Delay Updating in Software-Defined Datacenter Networks

Ting Qu*, Deke Guo*✉, Jia Xu† and Zhong Liu*
*Science and Technology on Information System Engineering Laboratory
National University of Defence Technology
†School of Computer, Electronics and Information, Guangxi University
E-mail: {quting14, dekeguo}@nudt.edu.cn, xujia@gxu.edu.cn, liuzhong@nudt.edu.cn.

***Abstract*—Software-Define Datacenter networks (SDDCs) are constantly changing due to various network update events. A set of involved flows of each update event should be migrated to the feasible paths without congestion. To tackle a single update event, prior methods find and execute a migration sequence so as to transform the initial traffic distribution to the final traffic distribution. However, when handling a queue of multiple update events, the head-of-line blocking problem always appears and considerably lower the efficiency of network update. To tackle this problem, we employ the idea of delay updating which schedules other queued events instead of the blocked head-event, aiming to break the blocking state and provide updating opportunities for other events. We formulate this problem as an optimization problem and propose partial delay updating (PDU) strategy. PDU tackles the queued update events based on their arrival order to preserve the fairness. Moreover, it prefers to just delay those blocked flows, lacking enough bandwidth resources, instead of all flows in the head-event.The evaluation results show that our delay updating strategy achieves $60\% - 80\%$ reduction in average completion time of update events, compared to FIFO scheduling strategy, in four types of flow size.**

I. Introduction

Software defined network (SDN) is a new network paradigm to simplify the network management and enable the innovations through the programmability of networks. The insight of SDN has been widely used in traffic engineering in WAN [1], [2] as well as datacenter applications. An essential issue of maintaining a data center is to tackle various update events, such as visual machines (VM) migration, link failure, upgrade of physical devices and softwares. Such update events occur frequently and unpredictably; hence, they cause the changes of network topology and traffic distribution. For an update event, some around flows would be rerouted and migrated to the feasible paths without congestion. Such a migration should be carefully planned in advance. Otherwise, it will lead to the large-scale congestion and packet loss. Then, the network might slow to a crawl and become unusable. The appearance of software-defined data centers (SDDC) bring new opportunities to tackle such update events. With the global information of the datacenter, such as topology, traffic distribution and link utilization, the SDN controller can calculate a practical migration plan for each update event. More precisely, the controller would compute the optimized paths for those rerouted flows to avoid link congestion and packet loss [3], and then update the flow tables of related switches.

The migration plan of an update event involves a series of intermediate network states and the transformation between adjacent network states, exhibiting various challenging issues. One is how to guarantee the consistent update. Reitblatt et al. [4] use different labels on packet headers to avoid the effects of asynchronous updating of switches. Cupid [5] performs a consistent flow table updating with one flow entry on switches to save the space of flow table. The two-phase method [6] and its variants [7], [8] follow this scheme. Congestion-free is another important requirement during the update process. SWAN [9] and zUpdate [10] utilize reserved link capacity in advance to guarantee a congestion-free update in SDDC. Moreover, accelerating the update process is also a challenging issue. Dionysus [11] finds out a series of consistent update schemes and dynamically schedule the transient network states to speedup update process. Zheng et al. slightly relax the congestion constraint for enhancing the update speed [12].

Actually, update events may occur simultaneously due to the dynamic network; hence, they form an update queue according to a given fairness scheme. It seems that the combination of involved flows of all update events can be treated as a new update event. Accordingly, we can simply use existing methods for a single update event to find a migration sequence from an initial traffic distribution to the final traffic distribution. However, it is NP-hard to ensure the existence of such a consistent migration [13]. Moreover, such kind of migration sequence can not be found sometimes. Even though the migration sequence exists, a series of network state migrations will lead to a large mount of network-scale migration of background traffic except for those flows involved by the update event. Such migrations do disturb the network applications, leading to a low network utilization and lengthening the execution time of applications. Thus, those queued update events should be handled individually and the migration of other background traffic is forbidden during the process of each update event.

