IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 21, 2018, accepted January 1, 2019, date of publication January 11, 2019, date of current version April 5, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2892557

Simplifying Flow Updates in Software-Defined Networks Using Atoman

LONG LUO[1], ZONGHANG LI[1], JINGYU WANG[2], AND HONGFANG YU[1]

[1]Key Lab of Optical Fiber Sensing and Communications (Ministry of Education) and Center for Cyber Security, University of Electronic Science and Technology of China, Chengdu 611731, China

[2]State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China

Corresponding author: Hongfang Yu (yuhf@uestc.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61571098 and Grant 61871097, in part by the Fundamental Research Funds for the Central Universities under Grant ZYGX2016J217, in part by the 111 Project under Grant B14039, in part by the Open Foundation of Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, under Grant SKLNST-2016-2-24, and in part by the scholarship from the China Scholarship Council under Grant 201806070026.

ABSTRACT Flow updates are common in today's networks, and software-defined networking (SDN) enables network operators to reconfigure switches for updating flows easily. However, the implementation of flow updates requires to meet many different expectations regarding consistency, resource constraints, and performance. To carry updates out as intended, network operators often need to spend significant effort in update management, developing complex network optimizations and customized heuristics on a case-by-case basis. In this paper, we strive to simplify the flow updates in SDN networks. To this end, we present Atoman, a framework that uses high-level abstractions to capture various update intents and formulates flow updates problems as segment-based update scheduling optimizations to obtain satisfied update solutions. The captured update intents are translated into constraints and objectives of update scheduling optimizations. By extracting critical updating flows and employing decomposition techniques, Atoman can efficiently reduce the scale of problems in each solving and generate near-optimal update solutions. We conduct extensive simulations to evaluate Atoman and the simulation results show that Atoman significantly saves operator efforts in managing flow updates and provides comparable or better efficiency than prior customized solutions.

INDEX TERMS Software-defined Networks, flow updates, intents.

I. INTRODUCTION

Flow updates frequently happen to cope with network dynamics such as traffic fluctuations and failures [1]–[6]. For example, when traffic surges, to avoid the congestion and balance the load, network operators adjust flow routes and migrate traffic flows from the overloaded paths to the underutilized ones; while when a link fails, to prevent significant packet losses, network operators need to immediately redirect the affected flows to the paths that bypass this failure. Software-Defined Networking (SDN) provides a flexible and simple way to manage a network by integrating control capacities to a logically centralized controller [7], [8]. In SDN networks, operators perform flow updates by only telling the controller the updating flows and what new rules should be configured to which network devices (*e.g.*, switches).

Although network management is greatly simplified, there still requires significant work to coordinate updates across network devices and flows as expected. As we know, switches update rules asynchronously without coordination [9], [10], which may lead to a mixed-use of rules and further cause the network to behave inconsistently with either the current or the new rules [2]. Thus, to update flows consistently, additional efforts are required to manage updates. Also, different flows may require different levels of consistency, which makes it more complicated. For example, the packets of a background flow often need to be delivered correctly to the destination [11], while those generated by a service flow may be expected to visit waypoints (*e.g.*, functional nodes) before reaching the destination [11], [12]. In addition to consistency, there are many other expectations (refer to [13] for a summary) that an update implementation needs to satisfy. For example, the implementation of an update driven by Traffic Engineering (TE) is often expected not to violate resource

The associate editor coordinating the review of this manuscript and approving it for publication was Yupeng Wang.

2169-3536 © 2019 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

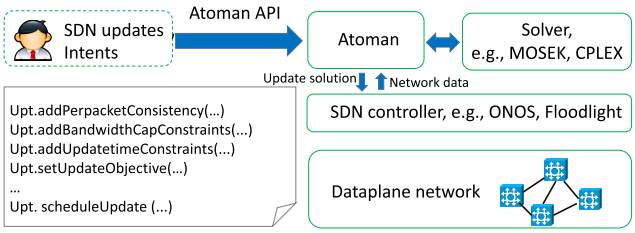


FIGURE 1. Operators use the Atoman high-level APIs to express updates and associated intents on the implementation. Atoman generates update solutions that are inputs of the SDN control platform.

capacities and avoid transient congestion during the transition. While for updates triggered by link or node failures, the updates of impacted flows are often expected to be completed as soon as possible to recover from failures quickly. To satisfy these various expectations or intents, network operators often schedule flows to be updated in multiple rounds and formulate the update scheduling as an optimization problem with the objective like minimizing bandwidth violations or update completion time [1], [3], [14]–[19]. These update scheduling problems are often NP-Hard [3], and the optimization counterparts are intractable to solve with current commodity solvers (*e.g.*, CPLEX [20] or MOSEK [21]). Thus, network operators usually make a conscious effort to develop customized heuristics case-by-case such that flows can be changed as intended.

In this paper, we aim to simplify the flow updates and save operator efforts in update management. To this end, we propose a framework named **Atoman** that enables operators to express update problems and different intents on the update implementation with high-level APIs (see Fig. 1). To start a new update problem, network operators only need to tell the updating flows and associated update intents using **Atoman** APIs. **Atoman** will formulate an update scheduling optimization problem for each input update, translate intents into constraints and the objective of this optimization problem and generate satisfied update solutions that can be carried out by SDN control platforms.

Generality and efficiency should be two essential requirements for **Atoman**. First, **Atoman** should be *general* to express a broad spectrum of update intents that may arise in practical networks. Second, **Atoman** is expected to *efficiently* generate satisfied update solutions on a timescale that is responsive to network needs. Because of the diversity of intents and the intrinsic complexity of update management problems, either generality or efficiency is tricky, let alone both. But, we want to make it in this work.

Regarding generality, the key insight in **Atoman** is that update problems have limited types of intents and these updates can be formulated as update scheduling optimization problems. For many practical flow updates, we find that their update intents can be roughly classified into three categories: forwarding consistency, resource constraints and performance So, we can enumerate all possible intents of each category. For a given update, **Atoman** formulates it as an update scheduling optimization problem with constraints and objectives respecting specified intents. For example, if operators have an intention of avoiding congestion during the update, then **Atoman** adds constraints on the link load to not exceed the available capacity. According to the knowledge of traffic demands and flow paths before and after an update, it is easy to know how an update operation affects link load and thus formulate this constraint. Details of intent expressions can be found in Section IV.

However, the optimizations formulated for all updating flows inevitably introduce efficiency challenges because there are often a large number of flows to be changed in an update and the constraints and integer variables involved in these optimizations almost grow exponentially with updating flows and the network size. To address the efficiency challenge, we reduce the problem scale by optimizing the update scheduling only for some critical flows and network elements (*e.g.*, links and switches) while satisfying update intents. Meanwhile, we decompose the original large scheduling optimizations into a set of small ones, which enables **Atoman** to quickly solve each small problem and generate solutions in practice for all updates this work considered.

We summarize contribution of this work as follows.

- We study how to simplify the flow updates in SDN networks via raising the abstraction level of developing update solutions. To the best of our knowledge, this is the first work in this direction.
- We present a unifying framework, **Atoman**, for network operators to manage flow updates with various intents of consistency, constraints and performance in a simple way.
- We formulate flow updates as update scheduling optimization problems and propose efficient approaches to obtain the satisfied update scheduling solutions within seconds.
- Extensive simulations are conducted to evaluate **Atoman**, and our evaluations show that **Atoman** outperforms prior heuristics for scheduling flow updates, scales well as the network size and updating flows increase, substantially reduces the effort required to arrange new flow updates.

II. RELATED WORK

This section overviews recent research work on flow updates in SDN networks, in terms of the considered update intents on consistency, resource constraints and performance.

Consistency regarding forwarding is an essential requirement for the update of a single flow, and a lot of works studied the consistent update problem in recent years. From a historical perspective, the two-phase update technique [2], [22] was the first work that formally studied the consistency of update problems in SDN networks. The authors observed that packet may be forwarded inconsistently with neither the current nor the new state during the transition and showed that inconsistency might mostly impact some additional (*e.g.*, security) abilities of a network. So, they proposed to

preserve per-packet consistency (PPC) that per packet of an updating flow should traverse through a network entirely along the old path before the update or the new one after the update. To this end, Hong *et al.* [1] and Dudycz *et al.* [22] developed a two-phase update technique that lets switches to maintain both the old and the new rules during the update and tags packets with a rule version number at ingress switches to tell which version of rules to be applied.

Since the publication of [2], researchers are inspired by the two-phase update technique and proposed many methods to improve it.

Some researchers [11], [12], [23]–[26] followed the footsteps of [2] and studied the consistent single flow update problem. One of the main drawbacks of the two-phase update technique is that it requires all the switches to maintain both the old and new rules during the transition, doubling the rules on switches. Therefore, [12] and [23]–[25] aimed to reduce the rules added to switches. With analysis of many network applications that trigger flow updates, they observe that a significant amount of flows are changed for purposes unrelated to network functions (*e.g.*, security) and argued that the forwarding correctness is enough for many updating flows [12]. So, they focused on the forwarding correctness preservation and proposed to compute a sequence of rule replacements and replace switch rules to avoid additional rules on switches [12], [23]–[25]. Also, there are also some works [11], [26] that focused on the consistency related to forwarding policies (*e.g.*, enforcing packets to go through a waypoint [26] or link [11]). FLIP [11] proposed to combine the two-phase update technique and the ordered rule placement to guarantee the required forwarding policies during an update. Although ordered rule replacement based approaches can reduce the rule additions on switches, it is worth noting that they may fail because an ordered set of rule replacements that can consistently change a flow does not always exist [11].

