Efficient Loop-Free Rerouting of Multiple SDN Flows

Arsany Basta, Andreas Blenk, Szymon Dudycz, Arne Ludwig, and Stefan Schmid

***Abstract*— Computer networks such as the Internet or datacenter networks have become a crucial infrastructure for many critical services. Accordingly, it is important that such networks preserve the correctness criteria, even during transitions from one correct configuration to a new correct configuration. This paper initiates the study of how to simultaneously update, i.e., reroute *multiple policies* (i.e., *flows*) in a software-defined network in a transiently consistent and efficient manner. In particular, we consider the problem of minimizing the number of controller–switch interactions, henceforth called *touches*, while preserving fundamental properties, in particular loop freedom, at any time. Indeed, we empirically show that the number of such interactions affects the resource consumption at the switches. Our main result is a negative one: we rigorously prove that jointly optimizing multiple route updates in a consistent and efficient manner is $\mathcal{NP}$-hard, already for two routing policies. However, we also present an efficient polynomial-time algorithm that, given a fixed number of correct update schedules for independent policies, computes an optimal global schedule with minimal touches. This algorithm applies to any per-flow independent consistency property, not only loop freedom.**

***Index Terms*— Communication networks, software-defined networks, scheduling algorithms.**

I. Introduction

THE availability and protection of computer networks such as the Internet or datacenter (cloud) networks, is becoming a concern of high priority. Already today, many individuals and organizations need to place great reliance on the services of computer networks. At the same time, the Internet core suffers from ossification, and has hardly evolved over the last decades. Despite the huge success of the Internet in the past, the increased dependability requirements raise concerns whether today's network protocols will be sufficient in the future [4].

Software-defined networking is an interesting new paradigm which promises to overcome some of the shortcomings of today's Internet architecture. A Software-Defined Network (SDN) outsources and consolidates the control over multiple data-plane elements to a centralized software program, enabling fast innovations while supporting formal verifiability through a simple match-action paradigm. Especially the traffic engineering flexibilities introduced by SDN [1], [21] as well as the potentially more scalable network virtualization [9], [25] have received much attention over the last years.

However, while a programmatic, logically centralized network control is appealing, exploiting the introduced flexibilities and operating an SDN in a consistent and efficient manner is non-trivial. In particular, an SDN still needs to be regarded as a distributed system [5], [23], [36], posing many challenges [7], [15], [26], [32], [34], [41], [47], [48]. Several of these challenges are due to the asynchronous communication channel between switches and controller, which exhibits non-negligible and varying delays [47], [54].

A fundamental problem which has recently received much attention regards the consistent update of network routes (also called *policies*) [15], [31], [34], [47], [53]: how to reroute a set of flows from their current paths to their respective new paths, without transiently violating certain properties such as loop-freedom or blackhole-freedom during the update? A particularly interesting approach to solve the update problem is to proceed *in rounds* [31], [34]: in each round, a "safe subset" of switches is updated, such that, independently of the times and order in which the updates of this round take effect, the network is always consistent. The scheme can be implemented as follows: after the switches of round t have confirmed the successful update (e.g., using acknowledgments [26]), the next subset of switches for round $t+1$ is scheduled. The appeal of this round-based approach is that it does not require packet tagging (which comes with overheads in terms of header space and also introduces challenges in the presence of middleboxes [56] or multiple controllers [7]) or additional TCAM entries [7], [47] (which is problematic given the fast table growth both in the Internet as well as in the highly virtualized datacenter [6]). Moreover, this approach also allows (parts of the) paths to become available sooner [34].

However, so far most research focused on devising network update schemes for a *single flow* (resp. *single policy*), that is, for scenarios where a single route [31], [47], or all (destination-based) routes to a single destination [34] need to be updated. Such a single flow update problem can be described as an "update pair" consisting of the old and the new path for that flow. However, especially in large and dynamic networks, it is likely that multiple routes have to be

Manuscript received January 1, 2017; revised September 11, 2017; accepted February 15, 2018; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor M. Andrews. This work was supported by the Danish Villum Foundation Project *ReNet*. *(Corresponding author: Stefan Schmid.)*

A. Basta and A. Blenk are with the Department of Electrical and Computer Engineering, Technical University of Munich, 80333 Munich, Germany.

S. Dudycz is with the Instytut Informatyki, University of Wroclaw, 1260 Wroclaw, Poland.

A. Ludwig is with the Department of Telecommunication Systems, Technical University of Berlin, 10587 Berlin, Germany.

S. Schmid is with the Faculty of Computer Science, University of Vienna, 1010 Vienna, Austria (e-mail: schmiste@gmail.com).

Digital Object Identifier 10.1109/TNET.2018.2810640

updated simultaneously [44]. For example, consider a Content Distribution Network where traffic is reassigned to servers in batches [17]. It is well-known that updating a switch and its datastructures comes at a certain resource cost [37], [54], and it is useful to batch updates [27].

A. Our Contributions

This paper initiates the study of how to jointly optimize the update schedule of, and reroute, multiple flows in a transiently consistent yet efficient manner.

In particular, we consider a most fundamental consistency requirement, *loop-freedom* [31], [34]: loops are known to harm the dependability of a network, due to packet drops, TCP packet reorderings, etc. Accordingly, there exist several RFCs and standards [49] on loop-free layer-2 spanning tree constructions [46], on avoiding microloops in MPLS [42], on loop-free IGP migration [14], etc.

Nevertheless, interestingly, today, we still do not have a a good understanding of the fundamental underlying *algorithmic* problem of how to update routes in a *transiently loop-freedom manner*. In particular, we in this paper study how to reroute multiple flows in a manner which minimizes the controller-switch interactions, henceforth called *touches*. We empirically show that such interactions consume precious computational resources, and can hence become a bottleneck.

Our main result is a negative one: we prove that the problem is computationally hard, already for three (by reduction from shortest common supersequence problems) resp. even two policies (by reduction from *Max-2SAT*), where each policy taken alone, could be updated in two rounds.

We complement this negative result by presenting an optimal polynomial-time algorithm to combine a *fixed* number of consistent update schedules computed for individual policies (e.g., using any existing algorithm, e.g., [31], [34]), into a global schedule guaranteeing a minimal number of touches. This algorithm is consistent and optimal not only for the loop-free property, but any property to be satisfied already for a single policy, as long as policies are independent (and e.g., do not compete for resources). However, we also point out the limitations of such efficient schedule compositions: we prove that for a non-constant number of policies, the problem becomes $\mathcal{NP}$-hard again.

B. Organization

The remainder of this paper is organized as follows. Section II introduces preliminaries and presents our formal model. Section III presents an empirical motivation. In Section IV, we give proofs for the computational hardness. Section V describes optimal polynomial-time algorithms under the assumption that only one switch is updated per round. After reviewing related work in Section VI, we conclude in Section VII.

II. Model

We are given a network which is controlled by a (logically) centralized software (the so-called controller) that communicates forwarding rule updates to the switches (the *nodes*), over an asynchronous but reliable channel. The controller is responsible to manage and update the routes taken by the network flows.

Throughout this paper, we assume that each policy or flow has its own set of forwarding rules stored at the different nodes. In particular, we do not restrict routing to be shortest path or destination based, but rather support the full routing flexibilities introduced by SDN, allowing for *arbitrary paths* as long as they are loop-free.

A. Multi-Round Update Scheduling

We consider the problem that the controller needs to simultaneously update k policies (i.e., the *routes* taken by the flows), from their old to their respective new paths. The k routing policies are *independent* in the sense that packets of different flows are forwarded according to different (and non-aggregated) rules; rules of different flows can hence be changed independently. In the following, we denote the set of to-be-updated nodes by U, and define $n = |U|$.

Each policy update is modelled as a pair $(\pi_1^{(i)}, \pi_2^{(i)})$, where $\pi_1^{(i)}$ is the *old route* (resp. path) and $\pi_2^{(i)}$ is the *new route* (resp. path) of the i-th policy, $i \in [1, k]$. Both $\pi_1^{(i)}$ and $\pi_2^{(i)}$ are simple directed paths, for any i. In other words, packets of policy i are initially forwarded, using the *old rules*, henceforth also called *old edges* (often indicated with *solid* edges in the figures), along $\pi_1^{(i)}$, and eventually they should be forwarded according to the new rules of $\pi_2^{(i)}$ (*dashed* edges). W.l.o.g. [31], we will assume that both the old as well as the new path of the i-th update have the same source s_i and the same destination d_i.

We require that during the update, packets should not be delayed or dropped at a node, nor redirected via the controller. In other words, whenever a packet arrives at a node, a matching forwarding rule should be present.

Let, for each node $v \in V$, $out_1^{(i)}(v)$ (resp. $in_1^{(i)}(v)$) denote the outgoing (resp. incoming) edge according to policy $\pi_1^{(i)}$, and $out_2^{(i)}(v)$ (resp. $in_2^{(i)}(v)$) denote the outgoing (resp. incoming) edge according to policy $\pi_2^{(i)}$. Moreover, let us extend these definitions for entire node sets S, i.e., $out_j^{(i)}(S) = \bigcup_{v \in S} out_j^{(i)}(v)$, for $j \in \{1, 2\}$, and analogously, for $in_j^{(i)}$.

Due to this asynchrony, of communication as well as of the datastructures updates at the switch itself [54], we require the controller to schedule and partition the updates in *rounds*, and send out simultaneous updates in the same round only to a "safe" subset of nodes: the correctness of the network configuration is always preserved independently of the order in which these updates take effect at the switches. Only after these updates have been confirmed (*ack*ed), the next subset is updated in the next round.

