Minimizing Traffic Migration During Network Update in IaaS Datacenters

Ting Qu, Deke Guo, Yulong Shen, Xiaomin Zhu, Lailong Luo, Zhong Liu

Abstract—The cloud datacenter network is consistently undergoing changing, due to a variety of topology and traffic updates, such as the VM migrations. Given an update event, prior methods focus on finding a sequence of lossless transitions from an initial network state to an end network state. They, however, suffer frequent and global search of the feasible end network states. This incurs non-trivial computation overhead and decision-making delay, especially in large-scale networks. Moreover, in each round of transition, prior methods usually cause the cascaded migrations of existing flows; hence, significantly disrupt production services in IaaS data centers. To tackle such severe issues, we present a simple update mechanism to minimize the amount of flow migrations during the congestion-free network update. The basic idea is to replace performing the sequence of globally transitions of network states with local reschedule of involved flows, caused by an update event. We first model all involved flows due to an update event as a set of new flows, and then propose a heuristic method *Lupdate*. It motivates to locally schedule each new flow into the shortest path, at the cost of causing the extra migration of at most one existing flow if needed. To minimize the amount of migrated traffic, the migrated flow should be as small as possible. To further improve the success rate, we propose an enhanced method *Lupdate-S*. It shares the similar designs of *Lupdate*, but permits to migrate multiple necessary flows on the shortest path allocated to each new flow. We conduct large-scale trace-driven evaluations under widely used Fat-Tree and ER data center. The experimental results indicate that our methods can realize congestion-free network with as less amount of traffic migration as possible even when the link utilization of a majority of links is very high. The amount of traffic migration caused by our *Ludpate* method is 1.2 times and 1.12 times of the optimal result in the Fat-Tree and ER random networks, respectively.

Index Terms—IaaS Datacenter, network update, traffic migration, congestion-free

1 Introduction

Infrastructure as a Service (IaaS) is a form of cloud computing, which enables tenants to multiplex computing, storage and network resources in data centers. With the rapid growth of IaaS, service providers have to rent or build a large-scale data centers. Inside each data center, thousands of switches and hundreds of thousands of servers are interconnected using a specific data center network (DCN). A cloud data center frequently occurs DCN updates, such as topology update and traffic update triggered by operators, applications, even switches failures [1]. For example, data center operators routinely upgrade existing switches and introduce new switches to interconnect more servers; hence, they cause the topology updates. For applications, the normal migration of VMs and reconfiguration of load balancers incur the traffic updates.

DCN updates are very challenging due to the complex nature. First of all, an update process consists of multiple stages, each of which must be well performed so as to guarantee the per-flow and per-packet consistency of the network configuration [2]. Thus, a plan for each update stage should be calculated in advance to arrange the update order of those involved flows [1–4]. Secondly, the update process has to ensure that any flow or packet is forwarded according to either old network configuration or new configuration, not the combination of them. Thirdly, it is time-consuming to perform an update in large-scale data center network [4–7]. Rapid network update is beneficial for achieving high network utilization and enhancing the network flexibility.

Reitblatt et al. propose a two-phase commit mechanism to implement a consistent update. A simple label is used to judge which configuration should be employed to forward a flow, using either new or old network configurations [2]. This mechanism can ensure the per-packet and per-flow consistency, but is incapable when the traffic matrix and the network topology change jointly. Hong et al. divide a flow into several segments, which are delivered along multiple feasible paths so as to improve the utilization of link bandwidth [4]. Even the traffic matrix varies a little bit, the time-consuming method needs to recalculate all possible network states again. Liu et al. prefer to calculate a complex update plan for each update event in advance to perform a congestion-free update [1]. Undoubtedly, it is time-consuming to derive an update plan and consumes non-trivial space to store the update sequences. For any update event, Jin et al. aim to fasten the process of consistent update by dynamically adjusting the preferred update plan among all feasible update plans [8].

Given a DCN update event, previous methods focus on finding a sequence of lossless transitions from an initial network state to a feasible end network state. They, however, suffer frequent and global search of the feasible end network states by taking a long time to solve an optimization problem. This incurs non-trivial computation overhead and decision-making delay especially in large networks.

• *T. Qu, D. Guo, X. Zhu, L. Luo and Z. Liu are with the Key laboratory for Information System Engineering, College of Information System and Management, National University of Defense Technology, Changsha 410073, P.R. China. E-mail: dekeguo@nudt.edu.cn.*
• *Y. Shen is with the School of Computer Science and Technology, Xidian University, Xian 710071, P.R. China. E-mail: ylshen@mail.xidian.edu.cn.*

Moreover, some involved flows have to be rerouted and migrated during each round of transition of network state, and usually cause cascaded migrations of other flows, and so on. Such kind of cascaded flow migration incurs considerable extra overhead and significantly disrupts existing applications. That is, this cascaded behavior results in a mass of unnecessary traffic migration and installation of flow rules. This will not only causes serious link congestion and packet loss, but also increases the delay of completing a network update event. To solve such severe issues, we design a simple update mechanism to minimize the amount of flow migrations during the process of congestion-free network update. For an update issue, the basic idea is to replace performing the globally transitions of network state with dynamic but local reschedule of involved flows.

We first model all involved flows due to a network update as a set of new flows, and then propose a heuristic method *Lupdate*, where " *L* " stands for migrating the least amount of traffic to accommodate a set of new flows. With *Lupdate*, operators neither need to consider the search of the feasible end network state nor carefully design and perform an update order of involved flows. The reason is that *Lupdate* motivates to locally schedule each new flow into the shortest path at the cost of causing the extra migration of at most one existing flow in most cases. As you can imagine, increased number of migrated flows means more flow rules will be installed on switches, which achieves update with more cost. To improve the update affection, we should select migrated flow which passes through all bottleneck links. To minimize the amount of migrated traffic, the removed flow for the new flow should be as small as possible. Thus, *Lupdate* migrates the least amount of traffic and the least number of flows to other routing paths, according to the real-time network state.

In reality, there may exist some new flows that cannot be accommodated through our *Lupdate* method timely. The root cause is that the available capacity of each desired shortest path for a new flow is not sufficient to accommodate that new flow, if we just migrate one existing flow in those bottleneck links. To improve the success rate of a network update, we further propose an enhanced method *Lupdate-S*. It shares the similar design methodology of *Lupdate*, but permits to migrate multiple necessary flows on the shortest path allocated to each new flow. Similarly, any migrated flow should be rerouted to another appropriate shortest path. To avoid cascaded flow migrations, the new flow should not be injected immediately if any migrated flow could not find another appropriate shortest path. Considering that flows by an update events may be updated concurrently, as well as update events. Thus, our methods have room to be improved on the update speed in the future while facing multiple flows or update events. To achieve it, we face two main challenges: conflicts between migrated flows and update events. In concrete terms, the migrated sequence of existing flows will affect whether a new flow could be transmitted in the network. That is, the occupied or emptied link space of migrated flows will block or cannot satisfy the requirements of the new flow. Similarly, the update sequence of multiple update events will be affected each other.

The major contributions of this paper are summarized as follows:

- We propose the problem of minimizing the traffic migration during network update in IaaS data centers and formulate it as an optimization problem.
- After modeling involved flows due to a network update event as a set of new flows, we present a simple update method *Lupdate* to resolve the defined optimization problem with high success rate, at the cost of migrating at most one existing flow.
- To improve the success rate of network update, we further enhance the *Lupdate* by designing a general update method *Lupadte-S*. It may migrate multiple existing flows to ensure that the scheduled path for a new flow has sufficient capacity. We also define a delay update scheme to ensure the success updates.
- We evaluate the performance of previous and our update methods via large-scale trace-driven simulations under widely used Fat-Tree and ER random datacenter networks. The results indicate that our methods can realize congestion-free network update with as less amount of traffic migration as possible even when the proportion of links with high utilization is very high. More precisely, the amount of traffic migration caused by the *Ludpate* method is 1.2 times and 1.12 times of the optimal result in the Fat-Tree network and the ER random network, respectively. Additionally, the *Lupdate-S* method can accommodate update flows with higher success rate than the *Lupdate* method as expected.

The remainder of this paper is organized as follows. Section 2 introduces the related work and Section 3 proposes the network update problem with minimal amount of traffic migration and formulates network update problem as an optimization model under given constraints. In Section 4, we propose two dedicated update methods to resolve the defined network update problem. We discuss some important issues of our model and methods in Section 5. Finally, we evaluate the performance of our methods and conclude this paper in Section 6 and Section 7, respectively.

2 Related work

IaaS DCNs has become an increasingly hot topic, along with several existing studies, ranging from bandwidth allocation and performance guarantee. Guo et al. propose several effective methods to guarantee bandwidth for VMs and balance the tradeoff between bandwidth guarantee and bandwidth sharing [9]. To improve system performance, Guo et al. proposed a practical distributed bandwidth allocation algorithm to provides performance guarantee to users [10]. In addition, several methods are proposed to guarantee VMs performance [11–13].

