Joint Route Selection and Update Scheduling for Low-Latency Update in SDNs

Hongli Xu, *Member, IEEE*, Zhuolong Yu, Xiang-Yang Li, *Fellow, IEEE*, Liusheng Huang, *Member, IEEE*, Chen Qian, *Member, IEEE, ACM*, and Taeho Jung, *Member, IEEE*

***Abstract*— Due to flow dynamics, a software defined network (SDN) may need to frequently update its data plane so as to optimize various performance objectives, such as load balancing. Most previous solutions first determine a new route configuration based on the current flow status, and then update the forwarding paths of existing flows. However, due to slow update operations of Ternary Content Addressable Memory-based flow tables, unacceptable update delays may occur, especially in a large or frequently changed network. According to recent studies, most flows have short duration and the workload of the entire network will vary significantly after a long duration. As a result, the new route configuration may be no longer efficient for the workload after the update, if the update duration takes too long. In this paper, we address the real-time route update, which jointly considers the optimization of flow route selection in the control plane and update scheduling in the data plane. We formulate the delay-satisfied route update problem, and prove its NP-hardness. Two algorithms with bounded approximation factors are designed to solve this problem. We implement the proposed methods on our SDN test bed. The experimental results and extensive simulation results show that our method can reduce the route update delay by about 60% compared with previous route update methods while preserving a similar routing performance (with link load ratio increased less than 3%).**

***Index Terms*— Route update, software defined networks, low-latency, load balancing, rounding.**

Manuscript received January 14, 2017; revised April 22, 2017; accepted June 14, 2017; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor A. Ferragut. Date of publication July 3, 2017; date of current version October 13, 2017. This work was supported in part by the NSFC under Grant 61472383, Grant U1301256, and Grant 61472385 and in part by the NSF of Jiangsu in China under Grant BK20161257. The work of X.-Y. Li was supported in part by China National Funds for Distinguished Young Scientists under Grant 61625205, in part by the Key Research Program of Frontier Sciences, CAS, under Grant QYZDY-SSW-JSC002, in part by the NSFC under Grant 61520106007, and in part by the NSF under Grant ECCS-1247944, Grant CMMI 1436786, and Grant CNS 1526638. The work of C. Qian was supported by the NSF under Grant CNS-1701681. *(Corresponding author: Hongli Xu.)*

H. Xu, Z. Yu, and L. Huang are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China, and also with the Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou 215123, China (e-mail: xuhongli@ustc.edu.cn; yzl123@mail.ustc.edu.cn; lshuang@ustc.edu.cn).

X.-Y. Li is with the School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China, and also with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616 USA (e-mail: xiangyang.li@gmail.com).

C. Qian is with the Department of Computer Engineering, University of California at Santa Cruz, Santa Cruz, CA 95064 USA (e-mail: cqian12@ucsc.edu).

T. Jung is with the Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556 USA (e-mail: tjung@nd.edu).

Digital Object Identifier 10.1109/TNET.2017.2717441

I. INTRODUCTION

SOFTWARE-DEFINED networking (SDN) is a new paradigm that separates the control and data planes on independent devices [1]. The controller provides centralized and flexible control by installing forwarding rules in the data plane, and the switches perform flow forwarding according to the rules. As the controller can provide flexible control over the entire network, it is possible for SDN to implement a wide variety of network applications ranging from basic functions (*e.g.*, routing and flow scheduling) to complex applications (*e.g.*, network function virtualization), etc. Due to frequent flow dynamics in a network [2], the data plane needs to be timely updated to avoid sub-optimal flow routes that may cause network congestion. The controller should respond to events such as shifts in traffic intensity, and new connection from hosts, by pushing forwarding rules to flow tables on the switches so as to achieve various performance requirements, such as load balancing and throughput maximization. Thus, network updates (also called route updates) help to significantly improve network performance and resource utilization [3].

The speed of network updates is an important metric in many application scenarios because it determines the agility of the control loop. The effectiveness of network updates is tied to how quickly they adapt to changing workloads. Slow network updates will make network utilization lower and decrease the route performance, such as imbalanced link utilization [1], [4]. The route update procedure consists of two main components: *route selection* in the control plane and *forwarding table update* in the data plane. Most existing update studies first compute a new route configuration *only* based on the current workload (or the collected flow intensity information) in a wide area network [1] or data center [5]. To minimize the update delay, these methods then schedule the data plane updates from the current route configuration to the new one in a fine-grained manner. For example, Hong *et al.* [1] divided all flows into minimum number of sets, and updated the flows in one set per round while avoiding the transient congestion. In [6], the number of switch interactions was minimized for consistent route update. Jin *et al.* [3] encoded the consistency-related dependencies among updates at individual switches as a graph, and dynamically scheduled these updates on different switches.

However, the previous solutions [1], [3] do not pay attention to the impact of route selection in the control plane, including

how many flows the controller will update and which path a flow will be updated to, on the update delay. Thus, they may still result in a long update duration, especially in a large and dynamic network. For example, in a moderate-size data center network, the volume of flows arriving at a switch can be in the order of 75k-100k flows/min for a rack consisting of 40 servers and the same would be 1,300k for the servers hosting around 15 virtual machines per host [7], where k denotes one thousand. At one time instant, assume that it requires to update the routes of 8k flows (less than 1%) on one switch. The update delay depends on two main factors: the total number of rules that need to be updated or inserted, and TCAM's speeds for update operations (*e.g.*, insertion or modification). By testing on today's commodity switches [3], it often takes about 5ms and 10ms for each insertion and modification, respectively, on the TCAM-based flow table. For the above scenario, assume that there needs 4k insertion operations and 4k modification operations. A rule update on this switch will hence last for at least $60s$.

Unfortunately, *a long update delay hurts the quality of route selection*. The existing studies [2] have presented the traffic characteristics by testing in various data centers. We can make several observations from existing measurement results [2]: 1) More than 80% flows last less than $10s$. 2) Fewer than 0.1% flows last longer than $200s$. 3) More than 50% bytes are in flows lasting less than $25s$. If a network-wide route update takes a long duration (*e.g.*, $60s$), the selected routes may not be useful because many flows have already terminated and many new flows have arrived [8]. As the optimal route configuration is usually derived according to the current workload [1], one route configuration is efficient for the current workload, but may be no longer efficient for another scheme after a long time duration. In other words, route updates with a shorter duration help to enhance the route performance. Therefore, low-latency network update is necessary for an SDN.

In fact, besides update scheduling, route selection also greatly impacts the update delay. First, when the controller updates more flows, though the flow routes may be optimized, the update delay will be increased significantly. Second, if only a few flows are updated, the update delay is smaller. These conclusions are also validated by testing on the SDN platform in Section VI. So, there is a trade-off between route update delay and flow path optimization. Different from these previous works, *we will consider the performance trade-off by jointly optimizing the route selection and update scheduling*. To satisfy low-latency requirements, we only update routes for a subset of chosen flows, including selecting new routes for these flows and scheduling the update operations. As a result, the final route configuration can still achieve a close-to-optimal route performance, such as load balancing, with update delay constraint. One may say that we only update the routes of those large flows (also called elephant flows [4], [5]). However, our simulation results show that the update delay of this method is still unacceptable under many network situations, especially with a large number of (elephants) flows.

To address this challenge, we formulate the delay-satisfied route update (DSRU) problem, and prove its NP-Hardness by reduction from the classical unrelated processor scheduling (UPS) problem [9]. Due to its difficulty, we design an approximation algorithm, based on the randomized rounding method [10], to solve the low-latency route update challenge through *joint optimization of route selection and update scheduling*. We show that the proposed algorithm can achieve the bi-criteria constant approximation performance under most network situations. We implement our proposed route update algorithm on a real SDN platform. The experimental results on the platform and extensive simulation results show that our algorithm can significantly decrease the route update delay while achieving similar load balance. For example, our method decreases the route update delay by 60% compared with the previous method [3] while preserving a close route performance (with link load ratio increased less than 3%).

The rest of this paper is organized as follows. Section II discusses the related works on the route update in an SDN. Section III introduces the preliminaries and problem definition. In Section IV, we propose an algorithm to deal with the DSRU problem. We study the DSRU problem with variable update delay in Section V. The testing results and the simulation results are given in Section VI. We conclude the paper in Section VII.

II. Related Works

The comprehensive survey of route update can be found in [11]. Almost all the previous methods usually compute the target route configuration based on the current workload, and design different algorithms for route update from the current route configuration to the target one. These studies can be divided into several categories by their optimization objectives, such as consistency-guarantee, and low update delay, etc.

The first category is to ensure the route consistency during the network update. Loop-freedom is the most basic consistency property and has been intensively studied. Ludwig *et al.* [12] defined the problem of arbitrary route updates, and transformed this presented problem as an optimization problem in a very simple directed graph. Reitblatt *et al.* [13] introduced two abstractions for network updates: per-packet and per-flow consistency. These two abstractions guaranteed that a packet or a flow were handled either by the current route configuration before an update or by the target route configuration after an update, but never by both. Vissicchio *et al.* designed FLIP [14], which combined per-packet consistent updates with order-based rule replacements, in order to reduce memory overhead for additional rules when necessary. Moreover, Hua *et al.* [15] presented FOUM, a flow-ordered update mechanism that was robust to packet-tampering and packet dropping attacks. Dudycz *et al.* [6] studied how to jointly optimize the update of multiple routing policies in a transiently loop-free yet efficient manner. They aimed to devise loop-free update algorithms for multiple policies in SDNs, such that the number of switch interactions was minimized. An enhanced consistency property was blackhole freedom, *i.e.*, a switch should always had a matching rule for any incoming packet, even when rules were updated. This property was easy to guarantee by implementing some default matching rule which was never updated [16].

Besides loop-freedom, some special consistency requirements are also studied for route update. Katta *et al.* [17] introduced a generic algorithm for implementing consistent updates that traded update time for rule-space overhead. They divided a global policy into a set of consistent slices and updated to the new policy of one slice at a time. By increasing the number of slices, the rule-space overhead on the switches could be reduced, and the route update delay could be increased. Mahajan and Wattenhofer [18] highlighted the inherent tradeoff between the strength of the consistency property and dependencies it imposed among rules at different switches. zUpdate [19] provided a primitive to manage the network-wide traffic migration for all the datacenter network updates. Given the end requirements of a specific datacenter network update, zUpdate would automatically handle all the details, including computing a lossless migration plan and coordinating the changes to different switches. Canini *et al.* [20] studied a distributed control plane that enabled concurrent and robust policy implementation. They introduced a formal model describing the interaction between the data plane and a distributed control plane, and formulated the problem of consistent composition of concurrent network policy updates. Since scalability was increasingly becoming an essential requirement in SDNs, the authors of [21] proposed to use time-triggered network updates to achieve consistent updates. The proposed solution required lower overhead than existing update approaches, without compromising the consistency during the update. It provided the SDN programmer with fine-grained control over the tradeoff between consistency and scalability.

