The Power of Two in Consistent Network Updates: Hard Loop Freedom, Easy Flow Migration

Klaus-Tycho Förster*
ETH Zurich
foklaus@ethz.ch

Roger Wattenhofer
ETH Zurich
wattenhofer@ethz.ch

***Abstract*—We study complexity and algorithms for network updates in the setting of Software Defined Networks. Our focus lies on consistent updates for the case of updating forwarding rules in a loop free manner and the migration of flows without congestion. In both cases, we study how the power of two affects the respective problem setting. For loop freedom, we show that scheduling consistent updates for two destinations is NP-hard for a sublinear number of rounds. We also consider the dynamic case, and show that this problem is NP-hard as well via a reduction from Feedback Arc Set. While the power of two increases the complexity for loop freedom, the converse is true when allowing to split flows twice. For the NP-hard problem of consistently migrating unsplittable flows to new routes while respecting waypointing and service chains, we prove that two-splittability allows the problem to be tractable again.**

I Introduction

The power of two is an omnipresent concept in computer science. E.g., when you randomly throw n balls into n bins, the biggest bin will have roughly $\log n$ balls in it. However, if you allow a second choice for each ball, and choose the bin with less balls in it, the biggest bin will just have around $\log \log n$ balls [1]!

Similarly [2], if you consider the difficulty of 3-colorability vs 2-colorability of graphs, or 3-satisfiability vs 2-satisfiability of boolean formulas, going from 3 to 2 choices changes the respective problem from being NP-hard to a tractable one.

On the other hand, the power of two can also turn problems intractable: E.g., Maximizing a single unsplittable $s-t$ flow is easy, but maximizing a 2-splittable $s-t$ flow is NP-hard again [3]!

In this paper, we study on the effects of the power of two on two fundamental network update problems, loop free updates and consistent flow migration: The ongoing rise of Software Defined Networks (SDNs) [4], where the control plane computation is logically centralized, has enabled the use of centralized algorithms in the distributed environment of networks. One of the structural problems in SDNs is related to one of its greatest advantages, the possibility of network updates from a global point of view [5], but in an inherently asynchronous system [6], [7], [8].

In loop free updates, the SDN controller would like to switch from a set of old forwarding rules to a new set of forwarding rules (for flows: to new paths), but without inducing (temporary) forwarding loops in the process.

*Klaus-Tycho Förster was partially supported by Microsoft Research.

Even if the forwarding loop just persists for miliseconds, a large amount of data will be lost in networks with a high throughput [9], [10], [11], [12], [13], [14].

An analogous problem occurs in the migration of flows to new paths, where already small time inconsistencies in the swapping of flows will lead to congestion and increased latency [7], [15], [16], [17], [18], [19].

II Contributions

In particular, our focus lies on the power of two, i.e., how do two destinations/choices or two-splittability of flows affect the tractability of the respective network update problems. Our main contributions are:

1. NP-hardness of sublinear/dynamic strong loop freedom
We show that having forwarding rules that can cover two destinations adds considerable complexity compared to the known one destination case. Specifically, deciding if a loop free update schedule of length $3 \leq r \leq n^{1-\epsilon}$ exists is NP-hard. Furthermore, we consider the dynamic case, where one wants to loop free update as many forwarding rules at once as possible. As it turns out, already the two choices of old and new forwarding rule make this problem equivalent to the NP-hard Feedback Arc Set problem.

2. Tractable consistent migration of 2-splittable flows
We focus on the case of non-mixing old and new flow paths to ensure waypoint enforcement, for the setting of multi-commodity flows. Even if the old and new unsplittable flow assignments are known, we show that deciding if there is a consistent migration is a NP-hard problem. Thus, we turn our attention to 2-splittable flows, and prove how to determine in polynomial time if a consistent migration is possible.

The remaining paper is organized as follows: We start in Section III by formalizing the notion of loop free updates and giving an introductory example, followed by proving the NP-hardness of sublinear update schedules in Section IV, and the NP-hardness of dynamic updates in Section V. We then formalize the notion of consistent (multi-commodity) flow updates respecting waypointing for network migration in Section VI, before covering its intractability in Section VII for unsplittable flows, and its tractability for 2-splittable flows in Section VIII. Lastly, we discuss related work in Section IX, and conclude with a summary in Section X.

III Model for Loop Free Updates

We start modeling the case of one destination d:[1] Let N be a network with old and new forwarding rules f_{old}, f_{new}, each forming an in-tree directed towards d. Due to construction, neither the old nor the new forwarding rules have any loops. The goal is to migrate from the old to the new rules without introducing any (temporary) forwarding loops via a sequence of updates, where one single update consists of a set of nodes changing their forwarding rule from old to new for d, cf. Figure 1.

Due to the inherent asynchrony in networks, it is not possible to control the order in which the nodes contained in an update U change from old to new. Thus, we call an update loop free if the union of the old and the updated forwarding rules is loop free, cf. Figure 1. For subsequent updates, e.g., U_2, we define the current network forwarding state as old.

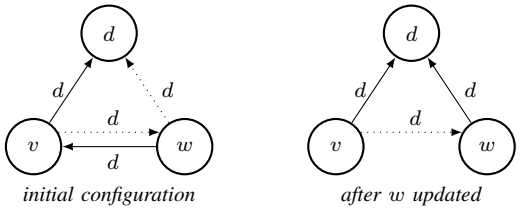

Fig. 1. In this introductory example the initial network configuration is depicted on the left side. Both v and w have old forwarding rules for the destination d, drawn with solid lines. The desired new forwarding rules for d are drawn dotted. If v and w update in the same round, v could update before w due to asynchrony, leading to a loop between v and w. On the other hand, if just w updates to its new forwarding rule, no loop can appear, resulting in the network configuration depicted on the right side. Then, in the next round, v can update loop free to its new forwarding rule by sending all packets for destination d to w.