For any update event with a collection of flows, the update cannot be finished until every flow is completed. However, not all flows of an update event can get enough network resources and be updated immediately. For this reason, the head-event of the update queue may be blocked; hence, all subsequent update events have to wait. The waiting duration may be infinite and cannot be accurately estimated due to the dynamic properties of network. This head-of-line blocking problem would prolong the completion time of all update events in the queue. To the best of our knowledge, prior update methods remain inapplicable to tackle the update problem of multiple update events under two constraints, guaranteeing both fairness and efficiency but not migrating other background traffic.

In this paper, we characterize the problem of updating multiple update events and propose the *delay updating* to significantly improve the update efficiency, while slightly relaxing the scheduling fairness. The most related work is the *delay scheduling*, which improves the scheduling efficiency of a queue of computing jobs against the fairness strategy FIFO [14]. The head-job would wait a short time for the release of specific nodes for achieving the data locality. Thus, other jobs may be launched instead of the head-job if possible, so as to increase the network throughput. Through the experimental observation, we find that simply delaying the entire head-blocking event remains inapplicable to efficiently update multiple events.

For this reason, we propose a partial delay updating (PDU) method to improve the efficiency of updating queued events, by solving the head-of-line blocking problem. The PDU s-trategy processes the queued update events in order of arrival and just delays partial blocked flows of the head-event. Those blocked flows will be removed to another queue, called delay queue. Thus, the following update events have opportunities to be executed in time, thus considerably improving the efficiency of tackling all update events. Note that all unblocked flows in the head-of-line event are not necessary to be delayed. This avoids the drawbacks of delaying the entire update event. The delay queue is checked orderly if remaining flows of any delayed event can be executed after processing an event in the update queue. This brings additional opportunities to process those delayed update events in time, thus ensuring the fairness of our PDU method.

The remainder of this paper is organized as follows. Section II reports the problem of updating multiple events and the basic idea of delay updating. Section III presents the delay updating model and a partial-event delay method. We report the evaluation methodology and results in Section IV. Finally, Sections V and VI discuss related work and conclude this paper, respectively.

II. Delay update overview

In this section, we characterize the problem of multiple update events caused by head-of-line blocking in II-A and propose the idea of delay updating to tackle this issue in II-B.

A. The problem of updating multiple events

Given a fairness FIFO scheduling strategy, Multiple update events can be updated according to their arrival time. In FIFO, to be specific, the head-event is blocked due to the insufficiency of the network bandwidth, the other update events still have no opportunity to be updated. As a result, all update events following the head-event have to wait until the head-event is unblocked. Thus, the average ECT of all queued update events will increase.

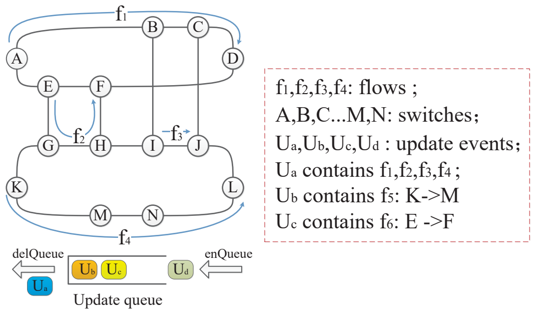


Fig. 1. Head-of-line blocking in a fairness queue with multiple update events.

FIFO strategy gives absolute fairness to each update event, but brings low efficiency. As shown in Fig.1, the bandwidth of each link is fixed to 1 MB and the bandwidth requirement of each flow is 1 MB. The event U_a contains four flows f_1, f_2, f_3, f_4 that are executing in the network. Two update events U_b, U_c arrive, which two contain one flow f_5 and f_6, respectively. The link bandwidth demands of these flows are all 1 MB and the final paths of f_5 and f_6 are $K \rightarrow M$, $E \rightarrow F$, respectively.

