Scheduling Congestion- and Loop-free Network Update in Timed SDNs

Jiaqi Zheng, *Member, IEEE,* Guihai Chen*, *Member, IEEE,* Stefan Schmid, *Member, IEEE,* Haipeng Dai, *Member, IEEE,* Jie Wu, *Fellow, IEEE,* Qiang Ni, *Senior Member, IEEE.*

***Abstract*—Software-Defined Networks (SDNs) introduce interesting new opportunities in how network routes can be defined, verified, and changed over time. Despite the logically-centralized perspective offered, however, 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. This can lead to (temporary or permanent) inconsistencies and triggered much research over the last years. 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. While technology enabling tightly synchronized SDNs is emerging, the resulting algorithmic problems have not been studied yet.**

This paper presents, implements and evaluates Chronus, a system which provides provably congestion- and loop-free network updates while avoiding the flow table space headroom required by existing two-phase update approaches. We formulate the Minimum Update Time Problem (MUTP) as an optimization program and propose two polynomial-time algorithms which lie at the heart of Chronus: a decision algorithm to check feasibility and a greedy algorithm to find a good update sequence. Extensive experiments on Mininet and numerical simulations show that Chronus can substantially reduce transient congestion and save over 60% of the rules compared to state of the art.

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

I. Introduction

Software-Defined Networks (SDNs) outsource and consolidate the control over switches to a logically centralized software. This introduces interesting opportunities to optimize and innovate communication networks: SDNs allow to evolve the control plane independently from the data plane, and introduce many flexibilities in terms of traffic engineering, efficient failover, and network virtualization. Moreover, network policies can in principle be specified and verified in an automated manner [9].

However, despite the centralization of the control plane, an SDN needs to be regarded as a distributed system. 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. The resulting out-of-order arrival can 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 take effect *atomically* [18] in the data plane may lead to congestion during the update, which in turn leads to packet loss and poor performance [6].

This is problematic, as network updates are expected to happen frequently in software-defined networks, for several reasons, including an increasingly more fine-grained traffic engineering [7] (to minimize the maximal link load, an operator may decide to reroute parts of the traffic along different links), adaptive changes in security policies and function virtualization [13] (e.g., traffic from one subnetwork may have to be rerouted via a flexibly allocated and virtualized middlebox before entering another subnetwork), network maintenance [11], [12] (e.g., in order to replace a faulty router, it can be necessary to temporarily reroute traffic), fast reaction to link failures [20] (e.g., fast network update mechanisms are required to react quickly to link failures and determine a failover path).

The problem of consistent network updates has received much attention over the last years and 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 [19]. 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.

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 [15], [17] which promise a more predictable and synchronous data plane, allowing the coordination of network updates using accurate time, in the order of microseconds [16]. 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

* The corresponding author is Guihai Chen (gchen@nju.edu.cn).

Jiaqi Zheng, Guihai Chen, Haipeng Dai are with State Key Laboratory for Novel Software Technology, Nanjing University, China. (e-mail: jiaqi369@gmail.com, gchen@nju.edu.cn, haipengdai@nju.edu.cn) Stefan Schmid is with Department of Computer Science, Aalborg University, Denmark. (e-mail: schmiste@cs.aau.dk) Jie Wu is with Center for Networked Computing, Temple University, USA. (e-mail: jiewu@temple.edu) Qiang Ni is with School of Computing and Communications, Lancaster University, UK. (email: q.ni@lancaster.ac.uk)

in time. More specifically, this paper makes the following contributions:

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 and prove its hardness.

Our second contribution is Chronus, a system and set of algorithms to solve MUTP. Our Chronus scheme 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 decision 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.

Our third contribution is a concrete implementation and evaluation of Chronus. In particular, we develop a prototype of Chronus on Mininet using OFSoftSwitch and Dpctl [1] as Openflow switches and the controller. Extensive experiments and numerical simulations show that Chronus can substantially reduce transient congestion and save over 60% of the forwarding rules compared to 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 [4]. 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** [18] and **order replacement updates** [8], [14]. A possible order replacement update sequence is shown in Fig. 1(b) $\rightarrow$ (c) $\rightarrow$ (d). In the first round, v_2 is updated. And 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 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) $\rightarrow$ Fig. 1(f) $\rightarrow$ Fig. 1(g) $\rightarrow$ Fig. 1(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 condition are ensured at any moment in time. Fig. 2(d) shows the timed updates process in the time-extended network, which will be discussed soon. This timed update plan can be acceptable in practice because the updates can be scheduled accurately on the order of one microsecond [16]. In addition, we only modify the action in the flow table during the update process, which neither requires packet tagging nor increases additional flow table space, and thus overcome the drawback of two-phase updates.

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 $t \in \mathcal{T}$.