Dynamic loop free updates

The case of dynamic loop free updates is motivated by applying a new update as soon as some subset of nodes reported successful change of their forwarding rules from old to new. In the following, we will lay special focus on the first update. We define the problem of dynamic loop free updates as finding an update of maximal cardinality s.t. the number of not updated nodes c is minimized.

Scheduling loop free updates

The problem of scheduling loop free updates for one destination d is defined as follows: Given a network N and old and new forwarding rules f_{old}, f_{new}, find a sequence of r loop free updates $U_1, U_2, \ldots, U_r$ s.t. after U_r (i.e., r rounds), the current network forwarding state is identical to f_{new}.

For the case of two destinations d, d', each node can either have two separate old forwarding rules for d, d' or a combined one, analogously for the new rules. We note that the separate forwarding rules of a node might still point towards the same node, and that a node can have separate old and combined new forwarding rules, or vice versa. If a node has a combined new forwarding rule, then updating to this rule replaces the old rules. For the case of separate forwarding rules however, they can be updated in different rounds[2]. Again, we call an update for two destinations loop free, if the union of the old and the updated forwarding rules is loop free for each destination. Thus, we can define the problem of scheduling loop free updates for two destinations in an analogous fashion to one destination, i.e., find a sequence r loop free updates $U_1, U_2, \ldots, U_r$ s.t. after U_r, the current network forwarding state is identical to f_{new}.

[1]We follow some notation standards from [10], [14] for ease of readability.

IV Loop Free Updates with Two Destinations

In this section, we are going to discuss how the complexity of scheduling loop free updates is augmented by adding a second destination. We first restate a result from Ludwig et al. [10], adjusted to our notation:

Theorem 1 ([10])**.** *The problem of scheduling loop free updates for one destination d is NP-complete for a 3–update sequence.*

The proof of Theorem 1 is quite involved, yet elegant. As we will use it as a black box in our own construction, we omit its proof details due to space constraints.

In fact, adding a second destination allows us to make use of the following idea: An update of a node v in the construction of Theorem 1 can be delayed by one round by adding an additional destination, cf. Figure 2.

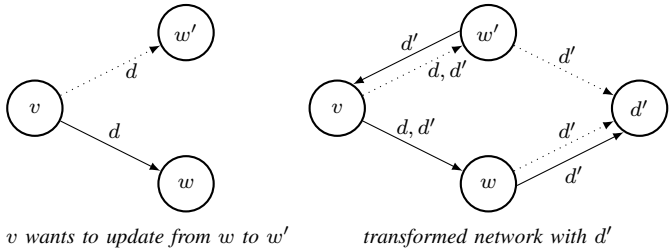

Fig. 2. Old forwarding rules are solid, new ones dashed, the node for destination d is omitted. To delay the update of v from w to w', we transform the network on the left side to the network on the right side. Now, node v cannot update its combined new rule for the destinations d, d' to the node w' until w' switches its rule for d' from v to d'. Note that in general, w' could have arbitrarily many incoming new forwarding rules, but may only have one outgoing old forwarding rule for d'; Thus, a new destination is needed for every delay of an edge when using this construction.

This concept can be performed on all forwarding rules, allowing every update to be delayed by one round by adding $\Omega(n)$ additional destinations. Iterating this idea beyond one round yields the NP-completeness of the problem for a logarithmic number of rounds with a linear number of destinations.

Nonetheless, the deciding factor in this hardness augmentation comes from adding a second destination, as we will show in the following:

Theorem 2. *Let $0 < \epsilon < 1$. For any $3 \leq r \leq n^{1-\epsilon}$, there is a $r^* \geq r$ s.t. the problem of scheduling loop free updates for two destinations d, d' is NP-complete for a r^*–update sequence.*

[2]Following the model of [14], while in the model of [13] even separate forwarding rules must be all updated at once. All updates in the model of [13] are possible in the stronger model of [14] used here, but the reverse is not true.

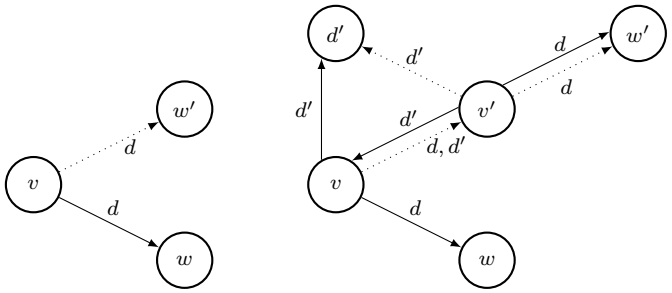

Fig. 3. In this construction, every new forwarding rule for d from a node v to w' gets replaced by a forwarding rule for d, d' to a node v', the node for destination d is omitted among some further edges for simplicity of the illustration. For the node v', old and new forwarding rules for d to w' are added. Again, v cannot update to its new combined forwarding rule to v' before v' has updated its forwarding rule for the destination d'. Observe that in this construction, even when applied to every new forwarding rule of the original network, every node from the original network has only one incoming/outgoing forwarding rule for the new destination d'.

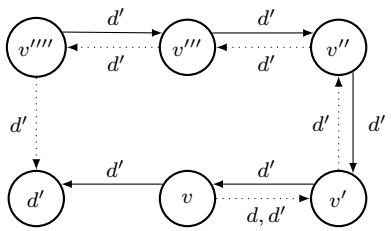

Fig. 4. In Figure 3, v' had a new forwarding rule for d' pointing directly at d', inducing a delay of one round for the update of the new forwarding rule of v. This delay can be extended by adding additional nodes $v'', v''', \ldots$ as depicted in the construction shown in this Figure.

