Chronus: Consistent Data Plane Updates in Timed SDNs

Jiaqi Zheng*§, Guihai Chen*†, Stefan Schmid‡, Haipeng Dai*, Jie Wu§,
*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210024, China
†Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, Shanghai 200240, China
‡Department of Computer Science, Aalborg University, Denmark
§Department of Computer and Information Sciences, Temple University, USA

***Abstract*—Software-Defined Networks (SDNs) introduce interesting new opportunities in how network routes can be defined, verified, and changed over time. Yet despite the logically-centralized perspective offered, an SDN still needs to be considered a distributed system: rule updates communicated from the controller to the individual switches traverse an asynchronous network and may arrive out-of-order, and hence lead to (temporary or permanent) inconsistencies. Accordingly, the consistent network update problem has recently received much attention. Motivated by the advent of tightly synchronized SDNs, we in this paper initiate the study of algorithms for consistent network updates in "timed SDNs"—SDNs in which individual node updates can be scheduled at specific times.**

This paper presents Chronus, which is based on provably congestion- and loop-free update scheduling algorithms, and avoids the flow table space headroom required by existing two-phase update approaches. We formulate the Minimum Update Time Problem (MUTP) as an optimization program. We propose a tree algorithm to check the feasibility and a greedy algorithm to find a update sequence in polynomial time. Extensive experiments on Mininet and numerical simulations show that Chronus can substantially reduce transient congestion by 75% and save over 60% of the rules compared to the state of the art.

***Index Terms*—SDN, network updates, clock synchronization, congestion-free, loop-free.**

I. Introduction

By decoupling and outsourcing the control over switches to a logically centralized server, Software-Defined Networks (SDNs) introduce interesting new flexibilities. In an SDN, the control plane can evolve independently of the data plane, enabling faster innovations. SDNs also introduce flexibilities in terms of traffic engineering, efficient failover, and network virtualization. In addition to the introduced flexibilities, the logically centralized perspective and the OpenFlow match-action paradigm enable formal verifiability. In principle, network policies can be specified and verified in an automatic manner [10].

However, SDNs also introduce new challenges. Despite the centralization of the control plane, an SDN needs to be regarded as a distributed system. In particular, the communication between the controller(s) and the switches occurs over a network: the times and orders in which update commands sent by the controller arrive and take effect at the different switches may be hard to predict. If no care is taken, such an out-of-order arrival could cause various inconsistencies, not only in terms of forwarding correctness, but also in terms of performance and security (policy compliance). For example, the fact that network updates do not occur *atomically* [20] in the data plane and a high degree of congestion may be introduced during the update, which would inevitably lead to packet loss and poor performance [7], [13].

This is problematic, as network updates are of increasing importance and are expected to happen more frequently in more flexible software-defined networks [5]. There are several reasons for this: (1) changes in security policies [14] (e.g., traffic from one subnetwork may have to be rerouted via a firewall before entering another subnetwork); (2) traffic engineering in the network [8] (to minimize the maximal link load, an operator may decide to reroute parts of the traffic along different links); (3) network maintenance work [12], [13] (e.g., in order to replace a faulty router, it may be necessary to temporarily reroute traffic); (4) reaction to link failures [24] (e.g., fast network update mechanisms are required to react quickly to link failures and determine a failover path).

For these reasons, the problem of consistent network updates has received much attention in recent years. Existing network update algorithms can roughly be classified into two categories: (1) two-phase update protocols and (2) node ordering protocols. Oversimplifying things slightly, the former approaches have the advantage that they are simple and relatively fast, however, they come with the drawback that they require packet tagging, which implies overheads in terms of additional forwarding rules to match these tags (additional flow table space headroom) and which causes problems in the presence of middleboxes [22]. The latter approaches have the advantage that they do not require packet tagging, but it has been shown that the corresponding scheduling algorithms come with strict tradeoffs in terms of the levels of transient consistency they provide and update time.

A. Our Contributions

In this paper we initiate the algorithmic study of a promising new approach to update networks consistently, which has the potential to overcome the drawbacks of the two approaches above. Our work is motivated by the advent of systems such as Time4 [16], [18] which promise a more predictable and synchronous data plane, allowing the coordination of network updates using accurate time, in the order of microseconds [17]. We introduce a natural and new optimization problem for

timed SDNs as we aim to find a network update schedule *which minimizes the overall network update time*, while ensuring loop-freedom and congestion-freedom at any moment in time. More specifically, this paper makes the following contributions:

1) **Problem formulation:** We introduce a novel problem motivated by the advent of more synchronous networks: we ask for accurate time schedules—specifying update time points for each switch—such that the total update time is minimized and congestion- and loop-freedom are ensured at any moment in time. We formulate this problem as an optimization program.
2) **Chronus and algorithms:** Our second contribution is Chronus, a system and set of algorithms for solving MUTP. Chronus does not require additional forwarding rules during the update and hence can be effectively applied to scenarios where the flow table space is limited. We first propose a tree algorithm to check the existence of a feasible congestion- and loop-free update sequence in polynomial time. Furthermore, based on the time-extended network model, we propose a fast greedy algorithm to tackle MUTP.
3) **Evaluation:** Our third contribution is a concrete implementation and evaluation of Chronus. In particular, we develop a prototype of Chronus on Mininet using a Floodlight controller. Extensive experiments and numerical simulations show that Chronus can substantially reduce transient congestion by 75% and save over 60% of the forwarding rules compared to the state of the art.

II. An Optimization Framework

A. A Motivating Example

We consider a Software-Defined Network (SDN) where a controller updates the forwarding rules at the switches whenever a route changes. Fig. 1(a) illustrates a simple example: there are six switches $v_1, \ldots, v_6$ and the link capacity is one unit. The transmission delay of each link is assumed to be one time unit in this example. That is, if one unit of flow leaves switch u at time t on the link $\langle u, v\rangle$, one unit of flow arrives at switch v at time $t+1$. The demand of the "dynamic flow" is one unit, which is routed from the source v_1 to the destination v_6. The initial routing is depicted as a solid line and the final routing is depicted as a dashed line. The notion of dynamic flow used in this paper is inspired by [6]. In a dynamic flow, the utilization of a link varies over time. Going back to our example in Fig. 1(b), assume we first only update v_2: hence, the subsequent flow is routed directly to v_6 through the link $\langle v_2, v_6\rangle$. Note that at this point, due to the link propagation delay, the old flow is still on the path $\langle v_2, v_3, v_4, v_5, v_6\rangle$ and will arrive at v_6 after four time units. Before that, the congestion will happen if we route new flow on this path.

