Future Generation Computer Systems 97 (2019) 743–754

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Fast congestion-free consistent flow forwarding rules update in software defined networking

Pan Li [a], Songtao Guo [b,a,*], Chengsheng Pan [c,**], Li Yang [c], Guiyan Liu [a], Yue Zeng [a]

[a] Chongqing Key Laboratory of Nonlinear Circuits and Intelligent Information Processing, College of Electronic and Information Engineering, Southwest University, Chongqing, 400715, China
[b] College of Computer Science, Chongiqng University, Chongqing, 400044, China
[c] College of Information Engineering, Dalian University, Dalian, 116622, China

HIGHLIGHTS

- We analyze how to avoid black holes, loops and transient congestion, and build the corresponding avoidance models.
- We propose the black holes avoidance algorithm, loops avoidance algorithm and congestion avoidance algorithm.
- We propose a RU algorithm that combines these three algorithms to update the flows rules.
- Simulation results show that our scheme can increase the number of directly updated flows by 75% on a single congestion link and reduce the update time of the flows by 34%.

ARTICLE INFO

Article history:
Received 26 July 2018
Received in revised form 16 January 2019
Accepted 16 March 2019
Available online 19 March 2019

Keywords:
Flow migration
Transient congestion
Update consistency
Dependency graph
Software defined networking (SDN)

ABSTRACT

In software defined networking (SDN), flow migration will be required when topology changes to improve network performance such as load balancing. However, black holes, loops and transient congestion may occur during flow migration due to the asynchronous update of switches on the data plane. Therefore, in this paper, we propose a novel segmented update method to shorten the time of rules update, and a novel transient congestion avoidance algorithm to minimize the number of delayed updating flows, which can both reduce update time of flows. Specifically, we construct three novel models to guarantee no black holes, no loops and no transient congestion, respectively. The first two models to avoid black holes and loops can update multiple nodes in each segment instead of updating the nodes one by one like Cupid. The third model to avoid transient congestion minimizes the number of delayed updating flows. Subsequently, three novel black holes avoidance algorithm, loops avoidance algorithm and congestion avoidance algorithm are respectively proposed. Furthermore, we propose a novel rules update (RU) algorithm which combines these three algorithms to update the rules to avoid black holes, loops and transient congestion simultaneously. Simulation results show that our scheme can increase the number of directly updated flows by 75% on a single congestion link and reduce the rules update time of the flows by 34% compared with the existing work.

1. Introduction

Software defined networking (SDN) composed of both control plane and data plane achieves the forwarding of packets according to the rules installed in the flow tables in the switches [1]. In SDN, a logically centralized controller in the control plane has a global view of the network state and is responsible for instructing the switches on the data plane to add, modify, or delete rules in their flow tables. Due to the flexibility of SDN in network management, it brings tremendous advantages to various networks [2]. For example, many studies have proved that SDN can improve link utilization [3–6] and reduce energy consumption [7–10].

In SDN, although the control plane is centralized, the data plane is still a distributed system. When traffic or network topology changes, the controller needs to recalculate paths for flows to optimize network performance. During flow migration, the update of flow forwarding rules in the data plane may be inconsistent due to asynchronous update in switches. The inconsistent update can cause black holes (i.e., a switch has no rule to forward a packet when the packet arrives at the switch), loops, and transient congestion (i.e., the new flows move into the link before

* Corresponding author at: College of Computer Science, Chongiqng University, Chongqing, 400044, China.
** Corresponding author.
E-mail addresses: songtao_guo@163.com (S. Guo), pancs@sohu.com (C. Pan).

https://doi.org/10.1016/j.future.2019.03.030
0167-739X/© 2019 Elsevier B.V. All rights reserved.

the original flows are moved out the link, and the total traffic on the link exceeds the link capacity) during flow migration [11,12]. Unfortunately, the occurrence of black holes, loops, and transient congestion can cause temporary interruption of flows and reduction of throughput.

Currently, many methods have been proposed to avoid black holes and loops, such as two-phase update [13], time-triggered update [14,15], reverse order update [16], the update of combined with rule replacement and addition [12,17,18], and segmentation reverse order update [19]. Meanwhile, there are also some ways to avoid transient congestion, such as introducing intermediate stages [5,20,21], building rule update dependency graphs [2,19] and time triggering [22,23]. But almost all previous works [2, 5,12–18,20,21] either focused on how to avoid black holes and loops, or on how to avoid transient congestion. In [19], the Cupid algorithm is designed to avoid black holes, loops, and transient congestion simultaneously. However, the algorithm consumes a lot of time to avoid black holes and loops by segmentation reverse order update. Besides, it generates considerable cost to avoid transient congestion by delaying the update of all flows that are moved into potential congestion links.

Therefore, in this paper, we propose a novel rules update (RU) algorithm consisting of black holes avoidance algorithm, loops avoidance algorithm and congestion avoidance algorithm, which reduces the time consumed for rules update while avoiding black holes, loops, and transient congestion simultaneously. Specifically, we first identify the segmented nodes which control the flow to switch paths and then divide the new path of each flow into several segments based on the segmented nodes. To avoid black holes and loops during updating, we first update the nodes except the segmented nodes in each segment simultaneously and then update the segmented nodes. To avoid transient congestion during updating, we minimize the number of delayed updating flows and then build a dependency graph with potential congestion links. Therefore, the RU algorithm reduces the search overhead of the rule update dependency graph and the queue length of the rule update compare with the Cupid algorithm [19], thereby reducing the time taken to update the rules.

The main contributions of this paper are as follows:

- We analyze how to avoid black holes, loops and transient congestion, and build three corresponding avoidance models. The first two models to avoid black holes and loops can update multiple nodes in each segment instead of updating the nodes one by one like Cupid [19]. The third model to avoid transient congestion minimizes the number of delayed updating flows.
- We propose the black holes avoidance algorithm, loops avoidance algorithm and congestion avoidance algorithm to guarantee no black holes, no loops and no transient congestion during the rules update of flows, respectively. These three algorithms greatly reduce the queuing time required for the rules update of flows. Furthermore, we propose a RU algorithm that combines these three algorithms to update the rules to avoid black holes, loops and transient congestion simultaneously.
- We evaluate our algorithm in DCN and WAN topologies. Simulation results show that our scheme can increase the number of directly updated flows by 75% on a single congestion link and reduce the rules update time of the flows by 34% compared with Cupid [19].

The remainder of this paper is organized as follows. Section 2 discusses the related work for routing updates in SDN and Section 3 introduces the motivation of our work. Then, Section 4 describes how to avoid black holes and loops during rules update, and Section 5 details how to avoid transient congestion during rules update. And in Section 6, we introduce a heuristic algorithm that simultaneously avoids black holes, loops, and transient congestion. Finally, we evaluate our methods and compare them with other existing methods in Section 7 and Section 8 concludes this paper.

2. Related work

In this section, we will review the previous methods of network routing updating in SDN and divide these methods into two parts based on their concerns: how to avoid black holes and loops during flow migration and how to avoid transient congestion during flow migration due to asynchronous update of switches in the SDN data plane.

On the one hand, some of the research works [12–19,24–26] focused on how to avoid black holes and loops during flow migration. Reitblatt et al. [13] proposed a two-phase commit protocol to update the rules, which avoids black holes and loops, but this approach increases TCAM overhead. Thus, several update methods were proposed to reduce the overhead of TCAM, such as the reverse order update method [16], the time triggered method [14,15] and the rule redundancy reduction method [24]. As an improved version, a method of combining rule replacements and additions is considered to reduce the overhead of the TCAM and reduce the delay of the reverse order update [12,17, 18]. Based on the reverse order update [16], Cupid [19] proposed a method of segmentation reverse update to reduce the time consuming of rule update, and Firster et al. [26] emphasized the trade-off between the strength of the consistency property and the dependencies. In addition, Basta et al. [25] studied how to minimize the number of interactions between switches and controllers when avoiding loops.

On the other hand, some of the research works [2,5,19–23,27, 28] focused on how to avoid transient congestion during flow migration. SWAN [5] and zUpdate [20] introduced an intermediate stage in WAN and DCN to avoid transient congestion during updating. However, these methods require to solve a series of Linear Programs (LPs) which is time consuming. Therefore, some update methods were proposed to improve link utilization and reduce the time for intermediate state transitions, such as the congestion-minimizing method [21], the time-triggered method [22,23] and the dynamic scheduling method [2]. Based on the dynamic scheduling method [2], Cupid [19] divided the global rule update dependencies among switches into local restrictions to reduce the time for rule update, and Luo et al. [27] proposed a method of calculating the network update plan according to the user's requirements. In addition, Xu et al. [28] solved the low-latency routing update challenge by jointly optimizing routing and updating scheduling.