Proof: **The problem is in NP:** Given a sequence of (supposedly loop free) updates, each single one can be checked sequentially in polynomial time to be loop free by using, e.g., Tarjan's algorithm [20].

NP-hardness of $r = 4$: We start by proving the theorem for $r = 4$, before extending it to larger $r \leq n^{1-\epsilon}$. The case of $r = 3$ was already covered above in Theorem 1. We will again use the idea of delaying the update of every forwarding rule by one round akin to the construction in Figure 2, but we will just use one additional destination d', cf. Figure 3.

First, we add a new destination d' to the network. Now let v, w' be two nodes in the network s.t. there is a new forwarding rule for d from v to w'. We add a new node v', and replace the forwarding rule for d from v to w' with one for d, d' from v to v'. Additionally, we add old and new forwarding rules for d from v' to w', a new forwarding rule for d' from v' to d', an old forwarding rule for d' from v' to v, and old and new forwarding rules for d' from v to d', see Figure 3.

After applying this construction for every new forwarding rule in the original network, we added $O(n)$ additional nodes and forwarding rules. We note that every node from the original network has at most one incoming/outgoing forwarding rule for the new destination d'. Now, the update of every forwarding rule is delayed by one round, i.e., it is NP-hard to decide if the network can be updated loop free in $r = 4$ rounds.

NP-hardness of $r \in \mathbb{N}, r \geq 3$: We can extend the construction used above for the NP-hardness of $r = 4$ to any larger natural number, cf. the construction in Figure 4: By adding the node v'' to the construction, the problem becomes NP-hard for $r = 5$, as v cannot update before v', which in turn cannot update before v''.

This can be iterated with v''' for the NP-hardness of $r = 6$, v'''' for $r = 7$, and so on. Note that if the original network has n nodes, the newly constructed network still has $O(n)$ nodes for any $r \in \mathbb{N}$.

NP-hardness for $\leq n^{1-\epsilon}$: In the construction for $r \in \mathbb{N}$, we achieved NP-hardness for r in a network with $\Theta(n + rn) \in \Theta(n)$ nodes. The situation changes when r is dependent on n, e.g., by setting $r = n$, we get networks with $n' \in \Theta(n^2)$ nodes, meaning that the additional delay is just $\Theta(\sqrt{n'})$ opposed to $\Theta(n')$.

However, by setting the delay in the construction to n^x, $n \in \mathbb{N}$, it is NP-hard to decide if a schedule of length r^* exists for some $r^* > \frac{n^x}{n^{x+3}} = n^{1-\frac{3}{x+3}}$. Hence, the theorem holds by setting $x > \frac{3}{\epsilon} - 3$ for any ϵ, $0 < \epsilon < 1$. ■

Thus, adding a second destination increases the complexity of scheduling loop free updates remarkably.

V Dynamic Loop Free Updates

This section covers the case of highly dynamic update times per node, which can result in scheduling the updates beforehand being slower than a dynamic approach. Jin et al. [7] showed that nodes in a production network can take up to 100 times longer than average to apply updates, motivating the use of a dependency graph for dynamic updates of various consistency properties.

The dynamic model was later studied by the authors of [14] for loop free updates under the lens of algorithms and complexity. They showed that for a number of $\Theta(n)$ destinations, the problem of dynamic loop free updates is NP-hard via a reduction from 3-satisfiability.

However, is this problem really hard due to adding a linear amount of destinations – or is the complexity already hidden in the two choices every node has with one destination? Should a node 1) update, or 2) not update? As it turns out, already these two choices make the problem hard:

Theorem 3. *Let N be a network with a single destination d and forwarding rules f_{old} and f_{new}. The problem of dynamic loop free updates for one destination is NP-complete.*

Proof: **The problem is in NP:** Observe that the problem is indeed in NP: Given an update U of new forwarding rules, checking the union of the old and to be updated forwarding

rules to be loop free can be performed in polynomial time, e.g., by Tarjan's algorithm [20].

NP-hardness: It is left to show that the problem is NP-hard. Our proof is a reduction from the classic NP-complete problem Feedback Arc Set (FAS) [2]: Given a directed graph, are there c edges s.t. their removal results in a loop free graph?

We show that for every instance I of FAS, we can construct in polynomial time an instance I' of the corresponding decision problem of the dynamic loop free update problem, s.t. I is satisfiable if and only if I' is a *yes*-instance.

For an illustration of the technique used in the remainder of the construction we refer to the Figures 5, 6, where the network in Figure 6 represents an instance I' created from the instance I of the network in Figure 5.

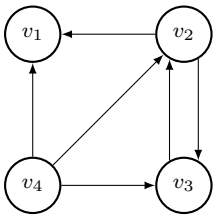

Fig. 5. In this network instance I, there is exactly one loop between v_2 and v_3. Removing either of the two edges would solve the Feedback Arc Set problem in an optimal fashion.

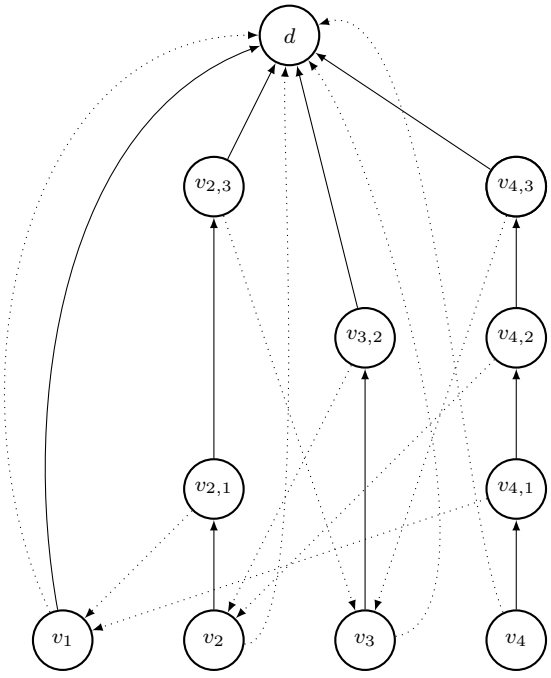

Fig. 6. The corresponding network instance I' from I in Figure 5. The loop between v_2 and v_3 in I is represented by the loop between the nodes $v_2, v_{2,1}, v_{2,3}, v_3, v_{3,2}, v_2$. No further loops exist in I'.