Prior work on the network update problem usually relies on one of two fundamental update techniques: **two-phase updates** [20] and **order replacement updates** [9], [15].

TABLE I
Key notations in this paper.

$\mathcal{F}$	The set of dynamic flow f
$\mathcal{V}$	The set of switches v
$\mathcal{E}$	The set of links $\langle u, v\rangle$
$\mathcal{G}$	The directed network graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
t_i	The time point. $t_{i+1} > t_i$
$\mathcal{T}$	The set of time point. $\mathcal{T} = \{t_0, t_1, \ldots, t_n\}$
$\mathcal{F}^{\mathcal{T}}$	The set of flows in the time-extended network
$\mathcal{V}^{\mathcal{T}}$	The set of switches $v(t)$, where $v \in \mathcal{V}$ and $t \in \mathcal{T}$
$\mathcal{E}^{\mathcal{T}}$	The set of links $\langle u(t_i), v(t_j)\rangle$
$\mathcal{G}^{\mathcal{T}}$	The time-extended network $\mathcal{G}^{\mathcal{T}} = (\mathcal{V}^{\mathcal{T}}, \mathcal{E}^{\mathcal{T}})$
$C_{u,v}$	The capacity of link $\langle u, v\rangle$
$P(f)$	The set of possible path in the time-extended network
p^{init}	The initial path for the dynamic flow f
p^{fin}	The final path for the dynamic flow f
d	The demand of the dynamic flow f
n	The number of the switches. $n = \lvert\mathcal{V}\rvert$
$\sigma_{u,v}$	The transmission delay for the link $\langle u, v\rangle$.
O_t	The dependency relation set at t, where $t \in \mathcal{T}$.

A possible order replacement update sequence is shown in Fig. 1(b) $\rightarrow$ (c) $\rightarrow$ (d). In the first round, v_2 is updated. Then v_3, v_4 and v_5 are updated and finally v_1 is updated in the last round. In the second round, due to the asynchronous nature of the data plane, the new routing configuration for v_4 may become functional earlier than that for v_3. Thus a transient forwarding loop occurs since the flow passing through v_4 will be routed back to v_3 and then again arrive at v_4. Similarly, if the new routing configuration for v_5 is functional earlier than that for v_3 and v_4, the old flow on the path $\langle v_2, v_3, v_4, v_5\rangle$ will pass through the link $\langle v_2, v_6\rangle$ from $\langle v_5, v_2\rangle$. Note that v_1 is already updated in the first round and the new flow from v_1 will pass through $\langle v_2, v_6\rangle$. Here the new flow and the old flow together will result in a transient congestion on the link $\langle v_2, v_6\rangle$ as the sum of flow demand is two units, which are beyond the one unit link capacity. As for two-phase updates, it doubles the number of forwarding rules during the update and hence cannot be applied to scenarios where the flow table space is limited.

The timed updates can effectively solve this problem. Fig. 1(e) $\rightarrow$ (f) $\rightarrow$ (g) $\rightarrow$ (h) shows a congestion- and loop-free timed update sequence. Switch v_2 is updated at t_0. And then v_3 is updated at t_1. Next v_1 and v_4 are updated simultaneously at t_2. Finally, v_5 is updated at t_3. The congestion- and loop-free conditions are ensured at any moment in time as shown in Fig. 2(d). This timed update plan can be acceptable in practice because the updates can be scheduled accurately on the order of one microsecond [17]. In addition, we only modify the action in the flow table during the update process, which cannot incur additional flow table space headroom and overcome the drawback of two-phase updates.

B. Dynamic Flow Model And Problem Formulation

Before formulating the problem, we first present our network model. A network is a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of switches and $\mathcal{E}$ the set of links with capacities

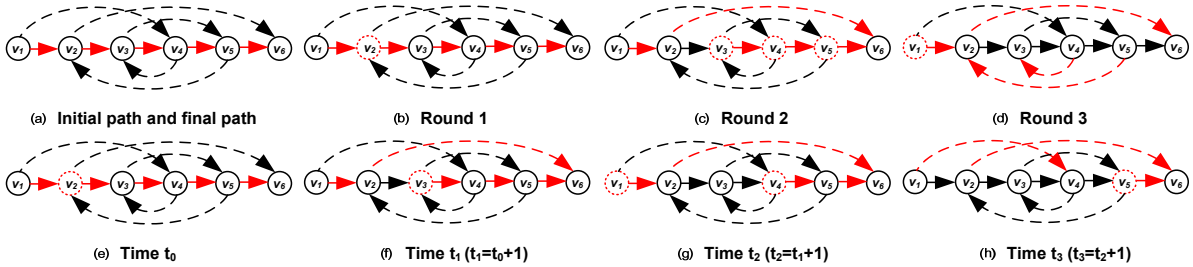
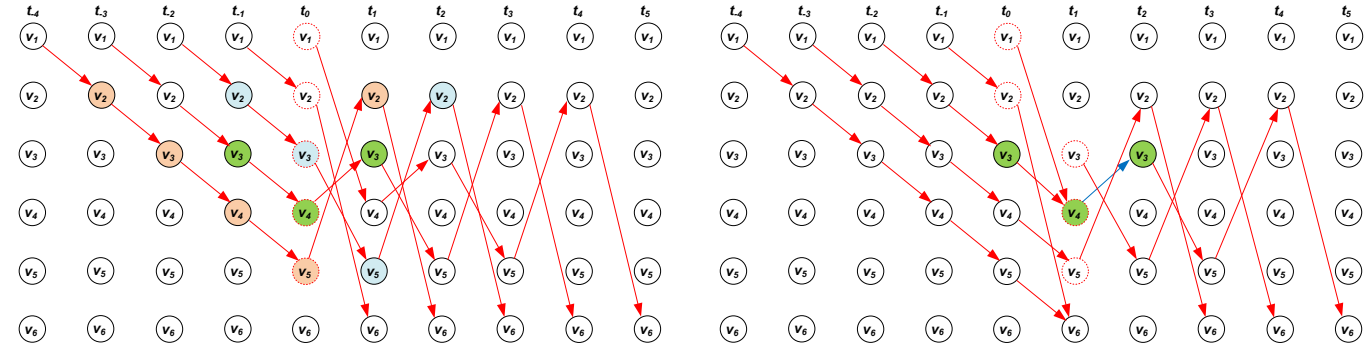


Fig. 1. Illustration of the network update problem considered in this paper. In this example topology, v_1 is the source and v_6 is the destination of both the old (initial) route and the new route. The initial routing is illustrated as a solid line, while the final routing is represented as a dashed line. The red solid and dashed links represent that the load on the link is greater than zero, which indicates that the dynamic flow is passing through this link. The black solid and dashed links represent the load on the link is zero. In our example, the link capacity and the link propagation delay is assumed to be one unit. The order replacement update sequence is: Fig. 1(b) → (c) → (d), while a congestion- and loop-free timed update sequence is: Fig. 1(e) → (f) → (g) → (h).



