Arrange Your Network Updates as You Wish (Extended version)

Shouxi Luo, Hongfang Yu, Long Luo, Lemin Li
Key Laboratory of Optical Fiber Sensing and Communications, Ministry of Education
University of Electronic Science and Technology of China, Chengdu, P. R. China

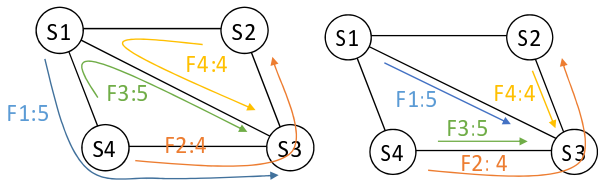
***Abstract*—Updating network configurations responding to dynamic changes is still a tricky task in SDN. During the update process, in-flight packets might misuse different versions of rules, and "hot" links could be overloaded due to the unplanned update order. As for the problem of misusing rule, recently proposed suggestions like *two-phase mechanism* and *Customizable Consistency Generator (CCG)* have provided *generic* and *customizable* solutions. Yet, there does not exist an approach that is flexible to avoid the transient congestion on hot links respecting to diverse user requirements like guaranteeing update deadline, managing transient throughput loss, etc.; controllers urgently need one.**

In this paper, we propose CUP, Customizable Update Planner, to seek the solution. Different from prior approaches that adopt fixed designs for a single purpose like optimizing the update speed (e.g., Dionysus) or avoiding congestions (e.g., zUpdate, SWAN), CUP introduces generic linear programming models to formulate user-specified requirements and the update planning problem. By solving these customized models, CUP is able to plan network updates according to a large fraction of user requirements, such as guaranteeing deadlines, prioritizing operation orders, managing throughput loss, etc., while avoiding transient congestion. We prototype CUP on Ryu and employ it to arrange updates for networks built upon Mininet. Results confirm the flexibility of CUP while indicating that it always obtains the "best" update plans following the user's wish.

I. Introduction

Reconfiguring forwarding rules in networks responding to dynamic demands such as periodical traffic optimization, unexpected failover, is always a tricky task for operators [1]–[6]. Recent trends toward Software Defined Networking (SDN) seem to provide a promising solution for network management—with a logical central controller, operators can directly operate the forwarding rules on all switches. Even so, the network is still an asynchronous system in essence. It is difficult to synchronize the changes to flows from different ingress switches. Therefore, when migrating a group of flows to their new paths, even if the network is safe both before and after the reconfiguration, some "hot" links could be overloaded during the update process in case new flows move in before those old ones move out [2]–[4].

As an example, consider the toy case shown in Fig. 1. On executing WAN optimizations [3], the controller wants to update the network's configuration from Fig. 1a to Fig. 1b. For simplicity, we assume that the network uses tunnel-based routing and all necessary tunnels have already been established. If the controller carries out the update in one-shot, link S4-S3 or S1-S3 might be overloaded during the update, corresponding to the case that switch S4 happens to change



(a) Current State (b) Target State

Fig. 1: A network update example. Each link has 10 units of capacity and flows are labeled with their sizes. If the controller carries out the update in one-shot, link S1-S3 or S4-S3 will be overloaded during the update.

F3 to its new path before S1 moving F1 away from link S4-S3, or vice versa. The congestion can not be evaded by simply letting F1 and F3 be switched to their new paths at exactly the same time [7]—because the incoming packets of F2 and F3, together with the in-flight packets of F1, could still congest S4-S3 until F1 drains; and so does S1-S3.

Such a type of congestion disappears following the completion of update, but its destructibility lasts long—burst traffic leads to serious queuing delay, and even, packet drops, which will let involved TCPs' windows collapse, or worse, kill flows. These bad influences are not desirable, especially for real-time applications. Accordingly, carrying out network reconfigurations without introducing transient congestion is a fundamental function required by SDN controller.

Planning network updates to avoid transient congestion is never an easy task. Recent approaches like zUpdate [2] and SWAN [8] try to solve the problem by introducing a sequence of intermediate configurations, among which, the update from a former stage to the latter must always be congestion-free. To ensure such a stage sequence exists, they require part of the link capacity to be left vacant, which results in a great waste of link capacities [8, 9]. Furthermore, the intermediate configurations they introduce will greatly complicate the update process, and might even disturb user's QoS—e.g., an intermediate path might have a larger latency than both the initial and target ones. In contrast, Dionysus [3] and ATOMIP [4] address the challenges by scheduling updates in thoughtful orders without bringing in additional stages. For instance, by executing the update illustrated in Fig. 1 following the 3-round sequence of [F4→F1→F3], no link would be overloaded and no extra paths are introduced. Order arrangement provides a more practical solution. However, it is not always the panacea because such a congestion-free operating sequence does not always exist.

Indeed, due to the various update scenarios and user demands that a controller would deal with, simply arranging the update operations, or introducing intermediate stages, is far from enough for a practical solution. We argue that, a practical planner should have these properties.

1) Effective to handle deadlock and deadline. First of all, the planner must be able to find feasible congestion-free solutions for any given task. On one hand, in some update scenarios, there does not exist a congestion-free sequence [3, 4]. For instance, in the case of Fig. 1, if the demand of either F1 or F3 increases to 6, it is impossible to migrate the network to its target routing state by arranging the execution order without overloading S1-S3 or S4-S3. This is a deadlock in update planning. On the other hand, even though congestion-free schedules are found, they may not meet the deadline requirements. This is because to remove overloads, the controller can not switch flows belonging to round-$(i+1)$ to their new paths until flows moved out from these paths in round-i have exited. Suppose in-flight packets require about τ units of time to exit from a path on average; then, it would take about $k \cdot \tau$ for the entire network to perform a k-round update. Such an update delay/duration might be unacceptable for time-critical cases like failover routing [10]. Therefore, *on planning updates, the planner should have the ability to break deadlocks and guarantee deadlines.*

Fortunately, for any update, by limiting the rates of some flows at their senders or traffic shapers, controllers can always obtain a congestion-free update sequence that involves fewer rounds and satisfies the deadline requirements. Indeed, there is a trade-off between the time an update takes, and the throughput the network has to drop (induced by congestion or rate-limiting). For example, one can carry out the update request demonstrated in Fig. 1 within 2 rounds by limiting the rate of either F2 or F4 to 0 (e.g., when F2's rate is limited to 0, [F3→F1, F4] is congestion-free), or even perform the update within 1 round by limiting the rate of both F2 and F4 to 0. This example gives us a valuable insight: *the planner should have the ability to trade throughput loss for update speed*.

2) Expressive to deal with user-specified requirements. As infrastructure, today's network is shared by numerous customers while simultaneously carrying various kinds of traffic. To be a universal tool for controller, the update planner should be extensible and easy to adapt to user-specified requirements. As an example, consider the case of removing transient congestion for the update illustrated in Fig. 1 again. Provided the reconfiguration is time-sensitive and required to complete within 1 round, the controller has to reduce some flow rates to avoid congestion. Suppose this is an instance of inter-datacenter traffic optimization [8], in which both F1 and F3 are *interactive* traffic while F2 and F4 are *background* traffic, and the operator prefers to minimize the amount of interactive traffic disturbed by the update. In such a scenario, the planner should temporarily reduce the rates of F2 and F4 to 0 to execute the update, i.e., limit the rates of {F1, F2, F3, F4} to $\{5, 0, 5, 0\}$. On the contrary, if F2 and F4, instead of F1 and F3, are interactive, the result would be $\{1, 4, 1, 4\}$. As another example, if all flows share the same class and a fairness alike policy is expected [11], the planner should set their rates to $\{\frac{5}{14}, \frac{4}{14}, \frac{5}{14}, \frac{4}{14}\}$, with the target of letting the decrease of throughput be fairly shared in proportion.

Indeed, due to network's diversity, such a special constraint of rate-limiting is only the tip of an iceberg. In practice, there are plenty more kinds of user-specified demands (about the update execution time or throughput loss) that a controller would deal with. It follows that, *on planning rate-limiting schemes, the planner should be flexible enough to suit various update scenarios, as well as user-specified demands*.