Almost all of the previous works either focused on how to avoid black holes and loops, or on how to avoid transient congestion and the work [19] designed different algorithms to avoid black holes, loops, and transient congestion simultaneously. However, since this method increases unnecessary update sequences and does not consider minimizing the queuing time of all flows, it may still result in longer routing delays. If the route update delay is too long, the final routing configuration may be inefficient for the updated workload [28]. Therefore, we need to propose a more efficient low-latency update algorithm to avoid black holes, loops, and transient congestion simultaneously.

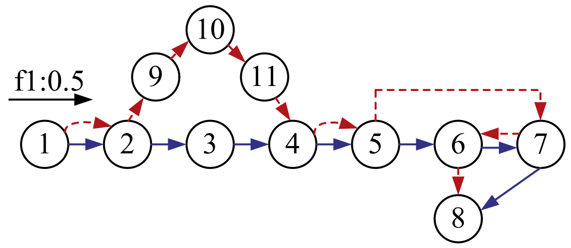


Fig. 1. An example of route update. The link capacity is 1 unit and flow is labeled with its size. The blue arrow and the red dotted arrow indicate the old path and the new path of the flow $f1$ respectively.

3. Illustrate for rules update

In this section, we will illustrate the concepts of black holes, loops and transient congestion by giving examples. Then we introduce the Cupid algorithm how to avoid black holes, loops and transient congestion.

In SDN, when the topology changes or it needs to balance the load, the controller will plan new paths for affected flows, that is, the flows will be migrated from the current paths to the new paths. However, black holes and loops may occur during flow migration due to the asynchronous update of switches. As shown in Fig. 1, there is a flow $f1$ with size of 0.5 units, passing the path: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8$. In order to release more available bandwidth for other flows on link $2 \rightarrow 3$, the controller recalculates a new path for flow $f1$: $1 \rightarrow 2 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 6 \rightarrow 8$. During the migration of flow $f1$, if the rule in switch 2 is updated but the rule in switch 9 has not been updated, then the packets sent to switch 9 will be buffered or even dropped. In this case switch 9 is a black hole. In addition, if the rule in switch 7 is updated but the rule in switch 6 has not been updated, then a loop $7 \rightarrow 6 \rightarrow 7$ may occur.

In order to ensure that there are no black holes and no loops during flow migration, Cupid [19] divides new path into multiple segments according to the critical nodes, and then updates the nodes in each segment by reverse order one by one. As shown in Fig. 1, in order to avoid black holes and loops during migration, by Cupid [19] algorithm, the new path of flow $f1$ will be divided into five segments: $1 \rightarrow 2$, $2 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 4$, $4 \rightarrow 5$, $5 \rightarrow 7$, $7 \rightarrow 6 \rightarrow 8$. And then the five segments are updated in reverse order, for example, in the second segment, nodes 4, 11, 10, 9, and 2 are updated one by one, and need to be updated five times. Therefore, the time required to update the rules for the new path of flow $f1$ is $max\{(t_1+t_2), (t_2+t_9+t_{10}+t_{11}+t_4), (t_4+t_5), (t_5+t_7), (t_7+t_6+t_8)\}$ ms.

However, it is worth noting that as long as the rules in switches 9, 10, 11 are updated before the rule update in switch 2, black holes can be avoided during flow f1 migration. And as long as the rule in switch 6 is updated before the rule update in switch 7, the loop $7 \rightarrow 6 \rightarrow 7$ can be avoided. In other words, there is a new segment update method to avoid black holes and loops, which can update multiple nodes at the same time. With this new segment update method, the time required to update the forwarding rules of flow $f1$ is $max\{(t_2+max(t_9, t_{10}, t_{11}, t_4)), (t_5), (t_7+max(t_6, t_8))\}$ ms. The results are shown in Table 1 and we can observe that the time to update the rules by the new segment update method can be greatly reduced compared to Cupid.

Transient congestion may also occur during flow migration due to asynchronous update of the rules on switches. As shown in Fig. 2(a), there are 5 flows from switch 1 to switch 3 using $1 \rightarrow 2 \rightarrow 3$ and $1 \rightarrow 4 \rightarrow 3$ respectively. In order to release more available bandwidth for other flows on link $1 \rightarrow 2$, these flows need to be migrated from the old paths (a) to the new paths (b) in Fig. 2. We can see that if the flows $f3, f4$ and $f5$ are moved into link $1 \rightarrow 2$ before the flows $f1$ and $f2$ leave the link $1 \rightarrow 2$, it will cause transient congestion on link $1 \rightarrow 2$.

In order to avoid transient congestion, Cupid [19] proposes a congestion avoidance method, which first updates the flows that need to move out of the potential congestion link to release the bandwidth resources, and then updates the flows that need to be moved into the potential congestion link. Therefore, in order to avoid transient congestion on the link $1 \rightarrow 2$ in Fig. 2, by Cupid algorithm [19], flows $f1$ and $f2$ are first updated, and then flows $f3, f4$, and $f5$ are updated.

However, note that the total bandwidth demand of four flows $f1, f2, f3$, and $f4$ does not exceed the link capacity. If flows $f1, f2, f3$, and $f4$ are updated first and then flow $f5$ is updated, then the update delay of flows $f3$ and $f4$ is reduced. Therefore, as long as the traffic does not exceed the link bandwidth, some of the flows that need to be moved into the potential congestion link can be updated simultaneously with the flows that need to be moved out. In this way, the update time of some flows that need to be moved into the potential congestion link is shortened.

Therefore, in the process of avoiding black holes and loops, we propose a more efficient method, which can update multiple nodes at the same time, further reducing the time of rules update. In the process of avoiding transient congestion, the number of delayed updating flows is minimized, that is, the partial flow that needs to be moved into the potential congestion link can be updated in advance, thereby the rules update time of the flows is reduced.

4. Our black holes and loops avoidance algorithm

In this section, we propose algorithms to avoid black holes and avoid loops, and illustrate the algorithms by examples.

4.1. Black holes avoidance algorithm

Considering the inefficiency of reverse order updating, we propose a new method of avoiding black holes by two steps, (i) dividing the different parts of the new path and the old path (i.e., the path before update) into independent segments, and (ii) making the update of the first node later than other nodes in the segment.

In order to find the difference between the new path and the old path of a flow f, we need to identify the segmented nodes which control the flow f to switch paths. The segmented node sn_f is the common node between the new path $P_{new}(f)$ and the old path $P_{old}(f)$, but the next hop $nh(sn_f, P_{new}(f))$ of the common node in the new path is different from the next hop $nh(sn_f, P_{old}(f))$ of the common node in the old path, as shown in Eq. (1).

$$\begin{aligned} SN(f) = \{&sn_f | sn_f \in P_{old}(f) \cap P_{new}(f), \\ &nh(sn_f, P_{old}(f)) \neq nh(sn_f, P_{new}(f))\} \end{aligned} \tag{1}$$

After obtaining the segmented nodes $SN(f)$, in order to avoid black holes during the flow f migration, we first need to divide the new path $P_{new}(f)$ of flow f into several segments $S(f)$ according to the segmented nodes $SN(f)$, as shown in Eq. (2). The start node $s[0]$ of each segment $s \in S(f)$ is the segmented node sn_f or the start node $P_{new}(f)[0]$ of the new path, and the end node $s[length-1]$ of each segment is the precursor node $ps(f)$ of the segmented node in the new path or the destination node $P_{new}(f)[length-1]$ of the new path, and there is no same node between segments.

$$\begin{aligned} S(f) = \{&s \mid s \subseteq P_{new}(f), s[0] \in SN(f) \cup P_{new}(f)[0], \\ &s[length-1] \in PS(f) \cup P_{new}(f)[length-1] \\ &\forall s' \in S(f), s \cap s' = \emptyset\} \end{aligned} \tag{2}$$

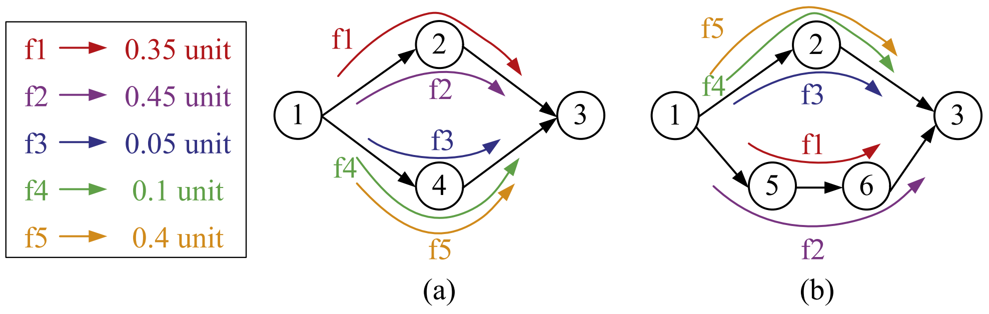


Fig. 2. An example of route update. The link capacity is 1 unit, and the legend shows the size of flows. (a) and (b) represent the old paths and the new paths of flows $f1$, $f2$, $f3$, $f4$ and $f5$ respectively.