(a) If all the switches are updated at t_0, there would be three forwarding loops: $\langle v_2(t_{-1}), v_3(t_0)\rangle, v_5(t_1), v_2(t_2)\rangle$, $\langle v_3(t_{-1}), v_4(t_0)\rangle, v_3(t_1)\rangle$ and $\langle v_2(t_{-3}), v_3(t_{-2}), v_4(t_{-1}), v_5(t_0), v_2(t_1)\rangle$, which are depicted in different colors respectively.

(b) If we update v_1 and v_2 at t_0 and then update v_3, v_4 and v_5 at t_1, the capacity of the link $\langle v_4(t_1), v_3(t_2)\rangle$ cannot accommodate the flows from v_1 and v_3 at t_0 and the congestion occurs.

Fig. 2. Illustration of the different timed update sequences in the time-extended network.

$C_{u,v}$ and transmission time $\sigma_{u,v}$ for each link $\langle u, v\rangle \in \mathcal{E}$. The graph contains two paths: p^{init} and p^{fin}. The former is the old routing path which is depicted as a solid line in our example and the latter is the new routing path depicted as a dashed line. Both p^{init} and p^{fin} have the common source V^+ and destination V^-. For convenience, we summarize important notations in Table I.

Let us introduce four related notations first.

Definition 1: **Dynamic flow [6]:** A dynamic flow on G is a function $f : \mathcal{E} \times \mathcal{T} \to Z_+$ (Z_+ represents the set of positive integers) that satisfies the following conditions:

$$\sum_{(u,v)\in\mathcal{E}^+(v), t-\sigma_{u,v}\geq 0} x_{u,v}(t-\sigma_{u,v}) - \sum_{(u,v)\in\mathcal{E}^-(v)} x_{u,v}(t) = \begin{cases} -d & v = V^-, \forall t \in \mathcal{T} \\ 0 & \forall v \in \mathcal{V} - \{V^-, V^+\}, \forall t \in \mathcal{T} \\ d & v = V^+, \forall t \in \mathcal{T} \end{cases} \quad (1)$$

The dynamic flow conservation condition (1) indicates that if one unit of flow leaves switch u at $t-\sigma_{u,v}$ on link $\langle u, v\rangle$, one unit of flow arrives v at t. Here d is the flow demand, which is a positive integer. The time $\mathcal{T}$ is measured in discrete steps, where $\mathcal{T} = \{1, 2, \ldots, t\}$. $x_{u,v}(t)$ characterizes the load at t, which cannot go beyond the link capacity at each moment in time.

$$0 \leq x_{u,v}(t) \leq C_{u,v}, \forall \langle u, v\rangle \in \mathcal{E}, \forall t \in \mathcal{T} \quad (2)$$

Condition (2) ensures that the link capacity $C_{u,v}$ cannot be violated for $\forall t \in \mathcal{T}$.

Definition 2: **Loop-free condition:** If one unit of flow is routed through switch v at t, then it should not be routed through the switch v at t', where $t' > t$.

Definition 3: **Congestion-free condition:** The congestion-free condition holds if and only if Condition (2) always holds for $\forall t \in \mathcal{T}$ throughout the update process.

Our model and approach can be visualized nicely with a *time-extended network concept*: a network in which there is a copy of each switch for every time step $t_i \in \mathcal{T}$ and the links are redrawn between these copies to express their transmission delay. Succinctly:

Definition 4: **The time-extended network:** The time-extended network $\mathcal{G}^{\mathcal{T}}$ is a directed graph $\mathcal{G}$ with switches $v(t)$ for all $v \in \mathcal{V}$ and $t \in \mathcal{T}$. For each link $\langle u, v\rangle \in \mathcal{E}$ with transmission delay $\sigma_{u,v}$ and capacity $C_{u,v}$, the network $\mathcal{G}^{\mathcal{T}}$ has link $\langle u(t), v(t+\sigma_{u,v})\rangle$ with capacity $C_{u,v}$.

The time-extended network captures the dynamic process of flow transmission in the network. Fig. 2 gives a time-extended network example of Fig. 1(a), where t_{-1}, …, t_{-4}

and t_{-5} represent the history time steps, t_0 represents the current time step, and t_1, t_2, $\cdots$ represent the future time steps. We can only update the switches in the current and future time step and cannot update them in the history steps. The reason why we illustrate history steps is because we are required to check the existence of the forwarding loops defined in (2). In Fig. 2(a), the flow on the link $\langle v_1(t_0), v_4(t_1)\rangle$ starts at current time step t_0, while the flow on the link $\langle v_2(t_0), v_6(t_1)\rangle$, ..., $\langle v_5(t_0), v_2(t_1)\rangle$ starts at history time step t_{-1}, $\cdots$, t_{-4}, respectively. For simplicity, we do not draw the links in the time-extended network once the update is done.

Based on the above model and definition, we formulate the *Minimum Update Time Problem (MUTP)* as an integer linear program (3) in the time-extended network, where the initial (solid line) and final (dashed line) routing paths are given. We seek to find an optimal timed update sequence so as to minimize the total update steps, such that the congestion- and loop-free conditions hold at any moment in time. The path set $P(f)$ is pre-computed such that all paths are loop-free defined in (2). The resulting path set $P(f)$ are the input in our formulation.

$$\text{minimize} \quad |\mathcal{T}| \tag{3}$$

$$\text{subject to} \quad \sum_{f\in\mathcal{F}^{\mathcal{T}}} d \sum_{p\in P(f):\langle u(t_i),v(t_j)\rangle\in p} x_{f,p} \le C_{u(t_i),v(t_j)},$$

$$\forall \langle u(t_i), v(t_j)\rangle \in \mathcal{E}^{\mathcal{T}}, t_i, t_j \in \mathcal{T} \tag{3a}$$

$$\sum_{p\in P(f)} x_{f,p} = 1, \quad \forall f \in \mathcal{F}^{\mathcal{T}}, \tag{3b}$$

$$x_{f,p} \in \{0,1\}, \quad \forall f \in \mathcal{F}^{\mathcal{T}}, \forall p \in P(f). \tag{3c}$$

The formulation of the minimum update time problem is shown in (3). The objective aims to minimize the number of elements in set $\mathcal{T}$, i,e., the time steps during the update. The LHS of constraint (3a) characterizes the load of total flows at link $\langle u(t_i), v(t_j)\rangle$, which must be less than or equal to its capacity in order to meet the congestion-free condition defined in (3). The optimization variable $x_{f,p}$ indicates whether flow f is routed through path p in the time-extended network. This also determines that which switch should be updated at which time point. Constraint (3b) represents the flow can only be routed through one path in the time-extended network. The variable $x_{f,p}$ in constraint (3c) equals one if and only if the flow is routed through path p, and equals zero otherwise.

