# Consistency, Feasibility, and Optimality of Network Update in SDNs

Yang Chen, *Student Member, IEEE,* Huanyang Zheng, *Student Member, IEEE,* and Jie Wu, *Fellow, IEEE*

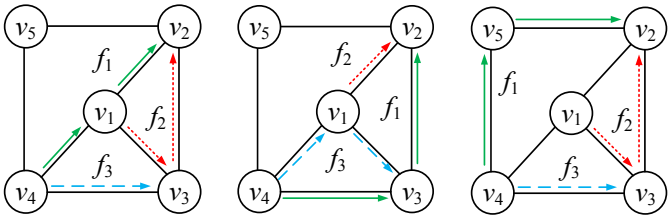
**Abstract**—In software defined networking, to maximize the network utilization, its control plane needs to frequently update the data plane via flow migration as the network conditions change dynamically. Since each switch updates its flow table independently and asynchronously, the network state transition may result in serious link congestion and packet loss if it is done directly from the initial to the final stage. Deadlocks among flows and links may also block the update process. In this paper, we pay close attention to elaborately resolving deadlocks with the help of spare paths during the network update. We prove that the feasibility of the consistent flow migration can be determined in exponential time. Furthermore, we demonstrate that even if there are multiple consistent migration plans, finding the optimal one that occupies the least leisure bandwidth resources is NP-hard. For the case in which no consistent plan is found, we introduce an efficient method to rate limit flows in order to reduce the packet loss. Extensive simulations show that our solution achieves a much smaller traffic loss rate at the cost of affordable spare link resource usage compared to prior methods.

**Index Terms**—Software Defined Networks (SDNs), spare link, consistent flow migration, feasibility and optimality.

## 1 Introduction

Configurations of Software Defined Networks (SDNs) are routinely updated in order to achieve shorter transmission latency and better bandwidth utilization [1–3]. SDN utilizes its centralized controller to configure and execute new update policies [4]. With the rise and development of SDNs [5], the requirements of high performance networks are becoming more and more intense. For example, regarding the packet loss rate, data centers usually claim it to be around 2% [6], while the requirements of wide area networks (WANs) and carrier-grade networks are much higher [7]. Specifically, the carrier-grade performance is often associated with the term "five nines" that means an availability of 99.999%. However, complex network updates are becoming increasingly common. For example, Microsoft's SWAN [8] and Google's B4 [9] run updates every few minutes, hundreds of times per day. Key challenges come from the fact that some unexpected events during the update may happen. These unexpected events will disrupt network functionality and cause traffic congestion as well as packet loss, resulting in a Quality of Service (QoS) disqualification. These events consist of unpredicted, long switch update times and abnormal communication delays between the controller and the switch. There are multiple reasons for those chaotic situations, such as imperfect clock synchronization and transient controller-data plane disconnection. In this paper, we study the consistent flow migration problem, which requires moving network flows from their initial routing paths to the target ones in a lossless way [10].

To prevent the above anomalies, we explore three properties in network flow migration: *consistency*, *feasibility*, and *optimality*. Consistency requires that the bandwidth demands of all flows be satisfied during the whole migration process and there be no congestion and packet loss during the update procedure. Because of the demanding packet loss rate, consistency is the most important property of flow migration. However, due to insufficient bandwidth resources, a consistent flow migration may not always exist. Therefore, we use *feasibility to refer to whether or not a consistent flow migration exists*. Optimality is raised when there are multiple consistent flow migration. This means finding the plan that takes the least spare resource to finish the update. This paper explores the feasibility and optimality of flow migrations under the consistency constraint in SDN updates.




(a) Initial path. (b) Final path. (c) Flow migration.

Fig. 1: An example to illustrate migration.

Fig. 1 illustrates the flow migration problem with three flows, $f_1, f_2$ and $f_3$. In each graph, the nodes represent the switches, and the edges are the links, all of which are bidirectional. Each directional link has a capacity of 1 Gbps and each flow has a bandwidth of 1 Gbps. Figs. 1(a) and 1(b) show the initial and final routing paths of the three flows. Unfortunately, none of them can be directly migrated to their final paths, because the initial routing path of each flow overlaps the final routing paths of the other two flows. Such a deadlock is challenging in terms of flow migrations. This paper uses *spare resources* to achieve consistent flow migration. A feasible flow migration is shown in Fig. 1(c) in which $f_1$ is migrated to an intermediate routing path on

• *Y. Chen, H. Zheng, and J. Wu are with Center for Networked Computing, Temple University, USA.*
*E-mail: {yang.chen, huanyang.zheng, jiewu}@temple.edu*

two spare links ($v_4$ to $v_5$ and $v_5$ to $v_2$). Such an intermediate routing path of $f_1$ releases the necessary bandwidth resource for $f_2$ and $f_3$, and thus, it breaks the deadlock. Afterwards, $f_2$ can be migrated to its final path and then to $f_3$, finally $f_1$ moves to its final path. The flow migration is completed without any packet loss.

Due to the complexity of the assistant intermediate routing paths, our problem becomes challenging. We study the consistent flow migration problem, which requires that the bandwidth demands of all flows be satisfied during the entire migration process. Moreover, we want to find the optimal one when several consistent update plans exist. Existing works focus on the resource dependency between the initial and final routing paths of flows. They aim at finding a specific migration order of flows [11, 12] or using a two-phase tagging scheme [13, 14] to complete a lossless flow migration to the final state. However, when the remaining bandwidth resources in the flows' final paths are insufficient, , these methods cannot make any progress except in reducing flow rates. Consequently, such flow rate limitations will lead to packet loss as well as QoS deterioration [15]. In contrast, our paper proposes a generic approach to consistently migrating flows with the help of spare paths. An algorithm is proposed to determine the feasibility of consistent flow migration. Furthermore, to efficiently resolve deadlocks, we demonstrate that even if there are multiple consistent flow migrations, it is NP-hard to find the optimal one that occupies the fewest bandwidth resources. Therefore, a greedy algorithm is proposed to consistently migrate flows in a sequence within a reasonably competitive ratio. If no consistent plan is found, we also conduct an efficient method to rate limit flows in order to reduce the packet loss.

Our main contributions are summarized as follows:

- We formulate the network update problem as a mixed integer programming and introduce a new definition of Resource Dependency Graph.
- We design a spare-path-assisted algorithm to determine the feasibility of a consistent flow migration and prove its polynomial time complexity for the case of unit size flows, unit size edges, and constant-size spare path lengths.
- We prove that it is NP-hard to find the optimal solution that occupies the least bandwidth resources. An approximation algorithm with a feasible RDG as an input is then proposed with a detailed time complexity analysis.
- If no consistent solution can be found, we propose a heuristic rate-limiting-flow approach to resolve deadlocks.
- We conduct extensive real data-driven experiments to demonstrate the significant advantages of our approach with regard to update time.

The remainder of this paper is organized as follows: Section 2 surveys related works. Section 3 describes the model and formulates the problem. Section 4 discusses the feasibility and optimality of flow migrations in SDNs as well as an efficient method of rate-limiting flows. Section 5 includes the experiments. Finally, Section 6 concludes the paper and shows potential extensions.

## 2 Related Work

There are two basic mainstream methods for flow migration implementation: ordering [12, 16–19] and two-phase [13, 20, 21]. The ordering strategy usually updates the forwarding table of the switches one-by-one in a specified order, which is carefully calculated in order to preserve some required properties, like being loop-free and blackhole-free. It does not introduce an additional update overhead. However, this order might not exist when it needs to guarantee both forwarding and policy demands. The two-phase scheme installs both the initial and final rules on all switches and tags packets with a rule's version number. This method is simple and fast. It ensures the success of the update, but it doubles the number of rules on every switch, which wastes expensive and power-hungry Ternary Content Addressable Memory (TCAM) memory resources. This paper performs the two-phase commit using version numbers for flow migrations.