3) Efficient to scale up. Last but not least, to be practical, the planner must be time-efficient to find feasible solutions for update requests in time. In consideration of that the size of today's network might be really huge (e.g., Datacenter or backbone), the planner needs to easily scale up.

As the first step, this paper proposes CUP, Customizable Update Planner, to help controller deal with various updating requirements. CUP suggests adopting generic methods such as *two-phase mechanism* [5, 6] to enforce rule consistency, and focuses on eliminating the transient congestion during updates. Distinguished from existing solutions proposed for fixed targets, CUP is effective and expressive to deal with deadlock, deadline, prioritization, and many other user-specified requirements as Table I summaries (Note that, proposals focusing on enforcing rule consistency are not listed, e.g., CCG [12].). We analyze various demands and realize that, besides consistency, what users/operators concern about the implementation of an update, no matter how complex it is, generally involves two types of fundamental issues—i) *when a flow could enjoy its new path(s)* and ii) *how its throughput would be impacted during the update process?*

At a high-level, CUP provides an expressive user-friendly language, with which, customers and operators can describe their own requirements easily and explicitly. When the network is to be updated, CUP maps these high-level requirements into the essence (involved) flows, and translates them into low-level linear constrains. At its core, CUP builds a couple of generic linear programming models to formulate the update request while capturing constrains from users. Via solving these customized models, CUP obtains a congestion-free update execution plan that explicitly follows the user's wish.

Roughly, CUP's model involves two parts, *Order Scheduler* and *Rate Manager*, which respectively answer the two basic problems mentioned above. On planning an update, *Order Scheduler* first determines the operation order respecting to time-related requirements. If congestion-free sequences are found, *Order Scheduler* outputs the one involving the minimum rounds; otherwise, it chooses the sequence causing least overload on links. For the overloaded traffic, *Rate Manager* then figures out the optimal rate-limiting scheme that is able to erase the congestion while satisfying all throughput-related requirements. As the core of both *Order Scheduler* and *Rate Manager* is to solve a single Linear Program (LP), with high performance LP solvers, CUP obtains solutions within polynomial time and is able to scale up.

We prototype CUP upon Ryu[1] and use it to plan updates for

[1]An open-source SDN controller framework: https://osrg.github.io/ryu/

TABLE I: Summary of previous approaches and comparison to CUP.

#Proposal	Introduce intermediate status?	Effectiveness		Expressiveness
		Handle deadlock	Deal with deadline	Meet user-specified requirements
zUpdate [2]	Yes	No	No	No
SWAN [8]	Yes	Yes	Single deadline for all	No
GI [9]	Yes	Yes	Single deadline for all	No
Dionysus [3]	No	Yes	No	No
ATOMIP [4]	No	Yes	Single deadline for all	No
CUP	**No**	**Yes**	**Per-flow deadline**	**Yes (any time- and rate- related requirements)**

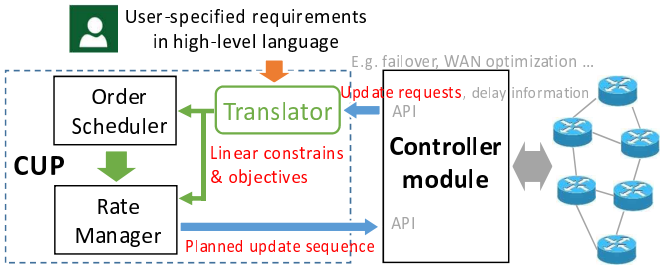


Fig. 2: The workflow of CUP on planning updates.

Grammar

pol	::=	$(s_1; \ldots; s_n)$	CUP Policy
s	::=	$t \mid r$	Rule
t	::=	$T(m) \leq val \mid T(m_1) \leq T(m_2)$	Time Related Req.
r	::=	$R(m) \geq amap \mid R(m_1) \geq R(m_2)$	Rate Related Req.

Notation

m	:	a match string/predicate specifying flows
val	:	a value specifying a deadline requirement
$T(m)$	:	the waiting time before m enjoys the new path(s)
$amap$	:	the keyword specifying objectives (as much as possible)
$R(m)$	:	the rate-limit setting of flow(s) defined by m

Fig. 3: Syntax of CUP high-level language.

networks conducted by Mininet [13]. Results show that CUP is quite flexible to exactly meet user-specified requirements, while effective to outperform existing approaches.

In summary, we make three contributions in this paper.

- **Abstraction:** We show how to express various user-specified updating requirements with a high-level language, and show how to dynamically translate them into low-level linear constraints (Section II).
- **Model:** We propose generic linear programming models to formulate and solve the customized update planning problem, with which, controllers obtain the "best" update plan explicitly following user's wish (Section III).
- **Evaluation:** We show that our CUP tool is flexible and effective to make update plans for "real" networks built by Mininet (Section IV).

II. FLEXIBLE CUP

In CUP, network users as well as operators describe their desired properties about the update with the high-level CUP language; they can change the clauses at any time. On planning a network update, at the first step, CUP "compiles" the user's codes to figure out their exact "meaning" in this instance. After that, CUP employs back-end solvers, *Order Scheduler* and *Rate Manager*, to find the update processing plan that exactly follows the user's wish. Roughly, the entire workflow of how CUP produces is as Fig. 2 shows.

In the following, we present the high-level language in Section II-A and show the compilation process in Section II-B. After that, in next section, we introduce how CUP solves the planning problems and discuss how it handles multi-tenants and concurrent update requests.

A. High-level language

CUP language (Fig. 3) provides end-users and operators with an easy way to specify their requirements on configuring the network. A CUP policy is a collection of rules, in which, each term specifies a specific requirement, of either the activation time of new paths or the degradation of throughputs, for a group of packets. CUP uses a regular expression on the match fields of packet header to define the involved packets. For instance, $*$ defines all packets pass through the network; dstTCP=80 defines all web access traffic; srcIP=10.0.0.1/24 $\wedge$ dstIP=20.0.0.11 defines those packets from network 10.0.0.1/24 to destination 20.0.0.11; and srcIP=10.0.0.2 $\vee$ dstIP=10.0.0.4 defines the traffic from 10.0.0.2, or to 10.0.0.4.

For the update of a collection of packets specified by m, there are two basic types of indicators that customers and operators might concern: 1) how long it would wait before enjoying the new path(s), and 2) how its throughput (i.e., rate) would be limited to avoid transient congestion. CUP uses $T(m)$ and $R(m)$ to denote, respectively. Using their relation expressions, these two basic elements can generate other complicated requirements. For instance, $T(m_1) \leq T(m_2)$ says, flows matched with m_1 should be switched into the new paths "no later than" those matched with m_2, while $T(m_2) \leq val$ indicates the waiting time before m_2 switched should be "no larger than" val. Similarly, $R(m_1) \geq R(m_2)$ implies the effective bandwidth of m_1 during the update should "no less than" that of m_2, while $R(m_3) \geq amap$ means the user would like the effective bandwidth of m_3 be maximized.

CUP language is simple yet expressive for most requirements. As examples, revisit the toy update cases of Fig. 1. With CUP language, users can formulate their own requirements precisely and concisely as the instances in Table II illustrate.

B. Dynamic translator

High-level CUP policies are compiled into low-level restrictions, which tell the planner how to process each flow's reconfiguration is in line with user requirements. To achieve this, the most challenging task is to figure out the exact time

TABLE II: Examples of CUP language on describing update cases shown in Fig. 1.

#	Update scenarios	Policy expression
1	Minimize transient congestion without deadline requirements on the update process.	$(R(*) \geq amap)$
2	Let interactive flows,F_2 and F_4, enjoy new paths no later than 1 unit time, while minimizing the impacts on their throughputs (e.g., inter-DC WAN optimization [8, 14]).	$(T(m_{F2} \vee m_{F4}) \leq 1;$ $R(m_{F2} \vee m_{F4}) \geq amap)$
3	Execute all flow migrations no later than 1 unit time, and let the throughput loss be shared in proportion since they are in the same class.	$(T(*) \leq 1;$ $R(m_{F1}) \geq amap;$ $R(m_{F2}) \geq amap;$ $R(m_{F3}) \geq amap;$ $R(m_{F4}) \geq amap)$

cost of migrating a flow. CUP employs the approach of pre-installing new rules then triggering two-phase reconfigurations to address the problem. In this part, we first present how to make the estimation of reconfiguration's time cost possible in Section II-B1, then introduce the way of binding high-level requirements with flows and translating them into low-level linear constraints in Section II-B2.