Let $U^{(i)}$ be the set of to-be-updated nodes for the i-th policy. We want to assign each update in $U^{(i)}$ to a round, such that the resulting schedule fulfills certain consistency properties. That is, we want to find an *update schedule* $U_1^{(i)}, U_2^{(i)}, \ldots$, i.e., a sequence of subsets $U_t^{(i)} \subseteq U^{(i)}$ where the subsets form a partition of $U^{(i)}$ (i.e., $U^{(i)} = U_1^{(i)} \uplus U_2^{(i)} \uplus \ldots \uplus U_{r_i}^{(i)}$), with the property that for any round t, given that the updates $U_{t'}^{(i)}$

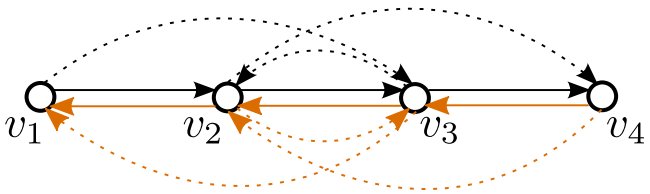


Fig. 1. Example with two concurrent policy updates: at the *top* update $(\pi_1^{(1)}, \pi_2^{(1)})$ in *black*, at the *bottom* update $(\pi_1^{(2)}, \pi_2^{(2)})$ in *orange*. The old policies ($\pi_1^{(1)}$ and $\pi_1^{(2)}$) are drawn using *solid* lines, the new policies ($\pi_2^{(1)}$ and $\pi_2^{(2)}$) using *dashed* lines. At least one node cannot install both updates simultaneously without creating a loop, and hence, needs two rounds of interactions (*touches*).

for $t' < t$ have been made, all updates $U_t^{(i)}$ can be performed "asynchronously" that is, in an arbitrary order, without violating some consistency property (mainly loop-freedom, but also others, see below): consistent paths will be maintained for any subset of updated nodes, independently of how long individual updates may take. We will refer to r_i as the number of update rounds (of the update schedule of the specific policy).

When reasoning about legal update schedules, it is hence useful to consider the following notion of *temporary forwarding graph*. For each policy update $(\pi_1^{(i)}, \pi_2^{(i)})$, let $U_{<t}^{(i)} = \bigcup_{j=1,\ldots,t-1} U_j$ denote the set of nodes affected by the i-th policy which have already been updated before round t, and let $U_{\leq t}^{(i)}$, $U_{>t}^{(i)}$ etc. be defined analogously. Since updates during round t occur asynchronously, an arbitrary subset of nodes $X \subseteq U_t^{(i)}$ may already have been updated while the nodes $\overline{X} = U_t^{(i)} \setminus X$ still use the old rules, resulting in a temporary forwarding graph $G_t(U^{(i)}, X, E_t)$ over nodes $U^{(i)}$ for this policy, where $E_t = out_1^{(i)}(U_{>t}^{(i)} \cup \overline{X}) \cup out_2^{(i)}(U_{<t}^{(i)} \cup X)$.

B. Transient Loop-Freedom

We require that the update schedule $U_1^{(i)}, U_2^{(i)}, \ldots, U_{r_i}^{(i)}$ fulfills the property that for all t, all policies i and for any $X \subseteq U_t^{(i)}$, $G_t(U^{(i)}, X, E_t)$ is loop-free. This property is also known as *strong loop-freedom* [31] in the literature: essentially, the temporary forwarding graph, at any point in time, is *topologically* loop-free.

While we will focus on the above notion of loop-freedom in this paper, we note that all our results also hold for the alternative notion of *relaxed loop-freedom* [31]. Relaxed loop-freedom only requires that the forwarding graph *induced by the source* s is loop-free: traffic from s will never be forwarded into a loop directly. There may however be some topological loops in other parts of the forwarding graph, which however are not connected to the source.

C. A First Example

Figure 1 shows an example of a concurrent route update of two routes resp. policies: at the *top*, the update pair $(\pi_1^{(1)}, \pi_2^{(1)})$ is shown in *black*, at the *bottom*, the update pair $(\pi_1^{(2)}, \pi_2^{(2)})$ in *orange*; the old policies ($\pi_1^{(1)}$ and $\pi_1^{(2)}$) are drawn using *solid* lines, the new policies ($\pi_2^{(1)}$ and $\pi_2^{(2)}$) using *dashed* lines. Let us first just have a look at the black policy update. The old policy traverses the nodes from v_1 to v_4 in numerical order, whereas the new policy traverses them in the following order: v_1, v_3, v_2, v_4. In order to guarantee a loop-free update, we need to make sure that the update on v_2 is installed before we send out the update for v_3; otherwise we risk a loop between the two nodes. Let us now focus on the orange policy update, in which the nodes are traversed in exactly the opposite order (in the old and the new policy), and thus, for the orange policy we need to update v_3 before we update v_2. In a concurrent update of these two policies, we are forced to choose one of the nodes (v_2 or v_3), and to send only one update (for a single policy) to break the cycle. This means that we need an extra interaction round (or touch) for this node, to install the update for the second policy in a later round. This leads to a possible update schedule of $U_1^{(1)} = \{v_1, v_2\}, U_2^{(1)} = \{v_3\}$ for the black policy and $U_1^{(2)} = \{v_4\}, U_2^{(2)} = \{v_3\}, U_3^{(2)} = \{v_2\}$ for the orange policy. The overall update schedule therefore then is: $U_1 = \{v_1, v_2, v_4\}, U_2 = \{v_3\}, U_3 = \{v_2\}$ showing that v_2 is touched twice.

D. Goal: Minimum Number of Touches

Interactions with a node come at a certain resource cost (see also our empirical motivation in Section III), and should be minimized. Accordingly, we are interested in schedules which jointly optimize the updates of multiple (namely k) policies, in such a manner that the number of interactions with nodes, henceforth also called *touches*, is minimized. That is, while when reasoning about consistency, we focused on individual update schedules, we now want to jointly optimize the possible individual r_i-round policy upate schedules $U^{(i)} = U_1^{(i)} \cup U_2^{(i)} \cup \ldots \cup U_{r_i}^{(i)}$, to form a global schedule $U = U_1 \cup U_2 \cup \ldots \cup U_R$, where U_i is the set of nodes which are updated in round i. The U_i sets do not have to be disjoint: switches may be touched multiple times.

Our objective is to minimize the total number of touches (summed over the entire update schedule), i.e., $\sum_i |U_i|$, where U_i denotes the set of nodes which are updated in round i. Observe that a solution to our problem always exists: we can simply concatenate the individual policy schedules. However, the resulting number of touches is high: each node is touched k times, once for each policy. It is also easy to see that it is not always possible to align the k policy updates in such a manner that each node is only touched once: in order to preserve consistency for the individual policy updates, in the global schedule U, nodes may occur repeatedly, in multiple rounds as seen in Figure 1.

E. A Second Example

Let us give an example. Figure 2 shows the construction of a worst case scenario, henceforth called *multi-touch lock*, requiring a maximal number of touches. Our example is for four concurrent policy updates. Each policy update consists of a source and a destination node on the outside, as well as the four nodes in the center of the figure. The order in which the nodes in the center are traversed in the new policy is exactly the reversed order in which they are traversed in the old policy. This leads to a chain of backward edges,

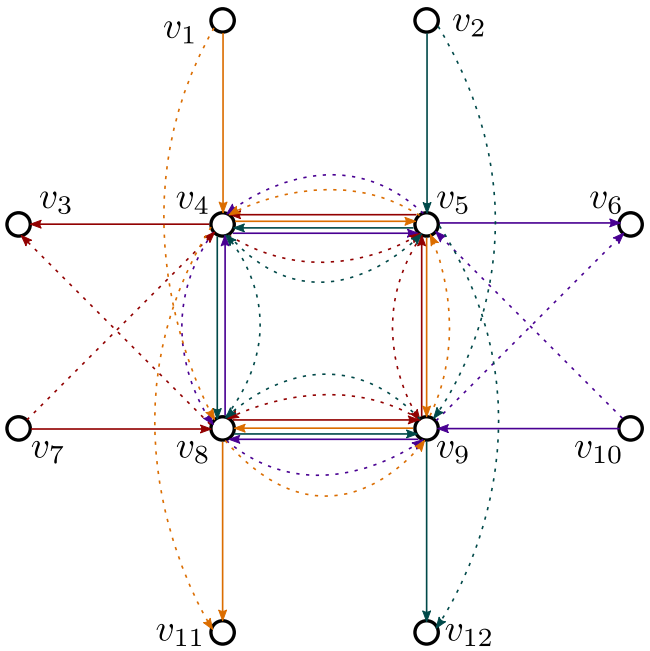


Fig. 2. Construction of a *multi-touch lock*. Four concurrent policy updates are shown in different colors, the old policies are shown using solid lines, and the new policies are shown using dashed lines of the same color.

e.g., the policy from v_1 to v_{11} traverses the nodes in the order v_4, v_5, v_9, v_8 whereas the nodes in the new policy are traversed as v_8, v_9, v_5, v_4. Hence, the nodes need to be updated one by one in a given order. Since the other policy updates have a similar structure, they also require a certain order of node updates. An update with a minimum number of touches always needs as many extra touches as there are different policies: thus, we need to touch four nodes twice.

F. Edge/Node Classification

We introduce the following useful edge (resp. node) classification. For each edge or equivalently node, and with respect to each policy update $(\pi_1^{(i)}, \pi_2^{(i)})$, we define a direction *forward* resp. *backward* with respect to a policy update $(\pi_1^{(i)}, \pi_2^{(i)})$, depending on whether the new edge (according to $\pi_2^{(i)}$) points in the same direction as the old policy (according to $\pi_1^{(i)}$), or in the opposite direction. That is, if for a given *new* edge (v_1, v_2) of policy i, v_1 also appeared before v_2 in the old route of policy i, then the edge (v_1, v_2) (and node v_1) is called *forward* with respect to policy i; otherwise edge (v_1, v_2) and node v_1 are called *backward*. As we will see, this distinction is useful as it is often safe to update any number of forward-pointing edges as they cannot introduce loops, while it can be harmful to update backward edges.