In addition to the consistency, there are a lot of literature considering resource constraints and performance of an update implementation, especially for the multiple flow updates in SDN networks [1], [3], [14]–[19], [27]–[30]. In realistic networks, typically a vast many of flows are planned to update simultaneously to achieve network-wide goals. To change flows consistently, many previous flow updates approaches [1], [3], [14], [15], [17]–[19], [27], [29], [30] adopted the two-phase update technique to update each flow due to its operation simplicity, in addition to [16] that utilizes the two-phase update technique to change each segment of flows. However, the two-phase update technique has two main drawbacks when applied to multiple flow updates in a resource-constrained network. First, as mentioned above, the rules on switches are doubled, which will become exaggerated and easily exceed the limited size of the switch forwarding table as the number of updating flows grows. Second, although the two-phase update technique can atomically change a single flow, multiple flows are still updated asynchronously, which may cause link congestion during the transition and long-lasting damage to network performances [3]. Therefore, previous works [1], [3], [14]–[18], [27]–[30] on flow updates usually took capacity constraints of switches or links into account, and [1], [3], [14]–[18], [27], [29], [30] proposed to schedule flows to be updated in multiple rounds, each round updates a subset of flows, such that some intents on resource constraints and performance can be satisfied. [17] and [18] studied how to schedule updates to prevent the rule additions on switches exceeding the forwarding tables size. [1], [3], [14], [16] and [27]–[29] focused on avoiding congestion caused by transiently overloaded links, and [1], [3], [14], [16], [29] further aimed to achieve fast updates by minimizing the number of rounds required to complete the updates. Reference [15] focused on congestion-tolerant networks and aimed to minimize the congestion to complete the updates within a given time. Reference [30] disclosed that there exists a tradeoff between the update completion time and network throughput, and focused on balancing the intents on this two performances. In total, previous works considered many different intentions on updates and proposed customized heuristics to handle updates associated with specific intents they considered.

Till this year, [31] appeared to focus on simplifying network updates, which has the same vision of this paper. It only takes different levels of consistency into account, however, we further simplify update managements associated with different intents on resource constraints and performance preferences in addition to consistency.

In this paper, we focus on the multiple flow updates and propose to provide a unified framework for operators to manage flow updates easily. Despite the importance and frequency of flow updates in SDN networks, we find that existing flow update approaches are dedicated to specific or a few types of update intents, which are not general and hard to be applied to flow updates associated with different or new intents. Indeed, because of new requirements (e.g., as dictated by use cases like virtualized switches and network functions [32]–[35]) and unique opportunities (*e.g.*, programmability and flexibility) opened by SDN, the intents on update implementation to be satisfied tend to grow in number and complexity over time. Significant efforts will be required to develop new approaches/heuristics to deal with updates associated with new and more complex intents, which is not preferred concerning development and maintenance costs in the long run.

III. ATOMAN OVERVIEW

By and large, our vision of this paper is to eliminate some of the tedious work in managing updates with different intents, making it simpler for SDNs operators by high-level abstractions. To achieve this goal, we develop Atoman that enables operators to only focus on the high-level descriptions on updates and associated intents (see Fig. 1), shielding low-level details about problem formulations and optimization solvers. When starting a new update, network operators only need to "tell" Atoman the set of updating flows (including

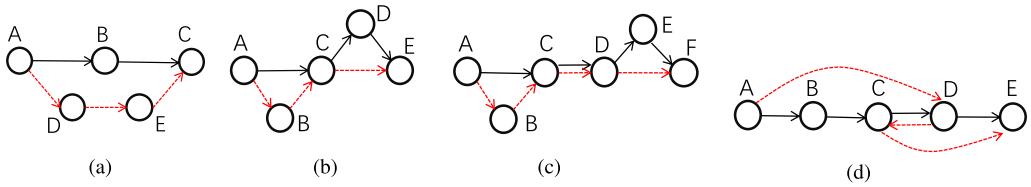


FIGURE 2. The paths of an updating flow also determines the segment generation, where solid lines indicate old paths and dotted lines indicate new paths. (a) $f_{A,C}$: the old path and the new one are disjoint. (b) $f_{A,E}$: the old and the new path share a in-path node. (c) $f_{A,F}$: the old and the new path share a link. (d) $f_{A,E}$: the old and the new path share a link but in the opposite direction.

the old&new rules and rate demands) and update intents using Atoman API functions. Then, Atoman builds an update scheduling optimization problem for this update by transforming input intents into the constraints and objective. By solving this problem, Atoman can obtain a satisfied update schedule solution. Finally, Atoman translates obtained update schedules into rule operations (*i.e.*, rule insertion/modification/deletion) that can be implemented by switches via SDN control platforms.

A. SEGMENT ABSTRACTION

For Atoman to be generic and expressive, we abstract the minimum scheduling and updating unit the and use this abstract to depict update schedule problems. Atoman adopts *segment* as a core abstraction and employs the two-phase update technique to update each segment automatically for simplicity.

Segments are flexibly created, which mainly depends on the concrete consistency intents of an update and the trajectory of forwarding paths. For an updating flow, Atoman creates one segment or multiple segments that can be updated independently respecting the required consistency. In particular, For a flow that requires to preserve per-packet consistency (PPC), Atoman directly takes it as a segment. While for a flow that only requires forward correctness consistency (both blackhole-freedom and loop-freedom), Atoman tries to partition it into several segments. For example, assuming that all the updating flows in Fig. 2 requires forwarding correctness, the flow $f_{A,C}$ in Fig.2(a) and the flow $f_{A,E}$ in Fig.2(d) are considered as segments because partitioning them into segments to update independently may break the network connectivity and cause packet losses during the updates. In contrast, the flow update $f_{A,E}$ in Fig.2(b) can be partitioned into two segments $f_{A,E}^{A,C}$ and $f_{A,E}^{C,E}$ by node C. The updating flow $f_{A,F}$ in Fig. 2(b) is similar, which can be divided into three segments $f_{A,F}^{A,C}$, $f_{A,F}^{C,D}$ and $f_{A,F}^{D,F}$ by link l_{CD}. Section IV present more details about segment generation.

The use of segment abstraction differs Atoman from previous scheduling techniques for flow updates. Many of exiting work [1], [3], [14], [17]–[19], [30] use the *flow* as the minimum scheduling and updating unit and adopt the two-phase update technique to guarantee a strong consistency at the level of per packet, even though a weaker consistency like forward correctness is enough for updating flows. In addition, we also noticed that there exists one work adopting the segment abstraction. But, it partitions each flow into segments according to a fixed criterion regarding congestion and connectivity [16]), which may only be suitable for specific updates and cannot guarantee additional consistencies like PPC. Atoman is entirely different from this work in that it partitions segments according to concrete required consistency. So, segments can be updated consistently and only need to be scheduled to satisfy intents on other aspects like resource constraints and performance.

B. TRAFFIC MIGRATION

When implementing an update in realistic networks, traffic migration inevitably happens as the forwarding paths of flows change. With the maturity of traffic splitting technique on dedicated middleboxes, servers or commodity switches [36]–[39], the traffic of a flow can be split and migrated from the old to the new path in multiple rounds, each migrates a fractional amount. Considering the widespread deployment of traffic splitting techniques [39], Atoman assumes splittable flows by default and move segments of flows in multiple rounds unless explicitly stated. This choice enables the system to be effective in reducing network violations (*e.g.*, congestion) during the updates. While there may be a few of networks that do not support traffic splitting techniques. Atoman provides APIs for operators to express this explicitly. In these cases, a flow is unsplittable and can only be migrated to the new path in one round.

C. UPDATES SCHEDULING

A critical insight in Atoman to achieve generality is that many flow updates can be modeled as update scheduling problems and formulations. In the backend of Atoman, every input update problem is formulated as an update scheduling optimization problem that builds upon segments. To be efficient, Atoman firstly divides the original big update scheduling problem into several small problems according to the user-specified and internal resource dependencies and constraints. Then, Atoman models each small problem as an optimization programming and uses an efficient heuristic to solve it and obtain the near-optimal segment scheduling solution within seconds.

IV. ATOMAN DETAILED DESIGN

This section presents the Atoman high-level API functions that network operators would like to use to portray flow updates problems and shows how updates can be formulated as update scheduling optimizations.

The network operator can instantiate an update via calling the createUpdateInstance function, which feeds Atoman the to-be-updated flows and old&new rules in the scope of this update. Besides, the operator can easily set up an update object via the getUpdateObj function and add intents regarding consistency, resource constraints and performance by using *constraint functions* (details see Table 2). Atoman takes the data from the called API functions to build a scheduling optimization for each update. Details are explained below.

A. NETWORK AND AN UPDATE

The preliminaries input of Atoman is a network and an update that includes a set of updating flows along with rate demands and current and final rules to be applied. A network is represented as a graph with nodes of switches and their interconnecting links. We consider a flow in its broadest sense, as the collection of all packets whose headers match a specific bitmask consistently across forwarding devices.

In addition to the network and updating flows and rules, operators also feed Atoman additional intents regarding consistency preferences, constraints on network resources, and performance goals pursued (see the following sections).

B. CONSISTENCY PREFERENCES

Consistency is essential to avoid packet losses and service interruptions caused by incorrect forwarding behaviors of switches in production networks. Atoman provides network operators API functions to express the following three levels of consistencies that are often expected to preserve throughout an update.