The second category is to minimize the update delay while satisfying other performance requirements, such as congestion-free and consistency-conservation, etc. Hong *et al.* [1] tried to minimize the number of rounds for congestion-free update through flow splitting. They formulated the route update problem into a linear program, solved it in polynomial time, and analyzed the possibly maximum round (or delay) for route update. Jin *et al.* [3] described Dionysus, a system for fast, consistent network updates in SDNs. Dionysus encoded as a graph the consistency-related dependencies among updates at individual switches, and it then dynamically scheduled these updates based on runtime differences in the update speeds of different switches. McClurg *et al.* [22] presented an approach for synthesizing updates that were guaranteed to preserve specified properties. They formalized network updates as a distributed programming problem and developed a synthesis algorithm based on counterexample-guided search and incremental model checking. Mizrahi *et al.* [23] presented a practical method for implementing accurate time-based updates, TIMEFLIPs, which could be used to implement atomic bundle updates, and to coordinate network updates with high accuracy. A TIMEFLIP was a time-based update that was implemented using a timestamp field in a TCAM entry. TIMEFLIPs could be used to implement atomic bundle updates, and to coordinate network updates with high accuracy. Clad *et al.* [24] studied the more general problem of gracefully modifying the logical state of multiple interfaces of a router, while minimizing the number of weight updates. They presented efficient algorithms that computed minimal sequences of weights enabling disruption-free router reconfigurations. The paper [20] studies a distributed SDN control plane that enabled concurrent and robust policy implementation. Our earlier work [25] studied low-latency route update by joint optimization of route selection and update scheduling in an SDN.

Almost all the previous works update the network from the old route configuration to a new one, which is derived only based on the current workload. Though some works [3] have designed different algorithms to decrease the route update delay with consistency-guarantee, due to low-speed of TCAM operations, it may still result in a longer delay for route updates, especially in a large-scale or dynamically changed network. In most situations, the workload in a network has changed significantly after a certain period, *e.g.*, 20-60s [7]. If the route update takes a longer delay, the final route configuration may be inefficient for the workload after update. So, we need a low-latency route update for an SDN, so as to achieve the trade-off between update delay and route performance.

III. Preliminaries and Problem Formulation

In this section, we will introduce the network and TCAM update models in an SDN, describe two update requirements for congestion freedom and route consistency, define the delay-satisfied route update problem, and prove its NP-Hardness.

A. Network Model

An SDN typically consists of two device sets: a controller, and a set of switches, $V = \{v_1, \ldots, v_n\}$, with $n = |V|$. The controller determines the routes of all flows, and the switches forward packets based on the centralized route control. Thus, the network topology can be modeled by $G = (V, E)$, where E is a set of links connecting switches. When a flow arrives at a switch, if there is a matched flow entry for the header packet, the switch takes the action specified in the entry, such as forwarding packets to a certain port. Otherwise, the switch reports the header packet to the controller by the standard interface, such as the PacketIn interface in Openflow. Then, the controller determines the route path for this flow, and deploys flow entries on all switches through this path. Similar to many previous works [3], [13], we also adopt the unsplittable flow mode for its simplicity in this paper.

B. Delay Model for TCAM Updates

We introduce the delay model of TCAM updates, including insertion and modification, on a switch. The delay for each entry operation usually consists of two main parts. One is the delay for sending a control command from the controller to a switch. For example, each flow entry needs 88 bytes using the HP ProCurve 5406zl switch [26]. Then, it takes less than 0.01ms for sending a command through the control link with a bandwidth of 100Mbps. The other is delay for TCAM updates, which often take about 5-15ms on flow tables of the current commodity switches [3]. Thus, the delay for

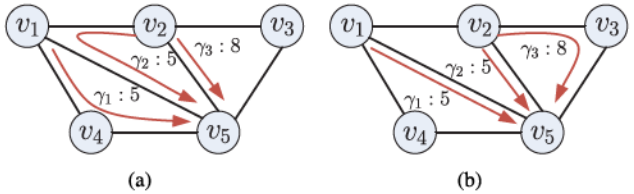


Fig. 1. An example of route update. Each link has 10 units of capacity. To avoid transient congestion, v_2 should apply the update for moving γ_3 before v_2 moves γ_2. Otherwise, link v_2 v_5 will be congested. (a) Current Route Configuration. (b) Target Route Configuration.

sending a control command can be ignored compared with the delay for TCAM updates. More specifically, Jin *et al.* [3] have shown that the delay for inserting/modifying flow entries is almost linear with the number of being inserted/modified flow entries if all flow entries have the same priority level. In fact, most flows have the unique priority level in many practical applications [27]. Even though in some applications with microflow and macroflow rule schemes [28], they have two different priority levels, the higher one for macroflows and the lower one for microflows. Since we only update some selective macroflows (or elephant flows), all of them have the unique priority level. Thus, it is reasonable to assume that the operation delay for insertion/modification of each flow entry is a constant. Let t_i and t_m denote the required delays for the insertion and modification operations of a flow entry, respectively. For example, by testing on the practical commodity switches [3], t_m may be 10ms or more on some switches due to low-speed of TCAM updates. The values of two constants t_i and t_m mainly depend on the hardware capacity. To be more practical, we will discuss how to deal with the various latency of TCAM updates in Section V.

C. Congestion-Free Route Updates [1]

We illustrate the downside of static ordering of rule updates with the example in Fig. 1. Each link has a capacity of 10 units and the size of each flow is marked. The controller wants to update the network from the current route configuration (a) to the target one (b) in Fig. 1. If we update all switches in one shot (*i.e.*, send all update commands simultaneously), since different switches will apply the updates at different times, such a strategy may cause transient congestion on some links. For instance, if v_2 applies the update for moving flow γ_3 after v_2 moves γ_2, the transient traffic load on link v_2v_5 may reach $5 + 8 = 13$, which will be congested. Thus, it requires us to carefully schedule the updates for congestion freedom.

D. Packet/Flow Consistent Route Updates [13]

When network updates are triggered, the packet/flow consistency guarantee persists: each packet (or flow) is forwarded either using the configuration in place prior to the update, or the configuration in place after the update, but never a mixture of the two [13]. This strong requirement is important for some applications such as HTTP load balancers, which need to ensure that all packets in the same TCP connection reach the same server replica to avoid breaking connections. Among many previous algorithms, the two-phase update mechanism [3], [13] has been proposed and widely used, because it can provide a simple, consistent and efficient route update way. Thus, our route update is also built on this method. To guarantee packet/flow consistency, the two-phase update mechanism should satisfy the following constraint.

Definition 1 (Consistent Update Order): Given a flow, assume that its final path after the update is $v_0 \dots v_m$, with $m \geq 1$. v_0 is the ingress switch, and others are the internal switches. The controller should start the route update on the ingress switch of this flow after the route updates are finished on all the internal switches. We call this as *consistent update order*.

The reader can refer [3] and [13] for the detailed procedure of the two-phase update method.

E. Definition of Delay-Satisfied Route Update (DSRU)

This section defines the delay-satisfied route update (DSRU) problem. In an SDN, since the header packet of each new-arrival flow will be reported to the controller, the controller saves the information of each flow. Thus, it is reasonable to assume that we know the current flow set, denoted by $\Gamma = \{\gamma_1, \dots, \gamma_r\}$ with $r = |\Gamma|$, in the network. After a flow entry is setup for flow γ on one switch, this switch can count the traffic size of this flow. By collecting the flow statistics information from switches, the controller knows the size (or intensity) of each flow γ as $s(\gamma)$. In some scenarios, the intensity of each flow may vary dynamically [27]. Thus, it is difficult to master the accurate intensity of each flow. We will discuss how to deal with the more practical case without accurate intensity information of each flow in Section IV-E. Each flow γ will be assigned a set $\mathcal{P}_\gamma$ of feasible paths, and be routed through one feasible path in $\mathcal{P}_\gamma$. We will further discuss $\mathcal{P}_\gamma$ in the next section when we present our route update algorithm.

The route update procedure can be divided into route selection and update scheduling. To obtain the trade-off optimization among route performance and update delay, we should consider the joint optimization of route selection and update scheduling. Assume that the current route configuration is denoted by $\mathcal{R}^c$, in which the route of flow γ under this route configuration is denoted by $\mathcal{R}^c(\gamma)$. To support low-latency update, we will determine a subset of flows, denoted by Γ^u, and select a feasible path as the target route of each flow. That is, the controller just updates the routes of flows in Γ^u. Assume that the target route configuration is denoted by $\mathcal{R}^f$, in which the route of flow γ is denoted by $\mathcal{R}^f(\gamma)$. The route update from $\mathcal{R}^c$ to $\mathcal{R}^f$ should satisfy the following three constraints:

- *The congestion-free constraint*: During the route update, there is no transient congestion in a network, illustrated in Fig. 1, by proper update scheduling.
- *The packet/flow-consistency constraint*: For each flow $\gamma \in \Gamma^u$, the consistent route update should be guaranteed. That is, the flow entry modification on the ingress switch should start after flow entries have been setup at all internal switches on the final route of this flow.
- *The low-latency constraint*: The maximum delay of route update on all the switches should not exceed T_0, where T_0 is the tolerated delay.

After route update, we measure the traffic load on each link e as $l(e) = \sum_{\gamma\in\Gamma, e\in\mathcal{R}^f(\gamma)} s(\gamma) \leq \lambda \cdot c(e)$, where λ is the maximum link load factor and $c(e)$ is the capacity of link e. To provide more flexible routes for new-arrival flows, our objective is to achieve the load balancing in a network, by minimizing λ.

Theorem 1: The DSRU problem is NP-hard.

We will show that the unrelated processor scheduling (UPS) problem [9] is a special case of the DSRU problem. The detailed proof has been relegated to the Appendix .

IV. Real-Time Route Update Algorithm

Due to NP-hardness, it is difficult to optimally solve the DSRU problem. This section first gives a simplified version of the DSRU problem, and explores the quantitative relationship between DSRU and its simplified version (Section IV-A). Then, we present a rounding-based route selection and update (RRSU) algorithm for the DSRU problem (Section IV-B), and analyze the approximation performance (Section IV-C). We give the complete version of the RRSU algorithm so as to satisfy both the update delay and link capacity constraints (Section IV-D). Finally, we give some discussion on our algorithm (Section IV-E).