It is also useful to classify edges not only for update schedules from $\pi_1^{(i)}$ to $\pi_2^{(i)}$, but also "*in the reverse order*', from $\pi_2^{(i)}$ to $\pi_1^{(i)}$. Given this perspective, we can classify the old (*solid*) rules as *backward* or *forward* relative to the new ones (*dashed*): we just need to draw the new route as a straight path and see, if the old rule points forward or backward. Again, if for a given *old* edge (v_1, v_2) of policy i, v_1 also appears before v_2 in the *new* route of policy i, then the edge (v_1, v_2) (and node v_1) is called *forward* with respect to policy i; otherwise edge (v_1, v_2) and node v_1 are called *backward*.

Now observe that nodes and edges of a given policy i may be forward (resp. backward) *in the original update order* (i.e., new relative to old policy) but backward (resp. forward) *in the reverse update order* (i.e., old relative to new policy). Accordingly, we propose *two-letter* codes to describe the edges resp. nodes with respect to each policy update $(\pi_1^{(i)}, \pi_2^{(i)})$—the first letter will denote, whether the outgoing dashed edge of $\pi_2^{(i)}$ points forward (*F*) or backward (*B*) with respect to $\pi_1^{(i)}$. Similarly, the second letter will describe the old edge relative to the new path.

For example, consider the black policy in Figure 1. With respect to this policy, v_1 is an *FF* node: the dashed edge points forward w.r.t. the solid policy ($F\cdot$, where $\cdot$ is still unspecified, either F or B), but also the solid edge points forward w.r.t. the dashed policy ($\cdot F$, where $\cdot$ is still unspecified, either F or B). Similarly, v_2 is *FB* and v_3 is *BF*.

It is easy to see that in the first update round, we can safely update *any* subset of rules which are either *FF* or *FB*: a forwarding edge can never introduce a loop. By symmetry, a similar observation holds for the last round: consider an update $(\pi_1^{(i)}, \pi_2^{(i)})$. The last round of updating $(\pi_1^{(i)}, \pi_2^{(i)})$ can be seen as the first round of an update $(\pi_2^{(i)}, \pi_1^{(i)})$. Accordingly, in the last round, we can safely update any subset of rules which are either *BF* or *FF*, just like in the first round where we can update any *FB* or *FF*. (Nodes and edges which are only part of either the old or the new policy but not both can be updated trivially and are hence not considered explicitly here.)

In summary, for each node resp. each link and each policy, we define a 2-letter code. As a node can be involved in multiple policies, we can concatenate the 2-letter codes of the different policies to fully characterize the node. For example, in case of two policies, we will have nodes of the form $(F|B)^4$= {*FFFF,FFFB,* ...}. The first two letters denote the orientation regarding the first policy and the last two letters denote the orientation regarding the second policy. For example, in Figure 1, v_2 is *FB* in the black policy and *BF* in the orange policy, so overall it is *FBBF*.

III. Empirical Motivation

Before delving into the details of our flow rerouting algorithm, we provide some empirical motivation for the need to minimize the number of controller-switch interactions. In particular, in a small experiment, we verified that minimizing the number of touches might positively impact the performance of SDN switches: not only is the total control traffic reduced, but less touches also imply lower switch resource consumption (in terms of CPU). This in turn improves switch performance and throughput.

A. Methodology

We conduct measurements in a real SDN testbed. For this, we use the OpenFlow benchmark tool *perfbench* to measure the performance of an HP2920-24G hardware switch.

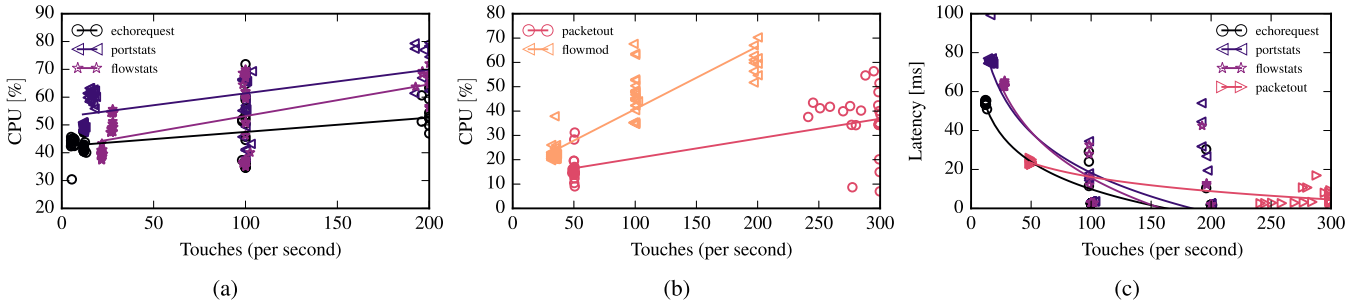


Fig. 3. Impact of touches (per second) on CPU consumption and latency for different OpenFlow message types. (a) Touches (per second) vs CPU consumption for EchoRequest, FlowStats, PortStats. (b) Touches (per second) vs CPU consumption for FlowMod and PacketOut. (c) Touches (per second) vs latency for EchoRequest, PortStats, FlowStats, PacketOut.

TABLE I

SWITCH OPENFLOW MESSAGE THROUGHPUT FOR SENDING INTERVAL 1 ms, 5 ms, AND 1000 ms

		OpenFlow Messages (per second)				
		EchoRequest	FlowMod	PortStats	FlowStats	PacketOut
Send Interval	1 $[ms]$	1240	1000	700	1230	3300
	5 $[ms]$	1510	1200	980	1230	2900
	1000 $[ms]$	1560	1200	1040	1560	3300

perfbench is designed to simultaneously benchmark the performance of the control plane as well as data plane of SDN switches [45]. Furthermore, it is able to generate constant message rates of the most relevant OpenFlow message types, e.g., FlowMod, PacketOut or FlowStatsRequest messages. While the OpenFlow control channel uses TCP, and hence the OpenFlow message aggregation per network packet depends on the specific TCP algorithm, *perfbench* can emulate different aggregation levels. For this, *perfbench* makes use of the TCP_NO_DELAY feature. While *perfbench* determines the number of OpenFlow messages that are written simultaneously into TCP's sending buffer, TCP_NO_DELAY leads to an immediate flush of the sending buffer. Of course, however, the sending behavior is also controlled by the receiver side, according to the receive window. Accordingly, by using the explained features, it is difficult to precisely generate a specific amount of touches. In order to indirectly generate varying touch rates, the sending interval between writing actions towards TCP's sending buffer can be set. For our measurements, we varied the sending interval between 1 ms and 1000 ms. This allows us to generate a different number of touches per second for varying OpenFlow messages, under a constant message rate (i.e., 500 messages per second). Note that a sending interval smaller than 5 ms leads to short switch instabilities, e.g., high CPU peaks, for long duration measurements. Although these switch instabilities last only for a few seconds, we consider 5 ms as the smallest send interval when we report on the improved CPU utilization.

B. Improved CPU Utilization

Fig. 3a shows the touches versus the CPU utilization for synchronous OpenFlow messages. For all message types, the smallest send interval (5 ms) produces the highest amount of touches, namely 250. The CPU utilization is increasing with the amount of touches from 40 % to nearly 70 %. This holds generally for all synchronous message types. For OpenFlow messages that do not request a response, i.e., asynchronous messages, Fig. 3b shows that the CPU consumption also increases with the amount of touches. For messages without a switch response (FlowMod, PacketOut), the CPU consumption is generally lower than for messages awaiting a response. This is due to the fact that the switch does not need to create a response for the messages. The CPU utilization over touches also varies depending on the message types. We generally note that the more touches we have, the higher the observed CPU utilization of SDN switches. Accordingly, minimizing the amount of touches can significantly save switch resources under the same OpenFlow message workload. It has to be noted however that, while a higher message aggregation, i.e., less touches, lowers the CPU consumption of switches, it leads to additional waiting times, for the OpenFlow messages. The latency values over touches is illustrated in Figure 3c. While less touches have the lowest CPU, they lead to the highest latency as aggregated messages have to wait longer to be processed.

C. Impact on Throughput

In general, we conclude that less touches lower the CPU consumption for SDN switches. This leaves room for switches to process higher workloads. Thus, we conducted a further measurement to study the switch throughput for sending intervals of 1 ms, 5 ms and 1000 ms. We increase the OpenFlow message rate until the system buffers overflow, i.e., the latency per message steadily increases. Table I shows that a higher send interval leads to higher amount of processable messages. Only PacketOut did not show the same effect. Here, the switch was not able to handle the data plane bursts in case of the 1000 [ms] sending interval. Accordingly, mechanisms and algorithms are needed that provide operators with the ability to control those trade-offs.

IV. COMPUTATIONAL HARDNESS

In this section, we initiate the study of consistent flow rerouting from an algorithmic perspective. In particular,

we present a negative result: optimizing the number of touches, when the number of rounds is constrained, is $\mathcal{NP}$-hard. We first leverage a connection to *Shortest Common Supersequence (SCS)* problems, to show that the problem is computationally hard already for three policies ($k = 3$), which individually (without optimizing the touches) could in principle be updated in two rounds ($r_i = 2 \ \forall i$). We then present our main technical result, a theorem stating that the problem is even hard for two policies ($k = 2$) which could be updated in two rounds each ($r_i = 2 \ \ \forall i$), by a reduction from Max-2SAT [28].