Construction of the new Instance I': Let I be an instance of FAS, i.e., a directed graph $G = (V, E)$ and $c \in \mathbb{N}$: Is there a set of at most c edges $E_c \subseteq E$, s.t. the removal of those edges yields a loop free graph $G^* = (V, E \setminus E_c)$? W.l.o.g., let $V = \{v_1, \ldots, v_n\}$, and let $\Delta(v_i)$ be the out-degree of the node v_i, with the edges $\{e_{i,j_1}, \ldots, e_{i,j_{\Delta(v_i)}}\}$. Again, w.l.o.g. let e_{i,j_k} be the edge from v_i to v_k.

We construct an instance $I' = (V', E_d^{old}, E_d^{new})$ by first defining the set of nodes, then the set of old forwarding rules, and then the set of new forwarding rules.

V' consists of a destination $d \in V$, the nodes V, and $\forall v_i \in V$ we add $\Delta(v_i)$ nodes $\{v_{i,j_1}, \ldots, v_{i,j_{\Delta(v_i)}}\}$. Thus, V' has in total $1 + |V| + |E|$ nodes.

The set of old forwarding rules is defined as follows: For each node $v_i \in V$, we construct a directed path starting at v_i and ending at d as $v_i, v_{i,j_1}, \ldots, v_{i,j_{\Delta(v_i)}}, d$. I.e., we basically have an in-tree for d that consists of $|V|$ paths. The total number of forwarding rules in E_d^{old} is $|V| + |E|$.

The set of new forwarding rules mimics the edges of the graph G in the instance I. For each edge e_{i,j_k} in E, we construct a forwarding rule from v_{i,j_k} to v_{j_k} in E'. Furthermore, for each node $v_i \in V$, we add a forwarding rule $e_{i,d}$ from v_i to d in E'. Thus, the total number of forwarding rules in E_d^{new} is again $|V| + |E|$.

This set of new forwarding rules is loop free: i) The destination d has no outgoing rules, so all $|V|$ forwarding rules pointing at d cannot be part of a loop. Note that those $|V|$ forwarding rules origin from nodes v_i. ii) All other $|E|$ forwarding rules point at nodes v_i, and thus cannot be part of a loop either. Hence, both the old and the new forwarding rules are loop free.

Note that the instance I' can be constructed from the instance I in polynomial time. We now pose the following decision problem for the constructed instance $I' = (V', E_d^{old}, E_d^{new})$: *Is the maximum number of forwarding rules that can be updated loop free at least* $(|E_d^{new}| - c)$*?*

If I is a *yes*-instance, then I' is a *yes*-instance Let E_c be a set of c edges, s.t. the removal of those edges from E yields a loop free directed graph $G^* = (V, E \setminus E_c)$. Let I^*, be the instance I', with the forwarding rules constructed out of E_c removed, i.e. $I^* = (V', E_d^{old}, E_d^{new,*})$. Note that I^* still has $1+|V|+|E|$ nodes. By definition, G^* is loop free. For contradiction, let us assume that $(V, E_d^{old} \cup E_d^{new,*})$ contains a loop L. L must contain a mix of old and new forwarding rules, since each set is loop free individually. Since the outgoing new forwarding rules from v_i point directly at d, the loop L cannot contain two new forwarding rules consecutively.

We now contract the paths of old forwarding rules originating at v_i in I^*, i.e. $v_i, v_{i,j_1}, \ldots, v_{i,j_{\Delta(v_i)}}, d$ into just one new node v_i^{contr} that points at d with an old edge. The new contracted node v_i^{contr} has all ingoing new forwarding rules/edges from v_i and all outgoing new forwarding rules/edges from the nodes $v_{i,j_1}, \ldots, v_{i,j_{\Delta(v_i)}}$, plus one old forwarding rule/edge that points at d. We do this for all nodes $v_i \in V$ in I^*, leading to a contracted graph with $|V| + 1$ nodes, given by

$v_1^{contr}, \ldots, v_n^{contr}, d$. If we were to remove the node d from the contracted graph (and all ingoing edges to d), we would have a graph isomorphic to $G = (V, E)$: Each node v_i^{contr} corresponds to the node v_i in G^*. The same holds for the adjacency relations, as the contracted graph without d contains only new forwarding rules/edges, which in turn were created out of the edges from the edges of G^*.

Thus, if $(V, E_d^{old} \cup E_d^{new,*})$ were to contain a loop L, then the contracted graph would contain a loop as well, and hence the graph G^* too. However, by definition, G^* was loop free, leading to a contradiction.