1) Estimating time cost of traffic migration: As Section I and Fig. 1 have shown, to not overload any link during the update, the controller has to wait the flow that moved out from a link exits, before moving other flows in. Thus, the time cost of migrating a flow to its new path(s) mainly involves two parts of i) waiting the moved-out traffic exits (if any); and then ii) installing rules to shift the flow to its new path(s).

As for the first part of draining time, we can simply use the well-known One-Way Delay (OWD) as an approximation, which can be estimated at end hosts [15, 16], or at edge switches in OpenFlow-enabled networks. CUP suggests adopting *two-phase update mechanism* to guarantee strong rule consistency (refer to Appendix A in [17] for the discussion). On carrying out an N-rounds flow migration, at the first step, CUP pre-installs the new configurations and sets rate-limits. Supposing the time of installing/modifying a rule from the controller is ϵ, the total time cost of this step is ϵ because all rule installations (for both new paths and rate-limits) can perform in parallel. Thus, the rest operations for each round are to i) wait a draining time then ii) touch some flows' ingress switches to activate their new paths. Provided the largest OWD in network is τ, we get the point that flows migrated in the kth round would enjoy their new paths at time $k \times \tau + (k+1) \times \epsilon$. Consequently, if a flow's deadline requirement on the update process is val, we know that the controller should make sure it get migrated no later than round $\lfloor \frac{val-\epsilon}{\tau+\epsilon} \rfloor$.

In practice, the time cost of modifying a rule on physical switches is usually inconstant [3, 14, 18, 19]. Yet, recent studies have shown its long-tailed characteristic [3]. That is to say, simply choosing the 95th percentile value (or other thresholds) as the estimated time is reasonable in most cases. Moreover, since OpenFlow-style control is still in its early stages, most switch software and SDKs are not optimized for dynamic table programming yet [14]. Some effects have been put on improving this and we argue that future switches will be more stable and fast for table changes [18, 20].

TABLE III: The key notations of the network model

Notation	Description
M_R^{due}	the set of predicates (m) holding $T(m) \leq val$
M_R^{amap}	the set of predicates (m) holding $R(m) \geq amap$
MP_T	the set of $\langle m_x, m_y \rangle$ pairs holding $T(m_x) \leq T(m_y)$
MP_R	the set of $\langle m_x, m_y \rangle$ pairs holding $R(m_x) \geq R(m_y)$
$\hat{N}_m^{due}$	the round deadline for flow matching with m
$f \in F$	the set of all current flows in the network
$F(m)$	the set of all flows matching with predicate m
t_f	the demand of flow f
r_f	the rate-limit setting of f during the update
r_m^*	the rate-limit setting for all flows matching with m
$e \in E$	the set of all (directed) links in the network
c_e	the capacity of link e
$t_{f,e}$	the load of f on link e before the update
$t'_{f,e}$	the load of f on link e after the update
F^B	the set of flows that will not be updated/migrated
F^U	the set of flows that will be updated/migrated
$F^U(m)$	the set of to-be-updated flows matching with m
FP_T	$\forall \langle f_i, f_i \rangle \in FP_T$: f_i should be updated no later than f_j
N_f^{due}	f's update deadline, in the form of round number
$y_{f,k}$	whether f has been updated in round-k
$t_{f,e,k}$	the (maximum) load of f on e in round-k

As yet, we have found a way to estimate the time cost of migrating a flow based the network's maximum OWD and ingress's rule modification delay. In real networks, both types of delays can be measured by the controller. With this information, CUP is able to translate the absolute deadline requirements into round requirements. For simplicity, hereafter, all deadline requirements we discuss in this paper are in the form of round number.

2) Mapping requirements to each flow: Now, we show how CUP maps user requirements into each flow. The basic notations that CUP's model uses are summarized in Table III.

Lexical analysis and preprocessing. CUP first parsers user-specified policies to get the semantics. Obviously, there are four types of constraints on the flow predicates, indicating the absolute update deadline (i.e., $T(m) \leq val$), the relative update order in "no-later-than" form (i.e., $T(m_x) \leq T(m_y)$), relative rate-limiting setting in "no-less-than" form (i.e., $R(m_x) \geq R(m_y)$), and the expected targets that should be optimized (e.g., $R(m) \geq amap$). Without loss of generality, we let M_T^{due} be the set of predicates holding the relation of $T(m) \leq val$, and M_R^{amap} be the set of predicates holding $R(m) \geq amap$. As well, we further use MP_T and MP_R to denote the set of predicate pairs (e.g., $\langle m_x, m_y \rangle$) that have the relation of $T(m_x) \leq T(m_y)$ and $R(m_x) \geq R(m_y)$, respectively. As discussed above, for a deadline requirement on flows specified by predicate m, CUP can transfer it into a round number requirement with Equation (1), where $\hat{\tau}$ is the network's measured maximum OWD and $\hat{\epsilon}$ is the measured 95th rule modification delay.

$$\hat{N}_m^{due} = \lfloor \frac{val_m - \hat{\epsilon}}{\hat{\tau} + \hat{\epsilon}} \rfloor \tag{1}$$

Basic network model. We assume that the network, G, is hosting a set of flows F with links E. The rate of flow $f \in F$ is denoted by t_f while the capacity of link $e \in E$ is denoted by c_e. By letting $t_{f,e}$ be the traffic load of

flow f on link e, the network's state can be formulated as $S = \{t_{f,e} | \forall (f \in F, e \in E)\}$. Then, a network update is to change its state from S to $S' = \{t'_{f,e} | \forall (f \in F, e \in E)\}$ by rerouting some flows, or changing their traffic split ratios in the case of multi-path routing. For the update of $S \mapsto S'$, let F^U be the set of updated flows and F^B be the set of unmodified flows. Obviously, there must be $F^U \cap F^B = \emptyset$, $F^U \cup F^B = F$, and $t_{f,e} = t'_{f,e}$ for $\forall (f \in F^U, e \in E)$. We assume that the update of flow f is required to be finished within N_f^{due} rounds, and use bin variable $y_{f,k} (1 \leq k \leq N_f^{due})$ to indicate whether f ($f \in F^U$) has been migrated/updated in the k-th round. By defining $y_{f,0} = 0$ for convenience, we get the constraints as Equation (2) and (3) show.

$$\forall k, f \in F^U : y_{f,k} \in \{0, 1\} \tag{2}$$

$$\forall f \in F^U : 0 = y_{f,0} \leq y_{f,1} \leq \ldots \leq y_{f,N_f^{due}} = 1 \tag{3}$$

Besides, we let r_f denote the proportion of rate-limiting that flow f would be set to during the update. Then, after rate-limiting is enabled, the total load of f would be reduced to $t_f \cdot r_f$, and the subpart on e before and after the update would also decrease to $t_{f,e} \cdot r_f$ and $t'_{f,e} \cdot r_f$, respectively.

$$\forall f : 0 \leq r_f \leq 1 \tag{4}$$

Embedding user-specified requirements. In networks, flows are also defined by predicate strings of the packet header fields. By checking whether a flow's predicate intersects with the user-specified predicate, CUP figures out which flows are involved with that rule. For rule predicate string m, we denote $F(m)$ as the set of flows that it intersects with, and $F^U(m)$ as the subpart of to-be-updated flows in $F(m)$. Then, via Equation (5), CUP gets the set of rules that a flow is matched with and gets the exact deadline requirement of each flow. It should be noted that, the entire update process will never exceed $|F^U|$, the number of flow to be updated. So, in case the estimated round calculated from user policies is larger than F^U, or no deadline is required, N_f^{due} will be set to $|F^U|$.