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 $C_{u,v}$ 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 of p^{init} and p^{fin} have the common source V^+ and destination V^-. For convenience, we summarize important

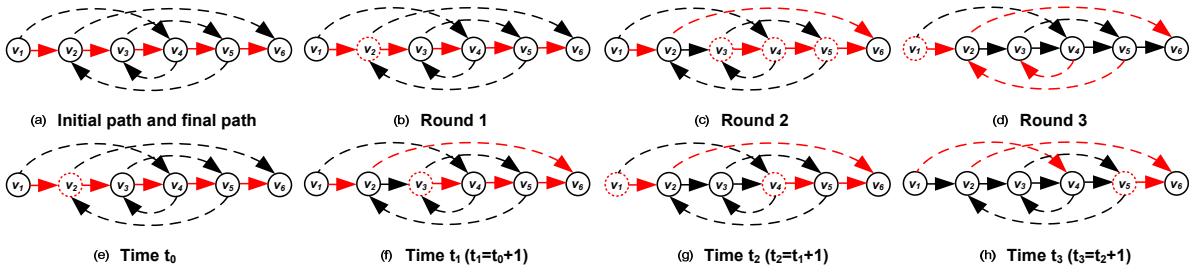
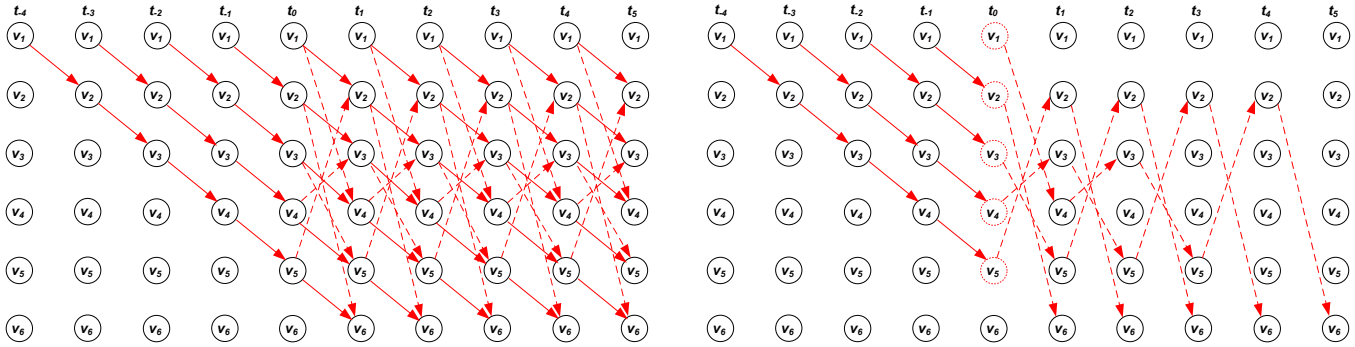
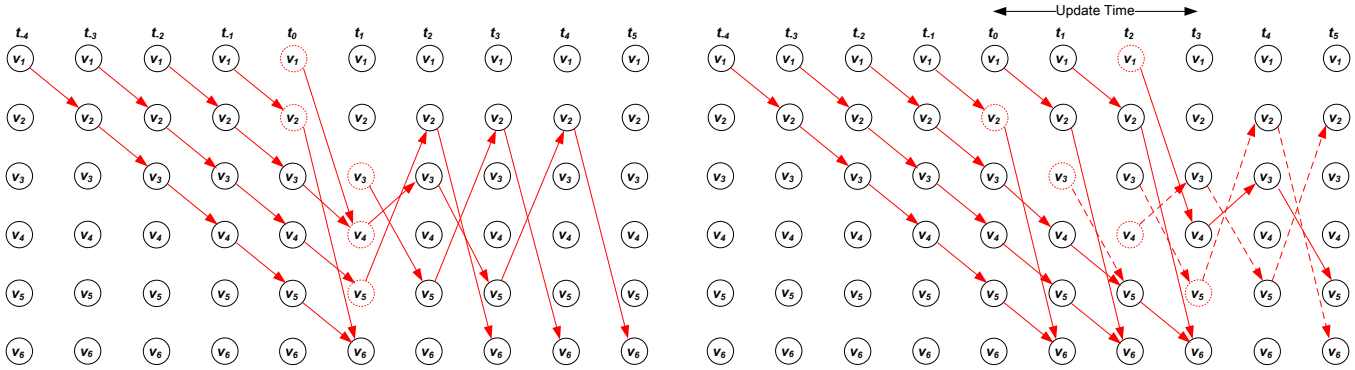


Fig. 1. Illustration of the network update problem considered in this paper. In this example topology, v_1 is the source and v_6 is the destination of both the old (initial) route and the new route. The initial routing is illustrated as a solid line, while the final routing is represented as a dashed line. The red links (both solid and dashed) represent that the load on the link is greater than zero, which indicates that the dynamic flow is passing through this link. The black links (solid and dashed) 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) The time-extended network model for all possible updates. The solid lines represent the initial routing path, while the dashed lines represent the final routing path. The link capacity and propagation delay are both one unit.