If I is a *no*-instance, then I' is a *no*-instance Let us assume that no set of c edges $E_c \subseteq E$ exist, s.t. the graph $G^* = (V, E \setminus E_c)$ is loop free. Thus, I is a *no*-instance. Now, let E_c be any set of at most c edges, but let E_c be fixed, and let $G^* = (V, E \setminus E_c)$. Note that G^* contains at least one loop L by definition. Again, as in the argumentation above, we look at the contracted graph without the destination d constructed from the instance I^*. Due to isomorphism, G^* contains a loop as well, which concludes the proof of Theorem 3. ∎

As seen in the above proof, the problem of dynamic loop free updates is strongly related to the Feedback Arc Set problem. The best known approximation ratio which can be achieved for FAS in polynomial time is $O(\log n \log\log n)$, as shown by Even et al. in their seminal paper [21]. Thus, we can show that finding a better guarantee for the dynamic loop free update problem cannot be achieved unless simultaneously improving the general case of the FAS problem:

Corollary 1. *A polynomial algorithm for the problem of dynamic loop free updates for one destination with a better approximation ratio than $O(\log n \log\log n)$ would imply a better polynomial approximation ratio than $O(\log n \log\log n)$ for the Feedback Arc Set problem.*

Proof: Observe that in the proof of Theorem 3, the construction increases the number of nodes from n to $n+m+1$. As $O(\log n \log\log n)$ is equivalent to $O(\log n+m+1 \log\log n+m+1)$ due to logarithmic identities, a better approximation bound would immediately imply a better approximation bound for the Feedback Arc Set problem. ∎

VI Model for (Un)splittable Flow Migration

We model a network N as a directed graph $G = (V, E)$ with non-negative edge-capacities c. An (unsplittable) flow F_j of size d_j starts at a source $S_j \in V$ and is routed along a cycle-free path P_j of enough capacity to its destination $T_j \in V$, i.e., $\forall e \in P_j : d_j \leq c(e)$. We call a set of k flows $F_1, \ldots, F_k$ a multi-commodity flow $\mathcal{F}$ if $\forall e \in E : \sum_{i=1}^{k} F_i(e) \leq c(e)$, i.e., their combined sizes do not violate any capacity constraints. For the sake of simplicity, we assume that the old flow size d_j on P_j is identical to its new size d'_j on $P'_j \neq P_j$, else one could, e.g., reduce the flow size to d'_j on P_j before updating, or migrate and then increase on P'_j.

Consistent flow bandwidth updates. If a set of flows is to be rerouted along other paths in a network update, the inherent asynchrony can lead to them being updated in any order. E.g., if two flows of size one are to be swapped along two paths of capacity one, one of the flows could be updated first, leading to congestion. This lead to the consistency model introduced by *SWAN* [19]: A network update of flows $U = (N, \mathcal{F}, \mathcal{F}')$ is bandwidth consistent, if $\forall e \in E : \sum_{i=1}^{k} \max(F_i(e), F'(e)) \leq c(e)$ holds. I.e., it doesn't matter if a flow is on the old or new path, the edge capacities may not be violated in either case.

Consistent non-mixing flow updates Network flow routes often have to adhere to waypointing, or even service (e.g., firewalls, caches, etc.) chaining [22]. As thus, a flow should only be routed along its old path or its new path, but not a mix of these two, and especially not along totally different paths. This is easy to guarantee if the old and the new flow path are node-disjoint, but not so much when the old and the new path mix: When old and new packets arrive at a switch, marked as being from the same flow, forwarding them according to either or the old rule will lead to the violation of waypointing or the traversal of network functions in the wrong order.

Therefore, we apply the packet stamping method from [23] in the context of congestion-avoidance[3]: We introduce flow rules F_i^{old} for the old path and F_i^{new} for the new path, allowing each packet to respect the service chains. Hence, inspired from a combination of [19] and [23], we call a network update of flows non-mixing consistent, if $\forall e \in E : \sum_{i=1}^{k} \max\left(F_i^{old}(e) + F_i'^{old}(e)\right) + \max\left(F_i^{new}(e) + F_i'^{new}(e)\right) \leq c(e)$ holds, cf. Figure 7.

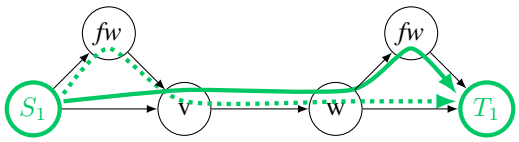

Fig. 7. In this network, the old flow path is drawn solid, the new dotted. Assume that the flow would be split along the old and the new path, and that there would be firewalls on the nodes marked with *fw*. If there is just one flow rule at w, namely to split the flow along the two outgoing edges, then there are flow packets not traversing any firewall. As thus, we introduce separate old and new flow rules for the non-mixing property.

Observe that bandwidth consistency and non-mixing consistency are identical if the old and new flow paths are disjoint, the only difference is in the case when the old and the new path are joint at some point. Thus, the packet-stamping approach can also be omitted in an implementation of the disjoint case.

Consistent non-mixing flow migration So far we described flow updates of one round, but due to dependencies it can easily be the case that one needs to apply various updates before the desired outcome is reached [7]. E.g., F_1 wants to move to the path of F_2, but before that can happen, F_2 first needs to be moved to its new path, cf. Figure 8.

Let $\mathcal{F}$ be the old flow assignment and $\mathcal{F}'$ be the new flow assignment. We call a series of non-mixing consistent flow updates a non-mixing migration from $\mathcal{F}$ to $\mathcal{F}'$, if the following two conditions are met: 1) each flow packet may only use the old or the new flow path, 2) the final flow assignment is $\mathcal{F}'$.

[3]Reitblatt et al. [23] use them to guarantee the non-mixing property (which they call per-flow/per-packet consistency), but do not consider congestion.