- *Forwarding correctness consistency* refers to the network ability to correctly deliver packets to destinations, which is a basic consistency. Forwarding blackholes and loops are two culprits that prevent the network from successfully forwarding packets to destinations. Forwarding blackholes are often caused by the absence of forwarding rules in switches, which causes packets to be dropped at switches; while evil forwarding loops are formed by several switches that forward packets back and forth infinitely and finally drop these packets when the TTL expires. Thus, to correctly forward packets to destinations, forwarding blackholes and evil loops should be excluded throughout an update. For an updating flow f, addNoBlackhole(f) can be called to specify the intent of avoiding forwarding blackholes and addLoopFreedom(f) is used to state the willingness to prevent evil forwarding loops. addNoBlackhole(f) and addLoopFreedom(f) together guarantees the forwarding correctness of flow f. The update of the large proportion of background flows often declare this type of consistency.
- *Policy enforcement consistency* relates to some additional properties of the forwarding behavior that need to preserve on the basis of correctness. For example, for security purposes, some flows may be expected to always visit one or several waypoints of function nodes like firewalls and DPIs for data scrubbing. addWayPoint(f,u) can be used to express such intent, which imposes the packets of flow f to visit a given waypoint u when traversing through the network. In addition, the traversal of several nodes in order may also be expected during the update. addServiceChain(f, *nodelist*) can be used to express this, where *nodelist* specifies the nodes to be traversed as well as the order of traversals.
- *Per-packet consistency* expresses that each packet should be atomically forwarded along either the old or the new path, but never a mixture of the two. addPerPacketConsistency(f) can be used to state this consistency intent on flow f. To protect flows generated by some very significant customers, network operators may claim such consistency to avoid malicious data manipulation and data leakage from the network layer.

C. FLOW MIGRATION

As mentioned before, Atoman schedules flow updates in the granularity of abstraction segment and migrate flows in a splittable manner by default. Let $x_{S^f,k}$ to denote the traffic proportion of the segment S^f of flow f to be migrated in the k-th round; thus we have

$$\forall f \in \text{flows}, \quad S^f \in \mathcal{S}^f, \ k : 0 \leq x_{S^f,k} \leq 1 \tag{1}$$

While, if operators want to force unsplittable migration, they can use the template function addUnsplittableMigration(f) or addUnsplittableMigration(). Then, the decision variable $x_{S^f,k}$ should have

$$\begin{aligned} & S^f \in \mathcal{S}^f, \quad k : x_{S^f,k} \in \{0,1\} \\ \text{or } \forall f \in \text{flows}, \quad & S^f \in \mathcal{S}^f, \ k : x_{S^f,k} \in \{0,1\} \end{aligned} \tag{2}$$

In addition, after the k round updates, a flow f should be migrated to the new path, which can be expressed as

$$\forall f \in \text{flows}, \quad S^f \in \mathcal{S}^f : \sum_k x_{S^f,k} = 1 \tag{3}$$

D. RESOURCE CONSTRAINTS

For real networks, network links and nodes are associated with limited capacities. We use C_l to indicate the capacity of link l in term of the maximum transmission rate it can achieve, and C_v to denote the capacity of switching node v in term of the maximum number of rules that can be added to respecting to the forwarding table size. Despite no resource violation before and after an update, the load on some links or switches may unexpectedly exceed the

TABLE 1. Key notations in the update problem formulation.

Network and update inputs (*required*)	
nodes:	the set of all nodes/switches in the network
links:	the set of all links in the network
flows:	the set of all to-be-changed flows in an update
$\mathcal{P}/\mathcal{P}'$	The initial/final configurations of all to-be-changed flows
Constraint- and performance-related parameters (*optional*)	
C_l	the available capacity of link l
C_v	the number of rules that can be added to node v with respect to its forwarding table size
θ	the allowed maximum utilization of network links
ϖ	the allowed maximum number of rules added to switches
τ	the expected update completion time
Internal and decision variables	
$\mathcal{S}^f$	the set of segments generated for an updating flow f
S^f	a segment of an updating flow f, $S^f \in \mathcal{S}^f$
$r_{S^f,v}/r'_{S^f,v}$	the number of rules added to switch v for S^f before/after the update
$b_{S^f,l}/b'_{S^f,l}$	the amount of traffic that link l carries for S^f before/after the update
$x_{S^f,k}$	the traffic proportion of a segment S^f to be migrated in the k-th round

capacity during the procedure of this update. Since violating resource capacities may lead to packet losses, service disruption or even network crashes, it is necessary to avoid resource violation throughout an update.

The template function addBandwidthCapConstraint(l,θ) express that the traffic load on link l should not exceed θ (*e.g.*, $\theta = 1$) time of the capacity C_l in any update round. Let's use $load_{l,k}$ to denote the maximum achievable load of link l in the k-th update round, then the internals can be expressed as

$$\forall k : load_{l,k} \leq \theta C_l \tag{4}$$

Let $z^k_{S^f,l}$ indicate the maximum possible traffic of segment S^f on link l in round k, and it can be calculated via update decisions $\{x_{S^f,k}|\forall(S^f,k)\}$.

$$\forall S^f \in \mathcal{S}^f, \quad l \in \textsf{links}, \; k : z^k_{S^f,l} = b_{S^f,l}(1 - \sum_{\tau=1}^{k-1} x_{S^f,\tau}) + b'_{S^f,l} \sum_{\tau=1}^{k-1} x_{S^f,\tau} + x_{S^f,k} \max((b'_{S^f,l} - b_{S^f,l}), 0) \tag{5}$$

Then, the achievable maximum load $load_{l,k}$ on link l in the k-th round can be calculated by $\sum_{f \in \textsf{flows}} \sum_{S^f \in S^f} z^k_{S^f,l}$.

Also, function addBandwidthCapConstraint(θ) can also be used to express capacity constraints on all network links. The corresponding internals of the optimization solver should be

$$\forall l \in \textsf{links}, \quad k : load_{l,k} \leq \theta C_l \tag{6}$$

For nodes with size-limited flow tables, we use addFlowtableConstraint(v, ϖ) to create constraints on the number of rules added to switch nodes. In particular, this function call enforces the rule additions on switch v to not exceed ϖ (e.g., $\varpi = 1$) time of the capacity C_v. Let $rules_{v,k}$ denote the achievable maximum number of rules added on switch $v$in the k-th update round, then the internal constraints of the optimization solver can be expressed as

$$\forall k : rule_{v,k} \leq \varpi C_v \tag{7}$$

Let $m^k_{S^f,v}$ indicate the maximum number of rules that need to be added to switch v to update segment S^f in round k, then we can compute $m^k_{S^f,v}$ according to update decisions $\{x_{S^f,k}|\forall(S^f,k)\}$.

$$\forall S^f \in \mathcal{S}^f, \quad v \in \textsf{nodes}, \; k : m^k_{S^f,v} = r_{S^f,v}(1 - \sum_{\tau=1}^{k-1} x_{S^f,\tau}) + r'_{S^f,v} \sum_{\tau=1}^{k-1} x_{S^f,\tau} + x_{S^f,k} \max((r'_{S^f,v} - r_{S^f,v}), 0) \tag{8}$$

The upper bound $rules_{v,k}$ of rule additions of on switch v in the k-th round can be calculated by $\sum_{f \in \textsf{flows}} \sum_{s^f \in S^f} m^k_{s^f,v}$.

Network operators can also create resource constraints on all switches via function addFlowtableConstraint(ϖ), and the internals are

$$\forall v \in \textsf{nodes}, \quad k : rule_{v,k} \leq \varpi C_v \tag{9}$$

E. PERFORMANCE PREFERENCES

Beyond consistency- and resource- related considerations, operators may also have preferences on performances (*e.g.*, update completion time) for an update.

Update completion time is a popular performance metric often associated with an update by network operators. This metric refers to the waiting time that new rules are applied to the network. It is important as it reflects the response speed of a network to the changing environment, especially for highly dynamic networks. Since the average time of updating a segment/flow by the two-phase update technique can be estimated [30], it is easy to transform a preferred update completion time into a counterpart of the number of update rounds. Therefore, same with existing works [1], [3], [14], [16], [30], we use the number of update rounds to reflects the update completion time in this paper.

If a flow f is expected to be updated within τ rounds, addUpdatetimeConstraint(f, τ) can be called to create a constraint for this intent. Since a flow may be partitioned into several segments, this constraint indeed requires all the segments of flow f to be updated within τ rounds.

$$\forall S^f \in \mathcal{S}^f, \quad k \geq \tau : x_{S^f,k} \geq 1 \tag{10}$$

addUpdatetimeConstraint(τ) constrain all updating flows to be updated within τ rounds.

$$\forall f \in \textsf{flows} : S^f \in \mathcal{S}^f, \quad k \geq \tau : x_{S^f,k} \geq 1 \tag{11}$$

[1]Because for a given upper bound on update time, one can calculate the upper bound on the number of update rounds according to [30].

TABLE 2. Atoman's selected consistency, resource constraints and performance objective functions for building optimizations for updates.

Group	Function	Description
Consistency (required)	addNoBlackhole(f)	Packets of flow f should be forwarded without forwarding blackholes
	addLoopFreedom(f)	Packets of flow f should be forwarded without forwarding loops
	addWaypoint(f, $nodelist$)	Packets of flow f are enforced to orderly traverse points specified in $nodelist$
	addPerpacketConsistency(f)	Packets of flow f should be forwarded consistently with either the old or the new rules
Network resource (selected)	addBandwidthCapConstraint(l, θ)	Traffic load on link l should not exceed θ times the capacity during the transition
	addFlowtableConstraint(u, ϖ)	Rules added to node u should be less than ϖ times the capacity during the transition
Others (optional)	addUpdatetimeConstraint(f, τ)	Flow f is desired to be updated within τ rounds
Objective	setUpdateObjective($name$)	Set the objective of an instance of flow updates (see Fig. 3 for details)

minUpdateTime	Minimize $\sum_{f}\sum_{S^f,k} x_{S^f,k}$
minLinkUtilization	Minimize $\max_{l} \theta_l$
minCongestion	Minimize $\sum_{l} \theta_l$
minMaxRuleAddition	Minimize $\max_{v} \varpi_v$
minRuleAddition	Minimize $\sum_{v} \varpi_v$

FIGURE 3. Atoman's common optimization objectives for update scheduling problems.