According to the FIFO strategy, the update event U_b will be performed first. However, the residual link bandwidth of $K \rightarrow M$ cannot provide abundant network channel. Thus, flow f_4 should be migrated following the other path $K \rightarrow G \rightarrow H \rightarrow I \rightarrow J \rightarrow L$, where such path is occupied by flows f_2 and f_3. Thus, f_2 and f_3 should be migrated as well. Flow f_2 can be moved to the path $E \rightarrow F$. Then, flow f_1 should be migrated before f_3. If we move f_1 to the path $A \rightarrow E \rightarrow F \rightarrow D$, flow f_3 can be migrated to the path $I \rightarrow B \rightarrow C \rightarrow J$. However, the migration of flow f_1 will conflict with flow f_2. If we migrate f_1 to the path $A \rightarrow E \rightarrow G \rightarrow H \rightarrow F \rightarrow D$, it will be conflict with f_4. Thus, we cannot find a feasible path for f_1 without congestion; hence, f_3 cannot be migrated. Thus, no matter how to adjust the paths of these flows, the new flow f_5 can not be transmitted in the network.

However, the update event U_c followed U_b cannot be performed, even if there is enough network bandwidth to accommodate f_6 of U_c. Thus, FIFO strategy leads to non-trival queueing time and a terrible average ECT of all queued update because of the blocked head-event. To strengthen the updating efficiency, we should relax the constraint of fairness slightly. As a choice, we can schedule the update event U_c instead when the event U_b is blocked by bandwidth resources. Such a regularization sacrifices a little fairness, but reduces the queueing time of U_c greatly and gets a high level of efficiency during whole network update.

B. The strategies of delay update

To tackle the problem aforementioned, an efficient method is to simply delay the blocked head-event and schedule other appropriate update events instead.

Delay an entire update event. As shown in Fig. 2, when the head-event U_a is blocked because of the limitation of network utilization, it will be enqueued to the delay queue and be executed until it gets enough bandwidth resources.

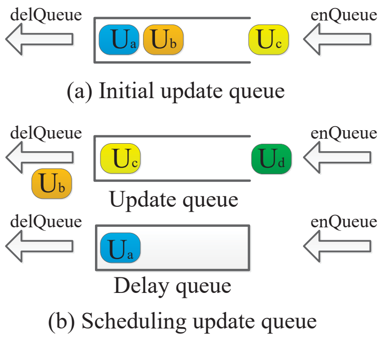
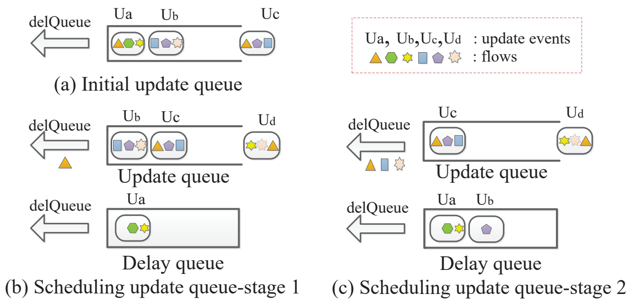


Fig. 2. Entire-event delay to break blocking state.



delQueue
Ua
Ub
Uc
Ua, Ub,Uc,Ud : update events
: flows
(a) Initial update queue
Ud
Update queue
Delay queue
(b) Scheduling update queue-stage 1
(c) Scheduling update queue-stage 2

Fig. 3. Partial-event delay to break blocking state.

The update event U_b close behind U_a will be scheduled if its requirement on network resources is satisfied and so on. Such a delay makes a big contribution to the average ECT of all queued update events, but prolongs the completion time of the delayed update event U_a. At this point, it is unfair for the event U_a. Thus, the delay queue should be checked periodically to see if any delayed event, such as U_a, can be performed. The update event U_a may grasp the opportunity and update.

Delay partial flows in an update event. To guarantee the fairness and improve the efficiency of the event-level network update, the inexecutable parts of the head-event U_a will be enqueued to the delay queue as shown in Fig. 3(b). When U_a is rescheduled, only the part of U_a stayed in the delay queue will be handled. Thus, it reduces the recomputing scale of each update event greatly. As similar to the strategy of delaying the entire event, the delay queue should be checked to find out the feasible update events. If not exist, the new head-event U_b will be scheduled as shown in Fig. 3(c).

