress-to-egress flows to $\text{ToR}_1$, each of which is 2Gbps, and the link capacity is 10Gbps. Figure 11(a) shows the initial traffic distribution under ECMP where each downward link to $\text{ToR}_1$ carries 7Gbps traffic. In Figure 11(b), 4 out of the 7 flows are selected as the test flows and are moved to the new $\text{CORE}_3$. 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 11(c) 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 [6, 13]. The total traffic rate of the tunnels is 6Gbps.

Figure 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 (§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 12, we omit the loss of of `zUpdate` and the post-transition loss of `zUpdate`-O, since all of them are 0.

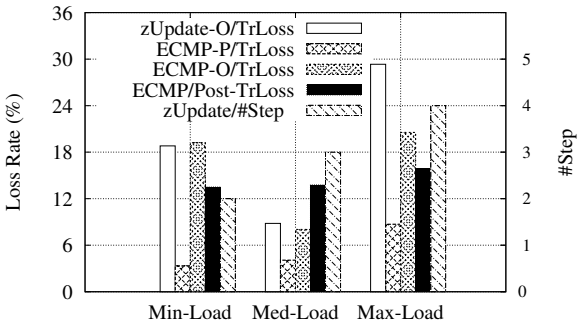



**Figure 12: Comparison under different traffic loads.**

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 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 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 §7.3 and §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: 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 12, where it takes 2.5 minutes to compute a 4-step transition plan. This is because of the heuristic in §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.

## 10. RELATED WORK

**Congestion during update:** Several recent papers focus on preventing congestion during a specific type of update. Raza *et al.* [16] study the problem of how to schedule link weight changes during IGP migrations. Ghorbani *et al.* [9] 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.* [18] and Francois *et al.* [8] seek to guarantee no forwarding loop during IGP migrations and link metric reconfigurations. Consensus routing [10] 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.* [7] built a tool to detect errors in BGP configurations. Header Space Analysis (HSA) [12] and Anteater [15] can check a few useful network invariants, such as reachability and no loop, in the forwarding plane. Built on the earlier work, VeriFlow [14] and realtime HSA [11] have been developed to check network invariants on-the-fly.

## 11. CONCLUSION

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.

## 12. ACKNOWLEDGMENTS

We thank our shepherd, Nick Feamster, and anonymous reviewers for their valuable feedback on improving this paper.

## 13. REFERENCES

[1] Floodlight. `http://floodlight.openflowhub.org/`.
[2] MOSEK. `http://mosek.com/`.
[3] OpenFlow 1.0. `http://www.openflow.org/documents/openflow-spec-v1.0.0.pdf`.
[4] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM'08*.
[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP DCTCP. In *SIGCOMM'10*.
[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *NSDI'05*.
[7] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *NSDI'05*.
[8] 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.
[9] S. Ghorbani and M. Caesar. Walk the Line: Consistent Network Updates with Bandwidth Guarantees. In *HotSDN'12*.
[10] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani. Consensus Routing: the Internet as a Distributed System. In *NSDI'08*.
[11] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI'13*.
[12] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *NSDI'12*.
[13] E. Keller, S. Ghorbani, M. Caesar, and J. Rexford. Live Migration of an Entire Network (and its hosts). In *HotNets'12*.
[14] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying Network-Wide Invariants in Real Time. In *HotSDN'12*.
[15] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. In *SIGCOMM'11*.
[16] S. Raza, Y. Zhu, and C.-N. Chuah. Graceful Network State Migrations. *Networking, IEEE/ACM Transactions on*, 2011.
[17] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM'12*.
[18] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure. Seamless Network-Wide IGP Migrations. In *SIGCOMM'11*.
[19] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang. NetPilot: Automating Datacenter Network Failure Mitigation. In *SIGCOMM'12*.

## APPENDIX

**Proof of Lemma 1**: 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.