A. Hardness for 3 Policies

Interestingly, the problem of finding an update schedule which minimizes the node interactions in an n-node network is already computationally hard for $k = 3$ policies, which could in principle be updated consistently in an $R = 2$-round schedule. The remainder of this section is dedicated to the proof of the following theorem:

Theorem 1: Computing an optimal update schedule which minimizes the number of touches is $\mathcal{NP}$-hard for $k = 3$ policies. This holds even if each policy individually could be updated consistently in $R = 2$ rounds.

To prove the claim, we first establish a connection to the SCS problem, limited to instances in which each sequence has length 2 and each character appears in at most 3 sequences. We will refer to this problem by $SCS(2, 3)$.

Generally, the SCS problem is defined as follows. Given two sequences $X = (x_1, \ldots, x_{\ell_1})$ and $Y = (y_1, \ldots, y_{\ell_2})$, a sequence $s = (u_1, \ldots, u_{\ell_3})$ is a common supersequence of X and Y if s is a supersequence of both X and Y: X and Y can be derived from s by deleting some elements without changing the order of the remaining elements. A shortest common supersequence is a common supersequence of minimal length. For example, for $X = abcbdab$ and $Y = bdcaba$, $s = abdcabdab$ is the shortest supersequence. The SCS(2,3) problem variant where each sequence has length two and each character appears in at most 3 sequences was proven to be $\mathcal{NP}$-hard by Timkovskii [50].

In our reduction we want to encode sequences using only $k = 3$ policies, so that each policy will consist of sequentially connected graphs, each representing one sequence. As we want to optimize the number of touches, in the reduction, we can focus on schedules where in each round only one node is updated: while our model (and solution) is general in the sense that it allows to update multiple nodes in the same round, we can make this simplication for ease of presentation and without loss of generality. To see this, recall that our goal is to minimize the total number of touches, i.e., the sum over all rounds. Accordingly, we aim to update as many policies per node simultaneously as possible. However, we can easily distribute the updates to different nodes over time, without changing the total number of touches: if two nodes are updated in the same round, we can update one node in a first round and the second node in the next round without changing the consistency properties and number of touches: whether there are x touches in a 1-round schedule or 1 touch per round over an x-round schedule does not change the total number of touches.

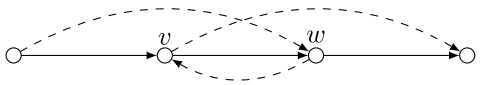


Fig. 4. Example configuration where node w must be updated after node v to avoid loops. A valid schedule is a supersequence of the sequence vw.

As an example, and to show the relationship to supersequence problems, let us consider the policy presented on Figure 4. In this instance, node w must be updated after node v: otherwise it will violate loop-freedom. Thus, a valid schedule is a supersequence of the sequence vw.

We will use this graph as a gadget representing sequences in the reduction, that is, for each sequence vw, we will create the graph in Figure 4 to force that v is updated before w. In the policy, we will connect these gadgets sequentially in an arbitrary order.

Because any node may appear at most once in each policy, we need to partition sequences into 3 sets, such that no character appears twice in one set. For some instances such a partition does not exist, and we will need the following lemma.

Lemma 1: Let S be an instance of $SCS(2, 3)$ and let $w = ab$ be any sequence in S. Let x be a new character (i.e., no sequence contains x) and let $S' = S \setminus \{w\} \cup \{ax, xb\}$. Then, S has a supersequence of length k iff S' has a supersequence of length $k + 1$.

Proof: First, let us assume that s is a supersequence of S of length k. Then, in s there is some character a, which is before some character b (there may be many occurrences of a and b, but there is at least one pair, such that a is before b). We add x immediately after a, and hence, this new sequence is a supersequence to all sequences in S and both ax and xb.

Now let us assume that s' is a supersequence of S' of length ℓ. We consider two cases:

- There is exactly one occurrence of x in s'. Then in s' there is an a before this x and a b after it, so s' is a supersequence to w. Therefore, if we remove x from s' we get a supersequence of S of length $\ell - 1$.
- There are at least two occurrences of x in s'. Then, we add a at the beginning of s' and remove all occurences of x. Such a sequence is a supersequence of ab, and in consequence of S, and has length at most $\ell - 1$. ■

We proceed to create the policies as follows. We will consider sequences in arbitrary order. Let $w = ab$ be any sequence. Then, if there is a policy without a and b we create a gadget for this sequence in this policy. Otherwise we create a new character x and two new sequences ax and xb. According to Lemma 1, after this change, we will be able to retrieve a shortest supersequence for the original problem.

In this situation we need to find policies where we can include the gadgets for ax and xb. We have created at most two gadgets with letter a, because there are at most three occurrences of a in total. Therefore there is at least one policy without a, and we create a gadget for ax in it. Similarly, there is at least one policy without b, hence, we create a gadget

for xb in it. Since, there was no policy without both a and b (as otherwise we would have created a gadget for ab in this policy), there is no policy with two repetitions of x (since we included the gadgets in two different policies). The length of the schedule is equal to the number of touches, and hence, this schedule is also a shortest supersequence.

B. Hardness for 2 Policies

We can strengthen our results further with a different reduction: we next provide a rigorous proof that the problem is already $\mathcal{NP}$-hard in n-node networks with $k = 2$ policies which could be consistently updated in $R = 2$ rounds. The remainder of this section is hence devoted to the proof of the following theorem.

Theorem 2: Computing an optimal update schedule which minimizes the number of touches is $\mathcal{NP}$-hard for $k = 2$ policies. This holds even if each policy individually could be updated consistently in $R = 2$ rounds.

1) Outline of Reduction: We prove the hardness by a reduction from Max-2SAT [28]. Recall that in Max-2SAT, the input is a formula in conjunctive normal form with two literals per clause, and the task is to determine the maximum number of clauses that can be simultaneously satisfied by an assignment. Unlike the decision problem 2SAT which is polynomial-time solvable, Max-2SAT is $\mathcal{NP}$-hard.

Let us first consider the problem of deciding whether the policies can be updated in 3 rounds using only n touches (so each node must be updated only once). An FB node cannot be the last updated node (as it is symmetric to updating a BF node in the first round, which violates loop-freedom), so nodes $FBFB$, $FBFF$ and $FFFB$ cannot be updated in the third round. They can always be updated in first round and there is no benefit of updating them in the second round (as they may be updated as first nodes during the second round); hence, we can assume that they will be updated in the first round. Similarly, we will assume that nodes $BFBF$, $BFFF$ and $FFBF$ are always updated in the third round. Because FB nodes cannot be updated in the third round and BF nodes cannot be updated in the first round, $FBBF$ and $BFFB$ nodes can only be updated in the second round. Finally, nodes $FFFF$ can be updated in any round, but because, similarly as before, there is no benefit in updating them in the second round, we will assume that they are updated in the first or the third round. Note that we only consider policies which are solvable within two rounds and hence, we do not need to classify nodes of type BB. No 2-round solvable policy update problem can include any BB nodes: such nodes cannot be updated neither in the first nor in the last (second) round.

Because we can always update FF and FB nodes in the first round, and FF and BF nodes in the third round, so to verify whether the schedule does not violate loop-freedom, it is enough to check, whether $FBBF$ and $BFFB$ nodes can be updated in the second round (that is that their update does not violate loop-freedom). See Table II for an overview.

We will use this classification in our reduction. For each variable, we will create an $FFFF$ node, and its value in the Max-2SAT formula will be decided based on whether the node is updated in the first or the last round. For each clause, we will create two nodes (one for each literal in the clause) and each of them will be a $BFFB$ node: they will always be updated in the second round. In what follows we will use x_i to denote both a variable and node for this variable, and for a clause $C_j = l \vee k$ we will use y_j^l and y_j^k to denote nodes created for this variable.

TABLE II

UPDATEABLE NODES PER ROUND FOR A 3-ROUND SCHEDULE. $FFFF$ NODES CAN BE UPDATED EITHER IN THE FIRST OR IN THE THIRD ROUND. NO BB NODES ARE POSSIBLE IN POLICY UPDATES SOLVABLE WITHIN 2 ROUNDS, AND HENCE, WE DO NOT NEED TO CONSIDER THEM

Round		
1	2	3
$FBFB$	$FBBF$	$BFBF$
$FBFF$	$BFFB$	$BFFF$
$FFFB$		$FFBF$
$FFFF$		$FFFF$

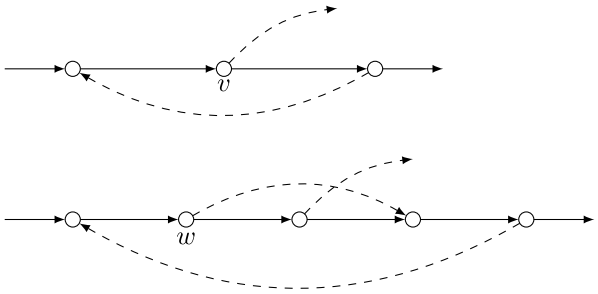


Fig. 5. Examples of $FFFF$ nodes which must be updated in either first or third round.

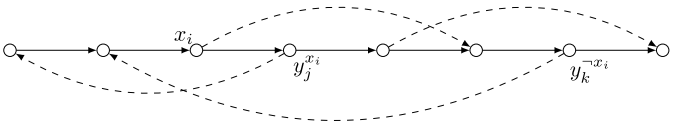


Fig. 6. Outline of a gadget for variables.

Let us consider the (partial) graphs in Figure 5. Let us assume that nodes v and w in both graphs are of type $FFFF$ and that the backward node in each graph is of type $BFFB$. Then, in the graph on the top, v must be updated before the backward edge (in the first round), and in the graph on the bottom, w must be updated after the backward edge (in the third round).