Reitblatt et al. propose a two-phase commit method to guarantee the per-packet and per-flow consistency [2]. It first installs the new routing rules on involved internal switches for the packets with new version numbers. Accordingly, the packets are marked with new version numbers when passing through the ingress switches. Such a mechanism is immensely useful. However, due to enormous traffic migrations during a network update, many factors will exacerbate the issue of link congestion during the update, such as the intrinsic difficulty of update on synchronous switches and the appearance of a traffic peak during the update. Meanwhile, the authors also propose a mechanism to perform the congestion-free traffic migration during DCN updates [2]. Note that, this work is designed based on the assumption that both the traffic matrix and the network topology remain constant during the update process. Such two assumptions, however, cannot be always achieved, due to the uncertainty of application traffic and the dynamic change of the network topology.

In contrast, Hong et al. divide a flow into several segments, which are delivered along diverse paths [4]. Such a design is help to improve the utilization of link bandwidth. However, even the traffic matrix varies a little bit, the proposed method needs to recalculate all feasible end network states, which is, undoubtedly, time-consuming and causes serious decision delay. Those issues become serious as the growth of network scale, due to the explosion of the feasible network states. Jin et al. provide another consistent update method to dynamic change the update plan for speeding the update process. The motivation is to exploit the diversity of update speed among different update sequences from an initial network state to an end network state [8].

The implementation of all previous update methods rely on sufficient information about the whole network and centralized decision-making. The emergency of Software Defined Network (SDN) has brought great convenience to support the network update. SDN is a new network architecture in which the data forwarding plane is decoupled from the control plane. A centralized controller maintains the global network state and calculates the routing path for each flow, which guides the forwarding configurations of underlying network equipment.

With the advents of SDN, recent proposals investigate how to maintain the consistency property during the process of a network update. That is to say, each flow must be delivered according to the old network configuration or new network configuration. Ludwig et al. design dedicated methods for secure network updates, where packets are forced to traverse certain waypoints or middleboxes [14]. Ghorbani et al. focus on the design of network update methods that guarantee even stronger consistency [15]. The authors in [16] introduce the notion of software transactional networking, and give a tag-based method to consistently compose concurrent network updates.

Additionally, several proposals focus on reducing the memory overhead due to maintain the packet coherence [3]. To guarantee the connectivity during network updates, R-BGP is proposed to ensure that Internet domains stay connected as long as the underlying network is connected [17]. Besides, some works dedicate to maintain loop freedom during congestion-free update, such as the efficient solution proposed in [18]. Liu et al. concern the problem of asynchronously updating flow tables on involved network devices in cast of the topology update and traffic matrix update [1]. In addition, SWAN [4], Raza et al. [19] and Ghorbani et al. [3] provide solutions for addressing various congestion issues during network update. Noyes et al. design the update plan to maintain some invariants specified by the operator [20]. There is also a rich set of work on preventing transient misbehaviors during the updates of routing protocols [21].

Compared with the previous methods, we replace performing the globally transitions of network states with dynamic but local reschedule of involved existing flows due to an update event. We regard the flows caused by an update event as new flows and migrate the existing flows to satisfy the bandwidth requirements of such new flows if necessary. To minimize the amount of migrated traffic, the migrated flows should be as small as possible. In this way, we avoid taking a long time to solve the optimal problem for globally searching the feasible transient and final network states. Our methods can realize network update with as less amount of traffic migration as possible.

3 Problem statement and Optimization Model

We start with abstracting the data center network update problem as the reschedule problem of a set of new flows no matter types of update events. During the schedule of each new flow, we aim to minimize the amount of traffic migrations of existing flows when the allocated path lacks sufficient capacity to accommodate the new flow. We then characterize the problem of minimizing the traffic migration during network update as a dedicated optimization model.

3.1 Problem statement

The network is consistently undergoing changing, due to a variety of hardware and software issues. A common negative impact of network updates is the serious traffic migration across the whole network. For example, in the settings of upgrading switches or repairing failed switches, network operators have to reroute all existing flows passing through those switches in order to guarantee the correctness of involved network applications. VMs migration is another cause of network update. All traffic associated with the VMs has to be migrated as well once such VMs are redistributed to other servers in an IaaS data center. Such network-scale flow migrations, resulting from any network update event, should be performed carefully so as to disrupt existing flows of critical applications as less as possible. Otherwise, it may result in serious traffic congestion.

Fig.1 depicts an illustrative example. Suppose that the flows f_1 and f_2 are injected from two ingress switches, s_1 and s_2, respectively. To realize the transition from the initial network status in Fig.1(a) to the end network status in Fig.1(b), the new forwarding rules on s_1 and s_2 must be installed at the same time. Otherwise, as shown in Fig.1(c), the link l_2 will carry both flow f_1 and flow f_2 and may exhibit congestion, if s_1 updates its forwarding rules before s_2. Link l_1 will appear the similar result if s_2 first updates its forwarding rules [1].

To enable the congestion-free network update, previous methods share a complex abstraction of the network update problem. The basic idea is to design and implement a lossless transition plan, consisting of a sequence of networks states from the initial one to the end one. Such an abstraction suffers global search of all feasible transient states and end network states, by taking a long time to solve an

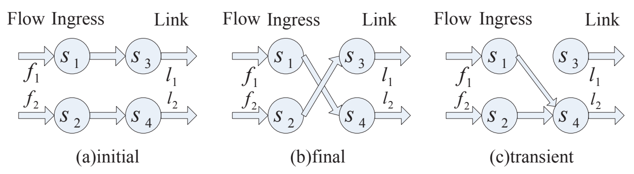


Fig. 1. Transient load congestion during the process of traffic migration.

optimization problem. After fixing the desired end traffic distribution, all intermediate traffic distributions should be calculated if necessary. Such kind of methods incur non-trivial computation overhead and decision-making delay, especially in large networks. Moreover, the designed update plan usually becomes invalid and should be recalculated, even in the case of a little bit change of the traffic matrix or network topology.

In summary, the previous abstraction about the network update is very complex and incurs frequent and global search. To tackle those severe issues, we present a simple abstraction from the view point of involved flows caused by update events, due to the following observations. That is to say, we regard such involved flows as new flows and allocate paths for them. If there is not enough link bandwidth to transmit them, we consider to migrate several existing flows to make room for new flows. For the update event of network topology, such as switch upgrade or failure, all existing flows around involved switches should be rescheduled to other available routing paths. For the update event of VM migrations, a mass of new flows has to be rescheduled to available routing paths towards those destination servers. In addition, the reconfiguration of load balancers will also lead to the change of routes to some existing flows.

Thus, we treat a network update event as the dynamic rescheduling problem of involved new flows, instead of global searching the possible end network states and realizing a series of transient states between traffic distributions. The basic idea is to locally allocate a feasible shortest path for each new flow, whose available bandwidth is sufficient to accommodate that flow. If none of shortest paths is feasible, we prefer to migrate the least number and amount of existing flows from any shortest path so as to make that path become feasible. In this way, operators neither consider the global search of all feasible end network states nor carefully perform a sequence of lossless transitions between the initial, intermediate, and the end network states.

Our update strategy can well realize the congestion-free transmission of those involved flows in a simple way. It, however, still faces two challenging issues, the cascaded migration problem and the minimization of traffic migration. Firstly, we have to find another feasible routing path to accommodate each removed existing flow if necessary. This can cause the cascaded migrations of other existing flows, and so on. Thus, we impose a constraint on the selection of a removed existing flow such that there exists at least one desired and feasible path to accommodate it after migration. Secondly, we have to minimize the total number and amount of migrated existing flows, so as to enable the reschedule of involved flows due to a network update event. This will minimize the disputation to existing flows and applications. Moreover, our experimental results show that the fewer amount of traffic we migrate, the fewer packets will lost during the update process.

TABLE 1
Symbols and notations.

V	The set of all switches
E	The links between all switches
G	The direct network graph $G = (V, E)$
$e_{i,j}$	Link connecting i and j. $i, j \in V$
$e_{i,j}^k$	The number of the flows on the link $e_{i,j}$ is k
$c_{i,j}$	The link capacity between i and j. $i, j \in V$
s_f	Ingress switch of flow f
d_f	Egress switch of flow f
$v(f)$	The size of flow f
f	A flow is defined as $f = (s_f, d_f, v(f))$
F	All existing flows in the network
F_{move}	The set of candidate migrated flows transmitted on the congested links
f_{move}	The set of migrated flows
$f_{e_{i,j}}$	The flow which passes through the link $e_{i,j}$
p_f	A path taken by f from s_f to d_f
$\|p_f\|$	The length of path allocated for f
L_f	A set of congestion links on the route which flow f passes by
$v(f_{i,j})$	Flow $f's$ load on link between i and j
G_f	The subgraph of the switches and links flow f passing by
$\|G_f\|$	The number of flows in the subgraph G_f
$\|f_{move}\|$	The number of flows in the set f_{move}
T	The load distribution matrix
D	The diameter of network

3.2 Optimization model

After abstracting the datacenter network update as the rescheduling problem of a set of new flows, we motivate to cause the least amount of migration of existing traffic to achieve the least cost and congestion-free update. Before characterizing our optimization model, we first report all used notations and symbols in Table 1.