A. Preliminaries

Since the DSRU problem requires to ensure the congestion-free and consistent update constraints for all flows, it makes problem formulation and algorithm design difficult. Thus, we first consider a simplified version of the DSRU problem, in which the controller sends all update commands and all switches execute the route updates of all flows simultaneously. This is also called Oneshot in [5]. Next, we formulate the simplified version, called S-DSRU, into an integer linear program as follows. Let variable $y_\gamma^p \in \{0,1\}$ denote whether the flow γ selects the feasible path $p \in \mathcal{P}_\gamma$ or not in the target route configuration $\mathcal{R}^f$. We use $t(v,\gamma,p)$ to express the necessary delay of flow-entry operation on switch v as the route of flow γ is updated to the target path p. Note that, if the route of flow γ does not change from the start configuration to the target one, *i.e.*, $\mathcal{R}^c(\gamma) = p$, $t(v,\gamma,p) = 0$. According to the two-phase update procedure [13], the constant $t(v,\gamma,p)$ is expressed as follows: If switch v is the ingress switch of path p, v will take the modification operation for route update of flow γ, and $t(v,\gamma,p) = t_m$. If v is the internal switch of path p, this switch will take an insertion operation. So, $t(v,\gamma,p) = t_i$. Otherwise, $t(v,\gamma,p) = 0$. We should note that two constants t_i and t_m have been defined in Section III-B. It follows

$$t(v,\gamma,p) = \begin{cases} t_m, & v \text{ is the ingress switch of path } p\ (\neq \mathcal{R}^c(\gamma)) \\ t_i, & v \text{ is one of internal switches on } p\ (\neq \mathcal{R}^c(\gamma)) \\ 0, & otherwise. \end{cases} \tag{1}$$

To pursue the low-latency feature, we expect that the route update in a network can be finished in a tolerated delay. Let variable λ be the maximum link load factor. S-DSRU solves the following problem:

$$\min\ \lambda \quad S.t. \begin{cases} \sum_{p\in\mathcal{P}_\gamma} y_\gamma^p = 1, & \forall\gamma\in\Gamma \\ \sum_{\gamma\in\Gamma}\sum_{v\in p: p\in\mathcal{P}_\gamma} y_\gamma^p \cdot t(v,\gamma,p) \leq T_0, & \forall v\in V \\ \sum_{\gamma\in\Gamma}\sum_{e\in p: p\in\mathcal{P}_\gamma} y_\gamma^p \cdot s(\gamma) \leq \lambda\cdot c(e), & \forall e\in E \\ y_\gamma^p \in \{0,1\}, & \forall p,\gamma \end{cases} \tag{2}$$

The first set of equations means that each flow will be forwarded through one feasible path from a source to a destination. The second set of inequalities denotes that the route update delay on each switch should not exceed a threshold, T_0 for the S-DSRU problem. The third set of inequalities expresses that the traffic load on each link e after update does not exceed $\lambda \cdot c(e)$, where λ is the maximum link load factor. Our objective is to achieve the load balance, *i.e.*, $\min\ \lambda$.

We use $\lambda^D(T)$ and $\lambda^S(T)$ to denote the optimal load-balance factors for the DSRU and S-DSRU problems under the delay constraint T, respectively. We have:

Lemma 2: For any delay constraint T, $\lambda^S(T) \leq \lambda^D(T)$.

Proof: Assume that a set of flows Γ^u can be updated within a delay constraint T so as to achieve the minimum load-balance factor for the DSRU problem. Since it permits to update all flows simultaneously, the total delay on each switch for S-DSRU should not exceed T if the routes for the same flow set are updated. In other words, we can at least update a flow set Γ^u within delay constraint T for the S-DSRU problem. As a result, the optimal load balance factor for the S-DSRU problem will not be worse than that for the DSRU problem under the same delay constraint. That is, $\lambda^S(T) \leq \lambda^D(T)$, $\forall\ T$. □

B. Rounding-Based Route Selection and Update

We describe a rounding-based route selection and update (RRSU) algorithm for low-latency route update with consistency and congestion-free guarantee in an SDN. Due to difficulty of the DSRU problem, the first step obtains the fractional solution for the simplified DSRU problem. In the second step, we choose one feasible path for each flow using the randomized rounding method [10], and obtain the target route configuration. Finally, we schedule the update operations of all flows on different switches so as to guarantee the congestion-free and consistency. Following [1] and [29], we assume that the controller has pre-computed a set of feasible paths between each pair of switches. These feasible paths may simply be the shortest paths, which can be found by depth-first search, between two switches. Given a flow γ, we use $\mathcal{P}_\gamma$ to be the set of feasible paths. There might be an exponential number of feasible paths between a source and a destination for each flow. Cohen *et al.* [30] have shown that polynomial number of feasible paths are enough for performance optimization. Thus, we assume that each set $\mathcal{P}_\gamma$ includes polynomial number of feasible paths for flow γ.

To solve the problem formalized in Eq. (2), the algorithm constructs a linear program as a relaxation of the S-DSRU

problem. More specifically, S-DSRU assumes that the traffic of each flow should be forwarded only through one feasible path. By relaxing this assumption, traffic of each flow γ is permitted to be splittable and forwarded through a path set $\mathcal{P}_\gamma$. We formulate the following linear program LP_1.

$$\min\ \lambda$$
$$S.t.\ \begin{cases} \sum\limits_{p\in\mathcal{P}_\gamma} y_\gamma^p = 1, & \forall\gamma\in\Gamma \\ \sum\limits_{\gamma\in\Gamma}\sum\limits_{v\in p:p\in\mathcal{P}_\gamma} y_\gamma^p\cdot t(v,\gamma,p)\le T_0, & \forall v\in V \\ \sum\limits_{\gamma\in\Gamma}\sum\limits_{e\in p:p\in\mathcal{P}_\gamma} y_\gamma^p\cdot s(\gamma)\le\lambda\cdot c(e), & \forall e\in E \\ y_\gamma^p\ge 0, & \forall p,\gamma \end{cases} \tag{3}$$

Note that, variable y_γ^p is fractional in Eq. (3). Since LP_1 is a linear program and contains polynomial number of variables, we solve it in polynomial time with a linear program solver. Assume that the optimal solution for LP_1 is denoted by $\widetilde{y_\gamma^p}$, and the optimal result is denoted by $\widetilde{\lambda}$. As LP_1 is a relaxation of the S-DSRU problem, $\widetilde{\lambda}$ is a lower-bound result for S-DSRU. Intuitively, the larger the variable $\widetilde{y_\gamma^p}$ is, the more probability the path p will be selected for this flow γ. Thus, one may say that we just select the feasible path with maximum weight for each path. However, it may lead to congestion on some links in the worst-case. To give the performance guarantee, we use the randomized rounding method for route selection to avoid the link congestion as possible. More specifically, variable $\widehat{y_\gamma^p}$, with $p\in\mathcal{P}_\gamma$, is set as 1 with the probability of $\widetilde{y_\gamma^p}$ while satisfying $\sum_{p\in\mathcal{P}_\gamma}\widehat{y_\gamma^p}=1$, $\forall\gamma\in\Gamma$. If $\widehat{y_\gamma^p}=1$, $\exists p\in\mathcal{P}_\gamma$, this means that flow γ selects p as its route.

Algorithm 1 RRSU: Rounding-Based Route Selection-Update

```
Step 1: Solving the Simplified S-DSRU Problem
Construct a linear program in Eq. (3) as Relaxed S-DSRU
Obtain the optimal solution ỹ_γ^p
Step 2: Selecting Routes Using Randomized Rounding
Derive an integer solution ŷ_γ^p by randomized rounding
for each flow γ ∈ Γ do
  for each feasible path p ∈ P_γ do
    if ŷ_γ^p = 1 then
      Appoint a feasible path p for flow γ
Step 3: Route Update Scheduling
Apply the previous Dionysus method [3] for route update
```

After the second step, we have determined the target route configuration. The third step just applies the previous Dionysus method for consistent and congestion-free route update. Specifically, Dionysus first encodes as a dependency graph the consistency-related dependencies among updates at individual switches. Then, this method dynamically schedules these updates based on runtime differences in the update speeds of different switches [3]. The RRSU algorithm is given in Alg. 1.

C. Approximation Performance Analysis

We analyze the approximate performance of the proposed RRSU algorithm. Assume that the minimum capacity of all the links is denoted by $c_{\min}$. We define a variable α as follows:

$$\alpha=\min\{\min\{\frac{\widetilde{\lambda}c_{\min}}{s(f)},f\in\Gamma\},\frac{T_0}{t_m}\} \tag{4}$$

In most practical situations, since the flow intensity is usually much less than the link capacity, for example, $c_{\min}$ =1Gbps, and $s(f)$ =4Mbps for high definition video, and t_m is usually much less than T_0, it follows that $\alpha\gg 1$. Since RRSU is a randomized algorithm, we compute the expected traffic load on links and the expected update delay on switches. We give two famous lemmas for probability analysis.

Lemma 3 (Chernoff Bound): Given n independent variables: $x_1,x_2,\ldots,x_n$, where $\forall x_i\in[0,1]$. Let $\mu=\mathbb{E}[\sum_{i=1}^n x_i]$. Then, $\Pr\left[\sum_{i=1}^n x_i\ge(1+\epsilon)\mu\right]\le e^{\frac{-\epsilon^2\mu}{2+\epsilon}}$, where ϵ is an arbitrary positive value.

Lemma 4 (Union Bound): Given a countable set of n events: $A_1,A_2,\ldots,A_n$, each event A_i happens with probability $\Pr(A_i)$. Then, $\Pr(A_1\cup A_2\cup\ldots\cup A_n)\le\sum_{i=1}^n\Pr(A_i)$.

Link Capacity Constraints: We first bound the probability with which the capacity of each link will be violated after route update. The first step of the RRSU algorithm will derive a fractional solution $\widetilde{y_\gamma^p}$ and an optimal result $\widetilde{\lambda}$ for the relaxed S-DSRU problem by the linear program. Using the randomized rounding method, for each flow $\gamma\in\Gamma$, only one path in $\mathcal{P}_\gamma$ will be chosen as its target route. Thus, the traffic load of link e from flow γ is defined as a random variable $x_{e,\gamma}$ as follows:

Definition 2: For each link $e\in E$ and each flow $\gamma\in\Gamma$, a random variable $x_{e,\gamma}$ is defined as:

$$x_{e,\gamma}=\begin{cases} s(\gamma), & \textit{with probability of}\ \sum\limits_{e\in p:p\in\mathcal{P}_\gamma}\widetilde{y_\gamma^p} \\ 0, & \textit{otherwise.}\end{cases} \tag{5}$$

According to the definition, x_{e,γ_1}, $x_{e,\gamma_2}\ldots$ are mutually independent. The expected traffic load on link e is:

$$\mathbb{E}\left[\sum_{\gamma\in\Gamma}x_{e,\gamma}\right]=\sum_{\gamma\in\Gamma}\mathbb{E}[x_{e,\gamma}]=\sum_{\gamma\in\Gamma}\sum_{e\in p:p\in\mathcal{P}_\gamma}\widetilde{y_\gamma^p}\cdot s(\gamma)\le\widetilde{\lambda}c(e) \tag{6}$$

Combining Eq. (6) and the definition of α in Eq. (4), we have

$$\begin{cases}\dfrac{x_{e,\gamma}\cdot\alpha}{\widetilde{\lambda}c(e)}\in[0,1] \\ \mathbb{E}\left[\sum\limits_{\gamma\in\Gamma}\dfrac{x_{e,\gamma}\cdot\alpha}{\widetilde{\lambda}\cdot c(e)}\right]\le\alpha.\end{cases} \tag{7}$$

Then, by applying Lemma 3, assume that ρ is an arbitrary positive value. It follows

$$\Pr\left[\sum_{\gamma\in\Gamma}\frac{x_{e,\gamma}\cdot\alpha}{\widetilde{\lambda}\cdot c(e)}\ge(1+\rho)\alpha\right]\le e^{\frac{-\rho^2\alpha}{2+\rho}} \tag{8}$$

Now, we assume that

$$\mathbf{Pr}\left[\sum_{\gamma\in\Gamma}\frac{x_{e,\gamma}}{\widetilde{\lambda}\cdot c(e)}\geq(1+\rho)\right]\leq e^{\frac{-\rho^2\alpha}{2+\rho}}\leq\frac{\mathcal{F}}{n^2} \tag{9}$$

where $\mathcal{F}$ is the function of network-related variables (such as the number of switches n, *etc.*) and $\mathcal{F}\to 0$ when the network size grows.

