Can't Touch This: Consistent Network Updates for Multiple Policies

Szymon Dudycz[1] Arne Ludwig[2] Stefan Schmid[2,3]

[1] University of Wroclaw, Poland [2] Technical University Berlin, Germany [3] Aalborg University, Denmark

***Abstract*—Computer networks such as the Internet or datacenter networks have become a a crucial infrastructure for many criticial services. Accordingly, it is important that such networks preserve correctness criteria, even during transitions from one correct configuration to a new correct configuration. This paper initiates the study of how to simultaneously update multiple routes in a Software-Defined Network (SDN) in a transiently consistent and efficient manner. In particular, we study the problem of minimizing the number of switch interactions, in this paper also called "touches". 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 correct update schedules for individual policies, computes an optimal global schedule with minimal touches.**

I. Introduction

The availability and protection of computer networks such as the Internet or datacenter (cloud) networks, is becoming a national and world-wide 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 [3].

Software-defined networking is an interesting new paradigm which promises to overcome the Internet ossification. 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], [14] as well as the potentially more scalable network virtualization [6], [16] 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, posing many challenges [5], [8], [17], [23], [25], [31], [37], [38]. Several of these challenges are due to the asynchronous communication channel between switches and controller, which exhibits non-negligible and varying delays [37], [43].

A fundamental problem which has recently received much attention regards the consistent update of network routes [8], [22], [25], [37], [42]. A particularly interesting approach to solve the update problem is to proceed *in rounds* [22], [25]: 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 [17]), 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 [44] or multiple controllers [5]) or additional TCAM entries [5], [37] (which is problematic given the fast table growth both in the Internet as well as in the highly virtualized datacenter [4]). Moreover, this approach also allows (parts of the) paths to become available sooner [25].

However, so far research focused on devising network update schemes for a *single policy*: for scenarios where a single route [22], [37], or all (destination-based) routes to a single destination [25] need to be updated. However, especially in large and dynamic networks, it is likely that multiple routes have to be updated simultaneously [35]. For example, consider a wireless network where users arrive in groups (e.g., at a train station), or a Content Distribution Network where traffic is reassigned to servers in batches [10]. It is well-known that updating a switch and its datastructures comes at a certain cost [43], [27], and it is useful to batch updates [18].

Our Contributions: This paper initiates the study of how to jointly optimize the update of multiple routing policies (i.e., multiple complete source-destination paths) in a transiently consistent (namely loop-free) yet efficient manner. In particular, we consider a most fundamental consistency requirement, *loop-freedom* [22], [25]: 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 [39] on loop-free layer-2 spanning tree constructions [36], on avoiding microloops in MPLS [32], on loop-free IGP migration [7], etc. We in this paper aim to devise loop-free update algorithms for multiple policies in SDNs, such that the number of switch interactions, called *touches*, is minimized.

We show that the network update problem features interesting connections to *Directed Feedback Vertex Set*, *Shortest Common Supersequence (SCS)*, *Supersequence Run* problems [28]. Our main result is a negative one: we prove that the problem is computationally hard, already for two policies which by themselves can be updated in two rounds, by a rigorous reduction from *Max-2SAT* [19]. We complement this negative result by presenting an optimal polynomial-time algorithm to combine consistent update schedules computed for individual policies (e.g., using any existing algorithm, e.g., [22], [25]), into a global schedule guaranteeing a minimal number of touches.

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

II. Model

We are given a network which is controlled by a (logically) centralized software (the so-called controller) which communicates forwarding rule updates to the switches (the *nodes*), over an asynchronous but reliable channel. Due to this asynchrony, we require the controller to send out simultaneous updates 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.

The controller needs to simultaneously update k routing policies, defined over a set U of $n = |U|$ to-be-updated nodes. Each policy update is a pair $(\pi_1^{(i)}, \pi_2^{(i)})$, where $\pi_1^{(i)}$ is the *old route* and $\pi_2^{(i)}$ is the *new route* 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. [22], 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 will assume that the k routing policies are defined over independent parts of the header space [33], i.e., packets of different flows are forwarded according to different (and non-aggregated) rules. However, multiple routes may include the same nodes. Accordingly, as we will see, when reasoning about consistency, we can focus on the correct update of different policies individually; however, for efficiency, we will coordinate the updates to shared nodes, to minimize the node interactions.