The major drawback of the basic ordering and two-phase methods of flow migration is that they cannot guarantee consistency as congestion may exist during flow migrations. To obtain consistent flow migrations, we see that there are mainly three kinds of approaches: link capacity reservation [8], intermediate state-involvement [22], and time-awareness update [23–25]. As a typical link capacity reservation approach, SWAN [8] has two main results. First, if a fraction of the capacity is guaranteed to be free on each link of a flow path, SWAN can update the network in constant steps. Second, in order to solve the problem efficiently, linear programming is used to check whether a solution with bounded steps exists. However, when there is no slack on some edges, it is unlikely that this algorithm will halt in certain steps, which will lead to high computation complexity. A representative of the intermediate state-involvement approach, ZUpdate [22], attempts to compute and execute a sequence of steps to migrate flows in a congestion-free way. However, it stretches the update time, which makes the chaos of traffic migration last longer. A typical time-awareness update approach, Dionysus [26], dynamically schedules the process based on the runtime differences of switches' update speeds, instead of a previous static ordering of rule updates. It is a path-based update method, meaning that a flow is scheduled to be migrated to its final path when all the links along its new path have enough bandwidth resources. If a single link along the path is not available, this method can only wait and waste a lot of link resources. Another kind of time-awareness plan is based on time synchronization technology. The timed consistent strategy [24, 25] utilizes time-triggered network updates to achieve consistency. However, this scheme asks too much of time synchronization. Even with a straggling switch, the whole following process is likely to be in total disorder.

Many prior works also strive to find a congestion-free update scheme with the property that there will be no congestion independent of the update order. However, most of the congestion-free update plans require part of the link capacity to be left vacant, which will decrease utilization of the expensive network infrastructure. Moreover, a majority of them always involve solving a series of linear program-

mings (LPs), which is slow and does not scale well. We are aware that there is little network update research on deadlock tackling even with the high-demanding low packet loss rate. Dionysus [26] mentions rate-limiting random number of flows until all the deadlocks are resolved. When a link has not enough remaining bandwidth for several flows to update at the same time, Dionysus utilizes the migration completion time as a default order of flow priority. Yet, this kind of opportunistic scheduling is likely to cause deadlocks where no progress can be made. MCUP [27] proposes a migration approach to minimize the transient congestion during the update procedure when there does not exist a congestion-free update order, or specifically, when there are deadlocks among update-awaiting flows. [28] solves the migrating problem by dynamically finding flow paths with a dependency graph. [29] proposes a innovative dynamic programming method to migrate flows.

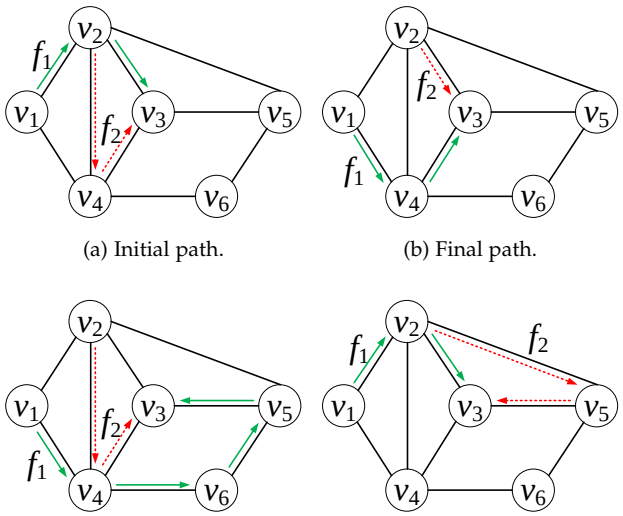
## 3 Framework

### 3.1 Motivation

Networking updates in SDNs consist of hardware upgrades, deployment modifications, and configuration changes. It is necessary to routinely update networks for better performances in failure recovery, transmission latency, and bandwidth utilization. However, this planned maintenance always puts the paths of network flows in a state of flux. Flow migration is an important kind of configuration change. It is a common source of instability in SDNs, leading to update deadlocks, broken connectivity, forwarding loops, and access control violations.

We notice that almost all the current flow migration strategies neglect to discuss deadlocks' situations or randomly select flows to stub out until all deadlocks are resolved, which will cause severe packet losses and QoS deterioration. In fact, deadlocks may frequently occur even when the initial and final traffic states are both congestion-free and valid, because when the network does traffic engineering to reallocate flow routes, the intertwined extent of different flows' initial and final paths is not taken into consideration. Furthermore, inappropriate scheduling order of flows can also lead to deadlocks [26]. Due to the high demand for packet loss rate, it is essential to migrate flows with the best effort to preserve the consistency even when there are deadlocks. Our key observation is that intermediate state involvement in the form of spare paths can vacate link resources of flows' initial paths in order to break the deadlocks. With the help of two-phase update commit, modifying the flow path means adding a new flow entry to each switch along the new path. Moreover, considering the limited and expensive Ternary Content-Addressable Memory (TCAM) in the switch routing tables, it is better to migrate flow one-by-one, which will not cause too much redundancy to the network. In addition, by migrating only one flow each time step, we can also control the network with less temporary disruption and congestion.

Fig. 2 shows a toy example with two flows, $f_1$ and $f_2$. In each graph, the nodes represent the switches, and the edges are the links, all of which are bidirectional. Each directional link has a capacity of 1 Gbps and each flow has a bandwidth of 1 Gbps. Figs. 2(a) and 2(b) show the initial and final routing paths of $f_1$ and $f_2$. Unfortunately, neither $f_1$ nor $f_2$ can be directly migrated to their final paths, because the initial routing path of $f_1$ overlaps with the final routing path of $f_2$, and vice versa. Such a deadlock is challenging in terms of flow migrations. This paper uses *spare resources* to achieve consistent flow migrations. A feasible flow migration is shown in Fig. 2(c) in which $f_1$ is migrated to an intermediate routing path on three spare links ($v_4$ to $v_6$, $v_6$ to $v_5$, and $v_5$ to $v_3$). Such an intermediate routing path of $f_1$ releases necessary bandwidth resource for $f_2$, and thus, it breaks the deadlock. However, the flow migration in Fig. 2(c) may not be the optimal one, as shown in Fig. 2(d), since only two spare links can sufficiently fulfill flow migrations. Instead of migrating $f_1$ to an intermediate routing path on three spare links, we can migrate $f_2$ to an intermediate routing path on two spare links ($v_2$ to $v_5$, and $v_5$ to $v_3$). Such a migration breaks the deadlock and uses less spare resources than the migration in Fig. 2(c), which is more desirable among the feasible consistent migrating plans.



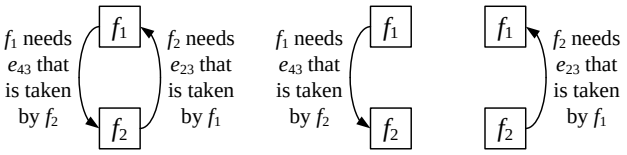

(a) Initial path. (b) Final path. (c) Flow migration. (d) Another case.

Fig. 2: A motivating example.

### 3.2 Model and Formulation

Our flow migration scenario is based on a directed network, $G = (V, E)$, where $V$ is a set of vertices (i.e., switches), and $E \subseteq V^2$ is a set of directed edges (i.e., links). We use $v_i$ to denote the $i$-th vertex and use $e_{ij}$ to denote the edge from $v_i$ to $v_j$. Each edge is capacitated, and we use $c_{ij}$ to denote the bandwidth capacity of $e_{ij}$. The network, $G$, includes a set, $F$, of given flows. We use $f_k$ to denote the $k$-th flow, and its bandwidth demand is $d_k$. The initial and final routing paths of $f_k$ are denoted by $p_k$ and $p_k^*$, respectively. A path is an ordered set of edges. For example, in Fig. 2(a), we have $p_1 = \{e_{12}, e_{23}\}$ and $p_2 = \{e_{24}, e_{43}\}$.

This paper studies consistent flow migrations, which require that the bandwidth demands of all flows be satisfied during the entire migration process. A round-by-round manner is used to migrate flows from their initial to final routing paths. In each round, only one flow is migrated to another path. Let $p_k^r$ denote the routing path of $f_k$ at




(a) Deadlock in Fig. 2(a). (b) Flow in Fig. 2(c). (c) Flow in Fig. 2(d).