We define a network as a graph $G{=}(V, E)$, where V and E denote the set of switches and the set of links connecting those switches, respectively. D is defined as the diameter of the network. Besides, a flow is defined as $f{=}(s_f, d_f, v(f))$, where s_f is the ingress switch, d_f is the egress switch and $v(f)$ is the size of flow f. Let f_{new} and f_{move} denote the involved flows caused by an update event and the flow that will be moved to other path caused by a new involved flow, respectively. Let $v(f_{i,j})$ denote the actual load of flow f on link $e_{i,j}$ and $c_{i,j}$ denote the link capacity. G_f represents the subgraph that contains all the switches and links employed by flow f. Furthermore, F records the size of each flow in the network, and a traffic distribution matrix T records the load of each link.

Given a network update event, we initialize the set f_{new} as the set of all involved flows due to this event. Apparently, the amount of migrated traffic will affect the functionality of the ongoing network. And thus, the process of update should migrate the least amount of traffic to limit the negative impact. Thus, our first object is:

$$\min \sum_{f \in f_{move}} v(f) \tag{1}$$

Besides, we migrate parts of existing flows for a new flow, which means these existing flows will be routed new paths. Further, the corresponding flow policies should be installed on the switches along the paths. Due to asynchronous switch, the process of installing will delay the migration of existing flows. As a result, congestion maybe happen on some links. Thus, we should try to minimize the number of the migrated flows. To achieve it, we select a minimal subset f_{move} from the candidate flow set F_{move}, which consists all existing flows on the congestion links of the path $p_{f'}$ and guarantee each link of $p_{f'}$ will be congestion-free.

$$\min \sum_{f_{move} \in F_{move}} |f_{move}| \tag{2}$$

$$\begin{aligned} &\forall f' \in f_{new}, \forall i, j \in G_{f'} : \\ &if \sum_{f \in F} v(f_{i,j}) + v(f'_{i,j}) > c_{i,j}, then\ f \in F_{move} \end{aligned} \tag{3}$$

$$\begin{aligned} &\forall f' \in f_{new}, \exists f_{move} \subset F_{move}, \forall i, j \in G_{f'} : \\ &\sum_{f \in F} v(f_{i,j}) + v(f'_{i,j}) - \sum_{f'' \in f_{move}} v(f''_{i,j}) \leq c_{i,j} \end{aligned} \tag{4}$$

Given T, we call that network is legal if parameters hold that:

$$\forall f, \forall i, j \in G_f : v(f_{i,j}) = v(f) \tag{5}$$

$$\forall f, e_{i,j} \notin G_f : v(f_{i,j}) = 0 \tag{6}$$

$$\forall f' \in f_{new} : l_{f'} \leq D \tag{7}$$

$$\forall f'' \in f_{move} : l_{f''} \leq D \tag{8}$$

$$\forall e_{i,j} \in L_f, \exists f'' \in f_{move} : e_{i,j} \in G_{f''} \tag{9}$$

$$\forall f' \in f_{new} : |f_{move}| \leq |G_{f'}| \tag{10}$$

Equation (3) records the set of migrated flows, which pass through congestion links of the path $p_{f'}$ allocated for the new flow. And (4) indicates that links, whether new flows or migrated flows caused by new flows occupy, are still within their capacities. That is, the link is always congestion-free. Equation (5) guarantees that each flow can be transmitted along the identified path in the network. (6) means the traffic of flow $f_{i,j}$ will not appear on the links which do not belong to G_f. (7) and (8) indicate that hops of paths allocated for new flows and moved flows cannot exceed the diameter of the network. (9) guarantees that the migrated flows which pass through the congestion links initially. (10) indicates that the number of migrated flows cannot exceed the number of flows on the path allocated for the new flow. Based on this model and given T and f_{new}, new flows can be allocated paths with the least cost.

4 Efficient update methods

After modeling an update event as the reschedule problem of a set of new flows, we design two methods, *Lupdate* and *Lupdate-S*, to approximate the optimal solution. *Lupdate* aims to reschedule each new flow to the shortest path at the cost of migrating at most one existing flow. The size of migrated flow should be as small as possible. *Lupdate-S* further improves the success rate of an update event, by migrating the least but not only one existing flow on the allocated shortest path for each new flow.

Algorithm 1 Lupdate (T, f)

```
1: Find all candidate shorter paths for flow $f_f$.
2: Sort these paths from short to long distance, $p_1$, $p_2$ ...
3: for $p_i$, $i \leftarrow 1, 2, 3...$ do
4:    if each link of $p_i \leq c_{i,j}$ then
5:       $flag = 1$
6:       return $p_i$
7: if $flag \neq 1$ then
8:    for $p_i$, $i \leftarrow 1, 2, 3...$ do
9:       Search all flows on every congestion links of $p_i$.
10:      Search for the common flows$f_1, f_2...f_k$ on every
         congestion links.
11:      if no common flow then
12:         return Fail to transform the new flow.
13:      else
14:         Sort these flows from small to big, $f_1, f_3....$
15:         for $f_i$, $i \leftarrow 1, 3...$ do
16:            if $p_i$ can enter after moving $f_i$ then
17:               $f_{move}$ = $f_i$
18:               Move $f_{move}$
19:               return $p_i$
```

4.1 Lupdate

Given a set of existing flows, the load distribution matrix T, and a set of new flow f_{new}, *Lupdate* tries to locally pick an appropriate path for each new flow from all possible shortest paths. First of all, the rest capacity of the selected path should be sufficient to accommodate a new flow $f \in f_{new}$. If all candidate paths are out of capacity, *Lupdate* identifies all existing flows on those congestion links of a candidate path. Then, the common flows on those congestion links will be found and sorted according to their sizes in the ascend order. The first least flow will be migrated if its release can make the path satisfy the capacity demand of the new flow. Finally, the migrated flow should be moved to another feasible path whose length is as short as possible. If all existing flows on the considered shortest path cannot be migrated, other candidate paths for the new flow will be verified again using the similar way.

The worse case is that a new flow $f \in f_{new}$ cannot be scheduled to any candidate path successfully. That flow should wait on the ingress switch until any candidate path becomes feasible due to the completion of some existing flows. In *Lupdate*, only one existing flow along each candidate path for a new flow is permitted to migrate for limiting the negative impact on the whole network. Algorithm 1 characterizes the basic idea of our *Lupdate* method formally.

For each flow $f \in f_{new}$, we first search out a set of candidate paths for flow f and sort them in the ascending order of the path length. Then, we check (lines $3-8$) whether flow f could be loaded on the first candidate path without migrating any existing flows. If the result is yes, flow f_{new} will enter the network by scheduling on the first candidate path. Otherwise, other candidate paths will be checked. If none of candidate paths can accommodate the new flow f without migrating any existing flows, we estimate whether one of the shortest paths can accommodate the new flow after migrating one existing flow. The migrated flow is selected according to the following steps:

Algorithm 2 Lupdate-S (T, f)

```
1:  Find all candidate shorter paths for flow $f$.
2:  Sort these paths from short to long distance, $p_1$, $p_2$ ...
3:  for $p_i$, $i \leftarrow 1, 2, 3...$ do
4:    if each link of $p_i \leq c_{i,j}$ then
5:      $flag = 1$
6:      return $p_i$
7:  if $flag \neq 1$ then
8:    for $p_i$, $i \leftarrow 1, 2, 3...$ do
9:      Search all flows on each congestion link on $p_i$.
10:     Search the common flows $f_1, f_2 ... f_k$ among those congestion links on $p_i$.
11:     if no common flow then
12:       for $\beta \leftarrow 1, 2, 3...n$ do
13:         $flag_1 = 0$, $flag_2 = 0$
14:         Copy the traffic matrix $T$ to $X$.
15:         for each link $e_{i,j}$ of $p$ do
16:           $X_{i,j} = v(f) + X_{i,j}$
17:         for each link $e_{i,j}$ of $p$ do
18:           if $X_{i,j} \leq c_{i,j}$ then
19:             $flag_1 = 1$, Success
20:           else
21:             Put existing flows on this link into a set $Move_{flows}$.
22:             Break
23:         Randomly migrates $\beta$ flows in the set $Move_{flows}$ to other shortest appropriate paths.
24:         if The exist flows have been moved then
25:           $flag_2 = 1$
26:         if $flag_1 = 1$ and $flag_2 = 1$ then
27:           $T = TT$
28:           return $p$
29:     else
30:       Call Algorithm 1
```

- Firstly, we should find out those bottleneck links along the path for flow f, which will become congestion after loading flow f on it.
- Secondly, after recording the set of existing flows on each congestion link, we identify those common flows that pass through all bottleneck links.
- Finally, after sorting those flows found in the second step, we try to migrate the least flow to enable the related path become feasible to accommodate the flow f.