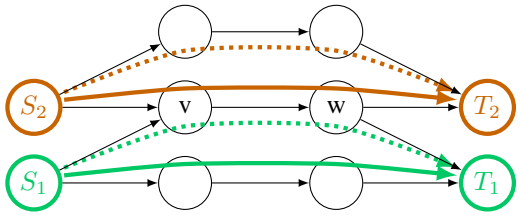

Fig. 8. In this network, the old flow paths are drawn solid, the new dotted. The green flow F_1 has to be moved from its bottom old to its top new path, while the orange flow F_2 has to be moved from its current path to its new top path. All flow sizes and edge capacities are one. If both F_1 and F_2 are updated together, F_1 could move before F_2, causing congestion on edge from v to w. A consistent way of migrating first moves F_2 and then moves F_1.

VII Unsplittable Flow Migration

While it is well studied if a set of demands can be met by unsplittable flows, what about the case of migrating between two known flow assignments? It is known that if the unsplittable flows are allowed to take any path, this problem is NP-hard [16]. We now consider the case where the unsplittable flows are just allowed to take either the old or the new path:

Theorem 4. *Let $\mathcal{F}$, $\mathcal{F}'$ be unsplittable multi-commodity flows. Deciding if there is a non-mixing consistent flow migration from $\mathcal{F}$ to $\mathcal{F}'$ is NP-hard.*

Proof: Our proof will be a reduction from the NP-hard problem Partition [2]: Given a multiset S of k positive integers $s_1, \ldots, s_k$ of combined size $\mathcal{S}$, can the integers be partitioned into two subsets S_1, S_2 s.t. their respective sum is identical to $\mathcal{S}/2$? For each instance I of Partition, we will create an instance I' of the migration problem s.t. I is a *yes*-instance if and only if I' is a *yes*-instance.

Construction of the new Instance I': The network N in the instance I' consists of the nodes $S_1, \ldots, S_k$, S_b, $T_1, \ldots, T_k$, T_b, v_{11}, v_{12}, v_{21}, and v_{22}. There is a directed edge with capacity $\mathcal{S}$ from each $S_1, \ldots, S_k$ to v_{11} and v_{21}, and from each v_{12}, v_{22} to $T_1, \ldots, T_k$. Furthermore, there is a directed edge from v_{11} to v_{12} and from v_{21} to v_{22}, each with capacity $\mathcal{S}$. Lastly, we have edges with capacity $\mathcal{S}$ from S_b to v_{11} and v_{21}, and from v_{12} and v_{22} to T_b. We refer to Figure 9 for an illustration. The old flow configuration $\mathcal{F}$ is given as follows. For $1 \leq i \leq k$, there is a flow F_i of size s_i from S_i to T_i along the path S_i, v_{11}, v_{12}, T_i. Also, there is a flow F_b of size $\mathcal{S}/2$ from S_b to T_b via S_b, v_{21}, v_{22}, T_b In the new flow configuration $\mathcal{F}'$, the flows F_i instead take the other path S_i, v_{21}, v_{22}, T_i, while the flow F_b takes the path S_b, v_{11}, v_{12}, T_b.

Observe that both $\mathcal{F}, \mathcal{F}'$ are valid multi-commodity flows.

If I is a *no*-instance, then I' is a *no*-instance Assume that it is not possible to partition $\mathcal{S}$ into two sets of summed up size $\mathcal{S}/2$ each. Note that for F_b to migrate to its new path S_b, v_{11}, v_{12}, T_b, there has to be a free capacity of $\mathcal{S}/2$ on the edge from v_{11} to v_{12}. The only possibility for that to happen is to select a subset of the paths from $F_1, \ldots, F_k$ of summed up size $\mathcal{S}/2$ to be routed along the edge from v_{21} to v_{22}. But, as the partition instance I is not solvable, no such subset exists.

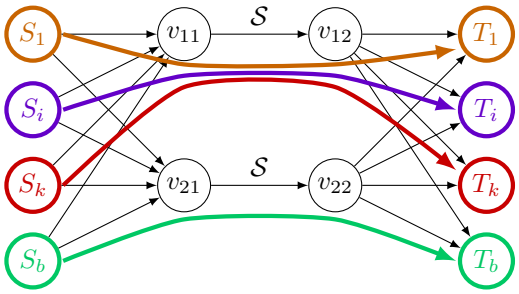

Fig. 9. In this network, there are k flows from $S_1, \ldots, S_k$ to $T_1, \ldots, T_k$ via the upper path v_{11}, v_{12} of combined size $\mathcal{S}$, and one flow from S_b to T_b via v_{21}, v_{22} of size $\mathcal{S}/2$. All edges have a capacity of $\mathcal{S}$. The task is to swap the assignments, i.e., move the k flows to the lower path, and the one bottom flow to the top path. Observe that the only consistent way to do so is to move flows of combined size $\mathcal{S}/2$ to the bottom path, then the bottom flow up, then the remaining flows down. As the k unsplittable flows correspond to a Partition instance, it is NP-hard to decide if this is possible.

If I is a *yes*-instance, then I' is a *yes*-instance Assume that it is possible to partition $\mathcal{S}$ into two sets $\mathcal{S}_1, \mathcal{S}_2$ of summed up size $\mathcal{S}/2$ each. We can then migrate as follows:

We select the flows corresponding to $\mathcal{S}_1$ and move them to their new path, their combined size is $\mathcal{S}/2$ and the edge from v_{21} to v_{22} has a free capacity of $\mathcal{S}/2$. Then, we move the flow F_b to its new path, which now also has a free capacity of $\mathcal{S}/2$. Lastly, we move the flows corresponding to $\mathcal{S}_2$ to their new path, rejoining the flows corresponding to $\mathcal{S}_1$ on the edge from v_{21} to v_{22}. ■

VIII Two-splittable Flow Migration

We saw in the last Section VII that unsplittable flow migration is NP-hard, even if the old and new flow paths are known. However, we will now use the power of two to turn the problem of non-mixing consistent flow migration into a tractable one.