Fig. 3: RDG, deadlock, and spare path.

the round $r$. Let $b_{ij}^r$ denote the bandwidth usage of $e_{ij}$ during round $r$, which is equal to the total bandwidth demands of its passing flows. We have $R$ rounds in total, i.e., $0 \leq r \leq R$. Our problem is similar to the Klotski game and is formulated as:

$$\text{minimize} \quad \sum_{e_{ij} \in E} [\max_{0 \leq r \leq R} b_{ij}^r] \tag{1}$$

$$\text{s.t.} \quad b_{ij}^r \leq c_{ij} \qquad \forall\, 0 \leq r \leq R, e_{ij} \in E \tag{2}$$

$$\left|\{ f_k \,|\, p_k^r \neq p_k^{r+1}\}\right| = 1 \qquad \forall\, 0 \leq r \leq R \tag{3}$$

$$p_k^0 = p_k \text{ and } p_k^R = p_k^* \qquad \forall\, f_k \in F \tag{4}$$

In Eq. 1, the maximum bandwidth usage of $e_{ij}$ among all rounds is $\max_r b_{ij}^r$. The objective is to minimize the total maximum bandwidth usage among all edges during flow migrations. We aim to use the minimum spare bandwidth resources to migrate flows. Two constraints are involved. Eq. 2 means that the bandwidth usage of $e_{ij}$ is smaller than or equal to its capacity, $c_{ij}$. Eq. 3 means that only one flow is migrated in each round, in terms of changing its routing path. Here, $\{ f_k \,|\, p_k^r \neq p_k^{r+1}\}$ is the set of flows that change their routing paths. Meanwhile, $|\cdot|$ the denotes set cardinality. Eqs. 2 and 3 represent the consistency requirement during flow migrations. However, this requirement may not be always satisfied, leading to the feasibility problem. Finally, Eq. 4 requires each flow to migrate from its initial path to its final one.

### 3.3 Resource Dependency Graph

In this subsection, we introduce several important definitions. We start with the concept of resource dependency:

***Definition 1.*** Flow $f_k$ depends on a minimal set of other flows (denoted by $F_k$) if $f_k$ could be immediately migrated from its current path to its final path after the removal of $F_k$ but not after the removal of $F_k \setminus \{f\}$ for $\forall f \in F_k$.

$F_k$ may be an empty set, and $f_k$ may depend on different minimal sets. If we map flows to nodes and map dependency relationships to directed edges, a resource dependency graph can be obtained. An example graph is shown in Fig. 3(a), which corresponds to Fig. 2(a). It can be seen that $f_1$ and $f_2$ depend on each other in terms of their initial routing paths.

***Definition 2.*** Let each flow correspond to one of its minimal dependency sets. The Resource Dependency Graph (RDG) is defined by mapping flows to nodes and their dependencies sets to directed edges. Given an RDG, a closed walk without repeated nodes is defined as a deadlock.

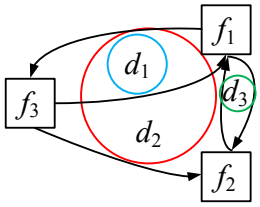



Fig. 4: Intertwined deadlocks in 1.

Definition 2 reveals resource deadlocks in flow migrations. Let $l_i$ denote the $i$-th deadlock and let $L$ denote the set of deadlocks. Note that multiple RDGs can be obtained for a given network since a flow may depend on different minimal flow sets. Additionally, deadlocks can be intertwined with each other, meaning that a flow takes part in more than one deadlock. Our first example in Fig. 1 shows the case. There is only one RDG with three deadlocks, $d_1$, $d_2$, and $d_3$, among $f_1, f_2$ and $f_3$, shown in Fig. 4. Because of the edge from $f_1$ to $f_3$ exists in both deadlocks, $d_1$ and $d_2$ are intertwined. Similarly, because of the edges from $f_2$ to $f_1$ exist in both deadlocks, $d_2$ and $d_3$ are intertwined. These intertwined deadlocks make the flow migration more challenging.

Once the RDG is determined, we have:

***Theorem 1.*** If the RDG does not include a deadlock (i.e., $L = \emptyset$), a feasible solution could use the topological order in the RDG to migrate flows. Each flow is immediately migrated from its initial to final paths [26].

The major challenge of our problem is to resolve deadlocks while migrating flows. As previously mentioned, our problem aims to use minimum spare bandwidth resources to break deadlocks by migrating flows. As shown in Figs. 2(c) and 3(b), three spare links ($e_{46}$, $e_{65}$, and $e_{53}$) are used to break the dependency from $f_2$ to $f_1$. As shown in Figs. 2(c) and 3(c), two spare links ($e_{25}$ and $e_{53}$) are used to break the dependency from $f_1$ to $f_2$. These spare bandwidth resources are formally defined as spare paths with respect to deadlocks:

***Definition 3.*** The spare path is defined for a given flow $f$. It is a path that (i) has enough bandwidth to hold $f$, (ii) has the same source and destination as $f$, and (iii) is at least one edge different from the initial and final paths of $f$.

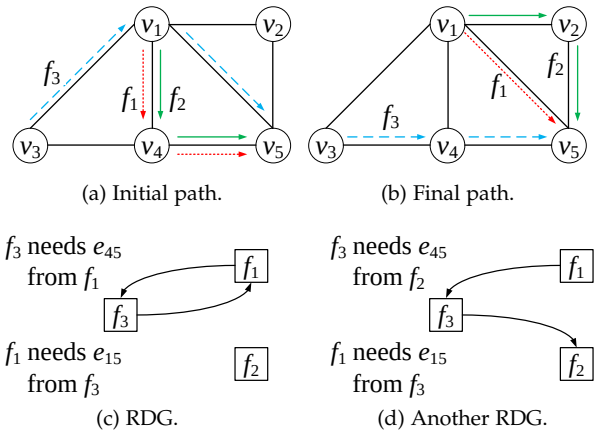
***Definition 4.*** Spare path collection is a set of spare paths for flows in a given deadlock in an RDG. Once flows in the deadlock are migrated to the corresponding spare paths, then (i) all remaining flows in this deadlock can be migrated to their final paths following the dependency order and (ii) all flows in the spare paths can be migrated back.

Note that a deadlock might have multiple spare collections. For example, the deadlock in Fig. 3(b) includes two spare path collections: one collection includes one path of $\{e_{46}, e_{65}, e_{53}\}$, and another collection includes one path of $\{e_{25}, e_{53}\}$. The key idea of the spare path collection is that they can resolve a deadlock, without considering intersections among different deadlocks. Let $S_i$ denote the set of spare path collections for deadlock $l_i$. To reduce the complexity, we limit the size of spare path collections: The

**Algorithm 1** Feasibility Determination

**Input:** Network $G$ and flow set $F$;
**Output:** Feasible solution existence;

1: **for** each flow $f_k \in F$ **do**
2: Compute its minimal dependency sets, $F_k$.
3: Generate RDGs based on flow dependency.
4: **for** each RDG of network, $G$ **do**
5: Find the set of deadlocks, $L$, in this RDG.
6: **for** each deadlock $l_i \in L$ **do**
7: Find the set of spare path collections, $S_i$, for $l_i$.
8: **if** $S_i = \emptyset$ **then**
9: **continue** to the next RDG;
10: **return** a feasible solution exists;
11: **return** a feasible solution may not exist;




(a) Initial path. (b) Final path. (c) RDG. (d) Another RDG.

Fig. 5: An example to illustrate Algorithm 1.

hop length of a spare path is no more than $H$ and the cardinality of a spare path collection is no more than $C$. Meanwhile, $H$ and $C$ are pre-determined parameters. Given an RDG, all spare path collections for each deadlock can be determined through an exhaustive search. Shorter spare paths are preferred over longer spare paths as they use less total bandwidth resources. Note that $H$ and $C$ bring a trade-off between accuracy and time complexity. Larger $H$ and $C$ can explore more possible spare path collections but run the risk of an exponentially larger time complexity.

This paper does not consider spare paths with hop lengths larger than H. This is because the number of paths grows exponentially with respect to the number of nodes. Given an RDG, all spare paths for each deadlock can be determined through exhaustively searching. Shorter spare paths are preferred over longer spare paths as they use fewer total bandwidth resources. Note that H brings a tradeoff between accuracy and time complexity. A larger H can explore more possible spare paths but may cause a larger time complexity.

## 4 Feasibility and Optimality

This section shows that the feasibility of the consistent flow migrations can be determined with the assistance of spare path collections. However, even if multiple consistent flow migrations exist, searching for the optimal solution that occupies the least bandwidth resources is NP-hard.