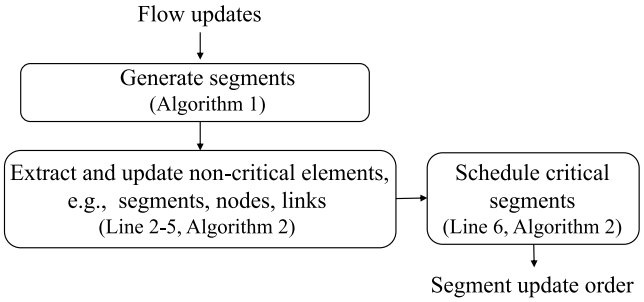


FIGURE 4. The workflow of Atoman's algorithms to schedule updates.

F. SPECIFYING UPDATE OBJECTIVES

The goal of scheduling flow updates is to optimize network-wide objective, *e.g.*, minimizing the update completion time, the link utilization, or the maximum number of rules added to switches. Fig.3 lists the most common goals, drawing on the flow updates considered in existing works [1], [3], [14]–[16], [30]. For example, operators may expect an update to be completed as fast as possible and thus add the following code snippet to express the intent of minimizing the number of update rounds.

```
opt.setUpdateObjective(minUpdateTime, round')
```

Operators can also express intents of minimizing the link utilization via a minLinkUtilization objective and specify other update objectives such as minMaxRuleAddition and minCongestion.

V. SEGMENT GENERATION AND UPDATE SCHEDULING

With the function templates provided by Atoman, a network operator can easily express an update and the associated intents. Then, the remaining question is how to arrange the input flow updates such that all associated intents can be satisfied. To this end, Atoman firstly generates segments for updating flows respecting to required consistency, and then schedules these segments to update to satisfy resource constraints, performance preferences and update scheduling goals, as Fig. 4 shown.

A. SEGMENT GENERATION

Taking updating flows and the desired consistency as inputs, Atoman runs Algorithm 1 to partition flows into segments. In particular, Atoman uses Algorithm 1 to partition each updating flow into one or several segments according to the concrete consistency intent. If per-packet consistency is desired, Algorithm 1 returns the updating flow itself as the segment (Algorithm 1: line 5-8). Otherwise, Algorithm 1 merges the old path P_f and the new one P'_f of an updating flow f as a directed forwarding graph DFG_f, and then create segments via using cut-vertice of DFG_f as partitions. Specifically, Algorithm 1 starts from the first node n, walks through the new path P'_f and produces a raw segment $S^f_{n,c}$ whenever reaching a cut node c of DFG_f. Algorithm 1 repeats the above steps to obtain all the raw segments (Algorithm 1: line 9-23). If forwarding correctness is desired, Algorithm 1 directly outputs all the raw segments. While if policy consistency of waypoint enforcement is wished, Algorithm 1 obtains final segments by adding a new segment composed of raw segments that contain waypoints and deleting those compositions (Algorithm 1: line 27-33).

B. UPDATE SCHEDULING

Given generated segments, the remaining work is to schedule them to update from the initial ($\mathcal{P}$) to the final ($\mathcal{P}'$) rule configurations respecting resource constraints and performance objectives. The segment update scheduling problem is naturally to be formulated as a MILP (Mixed Integer Linear Programming) with one of the objectives in Fig. 3 and some of the constraints in Table 2. However, we observe that there is often a large-scale number of flow segments to be updated,

Algorithm 1 Generate Segments for an Updating Flow

function generatesegment($\mathcal{P}_f$, $\mathcal{P}'_f$, *consistency*)
  $\mathcal{S}_f \leftarrow \varnothing$;
  n = the first node of $\mathcal{P}_f$ and $\mathcal{P}'_f$;
  m = the last node of $\mathcal{P}_f$ and $\mathcal{P}'_f$;
  if *consistency* = **PerPacketConsistency** **then**
    $\mathcal{S}_f.add(S^f_{n,m})$;
    return $\mathcal{S}_f$;
  end if
  $G_f \leftarrow$ **getMergedGraph**($\mathcal{P}_f$, $\mathcal{P}'_f$);
  ▷ Merge old and new paths into a forwarding graph.
  $V_{cut} \leftarrow$ **getCuts**(G_f);
    ▷ Find all the cut vertices in graph G_f.
  if $V_{cut} = \varnothing$ **then**
    $\mathcal{S}_f.add(S^f_{n,m})$;
    return $\mathcal{S}_f$;
  end if
  let c be the successor node of n in $\mathcal{P}'_f$
  while c != m **do**
    if $c \in V_{cut}$ **then**
      $\mathcal{S}_f.add(S^f_{n,c})$;
      update n to be c;
    end if
    update c to be the successor node of c in $\mathcal{P}'_f$;
  end while
  $\mathcal{S}_f.add(S^f_{n,m})$;
  if *consistency* = **ForwardCorrectness** **then**
    return $\mathcal{S}_f$;
  end if
  if *consistency* = **WaypointEnforcement** **then**
    $V_{wp} \leftarrow$ **getWaypoints**(f);
    $\mathcal{S}_{tmp} \leftarrow$ **pickSegment**(V_{wp}, $\mathcal{S}_f$);
      ▷ Pick all the segments from $\mathcal{S}_f$ that include one or several waypoints in V_{wp}
    $S' \leftarrow$ **Stitch**($\mathcal{S}_{tmp}$);
    ▷ Stitch all the segments in $\mathcal{S}_{tmp}$ together to be S'
    $\mathcal{S}_f.remove(\mathcal{S}_{tmp})$;
    $\mathcal{S}_f.append(S')$;
  end if
  return $\mathcal{S}_f$;
end function

which makes the update scheduling optimization problem difficult solve within real time. Thus, the challenge here is how to obtain the update schedule solutions efficiently. We adopt two simple yet efficacious tricks to reduce the problem size and accelerate the computation for the scheduling solutions.

The first trick is to reduce the update scheduling problem size by only formulating over updates of critical segments and network elements (e.g., links and switches). Our key observation is that only a subset of network elements may violate their resource capacities (or the expectation of resource usages) during an update and only a few of critical flow segments contribute to these violations. So, instead of scheduling the update for all segments, we can only schedule the updates of *critical segments* to remove violations on *critical network elements*. Meanwhile, the *non-critical segments* can be updated safely and silently and removed from the update scheduling optimization problem. In this following, we firstly show how to identify critical network elements, including links and nodes, and then present how to determine whether a segment is critical.

Critical/Congestion-risky links refer to links that are at risk of transient congestion/overload during an update. To find out critical links, we calculate the achievable maximal load of each network link during the implementation of an update. Since each flow segment S^f is updated by the two-phase update technique, the potential load values of each segment on each link l can be reduced to two, either $b_{S^f,l}$ or $b'_{S^f,l}$ [29]. We use the following calculation to find out whether a link $l \in$ links is at risk of transient congestion during the update. If this inequality holds, we consider link l is congestion-risky.

$$\sum_{S^f \in \mathcal{S}^f} ((\max(b'_{S^f,l}, b_{S^f,l})(1 - \sum_{\tau=1}^{k-1} x_{S^f,\tau}) + b'_{S^f,l} \sum_{\tau=1}^{k-1} x_{S^f,\tau}) \geq \theta C_l \quad (12)$$

Critical/Overflow-risky nodes have the similar concept with critical links, and they refer to switch nodes whose forwarding tables are at risk of transient overflow during an update. We use the following calculation to identify whether the forwarding table of a switch node $v \in$ nodes is at risk of transient overflow during the transition. If this inequality holds, we consider node v is overflow-risky.

$$\sum_{s^f \in \mathcal{S}^f} ((\max(r'_{s^f,l}, r_{s^f,l})(1 - \sum_{\tau=1}^{k-1} x_{s^f,\tau}) + r'_{s^f,l} \sum_{\tau=1}^{k-1} x_{s^f,\tau}) \geq \varpi C_v \quad (13)$$

With inequalities (12) and (13), we can easily obtain all the critical links and nodes of an update.

Algorithm 2 Schedule the Updates

function ScheduleUpdates(*Updates*, OPT_{raw})
  $S_{nc} \leftarrow \varnothing, L_{nc} \leftarrow \varnothing, N_{nc} \leftarrow \varnothing$; ▷ Store the non-critical segments, links and nodes.
  $S_{nc}, L_{nc}, N_{nc} \leftarrow$ **FindNCS**(*updates*) ▷ Extract the non-critical segments, links and nodes.
  Update(S_{nc}); ▷ Update non-critical segments
  $OPT_{tight} \leftarrow$ **RemoveConstriant**(OPT_{raw}, S_{nc}, L_{nc}, N_{nc}); ▷ Safely remove constraints for non-critical segments, links and nodes from the raw optimization model.
  solution $\leftarrow$ **Schedule**(OPT_{tight}) ▷ Schedule the update order for critical segments.
end function

Finding out critical segments. Given critical links and nodes, we identify the critical segments that are contributors. According to the traffic distribution and rule configurations before and after the update, it is not hard to know that updating a segment whether needs additional resources or frees up resources of critical network elements (*e.g.*, congestion-risky links and overflow-risky nodes). If updating a segment requires additional resources of a critical network element, we say this segment is critical as it is a contributor to of the formation of this critical network element. In addition, we can consider a segment to be non-critical if updating it does not need additional resources of all critical network elements.