C. Hardness Analysis

***Theorem** 1:* MUTP is NP-complete.

Proof: Our proof works by reduction from [21], which is a special case in our problem where all the link delays are zero. As the work in [21] has already been proved to be NP-complete, our problem is NP-complete as well. ■

III. A Tree Algorithm

In this section we design a tree algorithm to check the existence of a feasible update sequence. The detailed process is shown in Algorithm 1. We first explain the high level of this algorithm. We construct a binary tree to perform node updates step-by-step, where the root at the top is the destination and the source node is located at the bottom of the left or right branch. If the source node belongs to the left branch, we update one of the nodes whose outgoing dashed line points from the left to the right branch. Then the source node belongs to the right branch and accordingly the flow is routed through the new path once the update is complete. Next, we update a node whose dashed line points from the right to the left branch. We iteratively update the nodes from one branch to the other until all the nodes are updated. Note that the update operation from one branch to the other can always guarantee the loop-free condition, and thus we only need to check the congestion-free condition in our algorithm.

In Algorithm 1, the default root node is the destination V^-, which has no capacity constraints (line 1). We use $V^{\#}$ to denote a set of nodes whose updates have already been performed. The search process proceeds from the top to the bottom, and we add the nodes one-by-one into $V^{\#}$ (lines 5-10). If $V^{\#}$ is not empty, we merge the nodes into one node V' and record the minimum link capacity between them as $V'.cons$ (lines 12-13). Next we find node k through the incoming dashed line of node V'. By comparing the sum of the link delays between the new path $\langle k, V'\rangle$ and the old path p', we determine if the update of k is feasible or not (lines 14-19). After that we construct candidate path sets P_{v_i} (new path) and Q_{v_i} (old path), and update node v_i whose outgoing dashed line points from one branch to the other. We select one path p with the minimum path delay among the path set P_{v_i}, as well as its delay must be equal to or larger than that of the old path q (lines 20-22). If such a path p does not exist and $V'.cons$ cannot accommodate the old and new flows simultaneously, *false* is returned (lines 23-24). Otherwise, we update the node on the path p. The process is repeated until all the switches have been updated. A detailed example is illustrated in Fig. 3.

Based on the explanation above, we have the following theorem:

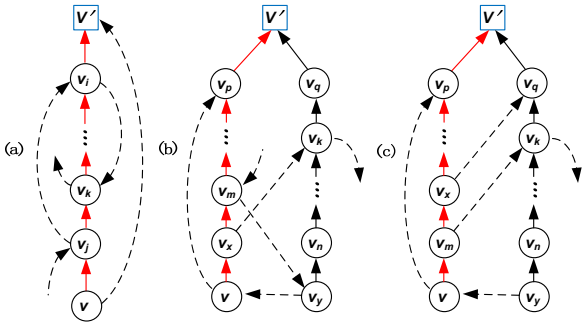


Fig. 4. Illustration of three network update scenarios shown in Algorithm 1.

***Theorem** 2:* Algorithm 1 can check the feasibility of Problem (3) in polynomial time if each link's transmission delay is identical.

Proof: Without loss of generality, we use the example in Fig. 4 to prove our theorem.

Case 1 (the update operation in line 18): As shown in

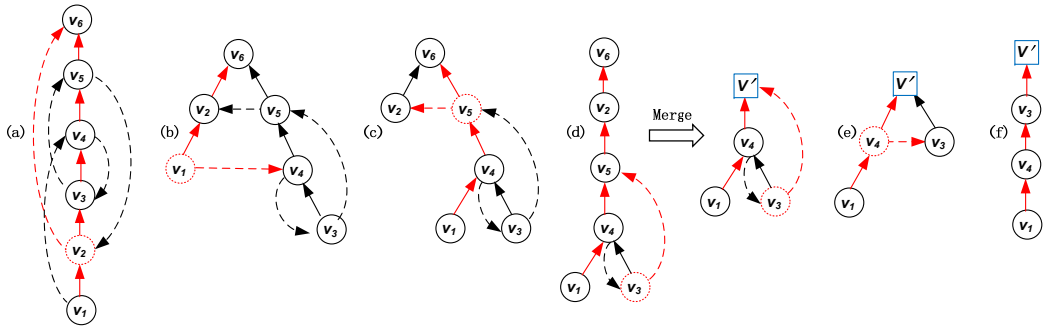


Fig. 3. An example for finding a feasible congestion- and loop-free update sequence. The flow demand and the link capacity are one unit. The delay at each link is one time unit. The path of the dynamic flow between source v_1 and destination v_6 is depicted as a red solid line. The red dotted circle represents that the node is updated in the current time step.

Algorithm 1 Checking the existence of a congestion- and loop-free timed update sequence

Input: The directed network $\mathcal{G}$; the initial path p^{init} and the final path p^{fin}; $\phi(p)$: the sum of link delay in path p.
Output: A boolean variance that indicates whether there exists a congestion- and loop-free update sequence or not.

1: $v = V^-$, $v.cons = +\infty$
2: $t = 0$
3: **repeat**
4: $V^{\#} = \emptyset$
5: **while** $v.in.dashedline.source = \emptyset$ **do**
6: **if** $v.in.solidline.source$ is not unique **then**
7: **break**
8: $u = v.in.solidline.source$
9: $V^{\#} = V^{\#} \cup \{u\}$
10: $v = u$
11: **if** $V^{\#} \neq \emptyset$ **then**
12: Merge all the nodes in $V^{\#}$ into one node, denoted as V'
13: $V'.cons = \arg\min_{\langle u,v\rangle \in V^{\#}} C_{u,v}$
14: $k = V'.in.dashedline.source$
15: $p' = \langle k, k.out.solidline.destination, \ldots, V'\rangle$
16: **if** k is active **and** $\sigma_{k,V'} \leq \phi(p')$ **and** $V'.cons < 2d$ **then**
17: **return false**
18: Update k at t
19: $t = t + \sigma_{k,V'}$
20: $P_{v_i} = \{\langle v_i, v_j, \ldots, V'\rangle | \langle v_i, v_j\rangle \in P^{fin}\}$
21: $Q_{v_i} = \{\langle v_i, v_i.out.solidline.destination, \ldots, V'\rangle\}$
22: $p = \arg\min_{p\in P_{v_i}, q\in Q_{v_i} | \phi(p)\geq\phi(q)} \phi(p)$
23: **if** $p = \emptyset$ **and** $V'.cons < 2d$ **then**
24: **return false**
25: **for** each node $z \in p$ **do**
26: Update z at t
27: $t = t + \phi(p)$
28: **until** all the switches are updated
29: **return true**