To express it clearly, a example is presented in the Fig 2, where six switches A, B, ... , F exist and the link capacity is 10 Mbps. Four flows, i.e. f_1, f_2, f_3, f_4 whose bandwidth requirements are 3 Mbps, 10 Mbps, 2 Mbps and 4 Mbps respectively, are transmitting in the network. The initial paths of them are illustrated using the blue lines. Suddenly, a new flow from A to F, whose bandwidth demand is 3 Mbps, demands to passes through D and E. Apparently, link congestion will occur if we transmit the new flow on the route $A \rightarrow D \rightarrow E \rightarrow F$. Thus, we should find out the bottleneck link of this path, i.e. $D \rightarrow E$. Then, f_1, f_3 and f_4 become candidate flows to be migrated. According to the principle of minimizing the amount of migrated traffic, f_1 is chosen to be removed, showed by the red line.

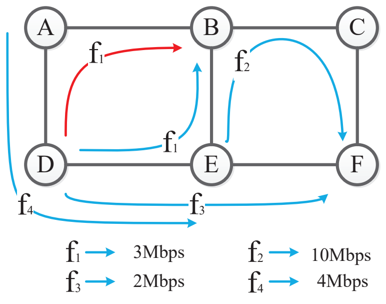


Fig. 2. An illustrative example of Algorithm 1.

In addition, the migrated flow should be removed to another shortest path as well (lines $10 - 20$). Theorem 1 proves the time complexity of our *Lupdate* method.

Theorem 1. Given a new flow and network, the number of candidate paths for a new flow is α. The length $|p_f|$ of each path and traffic of each existing flow on this path is fixed. It is apparent that the time complexity of *Lupdate* is $O(V + E + \lg \alpha + \alpha \times (|p_f| + \lg(k) + k))$.

Proof 1. Algorithm 1 consists of two major stages, including pre-update stage and move stage. For the pre-update stage, the time complexity is $O(V + E + \lg \alpha)$. Firstly, we search out all the appropriate paths for the new flow and use a modified depth-first search to generate the candidate paths which can be found in $O(V + E)$ time complexity. Then, we need to sort the length of these paths at $O(\lg \alpha)$ time complexity. So, the time complexity of the pre-update stage is $O(V+E+\lg \alpha)$. In the stage of move, we try to move the smallest existing flow on the candidate paths from the short to long, deciding which path should be allocated for the new flow. All of this should be performed for α times at most. In addition, we can get the number $e^k_{i,j}$ and the size of the existing flows on every congestion link from the SDN controller easily. Then, we sort the flows on each congestion link at $O(\lg k)$ time complexity and find out all common flows of all congestion links with the time complexity $O(k)$. Finally, we will move the chosen flow to the other path at $O(|p_f|)$ time complexity. Thus, the entire complexity of the second stage is $O(\alpha * (|p_f| + \lg(k) + k))$. In summary, the entire time of complexity of two stages is $O(V + E + \lg \alpha + \alpha \times (|p_f| + \lg(k) + k))$; hence, Theorem 1 is proved.

It is true that our *Lupdate* method may fail to reschedule a new flow after checking all candidate paths, when the proportion of high utilization links is very large in the network. A possible solution to address this rare case is the introduction of the delay update strategy. That is, that flow should wait on the ingress switch for a delay update time until any candidate path becomes feasible due to the completion of some existing flows. Another online solution is to migrate more existing flows on candidate paths for that new flow, as discussed by our improved update method *Lupdate-S*.

4.2 Lupdate-S

As aforementioned, our *Lupdate* method may cannot timely reschedule a new flow after checking all candidate paths

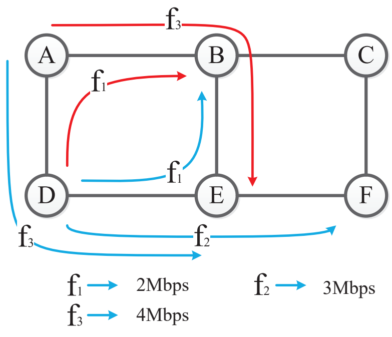


Fig. 3. An illustration example of Algorithm 2.

in rare cases. To resolve this problem, we introduce the *Lupdate-S* method, which is permitted to migrate multiple number of existing flows on candidate paths for accommodating the related new flow if necessary. It is clear that the *Lupdate-S* method is a general case of the *Lupdate* method. Thus, there may exist many candidate shortest paths, which can afford sufficient bandwidth for a new flow after migrating multiple existing flows.

The challenge issue is how to efficiently choose an appropriate path from more candidate ones. We explain the detailed process to tackle this issue in Algorithm 2. Firstly, we search the shortest path for each new flow and distinguish all the existing flows on this path (lines $1, 2$). The next step is to identify whether the available capacity of each link on this path is sufficient for accommodating the new flow. If the available capacity of each link is sufficient, the update strategy is successful since the current shortest path can load the new flow directly. Otherwise, we will collect those existing flows on congestion links and find the common flows passing through all congestion links (lines $3-10$). If such congestion links do not share any flow, we need to migrate several flows on each congestion link to satisfy the demand of the new flow. The number of migrated flows depends on the size of the new flow and the available bandwidth on each link along a candidate path. We will terminate the moving process of exist flows once the new flow can be transferred (lines $14-28$). If there exists some flows that pass through all the congestion links, we will call Algorithm 1 to perform network update (30).

In addition, we give an example to explain this algorithm as shown in Fig. 3. Three flows, i.e., f_1, f_2, f_3, originally pass along three blue paths $D \to E \to B$, $D \to E \to F$ and $A \to D \to E \to F$, respectively. The bandwidth of each link is 10 Mbps, which the required bandwidth of such three flows are 2 Mbps, 3 Mbps and 4 Mbps. A new flow with 6 Mbps data rate desires to transmit along the path $A \to D \to E \to F$. In this way, link $D \to E$ would become congested. To tackle this issue, we try to update the routing path of one existing flow to release bandwidth for the new flow. *Lupdate-S* permits to migrate multiple existing flows with as less traffic as possible. It seems that we can migrate the minimal flows f_1 and f_2. However, congestion still will happen on the link $A \to D$. Thus, we choose to migrate flows f_1 and f_3 along the new paths, as shown in red lines.

Theorem 2. The time complexity of the *Lupdate-S* method is $O(V \lg V + E + max(\beta \times (|p_f| + V \lg V + E), \gamma \times (|p_f| + \lg(\beta))))$.

Proof 2. Algorithm 2 consists of two major stages, including the pre-update stage and update stage. For the pre-update stage, the time complexity is $O(V \lg V + E)$. The reason is that we employ the Dijkstra algorithm, whose time complexity is $O(V \lg V + E)$ to search the shortest path for each new flow. Note that there may exist γ shortest paths for a new flow.

In the update stage, *Lupdate-S* migrates existing flows on congestion links along the shortest path. We analyze the time complexity from two sub-stages. First, if there exists no common flow passing through all the congestion links, we will migrate multiple flows on each congestion link, such that the available bandwidth of each bottleneck link become feasible to load the new flow. This stage consists of $|p_f|$ steps, where $|p_f|$ denotes the length of the path allocated for a new flow. Then, several existing flows are randomly migrated to other shortest paths at the $O(V \lg V + E)$ time complexity. The number of the migrated flow, i.e., β, depends on the size of the new flow, and the available bandwidth on related link. Thus, the total time complexity of this sub-stage is $O(\beta \times (|p_f| + V \lg V + E))$. Second, if there exists some flows that pass through all the congestion links, we will try to migrate the smallest flow to satisfy the demand of the new flow with the time complexity $O(\gamma \times (|p_f| + \lg(\beta)))$. In summary, the time complexity of Algorithm 2 is $O(V \lg V + E + max(\beta \times (|p_f| + V \lg V + E), \gamma \times (|p_f| + \lg(\beta))))$; hence, Theorem 2 is proved.

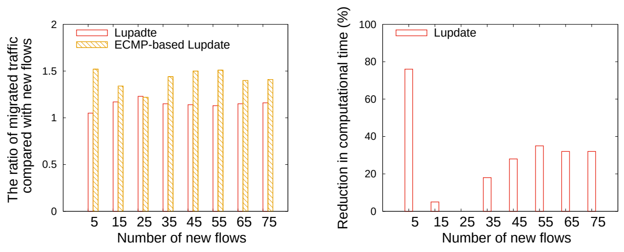
5 Discussion

Note that all involved flows due to a network update are modeled as a set of new flows. The current design of update methods process the set of such flows sequentially. To further speed the update process, we would improve the aforementioned update methods by introducing the design rational of concurrent update and delay update.