III. Design for Updating Multiple Events

In this section, we start with an optimization model for the efficiency problem of multiple update events in III-A. Then, we design a partial delay updating method, PDU, to improve average ECT during network update in III-B.

A. Problem formulation

Network model. The network is defined as a graph $G{=}(V,E)$, where V and E contain all switches and nodes, respectively. The flow is defined as $f{=}(s_f,d_f,b_f)$ with s_f being the ingress switches, d_f being the egress switch and b_f being the bandwidth requirement of the flow f. Let $v(f_{i,j})$ be the actual load of flow f on link $e_{i,j}$ and $c_{i,j}$ be the available capacity of link $e_{i,j}$. Then, a set of update events is expressed as $U{=}\{U_1,U_2,\ldots,U_w,\ldots U_n\}$. The update event U_w is denoted as $U_w{=}\{f_1,f_2,\ldots\}$ and $U_w^{'}$ is the subset of U_w, which is composed by a set of delayed flows of the update event U_w. Let $L(U_w)$ be the traffic load of the update event U_w and δ_w be the ratio of remaining traffic to all traffic load of an update event.

Given a set of network update events $U{=}\{U_1,U_2,\ldots,U_w,\ldots U_n\}$, we aim to reduce the average event completion time (ECT) of all update events in the set U.

$$\min \quad (\sum_{w=1}^{n} T_c(U_w))/n \tag{1}$$

Let $T_c(U_w)$ denote the completion time of the update event U_w. We can get

$$T_c(U_w){=}T_q(U_w)+T_p(U_w),$$

where $T_q(U_w)$ and $T_p(U_w)$ denote queueing time and execution time of the update event U_w, respectively. Queueing time is consisted of waiting time and the plan time. Execution time denotes the duration of an update event. The update event is usually formed by three types of flows, i.e., small flow, middle flow and big flow, whose proportions are α,β and γ, respectively. According to the traffic characteristics of datacenter [15], the durations of such three types flows can be set as empirical values, i.e., t_s,t_m and t_b. $|U_w|$ denotes the number of flows in the update event U_w.

$$T_q(U_w){=}T_c(U_{w-1}) \tag{2}$$

$$T_p(U_w){=}\alpha_w|U_w|t_s{+}\beta_w|U_w|t_m{+}\gamma_w|U_w|t_b \tag{3}$$

$$\alpha_w{+}\beta_w{+}\gamma_w{=}1 \tag{4}$$

We call that network is legal if parameters hold that:

$$\forall e_{i,j}\in,\ c_{i,j}\geq 0 \tag{5}$$

$$\forall f\in C_{U_w}U_w^{'},\ \forall e_{i,j}\in p_f,\ v(f_{i,j})\leq c_{i,j} \tag{6}$$

$$\forall f\in C_{U_w}U_w^{'},\ \forall i,j\in G_f, v(f_{i,j})=v(f_{j,i}) \tag{7}$$

$$\forall f\in C_{U_w}U_w^{'},\ \forall e_{i,j}\notin p_f,\ v(f_{i,j}){=}0 \tag{8}$$

$$\forall f\in C_{U_w}U_w^{'},\ \forall i,j\in G_f,\ v(f_{i,j}){=}b_f \tag{9}$$

Formula (5) guarantees that the initial network is congestion-free. Formulas (6) and (7) indicate that partial flows of the update event U_w can pass through the network without congestion and such flows get enough bandwidth resources (i.e., its bandwidth requirement b_f). Formula (8) guarantees that each flow transmits on the path p_f allocated to it. Formula (9) guarantees that the traffic will not pass through other links which are not belonged to the allocated path p_f.