The solution for Eq. (9) is expressed as:

$$\rho\geq\frac{\log\frac{n^2}{\mathcal{F}}+\sqrt{\log^2\frac{n^2}{\mathcal{F}}+8\alpha\log\frac{n^2}{\mathcal{F}}}}{2\alpha},\quad n\geq 2 \tag{10}$$

We give the approximation performance as follows.

Theorem 5: The proposed RRSU algorithm achieves the approximation factor of $\frac{4\log n}{\alpha}+3$ for link capacity constraints.

Proof: Set $\mathcal{F}=\frac{1}{n^2}$. Eq. (9) is transformed into:

$$\mathbf{Pr}\left[\sum_{\gamma\in\Gamma}\frac{x_{e,\gamma}}{\widetilde{\lambda}\cdot c(e)}\geq(1+\rho)\right]\leq\frac{1}{n^4},\quad \text{where}\ \ \rho\geq\frac{4\log n}{\alpha}+2 \tag{11}$$

By applying Lemma 4, we have,

$$\begin{aligned}&\mathbf{Pr}\left[\bigvee_{e\in E}\sum_{\gamma\in\Gamma}\frac{x_{e,\gamma}}{\widetilde{\lambda}\cdot c(e)}\geq(1+\rho)\right]\\&\leq\sum_{e\in E}\mathbf{Pr}\left[\sum_{\gamma\in\Gamma}\frac{x_{e,\gamma}}{\widetilde{\lambda}\cdot c(e)}\geq(1+\rho)\right]\\&\leq n^2\cdot\frac{1}{n^4}=\frac{1}{n^2},\quad\rho\geq\frac{4\log n}{\alpha}+2\end{aligned} \tag{12}$$

Note that the third inequality holds, because there are at most n^2 links in a network with n switches. The approximation factor of our algorithm is $\rho+1=\frac{4\log n}{\alpha}+3$. □

Route Update Delay Constraints: Similar to the above analysis, we can obtain the approximation factor for the route update delay constraint.

Lemma 6: After the rounding process, the total route update delay on any switch v will not exceed the constraint T_0 by a factor of $\frac{3\log n}{\alpha}+3$ for the S-DSRU problem.

Approximation Factor: To forward all the flows on chosen paths, the above analysis shows that, the link capacity will hardly be violated by a factor of $\frac{4\log n}{\alpha}+3$, and the route update delay constraint will not be violated by a factor of $\frac{3\log n}{\alpha}+3$ for the S-DSRU problem. For simplicity, we use F_v to denote the set of updated flows whose ingress switches are v. By Eq. (3), we have the following lemma:

Lemma 7: $\sum_{\gamma\in F_v}t_m\leq T_0,\ \forall v\in V$.

According to Lemma 7, we conclude that:

Theorem 8: If we omit the congestion-free constraint during update, the RRSU algorithm can guarantee that, the link capacity will hardly be violated by a factor of $\frac{4\log n}{\alpha}+3$, and the route update delay constraint will not be violated by a factor of $\frac{3\log n}{\alpha}+4$ for the DSRU problem.

Note that, the previous Dionysus method [3] has shown that, the congestion-free and consistent constraints will not bring significant route update delay compared with simultaneously updating (*i.e.*, the Oneshot method) through efficient scheduling. As our RRSU algorithm only updates a smaller number of flows, the congestion-free constraint will also not bring significant update delay increase. In most practical situations, the RRSU algorithm can reach almost the constant bi-criteria approximation. For example, let $\widetilde{\lambda}$ be 0.4. The link capacity of today's networks will be 1Gbps. Observing the practical flow traces, the maximum intensity of a flow may reach 1Mbps or 10Mbps. Under two cases, $\frac{c^e_{\min}}{s(f)}$ will be 10^3 and 10^2, respectively. In a larger network with 1000 switches, $\log n=10$. The approximation factor for the link capacity constraint is 3.04 and 3.4, respectively. Since $\frac{T_0}{t_m}$ is usually 10^2 at least, the approximation factor for the route update delay constraint is 4.3. In other words, our RRSU algorithm can achieve the constant bi-criteria approximation for the DSRU problem in many situations.

Algorithm 2 Complete RRSU Algorithm

Step 1: the same as that in Alg. 1
Step 2: the same as that in Alg. 1
Step 3: Route Selection with Link Capacity Constraint
for each flow $\gamma\in\Gamma$ **do**
  $z_\gamma=\sum_{p\in\mathcal{P}_\gamma-\{\mathcal{R}^c(\gamma)\}}\widetilde{y^p_\gamma}$
for each flow $\gamma\in\Gamma$ in the increasing order of z_γ **do**
  The route of flow γ being updated is denoted by p
  if at least one link on path p can not contain this flow **then**
    this flow will not be updated
Step 4: Update Scheduling with Delay Constraint
Sort all the flows $\gamma\in\Gamma^u$ in the decreasing order of z_γ
The current updated flow set is Γ'
repeat
  if $\Gamma'=\Phi$ or A flow has been updated **then**
    repeat
      Dequeue next flow γ from the queue
      if Eq. (13) is satisfied for each link $e\in\mathcal{R}^f(\gamma)$ **then**
        Schedule update of this flow, $\Gamma'=\Gamma'\cup\{\gamma\}$
        Remove flow γ from queue
    until (Such a flow is not found)
  Apply the previous Dionysus method [3] for route update
until (The update delay is running out)

D. Complete Algorithm for the DSRU Problem

Though the RRSU algorithm almost achieves the bi-criteria approximation performance for the DSRU problem, the randomized rounding mechanism cannot fully guarantee that both the route update delay and link capacity constraints are always met. Below we give the complete RRSU algorithm for route selection and update which satisfies both two constraints. The complete algorithm description is given in Alg. 2.

The complete algorithm mainly consists of four steps. Same to the original RRSU algorithm, the first step constructs a linear program by Eq. (3) as a relaxation of the S-DSRU problem. Assume that the optimal solution for Eq. (3) is denoted by $\widetilde{y^p_\gamma}$. The second step will choose a feasible path for each flow using the randomized rounding method.

In the third step, we will choose a subset of flows for route update while satisfying the link capacity constraint. Let variable z_γ denote the probability with which flow γ is selected for route update. For each flow $\gamma \in \Gamma$, we compute z_γ as $z_\gamma = \sum_{p \in \mathcal{P}_\gamma - \{\mathcal{R}^c(\gamma)\}} \widetilde{y_\gamma^p}$. The algorithm sorts all flows by the increasing order of z_γ, and checks these flows one by one. For each flow $\gamma \in \Gamma$, its target route path is denoted by p after randomized rounding. If at least one link on path p can not contain this flow, we will not update the route of this flow, that is, we just set $\mathcal{R}^f(\gamma) = \mathcal{R}^c(\gamma)$. As a result, the set of all flows being updated is denoted by Γ^u.

In the fourth step, the algorithm schedules the update operations on switches while satisfying several constraints: 1) the congestion-free constraint; 2) the route consistency constraint; and 3) the low-latency constraint. We sort all flows in Γ^u by the decreasing order of z_γ, and put them into a queue. The set of flows being updated is denoted by Γ'. When Γ' is null (*i.e.*, Φ) or the route update of a flow has been finished, we choose several flows from the queue for simultaneous update without congestion, which will be described in the next paragraph. To guarantee route consistency, the update on the ingress switch should start after the updates on the internal switches are finished, which can be implemented by the previous Dionysus method [3]. Due to link capacity constraint, we may not find such a flow for update. Under this situation, a natural way is to reduce the flow rate so as to satisfy the link capacity constraint. The algorithm will be terminated until the update delay is running out.

We introduce an efficient way to choose several flows for simultaneous update without congestion. We dequeue the next flow from the queue, and schedule the route update for this flow if this flow and all flows in set Γ' can be updated without transient congestion. This procedure is terminated until we cannot find such a flow from the queue. Assume that the temporary route configuration is denoted by $\mathcal{R}^1$. For the dequeued flow γ, the transient congestion can be avoided if the following constraint is satisfied on each link e of its target route:

$$\sum_{\gamma' \in \Gamma - \Gamma'} \sum_{e \in \mathcal{R}^1(\gamma')} s(\gamma') + \sum_{\gamma' \in \Gamma'} \sum_{e \in \mathcal{R}^1(\gamma') \cup \mathcal{R}^f(\gamma')} s(\gamma') \leq c(e) - s(\gamma) \tag{13}$$

Eq. (13) means that the traffic load on link e consists of two parts: 1) For each $\gamma' \in \Gamma - \Gamma'$, the controller will not change its route, *i.e.*, $\mathcal{R}^1(\gamma')$. 2) If the controller is updating the route of flow $\gamma' \in \Gamma'$, due to asynchronous operations on different switches, we cannot determine its route, either the route before the update ($\mathcal{R}^1(\gamma')$) or the route after the update ($\mathcal{R}^f(\gamma')$).

Now, we discuss the time complexity of the complete RRSU algorithm. Cohen *et al.* [30] have shown that polynomial number (with input the number of switches) of feasible paths are enough for performance optimization. Assume that Δ is the maximum number of feasible paths for all flows. Thus, we regard that Δ is polynomial of the number of switches (n). Moreover, the number of flows is denoted by $r = |\Gamma|$. The first step mainly solves the linear program. Since the number of variables in LP_1 is polynomial value of r and Δ, it takes polynomial time of the number of flows and the number of switches to solve this LP_1. The second step uses randomized rounding for route selection, and its time complexity is $r \cdot \Delta$. In the third step, the algorithm computes a weight for each flow, and its time complexity is $r \cdot \Delta$ as well. Then, the algorithm will check all links on the selected route of each flow, and its time complexity is $\delta \cdot r$, where δ is the maximum hop number of all feasible paths. In the fourth step, the algorithm will schedule the update operations on different switches. To fulfill this function, we will check all links on the final path of a flow. Thus, it takes a time complexity of $\delta \cdot r$. As a result, the total time complexity of the RRSU algorithm is polynomial of the number of flows (r), the number of switches (n) and the maximum hop number of all feasible paths (δ).