Fig. 4(a), if the update of v violates the congestion-free condition, both (4) and (5) hold at the same time:

$$V'.cons < 2 \cdot d \tag{4}$$

$$\phi(\langle v, V'\rangle) < \phi(\langle v, v_j, v_k, \ldots, v_i, V'\rangle) \tag{5}$$

Suppose there exists a path p^* such that the condition $\phi(\langle v, V'\rangle) > \phi(p^*)$ holds, p^* must contain at least an upward dashed link as any updates for downward links between v and V' will result in a forwarding loop in the current routing configuration. We assume that this upward dashed link is $\langle v_j, v_i\rangle$ and accordingly p^* is $\langle v, v_j, v_i, V'\rangle$. This indicates that the update time for v_j should be earlier than that of v. If the update is feasible, either (6) or (7) holds:

$$C_{v_j,V'} \geq 2 \cdot d \tag{6}$$

$$\phi(\langle v_j, v_k, \ldots, v_i\rangle) \leq \phi(\langle v_j, v_i\rangle) \tag{7}$$

However, the Condition (6) cannot be established as (4) holds. Thus Condition (7) must be established. Combining (5) and (7), we obtain,

$$\phi(\langle v, V'\rangle) < \phi(\langle v, v_j, v_i, V'\rangle)$$

This demonstrates that if the update of v is infeasible at the current time step, it is infeasible at any time step.

Case 2 (the update operation in line 26): As shown in Fig. 4(b), suppose the update time of v_x is earlier than that of v_m, so (8) holds in line 22 of Algorithm 1.

$$\phi(\langle v_x, v_k, v_q, V'\rangle) < \phi(\langle v_m, v_y, v_n, \cdots, v_k, v_q, V'\rangle) \tag{8}$$

If the update of v violates the congestion-free condition, both (4) and (9) hold.

$$\phi(\langle v, v_x, v_k, v_q, V'\rangle) > \phi(\langle v, v_p, V'\rangle) \tag{9}$$

Combining (8) and (9), we derive that:

$$\phi(\langle v, v_p, V'\rangle) < \phi(\langle v, v_x, v_m, v_y, v_n, \cdots, v_k, v_q, V'\rangle)$$

■

The inequation above indicates that the update of v is still infeasible, even though the update time of v_m is earlier than that of v_x. Similarly for the case shown in Fig. 4(c).

IV. A Greedy Algorithm

We now design a greedy algorithm which operates on the time-extended network, to solve MUTP. We explain how the algorithm works using the example in Fig. 2. At each time step, we plan to update as many switches as possible so as to minimize the total update time. In Fig. 2(a), assume all the switches (the destination switch v_6 does not require to be updated) are updated at t_0, and three forwarding loops will happen, as shown in Fig. 2(a): this violates the loop-freedom condition. Updating v_1 and v_2 at t_0 as shown in Fig. 2(b) is

also impossible, as the capacity of link $\langle v_4(t_1), v_3(t_2)\rangle$ cannot accommodate the flows from v_1 and v_3 simultaneously, which violates the congestion-free condition. To guarantee this, we use the dependency set to capture the update order among switches. According to the different link capacity constraints in the time-extended network, we construct the dependency relation set at each time step, as shown in Fig. 5. The detailed calculation process will be explained in Algorithm 3. We can observe that the dependency relation set at t_0 is $\{(v_2 \rightarrow v_4 \rightarrow v_3 \rightarrow v_1 \rightarrow v_5)\}$, where we can only update v_2. After that at t_1, the dependency relation set is $\{(v_3 \rightarrow v_1 \rightarrow v_5), (v_4)\}$. We can update v_3 and v_4 at the same time step, and this cannot violate link capacity constraints. However, a forwarding loop will happen if v_4 is updated. The procedure of checking forwarding loops is described in Algorithm 4. Therefore, only v_3 is updated at t_1. At the next time step t_2, we re-calculate the dependency relation set: it is $\{(v_1 \rightarrow v_5), (v_4)\}$. We update v_1 and v_4 simultaneously at t_2, and finally we update the last one v_5 at t_3. The whole update procedure is congestion- and loop-freedom.

Algorithm 2 Assigning a update time point for each switch

Input: The directed network $\mathcal{G}$; the initial path p^{init} and the final path p^{fin}; the number of switches n.
Output: A solution $\{v_i, t_j\}$ which indicates that v_i is updated at t_j.

1: Construct a set $\mathcal{V}^*$, which contains the to-be-updated switches
2: $\mathcal{T} = \{t_{-\sigma}, \ldots, t_{-1}, t_0, t_1\}$, where $\sigma = \sum_{k=1}^{n-1} \sigma_{v_k, v_{k+1}}$
3: Construct the time-extended network $\mathcal{G}^{\mathcal{T}}$
4: $t = t_0$, $i = 1$
5: **repeat**
6: Apply Algorithm 3 to obtain a dependency relation set O_t at t among the switches in set $\mathcal{V}^*$
7: **if** O_t contains a dependency cycle **then**
8: **return** $\emptyset$
9: **for** each $o \in O_t$ **do**
10: Pick the first element $\hat{v}$ from o
11: Apply Algorithm 4 to check whether there results a forwarding loop, if switch $\hat{v}$ is updated at t
12: **if** there is no forwarding loop **then**
13: Update switch $\hat{v}$ at t
14: $\mathcal{V}^* = \mathcal{V}^* - \hat{v}$
15: $t = t_i$
16: $i++$
17: $\mathcal{T} = \mathcal{T} \cup \{t_i\}$
18: Re-construct $\mathcal{G}^{\mathcal{T}}$ based on $\mathcal{T}$
19: **until** $O_t = \emptyset$

At the beginning of Algorithm 2, we construct $\mathcal{V}^*$, which represents the set of switches that need to be updated (line 1). The initial time set $\mathcal{T}$ contains the current time step t_0, the history time steps $\{t_{-\sigma}, \cdots, t_{-1}\}$ and the future time step t_1. We will add one future time step t_i at each loop, until all the switches are updated, or the update is infeasible (lines 5-19). Based on the time step set $\mathcal{T}$, we construct the time-extended network (line 3). Furthermore, we calculate the dependency relation set O_t, which is obtained from Algorithm 3 and which will be discussed soon. If O_t contains a cycle, the algorithm terminates, indicating that a congestion-free update order does not exist (lines 7-8). Otherwise, we can update the switches according to the order in each dependency relation (lines 9-14). At the same time, we apply Algorithm 4 to check the possibility of forwarding loops (line 11). If the occurrence of a forwarding loop is impossible, we update $\hat{v}$ at t and remove $\hat{v}$ from $\mathcal{V}^*$ (lines 12-14). Finally, we add one further time step t_i to re-construct the time-extended network, and enter the next loop (lines 16-18).