Partial delay model. Actually, if the head-event keeps waiting for enough bandwidth resources, other update events followed it will have no opportunities to be updated. Thus, the average ECT of all update events will be prolonged and the network utilization drops quickly. To disarm the head-of-line blocking, we propose partial delay updating strategy. The flows of an update event, without enough bandwidth resource, will be grouped and added to another queue called delay queue Q_d. Thus, the other queued update events with adequate bandwidth resources will be scheduled instead. Such delayed events will be rescheduled while bandwidth resources are available. Such a mechanism avoids the head-of-line blocking problem. In addition, network resource will get the utmost utilization. Then, Formulas (2) can be modified into:

$$T_q(U_w)=(\sum_{i=1}^{w-1} T(C_{U_i}U_i^{'})+\sum_{j=w+1}^{w+x} T(C_{U_j}U_j^{'}))/(w+x-1), \quad (1\leq w+x\leq n) \tag{10}$$

B. Delay updating of partial-event

Fairness and efficiency are the two critical metrics when scheduling multiple update events. First in first out (FIFO) scheduling strategy is widely used for its simpleness to implement and strict fairness. It is proved to be optimal for guaranteeing fairness and reducing tail completion time of all queued events if traffic load of each queued event is same [16]. Actually, such an assumption fails sometimes. The update events occur unexpectedly and exhibit various characteristics. In addition, if the duration has a characteristic of heavy-tailed distribution, FIFO strategy usually leads to the head-of-line blocking because of the limited bandwidth resources; hence, the average and tail ECT of all update events will increase. Moreover, waiting for bandwidth resources is usually a very time-consuming thing, because the required resources of an update event may be occupied by each other like Fig. 1, especially when the network utilization is very high. Under these circumstances, FIFO strategy can not ensure the normal execution of multiple update events, not to mention guaranteeing the fairness and efficiency.

To avoid the head-of-line blocking problem among a fairness queue of multiple update events, we fine tuning the update sequence and relax the constraint of the strict fairness slightly. An intuitional method is to iterate through all queued update events and select an update event, each flow of which can be transmitted immediately with adequate bandwidth resources. However, traversing all queued update events suffers a huge computation and time overhead. Especially when an update event contains superabundant flows or too many update events stay in the update event. Moreover, partial flows of an update event, more or less, usually cannot be scheduled due to the lack of bandwidth resources. Thus, we may fail to find an appropriate update event, each flow of which can be updated.

In this paper, we propose a partial-delay updating (PDU) method, a simple but effective scheduling method. To guarantee fairness, it schedules queued events based on their arrival time, but dynamically adjusts the update sequence. When no bandwidth resources for some flows of an update event, we group these flows and move them to another queue (called delay queue). Then, the other update events can get updating opportunities in advance. Such a method can effectively avoid head-of-line blocking problem.

Note that we execute each event in the update queue and the delay queue according to their arrival order. Such a updating sequence guarantees the fairness of updating, but it fails to reflect the discrepancies of several update events such as traffic load. That is, updating according to the order in the delay queue considers no residual traffic or completed traffic of an update event. However, some update events are delayed because a small fraction of traffic cannot be migrated. Thus, we propose another scheduling method based on our PDU method to further improve the updating efficiency.

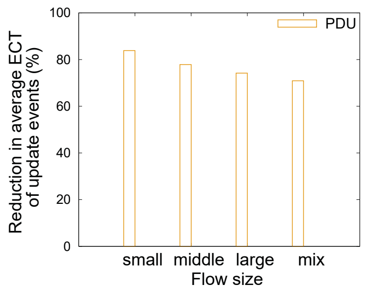


Fig. 4. Reduction in average ECT of update events with our method PDU against FIFO under different flow size when network utilization is higher than 70% in a 8-pod fat-tree network.

IV. Experiment and Evaluation

We start with the setup of our trace-driven evaluations in an 8-pod Fat-Tree datacenter network. We evaluate the performance of our partial delay updating methods against traditional fairness method FIFO in many aspects over a real data-set from Yahoo!'s data center [17].