### 4.1 Feasibility

This subsection studies the feasibility problem [14], that is, whether a consistent flow migration exists or not. The idea is to use spare paths to break deadlocks. As shown in Theorem 1, if the RDG does not include a deadlock, a feasible solution could use the topological order to migrate flows. Algorithm 1 is proposed to determine feasibility. For a given network, lines 1 to 3 construct the RDGs. Note that different RDGs can be obtained for the same network and flows since a flow might depend on different minimal flow sets. In lines 4 to 10, Algorithm 1 checks each possible RDG. If there is an RDG in which all deadlocks can be solved by spare path collections, Algorithm 1 returns that a consistent flow migration exists in line 10. If deadlocks cannot be broken in all RDGs, Algorithm 1 returns infeasibility in line 11.

Fig. 5 shows an example for Algorithm 1. Figs. 5(a) and 5(b) show the initial and final flow-routing paths, respectively. Links $e_{14}$, $e_{34}$, and $e_{45}$ have bandwidth capacities of 2 Gbps, while all other links have bandwidth capacities of 1 Gbps. Each flow has a bandwidth of 1 Gbps. In this scenario, $f_1$ depends on $f_3$, and $f_2$ does not depend on other flows. $f_3$ can depend on either $f_1$ or $f_2$, leading to two different RDGs as shown in Figs. 5(c) and 5(d). $f_3$ needs $e_{45}$ to be migrated to its final routing path. Meanwhile, $e_{45}$ can be released for $f_3$ by either $f_1$ (in Fig. 5(c)) or $f_2$ (in Fig. 5(d)). Algorithm 1 will traverse each RDG in lines 4 to 10. Let us start with Fig. 5(c), which includes only one deadlock among $f_1$ and $f_3$. This deadlock includes two spare path collections: one has a spare path of $\{e_{12}, e_{25}\}$ to migrate $f_1$ and another one has a spare path of $\{e_{31}, e_{12}, e_{25}\}$ to migrate $f_3$. Therefore, Algorithm 1 terminates.

Algorithm 1 correctly determines the feasibility. If there is an RDG in which all deadlocks have spare path collections, Algorithm 1 returns that a consistent flow migration exists. This is simply because, by definition, each deadlock can be broken by one of its spare path collections. The corresponding flows can be migrated to their spare paths and can be migrated back after breaking the deadlock. Algorithm 1 is a conservative approach: a feasible solution can exist but Algorithm 1 cannot identify its existence.

For a given network, the number of RDGs may be exponential with respect to the number of flows. This is because each flow may depend on different minimal flow sets. An example is shown in Fig. 5, in which $f_3$ can depend on either $f_1$ or $f_2$. However, the time complexity of Algorithm 1 can be reduced to a polynomial through capping the cardinalities of spare path collections. The number of RDGs is $O(|F|^C)$ and the number of spare paths is $O(|E|^H)$. In a special given network, if the bandwidth capacity of each edge is one unit and the bandwidth demand of each flow is one unit, the time complexity of Algorithm 1 is $O(H \times |E|^{H+1})$. When the bandwidth capacity of each edge is a unit and the bandwidth demand of each flow is also a unit, each flow corresponds to exactly one minimal set in terms of dependency relations. This is because each edge

can accommodate at most one edge. Therefore, only one RDG exists for the given network. For the same reason, the number of deadlocks belongs to $O(|E|)$. It takes $O(|H||E|^H)$ to find all spare paths for all deadlocks. As a result, the time complexity of Algorithm 1 is $O(H|E|^{H+1})$ in this special situation, which is acceptable even in the worst case.

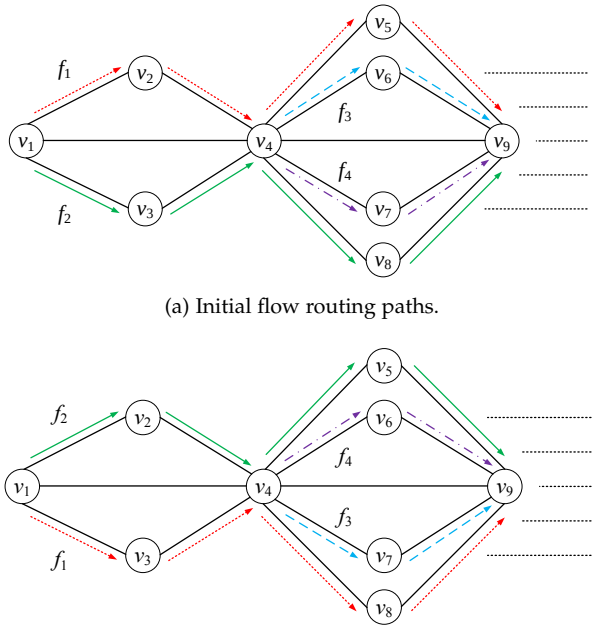
### 4.2 Optimality

The previous subsection discussed the problem feasibility, or whether a consistent flow migration exists or not. However, even if multiple consistent flow migrations exist, it is NP-hard to find the optimal one that occupies the least bandwidth resources. We start with the problem hardness.

***Theorem 2.*** Our flow migration problem is NP-hard.

*Proof*: [14] has shown the NP-hardness of the flow migration problem. Here we prove this through a different reduction of the set cover problem [30]. Given some elements and a collection of sets of elements, the set cover problem aims to select minimum sets to cover all given elements. We reduce sets and elements to sparse paths and deadlocks in RDGs. Let us consider a network $G$ in which the bandwidth capacity of each edge is one unit and the bandwidth demand of each flow is one unit. $G$ is constructed as an independent sequence of squares with one diagonal, and each flow is on one side of the sequence of squares. An example is shown in Fig. 6. Fig. 6(a) and Fig. 6(b) show the initial and final routing paths of flows on three squares: flows $f_1$ and $f_2$ are on squares $v_1v_2v_3v_4$ and $v_4v_5v_8v_9$ respectively, and flows $f_3$ and $f_4$ is on square $v_4v_6v_7v_9$. Each element in the set cover problem is mapped to a pair of flows in $G$, which is a deadlock (e.g., deadlock between $f_1$ and $f_2$ and deadlock between $f_3$ and $f_4$ in Fig. 6). Each set in the set cover problem is mapped to the spare path in $G$ (e.g., paths $\{e_{14}, e_{49}\}$ and $\{e_{49}\}$ in Fig. 6). At this time, the set cover problem is reduced to our problem, which selects minimum spare resources to break all deadlocks. Since the set cover problem is NP-complete and reduced to our problem in polynomial time, our problem is NP-hard. ■

Since our problem is NP-hard, Algorithm 2 is proposed as an approximation algorithm. In line 1, Algorithm 2 initializes the set of spare path collections to be $S = \emptyset$. In lines 2 and 3, it determines all deadlocks in the RDG and their corresponding sets of spare path collections. Lines 4 to 6 include a greedy selection until all deadlocks are broken. In line 5, a spare path collection, $s$, is selected from the unselected spare paths, $\cup_i S_i \backslash S$. We use $|\{\, l_i \,|\, s \in S_i\}|$ to denote the number of deadlocks that can be broken by $s$. On the other hand, we use $\sum_{e_{ij}\in S}[\max_r b_{ij}^r]$ to denote the total spare resources used by $S$. Therefore, $\sum_{e_{ij}\in S\cup\{s\}}[\max_r b_{ij}^r] - \sum_{e_{ij}\in S}[\max_r b_{ij}^r]$ is the marginal gain of spare resources after adding $s$ into $S$. Note that spare path collections in $S$ may overlap with each other. $\frac{|\{\, l_i \,|\, s\in S_i\}|}{\sum_{e_{ij}\in S\cup\{s\}}[\max_r b_{ij}^r]-\sum_{e_{ij}\in S}[\max_r b_{ij}^r]}$ represents the benefit-to-cost ratio of spare path collection $s$, in which the benefit is the number of broken deadlocks, and the cost is the marginal gain of spare resources. Line 6 updates the deadlocks to be $L = L\backslash\{\, l_i \,|\, s \in S_i\}$, i.e., it removes deadlocks broken by $s$. Lines 5 and 6 are iterated until all deadlocks are broken. Line 7 returns the result.




(a) Initial flow routing paths.

(b) Final flow routing paths.