$$N_f^{due} = \min(|F^U|, \min_{\forall m \in M_T^{due} : f \in F^U(m)} \hat{N}_m^{due}) \tag{5}$$

As for the "no-later-than" order requirements, $T(m_x) \leq T(m_y)$, if two to-be-updated flows, f_i and f_j, happen to hold the relations of $f_i \in F^U(m_x)$ and $f_j \in F^U(m_y)$, it means they have order-dependency on the update active time, namely, $y_{f_i,k} \geq y_{f_j,k}$ for all feasible k. Let FP_T be the set of such order-dependent flow pairs; CUP can easily get it by calculating Equation (6). Then, all "no-later-than" requirements are as Equation (7) shows.

$$\begin{aligned} FP_T = \{\langle f_i, f_j \rangle | \exists \langle m_x, m_y \rangle \in MP_T; \\ f_i \in F^U(m_x); f_j \in F^U(m_y)\} \end{aligned} \tag{6}$$

$$\forall (f_i, f_j) \in FP_T, k \leq \min(N_{f_i}^{due}, N_{f_j}^{due}) : y_{f_i,k} \geq y_{f_j,k} \tag{7}$$

Now, CUP deals with rate/throughput related requirements. Same to the case of time-related predicates, the predicate m in a rate-specified rule also might match with multiple flows at the same time. We denote the collection of involved flows as $F(m)$ and regard them as a "virtual" aggregated flow. For this "virtual" flow, we further use r_m^* to present what its rate-limit would be during the update process. Then the two types of throughput requirements could be formulated as Equation (8) and (9) show, in which r_m^* is defined by Equation (10) and $amap$ is the index/variable that should be optimized.

$$\forall (m_i, m_j) \in MP_R : r_{m_i}^* \geq r_{m_j}^* \tag{8}$$

$$\forall m_i \in M_R^{amap} : r_{m_i}^* \geq amap \tag{9}$$

$$r_m^* = \frac{\sum_{\forall f \in F(m)} r_f \cdot t_f}{\sum_{\forall f \in F(m)} t_f} \tag{10}$$

So far, CUP has translated all user-specified requirements into low-level flow-based constraints, which are all linear.

III. Efficient Solver

To handle various updates, CUP needs a generic yet efficient solver. However, the design is not easy since planning updates is computationally intractable in ordinary sense—even answering the question of whether there exists a congestion-free solution for a given update is NP-hard as Theorem 1 says.

Theorem 1. *Determining whether there is a congestion-free update order scheduling that meets user-specified deadline is NP-Hard in ordinary sense.*

Proof. Refer to Appendix B in [17] for details. □

Corresponding to the fact that planning an update involves two parts of 1) finding an execution order and 2) computing the relevant rate-limiting scheme, CUP heuristically decouples the original problem into two parts as Fig. 2 shows. On planning a group of flow migrations, the *Order Scheduler* module first determines which round each flow should be moved in, based on user-specified time-related requirements. If there exists congestion-free sequences, *Order Scheduler* outputs the one with the minimum rounds; otherwise, it suggests the sequence causing smallest traffic overloads. Then, for the congested traffic, *Rate Manager* further finds the optimal rate-limiting scheme that makes the update free of congestion, respecting to throughput/rate-related rules.

A. Order Scheduler

The first step of planning update to prevent transmit congestions is to evaluate what link loads would be during the update procedure. For flow $f \in F^U$, we let $t_{f,e,k}$ indicate its maximum possible load on link e when preforming the reconfiguration of round k. Then, the maximum (possible) load on link e in this round is $\sum_{f \in F^B} t_{f,e} + \sum_{f \in F^U} t_{f,e,k}$.

$$t_{f,e,k} = \begin{cases} t_{f,e} - y_{f,k-1} \cdot (\max(t_{f,e}, t'_{f,e}) - t'_{f,e}) \\ \quad + y_{f,k} \cdot (\max(t_{f,e}, t'_{f,e}) - t_{f,e}) & Changed\ ind. \\ t_{f,e} - y_{f,k-1} \cdot t_{f,e} + y_{f,k} \cdot t'_{f,e} & Otherwise \end{cases} \tag{11}$$

The calculation of $t_{f,e,k}$ for round k has two formulations depending on f's update senses as Equation (11) shows. In both formulations, it is certainly that f's load on link e equals $t_{f,e}$ if f has not been migrated yet, i.e., $y_{f,k-1} = y_{f,k} = 0$, or equals $t'_{f,e}$ if its migration has completed, i.e., $y_{f,k-1} = y_{f,k} = 1$. The difference exists in the case when f happens

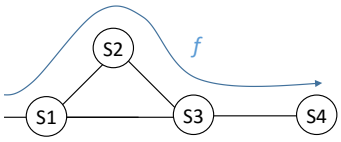


Fig. 4: An example of that the updated flow is not *changed independently*: move f from path S1-S2-S3-S4 to S1-S3-S4.

to be migrated in round k, i.e., $y_{f,k-1} = 0$ and $y_{f,k} = 1$, and the link is used by both f's old path(s) and new path(s).

In datacenter networks, the multiple paths between two end-hosts usually share the same hops and packets traveling through them are likely to experience the similar delay [2]. Accordingly, the load of f on link e during the update is either $t_{f,e}$ or $t'_{f,e}$. In this condition, f is *changed independently* [2] on link e, and its maximum possible load during the update is $\max(t_{f,e}, t'_{f,e})$, corresponding the upper case of Equation (11). However, the situation of WAN is quite different, in which multiple paths of a source-destination pair generally have distinct delays. In the worst case, the load of flow f on e would reach $t_{f,e} + t'_{f,e}$. As an example, consider the case of rerouting flow f from path S1-S2-S3-S4 to S1-S3-S4 shown in Fig. 4. On switching f to its new path, because of the transmission and buffer delays, incoming packets traveling through S1-S3, together with the in-flight packets on sub-path S1-S2-S3, would contribute a total load of $t_{f,e} + t'_{f,e}$ on link S3-S4. Fortunately, by comparing the new network configuration with the old one, CUP knows whether a flow is *changed independently* or not. Then, the right expression of $t_{f,e,k}$ for flows and links can be decided.

On computing the update order, CUP tries to minimize the overloaded traffic on links while optimizing the total required rounds. Provided o_e is the amount of overloaded traffic on link e (whose capacity is c_e), there are many alternative formulations that capture the link load situation of the entire network—E.g., $\sum_{\forall e} o_e$, $\max_{\forall e} o_e$, $\sum_{\forall e} \frac{o_e}{c_e}$, and $\max_{\forall e} \frac{o_e}{c_e}$. CUP adopts $\max_{\forall e} o_e$. With this design, even if the network is failed to apply the rate-limiting schemes, the scheduled update order will still let the transient congestions be distributed on most links, so that the overloaded packets are more likely to be held by switch buffers.

$$\forall e, k > 0 : \sum_{f \in F^B} t_{f,e} + \sum_{f \in F^U} t_{f,e,k} \leq c_e + o_e; o_e \geq 0 \quad (12)$$

Obviously, this order scheduling problem is naturally to be formulated as a MIP (Mixed Integer linear Program) as Fig. 5 shows, where γ is a small factor ($0 \leq \gamma \ll 1$) and the tail of $-\gamma \cdot \sum_{\forall (f,k)} y_{f,k}$ is to let flows be migrated as soon as possible. For the schedule of a small scale update, we can obtain the optimal order by directly solving this MIP with efficient solvers. However, as finding the optimization scheduling order is theoretically NP-hard, the computation process becomes quite time-consuming when the network scales up. To find scheduling orders quickly, we relax the original MIP into a Linear Program (LP), and develop an efficient heuristic solution based on the relaxed LP's outputs. Due to the lack of space, the detail of heuristic algorithm follows in our technical report (Appendix C) [17].

$$\begin{cases} \textbf{Input:} & F^B, F^U, FP_T, \{c_e\}, \{t_{f,e}\}, \{t'_{f,e}\}, \{N_f^{due}\} \\ \textbf{Output:} & \{y_{f,k} | \forall (f \in F^U, k)\} \\ \text{Minimize} & \max_{\forall e} o_e - \gamma \cdot \sum_{\forall (f,k)} y_{f,k} \\ \text{Subject to} & (2), (3), (7), (11), \text{and } (12), \end{cases}$$

Fig. 5: Schedule update orders to minimize the link overloads. γ is a small constant: $0 < \gamma \ll 1$.

$$\begin{cases} \textbf{Input:} & F^B, F^U, \{c_e\}, \{t_{f,e}\}, \{t_{f,e,k}\}, MP_R, M_R^{amap} \\ \textbf{Output:} & \{r_f | \forall f\} \\ \text{Maximize} & amap + \varrho \times \min_{\forall f} r_f \\ \text{Subject to} & (4), (8), (9), (10), \text{and } (13). \end{cases}$$

Fig. 6: Manage transient congestions in each update round $\{r_f | \forall f\}$ explicitly following user's requirement. ϱ is a small constant: $0 < \varrho \ll 1$.