Packets should never be delayed or dropped at a node: 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 $in_j^{(i)}$.

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* $U_1^{(i)}, U_2^{(i)}, \ldots,$

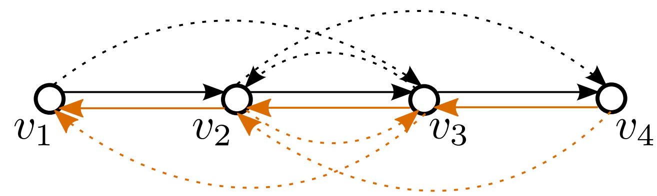

Fig. 1: Example with two concurrent policy updates: at the *top* update $(\pi_1^{(1)}, \pi_2^{(1)})$ in *black*, at the *bottom* update $(\pi_1^{(2)}, \pi_2^{(2)})$ in *orange*. The old policies ($\pi_1^{(1)}$ and $\pi_1^{(2)}$) are drawn using *solid* lines, the new policies ($\pi_2^{(1)}$ and $\pi_2^{(2)}$)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

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 $U_{t'}^{(i)}$ 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: 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), and consider a most fundamental consistency property: *loop-freedom*.

Loop-Freedom: 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 $E_t = out_1^{(i)}(U_{>t}^{(i)} \cup \overline{X}) \cup out_2^{(i)}(U_{<t}^{(i)} \cup X)$. 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. Note that there also exists an alternative definition of loop-freedom [22], where only the current path between the source and the destination needs to remain loop-free. Our results on the NP-hardness and the algorithm hold for both definitions.

Example: Figure 1 shows an example of a concurrent policy update of two policies: at the *top*, update $(\pi_1^{(1)}, \pi_2^{(1)})$ is shown 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. 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.

Goal: Minimum Number of Touches: Interactions with a node come at a certain cost, resource- and time-wise [2], [18], [43], 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 $U^{(i)} = U_1^{(i)} \uplus U_2^{(i)} \uplus \ldots \uplus U_{r_i}^{(i)}$, 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 $\sum_i |U_i|$, 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.

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, 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.

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

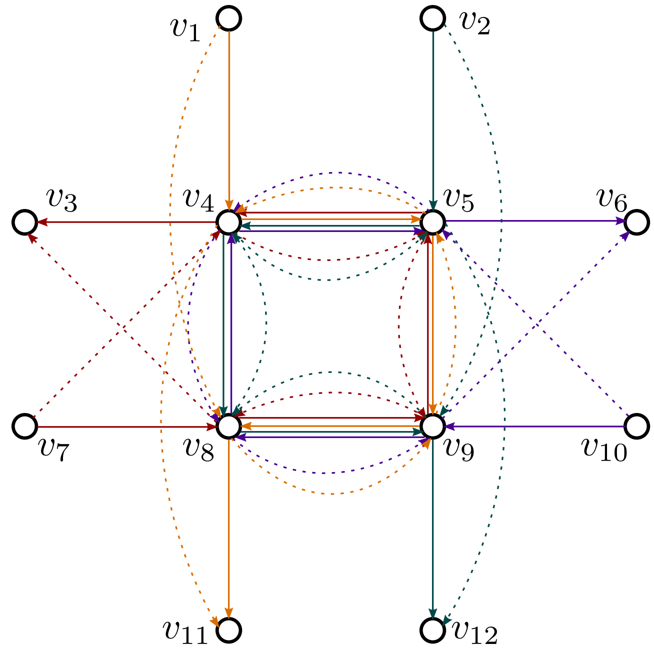

Fig. 2: Construction of a *multi-touch lock*. Four concurrent policy updates are shown in different colors, the old policies are shown using solid lines, and the new policies are shown using dashed lines of the same color.

old policy (according to $\pi_1^{(i)}$), or in the opposite direction. 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.

As we will see, it is useful to classify edges not only for update schedules from $\pi_1^{(i)}$ to $\pi_2^{(i)}$, but also "looking backward in time", 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. 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*·), but also the solid edge points forward w.r.t. the dashed policy (·*F*). 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*.