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



(c) 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.

(d) We first update switch v_2 at t_0, and update v_3 at t_1. Then we update v_1 and v_4 at t_2. Finally we update v_5 at t_3. The whole procedure is congestion- and loop-free.

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

notations in Table I. Let us introduce four related notations first.

***Definition** 1:* **Dynamic flow [4]:** 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(a) 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, 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 there is that we require to check the existence of the forwarding loops defined in (2). In Fig. 2(a), the flow on the link $\langle v_1(t_0), v_2(t_1)\rangle$ starts at current time step t_0, while the flow on the link $\langle v_2(t_0), v_3(t_1)\rangle$, ..., $\langle v_5(t_0), v_6(t_1)\rangle$ 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 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 condition always hold at any moment in time. The path set $P(f)$ is pre-computed such that all paths are loop-free defined in (2). Calculating all possible paths can be done in polynomial time as the in-degree and out-degree in the time-extended network are at most two. 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 successive 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. For example, as illustrated in Fig. 2(d), two flows starting at t_{-1} and t_2 are routed through the path $\langle v_1(t_{-1}), v_2(t_0), v_6(t_1)\rangle$ and $\langle v_1(t_2), v_4(t_3), v_3(t_4), v_5(t_5), v_2(t_6), v_6(t_7)\rangle$. Accordingly, we update v_2 and v_1 at t_0 and t_2, respectively. 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

We establish the hardness of MUTP below.

***Theorem** 1:* Computing a shortest congestion- and loop-free update sequence is NP-complete, already for a constant number of update rounds.

We refer the reader to our prior work [14] for a rigorous NP-hardness proof. Here we only give an intuition. We construct a polynomial reduction from the SAT problem [5] to our problem. Given a SAT boolean formula consisting of six boolean variables x_1, x_2, $\cdots$, x_5 and x_6 and seven clauses: $(x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee x_4 \vee x_5) \wedge (x_4 \vee x_5 \vee x_6) \wedge (\neg x_1 \vee x_5) \wedge (\neg x_2 \vee x_5) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_3 \vee x_6)$. The objective is to find a variable assignment that satisfies each clause. This problem can be reduced to the instance of MUTP, which is constructed as shown in Fig. 3. For each variable x_i in the formula we introduce one node named x_i in our instance. A true assignment for variable x_i corresponds to the update in the first time step. A false assignment corresponds to that in the second time step. There are three colors of nodes: white, gray and black. For gray nodes, we do not require to update them. For black nodes y_i, we can only update them in the second time step as the forwarding loop happens if any of them is updated firstly. As for white nodes x_i, we require to determine which nodes are updated in the first time step and which in the second time step.

Now we explain the meaning of each SAT clause. The clause $(x_1 \vee x_2 \vee x_3)$ indicates that at least one of the node x_1, x_2 and x_3 requires to be updated in the first time step. Otherwise, a forwarding loop $\langle u, v, x_1, x_2, x_3, y_1, u\rangle$ happens once y_1 is updated in the second time step. Similarly, we have clauses $(x_3 \vee x_4 \vee x_5)$ and $(x_4 \vee x_5 \vee x_6)$ to guarantee the loop-freedom. The clause $(\neg x_2 \vee x_5)$ represents that if x_2 is updated in the first time step, x_5 also should be updated together with x_2. Otherwise, a forwarding loop $\langle x_2, x_5, y_3, x_2\rangle$ happens which is depicted in the red line in Fig. 3. Similarly, we have clause $(\neg x_1 \vee x_5)$ and $(\neg x_3 \vee x_6)$. The congestion-freedom is guaranteed by the clause $(\neg x_1 \vee \neg x_2)$. It characterizes that x_1 and x_2 cannot be updated simultaneously, as the load of incoming flows from x_1, x_2 and x_4 is three units, which is beyond the link capacity. Therefore, a feasible variable assignment corresponds to a congestion- and loop-free update sequence within two time steps. The assignment results indicate that which white nodes should be updated in the first time step and which nodes in the second.

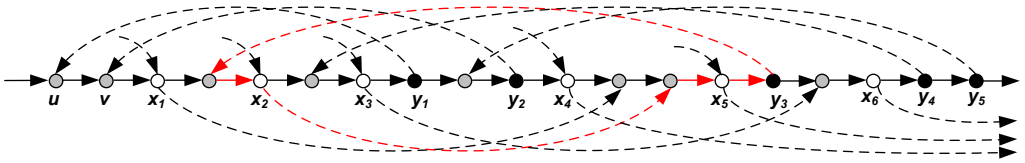


Fig. 3. An example for the formula $(x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee x_4 \vee x_5) \wedge (x_4 \vee x_5 \vee x_6) \wedge (\neg x_1 \vee x_5) \wedge (\neg x_2 \vee x_5) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_3 \vee x_6)$. The formula satisfiable problem corresponds to a feasible congestion- and loop-free update sequence within two time steps. The flow demand is one unit. The capacity of each link is two units. The delay of each link is one time unit. Black and white nodes need to be updated, while gray nodes do not need to be updated.