5.1 Concurrent update

If partial or all flows caused by a update event can be updated concurrently, our update method still has room to speed the completion of a network update event. To realize such a design rational, a group of new flows should be allocated paths together, while the migrated flows among existing flows should not conflict. Otherwise, serious congestion will happen and leads to considerable interruption on network applications. For this reason, we further explore the feasibility of our proposed methods to concurrently update involved flows of an update event.

Given an update event and the set of caused new flows F_{new}, an appropriate path should be identified for each flow $f \in F_{new}$ and the set of migrated existing flows for flow f is also recorded. We then verify whether the migrated flows for different flows in F_{new} occur potential conflicts. After that, we divide all migrated flows for the update event into a series of groups, in each of which all members are conflict-free. Thus, a group of flows can be updated concurrently. Clearly, such an improved design can accelerate the completion of a network update event. At the same time, the benefits of *Lupadte* and *Lupdate-S*, imposing the least influence to the network, are reserved.



(a) Migrated traffic (b) Computation time

Fig. 4. Migrated traffic ratios of our method Lupdate, ECMP-based Lupdate and new flows, and reduction in computational time with our method Lupdate against ECMP-based Lupdate where the network utilization is 40% in the 8-Pod Fat-Tree network.

5.2 Delay update

Given a network update event, there may exist few flows which could not be rerouted no matter using the Lupdate or Ludpate-S method. In this scenario, it incurs extra delay on the update process and enlarges the average completion time of a network updated event. Traditionally, new flows in the queue should be updated by our *Lupdate* and *Lupdate-S* in the FIFO order. To update the first flow, we should confirm that the network has extra bandwidth to load this new flow. That is, the *Lupdate-S* method can successfully reschedule that new flow. If not, the first new flow should wait until any candidate paths becomes feasible due to the completion of some existing flows. This design makes other new flows behind the first flow in the queue wait in the egress switch.

To address such an important issue, we introduce the design rational of delay update. The basic insight is to temporarily try other flows behind the first flow, when the first flow cannot be immediately allocated to a desired routing path. If any later flow in the queue meets the update requirement first, it will be rescheduled. After a period of time, we will check the rest flows in the queue in the order of FIFO and process them using the same way. This will considerably shorten the completion time of a network update event.

6 Performance Evaluation

In this section, we start with the setting and configuration of our trace-driven evaluations. We then compare our two methods with the *Shortest-Path* method, which allocates the shortest path for new flow and migrates multiple flows randomly to satisfy the demand of the new flow, in terms of the amount of migrated traffic, the number of migrated flows, and the success rate of updating a set of flows. The reported results denote the average values over 100 rounds of experiments for each performance metric.

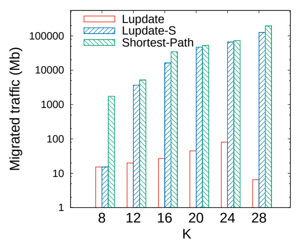
6.1 Evaluation configuration

Topology We emulate data centers with representative network structures, i.e., the Fat-Tree [22] and ER [23] random network. The link bandwidth is fixed as 1 Gpbs for both network models. The number of servers and switches in a Fat-Tree data center is determined by the setting of parameter k, ranging from 8 to 28 in our experiments. A Fat-Tree topology owes $5k^3/4$ switches and can support $k^3/4$ hosts, which are built by k-port switches comprises k pods with two layers of $k/2$ switches. The amount of switches in the ER random data center is set as the same as the Fat-Tree data center. That is, if the parameter is set as k in the Fat-Tree network, the number of switches in the ER network is set as $5k^3/4$ and the number of servers is $k^3/4$. A switch in an ER random network connects with each of other switches with probability p ($p = 2/3$). That is, other ports of a switch connect to servers with the probability $1 - p$. Such a probability is decided by a proportion of all links among switches to the total number of links in a Fat-Tree data center with the same setting of k. Note that the number of ports in each switch is limit. Thus, for the ER random network, the setting of ports in each switch is as same as the k-pod Fat-Tree network. That is to say, if there are k ports of each switch in the Fat-Tree network, then the number of ports for each switch is k in the ER random network. This application scenario brings some constraints on the ER model since it is not necessary to generate arbitrary node degree. That is, the number of ports in each switch is usually sufficient.

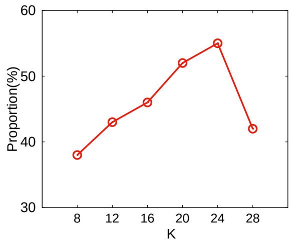
Methods Given a set of new flows caused by any update event as the input, we realize our two update methods, *Lupadte* and *Lupdate-S*, in the two kinds of emulated data centers. We also implement the *Shortest-Path* update method as an existing one, which only allocates the shortest path for each new flow at the cost of migrating multiple existing flows. During the selection of migrated flows, it does not consider whether links and existing flows would lead to the failure of loading the new flow on that path. Once a link lacks bandwidth to load the new flow, it only randomly migrates several existing flows until the available bandwidth of the link becomes sufficient.

For the method *Lupdate*, we focus on the scenarios of allocating one path for each new flow, due to the experimental observation as shown in Fig.4. Given an update event, Fig.4(a) reports the ratio of the size of migrated traffic to the size of new flows under our *Lupdate* and *ECMP-based Lupdate* methods. The two update methods would migrate $1.05-1.23$ and $1.22-1.52$ times existing traffic than new flows for accommodating the same set of new flows, respectively. It is clear that the *ECMP-based Lupdate* method migrated more traffic than our *Lupdate* method. We find that to allocate multiple paths for each new flow usually need to analyze the bottleneck links on multiple paths, and may migrate at least one existing flow on each path. Additionally, this would take much time to find available multiple paths for each new flow. As shown in Fig.4(b), our method Lupdate achieves $0\% - 80\%$ reduction in the computation time for finding feasible routing path compared to the *ECMP-based Lupdate* method.

Workloads To evaluate the three update methods in a trace-driven manner, we inject k^3 flows as the background traffic, which are generated by a real data-set from Yahoo!'s data center [24]. Such a trace records the basic information of each flow in its six distributed data centers, including the IP addresses of both source and destination servers, the size of each flow, etc. Note that, the real IP addressed in the trace are anonymous, hence, we employ a hash function to map the IP addresses of both source servers and destination servers into our networks. Recall that we abstract the impact

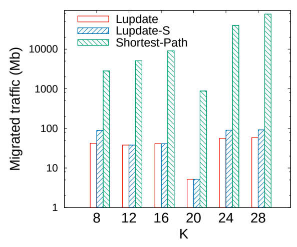


(a) The amount of migrated traffics (with y-axis log scale)

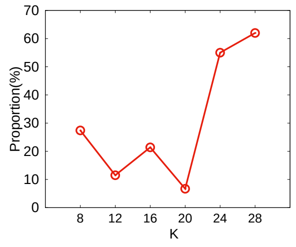


(b) Proportion of links whose utilization exceeds 90%

Fig. 5. The amount of migrated traffic in the Fat-Tree DCN under varied settings of k. The proportion metric refers those links whose link utilization exceeds 90%.



(a) The amount of migrated traffics (with y-axis log scale)



(b) Proportion of links whose utilization exceeds 90%

Fig. 6. The amount of migrated traffic in the ER random network under varied settings of k. The proportion metric refers those links whose link utilization exceeds 90%.

of varies network update events as a set of new flows in Section 3. We generate new flows as one of the following types for the upcoming different evaluation scenarios. The first type is the average flows, whose source and destination IP addresses are selected randomly, and the flow size is proportional to the average size of background flows in the network. The second type is the test flows, whose source and destination are deterministic. Inspired by the study of network traffic inside data centers [25], the size of such test flows is set randomly such that the average flow size ranges from 5 MB to 95 MB.

Metrics We evaluate the performance of the three update methods under any update event, in terms of three performance metrics, including the average amount of migrated traffic, the average number of migrated flows, and the rate of success. Furthermore, we also evaluate the influence of three design factors, the network size, the network topology, the average size of new flows, and the number of new flows on the performance.

6.2 The amount of migrated traffic

6.2.1 Impact of the network scale

We first evaluate the amount of migrated flows in Fat-Tree data centers, when k varies from 8 to 28. Fig. 5(a) depicts that our *Lupdate* method always migrates the least amount of traffic to accommodate new flows caused by an update event, irrespective of the value of k. On the contrary, the *Shortest-Path* method migrates too much traffic for accommodating the same flow set, almost $40{\sim}100$ times than the *Lupdate* method. The general *Lupdate-S* method allows to migrate multiple related existing flows, when a new flow fails to be accommodate by migrating just one flow. Although it incurs more traffic migration than the *Lupdate* method, but still outperforms the *Shortest-Path* method.