Table 1
Motivation summary.

Algorithm	The time required by updating rules of flow f1	The number of directly updated flows
Cupid algorithm	$max\{(t_1+t_2),(t_2+t_9+t_{10}+t_{11}+t_4),(t_4+t_5),(t_5+t_7),(t_7+t_6+t_8)\}$	2
Our RU algorithm	$max\{(t_2+max(t_9,t_{10},t_{11})),(t_5),(t_7+t_6)\}$	4

Then, based on the obtained segments, we construct an update dependency graph $d(s)$ in each segment s to avoid black holes. The update of the first node $s[0]$ is later than that of other nodes $\{s-s[0]\}$ in the segment, as shown in Eq. (3).

$$d(s) = s[0] \rightarrow \{n|n \in s - s[0]\} \tag{3}$$

We design an algorithm to avoid black holes by constructing black-holes-free dependency graph D for each flow f in Algorithm 1. The main processes of the algorithms are as follows. First, we find the segmented nodes by traversing the nodes in the new path of flow f. If the node in the new path $P_{new}(f)$ has appeared in the old path $P_{old}(f)$, and the next-hop of this node in the new path is different from that in the old path, then the node will be considered as a segmented node sn_f (lines 1–10). Then we segment the new path of flow f according to the segmented nodes. Finally, based on the obtained segments, we construct an update dependency graph $d(s)$ in each segment s to avoid black holes (line 19). For example, as shown in Fig. 1, the segmented nodes of flow $f1$ are node 2, node 5, node 7 and node 6. Then according to the segmented nodes, we segment the new path of flow $f1$, which can be divided into five segments: 1, $2 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 4$, 5, 7, $6 \rightarrow 8$. Obviously, in Fig. 1, the dependency graphs of the nodes update in these segments are: $\{1\}$, $\{2\} \rightarrow \{9, 10, 11, 4\}$, $\{5\}$, $\{7\}$, $\{6\} \rightarrow \{8\}$.

4.2. Loops avoidance algorithm

In order to avoid loops during flow migration, we first need to find the loops $Loop(f)$ formed by the old path and the new path. Then, we construct a dependency graph $d(loop)$ to update the nodes in each loop $loop(f) \in Loop(f)$, as shown in Eq. (4). The update of first loop node $loop(f)[0]$ that appears in the new path is later than that of other nodes $\{loop(f) - loop(f)[0]\}$ in the loop.

$$d(loop) = loop(f)[0] \rightarrow \{n|n \in loop(f) - loop(f)[0]\} \tag{4}$$

We design an algorithm to avoid loops by building a loops-free dependency graph for each flow f in Algorithm 2. The main processes of the algorithms are as follows. First, we calculate the loops $Loop(f)$ formed by the old path and the new path based on the concept of strong connected components [19]. We use Tarjan's algorithm [29] to efficiently find all loops in the directed graph formed by the old path and the new path (line 2). Its time complexity is $O(n + l)$, where n and l are the number of nodes and edges in the network, respectively. Then, we construct a dependency graph $d(loop)$ to avoid loops (lines 3–6). For example, as shown in Fig. 1, the new path and old path of flow $f1$ form a loop $7 \rightarrow 6 \rightarrow 7$, and then the loops-free dependency graph is $\{7\} \rightarrow \{6\}$ in Fig. 1.

Algorithm 1 Black holes Avoidance Algorithm

```
Input: flows $f$, old path $P_{old}(f)$, new path $P_{new}(f)$
Output: segmented nodes $SN(f)$ and black-holes free dependency graph $D$
1: $SN(f) = D = \emptyset$;
2: for $i : i \in P_{new}(f)$ do
3:   $j$ = the next-hop of $i$ on $P_{new}(f)$ ;
4:   if $i \in P_{old}(f)$ then
5:     $k$ = the next-hop of $i$ on $P_{old}(f)$ ;
6:   end if
7:   if $j \neq k$ then
8:     $sn_f = j$, $SN(f) = SN(f) \cup sn_f$;
9:   end if
10: end for
11: $i = P_{new}(f)[0]$;
12: while $i \in P_{new}(f)$ do
13:   $s = i$;
14:   $j$ = the next-hop of $i$ on $P_{new}(f)$;
15:   while $j \neq \phi \quad \wedge \quad j \notin SN(f)$ do
16:     $s = s \rightarrow j$;
17:     $i = j$, $j$ = the next-hop of $i$ on $P_{new}(f)$;
18:   end while
19:   $d(s) = s[0] \rightarrow \{n|n \in s - s[0]\}$;
20:   $D = D \cup d(s)$;
21:   $i = j$;
22: end while
23: return $SN(f)$, $D$;
```

5. Transient congestion avoidance algorithm

In this section, we will construct a model to avoid transient congestion, and propose a transient congestion avoidance algorithm. Then, we will illustrate the details of the algorithm by

Table 2
Notations.

Symbol	Meaning
d_f	The demand of flow f
$c(l)$	The capacity of link l
$F_{out}(l)$	The set of flows that need to be removed from link l
$F_{un}(l)$	The set of unchanged flows on link l
$F_{in}(l)$	The set of flows that need to be moved into link l
N	The number of flows in $F_{in}(l)$
x_i	Binary variable indicating whether the flow f_i is selected as a delayed updating flow

Algorithm 2 Loops Avoidance Algorithm

```
Input: flow $f$, old path, new path
Output: loops-free dependency graph $D$
1: $D = \emptyset$;
2: calculate loops $Loop(f)$ by strong connected components;
3: for $loop(f) : loop(f) \in Loop(f)$ do
4:     $d(loop) = loop(f)[0] \rightarrow \{n|n \in loop(f) - loop(f)[0]\}$;
5:     $D = D \cup d(loop)$;
6: end for
7: return $D$;
```

examples. For the sake of convenience, we first summarize some notations in Table 2.

5.1. Finding potential congestion links

During flow migration, the total traffic on the link l does not exceed the capacity of link l even before and after updating. However, in the process of updating, due to the asynchronous update of switches, the new flows may move into the link l before the existing flows move out of the link l, which may result in transient congestion. Therefore, in order to avoid transient congestion during updating, we should carefully design a scheduling sequence without congestion.

To avoid transient congestion, we first need to find the potential links that may be congestion. The potential congestion link l needs to satisfy the following three conditions:

(1) Before updating:

$$\sum_{f \in F_{out}(l)} d_f + \sum_{f \in F_{un}(l)} d_f \le c(l) \tag{5}$$

(2) After updating:

$$\sum_{f \in F_{un}(l)} d_f + \sum_{f \in F_{in}(l)} d_f \le c(l) \tag{6}$$

(3) The worst case during updating:

$$\sum_{f \in F_{out}(l)} d_f + \sum_{f \in F_{un}(l)} d_f + \sum_{f \in F_{in}(l)} d_f > c(l) \tag{7}$$

Eq. (5) indicates that before updating, the sum of all flows sizes on link l cannot exceed the link capacity $c(l)$. Eq. (6) means that after updating, the sum of all flows sizes on link l cannot exceed the link capacity $c(l)$. Eq. (7) represents the worst case during updating, i.e., the new flows move into the link l before the existing flows remove from the link l, resulting in the sum of all flows sizes on link l exceed the link capacity $c(l)$.