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. Computational Hardness

In this section we will prove that 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 ($r_i = 2 \quad \forall i$). 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 ($r_i = 2 \quad \forall i$), by a reduction from Max-2SAT [19].

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 a $R = 2$-round schedule. In this section, 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 $Y = (y_1, ..., y_{\ell_2})$, 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 [40].

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. Under these assumptions updating a schedule is a sequence of nodes.

As an example, and to show the relationship to supersequence problems, let us consider the policy presented on Figure 3. 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 3 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. Then, 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

With these intuitions in mind, we now present the main technical result of this paper: we 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.

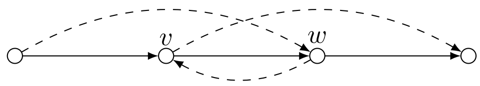

Fig. 3: Example configuration where node w must be updated after node v to avoid loops. A valid schedule is a supersequence of the sequence vw.

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

TABLE I: 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.

1) Outline of Reduction: We prove the hardness by a reduction from Max-2SAT [19]. 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 I 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.

Let us consider the (partial) graphs in Figure 4. 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).

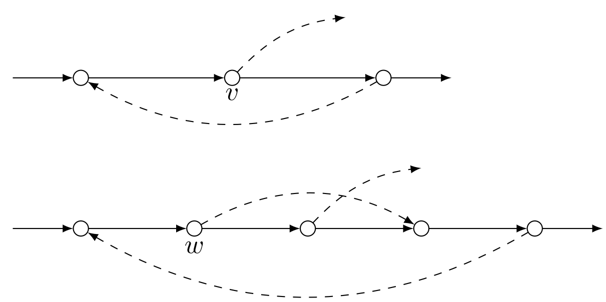

Fig. 4: Examples of $FFFF$ nodes which must be updated in either first or third round.

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 5. 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 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 III-B2). We will describe how to split clause nodes into policies in Section III-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 6, 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

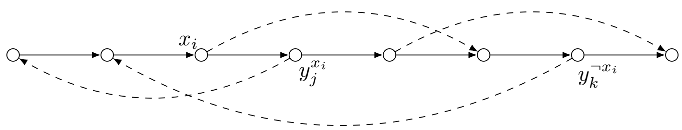

Fig. 5: Outline of a gadget for variables.

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.

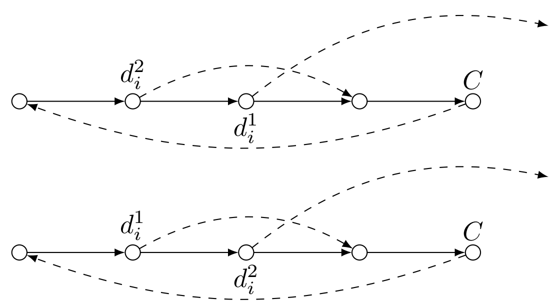

Fig. 6: Gadget for updating clauses in one of the policies.

3) Specifying node type: There are some nodes in the gadget, which we want to make 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 7, 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. The construction is depicted in Figure 7. 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.

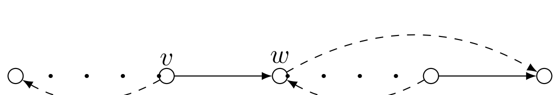

Fig. 7: Construction to make v a BF node.

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 8. In this gadget v is an FF node, w is a FB node and z is a BF node.

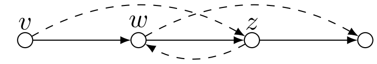

Fig. 8: Gadget for creating nodes of required type.

5) Complete gadget for variable: In Figure 9 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 5) 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 III-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 $x_1, x_2, \dots, x_{p_x}$, $\overline{x}_1$, $\overline{x}_2$, $\overline{x}_{n_x}$. 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 $i \in \{1, \dots, p_x\}$ we will create variables $t_1^i, t_2^i, \dots t_{n_x}^i$. Similarly for each $i \in \{1, \dots, n_x\}$ we will create variables $\overline{t}_1^i, \overline{t}_2^i, \dots \overline{t}_{p_x}^i$.