As it turns out, by allowing the flows to be two-splittable, we can decide in polynomial time if a non-mixing consistent flow migration is possible. E.g., in Figure 9, one could move half of each flow from the top path, move the bottom flow up, and lastly, move the remaining half of the original top flows down. We will now first give an overview of the problem before describing our algorithm, then prove its correctness and completeness, before lastly stating some additional methods for the case that no consistent migration exists.

Creation of slack. We are given two unsplittable multi-commodity flows, with the task to migrate from the old to the new one in a non-mixing consistent way. As we allow the flows to be two-splittable, each unsplittable flow can be separated into two distinct parts, each assigned to the old and new path, respectively.

The fundamental inherent problem is posed here by edges e where all capacity is used. In the non-mixing consistent model, no (part of a) flow can be moved to such an edge e until some free slack capacity has been created on e, else the capacity constraints in the consistency property would be violated.

Observation 1. *For a non-mixing consistent network update to increase the combined sizes of any subset of flows on an edge e by x, the combined total sizes of the flows on e must be at most $c(e) - x$.*

The slack s on an edge e is defined as the ratio of the non-used capacity and the capacity on e. E.g., an edge with capacity 10 and flows of combined size 9 on it has a slack of $1/10$. As thus, the question is: Can slack be created on all such edges e, with each flow being only split along its old and new path, i.e., with two-splittable flows?

The following algorithm will try to create slack on all relevant edges, by iteratively attempting to move parts of flows in a non-mixing consistent way until either slack has been created on all edges of $\mathcal{F}, \mathcal{F}'$, or a situation is reached when such movement is not possible. The idea is that we will only create slack, but never remove it completely from any edge.

Algorithm 1 (Creation of slack)**.**
***Input**: (Old) multi-commodity flow $\mathcal{F}$ and a desired mcf $\mathcal{F}'$.*
***Output**: A sequence of non-mixing consistent network updates, starting from $\mathcal{F}$, to create slack on all edges used by $\mathcal{F}, \mathcal{F}'$, or output that this is not possible.*
***Invariant**: Denote the current flows in the N by $\mathcal{F}^*, \mathcal{F}'^*$.*[4]

1) *Let x be the smallest free capacity on any edge in N used by $\mathcal{F}^*, \mathcal{F}'^*$.*
2) *Is there a flow $F_i^* \in \mathcal{F}_i^*$ that has no slack on some edge of their path, but there is slack on all edges of the corresponding path of $\mathcal{F}'^*$?*
   a) *If yes, perform a network update where the flow size of F'^* is increased by $x/2$ and the flow size of F^* is decreased by $x/2$. Then, go to step 1.*
   b) *Else*
      i) *If all edges of $\mathcal{F}^*$ have slack, output **yes** & all performed network updates so far.*
      ii) *If there is still an edge of $\mathcal{F}^*$ without any slack/free capacity, output **no**.*

Lemma 1. *The updates performed by Algorithm 1 are non-mixing consistent and just use the old and new flow paths.*

Proof: The lemma holds as parts of a flow are just moved to their new path if the new path has enough free capacity, no other paths are used. ∎

Lemma 2. *The runtime of Algorithm 1 is $O(k^2 n)$, with k being the number of flows/commodities. The number of network updates performed is at most k.*

Proof: Steps 1 & 2 of Algorithm 1 need to be repeated at most k times, with k being the number of commodities/flows, resulting in at most k network updates: If a flow path already has slack on every edge, this flow does not need to be updated again. Furthermore, each iteration of Step 1 & 2 needs to check $O(k)$ flows of length $O(n)$ in the worst case, resulting in a total runtime of $O(k^2 n)$. ∎

[4]Note that at the start of the algorithm, all flows in $\mathcal{F}'^*$ have a size of 0.

We note that if a sequence of non-mixing consistent network updates leads to a non-mixing consistent migration from $\mathcal{F}$ to $\mathcal{F}'$, then this sequence can also be applied "backwards" to $\mathcal{F}'$, leading to a non-mixing consistent migration from $\mathcal{F}'$ to $\mathcal{F}$. We cast this observation into the following statement:

Observation 2. *A non-mixing consistent migration from $\mathcal{F}$ to $\mathcal{F}'$ exists if and only if a non-mixing consistent migration from $\mathcal{F}'$ to $\mathcal{F}$ exists.*

Non-mixing consistent migration. Let $\mathcal{F}_{slack}$ be a multi-commodity flow in N where every edge used by the flow has a slack of at least s, and similarly, let $\mathcal{F}'_{slack}$ be a multi-commodity flow in N where every edge used by the flow has a slack of at least s. We can then use a method from SWAN [19] to migrate in a non-mixing consistent fashion with $\lceil 1/s \rceil - 1$ updates: Every network update moves a share of s (possibly less in the last update) of each flow to its new path, resulting in the given number of updates. E.g., if the slack is $1/10$, then 10% of the original flow size will be moved in every update, resulting in 9 updates in total. As the invariant of a slack of at least s will be maintained after every update, each performed update is non-mixing consistent.