E. Discussion

1) In the proposed update algorithm, we assume that all the switches along the final route path should take the update operations. In fact, the route update delay can be reduced by exploring route multiplexing. For a flow γ, we use $\mathcal{R}^c(\gamma)$ and $\mathcal{R}^f(\gamma)$ to denote the current path and the target path of this flow. If one switch v lies on both route paths, and its next-hop switch is same on both route paths, it is no need to update the forwarding rule of flow γ on this switch. To express this feature, we use $NS(p, v)$ to denote the next-hop switch of v on path p. Since the forwarding rules on some switches will not be changed, to guarantee the route consistency, we should determine the "ingress" switch (or the first switch to be updated) on the final route path $\mathcal{R}^f(\gamma)$ as follows: for each switch $v \in \mathcal{R}^f(\gamma)$ from the source to the destination, its next-hop switch is denoted by v'. If $e_{vv'}$ does not lie on the current path $\mathcal{R}^c(\gamma)$, *i.e.*, $NS(\mathcal{R}^f(\gamma), v) \neq NS(\mathcal{R}^c(\gamma), v)$, we determine switch v as its "ingress" switch for route update. For each switch $v \in \mathcal{R}^f(\gamma)$, if v is the "ingress" switch, there needs an update operation, and its update delay is denoted by t_m. If v is the internal switch of $\mathcal{R}^f(\gamma)$ and $NS(\mathcal{R}^f(\gamma), v) \neq NS(\mathcal{R}^c(\gamma), v)$, there needs an insertion operation, and its delay is t_i. Otherwise, there is no need for route update on this switch. Then, we can directly apply the RRSU algorithm for route selection and update.
2) Besides the two-phase update mechanism, there are other methods for consistent network update, such as [16] and [31]. Our RRSU algorithm can be easily extended to other non-two-phase update approaches. By the end of the third step, we have determined a set of flows being potentially updated. In each iteration of the fourth step, we determine a flow set Γ' with congestion-free, and construct dependence graph for Γ' according to different consistency requirements, such as congestion-free [31] or loop-freedom [16]. Then, the controller can schedule the update operations on switches by applying different consistent mechanism for route update.

V. Algorithm Extension

Jin *et al.* [3] have shown that the per-rule update latency may sometimes be varied for flow entries. We call this "the straggler case". Two main factors may lead to the straggler case. One is the switch's CPU load. Due to low CPU processing power on each commodity switch, the higher CPU load on a switch will increase the rule update latency. The other is the rule priority. The forwarding rules are stored from top to bottom in decreasing order of their priorities in the flow table [3]. When a forwarding rule will be inserted into a flow table, it may cause existing rules, especially with lower priorities, to move in the flow table, which may lead to various update latency. Accounting for these dynamic factors ahead of time is difficult. To deal with the case, we design an efficient algorithm, called RRSU-V, for route selection and update with variable update latency.

Similar to RRSU, our RRSU-V algorithm also consists of four steps. As described in [3], the update latency also depends on the current switch's state, such as the number of inserted flow entries in a flow table, and the current traffic load on this switch, etc. To express their difference among switches, each switch v will be assigned a weight ω_v, with $\omega_v \geq 1$. Intuitively, when the traffic load is high, its weight will be much larger. If the traffic load is low or it is with less control load, its weight is set as 1. Due to switch's state dynamics, it is difficult to accurately estimate the update time. According to [3], the update delay in the straggler case is about less than 2 times as that in the normal case. So, we will divide all the switches into three categories according to their traffic loads. If one switch is with a lighter traffic load, we set its weight as 1. If one switch is with a middle traffic load, its weight is set as 1.5. Otherwise, we set its weight as 2, *i.e.*, this switch with a higher traffic load. Similar to Eq. (2), we define the weighted S-DSRU problem, called WS-DSRU, as follows:

$$\begin{aligned} &\min\ \lambda \\ &S.t.\ \begin{cases} \sum_{p\in\mathcal{P}_\gamma} y_\gamma^p = 1, & \forall \gamma \in \Gamma \\ \sum_{\gamma\in\Gamma}\sum_{v\in p: p\in\mathcal{P}_\gamma} y_\gamma^p \cdot \omega_v \cdot t(v,\gamma,p) \leq T_0, & \forall v \in V \\ \sum_{\gamma\in\Gamma}\sum_{e\in p: p\in\mathcal{P}_\gamma} y_\gamma^p \cdot s(\gamma) \leq \lambda \cdot c(e), & \forall e \in E \\ y_\gamma^p \in \{0,1\}, & \forall p,\gamma \end{cases} \end{aligned} \tag{14}$$

Note that, $\mathcal{P}_\gamma$ denotes a set of permissible paths for flow γ. To solve the problem formalized in Eq. (14), the algorithm constructs a linear program as a relaxation of the WS-DSRU problem. More specifically, traffic of each flow γ is permitted to be splittable and forwarded through a path set $\mathcal{P}_\gamma$. After solving the relaxed WS-DSRU problem, we obtain the optimal solution $\widetilde{y_\gamma^p}$. Then, we can apply the following three steps of the RRSU algorithm for route selection and update scheduling. The RRSU-V algorithm is described in Alg. 3.

VI. Simulation Results

This section first introduces the metrics and benchmarks for performance comparison (*Section VI-A*). Then, we describe the implementation of delay-satisfied route update algorithms on our SDN platform, and present testing results (*Section VI-B*). We evaluate our algorithm through extensive simulations (*Section VI-C*).

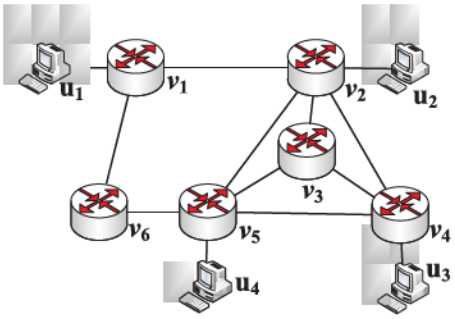


Fig. 2. Topology of the SDN Platform. Our platform is mainly composed of three parts: a controller, six OpenFlow enabled switches $\{v_1, v_2, v_3, v_4, v_5, v_6\}$ and four terminals $\{u_1, u_2, u_3, u_4\}$.

Algorithm 3 RRSU-V: Rounding-Based Route Selection and Update With Variable Update Time

1: **Step 1: Solving the Relaxed WS-DSRU Problem**
2: **for** each switch $v \in V$ **do**
3: Compute the weight ω_v for switch v
4: Construct a linear program in Eq. (14)
5: Solve Relaxed WS-DSRU, and obtain the optimal solution $\widetilde{y_\gamma^p}$
6: **Step 2: The Same as that in Alg. 2**
7: **Step 3: The Same as that in Alg. 2**
8: **Step 4: The Same as that in Alg. 2**

A. Performance Metrics and Benchmarks

Since this paper cares for low-latency route update by joint optimization of route selection and update scheduling, we adopt three main performance metrics to measure the route efficiency and update delay. The first metric is link load ratio (LLR), which can be obtained by measuring the traffic load $l(e)$ of each link e. Then, LLR is defined as: $LLR = \max\{l(e)/c(e),\ e \in E\}$. The second one is the throughput factor η, with $0 < \eta \leq 1$. This means that, for each flow γ, at least the traffic of $\eta \cdot s(\gamma)$ can be forwarded from source to destination without congestion. The last one is the route update delay, which refers the delay for the update procedure from the current route configuration to the target one.

We implement the delay-satisfied route update algorithm on both the SDN platform and Mininet [32], which is a widely-used simulator for an SDN. To show update efficiency of our RRSU algorithm, we compare it with some other benchmarks. First, the controller often determines the target route configuration based on the current workload using the different routing algorithms, *e.g.*, the multi-commodity flow (MCF) algorithm [33], and executes route updates from the current route configuration to the target one using the update scheduling algorithm, *e.g.*, Dionysus [3]. Since the controller may update all flows, including elephant flows and

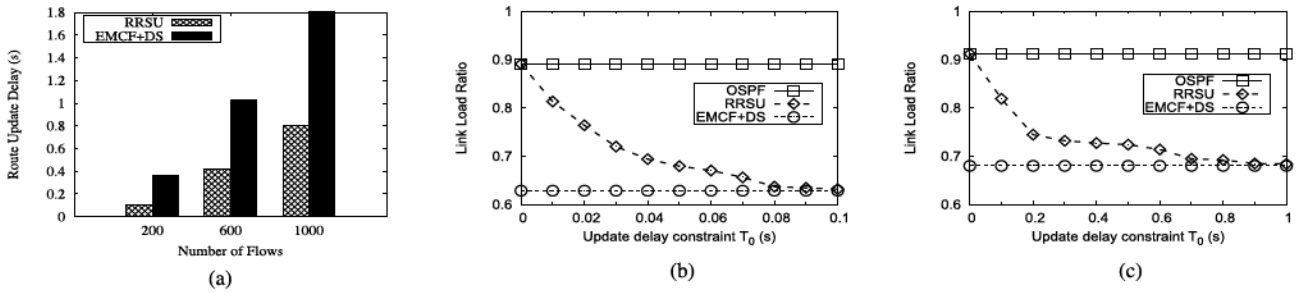


Fig. 3. Testing results through the SDN platform. (a) Route update delay vs. Number of flows. (b) Link load ratio vs. Update delay constraint with 200 flows. (c) Link load ratio vs. Update delay constraint with 1000 flows.

mice flows, by the MCF algorithm, the update delay may likely be larger. For example, our simulation results show that, when there are 40k flows in topology (b), the update delay by joint MCF and Dionysus methods may reach $65s$, which is unacceptable for many applications. An improved version is that we *only* update the routes of those elephant flows [27], denoted by EMCF, and also adopt the Dionysus method [3] for update scheduling. In this section, one flow is identified as an elephant flow, if its traffic size is more than 1Mbps. The combined method is denoted by EMCF+DS. The second one is the OSPF protocol, which only chooses the shortest path for route selection, and does not apply route updates. This benchmark is adopted for comparing the route performance of our proposed algorithm. The third one is the optimal result of the linear program LP_1 in Eq. (3), denoted by OPT. Since LP_1 is the relaxed version of the S-DSRU problem, and S-DSRU is the simplified version of DSRU, OPT is a lower-bound for both S-DSRU and DSRU problems. We mainly observe the impact of two parameters, *i.e.*, number of flows and route update delay constraint T_0, on the route update performance. Intuitively, when parameter T_0 increases, since the routes of more flows can be updated, the link load ratio will be reduced. Due to limited capacity, our commodity switch cannot support the delay measurement of insertion and modification operations on the flow table. According to the testing results on the HP ProCurve 5406zl switch [26], the delays for insertion and modification operations are set as 5ms and 10ms, respectively. We also take these results in our platform testing and simulations.

B. Test-Bed Evaluation

1) Implementation on the Platform: We implement the OSPF, EMCF+DS and RRSU algorithms on a real test-bed. Our SDN platform is mainly composed of three parts: a server installed with the controller's software, a set of OpenFlow enabled switches and some terminals. Specifically, we choose Opendaylight, which is an open source project supported by multiple enterprises, as the controller's software. The Opendaylight controller is running on a server with a core i5-3470 processor and 4GB of RAM. The topology of our SDN platform is illustrated in Fig. 2. The forwarding plane of an SDN comprises of 6 H3C S5120-28SC-HI switches, which support the OpenFlow v1.3 standard. During the platform implementation, each flow is identified by three elements, source IP, destination IP and TCP port, so that each terminal is able to generate different numbers of flows to other terminals.