Fig. 6: An example to explain NP-hardness.

**Algorithm 2** Spare Path Computation

**Input:** A feasible RDG for network $G$ and flow set $F$;
**Output:** The set of spare paths if feasible solutions exist;

1: Initialize the set of spare path collections, $S = \emptyset$.
2: **for** each deadlock, $l_i \in L$, in RDG **do**
3: Find the set of spare path collections, $S_i$, for $l_i$.
4: **while** $L \neq \emptyset$ **do**
5: Set $s = \max \frac{|\{\, l_i \,|\, s\in S_i\}|}{\sum_{e_{ij}\in S\cup\{s\}}[\max_r b_{ij}^r]-\sum_{e_{ij}\in S}[\max_r b_{ij}^r]}$ from $\cup_i S_i \backslash S$, and add $s$ into $S$.
6: Update deadlocks to be $L = L\backslash\{\, l_i \,|\, s \in S_i\}$.
7: **return** $S$ as the set of spare path collections;

Use the Fig. 1 as an example. We can find a spare path collection for $d_1$, i.e. $e_{45}, e_{52}$. $d_2$ has the same spare path collection. We do not need to choose the collections. Then we move $f_1$ to its spare path $e_{45}, e_{52}$ in the collection. Then the edge $e_{12}$ is freed, and $f_2$ can be migrated to its final path in Fig. 1(b). After that, both $e_{41}$ and $e_{13}$ are released and $f_3$ is able to be updated. Finally, we move $f_1$ to its final path and the flow migration finishes. Next, we analyze the provable efficient performance of Algorithm 2. We have:

***Theorem 3.*** Algorithm 2 achieves an approximation ratio of $O(H \cdot C \cdot \ln |L|)$ for the optimal algorithm.

*Proof*: The proof is done through an intermediate problem, which is defined as follows: (i) we map each spare path collection to a set, and map each deadlock to an element; (ii) an element is included in a set if its deadlock can be broken by the spare path collection; (iii) the intermediate problem is the traditional set cover problem that selects the minimum sets to cover all elements [31]. The cost of a set is the total spare resources of its spare path collection, i.e., $\sum_{e_{ij}\in s}[\max_r b_{ij}^r]$.

**Algorithm 3** Rate Limit Flows

**Input:** The RDG for network $G$ and flow set $F$;
**Output:** Migration plan;

1: Sort flows by the priorities;
2: **while** a feasible solution is not found **do**
3: **while** the deadlock $l \in L$ is not resolved **do**
4: Rate limit the flow with the highest priority inside the deadlock $l$;
5: Update the remaining capacities of links along the flow's initial path;
6: Migrate flows in topological order;
7: Apply Algorithm 1 to search for a feasible solution;
8: **return** Migration plan;

A greedy algorithm, which iteratively selects the set with the maximum ratio of marginal element coverage to set cost, has an approximation ratio of $O(\ln|L|)$ for the traditional set cover problem. $|L|$ is the number of deadlocks (i.e., elements). However, the set costs should be independent from each other in the traditional set cover problem. In contrast, spare paths can overlap with each other in our problem, meaning that:

$$\sum_{e_{ij}\in S\cup\{s\}}[\max_r b_{ij}^r]-\sum_{e_{ij}\in S}[\max_r b_{ij}^r]\leq\sum_{e_{ij}\in\{s\}}[\max_r b_{ij}^r] \quad (5)$$

The key observation is:

$$\sum_{e_{ij}\in S\cup\{s\}}[\max_r b_{ij}^r]-\sum_{e_{ij}\in S}[\max_r b_{ij}^r]\geq\frac{1}{HC}\sum_{e_{ij}\in\{s\}}[\max_r b_{ij}^r] \quad (6)$$

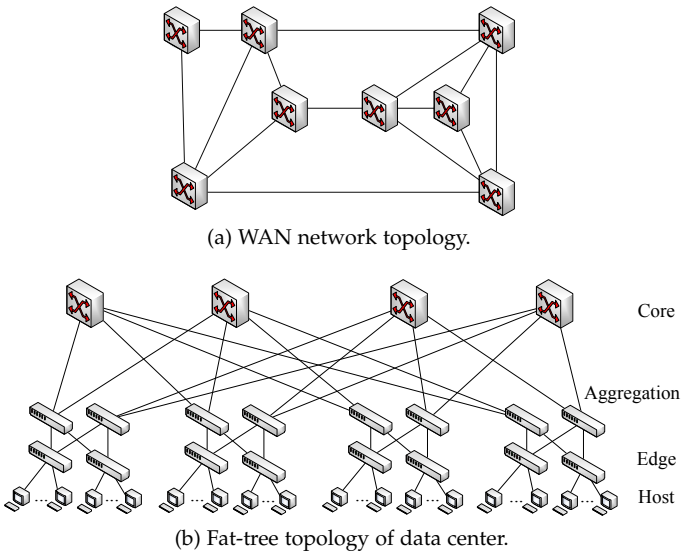
This is because the spare path collection, $s$, has at least $H\cdot C$ edges. Each spare path has at most $H$ edges and $s$ has at most $C$ spare paths. As a result, Eq. 6 shows that Algorithm 2 achieves a ratio of $O(H\cdot C\cdot\ln|L|)$. ■

The time complexity of the worst case is $O(|L|\times\sum_{i=1}^{C}\binom{|F|}{i}\times(|F|\times|E|^H)^i)$. It happens when we need to check all flows' spare path collections for $|L|$ times, each of which is for resolving one deadlock. For each spare path collection, we can have the number of spare paths ranging from 1 to $C$. Additionally, the complexity of using a spare path collection of $i$ paths is $\binom{|F|}{i}\times(|F|\times|E|^H)^i$. What's more, we can also improve the performance of our algorithm by reusing the spare paths. If one spare path is released by a flow, all links along the spare path will have an extra available bandwidth, which is equal to the traffic rate of the previous occupied flow.

### 4.3 Rate limit flows

Note that our above algorithms do not guarantee a consistent flow migration. This because a consistent flow migration may not exist or be found by Algorithm 1. In order to resolve the deadlocks, we primarily utilize the leisure link capacity resources as spare paths to release the links involved in deadlocks. If there are still unresolved deadlocks, flow migrations can be conducted through rate limiting flows. Rate-limit means to cut off the propagation of the flow and make its transmission rate zero. Randomly selecting flows to be rate limited can lead to severe packet loss. Moreover, in our previous work [32], we have proved that finding an update schedule with the minimum number of rate-limited flows with deadlocks is NP-hard in the absence of partial flow limiting. Thus, we propose a heuristic algorithm for efficiently rate-limiting flows. First, we define the priority of each flow: the number of nodes in its involved largest deadlock times the ratio between the numbers of its ingoing edges and outgoing edges. The priority of the flow demonstrates the intertwined extent of the flow as well as the popularity of its occupied link resources. Specifically, the number of nodes in its involved largest deadlock measures the complexity of a flow's involved deadlocks. The ratio measures the marginal gain of a flow's migration due to releasing and occupying different link resources in its initial and final paths.

In Algorithm 3, line 1 sorts the flows by their priorities. Lines 2-7 resolve deadlocks by rate-limiting the flows. Lines 3-6 are to resolve a single deadlock. Line 4 cuts off the flow with the highest priority. The bandwidth of edges are updated in line 5. We use the topological order in RDG to migrate other flows. Repeat this process until a feasible migration is found by applying Algorithm 1.



(a) WAN network topology.

(b) Fat-tree topology of data center.

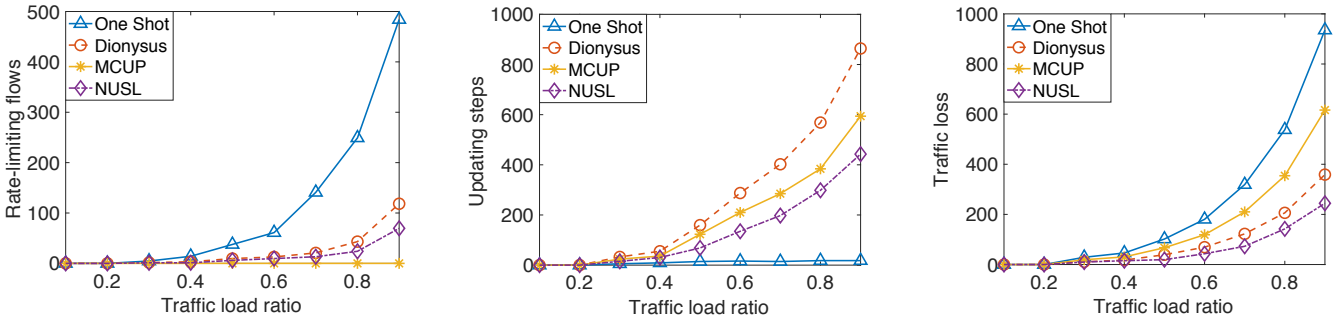
Fig. 7: Network topologies for simulations.