Algorithm 3 Finding a dependency relation set

Input: The time-extended network $\mathcal{G}^{\mathcal{T}}$; time point t
Output: A dependency relation set O_t

1: **for** each $v_i \in \mathcal{V}^*$ **do**
2: **if** $v_i.include =$ **true then**
3: continue
4: $v = v_i(t).out.dashedline.destination$
5: $t' = t + \sigma_{v_i, v}$
6: $v' = v(t').in.solidline.source$
7: $\tilde{v} = v(t').out.solidline.destination$
8: **if** $C_{v,\tilde{v}} < 2 \cdot d$ **then**
9: $O_t = O_t \cup \{(v' \rightarrow v_i)\}$
10: $v'.include =$ **true**
11: $v_i.include =$ **true**
12: Merge the dependency relation set with the common element.

The procedure of determining the dependency relation set is shown in Algorithm 3. Let $\mathcal{V}^*$ be the set of switches that need to be updated. We start from an arbitrary switch $v_i \in \mathcal{V}^*$. If v_i is updated at t, the flow will be routed through the link $\langle v_i(t), v(t')\rangle$ in the time-extended network, where $t' - t = \sigma_{v_i, v}$ (lines 4-5). And then we find v' and $\tilde{v}$, which are the last hop and next hop switches of $v(t')$ respectively (lines 6-7). If the capacity of link $\langle v, \tilde{v}\rangle$ cannot accommodate the flows from v_i and v', we establish a dependency relation between them (lines 8-9) and will not take them into account in the next loop (lines 10-11). When the loop terminates (lines 1-11), we merge the dependency relation set with the common element (line 12). For example, we can merge $\{v_1 \rightarrow v_2\}$ and $\{v_2 \rightarrow v_3\}$ into $\{v_1 \rightarrow v_2 \rightarrow v_3\}$ since both of them have the common element v_2.

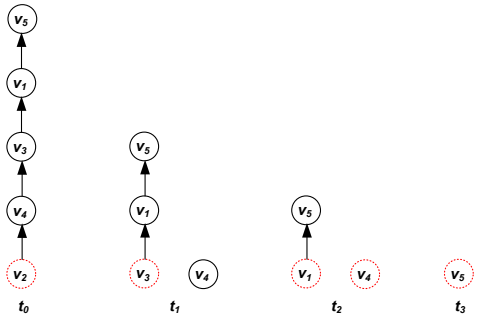


Fig. 5. Illustration of the resulting dependency relation set in the example of the time-extended network shown in Fig. 2. The red dotted circle represents that the switch is updated at the current time step.

Algorithm 4 describes how to check for the existence of a forwarding loop. We search the possible forwarding loops in the time-extended network. $v(t)$ represents the switch v at time step t, whose outgoing dashed line points to v^* (line 1). We look back through the incoming solid line of $v(t)$ and

Algorithm 4 Checking for forwarding loops

Input: The switch v; update time t
Output: A boolean variance which indicates if there exists a forwarding loop when v is updated at t.

1: $v^* = v(t).out.dashedline.destination$
2: **repeat**
3: $\hat{v} = v(t).in.solidline.source$
4: **if** $v^* = \hat{v}$ **then**
5: **return true**
6: **until** $\hat{v} = V^+$
7: **return false**

find the switch $\hat{v}$ (line 3). This searching procedure is repeated until the source switch V^+ is found. If v^* is equal to $\hat{v}$ during the searching procedure, it returns a true boolean variable that indicates a forwarding loop exists and the update operation of v at t is impossible (lines 4-5). If the condition (line 4) never holds during this procedure, *false* is returned that indicates the update is feasible (line 7).

Based on the analysis above, we have the following theorem:

***Theorem** 3:* The timed update sequence $\{v_i, t_j\}$ obtained from Algorithm 2 is congestion- and loop-free.

Proof: For each switch v, if it is updated at t, we go back to its last hop switch $v(t')(t' < t)$ in the time-extended network, to check whether the flow had already passed through it or not. This searching procedure stops when the source switch is found. Hence, the loop-freedom condition is always guaranteed by Algorithm 4. Furthermore, the resulting update order based on the dependency relation set from Algorithm 3 enforces the congestion-free condition. Therefore, Algorithm 2 can always produce a congestion- and loop-free update sequence if it exists. ■

V. Experimental Evaluation

We evaluate Chronus using both a prototype implementation and simulations.

Benchmark Schemes: We compare the following schemes with Chronus.

- **OR**: The order replacement updates [15] that minimize the number of rounds (i.e., the interactions between switches and the controller) and avoid the forwarding loops.
- **TP**: The two-phase updates [20] approach where we use VLAN IDs as version number in our experiments.
- **Chronus**: Our greedy algorithm as shown in Algorithm 2.
- **OPT**: The optimal solution of the integer program (3) obtained using the branch and bound method.

As discussed in Sec. I, the order replacement updates and two-phase updates both do not take network capacity and link transmission delay into account.

A. Implementation and Mininet Emulations

Implementation: We simplify Chronus with a control plane implementation: the forwarding rules are installed and updated via Floodlight's REST API. We use Floodlight's `sleep()` function to simulate time intervals and update the forwarding rules at a specified time point. Concretely, we develop a prototype of Chronus using the Floodlight 1.1 [1] controller with Openflow v1.3. We use the destination IP address as the matching field for forwarding. The corresponding routing configurations at source and destination switches are shown in Table II. For simplicity, we do not show the forwarding rules for ARP packets in Table II. ARP packets are flooded to all output ports. In our experiments, a flow is a traffic aggregate between source and destination switch.

TABLE II
Flow table at source switch R_1 and destination switch R_{12}.