The second trick is to accelerate the computation for the update scheduling solutions via recursively solving small update scheduling problems. In addition to the number of segments and network links and nodes, the bound of update rounds also largely affects the scale of an update scheduling optimization problem. Generally, a larger bound of the number of update rounds allows the solver to search a larger space for update scheduling solutions but reduces the efficiency to obtain the optimal/satisfied solution. To be efficient, we use the divide-and-conquer technique to divide the original problem into a series of small problems that can be solved efficiently. The technical details can be found in our early work [14].

C. CONFLICT HANDLING

Till now, we have seen that many flow updates can be expressed via the function templates and arranged through the internal solvers of Atoman. However, given that Atoman enables operators the ability to express an update with many different intents and requirements, we must declare that conflicts may exist if operators unintentionally express contradictory intents or requirements. For example, an operator may expect an update to be completed within R rounds and without congestion. However, this two expectations may contradict each other since it is impossible to simultaneously satisfy them by merely scheduling the updates. We have to deal with such conflicts since they may crash the Atoman's solver and their effects will persist if the operators are unaware of them.

To handle expectation conflicts of updates, we assign each concerned aspect a priority and make trade-off according to their priorities declared by operators when conflicts happen. It is based on the reality that the network operators have preferences for different update expectations, and we can use priority to reflect the difference in their preferences. Atoman provides API functions for operators to explicitly express their preferences of an update intent. This distinguishes Atoman from previous work that uses a fixed strategy to resolve such conflicts. When a conflict happens, we can remove it by trading the low-priority intent to satisfy the high-priority one. While if the conflicted two intents have the same priority, we schedule the flow updates to meet this two intents in a fair manner. For example, the operator may add the addPriority(*updatetime*, *congestion*) to declare that the *update completion time* has a priority over the *congestion*.

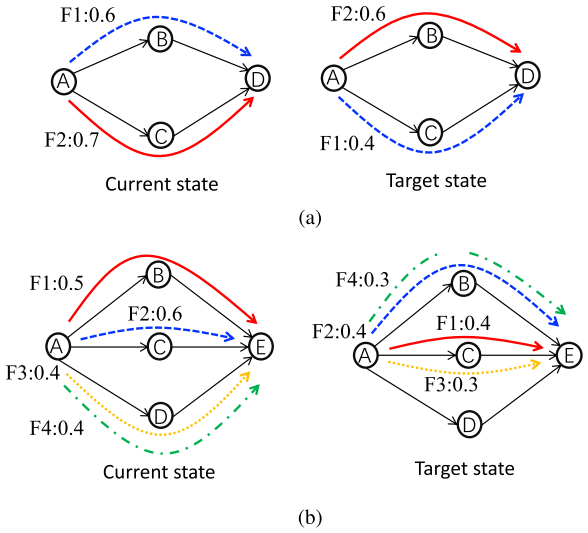


FIGURE 5. Two types of deadlocks may happen during the update scheduling. (a) An intrinsic deadlock between flows $f1$ and $f2$ that each of them cannot be updated without congesting the links. (b) An artificial deadlock between flows $f1$ and $f2$ if $f3$ and $f4$ are updated first. This deadlock can be avoided if $f3$ and $f4$ are updated after $f1$ and $f2$.

Then, for the conflict between the intents of congestion-free and completing the update within R rounds, we can resolve it by guaranteeing this update to be completed in R rounds and minimizing the transient congestion during the updates. Moreover, if a small amount of transient throughput loss is acceptable, we can remove such transient congestion by limiting the rate of some flows. Note that the implicit intents will be assigned the lowest priorities by default.

D. DEADLOCK HANDLING

Beyond the intents conflicts, there may exist some internal dependencies which cause deadlocks among the scheduling of updates. Deadlock refers to cases that no update operation can move on without violating some constraints or expectations on performances.

Let's consider the update in Fig. 5(a), where two unsplittable flows ($f1$ and $f2$) are swapping paths, and this update is expected to implement without congestion. The possible scheduling solutions of this update are $f1 \rightarrow f2$ or $f2 \rightarrow f1$. We can observe that this two update solutions will cause congestion on either path (A, B, D) or (A, C, D). There exists a deadlock between this tow updating flows. Fortunately, we have noticed that limiting flow rates can help break deadlocks [30]. Thus, we adopt the rate-limiting technique to break deadlocks and make the update move on by reducing the rate of some flows. For example, if the rate demand of flow $F2$ in Fig. 5(a) is reduced to 0.4 in the new configuration, then we can get a deadlock-freedom update schedule of $f2 \rightarrow f1$. After the update of $f1$ (remove from the top path to the bottom one), we can restore the rate of $f2$ to 0.6. In addition, rate-limiting can also reduce the update time. For example, if the rate of $f1$ in Fig. 5(a) is limited to 0.3, we can

swap the paths for $f1$ and $f2$ in one round, saving 50% time to complete this update. While if operators expect to maintain the flow rate to be no less than a given value, we can use the methods in our previous work [30] to deal with it.

VI. EVALUATION

This section firstly shows that network violations (e.g., link congestion) might happen if updating flows straightforwardly using the two-phase update technique without scheduling and then conducts extensive experimental simulations to evaluate Atoman. The experiments show that Atoman 1) performs either good or better than state-of-the-art heuristics, 2) significantly reduces development effort in comparison to manually developing heuristics for updates with different intents, and 3) scales well because it only formulates for a small-scale of critical flow segments and can solve these problems efficiently by employing the divide-and-conquer technique.

A. SIMULATION SETUP

We implement Atoman in Python 2.7 and use MOSEK[1] (via its existing provided Python API) as our backend solver of update scheduling optimization problems. The MOSEK can be substituted with other solvers like CPLEX or Gurobi.

1) DATASET

1) Topologies: we conduct simulations over two networks: D-WAN topology used in Dionysus [3] with 8 nodes and 28 links, and GScale topology from Google [4] that is consist of 12 nodes and 38 links. 2) Updates generation: we assume a flow between every node pair and use random link failure events to generate updating flows. In each experiment, we randomly choose one or several links to fail and collect flows that need to be updated to bypass these failed links to reach destinations. We consider a flow to be an updating one if its post-failure path changes. Unless otherwise stated, we adopt the Dijkstra algorithm [40] to compute the shortest paths for all synthetic flows before and after link failures. In addition, we randomly generate the rates for updating flows under the network capacity constraint and use the traffic load factor to simulate various traffic load in the network.

2) METHODOLOGY

We evaluate the update management tasks studied in previous works (e.g., [3], [14]–[16]) and compare the performances achieved by Atoman and that of approaches used in the original papers. We compare Atoman with the following alternative approaches:

1) One-Shot: updates each flow by the two-phase update technique [2], and updates all flows in one-shot without scheduling that means flows can be updated in any order.
2) Dionysus [3]: manages flow updates that require per-packet consistency and congestion-freedom and schedule updates to minimize the number of update rounds. To guarantee the per-packet consistency, Dionysus applies the two-phase update technique to change each flow. While to achieve congestion-freedom and minimize the update rounds, it builds a dependency graph for updates and heuristically schedules the flow updates.
3) Cupid [16]: manages updates that only require forwarding correctness. It is a variant of Dionysus that firstly divides each flow into multiple segments according to congestion-risky links and then builds a dependency graph for segments to heuristically schedule the updates of segments. Atoman also partitions flows into segments, but it partitions flows flexibly according to the concrete consistency intent, which is entirely different from Cupid.
4) Atoman: the approach proposed in this work, which enables operators to manage the network updates with different intents easily.

In the following, we evaluate Atoman from a broad scope. We compare the following factors: 1) network violations concerning congestion-risky links, 2) update completion time in terms of the number of update rounds, 3) the time to obtain update scheduling solutions, 4) deadlocks and 5) the rate needs to be cut down to break deadlocks.

B. THE NECESSITY OF CAREFUL FLOW UPDATES SCHEDULING

As we all know, the two-phase update technique [2] can atomically update every single flow consistently with either the old configuration or the new one. For the case of multiple flow updates, the naive approach is to apply the two-phase-commit technique to change all to-be-update flows in one-shot. We want to firstly look at whether network links will be overloaded/congested or not if traffic flows are updated by this one-shot approach, and then investigate how severe the congestion can be.

We use varying load factors to simulate different traffic load and conduct experiments under different traffic load. In each experiment, we count the number $NL_{congest}$ of congestion-risky links whose achievable maximum load may exceed the available capacity during the updates. Since one-shot approach cannot control the time which path a flow exactly takes during the transition, flows may change in any order and may traverse through either their old or new paths. Hence, the achievable maximum traffic load over each link can be the sum of demands of to-be-moved-in flows and those of to-be-moved-out flows when the former is updated before the latter. It is easy to infer whether a link is at risk of congestion by comparing its possible maximum traffic load and capacity. We calculate the ratio of congestion-risky links as $\frac{NL_{congest}}{NL_{total}} \times 100\%$, where NL_{total} is the number of network links. Meanwhile, we count the update experiments that have congestion-risky links and refer them as congestion-risky experiments.