In practice, a simple way to achieve both efficiency and effectiveness on order scheduling is to employ a "dual-core" trick. For each planning request, CUP can perform the MIP solving and heuristic computation, simultaneously. If MIP completes within a certain time (e.g., one second), CUP gets the optimal results; otherwise, CUP chooses the heuristic result and stops the task of MIP solving.

B. Rate Manager

Once the update order is determined, CUP gets the value of $\{t_{f,e,k} | \forall (f, e, k)\}$. The next issue is to find a rate-limiting scheme avoiding congestion respecting to user's requirements. As defined in Section II-B2, r_f is the ratio that flow f should decrease to for removing transient congestions; then, the straightforward solution to obtain the optimal rate-limiting scheme for user-specified requirements is to solve the corresponding LP shown in Fig. 6.

$$\forall e, k > 0 : \sum_{f \in F^B} r_f \cdot t_{f,e} + \sum_{f \in F^U} r_f \cdot t_{f,e,k} \leq c_e \quad (13)$$

Note that, when no $amap$-based rule is specified, CUP adopts $R(*) \geq amap$ by default, which results in minimizing the total throughput loss. In some cases, there might be multiple rate-setting schemes that obtain the same optimal $amap$. CUP adds a tail of $\varrho \times \min_{\forall f} r_f$ (ϱ is a small positive constant) into the objective to gain the one limiting less flows.

About efficiency. So far, we have built a generic solver made up of *Order Scheduler* and *Rate Manager* for CUP. Obviously, the core computation in both *Order Scheduler* and *Rate Manager* is solving LPs, which can be efficiently done within polynomial time by leveraging fast solvers like CPLEX and MOSEK. Consequently, the entire solver is a polynomial time approach as well. Furthermore, there are several simple yet efficacious tricks that CUP can employ to simplify the model and accelerate the computation. For example, if a link would never be overloaded during the update, CUP can exclude its related constraints from the model safely. We call such links *non-critical*, and they can be determined by Equation (14) easily. Corresponding, if a flow only encounters with *non-critical* links, there is no need to limit its rate. CUP can remove

its constraints from the rate-manage model. As well, if a to-be-updated flow is *non-critical* and does not have "no-later-than" relation with other flows, it can be migrated directly in the first round without planning computations.

$$E_{non\text{-}crit.} = \{\forall e \mid \sum_{\forall f \in F^B} t_{f,e} + \sum_{\forall f \in F^U} \max_{\forall k} t_{f,e,k} \leq c_e\} \quad (14)$$

Multi-tenant. In practice, a network might be shared by multiple tenants (or virtual operators) simultaneously [21]. The requirements specified by a tenant should only impact its own updates and own traffic. In such cases, CUP would look into the tenant information when embedding policies. As for CUP's solver, *Order Scheduler* is able to handle this directly because there is no difference on the sub-problem of order scheduling; however, *Rate Manager* needs a modification as the rate management problem is a *multi-objective optimization problem* now—$\max\ (amap_1, amap_2, \ldots, amap_n)$. Multiple-objective optimization has been studied for very long time and there are so many solutions, such as *scalarization*, *no-preference methods*, *priori methods*, *etc* [22]. In this paper, CUP simply adopts the approach of linearly *scalarizing* [22] the multiple objectives into the single objective of $\max \sum_{\forall i} w_i \cdot amap_i$, where $w_i \geq 0$ stands for the weight of the ith tenant. By simply pursuing this scalarizated objective, CUP supports multi-tenant updates. We note that there is room to improve and CUP is flexible to be upgraded.

Concurrent updates. In general, a "fat" update request involving many flow migrations would be planned to execute in more than one round. As the network configuration is volatile, new update request is likely to occur before the current "fat" one completes. This should be handled appropriately and immediately as some new flow migration requests might have urgent deadline requirements. CUP adopts the generic two-phase mechanism [5] to implement the reconfiguration of each round, which naturally supports update streams. Accordingly, CUP can immediately deal with a new request by just regarding it together with these unperformed rounds as a fresh request; rule consistency is always guaranteed.

IV. Evaluation

In this section, we implement CUP based on Ryu, and conduct virtual networks with Mininet to test CUP. Our results indicate that CUP is flexible enough to handle user-specified time- and throughput- requirements. Moreover, CUP is very effective. On each type of requirement, CUP always significantly outperforms the variant of Dionysus which is modified to handle that requirement type.

A. Implementation

We prototype CUP upon Ryu 3.26, and employ it to plan traffic migrations for toy virtual networks on Mininet 2.2 [13].

Network setup. When switches start up, the controller installs default routes and tunnel rules via OpenFlow 1.3. We let end-hosts send UDP packets with each other in steady rates to simulate the case of backbone traffic in WAN, and use VLAN tags to implement tunnel-based forwarding for them. We assume that the network adopts multi-path routing, in which ingress switches split and assign a flow to its sub-tunnels respecting to tunnel weights. Then, updating a flow is only to reconfigure its tunnel weights at the ingress, so that each update is consistent in essence [5, 6].

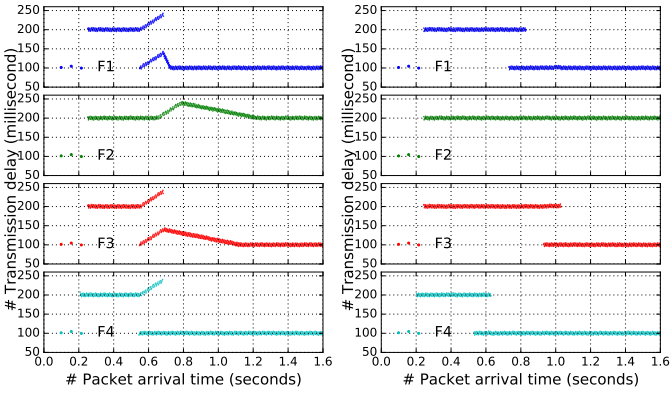
To carry out weighted traffic splitting on Open vSwitch, the controller installs a group of exact-match rules specifying the tunnel for each microflow.[2] Unfortunately, this approach makes rule management on ingress complex as the update of a single flow might trigger the modification of a collection of microflow rules. We address the problem by using the Multiple Flow Table mechanism provided by OpenFlow switches (supports start from OpenFlow 1.1). Basically, rules in an ingress switch are either stored in Table 0 or Table 1 depending on their types. In normal, forwarding functional rules like tunnels and default routes reside in Table 1, and these microflow rules that realize traffic splits and tunnel selections, together with a lower priority all-$*$ whose action is "goto Table 1", reside in Table 0. When a flow's splitting weights are to be updated, the controller first installs microflow rules that implement the new weights in Table 1, then installs a high-priority wildcard rule with action "goto Table 1" into the first table to "guide" involved packets to the new weights. After that, the controller silently modifies the actions of those unmatched microflow rules in Table 0 following the new weights, then deletes the previously installed wildcard rule and microflow rules. Following this, we make rules easy to manage and guarantee the consistency property during weight reconfigurations.