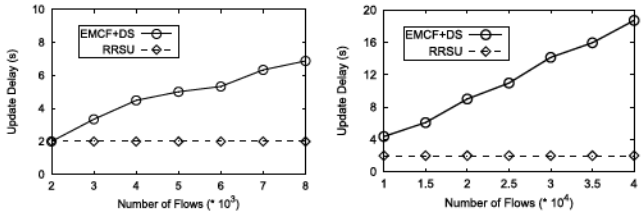


Fig. 4. Route Update Delay vs. Number of Flows under the Normal Case. *Left plot*: Topology (a); *right plot*: Topology (b).

2) Testing Results: We mainly observe the impact of update delay constraint on the performance of link load ratio. Three sets of experiments are run on the platform by generating different numbers of flows. In each experiment, there are 20% elephant flows and 80% mice flows. Fig. 3(a) shows that it takes about $0.36s$, $1.15s$ and $1.80s$ for the update procedure by the EMCF+DS algorithm when there are 200, 600 and 1000 flows in a network. Meanwhile, the RRSU algorithm takes $0.1s$, $0.42s$ and $0.80s$, respectively, so that RRSU can achieve the similar route performance as EMCF+DS (with link load ratio increased not more than 3%). Figs. 3(b) and 3(c) show that, the link load ratio is improved with the increase of the route update delay constraint by our RRSU algorithm. However, the improvement is much slower with the increasing route update delay. Note that, the route performance of EMCF+DS will not change with update delay constraint. Fig. 3(b) shows that our algorithm can reduce the route update delay by 77% compared with the EMCF+DS method while preserving a similar routing performance (with link load ratio increased about 1%). Fig. 3(c) shows that, our RRSU algorithm can achieve the close route performance as EMCF+DS (with link load ratio increased about 2%) while it reduces the update delay about 61%. From these testing results, our RRSU algorithm achieves the better trade-off performance between route performance and route update delay.

C. Simulation Evaluation

1) Simulation Setting: In the simulations, we choose two typical topologies. The first topology, denoted by (a), is derived from a commercial and operational network, which connects several data centers distributed at different locations in Beijing, China. This topology contains 20 switches and

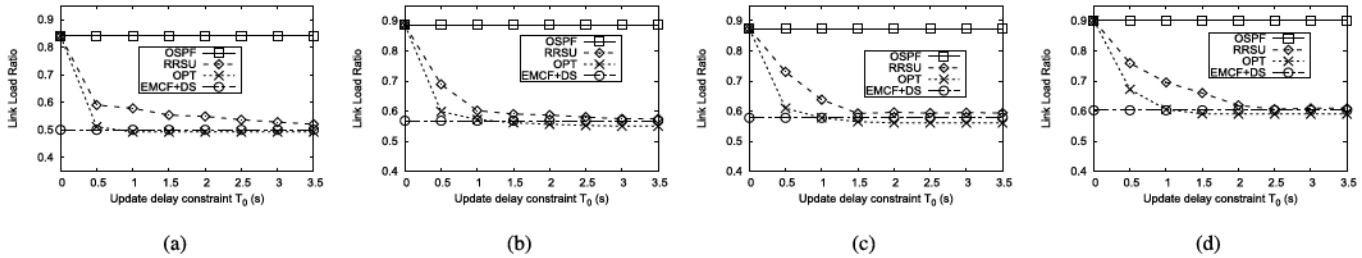


Fig. 5. Link load ratio vs. Update delay constraint for topology (a). (a) 2000 flows. (b) 4000 flows. (c) 6000 flows. (d) 8000 flows.

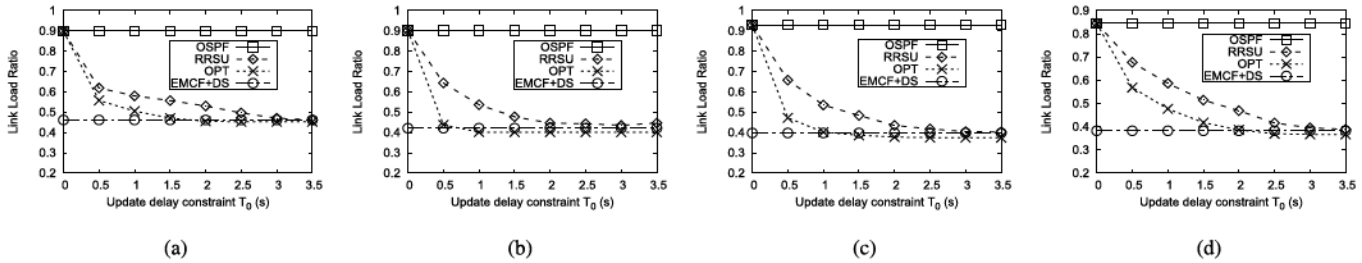


Fig. 6. Link load ratio vs. Update delay constraint for topology (b). (a) 10000 flows. (b) 20000 flows. (c) 30000 flows. (d) 40000 flows.

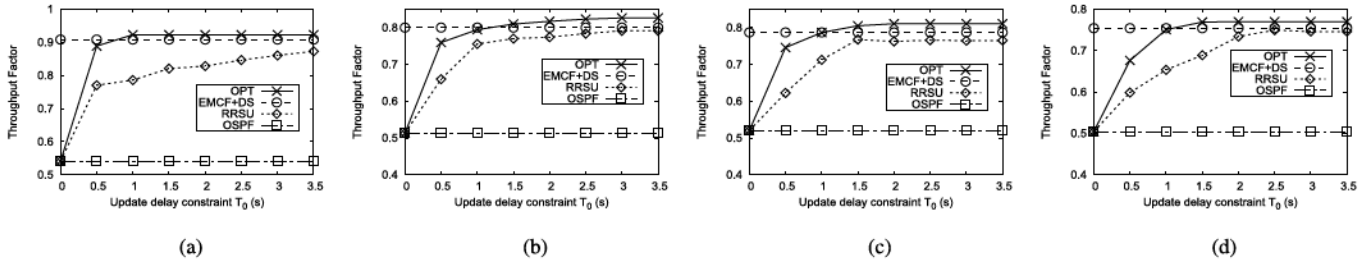


Fig. 7. Throughput factor vs. Update delay constraint for topology (a). (a) 2000 flows. (b) 4000 flows. (c) 6000 flows. (d) 8000 flows.

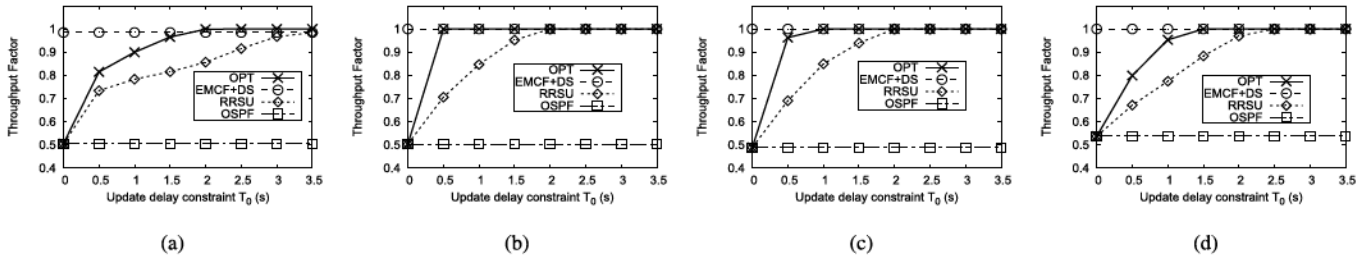


Fig. 8. Throughput factor vs. Update delay constraint for topology (b). (a) 10000 flows. (b) 20000 flows. (c) 30000 flows. (d) 40000 flows.

54 links. The second topology, denoted by (b), is derived from the Monash university [34], and contains 100 switches and 397 links. For the both topologies, each link has a uniform capacity, 100Mbps. We execute each simulation 100 times, and give the average simulation results. Curtis *et al.* [5] have shown that less than 20% of the top-ranked flows may be responsible for more than 80% of the total traffic. Thus, we generate different numbers of flows, and the intensity of each flow obeys this 2-8 distribution. Similar to [3], we show results in both normal setting and straggler setting. In the former case, the update delay setting is given in Section VI-A. For the straggler case, we draw rule update delay from [3].

2) Simulation Results for the Normal Case: We run three groups of experiments to check the effectiveness of our algorithm. The first group of two simulations shows the route update delay by varying the number of flows in an SDN. We execute two algorithms, EMCF+DS and RRSU, on two different topologies. Fig. 4 shows that the required route update delay by the EMCF+DS algorithm is almost linearly increasing with the number of flows in a network. In the large network with 40k flows, it requires more than 19s by the right plot of Fig. 4. In the following simulations, we limit the route update delay constraint no more than 4s (and 2s by default), and mainly compare the route performance with the EMCF+DS algorithm for fairness. Obviously, even though the controller only updates the routes of those elephant flows, the required update delay by EMCF+DS is still much more than the update delay constraint.

The second group of simulations mainly shows how the update delay constraint affects the route performance on two topologies. Given a fixed number of flows in a network, we change the route update delay constraints, and the link load

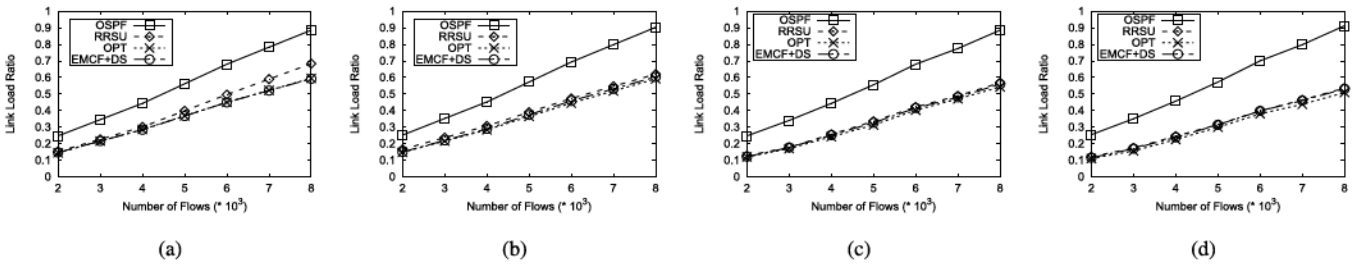


Fig. 9. Link load ratio vs. Number of flows for topology (a). (a) $T_0 = 1.0s$. (b) $T_0 = 2.0s$. (c) $T_0 = 3.0s$. (d) T_0=4.0s.