III. A Decision Algorithm

In this section, we design a decision algorithm to check the existence of a feasible update sequence. The detailed process is shown in Algorithm 1. We first explain the high level working of this algorithm. We construct a binary tree (all solid links) to perform node updates step by step, where the root in the top is the destination and the source node is located at the bottom of 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 a new path when the update is complete. Next we update one node whose dashed line points from the right to the left branch. We iteratively update the node 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 constraint (line 1). We use $V^\#$ to denote a set of nodes which have already been updated. The search process starts from the top to the bottom and adds the nodes one by one into $V^\#$ (lines 5-10). If $V^\#$ is not empty, we merge them 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 V'. By comparing the sum of link delays between 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 path candidate sets P_{v_i} (new path) and Q_{v_i} (old path) in order to update a node v_i whose outgoing dashed line points from one branch to the other. We select the path $p \in P_{v_i}$ with the minimum path delay given its delay is larger than in the old path q (lines 20-22). If the path p does not exist and $V'.cons$ cannot accommodate the old and the new flow simultaneously, a false variable is returned (lines 23-24). Otherwise, we update the node on the path p. The process is repeated until all the switches are updated. A detailed example is illustrated in Fig. 4.

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

***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. 5 to prove our theorem.

Case 1 (the update operation in line 18): As shown in Fig. 5(a), if the update of v violates the congestion-free

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: $\quad V^\# = \emptyset$
5: $\quad$ **while** $v.in.dashedline.source = \emptyset$ **do**
6: $\quad\quad$ **if** $v.in.solidline.source$ is not unique **then**
7: $\quad\quad\quad$ **break** //the loop terminates if more than one node point to v through solid line
8: $\quad\quad$ $u = v.in.solidline.source$ //the outgoing solid line of u points to v
9: $\quad\quad$ $V^\# = V^\# \cup \{u\}$
10: $\quad\quad$ $v = u$
11: $\quad$ **if** $V^\# \neq \emptyset$ **then**
12: $\quad\quad$ Merge all the nodes in $V^\#$ into one node, denoted as V'
13: $\quad\quad$ $V'.cons = \arg\min_{\langle u,v \rangle \in V^\#} C_{u,v}$ //indicate the bottleneck capacity
14: $\quad\quad$ $k = V'.in.dashedline.source$
15: $\quad\quad$ $p' = \langle k, k.out.solidline.destination, \ldots, V' \rangle$
16: $\quad\quad$ **if** k is active **and** $\sigma_{k,V'} \leq \phi(p')$ **and** $V'.cons < 2d$ **then**
17: $\quad\quad\quad$ **return false**//indicate that the update operation of node k will violate the congestion-free condition
18: $\quad\quad$ Update k at t
19: $\quad\quad$ $t = t + \sigma_{k,V'}$
20: $\quad$ $P_{v_i} = \{\langle v_i, v_j, \ldots, V' \rangle | \langle v_i, v_j \rangle \in P^{fin}\}$
21: $\quad$ $Q_{v_i} = \{\langle v_i, v_i.out.solidline.destination, \ldots, V' \rangle\}$
22: $\quad$ $p = \arg\min_{p \in P_{v_i}, q \in Q_{v_i} | \phi(p) \geq \phi(q)} \phi(p)$
23: $\quad$ **if** $p = \emptyset$ **and** $V'.cons < 2d$ **then**
24: $\quad\quad$ **return false**
25: $\quad$ **for** each node $z \in p$ **do**
26: $\quad\quad$ Update z at t
27: $\quad$ $t = t + \phi(p)$
28: **until** all the switches are updated
29: **return true**

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 a upward dashed link as any updates for downward links between v and V' will result in a forwarding loop at current routing configuration. We assume 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