## 5 Experiment

Experiments are conducted to evaluate the performances of our proposed algorithms. After presenting the topologies and basic settings, the evaluation results are shown from different perspectives to provide insightful conclusions.

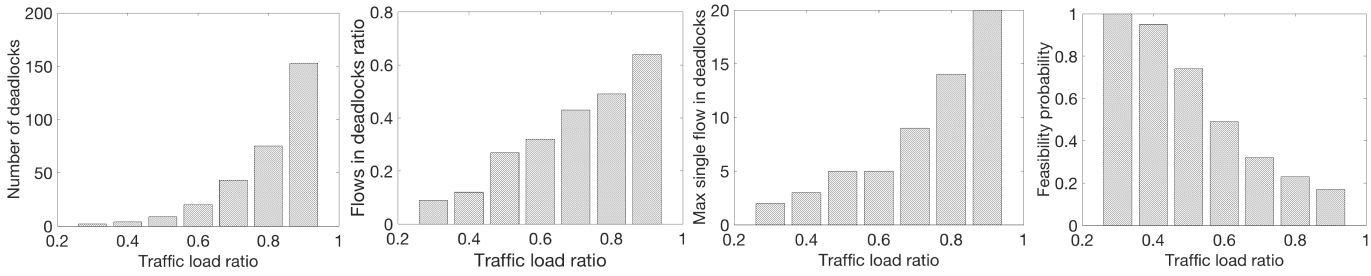
### 5.1 Experimental Settings

We conduct simulations in two real topologies. The first one is Microsoft's inter-data center WAN topology [26, 33], consisting of 8 switches that are connected as shown in Figure 7a. Each link is two-way and has a capacity of 1-Gbps. The second one is a fat-tree topology [34] for the data centers, shown in Figure 7b. There are 4 core switches, 8 aggregation switches, and 8 edge switches in this network. Each edge switch connects 8 hosts. Each switch has two 1-Gbps ports, resulting in a network with a full bisection




(a) Rate limiting flows. (b) Updating time. (c) Traffic loss.

Fig. 8: Performance in the WAN topology.




(a) The number of deadlocks. (b) Flows involved in deadlocks. (c) Max single flow in deadlocks. (d) Feasibility probability.

Fig. 9: WAN deadlock involvement.

TABLE 1: WAN topology

| Traffic load | 0.2 | 0.4 | 0.6 | 0.8 |
|---|---|---|---|---|
| Flow number | 729 | 1538 | 2387 | 3120 |

TABLE 2: Fat-tree topology

| Traffic load | 0.2 | 0.4 | 0.6 | 0.8 |
|---|---|---|---|---|
| Flow number | 101682 | 24637 | 36827 | 42372 |

bandwidth. We change the traffic load ratio from 10% to 90% to simulate independent variables. We list several flow numbers corresponding to different traffic load ratios in Table I and II.

Our proposed algorithms are denoted as Network Update through Spare Links (NUSL). There are two comparison algorithms in our simulations: One-shot, Dionysus [26] and MCUP [27]. One-shot updates the network directly from the initial to the final stage by cutting off all the current flows and allowing new ones in after the network is vacant. It causes severe packet loss and significantly reduces the QoS. The One-shot is not able to meet the demanding loss rate requirement in realistic networks even though it only takes one step to update the network. Dionysus has been specifically introduced in the related work part. It builds the dependency graph to describe the relationships among different flow states and migrates flows in a topological order. When there are deadlocks, it opportunistically rate-limits flows to zero in order to resolve deadlocks. MCUP uses a randomized rounding algorithm and improves the rounding result by greedily rerouting each flow in each stage.

We evaluate various aspects under two different topologies in Fig. 7(a) and Fig. 7(b). Our experiments study the relationships between the traffic load and three metrics:

1) the number of rate-limiting flows: when a consistent migration plan does not exist;
2) update steps: time from the first migration until all flows are migrated;
3) traffic loss: the total number of lost packets.

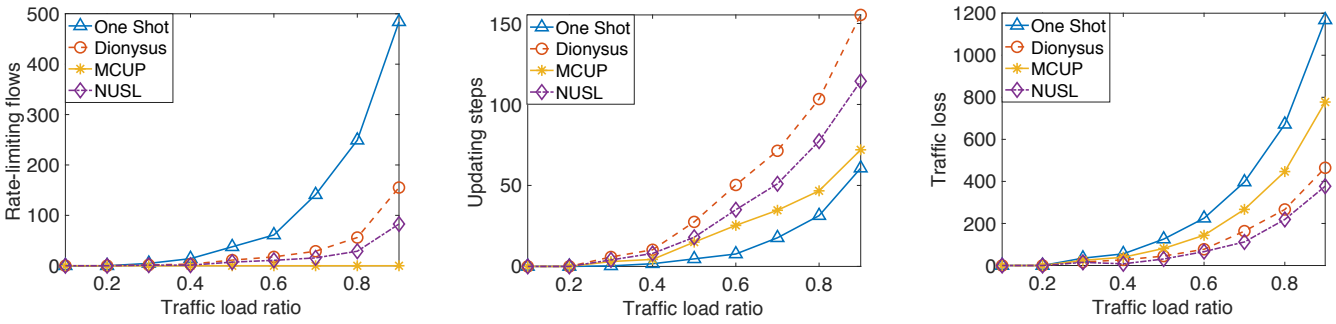
Metrics related to deadlocks are also evaluated:

1) the number of deadlocks;
2) the number of involved flows;
3) the maximum number of deadlocks that a single flow can become involved in;
4) the probability of finding a feasible update plan with spare paths.

Then, we test the spare bandwidth resource usage conditions including the ratio of flows with intermediate state, the spare resource cost as well as its ratio, and the average spare path length. Flows in the network are generated randomly at the granularity of 1Mbps. We assume the initial and final states of the network are all valid. There are no more new flows coming into the network during the update. No flow paths have any loops and each link load is within its capacity.

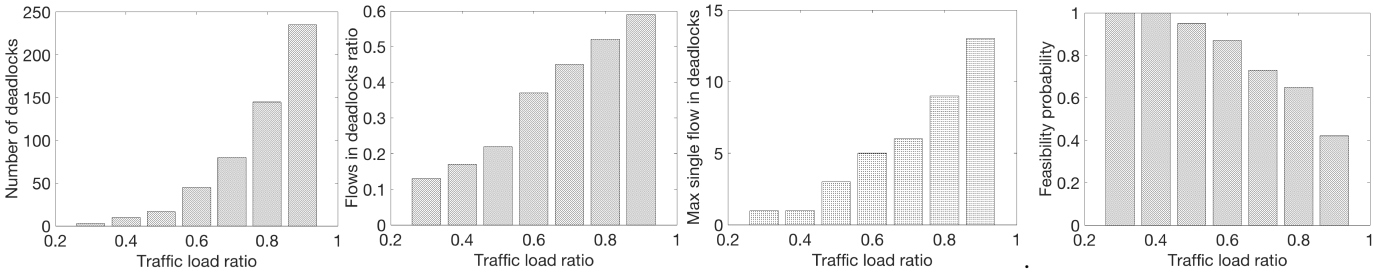
### 5.2 Evaluation Results for the WAN

The experiment results in the WAN topology are shown in Fig. 8, Fig. 9, and Tab. 3. Algorithm performances are compared with respect to different traffic load ratios. Fig.




(a) Rate limiting flows. (b) Updating time. (c) Traffic loss.

Fig. 10: Performance in the Fat-tree topology




(a) The number of deadlocks. (b) Flows involved in deadlocks. (c) Max single flow in deadlocks. (d) Feasibility probability.