A. Evaluation Setup

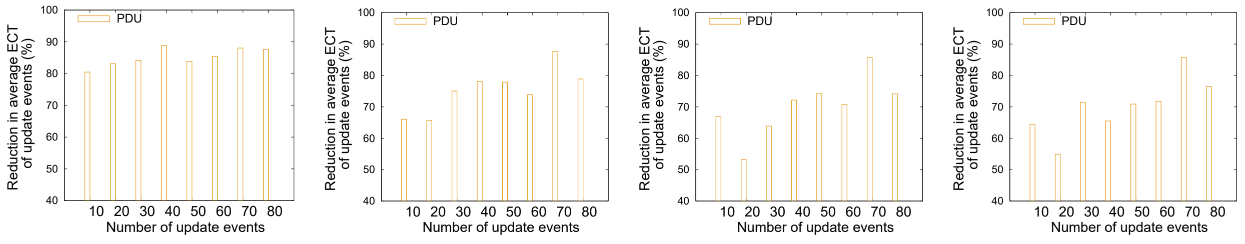
Topology. We consider a three-layer, 8-pod Fat-Tree [18] datacenter network with 80 switches and 128 servers. In a Fat-Tree data center, the number of servers and switches is determined by the setting of a parameter k. A Fat-Tree topology accommodates $5k^2/4$ switches and $k^3/4$ servers, which form k pods. The parameter k is set to 8 in our experiments. The bandwidth of each link is fixed as 1 Gbps in our experiments.

Workloads. More than $10,000$ flows are running in the network simultaneously as background traffic. These flows are generated from a real traffic data-set from Yahoo!'s data center [17]. In addition, we further inject a set of network update events into the network. The traffic distribution of each event follows the data center traffic characteristics in [15]. In addition, we also construct other update events consisted of flows with a certain size.

Methods. For a traditional scheduling method, the head-event keeps scheduling state even if partial flows of it wait for bandwidth resources. After all flows of the head-event are executed, next update event will be scheduled. Partial delay updating (PDU), delays parts of the update events which cannot be executed with the limited bandwidth resources. Once the network resources are available, they will be rescheduled.

B. Impact of the flow size of each update event

Lots of previous works focus on routing large flows to less congested path for load balancing [3], [19], [20]. To show the



(a) Small flow (b) Middle flow (c) Large flow (d) Mix-flow

Fig. 5. Reduction in average ECT of update events with our method PDU against FIFO under different number of events when network utilization is higher than 70% in a 8-pod fat-tree network.

difference of the flow size in updating, we classify the flows of each event in a set of update events into small flow($<10k$), medium flow ($10\sim500k$), large flow ($500k\sim3M$) and mix sizes of flows (80% small flows and 20% large flows). Fig. 4 shows the reduction in average ECT of update events with different flow size at 70% traffic load in a 8-pod fat-tree network. The PDU method achieves 70%$\sim$80% reduction in average ECT of all queued update events for small and medium flows, while 60%$\sim$70% for large and mix flows. Thus, all sizes of flows can be treated equally in our experimental condition, even the big flows.

C. *Impact of the number of update events*

Fig. 5 shows the impact of the number of update events. We first construct four types of update events with different size of flows in each event. Then, observing the influence of different number of update events with one type of flows on average ECT of update events. Needless to say, we can clearly see that our two methods effectively reduce the average ECT of all update events against the traditional scheduling method when the update events' number changes from 10 to 80.

With the increasing number of update events, the change of reduction in average ECT has no continuous rule. For the update events with small flows only, the reduction in average ECT continues increasing when the number of update events ranges from 10 to 40. Once the number of update events reaches 50, the reduction in average ECT decreases. After that, it increases. For other update events with other three types of flows only, the reduction in average ECT shows the similar trend, which increases in a certain range with the change of the number of update events and then drops and rises. Specially, there is an obvious indentation when the number of update events is 20, using only PDU method only with big flows or mix flows only. The reason is that the scheduling sequence of the delay queue. Several update events are blocked in the delay queue because of a small fraction of traffic, which further puts off the time at which the update events will be dispatched and slows than the average ECT of all queued update events.