We will combine these two graphs to create a gadget for each variable. Let us consider a variable x_i, and two clauses: C_j, which contains the literal x_i, and C_k, which contains the literal $\neg x_i$. Then, we will create a gadget as shown in Figure 6. We will make x_i an $FFFF$ node, and both $y_j^{x_i}$ and $y_k^{\neg x_i}$ $BFFB$ nodes. If we update the node for x_i in the first round, then we can update $y_j^{x_i}$, and if we update x_i in the third round, then we can update $y_k^{\neg x_i}$.

For each variable, we will create such gadgets in both policies, and the node corresponding to the variable will be the same (physical switch) in both policies; therefore, either

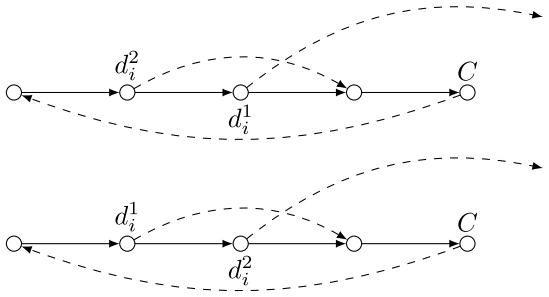


Fig. 7. Gadget for updating clauses in one of the policies.

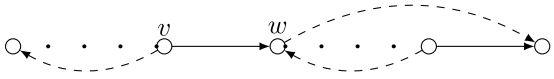


Fig. 8. Construction to make v a BF node.

it will be updated in the first round in both policies, or in the third round.

We will use the version of Max-2SAT, in which each variable occurs in at most three clauses. Therefore, we will split the clauses, such that in a variable gadget in one policy there will be two clauses, and in the other policy one clause. Also, the nodes for each clause must be in different policies (because of the clause gadget, which we will describe in Section IV-B2). We will describe how to split clause nodes into policies in Section IV-B7.

2) Clause Gadget: Since in the Max-2SAT problem it is enough that one literal in a clause is satisfied, we will need to be able to update one of the clause nodes independently of the variable nodes. To achieve this, we will use the gadget presented in Figure 7, which will be a part of the variable gadget. We will denote vertices created for clause C_i as d_i^1 and d_i^2. We will make them $FFFF$ nodes, and hence, they can be updated in either the first or the third round. If d_i^1 gets updated in the first round, then it enables the clause node in the first policy to be updated, but then, even if d_i^2 is updated, in the second policy, the clause node has to be updated using the variable gadget. Similarly if we update d_i^2 in the first round, and w in the third round, we can then update the clause node in the second policy in the second round.

Because this gadget shares nodes between policies, clause nodes must be in different policies.

3) Specifying Node Type: In order to introduce dependencies leading to the combinatorial complexity of the update, we want to assign certain types to the nodes in the gadget. For example, we know that in order to preserve loop-freedom, backward edges cannot be updated in the first round, and hence, first forward edges need to take effect which break a loop: we need forward nodes, when looking from the point of view of the new policy (that is, we want to guarantee that its second letter in the classification is F). As an example, in Figure 8, v is a backward node which we want to make a BF node. To do this, we will add a new node just after v, which we will denote as w, and create an edge from the end of the gadget to w. Then, we will create a new node after the gadget and create an edge from w to this new node.

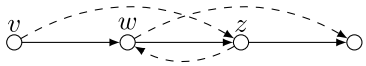


Fig. 9. Gadget for creating nodes of required type.

The construction is depicted in Figure 8. Node w is visited in the new policy after the whole gadget has been visited (so also after v), and therefore edge (v, w) is forward when looking from the point of view of the new policy. Node w is now an FB node, so it could possibly allow to update some $BFFB$ nodes, if updated in the first round, therefore we will make w a BF node in the other policy to force it being updated in the second round.

4) Nodes of Required Type: For some nodes in one policy there is a required type in the other policy (e.g. a clause node, which has to serve as an FB node). To create such nodes we will use the gadget shown in Figure 9. In this gadget v is an FF node, w is a FB node and z is a BF node.

5) Complete Gadget for Variable: In Figure 10 we present the gadget for variable x_i, and its two clauses C_j, containing literal x_i, and C_k, containing literal $\neg x_i$. In this gadget we included gadgets for both clauses. The essential edges of the gadget (presented in Figure 6) are drawn in loosely dashed black, edges of clause gadgets are drawn in loosely dashed grey, edges added to change the node type (described in Section IV-B3) are drawn in densely dashed grey and the other edges added to connect the graph are drawn in densely dashed black. We will set the type of all densely dashed black and grey edges to type BF in the other policy, so, unless 2 touches will be used for them, they will be updated in the second or third round, and therefore any update schedule must assume that they will be updated after clause vertices.

6) Transforming a Max-2SAT Formula: In this section we will show how to transform a Max-2SAT formula, so that each variable appears in at most three clauses. Let ϕ be a Max-2SAT formula with m clauses. Then for each variable x in ϕ, which has p_x positive occurences and n_x negative occurences, we will create variables $x_1, x_2, \ldots, x_{p_x}$, $\overline{x}_1$, $\overline{x}_2$, $\overline{x}_{n_x}$. We will use those variables to substitute occurences of x in ϕ (we will substitute literal $\neg x$ with variable $\overline{x}_i$, hence, we want $\overline{x}_i$ to be true iff x is false). For each $i \in \{1, \ldots, p_x\}$ we will create variables $t_1^i, t_2^i, \ldots t_{n_x}^i$. Similarly for each $i \in \{1, \ldots, n_x\}$ we will create variables $\overline{t}_1^i, \overline{t}_2^i, \ldots \overline{t}_{p_x}^i$.

Now for each $i \in \{1, \ldots, p_x\}$ we will create clauses $x_i \implies t_1^i \implies \ldots \implies t_{p_x}^i$ ($p \implies q$ in 2SAT can be written as $\neg p \vee q$). We also create similar clauses for each $\overline{x}_i$. Then for each $i \in \{1, \ldots, p_x\}$ and $j \in \{1, \ldots, n_x\}$ we create a clause $\neg t_j^i \vee \neg \overline{t}_i^j$. If all these clauses are satisfied, they guarantee that for each $i \in \{1, \ldots, p_x\}$ and $j \in \{1, \ldots, n_x\}$, x_i and $\overline{x}_j$ cannot be both true. However, note that these clauses do not guarantee that all variables for x have the same value, that is, there may be some i, j such that x_i is true and x_j is false.

For each variable in ϕ, we create $p_x(2(p_x - 1) + n_x) + n_x(2(n_x - 1) + p_x)$ variables; clearly, this reduction is polynomial. We will denote the resulting formula by ϕ' and we will denote the number of clauses of ϕ' by m'. Now to finish the reduction we will prove the following theorem.

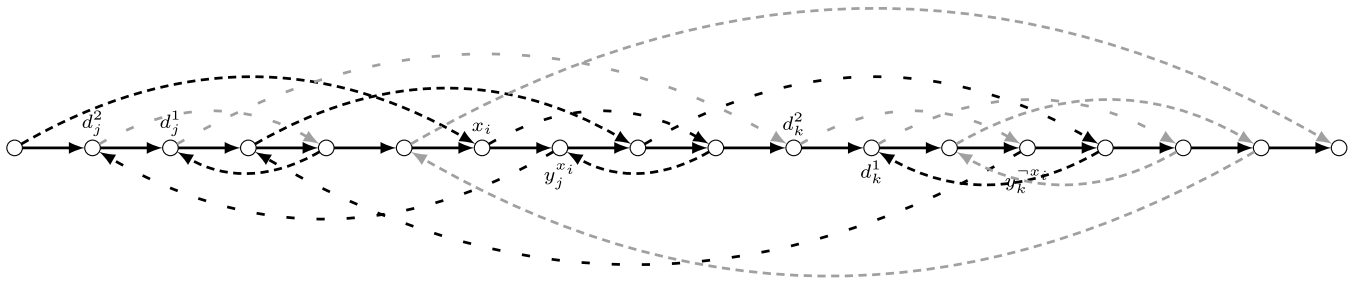


Fig. 10. Complete gadget for a variable. The essential edges of the gadget (presented in Figure 6) are drawn in *loosely dashed black*, edges of clause gadgets are drawn in *loosely dashed grey*, edges added to change the node type (described in Section IV-B3) are drawn in *densely dashed grey* and the other edges added to connect the graph are drawn in *densely dashed black*.

Lemma 2: There is an assignment satisfying $m - k$ clauses of ϕ if and only if there is an assignment satisfying $m' - k$ clauses of ϕ'.

Proof: First, let us assume that there is an assignment that satisfies $m - k$ clauses of ϕ. Then, we will set $x_i = x$, $t_j^i = x$, $\overline{x}_i = 1 - x$ and $\overline{t}_j^i = 1 - x$. Then, all new clauses added to ϕ' are satisfied, so exactly k clauses are unsatisfied.

Now let us assume that there is an assignment that satisfies $m' - k$ clauses of ϕ'. We will prove that there is an assignment which satisfies at least $m - k$ clauses of ϕ. For each variable x let $P_x = \{i \in \{1, \ldots, p_x\} \quad |x_i = 1\}$ and $N_x = \{i \in \{1, \ldots, n_x\} \quad |\overline{x}_i = 1\}$. Then we set x to be 1, if $|P_x| > |N_x|$, and to 0 otherwise (thus we choose the value of x based on the majority voting of variables x_i and $\overline{x}_i$).