5.2. Constructing congestion-free dependency graph

To avoid transient congestion, we need to determine which flows on the potential congestion link should be updated first and which flows should be updated subsequently. At the same time, in order to complete the flow migration as soon as possible, it is necessary to minimize the number of delayed updating flows.

We first find the set $F_s(l)$ of flows that needs to be delayed updating on the potential congestion link $l = (u, v)$, as shown in Eq. (9). After $\{x_i\}$ is obtained in Eq. (9), we can further get $\{f_i\}$.

$$x_i = \begin{cases} 1, \text{ if } f_i \text{ is delayed} \\ 0, \text{ otherwise} \end{cases} \tag{8}$$

$$\begin{aligned} F_s(l) = &\{f_i | min \sum_{i=1}^{N} x_i, f_i \in F_{in}(l), x_i \in \{0, 1\} \\ &\sum_{i=1}^{N} d_{f_i} x_i \ge \sum_{f \in F_{out}(l)+F_{un}(l)+F_{in}(l)} d_f - c(l)\} \end{aligned} \tag{9}$$

where x_i is a binary variable, and $x_i = 1$ if flow f_i is selected for delayed updating, otherwise, $x_i = 0$. $\sum_{f \in F_{out}(l)+F_{un}(l)+F_{in}(l)} d_f$ represents the sum of all flows sizes on link l in the worst case during updating. $\sum_{i=1}^{N} d_{f_i} x_i$ represents the sum of sizes of all delayed updating flows. $\sum_{i=1}^{N} d_{f_i} x_i \ge \sum_{f \in F_{out}(l)+F_{un}(l)+F_{in}(l)} d_f - c(l)$ indicates that the sum of sizes of all delayed updating flows is not less than the excess link bandwidth. This means that these delayed updating flows are only updated when there is enough bandwidth, and it can further avoid the congestion on link l.

Furthermore we find the last segmented node ($nf_s(l)$ or $nf_c(l)$) that controls each flow (i.e., the flow belongs to the set $F_s(l)$ or $F_{out}(l) \cup F_{in}(l) \backslash F_s(l)$) to move in or out of the potential congestion link $l = (u, v)$, as shown in Eqs. (11) and (11). The set $F_{out}(l) \cup F_{in}(l) \backslash F_s(l)$ represents the flows that can be updated first on the potential congestion link l. It should be noted that if node u is a segmented node, then the last segmented node that we are looking for is node u.

$$NF_s(l) = \{n_{f_s} | \forall f_s \in F_s(l)\} \tag{10}$$

$$NF_c(l) = \{n_{f_c} | \forall f_c \in F_{out}(l) \cup F_{in}(l) \backslash F_s(l)\} \tag{11}$$

Finally, a congestion-free dependency graph $d(l)$ is constructed between the segmented nodes based on the potential congestion link l. In the graph, the segmented nodes in the set $NF_s(l)$ are updated later than the segmented nodes in the set $NF_c(l)$, as shown in Eq. (12).

$$d(l) = NF_s(l) \rightarrow NF_c(l) \tag{12}$$

The congestion avoidance algorithm is described in Algorithm 3. In lines 1–11 of Algorithm 3, we calculate the maximum link load on each link and find out the potential congestion link CL by determining whether the maximum link load on each link exceeds the link capacity $c(l)$. Then, in lines 12–30, we find out the set $F_s(l)$ of delayed updating flows on the potential congestion link l. Specifically, we first calculate the traffic size $OverBw(l)$ that exceeds the link capacity on potential congestion link l. Next, the flows $F_{in}(l)$ that need to be moved into link l are sorted by their flow sizes (line 14). Then, the flow is selected one by one from the set $F_{in}(l)$ as a delayed updating flow, and the selected flow is added to the set $F_s(l)$ until the sum of the sizes of the flows in the set $F_s(l)$ is not less than $OverBw(l)$ (lines 15–20), i.e., no

Algorithm 3 Congestion Avoidance Algorithm

```
Input: flows $F$, segmented nodes $SN(f)$, old paths $F$, new paths $L$
Output: congestion-free dependency graph $D$
1: $D = \emptyset$;
2: for $l : l \in L$ do
3:     for $f : f \in F$ do
4:         if $l \in P_{old}(f)$ or $l \in P_{new}(f)$ then
5:             $F(l) = F(l) + d_f$;
6:         end if
7:     end for
8:     if $F(l) > c(l)$ then
9:         $CL = CL \cup l$;
10:    end if
11: end for
12: for $l : l \in CL$ do
13:     $OverBw(l) = \sum_{f \in F_{out}(l)+F_{uc}(l)+F_{in}(l)} d_f - c(l)$;
14:     Sort by the size of flow $F_{in}(l)$ in decreasing order;
15:     for $f : f \in F_{in}(l)$ do
16:         $F_s(l) = F_s(l) \cup f$;
17:         if the sum of the flow sizes in $F_s(l) \geq OverBw(l)$ then
18:             break;
19:         end if
20:     end for
21:     for $f_s : f_s \in F_s(l)$ do
22:         find the last segmented node $n_{f_s}$ of $f_s$ before node $v$;
23:         $NF_s(l) = NF_s(l) \cup n_{f_s}$;
24:     end for
25:     for $f_c : f_c \in F_{out}(l) \cup F_{in}(l) \backslash F_s(l)$ do
26:         find the last segmented node $n_{f_c}$ of $f_c$ before node $v$;
27:         $NF_c(l) = NF_c(l) \cup n_{f_c}$;
28:     end for
29:     $d(l) = NF_s(l) \to NF_c(l)$, $D = D \cup d(l)$;
30: end for
31: return $D$;
```

congestion will occur. Finally, in lines 21–30, a congestion-free dependency graph $d(l)$ is constructed to delay update the flows in set $F_s(l)$. For each potential congestion link $l = (u, v)$, we first need to find the last segmented node that controls each flow to move in or out of the link l, and then construct a congestion-free dependency graph $d(l)$ based on these segmented nodes.

As shown in Fig. 2, the potential congestion link CL is link $1 \to 2$, the set $F_s(l)$ of flows that need to wait for the update is $\{f5\}$, and the segmented node that controls the flows $f1, f2, f3, f4, f5$ to move in or out of the potential congestion link is node 1. Then, the dependency graph for determining the update order of flows $f1, f2, f3, f4$, and $f5$ on node 1 is: $\{1_{f5}\} \to \{1_{f1}, 1_{f2}, 1_{f3}, 1_{f4}\}$.

6. Rules update algorithm without black holes, loops and transient congestion

The problem of finding the fastest update schedule in the presence of link capacity constraints is an NP-complete problem, which is proved in [2]. Therefore, in this section, we propose a heuristic rules update algorithm to calculate the rule updating sequence to simultaneously avoid black holes, loops, and transient congestion. Next, we first give the definition of deadlock L as follows [19].

Definition 1 (*Deadlock L*)**.** For a set of potential congestion links $l_0, l_1, l_2, \ldots, l_k$, if $NF_c(l_0) \cap NF_s(l_1) = nf0 \neq \emptyset$, $NF_c(l_1) \cap NF_s(l_2) = nf1 \neq \emptyset$, …, $NF_c(l_k) \cap NF_s(l_0) = nfk \neq \emptyset$, then the sets of segmented nodes $nf0, nf1, \ldots, nfk$ form a deadlock L.

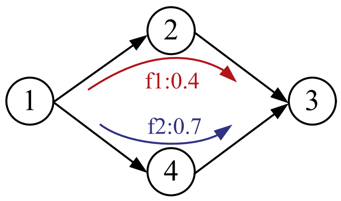


Fig. 3. The paths of flow $f1$ and $f2$. The link capacity is 1 unit and flows are labeled with their sizes.

For example, in Fig. 3, there is a flow $f1$ with size 0.4 unit going through path $1 \to 2 \to 3$ and a flow $f2$ with size 0.7 unit going through path $1 \to 4 \to 3$. In order to release more bandwidth for other flows on link $1 \to 4$, the paths of flow $f1$ and flow $f2$ need to exchange. However, transient congestion may occur when updating flow $f1$ and flow $f2$. To avoid transient congestion, we construct the congestion-free dependency graphs, $1_{f2} \to 1_{f1}$ and $1_{f1} \to 1_{f2}$ for flow $f1$ and $f2$. Therefore, a deadlock $1_{f1} \rightleftarrows 1_{f2}$ is formed.