Fig. 11: Fat-tree topology deadlock involvement

8 shows that our algorithm, NUSL, achieves a satisfying result compared to One-Shot and Dionysus. NUSL limits the fewest flows in Fig. 8(a) and maintains the highest throughput with the lowest traffic loss in Fig. 8(c). The number of rate limiting flows for MCUP is always zero because it does not allow to diminish flows' bandwidth. With a load ratio of 80%, NUSL limits only 47% of the flows in One-Shot and 78% in Dionysus since we have a more efficient Algorithm 3, Rate Limit Flows. NUSL does, however, have a longer update time . It takes about 33% more steps than Dionysus with a ratio of 70%. This is because NUSL introduces the intermediate state of the update flows. It utilizes the leisure bandwidth resources to reduce traffic loss during the update process by migrating some flows to their alternate paths. In this way, it vacates some competing link resources to break deadlocks among flows. At the same time, as shown in Fig. 8(b), it leads in extra update time. As long as there is no chaos during the update, it is acceptable to stretch the time when the controller operates flows one by one in an orderly manner. Dionysus performs just fine. It does not have a good strategy for breaking deadlocks, thus, it experiences more traffic loss than NUSL. Even in update time, Dionysus has no obvious advantages over our approach. One-Shot acts as an ideal baseline under both topologies in our simulation because it represents the most naive solution for updating the network without any strategy and does not have the limitations of the underneath network situations. When One-Shot is applied, there is no new traffic coming into the network during the update. In other words, all flows must be cut off to vacate the link resources they are occupying. As a result, One-Shot takes the fewest steps to accomplish the network update at the expense of the throughput. In Fig. 8(b), the traffic loss of MCUP is much larger than NUSL and Dionysus because it allows the traffic congestion within limits in order to update flows in less rounds.

Deadlock involvement situations with different metrics are described in Fig. 9. As shown in Fig. 9(a), more flows in the network can increase the traffic load, which makes more deadlocks emerge. For example, when the load ratio is 90%, the deadlocks are more than two times as much as when the ratio is 80%. The percentage of flows involved in deadlocks also increases quickly, as shown in Fig. 9(b). In Fig. 9(c), we use the maximum number of deadlocks in which a single flow can get involved, to measure the complexity of the resource dependency graph. The heavier the traffic is, the more deadlocks a single flow can become involved in. Fig. 9(d) shows the feasibility property of our scheme. From the results, we know that with the assistance of spare paths, it is highly possible to find a lossless update plan when the traffic is light. For example, when the load ratio is less than 50%, its probability is as high as 85%. From these four figures and the analysis, we can conclude that deadlocks are not negligible.

### 5.3 Experimental Results for the Fat-tree Topology

The experiment results for the Fat-tree topology in the data centers are shown in Fig. 10. The one-shot approach also acts as an ideal baseline in our simulation. From the figures, we can see that the basic tendency and relationships are almost the same as those of the WAN topology. We will focus on the differences between Fat-tree and WAN. It should be first noted that Fat-tree can hold more than five

times the flows WAN topology can under the same traffic load ratio. By contrast, the number of rate limiting flows and throughput maintenance with NUSL is better than in the WAN. This is due to Fat-tree's excellent load balancing property. The performance of our algorithms is better than the other two. In the update steps, the difference between NUSL and Dionysus is smaller, because the data center is able to achieve a relatively faster update as a result of its almost fixed path length. Additionally, MCUP has fewer updating steps than NUSL because all flows' in Fattree have similar path lengths and in almost the same pattern. In Fig. 10(c), there is actually little difference among these three approaches. NUSL less frequently uses spare paths under the Fat-tree topology than in WAN because this topology always distributes in a traffic more balanced way and we can update the network consistently without the help of spare paths when the traffic is light. Moreover, when the traffic is heavy, it is almost impossible to find spare paths under the balancing network. However, the ratio of the number of flows that need to be migrated to their backup paths is also much smaller, which indirectly proves the advantage of a regular topology. The ability to predict the performance in the data center is essential for providing a satisfying service.

Fig. 11 shows that there are comparatively fewer deadlocks in the data centers because of the regular topology and balanced traffic, which proves the sensibility of applying a regular topology in the data centers. The number of flows involved in the deadlocks is greater than WAN's 30%. This is because if one link is busy, it will affect a large number of flows that would compete for the bandwidth resources. The largest number of deadlocks a single flow gets involved in is smaller than the WAN topology, because it is harder to be intertwined with other flows that are routed in a tidy pattern. It is also more likely to find a feasible update plan with the Fat-tree topology. With the same load ratio, the probability is about 24% higher. Thus, NUSL is extremely suitable for the patterned topology. Fat-tree is better than WAN in terms of deadlocks.

### 5.4 Spare bandwidth resources cost comparison

We also study the spare link resource utilization condition under those two topologies in Tab. 3 and Fig. 12. For both of the topologies, it is intuitively easy to find spare resources within the light traffic network. As traffic becomes heavier, it becomes more and more difficult to find a spare path for a flow, and a helper path may not even exist. From Tab. 3, we can see that the possibility of finding a spare path decreases significantly by more than $\frac{3}{4}$ as the traffic load ratio increases from 30% to 90%. This explains the reason why NUSL needs to limit much more flows and suffer more severe losses under heavy traffic than light traffic. Moreover, in the spare resource cost rows of the table, we notice that the spare resource usage cost first increases then decreases. This is because we are not able to obtain enough spare paths in the congested network. There is little leisure bandwidth available for us to resolve all the deadlocks. Even if there is one, the cost of the path is so high that the loss outweighs the gain. As a result, it is necessary to limit the length of helper paths in order to control the cost and the achievement.

TABLE 3: Spare link resource usage

| Topology setting | Comparison metrics | The given traffic load | | | |
|---|---|---|---|---|---|
| | | $\varphi$=0.3 | $\varphi$=0.5 | $\varphi$=0.7 | $\varphi$=0.9 |
| WAN topology | Flow with int. state | 84% | 23% | 17% | 5% |
| | Spare resource cost | 9 | 37 | 87 | 52 |
| Fat-tree topology | Flow with int. state | 79% | 45% | 22% | 13% |
| | Spare resource cost | 11 | 21 | 40 | 31 |

We can attribute this phenomenon to the trade-off between spare resource usage and deadlock resolution possibilities.

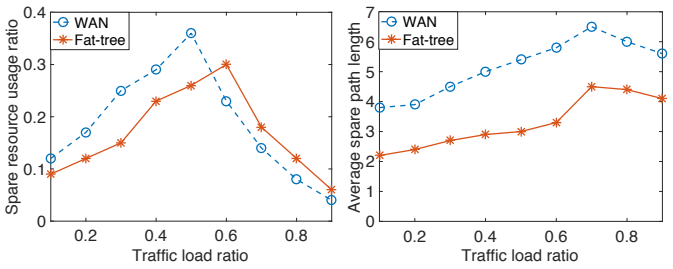
In Fig. 12(a), both ratios of the spare resource usage in WAN and Fat-tree topologies have a obvious tendency of increasing first and then decreasing. With a heavier traffic load, the spare resources become less. Additionally, it is more difficult to find an available spare path for a single flow though there are more flows with a larger traffic load. More interestingly, when the traffic is light with a ratio ranging from $0.1$ to $0.5$, WAN requires more spare resources while it uses less with a larger traffic load. It indicates the unbalanced traffic distribution inside the WAN topology. Fig. 12(b) shows the result of the average spare path length. Both lines have the tendency to increase first and then decrease, but the turning point around $0.7$ is later than in Fig. 12(a). It verifies that a heavier traffic load makes it difficult to find spare paths.

It is usually more likely that a spare path can be found in the Fat-tree infrastructure. Moreover, because all paths have fixed lengths in the Fat-tree, the cost of each flow with intermediate state $\rho$ is either 2 or 5. In data centers, most of the traffic is between different pods [34]. It costs less to use the spare resources, because the Fat-tree topology naturally limits the forwarding path length. The fixed path length also reduces the number of the available spare paths from an exponential to a polynomial. This implies that it is better for the data center to apply the spare-path assisted strategy.