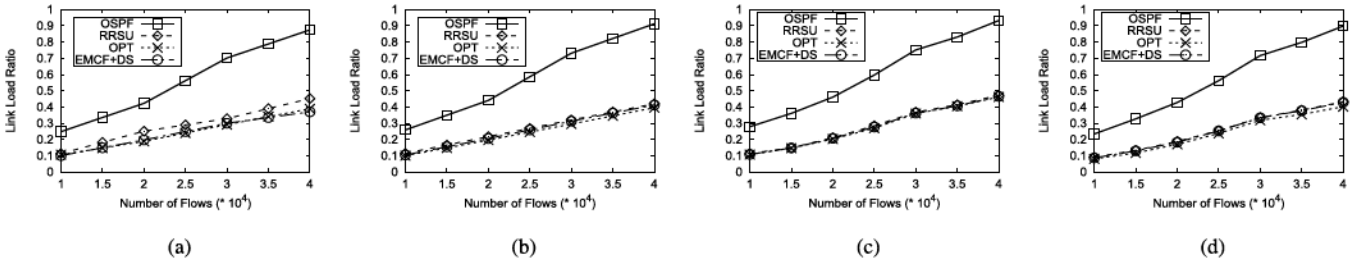


Fig. 10. Link load ratio vs. Number of flows for topology (b). (a) $T_0 = 1.0s$. (b) $T_0 = 2.0s$. (c) $T_0 = 3.0s$. (d) $T_0 = 4.0s$.

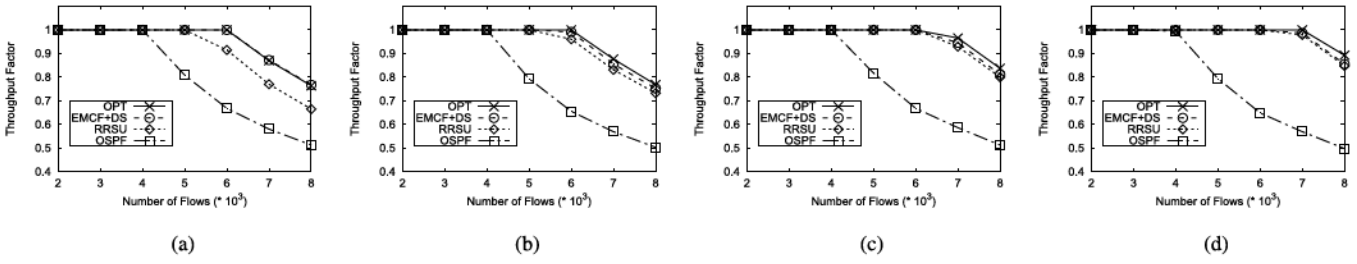


Fig. 11. Throughput factor vs. Number of flows for topology (a). (a) $T_0 = 1.0s$. (b) $T_0 = 2.0s$. (c) $T_0 = 3.0s$. (d) $T_0 = 4.0s$.

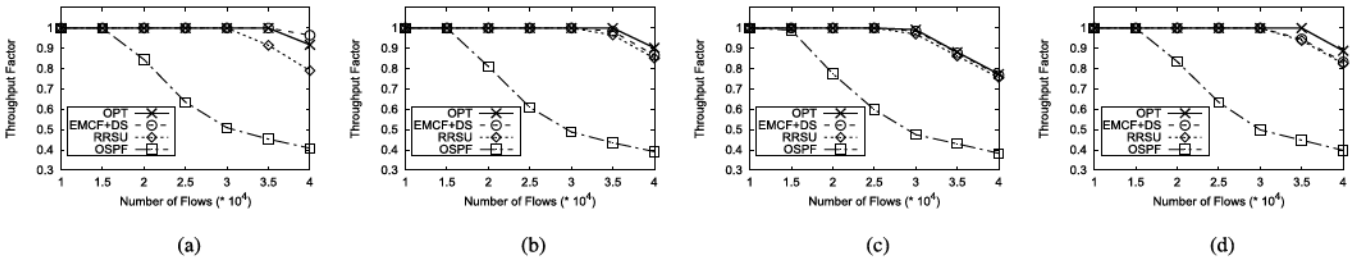


Fig. 12. Throughput factor vs. Number of flows for topology (b). (a) $T_0 = 1.0s$. (b) $T_0 = 2.0s$. (c) $T_0 = 3.0s$. (d) $T_0 = 4.0s$.

ratio performance is shown in Figs. 5 and 6. Two figures show that, the link load ratio is reduced when the route update delay becomes larger by our proposed RRSU algorithm. However, the route update delay constraint does not affect the link load ratio of the EMCF+DS algorithm, which always updates the routes of those elephant flows in a network. For the small topology, Fig. 5 shows that our RRSU algorithm reduces the route update delay by 60% compared with the EMCF+DS method while preserving a close route performance (with link load ratio increased less than 3%). For example, when there are 6000 flows in the network, the EMCF+DS method needs about 5.6s for route update by the left plot of Fig. 4. Fig. 5 shows that the RRSU algorithm can achieve the similar route performance with EMCF+DS only with a route update delay of 2s. For the large topology, we find that the RRSU algorithm can reduce the route update delay about 65% compared with EMCF+DS while still achieving the similar link load ratio performance by Fig. 6. Figs. 7 and 8 show that our RRSU algorithm can achieve the similar throughput factor compared with EMCF+DS when the update delay constraint is not less than 2s.

The third group of simulations shows how the number of flows affects the route performance on two topologies. Figs. 9 and 10 show that, the link load ratio performance of our RRSU algorithm is much closer to that of EMCF+DS with the increase of the route update delay constraint. More specifically, the RRSU algorithm with route update delay of 2s can achieve the similar route performance as EMCF+DS, which should take a route update delay of 5-8s in topology (a). In topology (b), Fig. 10 shows that our RRSU algorithm just takes about 2s for route update, so as to achieve the similar route performance as EMCF+DS, which will take a route

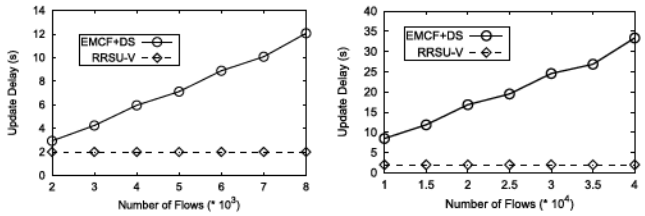


Fig. 13. Route update delay vs. Number of flows under the straggler case. *Left plot*: Topology (a); *right plot*: Topology (b).

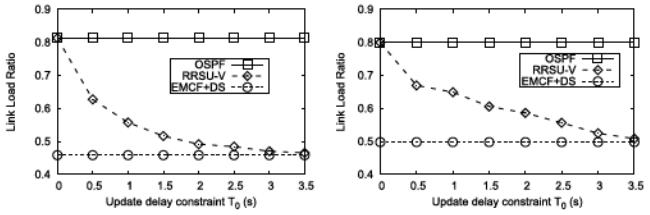


Fig. 14. Link load ratio vs. Update delay constraint for topology (a). *Left plot*: 4000 flows; *right plot*: 8000 flows.

update delay of 10-16s. Moreover, our RRSU algorithm can achieve almost the same throughput factor as EMCF+DS by Figs. 11 and 12.

From these simulation results, we can draw some conclusions. First, with the increase of the route update delay constraint, our algorithm can update more flows, and improve the route performance by Figs. 5-8. Second, Figs. 9-12 show that, with the increasing number of flows, the link load ratio will be increased and the throughput factor will be reduced under the same route update delay constraint by our RRSU algorithm. Third, Figs. 5-12 show that, the proposed algorithm almost achieves the similar performance as OPT, which is the lower bound for the DSRU problem, provided that the route update delay constraint is not too small. Fourth, Fig. 4 shows that our proposed algorithm decreases the route update delay about 60-80% compared with the EMCF+DS algorithm. However, it can reach almost the similar route performance, such as load balancing, as EMCF+DS by Figs. 5-12. Therefore, our proposed RRSU algorithm can achieve the better trade-off between the route performance and update delay by joint optimization of route selection and update scheduling.

3) Simulation Results for the Straggler Case: We run two groups of experiments to check the effectiveness of our algorithm under the straggler case. The first group of two simulations shows the route update delay by varying the number of flows in an SDN. We execute two algorithms, EMCF+DS and RRSU-V, on two different topologies. Fig. 13 shows that the required route update delay by the EMCF+DS algorithm is almost linearly increasing with the number of flows in a network. In a large network with 40k flows, it requires more than 33s by the right plot of Fig. 13. Obviously, even though the controller only updates the routes of those elephant flows, the required update delay by EMCF+DS is still much more than the update delay constraint.

The second group of simulations mainly observes how different parameters affect the link load ratio on two topologies under the straggler case. Given a fixed number of flows in a network, we observe the link load ratio performance by

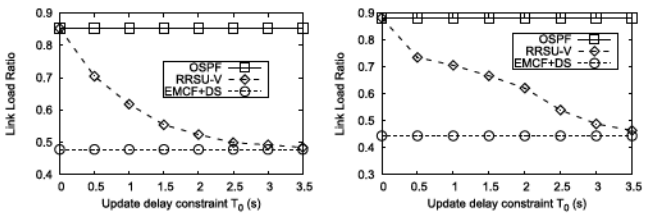


Fig. 15. Link load ratio vs. Update delay constraint for topology (b). *Left plot*: Topology (a); *right plot*: Topology (b).

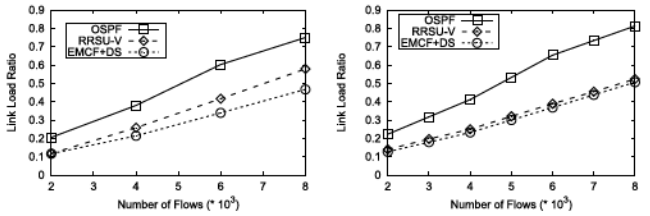


Fig. 16. Link load ratio vs. Number of flows for topology (a). *Left plot*: T_0 =1.0s; *right plot*: T_0 =2.0s.

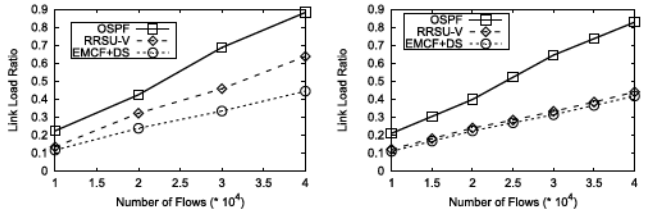


Fig. 17. Link load ratio vs. Number of flows for topology (b). *Left plot*: T_0 =1.0s; *right plot*: T_0 =2.0s.

changing the route update delay constraints. Figs. 14 and 15 show that, the link load ratio is reduced when the route update delay becomes larger by our proposed RRSU-V algorithm. We also find that the RRSU-V algorithm can achieve the similar route performance with EMCF+DS only with a route update delay of 3s. For example, for topology (a), when there are 4000 flows in topology (a), it needs about 6s for route update by the EMCF+DS method by the left plot of Fig. 13. The left plot of Fig. 14 shows that our RRSU-V method can reduce the route update delay by 50% compared with the EMCF+DS method while preserving a close route performance (with link load ratio increased less than 3%). For the large topology, we find that the RRSU-V algorithm can reduce the route update delay about 82% compared with EMCF+DS while still achieving the similar link load ratio performance from Fig. 15. Figs. 16 and 17 also show that our RRSU-V algorithm can achieve the similar route performance with EMCF+DS under different numbers of flows when we set the update delay constraint as 2s.