Fig. 6 shows the results over 5000 experiments across GScale and D-WAN networks. Fig. 6 only plots the results

[1] https://www.mosek.com/

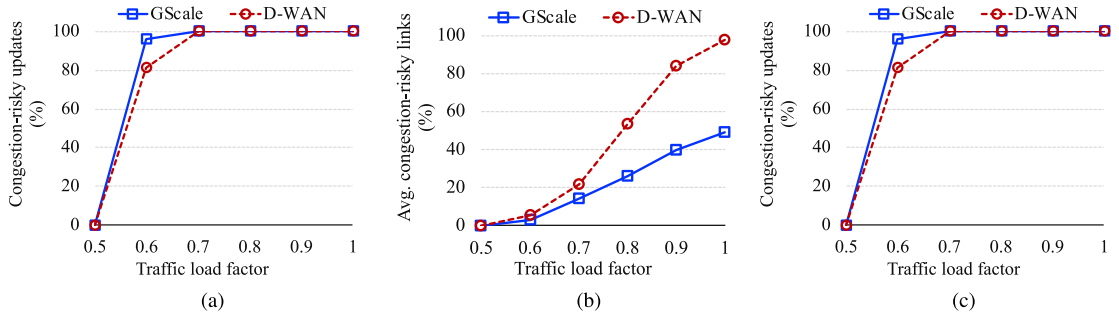


FIGURE 6. The congestion risk of applying one-shot commit to updates. (a) The updates that may incur network congestion grow sharply as the traffic load increases. (b) The links that may be congested almost grow linearly as the traffic load increases. (c) The flows that traverse congestion-risky links may be impacted, which grow drastically as the traffic load increases.

obtained in experiments that conducted with traffic load factor ranging from 0.5 to 1. In these experiments, the maximum (median) network utilization ranges from about 0.55 (0.23) to 1.0 (0.45). We omit to display these results due to the space limit. Fig. 6(a) highlights that a large proportion of updates, at least 80% and up to 100%, are at risk of congestion when the traffic load is larger than 0.5 in both simulated networks. These results indicate that network congestion may easily occur during the procedure of updates if one-shot approach is applied to update flows. In addition, from the experiment results in Fig. 6(b), we can observe that the average congestion-risky links almost increase linearly as the load factor increases from 0.5 to 1 for both networks. It is noticeable that the maximum (median) network utilization is 0.78 (0.31) when the load factor is 0.7. However, the near 20% congestion-risky links (in Fig. 6(b)) under this setting indicates that the network will be severely congested even though the network load is not hefty. Since the flows that traverse congested link may drop packets, decrease throughput or even lose connectivity, we collect the flows that traverse congestion-risky links to see how many updating flows may be impacted by such congestion during the updates. Fig. 6(c) shows the average results of flows impacted by congestion-risky links. Interestingly, We can observe that the impacted flows also almost increase linearly as the load factor increases. The at least 78% impacted flows, when the load factor is larger than 0.7, highlight the widespread impact of congestion on network updating flows. These results indicate that *updating all the flows in one shot may cause serious network congestion and impact a large proportion of updating flows.* Since such transient congestion may further cause a persistent impact on the performances of SDN networks, it is necessary to manage and schedule the updates of multiple flows carefully.

C. COMPARISON TO SPECIALIZED HEURISTICS

We compare **Atoman** to the state-of-the-art specialized heuristics of Dionysus [3] and Cupid [16]. Both approaches are proposed to handle the flow updates problems with intents of congestion-free and fast update completion time. But, they focus on different consistency intents. Dionysus is specifically targeted at per-packet consistency while Cupid focuses on forwarding-correctness consistency. To avoid transient congestion, both of them schedule flows to update in multiple rounds and adopt the rate-limiting technique to remove scheduling deadlocks via cutting down the rates of some flows.

We evaluate concerned approaches in experiments with a wide scope of load factors. For each load factor setting, we conduct 500 flow updates experiments and collect the number of update rounds, the number of deadlocks and the number of flows whose rate is cut to remove scheduling deadlocks in each experiment.

1) ATOMAN CAN COMPLETE FLOW UPDATES QUICKLY

Fig. 7 shows the results of the distribution of update rounds under different traffic load in two simulated networks. Overall, we can observe that **Atoman** outperforms both Dionysus and Cupid in terms of update rounds across all experiments. In addition, we can observe that the number of update rounds of three approaches increases as the traffic load increases and that of Dionysus and of Cupid increases more dramatically than that of **Atoman** with the increased traffic load. The results in Fig. 7(d), 7(h) highlight that **Atoman** can finish more than about 80% updates within 6 rounds, whereas Dionysus and Cupid need at least 4 and up to 15 rounds to complete these updates. The results indicate that **Atoman** can complete flow updates more quickly than Dionysus and Cupid, even though the traffic load is very heavy (e.g., load factor $= 1.0$) in networks. This results should thank the optimization model and global perspective of the **Atoman**'s scheduler, while Dionysus and Cupid only greedily schedule flow updates in a heuristic manner.

2) ATOMAN SCHEDULES UPDATES WITH FEW DEADLOCKS

We also count the updates that are scheduled with deadlocks by concerned approaches, and Fig. 8 shows the ratio of update schedules with deadlocks over the whole update experiments across various load factors and two networks. The results in Fig. 8 show that **Atoman**, Cupid and Dionysus only schedule 0-10% updates with deadlocks when the load factor is less than 0.8. However, with an increase of traffic load factor from

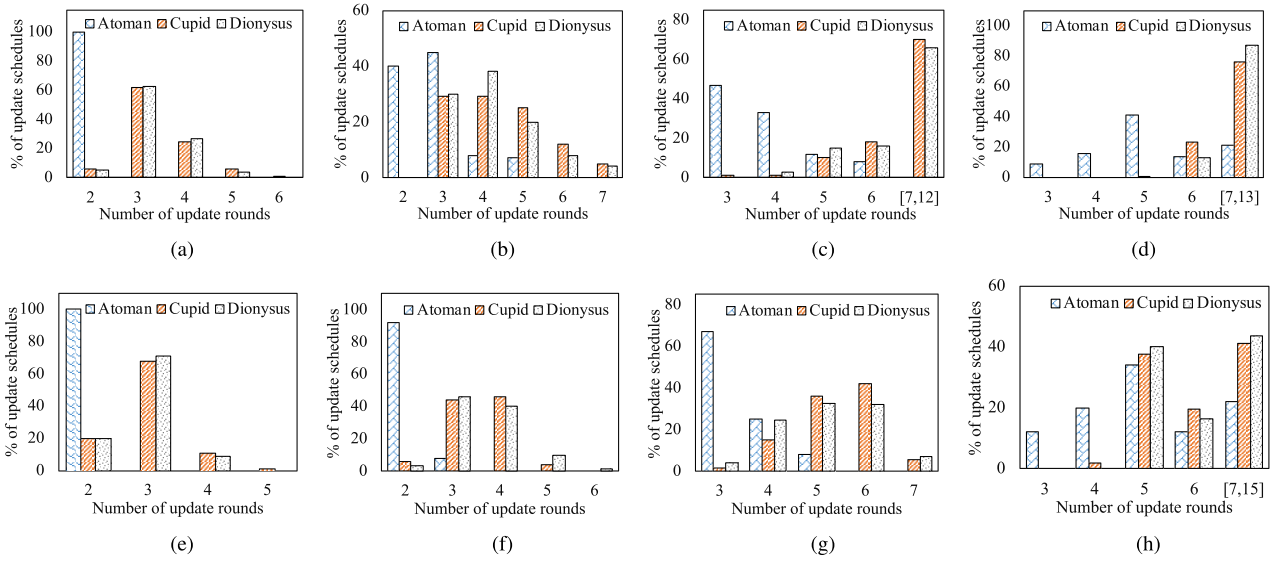


FIGURE 7. The number of update rounds under different traffic loads: Atoman can obtain update schedules with much less rounds than Dionysus and Cupid. (a) D-WAN:traffic load factor = 0.6. (b) D-WAN:traffic load factor = 0.8. (c) D-WAN:traffic load factor = 0.9. (d) D-WAN:traffic load factor = 1.0. (e) GScale:traffic load factor = 0.6. (f) GScale:traffic load factor = 0.8. (g) GScale:traffic load factor = 0.9. (h) GScale:traffic load factor = 1.0.

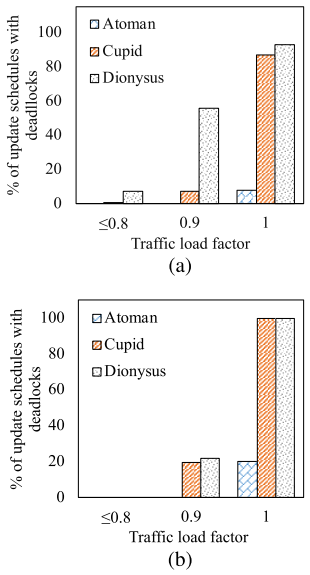


FIGURE 8. The update schedules with deadlocks: the deadlock-schedules of Atoman is no more than 20% over the update experiments while that of Dionysus and that of Cupid can achieve up to 100% when the traffic load is heavy (e.g., traffic load factor = 1).

0.9 to 1.0, we see that both Dionysus and Cupid run into a sharply increasing number of deadlock-schedules. Notably, for updates experiments simulated under load factor of 1.0, Dionysus and Cupid schedule 100% updates with deadlock, which marks that the network will be frequently trapped into deadlocks to update flows. While, **Atoman** can achieve no more than 20% deadlock-schedules, which indicates that it can schedule many updates without deadlocks and only obtain deadlock-schedules for very few updates.

3) ATOMAN ONLY LIMITS THE RATE OF A FEW FLOWS TO REMOVE DEADLOCKS

As mentioned before and stated in many existing network update works [3], [15], [16], [30], limiting the flow rate is a simple but efficient way to remove deadlocks and make updates move on without congestion. We collect the number $NF_{ratelimit}$ of updating flows that need to limit/reduce the rate to remove deadlocks and the amount $Rate_{reduce}$ of the reduced rate of limited flows in each experiment. We calculate the percentage of limited flows by $\frac{NF_{ratelimit}}{NF_{update}} \times 100$ and the percentage of the reduced rate by $\frac{Rate_{reduce}}{Rate_{total}} \times 100$, where NF_{update} and $Rate_{total}$ respectively denote the total number of updating flows and the overall rate amount of these updating flows. We here simulate flow updates that only require the per-packet consistency. As Cupid would reduce to Dionysus in such update cases, we here just compare **Atoman** with Dionysus and omit the comparison to Cupid.