In conclusion, there are two main reasons for the above results. First, the Fat-tree topology is more common than the WAN topology. Its switches can be divided into three kinds: core, aggregation, and edge. Aggregation and edge switches can serve as a pod, shown in the Fig. 7(b). All the forwarding paths between servers in different pods follow the same pattern: edge-aggregation-core-aggregation-core, whose length is 5. If the servers are in the same pod but have different edge switches, the pattern is edge-aggregation-edge. The forwarding paths between two servers that belong to the same edge switch pass through this switch. With such forwarding paths, the workload is able to be distributed more evenly, which reduces the possibility of deadlocks. Second, the Fat-tree topology has a hierarchical structure, but there is no such specific construction for the WAN topology. The bandwidths in Fat-tree vary from the rank where the links reside, and upper links with more s bandwidths. However, all links in the WAN have almost the same capacities. Unbalanced flow distribution is more likely to be blocked in WANs.

## 6 Conclusion and Future Work

This paper focuses on the network update problem in the setting of SDNs. Network administrators do not take




(a) Spare resource usage ratio. (b) Average spare path length.

Fig. 12: Spare resource comparison.

flow path overlapping information into consideration when they reallocate flow routes. Consequently, congestion will unavoidably happen resulting in deadlocks among flows and link resources, which will block the update process and cause severe packet loss. While current methods neglect deadlocks, we introduce an efficient approach to consistently update networks with the help of spare paths. We utilize a resource dependency graph to describe the relationships among different flow states and link resources. An algorithm is proposed to determine the feasibility of consistent flow migrations. We demonstrate that it is NP-hard to find the optimal scheme using the fewest spare link resources, even when there are several consistent update plans. An efficient algorithm, NUSL, is proposed to achieve a reasonably good competitive ratio. Our algorithms are evaluated in various aspects under different network scenarios. The evaluation results demonstrate the effectiveness and efficiency of our approach.

There are several interesting directions for future research. First, one interesting problem is to extend the network link capacities and flow bandwidth demands to multiple units, where inappropriate bandwidth allocation will cause more deadlocks. A second direction involves applying a finer granularity of flow migration such as link-based schemes. Specifically, the link-based scheduling approaches will reduce the possibility of deadlocks to a great extent as a result of better utilization of link bandwidths. However, the approaches should not induce a much higher computation complexity and should not require the excessive time synchronization accuracy. A third interesting research direction is to migrate flows in parallel, which will accelerate the process of network update.

## Acknowledgments

This research was supported in part by NSF grants CNS1629746, CNS 1564128, CNS 1449860, CNS 1461932, CNS 1460971, CNS 1439672, CNS 1301774, and ECCS 1231461.

## References

[1] R. Gallager, "A minimum delay routing algorithm using distributed computation," *IEEE Transactions on Communications*, vol. 25, no. 1, pp. 73–85, 1977.

[2] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," in *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM 2002.

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

[4] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "A distributed and robust sdn control plane for transactional network updates," in *INFOCOM 2015*.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Computer Communication Revie*, vol. 38, no. 2, 2008.

[6] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, "Packet-level telemetry in large datacenter networks," in *SIGCOMM 2015*.

[7] T. Mizrahi and Y. Moses, "Time4: Time for sdn," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 433–446, 2016.

[8] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in *SIGCOMM 2013*.

[9] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined wan," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, Aug. 2013.

[10] S. Raza, Y. Zhu, and C.-N. Chuah, "Graceful network state migrations," *IEEE/ACM Trans. Netw.*, vol. 19, no. 4, pp. 1097–1110, Aug. 2011.

[11] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li, and X. Chen, "Is every flow on the right track?: Inspect sdn forwarding with rulescope," in *INFOCOM 2016*.

[12] P. Francois and O. Bonaventure, "Avoiding transient loops during the convergence of link-state routing protocols," *IEEE/ACM Transactions on Networking*, vol. 15, no. 6, pp. 1280–1292, 2007.

[13] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *SIGCOMM 2012*.

[14] S. Brandt, K. T. Frster, and R. Wattenhofer, "On consistent migration of flows in sdns," in *INFOCOM 2016*.

[15] H. Zheng, W. Chang, and J. Wu, "Coverage and distinguishability requirements for traffic flow monitoring systems," in *IWQoS 2016*.

[16] A. Ludwig, M. Rost, D. Foucard, and S. Schmid, "Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies," in *HotNets 2014,*.

[17] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure, "Seamless network-wide igp migrations," in *SIGCOMM 2011*.

[18] J. McClurg, H. Hojjat, P. Černý, and N. Foster, "Efficient synthesis of network updates," in *ACM PLDI 2015*.

[19] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized "zero-queue" datacenter network," in *SIGCOMM 2014*.

[20] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent updates for software-defined networks: Change you can believe in!" in *HotNets 2011*.

[21] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *HotSDN 2013*.

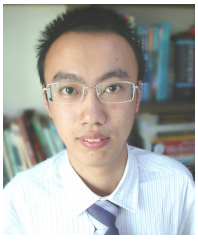
[22] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zupdate: Updating data center networks with zero loss," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 411–422, Aug. 2013.

[23] S. H. Tseng, C. L. Lim, N. Wu, and A. Tang, "Time-aware congestion-free routing reconfiguration," in *IFIP 2016*.

[24] T. Mizrahi, E. Saat, and Y. Moses, "Timed consistent network updates in software-defined networks," *IEEE/ACM*

*Transactions on Networking*, vol. 24, no. 6, pp. 3412–3425, 2016.
[25] T. Mizrahi, O. Rottenstreich, and Y. Moses, "Timeflip: Scheduling network updates with timestamp-based tcam ranges," in *INFOCOM 2015*.
[26] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *SIGCOMM 2014*.
[27] J. Zheng, H. Xu, G. Chen, and H. Dai, "Minimizing transient congestion during network update in data centers," in *ICNP 2015*.
[28] R. Gandhi, O. Rottenstreich, and X. Jin, "Catalyst: Unlocking the power of choice to speed up network updates," in *CoNEXT 2017*.
[29] K. Foerster, "On the consistent migration of unsplittable flows: Upper and lower complexity bounds," in *NCA 2017*.
[30] U. Feige, "A threshold of ln n for approximating set cover," *Journal of the ACM*, vol. 45, no. 4, pp. 634–652, 1998.
[31] D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms*, 2011.
[32] Y. Chen and J. Wu, "Max progressive network update," in *ICC 2017*.
[33] S. Kandula, I. Menache, R. Schwartz, and S. R. Babbula, "Calendaring for wide area networks," in *SIGCOMM 2014*.
[34] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM 2014*.

**Yang Chen** received her B.Eng. degree in Electronic Engineering and Information Science from University of Science and Technology of China in 2015. She is currently a Ph.D. candidate in the Department of Computer and Information Sciences, Temple University, Philadelphia, Pennsylvania, USA. Her current research focuses on Software Defined Networks.



**Huanyang Zheng** received his B.Eng. degree in Telecommunication Engineering from Beijing University of Posts and Telecommunications, Beijing, China, in 2012. He is currently a Ph.D. candidate in the Department of Computer and Information Sciences, Temple University, Philadelphia, Pennsylvania, USA. His current research focuses on mobile networks, social networks, and cloud systems.

**Jie Wu** is currently the chair and a Laura H. Carnell professor at the Department of Computer and Information Sciences, Temple University. Prior to joining Temple University, he was a program director at the National Science Foundation and a distinguished professor at Florida Atlantic University. His current research interests include mobile computing and wireless networks, routing protocols, cloud and green computing, network trust and security, and social network applications. He regularly publishes in scholarly journals, conference proceedings, and books. He has served on several editorial boards, including IEEE Transactions on Mobile Computing, IEEE Transactions on Service Computing, Journal of Parallel and Distributed Computing, and Journal of Computer Science and Technology. He was the general co-chair/chair for IEEE MASS 2006 and IEEE IPDPS 2008 and the program co-chair for IEEE INFOCOM 2011. He is currently the general chair for IEEE ICDCS 2013 and ACM MobiHoc 2014, and the program chair for CCF CNCC 2013. He was an IEEE Computer Society distinguished visitor, ACM distinguished speaker, and the chair for the IEEE Technical Committee on Distributed Processing. He is a China Computer Federation (CCF) distinguished speaker and a fellow of the IEEE. He received the 2011 China Computer Federation Overseas Outstanding Achievement Award.