VII. Conclusion

In this paper, we have studied the low-latency route update while considering the current workload, the speed of TCAM updates, and the delay requirement on each switch. We have designed a rounding-based route update algorithm for the DSRU problem, and analyzed its approximation performance. The testing results on the SDN platform and the extensive simulation results have shown the high efficiency of our

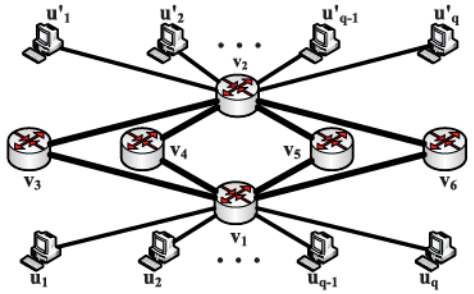


Fig. 18. A special example of the DSRU problem.

proposed algorithm. We should note that the problem that is here addressed is actually generic for any network with centralized control and low-speed flow/route table operations (*e.g.*, MPLS or GMPLS) and not limited to SDN.

Appendix
Proof of Theorem 1

Proof: We prove the NP-hardness by showing that the unrelated processor scheduling (UPS) problem [9] is a special case of the DSRU problem. We consider an example of the DSRU problem. As shown in Fig. 18, there are q flows in the network, in which each flow from u_i to u'_i, with $1 \leq i \leq q$, has a feasible path set, denoted by $\{u_i - v_1 - v_i - v_2 - u'_i, 3 \leq i \leq 6\}$. In this topology, we assume that $c(v_1v_3) = c(v_1v_4) = c(v_1v_5) = c(v_1v_6) = c_0$, where c_0 is a finite value, and others have infinite capacities. We consider a special version, in which there is no constraint on the route update delay. Then, we are possibly able to update routes of all flows for load balancing in an SDN. The controller will choose one feasible path for each flow to forward its traffic so as to minimize the maximum traffic load among links $\{v_1v_3, v_1v_4, v_1v_5, v_1v_6\}$. If we regard the flows and the link set $\{v_1v_3, v_1v_4, v_1v_5, v_1v_6\}$ as tasks and processors, this becomes the unrelated processor scheduling problem, which is NP-Hard [9]. Since UPS is a special case of the DSRU problem, DSRU is an NP-Hard problem too. □

References

[1] C.-Y. Hong *et al.*, "Achieving high utilization with software-driven wan," in *Proc. ACM SIGCOMM*, 2013, pp. 15–26.

[2] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proc. SIGCOMM Internet Meas. Conf.*, 2009, pp. 202–208.

[3] X. Jin *et al.*, "Dynamic scheduling of network updates," in *Proc. SIGCOMM*, 2014, pp. 539–550.

[4] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *Proc. 7th Conf. Emerg. Netw. Experim. Technol.*, 2011, p. 8.

[5] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," *Comput. Commun. Rev.*, vol. 41, no. 4, pp. 254–265, Aug. 2011.

[6] S. Dudycz, A. Ludwig, and S. Schmid, "Can't touch this: Consistent network updates for multiple policies," in *Proc. 46th IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jul. 2016, pp. 133–143.

[7] K. Kannan and S. Banerjee, "Compact TCAM: Flow entry compaction in TCAM for power aware SDN," in *Proc. Distrib. Comput. Netw.*, 2013, pp. 439–444.

[8] B.-Y. Choi, J. Park, and Z.-L. Zhang, "Adaptive packet sampling for flow volume measurement," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 3, p. 9, 2002.

[9] E. Davis and J. M. Jaffe, "Algorithms for scheduling tasks on unrelated processors," *J. ACM*, vol. 28, no. 4, pp. 721–736, 1981.

[10] P. Raghavan and C. D. Tompson, "Randomized rounding: A technique for provably good algorithms and algorithmic proofs," *Combinatorica*, vol. 7, no. 4, pp. 365–374, 1987.

[11] K.-T. Foerster, S. Schmid, and S. Vissicchio. (2016). "Survey of consistent network updates." [Online]. Available: https://arxiv.org/abs/1609.02305

[12] A. Ludwig, M. Rost, D. Foucard, and S. Schmid, "Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies," in *Proc. 13th ACM Workshop Hot Topics Netw.*, 2014, p. 15.

[13] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. SIGCOMM Appl., Technol., Archit., Protocols Comput. Commun.*, 2012, pp. 323–334.

[14] S. Vissicchio and L. Cittadini, "FLIP the (Flow) table: Fast lightweight policy-preserving SDN updates," in *Proc. IEEE INFOCOM*, Apr. 2016, pp. 10–15.

[15] J. Hua, X. Ge, and S. Zhong, "FOUM: A flow-ordered consistent update mechanism for software-defined networking in adversarial settings," in *Proc. IEEE INFOCOM*, Apr. 2016, pp. 1–9.

[16] K.-T. Förster, R. Mahajan, and R. Wattenhofer, "Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes," in *Proc. 15th IFIP Netw.*, 2016, pp. 1–9.

[17] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *Proc. 2nd SIGCOMM Workshop Hot Topics Softw. Defined Netw.*, 2013, pp. 49–54.

[18] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *Proc. 12th ACM Workshop Hot Topics Netw.*, 2013, p. 20.

[19] H. H. Liu *et al.*, "zUpdate: Updating data center networks with zero loss," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 411–422, 2013.

[20] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "A distributed and robust SDN control plane for transactional network updates," in *Proc. IEEE INFOCOM*, May 2015, pp. 190–198.

[21] T. Mizrahi, E. Saat, and Y. Moses. (2015). "Timed consistent network updates." [Online]. Available: https://arxiv.org/abs/1505.03653

[22] J. McClurg, H. Hojjat, P. Cerny, and N. Foster. (2015). "Efficient synthesis of network updates." [Online]. Available: https://arxiv.org/abs/1403.5843

[23] T. Mizrahi, O. Rottenstreich, and Y. Moses, "TimeFlip: Scheduling network updates with timestamp-based TCAM ranges," in *Proc. IEEE Infocom*, May 2015, pp. 2551–2559.

[24] F. Clad, S. Vissicchio, P. Mérindol, P. Francois, and J.-J. Pansiot, "Computing minimal update sequences for graceful router-wide reconfigurations," *IEEE/ACM Trans. Netw.*, vol. 23, no. 5, pp. 1373–1386, May 2015.

[25] H. Xu, Z. Yu, X.-Y. Li, C. Qian, and L. Huang, "Real-time update with joint optimization of route selection and update scheduling for SDNs," in *Proc. IEEE ICNP*, Nov. 2016, pp. 1–10.

[26] *HP Procurve 5400 ZL Switch Series*, accessed on Apr. 18, 2016. [Online]. Available: http://h17007.www1.hp.com/us/en/products/switches/HP_E5400_zl_Switch_Series/index.aspx

[27] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. Netw. Syst. Design Implement. Symp. (NSDI)*, vol. 10. 2010, p. 19.

[28] R. Narayanan *et al.*, "Macroflows and microflows: Enabling rapid network innovation through a split SDN data plane," in *Proc. Eur. Workshop Softw. Defined Netw.*, 2012, pp. 79–84.

[29] S. Agarwal, M. Kodialam, and T. Lakshman, "Traffic engineering in software defined networks," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 2211–2219.

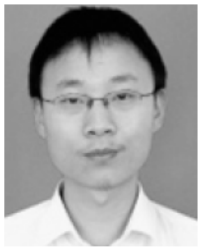
[30] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "On the effect of forwarding table size on SDN network utilization," in *Proc. IEEE Conf. Comput. Commun.*, May 2014, pp. 1734–1742.

[31] S. A. Amiri, S. Dudycz, S. Schmid, and S. Wiederrecht. (2016). "Congestion-free rerouting of flows on dags." [Online]. Available: https://arxiv.org/abs/1611.09296

[32] *The Mininet Platform*, accessed on Apr. 18, 2016. [Online]. Available: http://mininet.org/

[33] S. Even, A. Itai, and A. Shamir, "On the complexity of time table and multi-commodity flow problems," in *Proc. 16th Annu. Symp. Found. Comput. Sci.*, Oct. 1975, pp. 184–193.

[34] *The Network Topology From the Monash University*, accessed on Apr. 18, 2016. [Online]. Available: http://www.ecse.monash.edu.au/twiki/bin/view/InFocus/LargePacket-switchingNetworkTopologies



Hongli Xu (M'08) received the B.S. degree in computer science and the Ph.D. degree in computer software and theory from the University of Science and Technology of China in 2002 and 2007, respectively. He is currently an Associate Professor with the School of Computer Science and Technology, University of Science and Technology of China. He has authored over 70 papers, and held about 30 patents. His main research interest is software defined networks, cooperative communication, and vehicular ad hoc network.

Zhuolong Yu is currently pursuing the M.S. degree in computer science at the University of Science and Technology of China. His research interests include software defined networks and mobile computing.

Xiang-Yang Li (F'15) received the bachelor's degree with the Department of Computer Science and the bachelor's degree with the Department of Business Management, Tsinghua University, China, in 1995, and the M.S. and Ph.D. degrees with the Department of Computer Science, University of Illinois at Urbana–Champaign, in 2000 and 2001, respectively. He was a Professor with the Illinois Institute of Technology. He held an EMC-Endowed Visiting Chair Professorship at Tsinghua University. He is currently a Professor and the Dean with the School of Computer Science and Technology, University of Science and Technology of China. He published a monograph *Wireless Ad Hoc and Sensor Networks: Theory and Applications*. His research interests include wireless networking, mobile computing, security and privacy, cyber physical systems, social networking, and algorithms. He is an ACM Distinguished Scientist since 2014. He was a recipient of China NSF Outstanding Overseas Young Researcher (B). He and his students won several best paper awards and the Best Demo Award.

Liusheng Huang (M'07) received the M.S. degree in computer science from the University of Science and Technology of China in 1988. He is currently a Senior Professor and the Ph.D. Supervisor with the School of Computer Science and Technology, University of Science and Technology of China. He has authored six books and over 300 journal/conference papers. His research interests are in the areas of Internet of Things, vehicular ad hoc network, information security, and distributed computing.

Chen Qian (M'08) received the B.S. degree from Nanjing University in 2006, the M.Phil. degree from The Hong Kong University of Science and Technology in 2008, and the Ph.D. degree from The University of Texas at Austin in 2013, all in computer science. He is currently an Assistant Professor with the Department of Computer Engineering, University of California at Santa Cruz. His research interests include computer networking, network security, and Internet of Things. He has authored over 60 research papers in highly competitive conferences and journals. He is a member of the ACM.

Taeho Jung (M'17) received the B.E. degree in computer software from Tsinghua University, and the Ph.D. degree in computer science from the Illinois Institute of Technology. He is currently an Assistant Professor with the University of Notre Dame. His research area includes privacy and security issues in the big data ecosystem. His paper has won the Best Paper Award from the IEEE IPCCC 2014, and two of his papers were selected as a Best Paper Candidate from the ACM MobiHoc 2014 and the Best Paper Award Runner Up from BigCom 2015, respectively. He has served many international conferences as a TPC Member, including the IEEE DCOSS, the IEEE MSN, the IEEE IPCCC, and BigCom. He is a member of the ACM.