Benchmark schemes. We implement CUP's algorithm in Python and employ Mosek as the backend solver for LPs. As a benchmark, we implement the schedule algorithm of Dionysus. Although it is designed for dynamic scheduling of updates, under the situation that new rules are pre-installed and ingress switches share the similar time cost on enabling new configurations for flow, Dionysus would also derive a round schedule together with a rate limiting scheme for each update in advance [3]. If the obtained round number is larger than the deadline requirement, we assume that Dionysus adopts its deadlock-break mechanism for help—limit the rates of flows whose scheduled time would miss the deadline to zeros, and perform all their migrations in the last round.

B. Case study

To evaluate how transient congestion caused by unplanned updates would influence the traffic, we first conduct experiments for the toy update cases shown in Fig. 1. Note that all virtual hosts and switches in Mininet use the shared CPU and bandwidth resources for simulation [13]. To avoid resource competition between them and to highlight the results, we set link bandwidth to 5 Mbps with 100 ms delay, and let port buffer size be large enough to hold all overloaded traffic. Accordingly, in the case of no congestion, the transmission delay of all old paths is about 200 ms, same to the network's maximum OWD, and that of the new paths is about 100 ms.

[2] In tests, the traffic from a host to another is equally dispersed over 20 UDP flows, and its ingress switch holds a corresponding number of microflow rules for traffic splitting. Thus, the accuracy of traffic-splitting is 0.05.



(a) Unplanned One-Shot update (b) Planed update $(R(*) \geq amap)$

Fig. 7: Transient congestion during unplanned updates.

Fig. 7a shows the transmission delay of packets in each flow when the controller sends the "activate the new path" commands for {F1, F3, F4} in *One Shot* at the 0.4 s. About 150 ms later, receivers get packets through the new paths. Obviously, the latency of packets in all flows increase during the update process. That is to say, they all entered queues because of transient congestion. In the test, we set no artificial control delay between the controller and switches (however, there is still a delay about 50 ms for each flow table modification from CUP sending the command via REST API) so that all flows enjoy their new paths almost at the same time. As a result, the newly incoming packets of F1 together with the in-flight packets of F3 and F4 overload Link S1-S3, while F1's in-flight packets together with the newly incoming packets of F2 and F3 overload Link S4-S3. In practice, the activation time of new rule might be distinct on switches; transient congestion happens once a flow moves in the hot link before the old in-flight packets exists. And these overloaded packets in high speed network can be really huge, which would quickly eat up switch buffers and result in heavy packet loss [3].

As a comparison, Fig. 7b shows the case of migrating flows in order of [F4→F1→F3], which is the result planned by both Dionysus and CUP under the policy of $(R(*) \geq amap)$. In this case, the controller triggers flow migrations round by round, and waits the maximum OWD time (200 ms) between them. Following the plan, the update process takes about 600 ms to complete, but avoids all transient congestion.

Then, we look into the case of planning updates with time- and throughput- requirements. Provided the update request appear at the 0.4 s, and the operator wants all flows to enjoy their new paths no later than 300 ms; that is to say, all flow migrations must be carried out within one round,[3] and rate-limits are needed to avoid congestion. Fig. 8 and Fig. 9 show the results planned by CUP under user-specified policies $(T(*) \leq 1; R(*) \geq amap)$ and $(T(*) \leq 1; R(m_{F2} \vee m_{F4}) \geq amap)$, respectively. In the case of Fig. 8, all flows share the same importance and the operator prefers the total throughput be reduced as less as possible. With the objective function shown in Fig. 6, CUP's *Rate Manager* lets the throughput loss be shared by all flows in proportion as Fig. 8b shows, where $\frac{\Delta y}{\Delta x}$ stands for the flow rates observed by the sender or receiver—about $\{\frac{5}{14}, \frac{4}{14}, \frac{5}{14}, \frac{4}{14}\}$. Different from Fig. 8, Fig. 9 demonstrates the case that F1 and F4 are background traffic while F2 and F4 are interactive whose throughput should be keep as much as possible. As the results show, CUP finds the update plan exactly following the operator's wish. In contrast, Dionysus will handle the requirements in a rough way—completely kill F1 and F2 to avoid congestion.

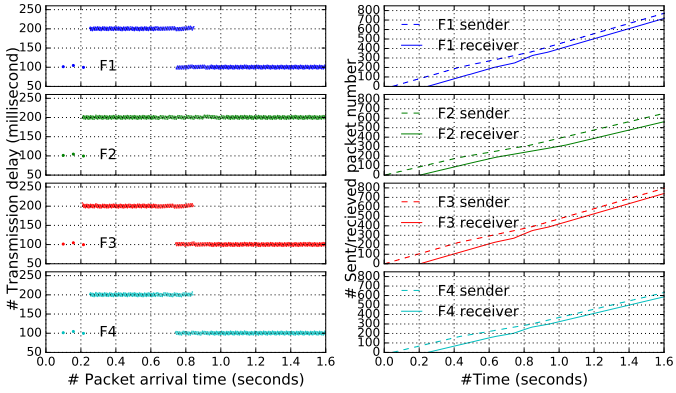
[3]It takes about 200 ms to pre-install new rules and wait rate-limits coming into force; then less than 100 ms is left for performing the updates.

C. CUP flexibility

To investigate the flexibility of CUP, we further employ it to plan updates for a small WAN [3, 9], which involves 8 nodes and 14 links as Fig. 10 illustrates. In this case, each link is assumed to have the capacity of 10 Mbps and delay of 200 ms. We consider the case of WAN optimization, where ingress switches split the traffic to a destination among its 4-shortest paths to pursue load balancing. Because of lacking real traffic matrices, we assume that all the possible paths of a source-destination share the equal weight initially, and use *gravity model* [4] to synthesize the current traffic demands, which make the maximum link load be 99% in the old configuration. Then, the update scenario is to reconfigure traffic split weights to the new one that reduces the maximum link load to the minimized value, 78%. The longest path(s) in tests involves 4 links; accordingly, the network's maximum OWD is 800 ms. For each link e, we consider it as *unchanged independently* for flow f, if f has more than one path going through e and these paths hold distinct lengths (i.e., delays).

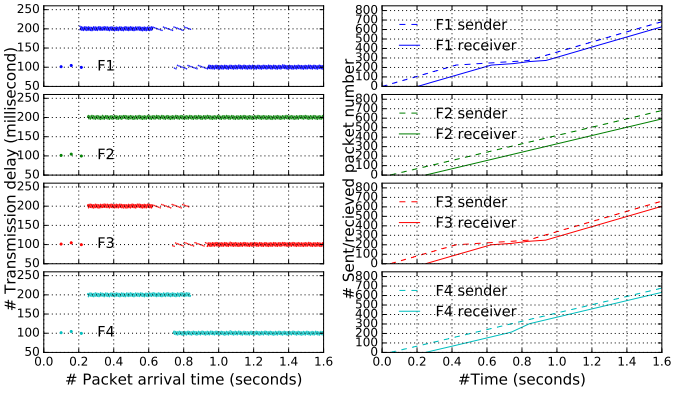
When no update deadline is required, CUP finds a congestion-free plan involving 5 rounds without limiting flow rates, while Dionysus obtains a 6-round plan that achieves the same goal. Then, we artificially add deadline requirements to all flows and compute the propitiation of network throughput that CUP, as well as Dionysus, has to abandon for congestion freedom. Numerical results indicate that CUP outperforms Dionysus about 3× on reducing the impact of network throughput as Fig. 11 shows. CUP is excellent because its *Rate Manager* always obtains the optimal rate-limiting scheme respecting to user's requirements. On the contrary, Dionysus just randomly kills some flows to move on. In addition, Dionysus would never touch the rate of the un-updated flows. But in some cases, slowing down some of them really helps.

We also study the cases that some traffic is background and the operator wishes interactive traffic be less impacted during the update. To this end, we assume that a certain percentage of traffic between each source-destination pair is background, then calculate how many round CUP, as well as Dionysus, would need to perform congestion-free reconfiguration without reducing the throughput of interactive traffic. Fig. 12 demonstrates the results. It implies that, with the proportion of background traffic increasing, the round number required by CUP rapidly decreases. And after the background traffic accounts for half of the traffic, CUP always performs congestion-free updates in one round without reducing the rates of interactive flows. In contrast, Dionysus can not achieve this because of its unawareness of user-specified requirements. If we pre-limit the rates of background traffic to zeros, Dionysus then obtains



(a) Impact on transmission delay (b) Impact on flow rate

Fig. 8: Plan under policy: $(T(*) \leq 1; R(*) \geq amap)$, i.e., all migrations should be finished within 1 round and the total network throughput should be as-much-as-possible.