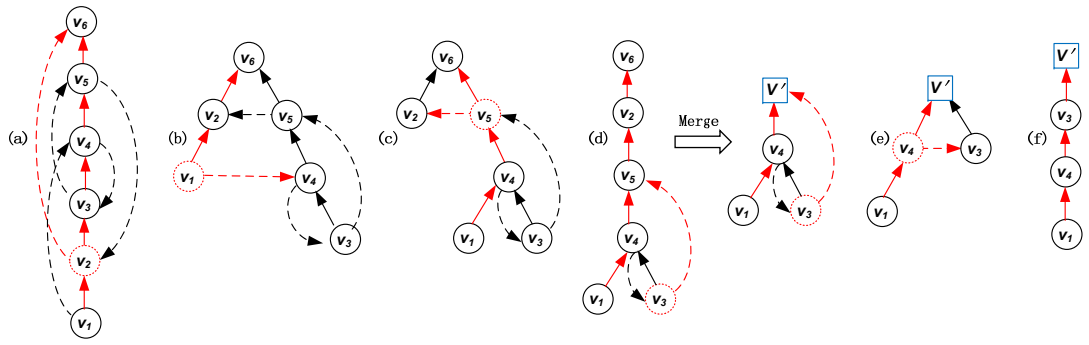


Fig. 4. An example for finding a feasible congestion- and loop-free update sequence. The flow demand and the link capacity are both one unit. The delay of each link is one time unit. The path of the dynamic flow between source v_1 and destination v_6 is depicted in as red solid line. The red dotted circle represents that the node is updated in the current time step.

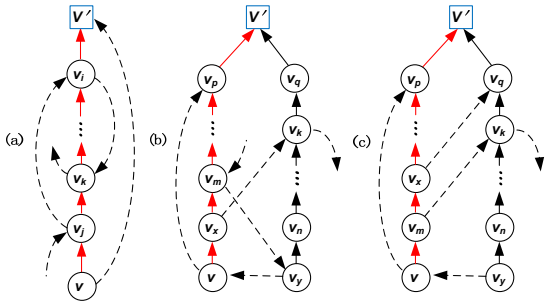


Fig. 5. Illustration of three network update scenarios shown in Algorithm 1.

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 current time step, it is infeasible at any time step.

Case 2 (the update operation in line 26): As shown in Fig. 5(b), suppose the update time of v_x is earlier than that of v_m, we have (8) holds from 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, the same as the case shown in Fig. 5(c). ■

IV. A Greedy Algorithm

We now design a greedy algorithm on the time-extended network to tackle 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, three forwarding loops will happen as shown in Fig. 2(b), which violates the loop-freedom condition. Assume we update v_1 and v_2 at t_0 as shown in Fig. 2(c), it 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. 6. The detailed calculation process will be explained in Algorithm 3. We can observe 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 constraint. However, a forwarding loop would 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 next time step t_2, we re-calculate the dependency relation set and 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 shown in Fig. 2(d).

At the beginning of Algorithm 2, we construct $\mathcal{V}^*$, which represents the set of switches that require to be updated (line 1). The initial time set $\mathcal{T}$ contains current time step t_0, history time steps $\{t_{-\sigma}, \cdots, t_{-1}\}$ and 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 will be discussed soon. If O_t contains a cycle, the algorithm terminates that indicates there does not exist a congestion-free update order (lines 7-8). Otherwise, we can update the switches according to the order in each dependency relation (lines 9-14). At the

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 set $\mathcal{V}^*$, which contains the switches required to be updated
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 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** $\emptyset$ //indicate congestion-free update is infeasible
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 exists 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: $\mathcal{T} = \mathcal{T} \cup \{t_i\}$ //add time step t_i to $\mathcal{T}$
18: Re-construct $\mathcal{G}^{\mathcal{T}}$ based on $\mathcal{T}$
19: **until** $O_t = \emptyset$

same time, we apply Algorithm 4 to check the possibility of forwarding loops (line 11). If the occurrence of the 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 reconstruct the time-extended network and enter into 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: $v = v_i(t).out.dashedline.destination$ //the outgoing dashed line of $v_i(t)$ points to v
5: $t' = t + \sigma_{v_i, v}$
6: $v' = v(t').in.solidline.source$ //the outgoing solid line of v' points to $v(t')$
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 requires to be updated. We start from a 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 switch 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 the 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.

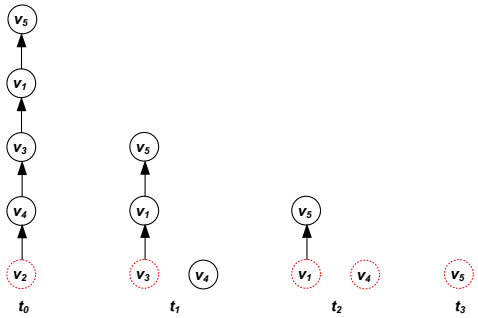


Fig. 6. Illustration of the resulting dependency relation set in the example of the time-extended network shown in Fig. 2(a). The red dotted circle represents that the switch is updated at current time step.

Taking Fig. 2(a) as an example, if we plan to update v_1 at t_0, we firstly need to find v_4 at t_1 in the time-extended network. Once $v_4(t_1)$ is found, we go back to its last hop $v_3(t_0)$ through the incoming solid edge of $v_4(t_1)$. If the link capacity $\langle v_4(t_1), v_5(t_2) \rangle$ cannot accommodate the two flows simultaneously, we establish a dependency relation $v_3 \rightarrow v_1$, which means that the update time of v_3 should be earlier than that of v_1, otherwise the congestion-free condition will be violated.