V. Related Work

Consistent update. Reitblatt et al. propose a concept of per-packet/per-flow consistent update, two-phase commit protocol, and abstract a formal model for software defined network [4]. The distributed control plane [21] is implemented based on this protocol. Luo et al. concerns how to use the two-phase update mechanism into practical network when current rules are sometimes overlapped with the wildcard rules [22].

To reduce the overhead of keeping new and old configurations at related switches, McGeeret al. buffer the affected packets to the controller [23]. Katta et al. try to split update rules into several rounds and update a subset of the update rules at each round [6]. These two works have to analysis policies in advance, thus they will prolong the overall update duration. However, the duration of switch updates are proven to be non-ignorable time and high variance in [11] and [24]. On the contrary, Mahajan and Wattenhofer [25] investigate the weak-consistency policies, particularly in loop-free for destination-based routing policies.

Congestion-free update. zUpdate [10], SWAN [9], Caesar [26] and [13] propose a solution to guarantee congestion-free update. For any update event, they make a congestion-free update plan in advance, contained a sequence of intermediate states from an initial network state to the final network state. However, on the one hand, making such an update plan has to solve a series of high-complexity LPs, especially facing large-scale network and frequent network update events. On the other hand, a portion ($10\% - 15\%$) of link capacity is leaved in advance to guarantee a series of congestion-free transitions, which leads to a low network utilization [9]. The work in [27] schedules flows locally to achieve minimal amount of traffic migration and guarantee congestion-free update. They maybe able to use the new structure of the datacenter network [28] to verify methods.

Fast network update. Dionysus updates networks via dynamically scheduling consistent sequences based on different running time of each switch [11]. The timed-based update methods [7], [8] use the accurate time to trigger a network update at each phase, which effectively reduce the duration of flow rules on the switches and the update duration. B4 utilizes the custom hardware to speed up the update process [1].

The work in [12] accelerates the update process at the cost of incurring a given level of congestion. Dudycz et al. study fast network update in a round-based model, which tries to minimize the number of connections (so called rounds) between controller and network nodes [29]. The work in [30] abstracts an event level and updates traffic belonged to an update event to speed up network update. Ludwig et al. try to optimize multiple policies collectively [31]. Pereíni et al.

prioritize shorter rules to speed up whole network update [32].

All these works are based on the fact that the bandwidth is available for rescheduling an update event. In other words, we can find a migration sequence of flows and perform it. However, such a migration sequence may not exist especially handling a fairness queue of multiple update events. Then, the head-of-line problem appears. Our work performs network update under this circumstance.

VI. Conclusion

To tackle the head-of-line blocking problem while handling a fairness queue of multiple update events. we propose partial delay updating strategies to improve updating efficiency, forbidding to migrate supernumerary traffic except for the traffic caused by the update events. The designed PDU strategy schedules the update event according to their arrival order to guarantee the fairness. In addition, the blocked flows of the head-event will be skipped; hence, more updating opportunities will be provided for other update events, aiming to break the blocking state and improve the average ECT of all queued update events. The experimental results show that PDU strategy achieves $60\% - 80\%$ reduction in average ECT of update events against FIFO strategy when the sizes of flow in an update event are small, middle, big and the mixture of these three types.

Acknowledgment

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 Nature Science Foundation of China under Grant Nos. 71471175 and 61402494, Program for New Century Excellent Talents in University, Hunan Provincial Natural Science Fund for Distinguished Young Scholars under Grant No. 2016JJ1002, Research Funding of NUDT under Grant Nos. JQ14-05-02 and ZDYYJCYJ20140601, Guangxi Natural Science Foundation under Grant No. 2015GXNSFBA139243 and Science Research Foundation of Guangxi University under Grant No. XGZ141182.

References

[1] 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*, Hong Kong, China, 2013.

[2] S. Agarwal, M. Kodialam, and T. Lakshman, "Traffic engineering in software defined networks," Turin, Italy, 2013.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. USENIX*, Boston, MA, 2010.