Flow table at source switch R_1

Match Field				Action
InPort	SrcPfx	DstPfx	Tag	
host 1	—	10.0.0.2	—	Output: solidline
...	...	...	...	...
host 1	—	10.0.0.n	—	Output: solidline

Flow table at destination switch R_{12}

Match Field				Action
InPort	SrcPfx	DstPfx	Tag	
—	—	10.0.0.2	—	Output: host 2
...	...	...	...	...
—	—	10.0.0.n	—	Output: host n

We now describe how to perform a timed network update using our algorithms in our implementation. The procedure is shown in Algorithm 5. We first obtain a solution to MUTP using the greedy algorithm (line 1). Next we sort the results and record the maximum time as t^* from $\{t_j\}$ (lines 2-3). Then we sequentially examine every time step and update the corresponding switch set $\{v_i\}$. We first send the update messages (line 6), and then send the barrier request messages (line 7). Upon receiving the barrier response message, the update for switch v_i is completed (line 8). In Floodlight, OpenFlow barrier messages [2] are implemented by the `OFBarrierRequest` and `OFBarrierReply` classes. The algorithm sleeps for one time unit to simulate time intervals and then enters into the next loop (line 9).

Algorithm 5 Performing the timed network update

Input: The directed network $\mathcal{G}$; the initial path p^{init} and the final path p^{fin}.
Output: Update sequence of switch rules.

1: Apply Algorithm 2 and obtain solutions $\{v_i, t_j\}$.
2: Sort $\{v_i, t_j\}$ according to t_j.
3: $t^* = \arg\max_j t_j$
4: **for** $t = 0$ to t^* **do**
5: **if** $t = t_j$ **then**
6: Send messages to update $\{v_i\}$
7: Send barrier request message b_i to each v_i
8: Wait until receive all the barrier reply messages
9: Sleep for one time unit.

For completeness we now explain the implementation of two-phase updates and order replacement updates. The two-phase updates rely on packet tagging. We use VLAN IDs in packet headers to index stages. In the first phase, new rules—whose matching fields use the new VLAN ID that corresponds

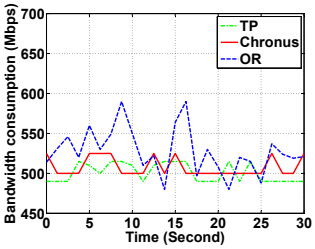


Fig. 6. Bandwidth consumption.

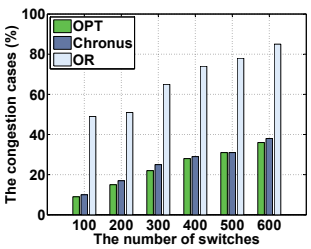


Fig. 7. Percentage of congestion cases.

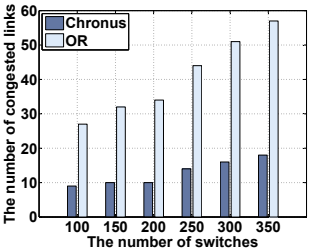


Fig. 8. # of congested links.

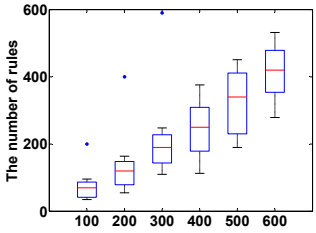


Fig. 9. # of forwarding rules.

to the second stage—are added. During this phase, flows are still forwarded according to existing rules as packets are still stamped with the VLAN ID of the first stage. Once the update is done for all switches, the protocol enters the second phase, when we stamp every incoming packet with the new VLAN ID. At this point the new rules become functional, and old rules are removed by the controller. The order replacement updates have already been proven to be NP-hard [15], when aiming to minimize the number of rounds while avoiding forwarding loops. We use the branch and bound method to obtain the optimal solution of this integer program. When performing updates in each round, our algorithm sleeps for a while, which is a random number from the data of [9], so as to simulate the asynchronous nature of data plane.

Mininet Setup: We conduct experiments on Mininet 2.2.1 [11], a high fidelity network emulator for SDNs, running on a PC with an Intel i5-2400 quad-core processor. We use OpenvSwitch version 2.3.1. Due to the single machine limitation of Mininet, we adopt a small scale network topology with 10 switches. We set the link capacity to 500 Mbps. The link delay is set to be an integer between 50ms to 10s.

Experiment Results: Fig. 6 shows that link bandwidth consumption varies with time during network updates. The aggregate flow rate is fixed at 500 Mbps. We use the Floodlight statistic module to measure the bandwidth consumption in a specified link every one second. The OpenFlow protocol does not provide a method to measure bandwidth consumption in the data plane. To determine bandwidth consumption, the controller queries the byte counters collected at every two time points. The difference between these two counters divided by the time intervals yields the bandwidth consumption. Usually, congestion happens when the bandwidth consumption is higher than the link capacity, and a larger value indicates more severe congestion in the network. We can observe that the peak value of OR is around 600 Mbps at 9th and 16th second, which can be beyond the buffer size and result in traffic loss, whereas the fluctuation of Chronus and TP is relatively stable and changes in the normal range.

B. Simulation

We also conduct extensive simulations to thoroughly evaluate Chronus at scale.

Setup. In addition to the small-scale network topology used in Mininet experiments, here we use a large-scale network topology. The initial routing path is fixed and the final routing path is chosen randomly (i.e., the final path is based on random routing). The initial and the final routing paths have the common source and destination. We run the algorithms on Intel i5-2400 quad-core processor. Each data point is an average of at least 30 runs.

We first investigate the percentage of congestion cases by comparing 500 different update instances in each run. In Fig. 7, the number of switches varies from 100 to 600 at the increment of 100 for each run. We find that Chronus performs very close to OPT with just slightly more congestion cases during updates. Specifically, when the number of switches is 600, more than 65% update instances using Chronus and OPT are congestion-free, while it is only 15% for OR. This demonstrates that Chronus in general leads to a small degree of congestion and significantly outperforms OR by around 60%. Fig. 8 shows the sum of congested links in comparison using the time-extended network. We can observe that Chronus can decrease the number of congested links by 70% compared with OR, especially when the number of switches becomes larger.

We now look at the rule space overhead of Chronus compared with TP. The box plot in Fig. 9 shows the number of rules for Chronus and the blue solid point shows them for TP. We do not show the results using TP when the number of switches is larger than 400, since its result is beyond the maximum value of the y-axis. We can see that the number of rules for TP increases more significantly than for Chronus, as the number of switches increases. Specifically, the average number of rules using TP and Chronus is 596 and 190 respectively, when the number of switches is 300. We observe that Chronus can save over 60% rules than TP on average as shown in Fig. 9. Note that these results become inaccurate for switches that apply longest prefix matching or wild-card rules. However, such rules are increasingly being substituted with exact match rules in SDNs [4], [9], [19].