Algorithm 4 Checking the 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**//indicate a forwarding loop forms
6: **until** $\hat{v} = V^+$
7: **return false**

Algorithm 4 describes how to check 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 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, a false boolean variable is returned that indicates the update is feasible (line 7).

Based on the analysis above, we have the following:

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

V. Large-Scale Simulations

We conduct extensive simulations and experiments to evaluate our algorithms. In this section we report our performance

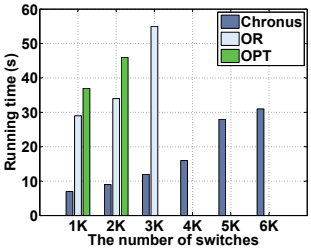


Fig. 7. Running time.

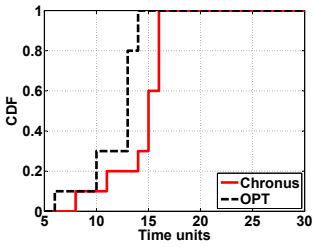


Fig. 8. Update time.

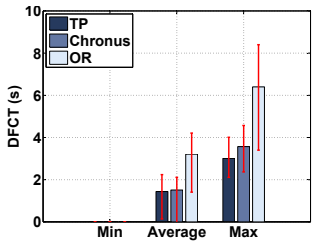


Fig. 9. Flow volume is 512 KB.

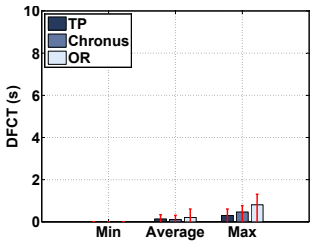


Fig. 10. Flow volume is 5 MB.

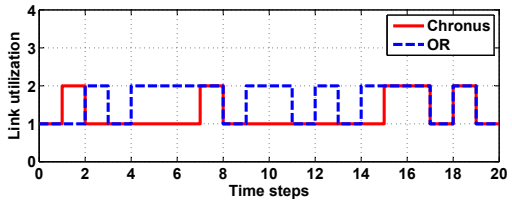


Fig. 11. The maximum link utilization at each time step in the time-extended network.

evaluation using large-scale simulations. In the next section we present our Mininet implementation results.

A. Setup

We use a large-scale linear network topology in our simulations. 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.

B. Benchmark Schemes

- **OR**: The order replacement updates [14] that minimize the number of rounds (i.e., the interactive between switches and the controller) and avoid the forwarding loops.
- **TP**: The two-phase updates [18] that we use VLAN ID 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. Thus they cannot be used to solve our problem defined in (3).

C. Basic Performance

We first investigate the percentage of congestion cases by comparing 500 different update instances in each run. In Fig. 12, 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. 13 shows the number of congested links comparison in the time-extended network. We can see that, as the number of switches increases, OR yields significantly more congested links compared to Chronus. Specifically, the number of congested links for OR and Chronus is 57 and 18, respectively, when the number of switches is 350, where Chronus can decrease the number of congested links by more than 65%.

Fig. 14 shows the CDFs of link utilization across all the links in the time-extended network for different schemes. For this simulation we fix the number of switches at 500. Intuitively, congestion happens when the utilization is larger than one. We can see that Chronus outperforms OR by around 20%. We now look at the rule space overhead of Chronus compared with TP. Fig. 11 shows the maximum link utilization within 20 time steps in the time-extended network. We define the congestion time step as the case that the maximum link utilization is larger than one. Essentially the number of congestion time steps measures the the duration of congestion. We can observe that there are in total 12 congestion time steps for OR, and only 5 for Chronus. Specifically, OR has 4 consecutive congestion time steps from 4 to 8, and Chronus only has 2 from 15 to 17.

The box plot in Fig. 15 shows the number of rules for Chronus and the blue solid point shows that 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 y-axis. We can see that the number of rules for TP increases more significantly than 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. 15. 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 SDN [8].

Finally we evaluate the running time and update time. The running time of Chronus, OR and OPT is illustrated in Fig. 7. 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 amount of their required time is orders of magnitude longer than Chronus. Chronus's running time is less than 60 seconds even when the number of switches is 6K. Fig. 8 shows the CDFs of update time when the number of switches is fixed at 40. We can

TABLE II
OPENFLOW MESSAGES USED IN CHRONUS.