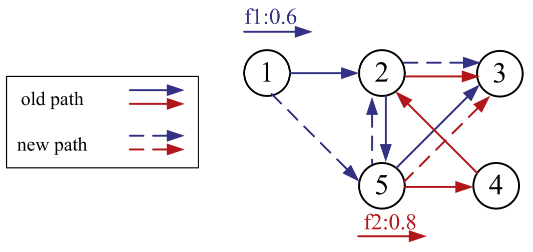
6.1. Algorithm description

We design a novel rules update (RU) algorithm to get a sequence US of rule updates that simultaneously avoid black holes, loops, and transient congestion in Algorithm 4. We first get the rules update dependency graph by Algorithms 1–3 (lines 2–3). Then we traverse the nodes in the dependency graph D. If a node does not have a predecessor node in D, then the node can be put into the update sequence US_i. If a node belongs to a deadlock and has only one predecessor node in D, then the node can be updated. For a deadlock, no flows can be directly updated, so we need to employ multipath transition method to update. For example, for deadlock $1_{f1} \rightleftarrows 1_{f2}$ in Fig. 3, if node 2_{f2} is updated and the predecessor node of node 1_{f2} has only node 1_{f1}, then node 1_{f2} can be updated.

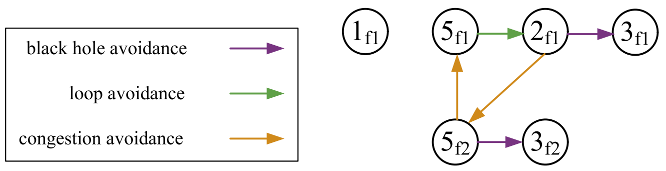
Since there is not enough bandwidth to update flow $f2$, we now need to spilt it at node 1. For flow $f2$, its available bandwidth ab in the new path $1 \to 2 \to 3$ is 0.6 unit, and we will shift 0.6 unit of $f2$ to path $1 \to 2 \to 3$. Then we recalculate the size of updated flow f in the old path and in the new path, and add the node n_f^{new} to the update sequence US_i (lines 13–15). If all the traffic on the old path d_f^{Pold} is finally shifted to the new path, then the node n_f is deleted from the dependency graph D (lines 16–18). Next, the above steps are repeated until the nodes in the dependency graph D are empty.

6.2. Algorithm feasibility analysis

Our proposed RU algorithm can solve most of the cases except there exist circles in the dependency graph. The circles may be caused by conflicts between black holes avoid dependencies, loop avoidance dependencies, and congestion avoidance dependencies. We take Figs. 4(a) and 4(b) as an example. In Fig. 4(a), for load balancing, flows $f1$ and $f2$ need to be migrated to a new path, i.e, $f1$: from $1 \to 2 \to 5 \to 3$ to $1 \to 5 \to 2 \to 3$, $f2$: from $5 \to 4 \to 2 \to 3$ to $5 \to 3$. To avoid black holes, loops, and transient congestion during updating, the rule update dependency graph is constructed and shown as Fig. 4(b). Due to the conflict between loop avoidance dependency and congestion avoidance dependency, there is a circle in Fig. 4(b), i.e., $5_{f1} \to 2_{f1} \to 5_{f2} \to 5_{f1}$. In this case, the rule update cannot be completed because each node in the circle has a precursor node and no node can be updated. Compared to Cupid [19], which does not consider the existence of circles in dependency graphs, we verify in Section 7 that the possibility of circles in the dependency graph is relatively low.



(a) Path Graph



(b) Dependency Graph

Fig. 4. An example of circles in the dependency graph. The link capacity is 1 unit and flow is labeled with its size. The blue and the red lines indicate the paths of flow $f1$ and flow $f2$, respectively. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Algorithm 4 Rules Update (RU) Algorithm

```
Input: : flows, old paths and new paths for each flow
Output: : updated sequence $US$
1: $US = \emptyset, i = ab = 0$;
2: Obtain dependency $\{\{d(s)\}, \{d(loop)\}, \{d(l)\}\}$ by Algorithms 1–3;
3: $D = \{\{d(s)\} + \{d(loop)\} + \{d(l)\}\}$;
4: while $D \notin \emptyset$ do
5:   $i = i + 1$;
6:   $US_i = \emptyset$;
7:   for $n_f : n_f \in D$ do
8:     if the precursor node of $n_f$ on $D = \emptyset$ then
9:       $US_i = US_i + n_f$;
10:    end if
11:    if $n_f \in deadlock \wedge$ the number of precursor nodes of $n_f$ on $D = 1$ then
12:      $ab = min\{AvailBw(l)\}(l \in p_{new}(f))$;
13:      $d_f^{p_{old}} = d_f^{p_{old}} - ab$ ;
14:      $d_f^{p_{new}} = d_f^{p_{new}} + ab$ ;
15:      $US_i = US_i + n_f^{new}$;
16:      if $d_f^{p_{old}} == 0$ then
17:        $D = D - n_f$;
18:      end if
19:    end if
20:  end for
21:  $US = US + US_i$, delete $US_i$ from $D$ ;
22: end while
23: return $US$;
```

6.3. Algorithm complexity analysis

Given l links, n nodes and m flow requests in the network, let d_1 and d_2 denote the depths in the dependency graphs of RU algorithm and Cupid algorithm, respectively. It is not difficult to find out that the complexity of RU algorithm and Cupid algorithm is $O(d_1m^2n^2)$ and $O(d_2m^2l^2)$ in the worst case. The depth of the rule update dependency graph is directly determined by the number of nodes updated at each time. Thus, we have $d_1 < d_2$ because RU algorithm can update multiple nodes at each time while Cupid algorithm that updates the nodes one by one in reverse order. In addition, $n \ll l$ in general. Therefore, RU algorithm has lower complexity than Cupid algorithm.

7. Performance evaluation

In this section, we will discuss the performance of our algorithms in different network topologies and traffic loads, including the number of segments, the number of directly updated flows on potential congestion link, and the update time. And we compare them with Cupid [19] and One Shot [21].

7.1. Simulation setup

Operating environment: We implement our algorithms using 1500+ lines of Python code on a computer with the hardware configuration of 3.1 GHz CPU, i7-5557U processor and 4G RAM.

Network topology: We evaluate our algorithm in the following two topologies. (1) A 4-pod fat-tree [30] DCN shown in Fig. 5(a) with 20 switches and 16 hosts. Each link bandwidth is 1 Gbps in the network. (2) A realistic WAN topology for interconnecting Microsoft's data centers [2] shown in Fig. 5(b), where each switch is connected to 2 hosts shown in the legend. Therefore, this network has 8 switches and 16 hosts. And each link bandwidth is 1500 Mbps in the network.

Traffic generation: In this experiment, traffic requests follow Poisson distribution. The source and destination nodes of each request are randomly selected. The size of all flows is constant, which is a different value in the different tests below. Each network has 100 flows running simultaneously.

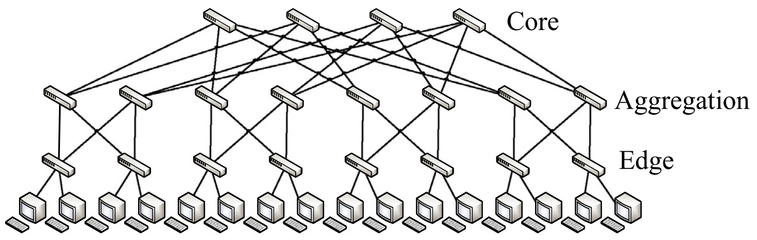
Paths generation: In this experiment, we first use the load balancing algorithm to calculate the path of all flows, and use the calculated paths as the current paths of flows. Then we randomly select 2 links as fault links in the entire network. We again use the load balancing algorithm to recalculate the paths for all flows and use the rerouted paths as the new paths for flows.

Time parameters: The total update time in the experiment consists of two parts: the time to generate the rule update sequence and the time to update the forwarding rules, and the update forwarding rule includes modification rule and addition rule. In the experiment, the time parameters of the modification rule and addition rule are set to 10 ms and 5 ms [2,28].