TABLE 2
Migrated traffic ratio of three methods in the Fat-Tree network

k	8	12	16	20	24	28
$ratio1$	0.87%	0.38%	0.07%	0.09%	0.11%	0.003%
$ratio2$	0.87%	71%	47%	89%	91%	65%

With different network scales, the ratios of migrated traffic of *Lupdate*, *Lupdate-S* and *Shortest-path* are present by ratio1 and ratio2, respectively.

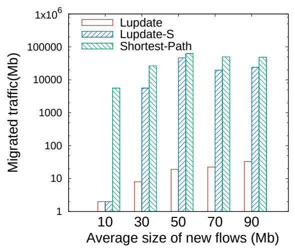
After injecting k^3 real trace flows as the background traffic, Fig.5(b) reports that the proportion of links, whose link utilizations exceeds 90%, fluctuates from 38% to 55% when k ranging from 8 to 28. Fig.5(a) and Fig.5(b) indicate that the *Lupdate* method always causes the least amount of traffic migration, even when the utilizations of a majority of links are very high. Moreover, the changing trend of the amount of traffic migration is similar to that of the proportion of links with high utilization under the *Lupdate* method. However, to the methods *Lupdate-S* and *Shortest-Path*, the amount of traffic migration is dominated by the network scale, i.e., the value of k. It is obviously that there is a weak connection between the amount of migrated traffic and the state of link utilization. To make it clearly, we show the ratio of the amount of migrated traffic to the size of new flows under such three methods in Table 2. Consider that *Lupdate* only migrates at most one existing flow for each new flow, the migrated traffic ratio of *Lupdate* and *Shortest-path* fluctuates between $0.03\% - 0.38\%$. *Lupdate-S* migrates much more traffic than *Lupdate*, due to pursue higher success rate to accommodate new flows. In addition, we note that the amount of migrated traffic for each new flow is not sensitive to the network scale as well.

6.2.2 Impact of the topology

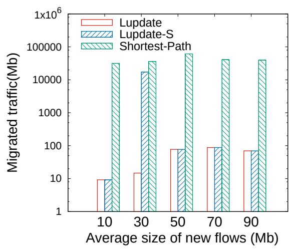
After evaluating those network update methods in Fat-tree data centers, we further study the impact of other network topologies of data centers, for example the ER random network. We also inject number of k^3 flows into the ER random network as background traffic and inject the first type of new flows mentioned in Section 6.1 to mimic an update event.

Fig.6(a) plots that the *Lupdate* method still migrates the least amount of traffic, irrespective of the values of k. The *Shortest-Path* method migrates more traffic than both *Lupdate* and *Lupdate-S*, which migrate the similar amount of traffic. Additionally, the difference of migrated traffic between *Lupdate* and *Shortest-Path* decreases as the expansion of the network scale.

As shown in Fig.6(a), the amount of migrated traffic grows up as the increase of the network scale except k=20 under each update method. We can explain such a phenomenon according to Fig.6(b). When k increases from 8 to 16, the amount of migrated traffic changes gradually while the proportion of links of high utilization varies within a certain range. However, when k reaches 20, the amount of

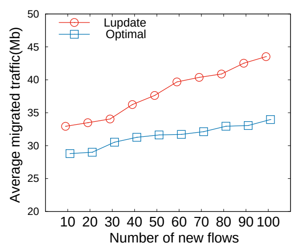


(a) The Fat-Tree DCN with k=20.

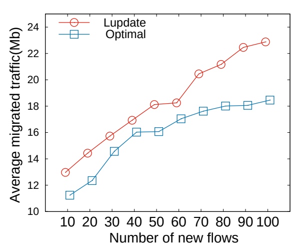


(b) The ER random DCN with 720 switches.

Fig. 7. The amount of migrated traffic in various average size of new flows (with y-axis log scale). The proportion of links (the utilization exceeds 90%) is 51.8% in the Fat-Tree DCN and 54.7% in the ER random network.



(a) The Fat-Tree DCN with k=8.



(b) The ER random DCN with 80 switches.

Fig. 8. The amount of migrated traffic in various number of new flows in the Fat-Tree DCN and ER random network.

migrated traffic sharply decreases under all three updated methods. At this moment, we can see that the proportion of links, whose utilization exceeds 90%, reduces to 6.7%. That is, there are sufficient bandwidths to accommodate new flows and little traffic needed to be migrated. When k reaches 24 and 28, the proportion of high utilization links increases rapidly and the amount of migrated traffic caused by each update method increases as normal circumstances. To display it clearly, we show the ratio of migrated traffic of *Lupdate* and *Lupdate-S* and *Shortest-path* in Table 3. when $k = 28$, the migrated traffic ratio of *Lupdate* and *Shortest-path* is 0.07%, and *Ludpate-S* and *Shortest-path* is 1.2%. In most cases, ratio1 and ratio2 are the same, for the reason that migrating at most one existing flow meets the link bandwidth requirements of majority of new flows.

Such evidences indicate that the amount of existing traffic migrated by those three methods increase in accordance with the regular before when the link utilization is back to normal. So, we can infer the link utilization and the scale of network can co-affect the amount of migrated traffic, and topology will not affect the size of the whole flow migration trend. According to the above results, our *Lupdate* method can remain the desired benefits under different network topologies of data centers.

6.2.3 Impact of the average size of new flows

As illustrated in Fig.7, we evaluate the impact of the average size of new flows caused by any update event under two scenarios. They are the Fat-Tree network with k=20 and the ER random network with 720 switches. Under the two scenarios, we inject a set of test flows mentioned in the Section 6.1 as new flows demanded to be rescheduled into the network, whose average size ranges from 10 MB to 90 MB.

TABLE 3
Migrated traffic ratio of three methods in the ER network

k	8	12	16	20	24	28
$ratio1$	1.49%	0.74%	0.45%	0.59%	0.14%	0.07%
$ratio2$	3.1%	0.74%	0.45%	0.59%	0.23%	1.2%

With different network scales, the ratios of migrated traffic of *Lupdate*, *Lupdate-S* and *Shortest-path* are present by ratio1 and ratio2, respectively.

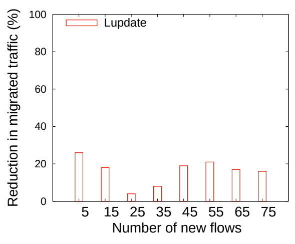
Fig.7(a) and Fig.7(b) describe the change rule of amount of migrated traffic, varying with the new flow size in the Fat-Tree network and ER network, respectively. The amount of traffic migration, caused by the *Lupdate-S* and *Shortest-Path* methods in Fat-Tree data centers, decreases sharply when the average sizes of new flows are 70 MB and 90 MB. The amount of migrated traffic, however, decreased sharply when the average size of new flows exceeds 30 MB to the *Lupdate-S* method under the ER random network. Later, the amount of migrated traffic keep within a certain range no matter the average size of new flows.

There are various reasons for above observations. First, there exists multiple shortest paths for each new flow. Due to the particularity of the ER network, we will choose a shortest path randomly. Second, as these three methods are a series of new flows injection scheduling, flow by flow. Thus, the routing plans for flows in front of the queue certainly affect the schemes of latter flows. Similarly, the injection order of new flows affects the schedule result as well. Third, due to new flows scheduled, certain existing flows may be migrated to other paths to satisfy the capacity demand of a new flow. These migrated flows will also occupy the paths which plans to allocate to latter new flows, which may be lead to loop migration. Therefore, it is not hard to understand that the amount of migrated traffic decrease as the average size of new flows increases in the Fat-Tree network and the mutation of traffic when the average size of new flows is 30 MB in the ER network.

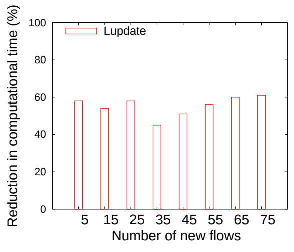
Whatever network, the bigger average size of new flows indeed generates more traffic migration to some extent. Fortunately, the result shows that the *Lupdate* method always migrates the least amount of existing traffic while the *Shortest-path* method migrates the most existing traffic, so as to accommodate the set of test flows.

6.2.4 Impact of the number of new flows

Fig.8 reports the average amount of migrated traffic, caused by different network update events, i.e. different number of new flows. No matter in the Fat-Tree network with k=8 or the ER random network of the same size, the amount of migrated traffic grows as the increase of the number of new flows. That is, if more new flows want to be injected into the network, more existing flows usually need to be rescheduled. Here, we only report the evaluation result of our *Lupdate* method. Note that the *Lupdate-S* method randomly migrates multiple flows from the candidate set when the method of *Lupdate* loses efficacy. Therefore, the

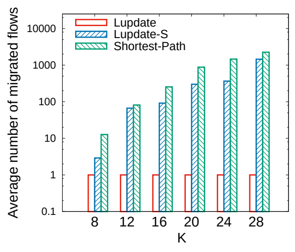


(a) Average migrated traffic.