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

[5] W. Wang, W. He, J. Su, and Y. Chen, "Cupid: Congestion-free consistent data plane update in software defined networks," in *Proc. IEEE INFOCOM*, San Francisco, CA, 2016.

[6] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *Proc. ACM SIGCOMM*, Hong Kong, 2013.

[7] T. Mizrahi, E. Saat, and Y. Moses, "Timed consistent network updates," in *Proc. ACM SOSR*, Santa Clara, CA, 2015.

[8] T. Mizrahi, O. Rottenstreich, and Y. Moses, "Timeflip: Scheduling network updates with timestamp-based TCAM ranges," in *Proc. IEEE INFOCOM*, Hong Kong, China, 2015.

[9] 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, China, 2013.

[10] 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, China, 2013.

[11] 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.

[12] J. Zheng, H. Xu, G. Chen, and H. Dai, "Minimizing transient congestion during network update in data centers," in *Proc. IEEE ICNP*, San Francisco, CA, 2015.

[13] S. Brandt, K.-T. Förster, and R. Wattenhofer, "On consistent migration of flows in sdns," in *Proc. IEEE INFOCOM*, Atlanta, GA, USA, 2016.

[14] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proc. ACM Eurosys*, Paris, France, 2010.

[15] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. ACM SIGCOMM*, New Delhi, 2010.

[16] A. Wierman and B. Zwart, "Is tail-optimal scheduling possible?" *Operations Research*, vol. 60, pp. 1249–1257, 2012.

[17] 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.

[18] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE transactions on Computers*, vol. 100, pp. 892–901, 1985.

[19] 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.

[20] T. Benson, A. A. andAditya Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Emerging Networking EXperiments and Technologies*, 2011.

[21] M. Caninio, P. Kuznetsov, D. Levin, and S. Schmid, "A distributed and robust sdn control plane for transactional network updates," in *Proc. IEEE INFOCOM*, Hong Kong, 2015.

[22] S. Luo, H. Yu, and L. Li, "Consistency is not easy: How to use two-phase update for wildcard rules?" *IEEE Communications Letters*, vol. 19, no. 3, pp. 347–350, 2015.

[23] R. McGeer, "A safe, efficient update protocol for openflow networks," in *HotSDN*, Helsinki, Finland, 2012.

[24] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about sdn flow tables," in *International Conference on Passive and Active Network Measurement*, 2015.

[25] M. Ratul and W. Roger, "On consistent updates in software defined networks," in *Proc. ACM hotNets*, 2013.

[26] G. Soudeh and C. Matthew, "Walk the line: consistent network updates with bandwidth guarantees," in *Proc. ACM HotSDN*, Helsinki, Finland, 2012.

[27] T. Qu, D. Guo, Y. Shen, X. Zhu, L. Luo, and Z. Liu, "Minimizing traffic migration during network update in iaas datacenters," *IEEE Transactions on Services Computing*, 2016.

[28] D. Guo, T. Chen, D. Li, M. Li, Y. Liu, and G. Chen, "Expandable and cost-effective network structures for data centers using dual-port servers," *IEEE Transactions on Computers*, vol. 62, pp. 1303–1317, 2013.

[29] A. Ludwig, J. Marcinkowski, and S. Schmid, "Scheduling loop-free network updates: It's good to relax!" in *Proc. PODC*, Donostia-San Sebastin, Spain, 2015.

[30] T. Qu, D. Guo, X. Zhu, J. Wu, X. Zhou, and Z. Liu, "An event-level abstraction for achieving efficiency and fairness in network update," in *Proc. ICDCS*, 2017.

[31] S. Dudycz, A. Ludwig, and S. Schmid, "Can't touch this: Consistent network updates for multiple policies," in *Proc. IFIP*, Austria, 2016.

[32] P. Pereíni, M. Kuzniar, M. Canini, and D. Kostić, "Espres: transparent sdn update scheduling," in *Proc. ACM hotSDN*, Chicago, Illinois, 2014.