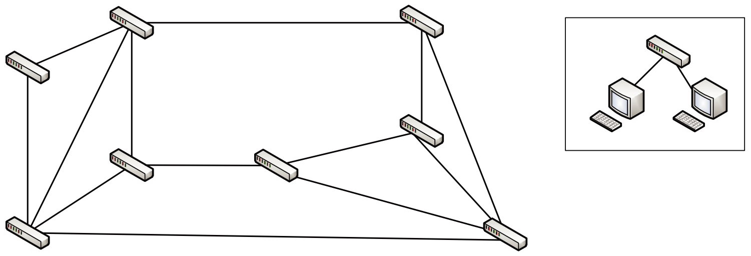
Algorithm comparison: (1) **One Shot** [21]: Transition directly from the old path to the new path. (2) **Cupid** [19]: Congestion-free consistent update algorithm discussed in Section 3. (3) **RU:** Our rules update algorithm proposed in Algorithm 4.

7.2. Comparison results for various network topologies

Fig. 6 shows the number of segments versus the corresponding occurrence probability in the new path of the flow in the DCN topology and the WAN topology. We can find from Fig. 6 that most of the flows have only one segment. This is because in the DCN topology and the WAN topology, there are many alternative paths so that most of flows only have the same ingress and egress switches as common nodes between the old paths and new ones. Therefore, the segmentation reverse update proposed by Cupid has little benefit for DCN topology and WAN topology. On the contrary, for our RU algorithm, the fewer the segments are, the more the benefits are. This is because our proposed RU algorithm can update multiple nodes in each segment instead of updating the nodes one by one like Cupid. It can also be observed that the number of segments obtained by the RU algorithm is less than the Cupid algorithm. This is because the segment method of the



(a) A 4-pod fat-tree DCN topology with 20 switches and 16 hosts.



(b) WAN topology with each switch connected to 2 hosts shown in the legend.

Fig. 5. Network topologies in different scenarios.

Cupid algorithm is different from that of the RU algorithm. In the Cupid algorithm, a node can become a segmented node as long as the node satisfies: (i) it is a common node of the new path and the old path, (ii) it has a different previous or next hop in the new path and the old path. However, with our RU algorithm, the segmented node can only be the common node of the new path and the old path and the next hop node of the common node is different from that in the new path and the old path.

Fig. 7 shows the number of flows that can be directly updated on a single potential congestion link during updating. The x-axis in Fig. 7 represents the minimum, average and maximum number of flows that can be directly updated on a single potential congestion link among 100 flows. The average number of flows is the sum of the number of flows that can be directly updated on all potential congestion links divided by the total number of potential congestion links. We can observe from Fig. 7 that using RU algorithm, the number of flows that can be directly updated on a single potential congestion link is increased by an average of 1 and 67% in the DCN and the WAN respectively, compared with Cupid algorithm. The reason is that our RU algorithm considers maximizing the number of updatable flows when constructing a congestion-free dependency graph. Fig. 7 shows that the minimum number of flows is 0. This is because in the case of low network utilization, many alternative paths to the flows are provided, so there are no potential congestion links. We also observe that the number of directly updated flows in Fig. 7(b) is less than the number of directly updated flows in Fig. 7(a). This is because there are more alternative links in the DCN topology than in the WAN topology. Therefore, in the DCN topology, the number of flows with changed paths is more than that in the WAN topology.

Figs. 8 and 9 show the update time of simultaneously updating 100 flows at light ($<$ 30% network utilization), medium (30–60% network utilization) and heavy ($>$ 60% network utilization) traffic load in DCN and WAN, respectively. The network utilization is measured by weighted link utilization. The update time includes two parts: the time to generate the update order and the time to update the forwarding rules [21]. Because the operation of modifying the rules may occur in each update sequence, we assume that it takes 10 ms to update the forwarding rules for each update sequence. We observe from Figs. 8 and 9 that in the DCN and WAN topology, RU algorithm takes longer time than the one shot algorithm, but it takes less time than the Cupid algorithm to schedule a feasible update sequence. The reason is that One Shot is an update method that directly transits from the old path to the new path without considering the dependencies between the rules update. Thus it does not take more time to generate the order, and only take the 10 ms to update the forwarding rules. The RU algorithm takes less time to update the rules than the Cupid algorithm, because the RU algorithm can update multiple nodes at once and minimize the number of delayed update flows, instead of updating the nodes one by one and delaying the update of all moved into flows on potential congestion links in Cupid. Therefore, compared with the Cupid algorithm, the RU algorithm reduces the search overhead of the rule update dependency graph and the number of rule update sequence waiting for updating, thereby the time for the rule update is reduced.

Another observation from Figs. 8 and 9 is that for light, medium, and heavy traffic load, the update time of the RU algorithm is 48%, 34%, and 25% less than the update time of Cupid algorithm, respectively. The reason is that under light traffic load, most flows have only one segment in DCN and WAN topologies as shown in Fig. 6. Therefore, the segment update by Cupid has little benefit for DCN and WAN topologies. Moreover, under light traffic load, few links become congested ones during the update, so the rule update dependency graph is relatively simple. When the network load is larger, the more segments are generated, thus the effect of multiple node updates in RU algorithm is not as obvious as that of light load traffic. However, the greater the network load, the higher the probability of transient congestion. In the process of avoiding transient congestion, the RU algorithm maximizes the number of flows that can be directly updated on potential congestion link. Thus the search overhead of the rule update dependency graph is reduced, and the RU algorithm can effectively reduce the time for rule update under different network traffic load. Furthermore, generating update sequences

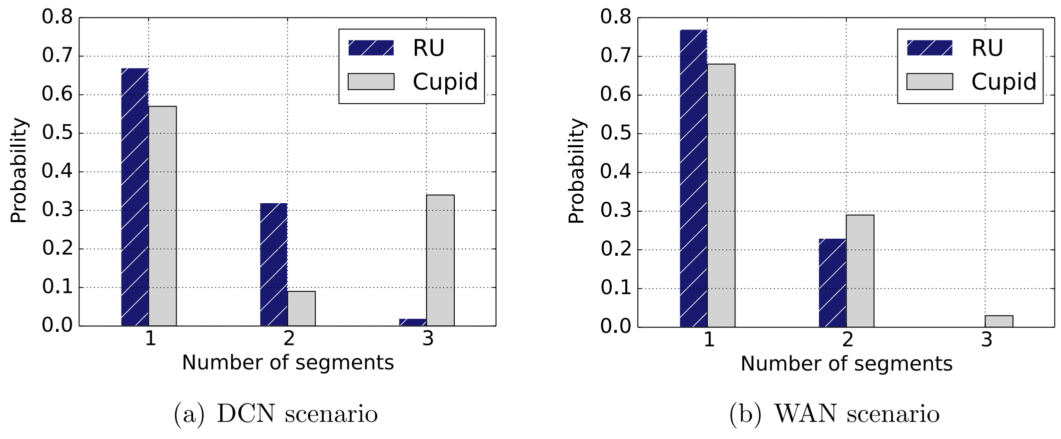


Fig. 6. The number of segments by Cupid algorithm and our RU algorithm for different topologies.

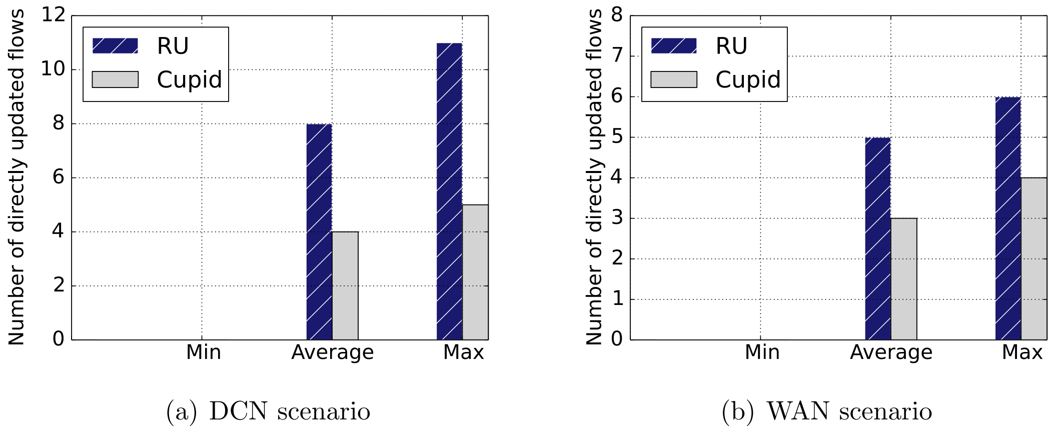


Fig. 7. The number of flows that can be directly updated on a single potential congestion link.