(a) Impact on transmission delay (b) Impact on flow rate

Fig. 9: Plan under $(T(*) \leq 1; R(m_{F2} \vee m_{F4}) \geq amap)$, i.e., all migrations should be finished within 1 round and the total throughput of F2 and F4 should be as-much-as-possible.

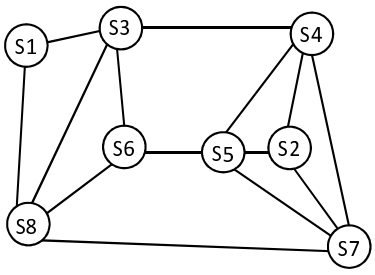


Fig. 10: WAN topology in [3].

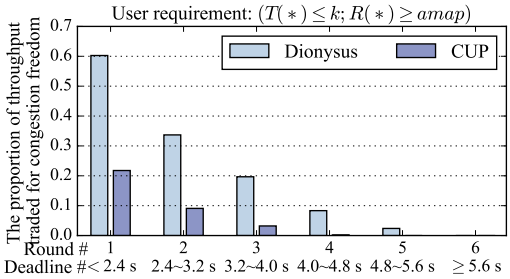


Fig. 11: Throughput loss V.S. update speed.

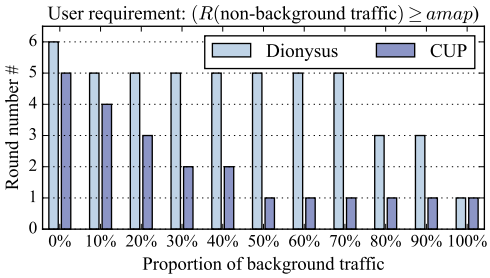


Fig. 12: Impact of background traffic.

small update rounds as CUP does. However, similar to the cases shown in Fig. 11, such a solution is far from good because too many flows are killed unnecessarily.

V. Related Work

As in-flight packets might be handled by a mix of different versions of rules during the update, several approaches are proposed to provide strong consistent properties such that no packet or flow misuse rules [5, 6, 23], or weaker yet specific properties such as loop freedom [12, 24, 25], and waypoint invariant [12, 26]. While orthogonal to them, CUP focuses on another problem of managing transient congestion during the reconfiguration process, and directly employs *generic two-phase* approach [5] to guarantee strong rule consistency for each step/round.

Schedulers like zUpdate [2], SWAN [8], and GI [9] attempt to avoid transient congestion by introducing a sequence of intermediate traffic distributions (i.e., configurations), following which, the transition might be congestion-free. These introduced intermediate configurations greatly complicate the update procedure, and make the network error-prone [27]. Even worse, these intermediate configurations might hurt user's QoS because of their paths might have unsatisfied delays and jitters. Moreover, for some updates, there does exist congestion-free transition plans. To avoid this, a portion (10%-50% [8]) has to be left vacant, which leads a great waste of link capacities (GI [9] chooses to bear the transient congestion instead of reserving vacant bandwidth). Differently, Dionysus [3] and ATOMIP [5] try to handle transient congestion by scheduling update operations according to a dynamic-determined or pre-designed order, which might avoid the problem of intermediate configurations. Even though, they only maintain a pre-defined specific objective by design—either towards fast speed, or congestion freedom. Accordingly, they can not deal with various update scenario properly. By comparison, CUP formulates the update planning problem with generic models. Via binding models with user-specific constrains and objective functions, CUP adapts to a large fraction of scenarios easily.

VI. Conclusion

As transient congestions are prone to occur during SDN updates, controllers are in urgent need of a planner to handle the trouble. We argue that planning the reconfiguration process respecting to specified requirements is an import issue. In this paper, we have analyzed the desired properties of such planners and proposed a case design–CUP. CUP translates high-level user-specific requirements into linear constraints and formulates the planning problem as generic linear programs. By solving customized LPs, CUP is flexible to obtain "best" plans for a large fraction of updates.

Acknowledgements. This work was supported in part by the 973 Program (2013CB329103), 863 Program (2015AA015702, 2015AA016102), NSFC (61271171, 61271165, 61571098), Ministry of Education - China Mobile Research Fund (MCM20130131), China Postdoctoral Science Foundation (2015M570778), Fundamental Research Funds for the Central Universities (2682015CX072), Science and Technology Program of Sichuan Province (2016GZ0138).

REFERENCES

[1] S. Raza, Y. Zhu, and C.-N. Chuah, "Graceful Network State Migrations," *IEEE/ACM Trans. Netw*, vol. 19, no. 4, pp. 1097–1110, Aug 2011.

[2] H. H. Liu *et al.*, "zUpdate: Updating data center networks with zero loss," in *SIGCOMM*, Aug 2013, pp. 411–422.

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

[4] L. Luo, H. Yu, S. Luo, and M. Zhang, "Fast lossless traffic migration for SDN updates," in *IEEE ICC*, June 2015, pp. 5803–5808.

[5] 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, March 2015.

[6] M. Reitblatt *el al.*, "Abstractions for network update," in *SIGCOMM*, Aug 2012, pp. 323–334.

[7] T. Mizrahi, E. Saat, and Y. Moses, "Timed consistent network updates," in *Proc. ACM SOSR*, 2015, pp. 21:1–21:14.

[8] C.-Y. Hong *el al.*, "Achieving high utilization with software-driven WAN," in *SIGCOMM*, Aug 2013, pp. 15–26.

[9] J. Zheng, H. Xu, G. Chen, and H. Dai, "Minimizing transient congestion during network update in data centers," in *Proc. 23rd ICNP*, Nov 2015.

[10] H. H. Liu *el al.*, "Traffic engineering with forward fault correction," in *SIGCOMM*, Aug 2014, pp. 527–538.

[11] V. T. Lam *el al.*, "Netshare and stochastic netshare: Predictable bandwidth allocation for data centers," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 3, pp. 5–11, Jun. 2012.

[12] W. Zhou *el al.*, "Enforcing customizable consistency properties in software-defined networks," in *NSDI*, May 2015, pp. 73–85.

[13] N. Handigol *el al.*, "Reproducible network experiments using container-based emulation," in *CoNEXT*, 2012, pp. 253–264.

[14] S. Jain *el al.*, "B4: Experience with a globally-deployed software defined wan," in *SIGCOMM*, Aug 2013, pp. 3–14.

[15] O. Gurewitz, I. Cidon, and M. Sidi, "One-way delay estimation using network-wide measurements," *IEEE Trans. Inf. Theory*, vol. 52, no. 6, pp. 2710–2724, June 2006.

[16] A. Pathak *el al.*, "A measurement study of internet delay asymmetry," in *Proc. 9th PAM*, 2008, pp. 182–191.

[17] S. Luo *et al.*, "Arrange your network updates as you wish," http://shouxi.name/publication/cup-tr.pdf, Tech. Rep., Dec 2015.

[18] J. H. Han *el al.*, "Blueswitch: enabling provably consistent configuration of network switches," in *Proc. ACM/IEEE ANCS*, May 2015, pp. 17–27.

[19] M. Kuzniar *et al.*, "What you need to know about SDN control and data planes," Tech. Rep., 2014, EPFL-REPORT-199497.

[20] R. Bifulco and A. Matsiuk, "Towards scalable SDN switches: Enabling faster flow table entries installation," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, pp. 343–344, Aug. 2015.

[21] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?" in *OSDI*, 2010, pp. 1–14.

[22] Wikipedia, "Multi-objective optimization," https://en.wikipedia.org/wiki/Multi-objective_optimization, 2015, [Online; accessed 23-Nov-2015].

[23] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *Proc. 2nd ACM HotSDN*, 2013, pp. 49–54.