Obviously in such an assignment of variables in ϕ there may be some clauses which are satisfied in ϕ', but not in ϕ. Let S_x be N_x, if $|P_x| > |N_x|$, and P_x otherwise (so S_x is the set of those literals of x, which were true in ϕ', but are false in ϕ) and let Q_x be $P_x \cup N_x \setminus S_x$. Let us assume w.l.o.g. that $Q_x = P_x$ and $S_x = N_x$. Each of the literals in S_x appears in exactly one clause of ϕ, so there are at most $|S_x|$ clauses in ϕ which were satisfied by literals in S_x in ϕ'. But for each $\overline{x}_j$ in S_x and x_i in Q_x there is a clause $\neg t_j^i \vee \neg \overline{t}_i^j$. Therefore there are three possibilities:

1) Some implication in $x_i \implies t_1^i \implies \ldots \implies t_j^i$ is unsatisfied.
2) Some implication in $\overline{x}_j \implies \overline{t}_1^j \implies \ldots \implies \overline{t}_i^j$ is unsatisfied.
3) Clause $\neg t_j^i \vee \neg \overline{t}_i^j$ is unsatisfied.

If for all literals in Q_x, Case 1 holds, then there are $|Q_x| > |S_x|$ unsatisfied clauses. Similarly if Case 2 holds for all literals in S_x, then there are $|S_x|$ unsatisfied clauses. Otherwise let $l = \max\{j \quad \overline{x}_j \in S_x\}$. Then let $x_i \in Q_i$ be such that $x_i = t_l^i$. Let k be number of literals in $S(x)$ for which Case 2 holds. Then for other $|S(x)| - k$ literals in $S(x)$ and x_i, Case 3 must hold. Therefore there are at least $k + |S(x)| - k = |S(x)|$ unsatisfied clauses.

None of these clauses is in ϕ and the sets of these clauses for different variables are disjoint, and hence, there are at least $\sum_x |S(x)|$ clauses which are unsatisfied in ϕ', but do not appear in ϕ. On the other hand by assigning the value of x based on majority voting we unsatisfy at most $S(x)$ clauses, so in total there are at most $\sum_x |S(x)|$ clauses which are unsatisfied in ϕ, but are satisfied in ϕ'. Therefore the number of unsatisfied clauses in ϕ is at most k. ■

7) Splitting Clauses Into Policies: Recall that for each variable in one gadget there may be at most two clause nodes, one containing the positive literal and one containing the negative literal. Also nodes for a clause must be in different policies, so that we are able to construct the clause gadget. In this section we will show how to split nodes for clauses into two policies to satisfy those requirements.

We will assume that the Max-2SAT formula was created using the reduction described in Section IV-B6. To split the clauses we consider the variables of ϕ in any order. Then, each variable x_i is in two clauses, once as a positive literal in the clause from ϕ, which we may be forced to put in one of the policies, if the other variable from this clause has already been processed. The other occurence is as a negative literal in implication $x \implies t_1^i$, which we put in any of the policies. Then each t_j^i appears in 3 clauses (except for $j = p_x$). As a positive literal it appears only in the implication $t_{j-1}^i \implies t_j^i$, which we assign to the other policy than t_j^i. As a negative literal, it appears in the clause $\neg t_j^i \vee t_k^l$, for some l, k; if t_k^l has already been processed, we may be forced to put it in one of the policies, and then to the other policy to which we assign clause $t_j^i \implies t_{j+1}^i$: this is always possible, as t_{j+1}^i has not been processed yet.

8) Proof of Reduction: We will start by proving that if the multiple policies instance can be updated using $n + k$ touches then at least $m - k$ clauses of the Max-2SAT formula can be satisfied. In what follows variable gadget nodes will be all nodes in the gadget except for those that are in the clause gadget (in terms of Figure 10 these are all nodes except those with an outgoing loosely dashed grey edge). Then let X_1 be the set of those variables, such that all nodes in their variable gadgets are updated using one touch. Also, let X_2 be the set of those variables for which there is a node in their variable gadgets which were updated twice. Also let D be the set of those clauses, such that there is some node in their gadgets, which used two touches. Because clause gadget nodes and variable gadget nodes are disjoint, $|D| + |X_2| \leq k$.

Then, we set each variable in X_1 to be 1, if its node is updated in the first round, or to 0, if its node is updated in the third round. Each variable x in X_2 appears in at most 3 clauses, therefore we can choose the assignment which does

not satisfy at most one of these clauses. In such an assignment a clause C can be unsatisfied if:

1) $C \in D$
2) One of the nodes of C was updated using the clause gadget, and the other using an extra touch in some variable gadget.

Now suppose that there is an unsatisfied clause C for which none of those cases hold. Then, both variables of C are in X_1. One of the nodes of C can be updated in the second round using the clause gadget. Then the other node, as we have seen in Section IV-B2, cannot be updated using the clause gadget. And because of our case assumption, it can also not be updated using a variable node. Since all of the other edges are updated in the same or a later round, such an update schedule would violate loop-freedom.

Therefore in the Max-2SAT formula, there are at most $|D|$ clauses for Case 1 and $|X_2|$ clauses for Case 2, so together there are at most $|D| + |X_2| \leq k$ unsatisfied clauses.

Now we will prove that if $m - k$ clauses of the Max-2SAT formula can be satisfied, then there exists a schedule that uses $n + k$ touches. For each variable we will update its node in the first round, if it is set to 1, or in the third round, if it set to 0. For each clause we will update one of its clause gadget nodes, which will allow us to update a clause node corresponding to the false literal (in case of satisfied clauses there is at most one such node, and in case of unsatisfied clauses we arbitrarily choose one of two nodes). Then, both nodes of the satisfied clauses and one node of the unsatisfied clauses can be updated in the second round. The nodes of the unsatisfied clauses, which cannot be updated in the second round, will be updated in the third round; we will need two touches to achieve this. The remaining nodes will be updated according to their type, using one touch.

All nodes of type $FBBF$ in the variable gadget can be updated in the second round, as the packets that traverse them would be forwarded to the end of the variable gadget, and all the other nodes can always be updated in the first or third round respectively; therefore,the schedule is correct. Since we use extra touches only for unsatisfied clauses (one extra touch for each clause), we have $n + k$ touches in our schedule.

V. Composing Individual Schedules

Most prior work on the network update problem, not only for loop-freedom [31] but also for other transient properties such as blackhole freedom [34], focused on the consistent update of *individual flows*. In this section, we show how to efficiently merge (or *compose*) correct update schedules of individual flows, into a global schedule with minimal touches. Concretely, the algorithm presented in the following, can serve as a generic post-processor, combining the outputs of the existing single-flow scheduling algorithms into an optimal global schedule for multiple flows, while preserving any per-policy consistency criterion (beyond loop-freedom, e.g., blackhole freedom or waypoint enforcement).

The algorithm comes with two requirements: first, for our algorithm to work, the different flows need to be independent and should not interfere, e.g., on resources such as bandwidth. Second, in order for our algorithm to complete in polynomial time, the number of policies must be fixed (i.e., *constant*). We will show later in this section that for non-constant number of policies, the schedule composition problem becomes computationally hard.

A. Efficient Algorithm for a Fixed Number of Flows

Let us first assume that we are given the order of to be updated nodes in their respective policies, and without loss of generality, only one node is updated per round (but ideally multiple policies on this node). As discussed above, updating multiple nodes in the same round does not help reduce the number of touches. Therefore we will assume that in the joint schedule also only one node is updated in each round.

Our goal is to construct a joint schedule that minimizes the number of touches without any constraints on number of rounds. For instance, a simple way to compute these individual correct update schedules *in case of loop-freedom*, is to update switches one by one, from the destination to the source. This creates a total order of the switches and guarantees loop-freedom.

The problem of how to optimally merge correct schedules is a special case of shortest common supersequence problem. Here, each node corresponds to a letter in the alphabet, and each policy order corresponds to an input sequence. Then the requirement that in the joint schedule there is an update of node v before an update of node u, is equivalent to the requirement that in supersequence w there is an occurence of character v before some occurence of character u. In comparison to the general SCS problem, in our problem, in each policy order, each node appears at most once: in the SCS input sequences each character is unique.

SCS is known to have a polynomial time algorithm if the number of input sequences is constant, and to be $\mathcal{NP}$-hard if the number of input sequences is not constant [35], [50]. Jiang and Li [22] proved that unless $\mathcal{P} = \mathcal{NP}$, SCS cannot be approximated with a constant factor, and provided an algorithm that on average returns a common supersequence of length $OPT + \mathcal{O}(OPT^{0.707})$. In the remainder of this section we will present the polynomial time algorithm for SCS with a constant number of input sequences and a proof of $\mathcal{NP}$-hardness of our problem.

The algorithm for solving SCS is dynamic. The idea of the algorithm is to compute the shortest common supersequence for all prefixes of input sequences. Let T be the m-dimensional matrix, one dimension per policy, and where each dimension lists different prefix lengths. The matrix stores the lengths of the shortest common supersequences of prefixes, i.e., $T[v_1, v_2, \ldots, v_m]$ stores the length of the shortest common supersequence of $v_1, v_2, \ldots, v_m$, where each v_i is a prefix of w_i. For two sets of sequences $A = \{v_1, \ldots, v_m\}$ and $B = \{u_1, \ldots, u_k\}$, we will also use $T[A]$ to denote $T[v_1, \ldots, v_m]$ and $T[A, B]$ to denote $T[v_1, \ldots, v_m, u_1, \ldots, u_k]$. Let $S_c(v_1, \ldots, v_m)$ be a set of those sequences from $v_1, \ldots, v_m$ that end with character c and let $Q_c(v_1, \ldots, v_m)$ be a set of those sequences that end with a character other than c. For a sequence v, let $v[-1]$ denote its last element, let $\tilde{v}$ be v without its last element, and let $\tilde{S} = \{\tilde{v} \;\; v \in S\}$.