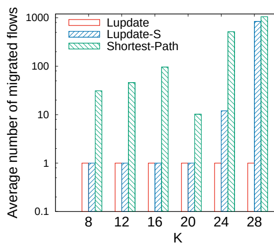


(b) Computation time.

Fig. 9. Reduction in two metrics with our Lupdate against simplified zUpdate under different number of new flows, where the network utilization is 40% in the 8-Pod Fat-Tree network.



(a) Fat-Tree DCN scenario



(b) ER random network scenari-o

Fig. 10. The number of migrated flows in various network size (with y-axis log scale).

change trend of the amount of migrated traffic exhibits uncertainty, due to the introduction of randomness.

In addition, we compute the optimal solution under each setting of the number of new flows. The amount of traffic migration caused by the *Lupdate* method is 1.2 times and 1.12 times of the optimal result in the Fat-Tree network and the ER random network, as shown in Fig.8(a) and Fig.8(b) respectively. The reason for the gap is that *Lupdate* only selects one shortest path randomly for the new flow. However, there usually exist multiple shortest paths and more longer paths for any flow in the two kinds of datacenter networks. The least amount of migrated traffic caused by a new flow may appear at other paths except the selected shortest path. In fact, we will install rules on more switches if we choose a longer path for a new flow, which will cause more cost to the network. Moreover, to derive the optimal solution for each flow will cause serious time-consumption. For this reason, we prefer to utilize the proposed *Lupdate* method, due to its low time complexity and near-optimal performance.

Fig. 9 reports the reduction in the average amount of migrated traffic and the computation time of our method *Lupdate* against simplified *zUpdate* method. *Lupdate* achieves a stable reduction by $5\% - 25\%$ in the amount of migrated traffic and $45\% - 65\%$ in the computation time. For *Lupdate*, it tries to migrate the minimum amount of existing traffic if necessary, whose data rate is near to the bandwidth demand of the new flow. However, simplified *zUpdate* method globally adjusts the routing paths of all new flows to satisfy the bandwidth demand of each new flow. In practice, less number of bottleneck links seriously limits the accommodation of new flows if not migrating several existing flows on bottleneck links. The experimental results prove that simplified *zUpdate* method has to migrate more traffic and take much computation time to calculate the update plan, irrespective of the number of new flows caused by an update event.

6.3 The number of migrated flows

6.3.1 Impact of the network size

Apart from the amount of migrated traffic, the number of migrated flows also has significant negative impact on the production services on IaaS data center. If lots of flows need to be migrated, we have to allocate appropriate paths with adequate link capacities for some existing flows. In addition, we should schedule some existing flows in order to avoid congestion. Then, the new forwarding rules for each flow will be installed on those involved switches. Additionally, each involved switch consumes more time to install forwarding rules for more new flows. It is obvious that the loads on the switches will aggravate with increasing number of migrated flows. In this case, it is very difficult to just migrate a few existing flows to accommodate a set of new flows.

As reported in Fig.10(a), the link utilizations of majority of links constantly improve as the expansion of the network scale. The number of migrated flows increases using the method of *Lupdate-S* and shortest-Path, but *Lupdate* migrates one existing flow all the time due to the particularity way. As expected, the *Lupdate* method still causes the least number of migrated flows for accommodating the same set of new flows, comparing with the *Lupdate-S* and *Shortest-Path* methods in the Fat-Tree datacenter networks. The *Shortest-Path* method migrates too many existing flows, about $20 \sim 40$ times than the *Lupdate-S* method.

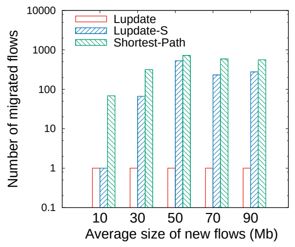
6.3.2 Impact of the topology

We further evaluate the impact of the ER random network on the three update methods, in terms of the number of migrated flows. As shown in Fig.10(b), the number of migrated flows grows up as the increase of the network size except $k = 20$. The reason is that the majority of links exhibit relative low utilization when the k is 20. Thus, it is easier to accommodate a set of new flows and migrate a few existing flows than the scenario of a majority of high utilization links.

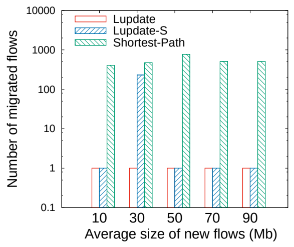
However, such a phenomenon does not happen in the Fat-Tree datacenter networks, with the same settings of switches and flows in the network. The reason is that the link utilization of Fat-Tree network does not appear drastically change.

6.3.3 Impact of the average size of new flows

Fig.11 shows the number of migrated existing flows for accommodating a set of new flows, in the settings of both Fat-Tree datacenter network and ER random network. We can see that our *Lupdate* method migrates one flow to accommodate each new flow, irrespective of the average size of new flows and the underling datacenter network topologies. Obviously, the trend of results under the *Lupdate-S* and *Shortest-Path* methods are similar to that as shown in Fig.7. We can derive from Fig.11(a) that the number of migrated flows grows up as the increasing of average size of new flows, except for the points of 70 MB and 90 MB.

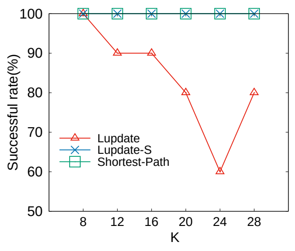


(a) Fat-Tree DCN scenario when k is 20.

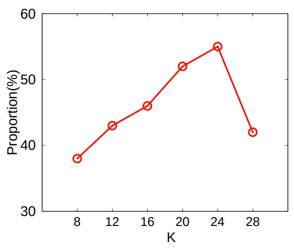


(b) ER random network scenario with 720 switches.

Fig. 11. The number of migrated flows in various average size of new flows (with y-axis log scale). The proportion of links (the link utilizations exceeds 90%) is 51.8% in the Fat-Tree DCN and 54.7% in the ER random network.



(a) Fat-Tree DCN scenario



(b) Proportion of links whose utilization exceeds 90%

Fig. 12. Success rate of network update events, under various network size on the Fat-Tree topology. The proportion metric refers those links whose link utilization exceeds 90%.

The behind reason is similar since the paths allocated to the new flow and those migrated exiting flows will affect the rescheduling results of latter new flows. It will lead to the increase of the number of migrated flows, causing imparity amplitude changes. On the contrary, the *Lupdate* method always migrates one flow since it just permits to migrate at most one existing flow for accommodating a new flow.

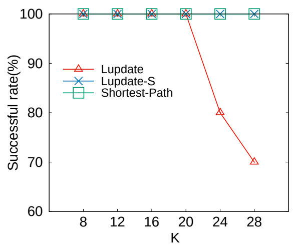
6.4 Success rate of network update

6.4.1 Impact of the network size

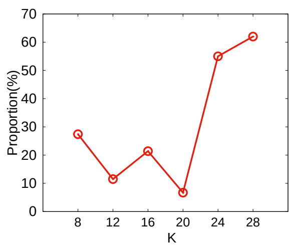
In reality, migrating at most one existing flow may not satisfy the bandwidth demand of a new flow. Thus, the *Lupdate* method may not timely perform an update requirement in rare case.

For this reason, we evaluate the success rate of network update under the Fat-Tree datacenter network. We can see from Fig.12(a) that the success rate of our *Lupdate* method, decreases as the increase of the network scale. However, there appears a turning point with $k = 24$, the success rate begins to rise after that point. The reason is that the proportion of high utilization links begins to decrease after k exceeds 24 as shown in the Fig.12(b). As the proportion of high utilization links changes, the success rate of our *Lupdate* method changes in the opposite direction.

Thus, we can conclude that the network scale and link utilization jointly dominate the success rate of an update event if using the *Lupdate* method. Fortunately, the success rate of the *Shortest-Path* and our *Lupdate-S* methods always keep on 100%, irrespective of the value of k and the proportion of high utilization links. It is clear that our method



(a) ER random network scenario



(b) Proportion of links whose utilization exceeds 90%

Fig. 13. Success rate of network update events, under various network size on the ER random topology. The proportion metric refers those links whose link utilization exceeds 90%.

Lupdate-S can successfully tackle the problem faced by our *Ludapte* and limit the impact of the network scale and link utilization dramatically.