Now for each $i \in \{1, \dots, p_x\}$ we will create clauses $x_i \implies t_1^i \implies \dots \implies t_{p_x}^i$ ($p \implies q$ in 2SAT can be written as $\neg p \vee q$). We also create similar clauses for each $\overline{x}_i$. Then for each $i \in \{1, \dots, p_x\}$ and $j \in \{1, \dots, n_x\}$ 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 $i \in \{1, \dots, p_x\}$ and $j \in \{1, \dots, n_x\}$, 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.

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 $P_x = \{i \in \{1, \dots, p_x\} \mid |x_i = 1\}$ and $N_x = \{i \in \{1, \dots, n_x\} \mid |\overline{x}_i = 1\}$. Then we set x to be 1,

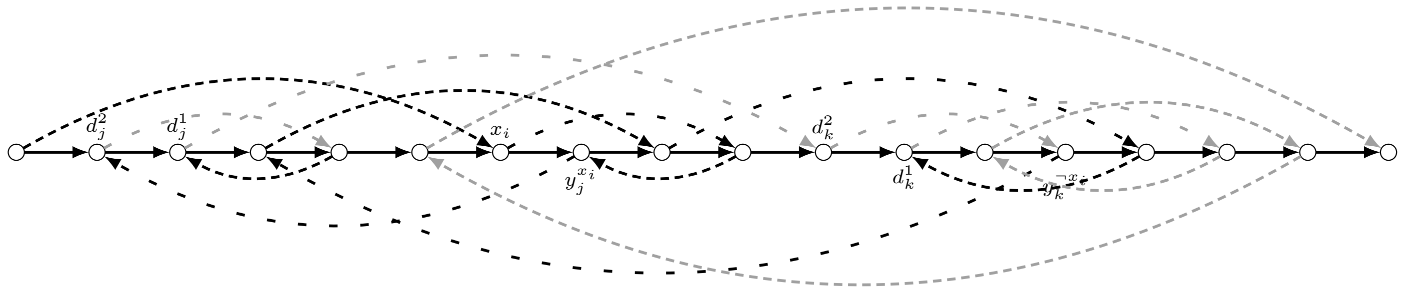

Fig. 9: Complete gadget for a variable. The essential edges of the gadget (presented in Figure 5) are drawn in *loosely dashed black*, edges of clause gadgets are drawn in *loosely dashed grey*, edges added to change the node type (described in Section III-B3) are drawn in *densely dashed grey* and the other edges added to connect the graph are drawn in *densely dashed black*.

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 $\neg t^i_j \vee \neg \overline{t}^j_i$. Therefore there are three possibilities:

1) Some implication in $x_i \implies t^i_1 \implies \ldots \implies t^i_j$ is unsatisfied.
2) Some implication in $\overline{x}_j \implies \overline{t}^j_1 \implies \ldots \implies \overline{t}^j_i$ is unsatisfied.
3) Clause $\neg t^i_j \vee \neg \overline{t}^j_i$ 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 $l = \max\{j \mid \overline{x}_j \in S_x\}$. Then let $x_i \in Q_i$ be such that $x_i = t^i_l$. 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 III-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 $x \implies t^i_1$, which we put in any of the policies. Then each t^i_j appears in 3 clauses (except for $j = p_x$). As a positive literal it appears only in the implication $t^i_{j-1} \implies t^i_j$, which we assign to the other policy than t^i_j. As a negative literal, it appears in the clause $\neg t^i_j \vee t^l_k$, for some l, k; if t^l_k 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 $t^i_j \implies t^i_{j+1}$: this is always possible, as t^i_{j+1} 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 9 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 III-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.

IV. Efficient Schedule Composition

We now present an efficient algorithm which allows to efficiently merge (or *compose*) correct update schedules of individual policies, into a global schedule with minimal touches. Indeed, over the last years, a number of algorithms have been proposed to update a single policy in a consistent manner [22], [25], and the algorithm presented in the following, could serve as a generic post-processor, combining the outputs of these existing algorithms into an optimal global schedule.

In the following, we first present the algorithm and prove that it is optimal and runs in polynomial time, for a constant number of policies. However, we then also show that if the number of policies can be non-constant, the problem of how to optimally merge schedules is computationally hard as well.