To compute the shortest common supersequence of $v_1, \ldots, v_m$, we have to decide on the last letter in the supersequence. Possible candidates are the last letters of any $v_1, \ldots, v_m$, hence, for each of them we compute the set of sequences that end with the same letter and remove it. All the other sequences remain the same. Therefore the formula to compute the length of the shortest common supersequence is as follows: $T[v_1, \ldots, v_m] = 1 + \min_{i \in \{1,\ldots,m\}} T[\tilde{S}_{v_i[-1]}(v_1, \ldots, v_m), \quad Q_{v_i[-1]}(v_1, \ldots, v_m)]$

Each sequence has a length of at most n, so we have to compute n^m values in the array and to compute each of them, we need $\mathcal{O}(m)$ time. Therefore the space complexity is $\mathcal{O}(n^m)$ and the time complexity is $\mathcal{O}(mn^m)$, which, as long as number of sequences (i.e., policies) is constant, is polynomial.

To clarify the algorithm, we provide a simple example on its procedure. Assume $v_1 = ab, v_2 = bc$. Obviously the shortest common supersequence is abc and has length 3.

$$T[ab, bc] = 1 + \min \begin{cases} T[\tilde{S}_b, Q_b] = T[a, bc] \\ T[\tilde{S}_c, Q_c] = T[b, ab] \end{cases} \quad (1)$$

$$T[b, ab] = 1 + \min \begin{cases} T[\tilde{S}_b, Q_b] = T[a] \\ T[\tilde{S}_b, Q_b] = T[a] \end{cases} \quad (2)$$

$$T[a] = 1 \quad (3)$$

In Eq. (1), we look for the minimum value of remaining vs after fixing the last character (b and c). We omit the details for $T[a, bc]$ (fixing b) which has a length of 4, and only show the path to the minimum solution. In Eq. (2) both sequences end with b, hence we do only have one character remaining. This leads to the correct solution of abc with length 3.

In summary:

Theorem 3: A fixed number of feasible individual update schedules can be merged optimally, minimizing the number of touches, in polynomial time.

To achieve a global order (as an input to our algorithm), we could for example define a canonic order on the nodes updated in the same round. As a heuristic, one could also generate a small number of random (but correct) schedules, and test with our algorithm, which one provides the overall best performance, before issuing the update requests to the nodes. Moreover, in order to minimize the number of rounds, the result of the optimal algorithm can in turn be post-processed by greedily grouping individual switch updates into rounds. Note that this however is only a heuristic: minimizing the number of rounds is NP-hard even for a given number of touches [31].

B. NP-Hardness for Many Policies

While the merging scheme is interesting, we can only achieve a polynomial runtime for a constant number of policies: the computational tractability does not extend to scenarios with arbitrarily many policies, even in settings where one node is updated per round. We will adapt the proof by Timkovskii [50] and present a polynomial-time reduction from the *Directed Feedback Vertex Set Problem (DFVS)*. The DFVS problem is defined over a directed graph $G = (V, E)$, and asks for a minimum size set of vertices whose removal leaves a graph without cycles: each feedback vertex set contains at least one vertex of any cycle in the graph. In a nutshell, the idea of the reduction is the following: Given the input graph $G = (V, E)$ to DFVS, for each edge (u, v), we create a policy enforcing an order $u \prec v$, i.e., $|E|$ policies in total. We will show that the nodes in a feedback set need to be touched twice, to guarantee that any order of nodes u, v can be updated. Any nodes not in the feedback set can be ordered, since they will not form a loop, and thus, updated one by one with a single touch. Minimizing the cardinality of the feedback set will therefore minimize the number of touches.

Theorem 4: The problem of finding a consistent update schedule minimizing the number of touches is $\mathcal{NP}$-hard in general.

Proof: Given the DFVS graph $G = (V, E)$, we create for each edge $e = (u, v) \in E$ a policy enforcing an order $u \prec v$, and prove the following: There is a directed feedback vertex set in G of size k, if and only if there is a joint schedule for a network update instance using $|V| + k$ touches: each node in the feedback set needs to be touched exactly twice, and all other nodes once.

Firstly let us assume that there is a directed feedback vertex set S of size k in G. Given the directed and loop-free resulting graph, the vertices in $V \setminus S$ can be ordered topologically. Let us consider a schedule σ in which we first update nodes in S, then those in $V \setminus S$ in the topological order, and finally those in S again. Obviously σ has length $|V| + k$.

We claim that σ is a correct solution for the network update problem. Having created one policy for each edge (u, v), we need to show that for each edge there is a corresponding subsequence $u \prec v$ in the correct schedule. There are 3 sub-cases:

1) If $u, v \in S$ then u is updated the first time when nodes in S are updated, and v when nodes in S are updated for the second time. They cannot be updated both in the first round, since we created a policy which forces an order $u \prec v$.
2) One of u, v is in S, and the other one in $V \setminus S$. If u is in S, then it is updated when nodes in S are updated for the first time, and therefore it is updated before v. If v is in S, then it is updated when nodes in S are updated for the second time, and therefore it is updated after u.
3) If $u, v \in V \setminus S$, then u is updated before v, because we ordered the vertices of $V \setminus S$ topologically.

This proves that we created a correct joint schedule. Now let σ be a joint schedule for a network update problem that uses $|V| + k$ touches. Then, let S be the set of those nodes, which are updated at least twice. As each node has to be updated at least once, the size of S is at most k. We claim that S is a directed feedback vertex set of G. For the sake of contradiction, let us assume that S is not a directed feedback vertex set of G. Then there is a cycle $(v_1, v_2, \ldots, v_\ell)$ in $G \setminus S$. For each $i \in \{1, \ldots, \ell - 1\}$, we created a policy with order v_i, v_{i+1}. In σ each of them appears only once (since every node which is touched more than once, is part of S), therefore, by transitivity, v_1 must be updated before v_ℓ. But in G there is an edge (v_ℓ, v_1) (since there is a cycle), so in σ, v_k must

be updated before v_1. Therefore σ is not a correct schedule. ■

VI. Related Work

The problem of updating [7], [29], [32], [34], [47], [54], synthesizing [19] and checking [43] SDN policies [40] as well as routes [33] has been studied intensively already. What is more, route update problems of course predate SDN, and we refer the reader to a recent survey by Foerster *et al.* [11] for an overview of research on update problems in both SDN and non-SDN contexts.

In their seminal work, Reitblatt *et al.* [47] initiated the study of network updates providing strong, per-packet consistency guarantees, and the authors also presented a 2-phase commit protocol. This protocol also forms the basis of the distributed control plane implementation in [7]. Per-packet consistency is a relatively strong requirement that fulfills many other properties (including loop-freedom), but it comes at the cost of requiring a two-phase update mechanism that incurs substantial delay between the two phases and doubles flow entries temporarily [53]. Mahajan and Wattenhofer [34] started investigating a hierarchy of weaker transient consistency properties, also introducing node-reordering algorithms for loop-freedom, for a single policy update. In their paper, Mahajan and Wattenhofer proposed an algorithm to "greedily" select a maximum number of edges which can be used early during the policy installation process. This study was recently refined in [2], [12], and [13], where several hardness results and approximation algorithms are presented; these papers however focus on the objective of maximizing the number of simultaneous updates. There also exist first results on consistent update schedules minimizing the number of update rounds [3], [8], [24], [31], [52], [55]. The measurement studies in [27] and [54] provide empirical evidence for the non-negligible time and high variance of node updates, motivating their and our work.

Our work builds upon [34], in the sense that we extend the study of loop-free network updates to multiple concurrent policy updates. The goal of minimizing the number of switch interactions renders the underlying algorithmic problem different in nature. To the best of our knowledge, we are the first to consider this extension.

More recently, researchers have also started investigating consistent updates for networks which include middleboxes and network functions [18]. Ludwig *et al.* [30] presented update protocols which maintain security critical properties such as waypointing, via a firewall, in a transiently consistent manner. Ghorbani and Godfrey [15] argue that in the context of network function virtualization, stronger consistency properties are required, and Zhou *et al.* [53] presented a general approach to enforce customizable consistency properties in SDNs.

Finally, we note that from a technical perspective, our work is also related to Middendorfs "supersequence runs" [38]: However, if in each input sequence each letter from the alphabet appears at most once (and that is the only case we are interested in this paper), the minimal run supersesequence is equivalent to shortest common supersequence, and hence the model does not provide us with additional insights. Also the polynomial-time algorithms presented in [38] for scenarios where the alphabet size is 2, does not have relevant implications for our work as it would concern networks of size two.

Bibliographic Note: An early version of this paper was presented at the IFIP DSN 2016 conference [10].

VII. Conclusion

Over the last years, even tech-savvy companies such as GitHub, Amazon, GoDaddy, etc. have reported major issues with their network, due to misconfigurations and including loops [16], [20], [39], [51]. Given the increasing importance computer networks play today, this is worrying.

While software-defined networking promises a formally verifiable network operation, the paradigm still poses fundamental challenges. In particular, correctly operating a network from a logically centralized perspective is non-trivial, because of the asynchronous and unreliable communication between switches and controller. With the advent of more adaptive SDNs, where routes can be changed more frequently and flexibly [11], as we have shown in this paper, it becomes algorithmically challenging to operate networks and dynamically change routes in a consistent and efficient manner. As these networks are currently moving into production (in data centers, but also in the wide-area Internet), this is problematic.

We understand our paper as a first step toward more efficient yet consistent multi-policy SDN updates, and believe that our work opens many interesting questions for future research. In particular, further work is required to fully chart the computational complexity landscape of loop-free network updates. More generally, it will be interesting to extend our work toward more sophisticated dependability properties, such as blackhole freedom or waypoint enforcement. Finally, it will be interesting to study approximation algorithms: do there exist fast scheduling algorithms which guarantee an "almost optimal" number of touches? Especially in large networks where there can be many switches along a path and where multiple policies need to be updated simultaneously, our hardness result may inhibit fast exact solutions.

Acknowledgments

The authors thank A. Feldmann for useful inputs.