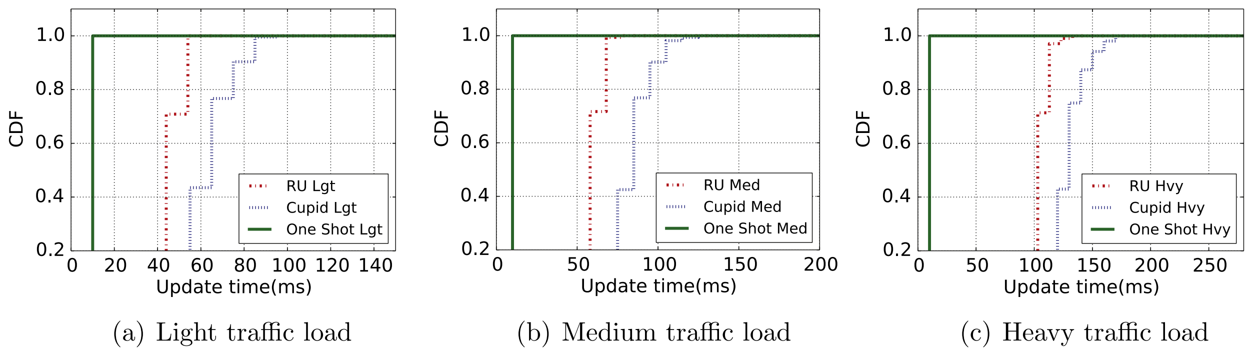


Fig. 8. Update time at different traffic load in DCN scenario.

in WAN topology takes less time than that in DCN topology. This is because there are fewer alternative paths in the WAN than that in the DCN, thus the number of flows that change paths in the WAN topology is less than that in the DCN topology, and further the complexity of the dependency graph is reduced.

Fig. 10 shows the changes of percentage of excess link capacity as link utilization increases. Congestion may happen when the link utilization is larger than 50%, and a larger value indicates a higher probability of congestion. The *y*-axis represents the maximum percentage of excess link capacity. We can observe that as the link utilization increases, the percentage of excess link capacity by the One Shot method increases. However, for our RU algorithm, as the link utilization increases, the percentage of excess link capacity is always zero. This is because our RU

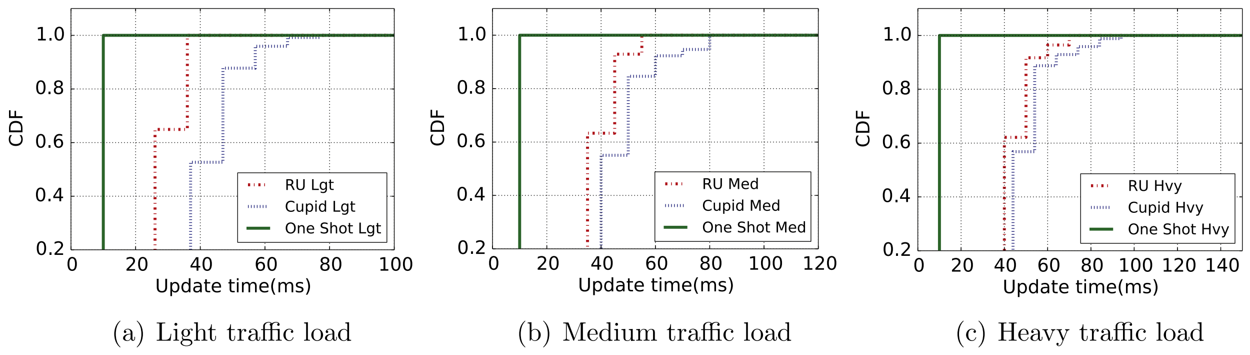


Fig. 9. Update time at different traffic load in WAN scenario.

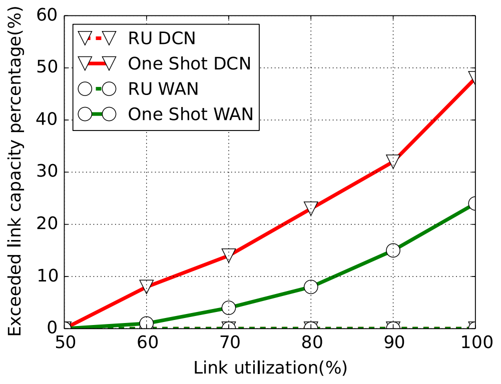


Fig. 10. Percentage of excess link capacity comparison.

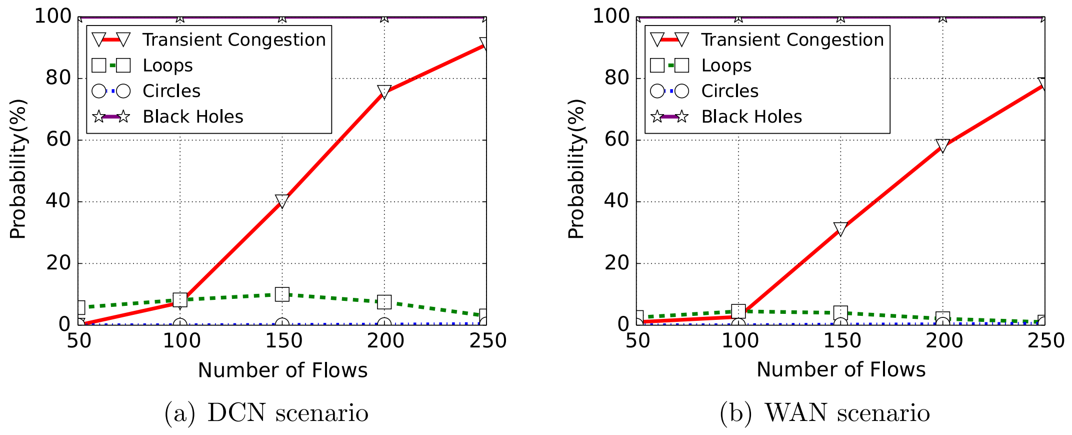


Fig. 11. The probability of occurrence of loops, congestion, circles in the worst case of inconsistent rule updates.

algorithm takes into account the update order of rules so that congestion can be avoided. On the contrary, One Shot method directly issues rules without considering the order of rule updates so that congestion will occur due to asynchronous update of switches.

7.3. Feasibility analysis

The proposed RU algorithm can avoid loops, black holes and congestion, except for circles. To verify the feasibility of the RU algorithm, we evaluate the probability of loops, transient congestion, block holes, and circles occurring in DCN and WAN topologies as shown in Fig. 5. All experiments are run 5000 times under different number of flows, and we record the number of loops, transient congestion, block holes, and circles and furthermore, obtain the probability of occurrence of loops, transient congestion, block holes, and circles in each experiment. In two networks, the number of flows varies from 50 to 250, and the size of the flows ranges from 10 M to 100 M, which represents different scale of traffic loads in the network.

Figs. 11(a) and 11(b) indicate that as the number of flows increases, the probability of occurrence of transient congestion during updating is increasing. This is because the network load increases as the number of flows increase, resulting in the reduction of the number of feasible alternate paths. Besides, the probability of black holes occurring is always 100%. This is because black holes are very easy to occur during flow migration, and a large number of flows in this experiment need to be migrated. In addition, we can also observe that the probability of loops occurrence during updating increases at first and then decreases with the number of flows increasing, but the probability of loops occurrence does not exceed 12%. It is obvious that the probability of loop occurrence is increasing with the number of flows increasing because more flows are required to change paths. However, when the number of flows is excessively large, the alternate path for each flow will be reduced in the network due to the limited link capacity, and the probability of loops occurrence will decrease. More importantly, we can see that the possibility of circles occurrence in the dependency graph is always close to zero, only 0.4% and 0.6% in DCN and WAN, respectively, when the number of flows equals to 250. Therefore, we can obtain that RU algorithm can solve most of cases, 99.6% in DCN and 99.4% in WAN, and the possibility of circles occurrence in the dependency graph is relatively low even when network traffic is very high.