Let us first assume that we are given the order of to be updated nodes in their respective policies, and without loss of generality, we assume that in each policy only one node is updated per round. 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, 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 [26], [40]. Jiang and Li 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 $OPT + \mathcal{O}(OPT^{0.707})$ [15]. 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 $\tilde{S} = \{\tilde{v} \mid v \in S\}$.

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: $T[v_1, \ldots, v_m] = 1 + \min_{i \in \{1,\ldots,m\}} T[\tilde{S}_{v_i[-1]}(v_1, \ldots, v_m),\ Q_{v_i[-1]}(v_1, \ldots, v_m)]$

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, ordered update schedules can be merged optimally 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.

While the merging scheme is interesting, we can only achieve a polynomial runtime for a constant number of nodes: 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 [40] 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 1. *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 subcases:

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 $(v_1, v_2, \dots, v_\ell)$ in $G \setminus S$. For each $i \in \{1, \dots, \ell - 1\}$, 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. ∎

V. Related Work

The problem of updating [5], [20], [23], [25], [37], [43], synthesizing [12] and checking [34] SDN policies [30] as well as routes [24] has also been studied intensively. In their seminal work, Reitblatt et al. [37] 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 [5]. 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 [42]. Mahajan and Wattenhofer [25] started investigating a hierarchy of weaker transient consistency properties, in particular also 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. There also exist first results on consistent update schedules minimizing the number of update rounds [22]. The measurement studies in [18] and [43] provide empirical evidence for the non-negligible time and high variance of node updates, motivating their and our work. Our work builds upon [25], 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 [11]. Ludwig et al. [21] presented update protocols which maintain security critical properties such as waypointing, via a firewall, in a transiently consistent manner. Ghorbani and Godfrey [8] argue that in the context of network function virtualization, stronger consistency properties are required, and Zhou et al. [42] 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" [28]: 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 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 [28] for scenarios where the alphabet size is 2, does not have relevant implications for our work as it would concern networks of size two.

VI. 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 [9], [13], [29], [41]. 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, as we have argued in this paper, correctly operating a network from a logically centralized perspective is non-trivial, because of the asynchronous and unreliable communication between switches and controller. Indeed, today, we do not have a good understanding how to design dependable software-defined networks. Given that 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.

Acknowledgments. We thank Anja Feldmann for useful inputs. We also thank the anonymous reviewers for their feedback. Our research was supported by the EU project UNIFY and a German BMBF Software Campus grant (01IS12056).

References

[1] A. Gupta et al. Sdx: A software defined internet exchange. In *Proc. ACM SIGCOMM*, 2014.

[2] A. R. Curtis et al. Devoflow: Scaling flow management for high-performance networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):254–265, 2011.

[3] Thomas Anderson, Larry Peterson, Scott Shenker, and Jonathan Turner. Overcoming the internet impasse through virtualization. *Computer*, 38(4):34–41, April 2005.

[4] Tian Bu, Lixin Gao, and Don Towsley. On characterizing bgp routing table growth. *Comput. Netw.*, 2004.

[5] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. A distributed and robust sdn control plane for transactional network updates. In *Proc. INFOCOM*, 2015.

[6] Dmitry Drutskoy, Eric Keller, and Jennifer Rexford. Scalable network virtualization in software-defined networks. *Internet Computing, IEEE*, 17(2):20–27, 2013.

[7] Pierre Francois and Olivier Bonaventure. Avoiding transient loops during igp convergence in ip networks. In *Proc. IEEE INFOCOM*, 2005.

[8] Soudeh Ghorbani and Brighten Godfrey. Towards correct network virtualization. In *Proc. ACM HotSDN*, 2014.

[9] GitHub. https://github.com/blog/1346networkproblemslastfriday. In *Website*, 2016.

[10] I. Poese et al. Improving content delivery with padis. *IEEE Internet Computing*, 2012.

[11] J. Martins et al. Clickos and the art of network function virtualization. In *Proc. USENIX NSDI*, pages 459–473, 2014.

[12] J. McClurg et al. Efficient synthesis of network updates. In *Proc. ACM PLDI*, 2015.

[13] J. Jackson. Godaddy blames outage on corrupted router tables. In *PC World*, 2011.