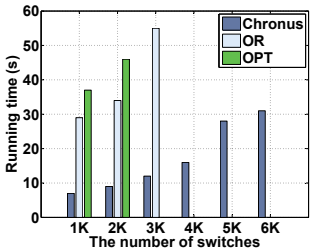


Fig. 10. Running time.

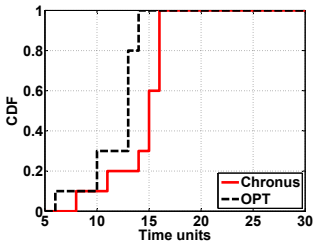


Fig. 11. Update time.

Finally we evaluate the running time and update time. The running time of Chronus, OR and OPT is illustrated in Fig. 10.

We do not include TP as it does not require to calculate the update sequence. We can observe that the running time of OR and OPT are both less than 60 seconds for up to 2K switches. When the number of switches is larger than 4K, OR and OPT do not complete within 60 seconds and the their required time is orders of magnitudes larger than Chronus. Chronus's running time is less than 60 seconds, even if the number of switches is 6K. Fig. 11 shows the CDFs of the update time when the number of switches is fixed at 40. We can see that most updates using Chronus finish within 15 seconds and OPT takes 13 seconds. The update time of Chronus can achieve near optimal performance compared to OPT.

VI. Related Work

We review prior art on network updates in SDNs. For a recent survey on the subject, we refer the reader to [5]. Reitblatt et al. [20] introduced a notion of per-flow consistent and per-packet consistent network updates. The authors also describe a two-phase commit protocol to preserve consistency when transitioning between two different routing configurations. FLIP [23] combines the advantages of the two-phase commit protocol and the order-based rules replacement technique, which preserves routing policies and significantly reduces the memory overhead during network updates. Ludwig et al. [15] aim to minimize the number of sequential controller interactions when transitioning from the initial to the final update stage. Another work by Ludwig et al. [14] considers consistent network updates in the presence of middleboxes. However, these works do not consider transient congestion. SWAN [7] and zUpdate [13] try to find congestion-free update plans in WAN and DCN, respectively. SWAN shows that if each link has a certain slack capacity, a congestion-free update sequence always exists. This condition is too strong to always hold in practice. Brandt et al. [3] show that a congestion-free update sequence still exists even if some links are fully utilized. Dionysus [9] employs dependency graphs to find a fast congestion-free update plan according to different runtime conditions of switches. Mizrahi et al. [16], [18] propose a time synchronization protocol between the controller and the data plane, which uses accurate timing to trigger network updates and reduce congestion. CCG [25] studies how to safely implement customizable consistency polices in order to minimize transition delay.

VII. Conclusion

We studied the problem of minimizing the route update time in timed SDNs. We proposed a tree algorithm to check the feasibility of an update in polynomial time and described a greedy algorithm to solve the problem. Our evaluation results show that our solutions can reduce transient congestion and save flow table space. We plan to continue our study by investigating approximation algorithms.

Acknowledgments

We thank the anonymous reviewers for their helpful comments on drafts of this paper. The work is partly supported by China 973 projects (2014CB340303), China NSF grants (61672353,61472252, 61321491, 61502229, 61373130), U.S. NSF grants CNS 1629746, CNS 1564128, CNS 1449860, CNS 1461932, CNS 1460971, CNS 1439672, CNS 1301774, ECCS 1231461, the Danish Villum project *ReNet*, and the program B for outstanding Ph.D. candidates of Nanjing University.

References

[1] Floodlight. http://floodlight.openflowhub.org/.
[2] Openflow switch specification. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.
[3] S. Brandt, K.-T. Forster, and R. Wattenhofer. On consistent migration of flows in sdns. In *INFOCOM*, pages 1–9, 2016.
[4] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A retrospective on evolving sdn. In *HotSDN*, pages 85–90, 2012.
[5] K.-T. Foerster, S. Schmid, and S. Vissicchio. Survey of consistent network updates. In *ArXiv Technical Report*, 2016.
[6] L. R. Ford and D. R. Fulkerson. Construct maximal dynamic flows from static flow. *Operation Research.*, 6:419–433, 1958.
[7] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven wan. In *SIGCOMM*, pages 15–26, 2013.
[8] V. Jalaparti, I. Bliznets, S. Kandula, B. Lucier, and I. Menache. Dynamic pricing and traffic engineering for timely inter-datacenter transfers. In *SIGCOMM*, pages 73–86, 2016.
[9] X. Jin, H. H. Liu, X. Wu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *SIGCOMM*, pages 539–550, 2014.
[10] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, pages 113–126, 2012.
[11] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *HotNets*, page 19, 2010.
[12] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter. Traffic engineering with forward fault correction. In *SIGCOMM*, pages 527–538, 2014.
[13] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz. zupdate: updating data center networks with zero loss. In *SIGCOMM*, pages 411–422, 2013.
[14] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid. Transiently secure network updates. In *SIGMETRICS*, pages 273–284, 2016.
[15] A. Ludwig, J. Marcinkowski, and S. Schmid. Scheduling loop-free network updates: It's good to relax! In *PODC*, pages 13–22, 2015.
[16] T. Mizrahi and Y. Moses. Software defined networks: It's about time. In *INFOCOM*, pages 1–9, 2016.
[17] T. Mizrahi, O. Rottenstreich, and Y. Moses. Timeflip: Scheduling network updates with timestamp-based TCAM ranges. In *INFOCOM*, pages 2551–2559, 2015.
[18] T. Mizrahi, E. Saat, and Y. Moses. Timed consistent network updates. In *SOSR*, pages 21:1–21:14, 2015.
[19] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, and S. Shenker. Software-defined internet architecture: decoupling architecture from infrastructure. In *HotNets*, pages 43–48, 2012.
[20] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, pages 323–334, 2012.
[21] S. S. Saeed Akhoondian Amiri, Szymon Dudycz and S. Wiederrecht. Congestion-free rerouting of flows on dags. In *ArXiv Technical Report*, 2016.
[22] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback-recovery for middleboxes. In *SIGCOMM*, pages 227–240, 2015.
[23] S. Vissicchio and L. Cittadini. Flip the (flow) table: Fast lightweight policy-preserving sdn updates. In *INFOCOM*, pages 1–9, 2016.
[24] J. Zheng, H. Xu, X. Zhu, G. Chen, and Y. Geng. We've got you covered: Failure recovery with backup tunnels in traffic engineering. In *ICNP*, 2016.
[25] W. Zhou, D. K. Jin, J. Croft, M. Caesar, and P. B. Godfrey. Enforcing customizable consistency properties in software-defined networks. In *NSDI*, pages 73–85, 2015.