References

[1] A. Gupta *et al.*, "SDX: A software defined Internet exchange," in *Proc. ACM SIGCOMM*, 2014, pp. 551–562.

[2] S. A. Amiri, A. Ludwig, J. Marcinkowski, and S. Schmid, "Transiently consistent SDN updates: Being greedy is hard," in *Proc. 23rd Int. Colloq. Struct. Inf. Commun. Complex. (SIROCCO)*, 2016, pp. 391–406.

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

[4] T. Anderson, L. Peterson, S. Shenker, and J. Turner, "Overcoming the Internet impasse through virtualization," *Computer*, vol. 38, no. 4, pp. 34–41, Apr. 2005.

[5] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "SNAP: Stateful network-wide abstractions for packet processing," in *Proc. ACM SIGCOMM*, 2016, pp. 29–43.

[6] T. Bu, L. Gao, and D. Towsley, "On characterizing BGP routing table growth," *Comput. Netw.*, vol. 45, no. 1, pp. 45–54, 2004.

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

[8] P. Černý, N. Foster, N. Jagnik, and J. McClurg, "Optimal consistent network updates in polynomial time," in *Proc. Int. Symp. Distrib. Comput. (DISC)*, 2016, pp. 114–128.

[9] D. Drutskoy, E. Keller, and J. Rexford, "Scalable network virtualization in software-defined networks," *IEEE Internet Comput.*, vol. 17, no. 2, pp. 20–27, Mar./Apr. 2013.

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

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

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

[13] K.-T. Förster and R. Wattenhofer, "The power of two in consistent network updates: Hard loop freedom, easy flow migration," in *Proc. 25th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Aug. 2016, pp. 1–9.

[14] P. Francois and O. Bonaventure, "Avoiding transient loops during IGP convergence in IP networks," in *Proc. IEEE INFOCOM*, Mar. 2005, pp. 237–247.

[15] S. Ghorbani and B. Godfrey, "Towards correct network virtualization," in *Proc. ACM HotSDN*, 2014, pp. 109–114.

[16] GitHub. (2016). [Online]. Available: https://github.com/blog/1346networkproblemslastfriday

[17] I. Poese *et al.*, "Improving content delivery with padis," *IEEE Internet Comput.*, vol. 16, no. 3, pp. 46–52, May/Jun. 2012.

[18] J. Martins *et al.*, "Clickos and the art of network function virtualization," in *Proc. USENIX NSDI*, 2014, pp. 459–473.

[19] J. McClurg, H. Hojjat, P. Černý, and N. Foster, "Efficient synthesis of network updates," in *Proc. ACM PLDI*, 2015, pp. 196–207.

[20] J. Jackson, "Godaddy blames outage on corrupted router tables," *PC World*, London, U.K., 2011. [Online]. Available: https://www.pcworld.com/article/262142/godaddy_blames_outage_on_corrupted_router_tables.html

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

[22] T. Jiang and M. Li, "On the approximation of shortest common supersequences and longest common subsequences," *SIAM J. Comput.*, vol. 24, no. 5, pp. 1122–1139, 1995.

[23] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani, "Consensus routing: The Internet as a distributed system," in *Proc. 5th USENIX Symp. Netw. Syst. Design Implement.*, 2008, pp. 351–364.

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

[25] T. Koponen *et al.*, "Network virtualization in multi-tenant datacenters," in *Proc. USENIX NSDI*, 2014, pp. 203–216.

[26] M. Kuzniar, P. Peresíni, and D. Kostić, "Providing reliable FIB update acknowledgments in SDN," in *Proc. 10th ACM CoNEXT*, 2014, pp. 415–422.

[27] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about SDN flow tables," in *Proc. PAM*, 2015, pp. 347–359.

[28] M. Lewin, D. Livnat, and U. Zwick, "Improved rounding techniques for the MAX 2-SAT and MAX DI-CUT problems," in *Integer Programming and Combinatorial Optimization*. Berlin, Germany: Springer-Verlag, 2002, pp. 67–82.

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

[30] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid, "Transiently secure network updates," in *Proc. ACM SIGMETRICS*, 2016, pp. 273–284.

[31] A. Ludwig, J. Marcinkowski, and S. Schmid, "Scheduling loop-free network updates: It's good to relax!" in *Proc. ACM PODC*, 2015, pp. 13–22.

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

[33] M. Casado *et al.*, "Ethane: Taking control of the enterprise," in *Proc. ACM SIGCOMM*, 2007, pp. 1–12.

[34] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *Proc. ACM HotNets*, 2013, Art. no. 20.

[35] D. Maier, "The complexity of some problems on subsequences and supersequences," *J. ACM*, vol. 25, no. 2, pp. 322–336, 1978.

[36] J. McClurg, H. Hojjat, N. Foster, and P. Černý, "Event-driven network programming," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 369–385, 2016.

[37] D. Medhi and K. Ramasamy, *Network Routing: Algorithms, Protocols, and Architectures*. San Francisco, CA, USA: Morgan Kaufmann, 2007.

[38] M. Middendorf, "Supersequences, runs, and CD grammar systems," *Develop. Theor. Comput. Sci.*, vol. 6, pp. 101–114, 1994.

[39] R. Mohan, "Storms in the cloud: Lessons from the Amazon cloud outage," in *Security Week*, 2011. [Online]. Available: https://www.securityweek.com/storms-cloud-lessons-amazon-cloud-outage

[40] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *Proc. NSDI*, 2013, pp. 1–13.

[41] O. Padon *et al.*, "Decentralizing SDN policies," in *Proc. ACM POPL*, 2015, pp. 663–676.

[42] Y. Ohba, "Issues on loop prevention in MPLS networks," *IEEE Commun. Mag.*, vol. 37, no. 12, pp. 64–68, Dec. 1999.

[43] P. Kazemian *et al.*, "Real time network policy checking using header space analysis," in *Proc. USENIX NSDI*, 2013, pp. 91–111.

[44] P. Pereíni, M. Kuzniar, M. Canini, and D. Kostic, "ESPRES: Transparent SDN update scheduling," in *Proc. ACM HotSDN*, 2014, pp. 73–78.

[45] *Perfbench*. (2017). [Online]. Available: https://github.com/tum-lkn/perfbench

[46] R. J. Perlman and G. P. Koning, "Bridge-like Internet protocol router," U.S. Patent 5 309 437, May 3 1994.

[47] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. ACM SIGCOMM*, 2012, pp. 323–334.

[48] S. Schmid and J. Suomela, "Exploiting locality in distributed SDN control," in *Proc. ACM HotSDN*, 2013, pp. 121–126.

[49] M. Shand and S. Bryant, *Internet Engineering Task Force (IETF)*, document RFC 5715, 2010.

[50] V. G. Timkovskii, "Complexity of common subsequence and supersequence problems and related problems," *Cybernetics*, vol. 25, no. 5, pp. 565–580, 1989.

[51] United. (2011). [Online]. Available: http://newsroom.united.com/newsreleases?item=124170

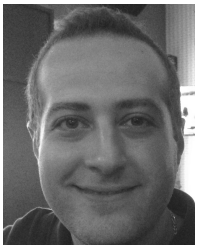
[52] S. Vissicchio and L. Cittadini, "FLIP the (Flow) table: Fast lightweight policy-preserving SDN updates," in *Proc. IEEE INFOCOM*, Apr. 2016, pp. 1–9.

[53] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey, "Enforcing customizable consistency properties in software-defined networks," in *Proc. USENIX NSDI*, 2015, pp. 73–620.

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

[55] Y. Yuan, F. Ivančić, C. Lumezanu, S. Zhang, and A. Gupta, "Generating consistent updates for software-defined network configurations," in *Proc. 3rd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2014, pp. 221–222.

[56] Z. A. Qazi *et al.*, "SIMPLE-fying middlebox policy enforcement using SDN," in *Proc. ACM SIGCOMM*, 2013, pp. 27–38.



Arsany Basta received the M.Sc. degree in communication engineering and the Dr.-Ing. degree (Ph.D.) (Hons.) from Technische Universität München (TUM), Munich, Germany, in 2012 and 2017, respectively. He joined the Chair of Communication Networks, TUM, in 2012, as a Research and Teaching Staff Member. His current research focuses on the application of software-defined networking and network functions virtualization to core networks (mobile) toward the next-generation 5G.

Andreas Blenk received the Diploma degree in computer science from the University of Würzburg, Germany, in 2012. He is currently pursuing the Ph.D. degree with Technische Universität München (TUM), München. He joined the Chair of Communication Networks, TUM, in 2012, where he is currently a Research and Teaching Associate. His research is focused on flexible and predictable virtualization of software-defined networks and data-driven networking algorithms.

Szymon Dudycz received the M.Sc. degree in computer science from the University of Wroclaw in 2016, where he is currently pursuing the Ph.D. degree. His main research interests are various matching problem variants and their application in approximation algorithms.

Arne Ludwig received the Diploma degree and the Ph.D. degree in computer science from the Technical University of Berlin, Germany, in 2011 and 2016, respectively. He is currently with the SAP Innovation Center, Potsdam, Germany. His research interests include algorithms (online), software-defined networks, network virtualization, and network economics.

Stefan Schmid received the M.Sc. and Ph.D. degrees from ETH Zurich, Switzerland, in 2004 and 2008, respectively. In 2009, he held a post-doctoral position with TU Munich and the University of Paderborn. Between 2009 and 2015, he was a Senior Research Scientist with the Telekom Innovation Laboratories, Berlin, Germany. From 2016 to 2017, he was an Associate Professor with Aalborg University, Denmark, and held a part-time position with TU Berlin. He is currently a Professor in computer science with the University of Vienna, Austria. His research interests revolve around the fundamental and algorithmic problems of networked and distributed systems. He was a recipient of the IEEE Communications Society ITC Early Career Award in 2016.