Fig. 9 plots the CDF of the percentage of rate-limited flows of **Atoman** and Dionysus under experiments with a load factor of 1.0. We can see that **Atoman** limits much fewer flows than Dionysus, and **Atoman** only limits the rate of up to 25% flows while Dionysus limits the rate for up to 43% flows across experiments over D-WAN network. In addition, the results obtained over GScale network show that **Atoman** only limits the rate for at most 7% flows while that of Dionysus is up to 25%. These results indicate that Dionysus would affect much more flows than **Atoman**, which further lead to a more severe impact on network applications like degradations in services and QoS. We also collect the reduced rate of limited flows, and Fig. 10 plots the CDF results. We can see that **Atoman** only limits at most 20% rate for less than 20%-75% flows while Dionysus needs to reduce

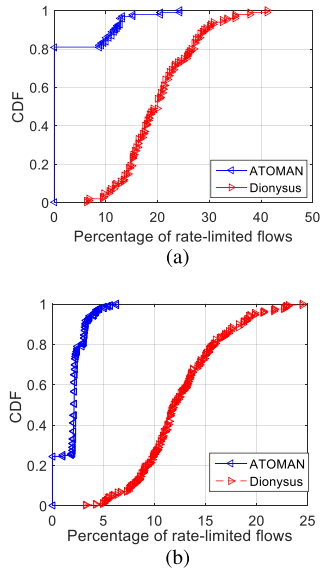


FIGURE 9. The number of flows that need to limit rate for deadlock removals. (a) D-WAN. (b) GScale.

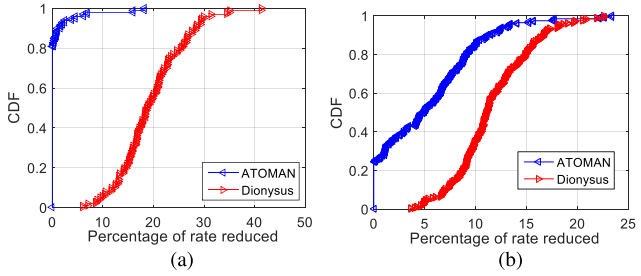


FIGURE 10. The flow rates reduced to remove deadlocks. (a) D-WAN. (b) GScale.

at least 3%-8% and up to 23%-41% of flow rate to remove deadlocks. These results show that compared to Dionysus, Atoman induce significant less rate reductions for much less updating flows, which indicates that very few impacts on network-wide performances (e.g., throughput and flow completion time) is caused even through scheduling deadlock happens.

D. IN-DEPTH ANALYSIS

1) Benefits of ATOMAN'S SIMPLICITY

We believe that Atoman enables operators to plan flow updates schedules in a more simpler manner, versus developing custom solutions from scratch and case by case. To quantify the benefits of simplicity, we respectively count the numbers of lines of code of our Atoman implementations and custom heuristics of flow updates schedule tasks. Then, to estimate the improvement in the"development effort" of operators, we calculate the ratio of the lines of code of the custom heuristics to that for Atoman implementations. Table 3 shows the results. The results of 30× and 35× improvement indicates that the "development effort" is significantly saved to manage flow updates for operators. This improvement is because that Atoman completely hides the particulars regarding optimization solvers from the developer. We acknowledge that lines-of-code comparisons are inexact, but we do not know of other ways to demonstrate the simplicity in the development effort. In addition, we believe that the improvements shown in Table 3 are conservative, as producing custom heuristic is much more complicated than writing Atoman code.

TABLE 3. Development effort benefits provided by Atoman.

Approaches	Atoman lines of code	Estimated improvement
Dionysus	25	30×
Cupid	30	35×

2) BENEFITS OF ATOMAN'S FLEXIBILITY

In addition to the simplicity, Atoman inherently supports flexible update scheduling. The flexibility of Atoman lies in two dimensions. On the one hand, Atoman supports both unsplittable and splittable flows, given the sophisticated techniques to the packet's out-of-order problem caused by the application of splitting flows on multiple paths (e.g., the old path and the new one). Meanwhile, it also enables an update to be scheduled in the granularity of either flow or segment. So to evaluate the benefits of Atoman's flexibility, we conduct update experiments under different traffic load and schedule updates in splittable and segment manner ("split+segment" in short) and unsplittable and flow one ("unsplit+flow" in short), respectively. Note that to explore the extreme cases, we here only simulate heavy traffic load because Atoman behaves well when the traffic load is light and medium, as shown in Fig. 7.

We first evaluate the benefits of Atoman's flexibility in terms of the update completion time (e.g., the number of update rounds). Fig. 11(a) and Fig. 11(d) plot the CDF of the number of update rounds in our experiments, where the number in brackets indicates the traffic load factor. We can see that "split+segment" policy can schedule updates within much few update rounds than "unsplit+flow" policy. In particular, we can also find that "split+segment" always schedules updates to be completed up to 5 rounds for D-WAN network and 6 rounds for GScale network, while "unsplit+flow" needs at least 6 rounds to complete 20% updates and even more than 10 rounds to finish some updates.

Meanwhile, we conduct experiments where updates are expected to be completed within a given time. To guarantee the update completion time, some other performances (e.g., congestion-free) may be compromised to obtain satisfied update schedules. To measure such performance compromise, we collect the number of update schedules that are at risk of congestion and the congestion amount of links (e.g., the amount of traffic load that exceeds the link capacity). Fig. 11(b) and Fig. 11(e) show the percentage of updates scheduled with congestion under various traffic load. We can see that "split+segment" always outperforms "unsplit+flow" because it always causes fewer updates to be scheduled with congestion. Interestingly, when traffic load

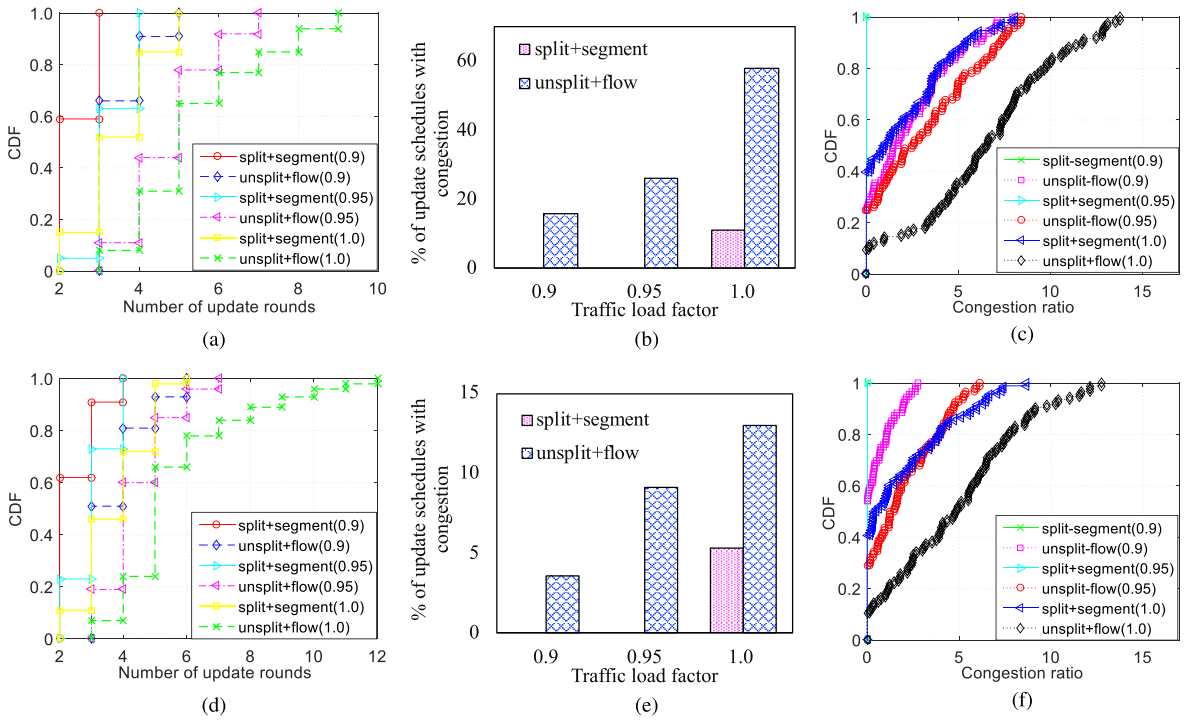


FIGURE 11. The benefit of Atoman's flexibility in update scheduling. (a) D-WAN. (b) D-WAN: updates restrict to be completed within 4 rounds. (c) D-WAN: updates restrict to be completed within 4 rounds. (d) GScale. (e) GScale: updates restrict to be completed within 4 rounds. (f) GScale: updates restrict to be completed within 4 rounds.

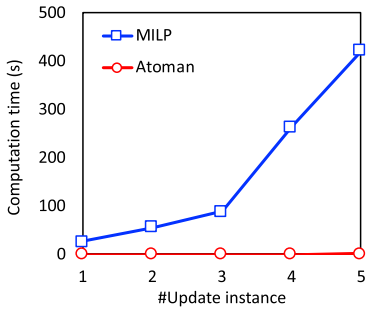


FIGURE 12. The runtime improvement of Atoman.

factor is no more than 0.95, "split+segment" can schedule all updates without congestion for both simulated networks while "unsplit+flow" schedules 20%-30% updates with congestion for D-WAN network and 5%-13% for GScale network. In addition, Fig. 11(c) and Fig. 11(f) plot the link congestion results in experiments where updates are expected to be completed within 4 rounds. These results show that "split+segment" achieves much less link congestion than "unsplit+flow", which is as expected.