Message	Parameter	Description
OFPBCT_OPEN_REQUEST	bundle_id	Open a bundle using bundle_id
OFPT_BUNDLE_ADD_MESSAGE	bundle_id, modi 1, ⋯, modi n	Add modifications to the bundle with bundle_id
OFPBCT_CLOSE_REQUEST	bundle_id	Close the bundle with bundle_id
OFPBCT_COMMIT_REQUEST	bundle_id, t	Commit the bundle with a specific time point t
OFPBCT_COMMIT_REPLY	bundle_id	The switch sends commit reply messages to the controller

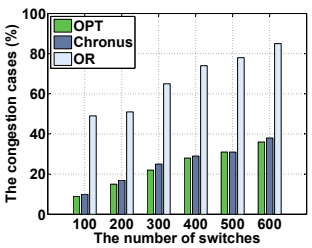


Fig. 12. % of congestion cases.

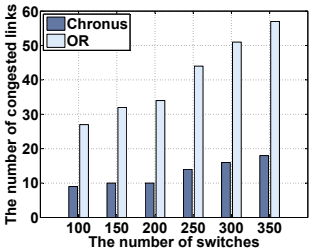


Fig. 13. # of congested links.

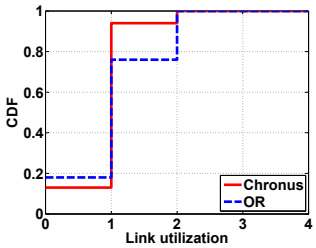


Fig. 14. Link utilization.

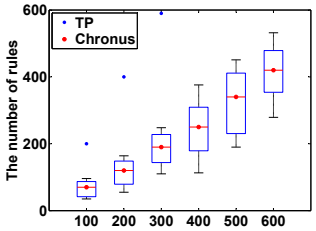


Fig. 15. # of forwarding rules.

see that most updates using Chronus finish within 15 seconds and OPT takes 13 seconds. The update time of Chronus can achieve near optimal compared to OPT.

VI. IMPLEMENTATION

In order to conduct experiments, complementing our simulations, we developed a prototype in Mininet 2.2.1 [10], a high fidelity network emulator for SDN. We use an Intel PC i5-2400 with quad-core processor. Mininet is configured to use OFSoftSwitch and Dpctl [1] as Openflow switches and the controller. The clock of all switches are synchronized by default in Mininet, and thus we do not require to run the Network Time Protocol (NTP). We use the `scheduled bundles` message [2] to guarantee accurate timing. Due to the single machine limitation of Mininet, we adopt a linear network topology with 10 switches. The forwarding rules are installed and updated via Dpctl API. We use `InPort` and `vlan_id` as a matching field to perform routing forwarding.

We now describe how to perform accurate timing for our algorithms. The procedure is shown in Algorithm 5. We first obtain a solution to MUTP using the greedy algorithm (line 1). Next we record the current clock as t_0 (lines 2). Then we sequentially examine every switch and update the corresponding forwarding rules. We first send a `OFPBCT_OPEN_REQUEST` message to open a bundle (line 4), and then send a sequence of `OFPT_BUNDLE_ADD_MESSAGE` messages to modify the rules (line 5). Modifications are stored into a temporary staging area without taking effect. Next we close the bundle (line 6). Finally when the bundle is committed, the modifications will be applied to the switch at a specific time point $t_0 + t_j + \theta$ (line 7). The threshold θ used here is to prevent that the predefined update time $t_0 + t_j$ becomes a past time point as a result of the bundle message installation delay. We set θ to be 2 seconds in our experiments. The openflow messages used in our algorithm are shown in Table II.

For completeness we explain the implementation of two-phase updates and order replacement updates in our implementation. The two-phase update relies on packet tagging. We use `vlan_id` in packet headers to index stages. In the first phase,

Algorithm 5 Performing the timed network updates

Input: The directed network $\mathcal{G}$; the initial path p^{init} and the final path p^{fin}; threshold θ;
Output: Update sequence of switch rules.
1: Apply Algorithm 2 and obtain solutions $\{v_i, t_j\}$.
2: Get the current clock t_0
3: **for** each $v_i \in \mathcal{V}$ **do**
4: The controller starts to open a bundle by sending `OFPBCT_OPEN_REQUEST` message, and receives a reply from the switch
5: The controller sends a sequence of `OFPT_BUNDLE_ADD_MESSAGE` messages to update the rules at switch v_i
6: The controller sends `OFPBCT_CLOSE_REQUEST` message to close the bundle
7: The controller sends `OFPBCT_COMMIT_REQUEST` message with `OFPBF_TIME` flag and `ofp_bundle_prop_time` is set to be $t_0 + t_j + \theta$, and then receives `OFPBCT_COMMIT_REPLY` message

new rules—whose matching fields use the new `vlan_id` that corresponds 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 where we stamp every incoming packet with the new `vlan_id`. At this point the new rules become functional, and the old rules are removed by the controller. We use the branch and bound method to obtain the optimal solution of the order replacement updates. When performing updates in each round, our algorithm sleeps for a while (using a random number from the data of [8]), so as to simulate the asynchronous nature of data plane.