[14] Jain et al. B4: Experience with a globally-deployed software defined wan. In *Proc. ACM SIGCOMM*, pages 3–14, 2013.

[15] Tao Jiang and Ming Li. On the approximation of shortest common supersequences and longest common subsequences. *SIAM Journal on Computing*, 24(5):1122–1139, 1995.

[16] Koponen et al. Network virtualization in multi-tenant datacenters. In *Proc. USENIX NSDI*, 2014.

[17] Maciej Kuzniar, Peter Peresíni, and Dejan Kostic. Providing reliable FIB update acknowledgments in SDN. In *Proc. 10th ACM CoNEXT*, pages 415–422, 2014.

[18] Maciej Kuzniar, Peter Peresini, and Dejan Kostic. What you need to know about sdn flow tables. In *Proc. PAM*, 2015.

[19] Michael Lewin, Dror Livnat, and Uri Zwick. Improved rounding techniques for the max 2-sat and max di-cut problems. In *Integer Programming and Combinatorial Optimization*, pages 67–82. Springer, 2002.

[20] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David A. Maltz. zUpdate: Updating Data Center Networks with Zero Loss. In *ACM SIGCOMM*, August 2013.

[21] Arne Ludwig, Szymon Dudycz, Matthias Rost, and Stefan Schmid. Transiently secure network updates. In *Proc. ACM SIGMETRICS*, 2016.

[22] Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. Scheduling loop-free network updates: It's good to relax! In *Proc. ACM PODC*, 2015.

[23] Arne Ludwig, Matthias Rost, Damien Foucard, and Stefan Schmid. Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. In *Proc. ACM HotNets*, 2014.

[24] M. Casado et al. Ethane: Taking control of the enterprise. In *Proc. ACM SIGCOMM*, 2007.

[25] Ratul Mahajan and Roger Wattenhofer. On Consistent Updates in Software Defined Networks. In *Proc. ACM HotNets*, 2013.

[26] David Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 1978.

[27] Deepankar Medhi and Karthikeyan Ramasamy. *Network Routing: Algorithms, Protocols, and Architectures*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[28] Martin Middendorf. Supersequences, runs, and cd grammar systems. *Developments in Theoretical Computer Science*, 6:101–114, 1994.

[29] R. Mohan. Storms in the cloud: Lessons from the amazon cloud outage. In *Security Week*, 2011.

[30] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software Defined Networks. In *Proc. NSDI*, 2013.

[31] O. Padon et al. Decentralizing sdn policies. In *Proc. ACM POPL*, 2015.

[32] Yoshihiro Ohba. Issues on loop prevention in mpls networks. *Communications Magazine, IEEE*, 37(12):64–68, 1999.

[33] P. Kazemian et al. Header space analysis: Static checking for networks. In *Proc. USENIX NSDI*, 2012.

[34] P. Kazemian et al. Real time network policy checking using header space analysis. In *Proc. USENIX NSDI*, 2013.

[35] Peter Peresíni, Maciej Kuzniar, Marco Canini, and Dejan Kostic. ESPRES: transparent SDN update scheduling. In *Proc. ACM HotSDN*, pages 73–78, 2014.

[36] Radia J Perlman and G Paul Koning. Bridge-like internet protocol router, May 3 1994. US Patent 5,309,437.

[37] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proc. ACM SIGCOMM*, pages 323–334, 2012.

[38] Stefan Schmid and Jukka Suomela. Exploiting locality in distributed sdn control. In *Proc. ACM HotSDN*, 2013.

[39] M. Shand and S. Bryant. Internet engineering task force (ietf). In *RFC 5715*, 2010.

[40] V.G. Timkovskii. Complexity of common subsequence and supersequence problems and related problems. *Cybernetics*, 25(5):565–580, 1989.

[41] United. http://newsroom.united.com/newsreleases?item=124170. In *Website*, 2011.

[42] W. Zhou et al. Enforcing customizable consistency properties in software-defined networks. In *Proc. USENIX NSDI*, 2015.

[43] X. Jin et al. Dionysus: Dynamic scheduling of network updates. In *Proc. ACM SIGCOMM*, 2014.

[44] Z. Qazi et al. Simple-fying middlebox policy enforcement using sdn. In *Proc. ACM SIGCOMM*, 2013.