8. Conclusions

In this paper, we study the consistency of flow forwarding rules update in data plane in SDN. First, we analyze how to avoid black holes, loops and transient congestion, and construct the models without black holes, loops and transient congestion, respectively. Furthermore, we propose the black holes, loops and transient congestion avoidance algorithms, which reduce the queuing delay of the rules update of flows. Subsequently, we propose a RU algorithm that combines the three avoidance algorithms to update the flow rules to simultaneously avoid black holes, loops and transient congestion. Finally, simulation results show that our algorithm significantly increases the number of flows that can directly be updated on potential congestion links and reduce the time for rules update.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 61772432, 61772433, 61722105), Preresearch Fund, China (No. 6140449XX61001) Natural Science Key Foundation of Chongqing, China (cstc2015jcyjBX0094), Natural Science Foundation of Chongqing, China (CSTC2016JCYJA0449), China Postdoctoral Science Foundation (2016M592619), and Chongqing Postdoctoral Science Foundation (XM2016002).

References

[1] N. Mckeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, Openflow: Enabling innovation in campus networks, ACM SIGCOMM Comput. Commun. Rev. (2008) 69–74.

[2] X. Jin, H.H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, R. Wattenhofer, Dynamic scheduling of network updates, ACM SIGCOMM Comput. Commun. Rev. (2014) 539–550.

[3] R.I.T.D.C. Filho, M.C. Luizelli, L.P. Gaspary, Scalable QoE-aware path selection in SDN-based mobile networks, in: IEEE INFOCOM, 2018, 1–9.

[4] Y. Wang, S. You, An efficient route management framework for load balance and overhead reduction in SDN-based data center networks, IEEE Trans. Netw. Serv. Manag. (2018) 1–13.

[5] C.Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, R. Wattenhofer, Achieving high utilization with software-driven WAN, in: ACM SIGCOMM 2013 Conference on SIGCOMM, 2013, pp. 15–26.

[6] Z. Guo, Y. Xu, R. Liu, A. Gushchin, K. yin Chen, A. Walid, H.J. Chao, Balancing flow table occupancy and link utilization in software-defined networks, Future Gener. Comput. Syst. (2018) 213–223.

[7] M.M. Tajiki, S. Salsano, L. Chiaraviglio, M. Shojafar, B. Akbari, Joint energy efficient and QoS-aware path allocation and VNF placement for service function chaining, IEEE Trans. Netw. Serv. Manag. (2018) 1–14.

[8] R. Maaloul, R. Taktak, L. Chaari, B. Cousin, Energy-aware routing in carrier-grade ethernet using SDN approach, IEEE Trans. Green Commun. Netw. (2018) 844–858.

[9] G.S. Aujla, N. Kumar, MEnSuS: An efficient scheme for energy management with sustainability of cloud data centers in edge-cloud environment, Future Gener. Comput. Syst. (2018) 1279–1300.

[10] G. Xu, B. Dai, B. Huang, J. Yang, S. Wen, Bandwidth-aware energy efficient flow scheduling with SDN in data center networks, Future Gener. Comput. Syst. (2017) 163–174.

[11] R. Mahajan, R. Wattenhofer, On consistent updates in software defined networks, in: Twelfth Acm Workshop on Hot Topics in Networks, 2013, pp. 1–7.

[12] S. Vissicchio, L. Cittadini, FLIP the (Flow) table: Fast lightweight policy-preserving SDN updates, in: IEEE INFOCOM 2016 - the IEEE International Conference on Computer Communications, 2016, pp. 1–9.

[13] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, D. Walker, Abstractions for network update, ACM SIGCOMM Comput. Commun. Rev. (2012) 323–334.

[14] T. Mizrahi, E. Saat, Y. Moses, Timed consistent network updates, in: Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, 2015, pp. 1–15.

[15] T. Mizrahi, E. Saat, Y. Moses, Timed consistent network updates in software-defined networks, IEEE/ACM Trans. Netw. (2016) 3412–3425.

[16] W. Zhou, D. Jin, J. Croft, M. Caesar, P.B. Godfrey, Enforcing customizable consistency properties in software-defined networks, in: Usenix Conference on Networked Systems Design and Implementation, 2015, pp. 73–85.

[17] S. Vissicchio, L. Cittadini, S. Vissicchio, L. Cittadini, Safe, efficient, and robust SDN updates by combining rule replacements and additions, IEEE/ACM Trans. Netw. (2017) 1–14.

[18] S. Vissicchio, L. Vanbever, L. Cittadini, G.G. Xie, O. Bonaventure, Safe update of hybrid SDN networks, IEEE/ACM Trans. Netw. (2017) 1–14.

[19] W. Wang, W. He, J. Su, Y. Chen, Cupid: Congestion-free consistent data plane update in software defined networks, in: IEEE INFOCOM 2016 - the IEEE International Conference on Computer Communications, 2016, pp. 1–9.

[20] H.H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, D. Maltz, Zupdate: updating data center networks with zero loss, ACM SIGCOMM Comput. Commun. Rev. (2013) 411–422.

[21] J. Zheng, H. Xu, G. Chen, H. Dai, J. Wu, Congestion-minimizing network update in data centers, IEEE Trans. Serv. Comput. (2017) 1–14.

[22] T. Mizrahi, O. Rottenstreich, Y. Moses, Timeflip: Scheduling network updates with timestamp-based TCAM ranges, Comput. Commun. (2015) 2551–2559.

[23] T. Mizrahi, Y. Moses, Software defined networks: It's about time, in: IEEE INFOCOM 2016 - the IEEE International Conference on Computer Communications, 2016, pp. 1–9.

[24] I. Maity, A. Mondal, S. Misra, C. Mandal, CURE: Consistent update with redundancy reduction in SDN, IEEE Trans. Commun. (2018) 1–8.

[25] A. Basta, A. Blenk, S. Dudycz, A. Ludwig, S. Schmid, Efficient loop-free rerouting of multiple SDN flows, IEEE/ACM Trans. Netw. (2018) 948–961.

[26] K.T. Firster, R. Mahajan, R. Wattenhofer, Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes, in: Ifip NETWORKING Conference, 2016, pp. 1–9.

[27] S. Luo, H. Yu, L. Luo, L. Li, Arrange your network updates as you wish, in: Ifip NETWORKING Conference, 2016, pp. 10–18.

[28] H. Xu, Z. Yu, X.Y. Li, L. Huang, C. Qian, T. Jung, Joint route selection and update scheduling for low-latency update in SDNs, IEEE/ACM Trans. Netw. (2017) 1–15.

[29] R. Tarjan, Depth-first search and linear graph algorithms, in: Switching and Automata Theory, 1971., Symposium on, 2008, pp. 114–121.

[30] M. Al-Fares, A. Loukissas, A. Vahdat, A scalable, commodity data center network architecture, in: Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, 2008, pp. 63–74.

Pan Li received the B.S. degree in Computer science and Engineering from Chongqing Three Gorges University, Chongqing, China, in 2016. She is currently working toward the Master degree in signal and information processing, Southwest University. Her research interests include data center networks and software defined networking.

Songtao Guo received the B.S., M.S., and Ph.D. degrees in computer software and theory from Chongqing University, Chongqing, China, in 1999, 2003, and 2008, respectively. He was a professor from 2011 to 2012 at Chongqing University and a professor from 2013 to 2018 at Southwest University. He is currently a full professor at Chongqing University, China. He was a senior research associate at the City University of Hong Kong from 2010 to 2011, and a visiting scholar at Stony Brook University, New York, from May 2011 to May 2012. His research interests include wireless sensor networks, wireless ad hoc networks, data center networks and mobile edge computing. He has published more than 100 scientific papers in leading refereed journals and conferences. He has received many research grants as a principal investigator from the National Science Foundation of China and Chongqing and the Postdoctoral Science Foundation of China.

Chengsheng Pan received his Ph.D. degree in Computer Application Engineering from Northeastern University, Shenyang China, in 2001. He is currently a full professor at Dalian University. His research interests include communication networks technology and its applications.

Li Yang received her Ph.D. degree in Nanjing University of Science and Technology. In 2009, she joined the Dalian University and now she is a professor of Dalian University. Her research interests include the theory and method of wireless network protocol.

Guiyan Liu received the B.S. degree and master's degree from Southwest University, Chongqing, China, in 2014 and 2017, respectively. She is currently working toward the PhD's degree in computing intelligence and information processing, Southwest University. Her research interests include traffic measurement and network scheduling in data center networks and software defined networking.

Yue Zeng received the B.S. degree in Computer science and Engineering from Chongqing Three Gorges University, Chongqing, China, in 2016. He is currently working toward the Ph.D. degree in signal and information processing, Southwest University. His research interests include network energy saving and software defined networking.