Experiment Results: We measure the difference of flow completion time (DFCT) with and without the updates for TCP flows to assess the impact of different update schemes. The definition of DFCT is described as follows:

$$DFCT = FCT_{update} - FCT_{normal}$$

where FCT_{update} and FCT_{normal} represent the flow completion time with and without network updates. In our ex-

periments, the TCP window size is 85.3KB by default for both server and client side. We set the link capacity to 10 Mbps. The link delay is set to be an integer between 100ms to 2000ms. The maximum queue length for each switch is 100. We perform the same experiment at least 30 runs for both TP, Chronus and OR, and report the minimum, maximum and average of DFCT measured by `iperf` for the flow. Fig. 9 and Fig. 10 show DFCT of two typical flows: short flow and long-lived flow, where the flow volume is 512KB and 5MB respectively. We can observe that DFCT of the short flow is larger than DFCT of the long-lived flow, which indicates that the network updates have a more significant impact on short flows. Specifically, the average DFCT of OR, TP and Chrouns is 1.44s, 1.51s and 3.21s in Fig. 9. OR has more DFCT than TP and Chrouns. This is because the number of congested links using OR is greater than that using TP and Chronus, due to the asynchronous network updates, which results in more FCT_{update}. Chronus takes advantage of accurate timing to reduce congestion during network updates and thus has a better FCT_{update} than OR. Compared with TP, Chronus is able to offer almost equivalent performance without additional flow table space overhead in the switches. Finally we note that FCT_{normal} for TP is slightly longer than OR and Chronus due to packet tagging.

VII. Related Work

We review prior art on network updates in SDNs. Reitblatt et al. [18] 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. Ludwig et al. [14] aim to minimize the number of sequential controller interactions when transitioning from the initial to the final update stage. The authors prove that finding a shortest update sequence that avoids forwarding loops is NP-hard. The authors also introduce a notion of relaxed loop-freedom, which provides an interesting consistency-runtime tradeoff. Another work by Ludwig et al. [13] considers consistent network updates in the presence of middleboxes. However, these works do not consider transient congestion. SWAN [6] and zUpdate [12] 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 [8] employs dependency graphs to find a fast congestion-free update plan according to different runtime conditions of switches. Mizrahi et al. [15], [17] propose a time synchronization protocol between the controller and the data plane, which uses accurate timing to trigger network updates and reduce congestion. CCG [21] studies how to safely implement customizable consistency polices in order to minimize transition delay.

VIII. Conclusion

In this paper, we studied the problem of minimizing update time in timed SDNs. We proposed a decision algorithm to check the feasibility in polynomial time and a greedy algorithm to solve the problem. Evaluation results show that our solutions can reduce transient congestion and save flow table space.

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, 61502229, 61373130, 61672276), China NSF of Jiangsu Province (BK20141319), EU FP7 CROWN project (PIRSES-GA-2013-610524), U.S. NSF grants (1629746, 1564128, 1449860, 1461932, 1460971, 1439672), the Danish Villum project *ReNet*, and the program B for outstanding Ph.D. candidates of Nanjing University.

References

[1] Cpqd ofsoftswitch. https://github.com/CPqD/ofsoftswitch13.
[2] Openflow switch specification. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf.
[3] S. Brandt, K.-T. Forster, and R. Wattenhofer. On consistent migration of flows in sdns. In *INFOCOM*, pages 1–9, 2016.
[4] L. R. Ford and D. R. Fulkerson. Construct maximal dynamic flows from static flow. *Operation Research.*, 6:419–433, 1958.
[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
[6] 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.
[7] 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.
[8] 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.
[9] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, pages 113–126, 2012.
[10] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *HotNets*, page 19, 2010.
[11] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter. Traffic engineering with forward fault correction. In *SIGCOMM*, pages 527–538, 2014.
[12] 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.
[13] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid. Transiently secure network updates. In *SIGMETRICS*, pages 273–284, 2016.
[14] A. Ludwig, J. Marcinkowski, and S. Schmid. Scheduling loop-free network updates: It's good to relax! In *PODC*, pages 13–22, 2015.
[15] T. Mizrahi and Y. Moses. Software defined networks: It's about time. In *INFOCOM*, pages 1–9, 2016.
[16] T. Mizrahi, O. Rottenstreich, and Y. Moses. Timeflip: Scheduling network updates with timestamp-based TCAM ranges. In *INFOCOM*, pages 2551–2559, 2015.
[17] T. Mizrahi, E. Saat, and Y. Moses. Timed consistent network updates. In *SOSR*, pages 21:1–21:14, 2015.
[18] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, pages 323–334, 2012.
[19] 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.
[20] 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.
[21] 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.