3) TIME COST OF ATOMAN

At last, we evaluate the computation time for **Atoman** and MILP to compute the scheduling for updates with an intent of congestion avoidance. We generate five updates with an increasing number of updating flows from 300 to 7760. Fig. 12 shows that **Atoman** can always find the update scheduling within seconds. While, the computation time of MILP is up to more than 400 seconds for an updating of 7760 flows and keeps increasing dramatically as the number of updating flows grows. The results indicate that, by removing non-critical segments and links from the optimization model and splitting the original MILP into small MILPs, **Atoman** reduces the model size greatly and further finds the updating scheduling in real-time.

VII. CONCLUSION

Flow updates frequently happen to realize various purposes of application optimizations and network management in SDN networks. While flow updates are central in SDN networks, few efforts attempt to make it simple and easy to manage. In this paper, we provide a general, efficient framework for expressing and arranging flow updates in SDN networks. Our framework, **Atoman**, achieves both generality and efficiency via segment-centric abstraction. We show that **Atoman** can express updates with different intents, and often yields better performance than custom solutions. Thus, **Atoman** can lower the barrier to managing flow updates driven by current or novel SDN optimization applications and management tasks.

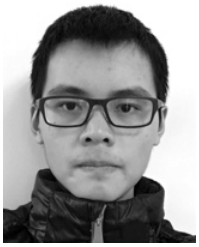
REFERENCES

[1] S. Luo, H. Yu, L. Luo, and L. Li, "Arrange your network updates as you wish," in *Proc. IFIP Netw.*, 2016.
[2] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. ACM SIGCOMM*, 2012, pp. 323–334.
[3] X. Jin *et al.*, "Dynamic scheduling of network updates," in *Proc. ACM SIGCOMM*, 2014, pp. 539–550.

[4] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *Proc. ACM SIGCOMM*, 2013, pp. 3–14.
[5] G. Sun, D. Liao, D. Zhao, Z. Xu, and H. Yu, "Live migration for multiple correlated virtual machines in cloud-based data centers," *IEEE Trans. Services Comput.*, vol. 11, no. 2, pp. 279–291, Apr. 2018.
[6] G. Sun, D. Liao, V. Anand, D. Zhao, and H. Yu, "A new technique for efficient live migration of multiple virtual machines," *Future Gener. Comput. Syst.*, vol. 55, pp. 74–86, Feb. 2016.
[7] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.
[8] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 3, pp. 1617–1634, 3rd Quart., 2014.
[9] K. He *et al.*, "Measuring control plane latency in SDN-enabled switches," in *Proc. 1st ACM SIGCOMM Symp. Softw. Defined Netw. Res.*, 2015, p. 25.
[10] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about SDN flow tables," in *Proc. Int. Conf. Passive Act. Netw. Meas.*, 2015, pp. 347–359.
[11] S. Vissicchio and L. Cittadini, "FLIP the (flow) table: Fast lightweight policy-preserving SDN updates," in *Proc. INFOCOM*, Apr. 2016, pp. 1–9.
[12] A. Ludwig, J. Marcinkowski, and S. Schmid, "Scheduling loop-free network updates: It's good to relax!" in *Proc. PODC*, 2015, pp. 13–22.
[13] K.-T. Föerster, S. Schmid, and S. Vissicchio, "Survey of consistent network updates," *CoRR*, vol. abs/1609.02305, pp. 1–34, Sep. 2016.
[14] L. Luo, H. Yu, S. Luo, and M. Zhang, "Fast lossless traffic migration for SDN updates," in *Proc. ICC*, Jun. 2015, pp. 5803–5808.
[15] J. Zheng, H. Xu, G. Chen, H. Dai, and J. Wu, "Congestion-minimizing network update in data centers," *IEEE Trans. Services Comput.*, to be published.
[16] W. Wang, W. He, J. Su, and Y. Chen, "Cupid: Congestion-free consistent data plane update in software defined networks," in *Proc. INFOCOM*, Apr. 2016, pp. 1–9.
[17] Y. Liu, Y. Li, Y. Wang, and J. Yuan, "Optimal scheduling for multi-flow update in software-defined networks," *J. Netw. Comput. Appl.*, vol. 54, pp. 11–19, Aug. 2015.
[18] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *Proc. HotSDN*, 2013, pp. 49–54.
[19] Y. Liu, Y. Li, Y. Wang, Y. Zhang, and J. Yuan, "On the resource trade-off of flow update in software-defined networks," *IEEE Commun. Mag.*, vol. 54, no. 6, pp. 88–93, Jun. 2016.
[20] *Cplex*. Accessed: Oct. 16, 2018. [Online]. Available: https://www.ibm.com/analytics/cplex-optimizer
[21] *Mosek*. Accessed: Oct. 16, 2018. [Online]. Available: https://www.mosek.com/
[22] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent updates for software-defined networks: Change you can believe in!" in *Proc. HotNet*, 2011, pp. 1–7.
[23] S. Dudycz, A. Ludwig, and S. Schmid, "Can't touch this: Consistent network updates for multiple policies," in *Proc. IFIP-DSN*, Jun. 2016, pp. 133–143.
[24] K.-T. Föerster, R. Mahajan, and R. Wattenhofer, "Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes," in *Proc. IFIP Netw.*, May 2016, pp. 1–19.
[25] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid, "Transiently secure network updates," in *Proc. ACM SIGMETRICS*, 2016, pp. 273–284.
[26] A. Ludwig, M. Rost, D. Foucard, and S. Schmid, "Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies," in *Proc. HotNets*, 2014, pp. 1–15.
[27] S. Brandt, K.-T. Föerster, and R. Wattenhofer, "On consistent migration of flows in SDNs," in *Proc. IEEE INFOCOM*, Apr. 2016, pp. 1–9.
[28] K.-T. Föerster and R. Wattenhofer, "The power of two in consistent network updates: Hard loop freedom, easy flow migration," in *Proc. IEEE ICCCN*, Aug. 2016, pp. 1–9.
[29] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zUpdate: Updating data center networks with zero loss," in *Proc. ACM SIGCOMM*, 2013, pp. 411–422.
[30] S. Luo, H. Yu, L. Luo, and L. Li, "Arrange your network updates as you wish," in *Proc. IFIP Netw.*, May 2016, pp. 10–18.
[31] L. Wang, Q. Li, Y. Liu, Y. Jiang, and J. Wu, "Simplifying network updates in SDN and NFV networks using GUM," in *Proc. IEEE ICCCN*, Jul. 2018, pp. 1–9.
[32] G. Sun, Y. Li, D. Liao, and V. Chang, "Service function chain orchestration across multiple domains: A full mesh aggregation approach," *IEEE Trans. Netw. Service Manage.*, vol. 15, no. 3, pp. 1175–1191, Sep. 2018.
[33] G. Sun, Y. Li, H. Yu, A. V. Vasilakos, X. Du, and M. Guizani, "Energy-efficient and traffic-aware service function chaining orchestration in multi-domain networks," *Future Gener. Comput. Syst.*, vol. 91, pp. 347–360, Feb. 2019.
[34] G. Sun, Y. Li, Y. Li, D. Liao, and V. Chang, "Low-latency orchestration for workflow-oriented service function chain in edge computing," *Future Gener. Comput. Syst.*, vol. 85, pp. 116–128, Aug. 2018.
[35] G. Sun, D. Liao, D. Zhao, Z. Sun, and V. Chang, "Towards provisioning hybrid virtual networks in federated cloud data centers," *Future Gener. Comput. Syst.*, vol. 87, pp. 457–469, Oct. 2018.
[36] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, "Let it flow: Resilient asymmetric load balancing with flowlet switching," in *Proc. NSDI*, 2017, pp. 407–420.
[37] P. Patel *et al.*, "Ananta: Cloud scale load balancing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 207–218, 2013.
[38] R. Gandhi *et al.*, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 27–38, 2015.
[39] O. Rottenstreich, Y. Kanizo, H. Kaplan, and J. Rexford, "Accurate traffic splitting on SDN switches," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 10, pp. 2190–2201, Oct. 2018.
[40] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959.



LONG LUO received the B.S. degree in communication engineering from the Xi'an University of Technology, in 2012, and the M.S. degree in communication engineering from the University of Electronic Science and Technology of China (UESTC), in 2015, where she is currently pursuing the Ph.D. degree. Her research interests include software-defined networks, inter-datacenter traffic engineering, and data-driven networking.



ZONGHANG LI received the B.S. degree in communication engineering from the University of Electronic Science and Technology of China (UESTC), in 2018, where he is currently pursuing the master's degree. His research interests include machine learning, network security, and data-driven networking.



JINGYU WANG received the Ph.D. degree from the Beijing University of Posts and Telecommunications, China, in 2008, where is currently an Associate Professor. He has published more than 50 academic papers in referred journals and conferences. His research interests include networking and communications, including future Internet, machine learning, traffic engineering, software-defined networks, network virtualization, cloud computing, game theory, and optimization.



HONGFANG YU received the B.S. degree in electrical engineering from Xidian University, in 1996, and the M.S. and Ph.D. degrees in communication and information engineering from the University of Electronic Science and Technology of China (UESTC), in 1999 and 2006, respectively, where she is currently a Professor. From 2009 to 2010, she was a Visiting Scholar with the Department of Computer Science and Engineering, University at Buffalo (SUNY). Her research interests include SDN/NFV, data center network, network for AI system and network security.