[24] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *Proc. ACM HotNets*, 2013, pp. 20:1–20:7.

[25] A. Ludwig *et al.*, "Scheduling loop-free network updates: It's good to relax!" in *Proc. ACM PODC*, 2015, pp. 13–22.

[26] A. Ludwig *el al.*, "Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies," in *Proc. ACM HotNets*, 2014, pp. 15:1–15:7.

[27] J. Miserez *et al.*, "Sdnracer: Detecting concurrency violations in software-defined networks," in *Proc. ACM SOSR*, 2015, pp. 22:1–22:7.

[28] S. Luo, H. Yu, and L. Li, "Fast incremental flow table aggregation in sdn," in *Proc. 23rd ICCCN*, Aug 2014, pp. 1–8.

[29] S. Luo, H. Yu, and L. Li, "Practical flow table aggregation in SDN," *Computer Networks*, vol. 92, Part 1, pp. 72 – 88, 2015.

[30] R. Tarjan, "Depth-first search and linear graph algorithms," in *12th Annual Symposium on Switching and Automata Theory*, Oct 1971, pp. 114–121.

[31] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962.

APPENDIX A
WHY CUP ADOPTS TWO-PHASE FOR RULE CONSISTENCY

As is known, when reconfiguring the network, in-flight packets might misuse different versions of rules [5, 6, 12, 24, 26] and the solution is to either perform rule changes following a well-designed order [12, 24, 26], or use version tags to avoid the mix use [5, 6]. Order arrangement does not require extra rule spaces, however, it has two fatal flaws. First, it is not universal and only works in specified cases [24, 26]. Second and crucially, it is time-costly since it has to change rules one-by-one; this results in big update durations [5]. For example, provided the longest path in the *dependency tree* of rule changing order is L and the average time for changing a rule from the controller is τ, it would take about $L \times \tau$ to go through the entire process. In contrast, the version-based two-phase mechanism is generic and fast. If new rules already exists, the controller only needs to modify the ingress to switch a flow to the new path(s)—the time cost is τ; even though new rules are absent, the controller can let all new rules ready within another τ because these rule installations can be executed concurrently—the total time cost is 2τ, still greatly smaller than $L \times \tau$.

The possible price of version-based methods is rule-space overheads—switches have to hold two version of rules temporarily. For this problem, recent study has shown that, with the help of *wildcard* in match fields, switches only needs to store two versions for rules that are being modified [5]; this greatly reduce the overheads. Moreover, after an update procedure completes, all old rules can be removed immediately. Thus, we argue that rule-space overheads are not serious in many cases. Even in the case that the rule-space is the bottleneck, the controller can still pre-split updates to reduce the demands of extra rule-space [23], or employs rule aggregation techniques to get more available rule spaces [28, 29].

APPENDIX B
THE PROOF OF THEOREM 1

Proof. The proof is quite similar to that of Theorem 2 in [3]. Consider a network in which a set of integer traffic demands travel through via link e_1 or link e_2, alternatively, and the capacity of both links is c. Initially, flows in group G_A go through e_1, while flows in group G_B go through e_2. Their total load are c_A and c_B, respectively, where $c_A \leq \frac{c}{2}$ and $c_B = c$. Suppose the update is to swap their routes. Obviously, the fastest updating plan that might be congestion-free is a 3-round solution: 1) migrate a part of G_B with the total load of $c - c_A$ from e_2 to e_1; 2) migrate G_A from e_1 to e_2; and finally 3) migrate the reset of G_B from e_2 to e_1 (with load $c_B - c_A$). However, to figure out whether this 3-round congestion-free solution exists, we have to solve the *subset sum problem* of finding a subset flows from G_B sum to $c - c_A$, which is known as NP-complete. □

APPENDIX C
EFFICIENT HEURISTIC ALGORITHM FOR ORDER SCHEDULER

1 Get $\{y_{f,k}|\forall(f \in F^U, k)\}$ by solving the following LP

$$\begin{cases} \textbf{Input:} & F^B, F^U, FP_T, \{c_e\}, \{t_{f,e}\}, \{t'_{f,e}\}, \{N_f^{due}\} \\ \textbf{Output:} & \{y_{f,k}|\forall(f \in F^U, k)\} \\ \text{Minimize} & \max_{\forall e} o_e - \gamma \cdot \sum_{\forall(f,k)} y_{f,k} \\ \text{Subject to} & (3), (7), (11), (12), \text{and} \\ & \forall k, f \in F^U : 0 \leq y_{f,k} \leq 1 \end{cases}$$

2 $X_i \leftarrow \arg\max_k (y_{f_i,k} - y_{f_i,k-1})$ for each flow f_i in F^U
3 $D \leftarrow$Build a edgeless graph with $node_i$ containing $f_i \in F^U$
4 **foreach** $\langle f_i, f_j \rangle \in MP_T$ **do**
5 Add directed edge/arrow $(node_j, node_i)$ into D
6 $D \leftarrow$Find and aggregate each SCC in D into a virtual node
7 $S \leftarrow$ Get the set of nodes with no incoming arrows in D
8 **while** $S \neq \emptyset$ **do**
9 $v \leftarrow$ POP(S)
10 $k \leftarrow \min_{\forall f_j \in v} X_j$
11 **foreach** $f_j \in v$ **do**
12 $X_j \leftarrow k$ `// order req.`
13 **foreach** *arrow* $(v, u) \in D$ **do**
14 **foreach** $f_i \in u$ **do**
15 $X_i \leftarrow \min(X_i, k)$ `// order req.`
16 Remove arrow (v, u) from D
17 **if** *node* u *has no incoming arrows* **then**
18 Add u into S
19 Sort values in set $\{X_i|\forall f_i \in F^U\}$ in ascending order, and denote the index of value X_i as $\pi(X_i)$, which is the round number that f_i it should be updated in

Fig. 13: Determines the update order schedule with a LP-based heuristics, LPHA; γ is a small constant: $0 < \gamma \ll 1$.

Given an update, the heuristic algorithm (refer as LPHA hereafter) processes as follows. It first solves the corresponding relaxed LP to get the relaxed value of $\{y_{f,k}|\forall(f,k)\}$ (Line 1), then greedily selects a round number for each flow based on the results (Line 2). Because of the relaxation, the obtained order might violate the relative time requirements (e.g., Equation 3). To remedy this, LPHA builds a directed graph D to capture the "no-later-than" requirements (Line 3-5) and use it to fix (Line 6-19).

Note that, these time requirements formulate a non-strict partial order for the to-be-updated flows. That is to say the time requirement on updating order is *reflexive*, *antisymmetric*, and *transitive*. For example, the *transitivity* implies that, if f_i should be updated no later than f_j while f_j should be updated no later than f_k, the implicit requirement is that f_i should be migrated no later than f_k; the *antisymmetry* indicates, if one of $\{f_i, f_j\}$ should be updated no later than the other, it means they must be updated in the same round. Therefore, there might be loops in the directed graph D, in which flows belonging to the same cycle are required to be updated in the same round. To handle this, LPHA finds out all the Strongly Connected Component (SCC) in the D with Tarjan's algorithm [30], whose time complexity is linear with the total number of nodes (V) and edges (E) in the graph– $O(|V| + |E|)$. As each SCC is a set of intertwined cycles, LPHA aggregates each into a virtual node, so that D shrinks into a virtual Directed Acyclic Graph (DAG) (Line 6).

The procedure of fixing the "no-later-than" relationship on DAG is similar to that of the topological sorting algorithm described by Kahn [31]. At each turn, LPHA 1) picks a node v with no incoming arrows (i.e., no-later-than requirements) from the DAG (Line 9), 2) computes a round number that satisfies the requirements of all flows belonging that node 10, and 3) sets the round number of these flows (Line 12). After the rounds of flows in this node are established, LPHA immediately updates the maximum possible round number for the un-scheduled flows having "no-later-than" requirements on this node (Line 15). This guarantees all "no-later-than" requirements always being kept.

Finally, LPHA makes a rearrangement to remove the unused round numbers, and get the final schedule—$\pi(X_i)$ (Line 19).