6.4.2 Impact of the topology

As reported in Fig.12(a) and Fig.13(a), we evaluate the success rate of different update methods in the Fat-Tree datacenter network and ER random networks, so as to study the impact of other topologies on the update performance. The successful rate and utilizations of links change in the opposite direction as described in the Fig. 12 and Fig. 13. As shown in Fig.13(a), the success rate of the *Ludpate* method keeps on 100%, when k is less than 20. After k exceeds 20, the success rate begins to decrease. Comparing with Fig 13(b), the link utilization of the majority of link stably changes within a certain scope when k does not exceed 20. Thus, the success rate of *Lupdate* keeps on a certain key. when k is greater than 20, however, link utilization grows rapidly, which would inevitably make *Ludpate* method update results failure. However, the success rate of our improved method *Lupdate-S* and initial method *Shortest-Path* always keep on 100%, irrespective of the value of k and the link utilization.

We can defer from such results that the success rate of our *Lupdate* method depends on the setting of k and the link utilization under different network topologies of data centers. The improved method *Ludpate-S*, however, always keep near 100% success rate.

7 Conclusion

The IaaS datacenter network is consistently undergoing changing, due to a variety of update scenarios. Each update should be executed carefully to disrupt existing flows of critical applications. In this paper, we characterize typical network update scenarios as the reschedule problem of a set of flows, a typical objective optimization problem. Then, we present two simple update methods, *Lupdate* and *Lupdate-S*, to minimize the amount of flow migrations during the process of congestion-free network update. The basic idea is to locally schedule each new flow into a shorter path at the cost of migrating the least amount existing flow if necessary. The difference between them is that if the migration of one flow cannot meet the link bandwidth demands of the new flow, *Lupdate-S* can additionally migrate multiple flows until meet the requirements of the new flow. We conduct large-scale trace-driven evaluations under widely used Fat-tree data center network and the ER random network. The experimental results indicate that our methods can realize congestion-free network update with as less amount of traffic migration as possible even when the proportion links with high utilization is very high. The amount of traffic migration caused by our *Ludpate* method is 1.2 times and 1.12 times of the optimal result in the Fat-Tree and ER random networks, respectively. The simplicity of *Lupdate* and *Lupdate-S* make them be applicable in a variety of networks.

Acknowledgments

This work is partly supported by the National Natural Science Foundation for Outstanding Excellent young scholars of China under Grant No.61422214, National Basic Research Program (973 program) under Grant No.2014CB347800, National Natural Science Foundation of China under Grant No.61661015, U1536202, 61571352, and 61373173, the Program for New Century Excellent Talents in University, the Hunan Provincial Natural Science Fund for Distinguished Young Scholars under Grant No.2016JJ1002, the Research Funding of NUDT under Grant Nos.JQ14-05-02 and ZDYYJ-CYJ20140601.

References

[1] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz, "zupdate: updating data center networks with zero loss," in *Proc. ACM SIGCOMM*, Hong Kong, 2013.
[2] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. ACM SIGCOMM*, Helsinki, Finland, 2012.
[3] M. Rick, "A safe, efficient update protocol for openflow networks," in *workshop on Hot topics in software defined networks*, 2012.
[4] C. Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM*, Hong Kong, 2013.
[5] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: experience with a globally-deployed software defined wan," in *Proc. ACM SIGCOMM*, HongKong, 2013.
[6] M. A. Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *USENIX*, 2010.
[7] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," in *Proc. ACM SIGCOMM*, Toronto, Ontario, Canada, 2011.
[8] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *Proc. ACM SIGCOMM*, Chicago, Illinois, CA, 2014.
[9] J. Guo, F. Liu, J. C. S. Lui, and H. Jin, "Fair network bandwidth allocation in iaas datacenters via a cooperative game approach," *IEEE/ACM Trans. Netw.*, vol. 24, pp. 873–886, 2016.
[10] J. Guo, F. Liu, X. Huang, J. C. S. Lui, M. Hu, Q. Gao, and H. Jin, "On efficient bandwidth allocation for traffic variability in datacenters," in *INFOCOM*, 2014.
[11] J. Esch, "Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions," *Proceedings of the IEEE*, vol. 102, pp. 7–10, 2014.
[12] F. Xu, F. Liu, L. Liu, H. Jin, B. Li, and B. Li, "iaware: Making live migration of virtual machines interference-aware in the cloud," *IEEE Trans. Computers*, vol. 63, pp. 3012–3025, 2014.
[13] F. Xu, F. Liu, and H. Jin, "Heterogeneity and interference-aware virtual machine provisioning for predictable performance in the cloud," *IEEE Trans. Computers*, vol. 65, pp. 2470–2483, 2016.
[14] A. Ludwig, M. Rost, D. Foucard, and S. Schmid, "Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies," in *HotNets*, Los Angeles, California, USA, 2014.
[15] S. Ghorbani and B. Godfrey, "Towards correct network virtualization," in *Proc. ACM HotSDN*, Chicago, IL, 2014.
[16] C. Marco, K. Petr, L. Dan, and S. Stefan, "Software transactional networking: Concurrent and consistent policy composition," in *Proc. ACM HotSDN*, Hong Kong, 2013.
[17] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow, "Rsvp-te: extensions to rsvp for lsp tunnels," 2001.
[18] M. Ratul and W. Roger, "On consistent updates in software defined networks," in *Proc. ACM HotNets*, 2013.
[19] S. Raza, Y. Zhu, and C.-N. Chuah, "Graceful network state migrations," *IEEE/ACM Transactions on Networking (TON)*, vol. 19, no. 4, pp. 1097–1110, 2011.
[20] A. Noyes, T. Warszawski, P. Černỳ, and N. Foster, "Toward synthesis of network updates," *arXiv preprint arXiv:1403.7840*, 2014.
[21] F. Pierre, B. Olivier, D. Bruno, and C. P-A, "Avoiding disruptions during maintenance operations on bgp sessions," *IEEE Transactions on Network and Service Management*, vol. 4, pp. 1–11, 2007.
[22] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE transactions on Computers*, vol. 100, pp. 892–901, 1985.
[23] E. Paul and A. Rényi, "On the evolution of random graphs," *Publ. Math. Inst. Hungar. Acad. Sci*, vol. 5, pp. 17–61, 1960.
[24] Y. Chen, S. Jain, V. K. Adhikari, Z.-L. Zhang, and K. Xu, "A first look at inter-data center traffic characteristics via yahoo! datasets," in *Proc. INFOCOM*, Orlando, FL, 2011.
[25] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. ACM SIGCOMM*, New Delhi, 2010.

Ting Qu received the B.S. degree in computer science from Xidian University, Xian, China, in 2014. She is currently working toward the Ph.D. degree in College of Information System and Management, National University of Defense Technology, Changsha, China. Her research interests include data centers and software defined networks.

DekeGuo received the B.S. degree in industry engineering from Beijing University of Aeronautic and Astronautic, Beijing, China, in 2001, and the Ph.D. degree in management science and engineering from National University of Defense Technology, Changsha, China, in 2008. He is a Professor with the College of Information System and Management, National University of Defense Technology, Changsha, China. His research interests include distributed systems, software-defined networking, data center networking, wireless and mobile systems, and interconnection networks. He is a member of the ACM and the IEEE.

Yulong Shen received the BS and MS degrees in computer science and the PhD degree in cryptography from Xidian University, Xian, China, in 2002, 2005, and 2008, respectively. He is currently a professor at the School of Computer Science and Technology, Xidian University, China. He is also an associate director in the Shanxi Key Laboratory of Network and System Security and a member in the State Key Laboratory of Integrated Services networks Xidian University, China. He has also served on chair or technical program committees of several international conferences, including NANA, ICEBE, INCoS, CIS, and SOWN. His research interests include wireless network security and cloud computing security.

Xiaomin Zhu received the BS and MS degrees in computer science from Liao ning Technical University, Liao ning, China, in 2001 and 2004, respectively, and Ph.D. degree in computer science from Fudan University, Shanghai, China, in 2009. In the same year, he won the Shanghai Excellent Graduate. He is currently an associate professor in the College of Information Systems and Management at National University of Defense Technology, Changsha, China. His research interests include scheduling and resource management in green computing, cluster computing, cloud computing, and multiple satellites. He has published more than 50 research articles in refereed journals and conference proceedings such as IEEE TC, IEEE TPDS, JPDC, JSS and so on. He is also a frequent reviewer for international research journals, e.g., IEEE TC, IEEE TNSM, IEEE TSP, JPDC, etc. He is a member of the IEEE, the IEEE Communication Society, and the ACM.

Lailong Luo received the B.S. and M.S. degree in school of information system and management from National University of Defence Technology, Changsha, China, in 2013 and 2015 respectively. He is currently working toward the Ph.D. degree in College of Information System and Management, National University of Defense Technology, Changsha, China. His research interests include data centers and software defined networks.

zhong Liu received the B.S. degree in Physics from Central China Normal University, Wuhan, Hubei, China, in 1990, and the Ph.D. degree in management science and engineering from National University of Defense Technology, Changsha, China, in 2000. He is a professor with the College of Information System and Management, National University of Defense Technology, Changsha, China. His research interests include information systems, cloud computing, and big data.