Algorithm 2 (Deciding non-mixing consistent migration)**.**
***Input**: (Old) multi-commodity flow $\mathcal{F}$ and a desired mcf $\mathcal{F}'$.*
***Output**: **yes**, if a non-mixing consistent migration from $\mathcal{F}$ to $\mathcal{F}'$ is possible, **no** otherwise.*

1) *Run Algorithm 1 on $\mathcal{F}, \mathcal{F}'$ and $\mathcal{F}', \mathcal{F}$. If either output is no, then output **no**, else output **yes**.*

Theorem 5. *Algorithm 2 decides in a runtime of $O(k^2 n)$ if a non-mixing consistent migration is possible.*

We defer the complete proof of Theorem 5 and first give the following algorithm for non-mixing consistent migration:

Algorithm 3 (Performing non-mixing consistent migration)**.**
***Input**: (Old) multi-commodity flow $\mathcal{F}$ and a desired mcf $\mathcal{F}'$.*
***Output**: Either a sequence of non-mixing consistent network updates, starting from $\mathcal{F}$, which form a non-mixing consistent migration to $\mathcal{F}'$, or an output that this is not possible.*

1) *Run Algorithm 1 on $\mathcal{F}, \mathcal{F}'$ and $\mathcal{F}', \mathcal{F}$. If either output is no, then output **no**. Else, denote the resulting networks gotten by applying updates $\mathcal{U}, \mathcal{U}'$ by $\mathcal{F}_{slack}$ and $\mathcal{F}'_{slack}$.*
2) *Let s be the smallest slack on any edge in N used by $\mathcal{F}_{slack}, \mathcal{F}'_{slack}$.*
3) *Apply the calculated non-mixing consistent updates $\mathcal{U}$ to $\mathcal{F}$, resulting in $\mathcal{F}_{slack}$.*
4) *Perform $\lceil 1/s \rceil - 1$ non-mixing consistent updates, each moving an original share of s to its new path, resulting in $\mathcal{F}'_{slack}$.*
5) *Apply the non-mixing consistent updates $\mathcal{U}'$ in reverse, resulting in $\mathcal{F}'$, and output **yes** .*

Combining the above argumentation and Lemma 1, 2 yields:

Corollary 2. *The migration performed by Algorithm 3 is non-mixing consistent, with at most* $2k + \lceil 1/s \rceil - 1$ *updates.*

As thus, we know that a migration performed by Algorithm 3 is non-mixing consistent, but we still need to show that an output of **no** is correct as well:

Lemma 3. *If Algorithm 3 outputs* ***no****, then no consistent non-mixing migration from* $\mathcal{F}$ *to* $\mathcal{F}'$ *is possible.*

Proof: Assume for the sake of contradiction that Algorithm 3 outputs **no**, but that a consistent non-mixing migration from $\mathcal{F}$ to $\mathcal{F}'$ exists with the updates $U_1, U_2, \ldots$. As Algorithm 3 outputs **no**, Algorithm 1 ouputs **no** as well for the case of $\mathcal{F}$ to $\mathcal{F}'$ or the case of $\mathcal{F}'$ to $\mathcal{F}$. Due to Observation 2, we can assume w.l.o.g it was at least for the case of $\mathcal{F}$ to $\mathcal{F}'$. Note that if Algorithm 1 would have output **yes** for both cases, then Algorithm 3 would have output **yes** as well.

Let U_j be the first update where a flow F_i was (partially) moved that Algorithm 1 failed to create slack for on its new path. By assumption, Algorithm 1 was able to create slack (or there was already slack) for all flows of $\mathcal{F}$ moved in $U_1, U_2, \ldots, U_{j-1}$. Note that if U_j was a non-consistent mixing network update, then we can create a non-consistent mixing network update U_j^* from U_j that just contains moving the respective parts of the flow F_i. Recall that Algorithm 1 never removed slack completely from any edge. As thus, Algorithm 1 would have been able to create slack for F_i, as it could have moved a part of flow F_i to its new path after making sure that there is slack for all new flow paths contained in the updates $U_1, U_2, \ldots, U_{j-1}$. Thus, no such non-mixing consistent update U_j could have existed, leading to the desired contradiction, which concludes the proof of Lemma 3. ∎

We now have all the methods necessary to prove Theorem 5:

Proof of Theorem 5: It directly follows that the runtime is $O(k^2 n)$, as we essentially just run Algorithm 1 twice, cf. Lemma 2. It is left to show that Algorithm 2 is correct, which we will infer from its usage in Algorithm 3: With Corollary 2 we know that an output of **yes** is correct. Similarly, with Lemma 3 we know that an output of **no** is correct, concluding the proof of Theorem 5. ∎

What if no non-mixing consistent migration exists? It can be the case that Algorithm 3 outputs that no non-mixing consistent migration of flows exists for two-splittable flows, but that the benefits of the desired new flow $\mathcal{F}'$ outweigh the downsides of congestion during the migration.

In this case, we can apply Algorithm 1 to $\mathcal{F}, \mathcal{F}'$ to pre-compute updates for as many edges with slack as possible in $\mathcal{F}_{slack}, \mathcal{F}'_{slack}$, and migrate non-mixing consistent to $\mathcal{F}_{slack}$ using these updates from Algorithm 1. Then, in the next step, we use the approach from *Dionysus* [7], which breaks consistency during some updates,[5] but still migrates the flows from $\mathcal{F}_{slack}$ to $\mathcal{F}'_{slack}$. Lastly, we can migrate in a non-mixing consistent fashion from $\mathcal{F}'_{slack}$ to the desired multi-commodity flow $\mathcal{F}'$, using the pre-computed updates from Algorithm 1.

[5]The congestion during these network updates can then be reduced by employing the methods of [17], [24].

IX Related Work

For an overview over the topic of network updates in Software Defined Networks, we refer to the recent article of Casado et al. [5]. In particular, the works of He et al. [8], Jin et al. [7], and Kuzniar et al. [6] provide evidence for the inherent asynchrony of switch updates, with widely varying variances in the implementation time. In the following, we now cover work on loop free updates and consistent migration of flows.

Loop free network updates Ito et al. [25] investigated avoiding loops in shortest-path routing by increasing link costs to bypass single links. Shortly after, the study of loop free network updates was initiated by François et al. in their investigation of the convergence of link-state routing protocols [11], [12]. They studied the scheduling of single-destination updates, showing that one can always update in a number of rounds equivalent to the depth of routing tree induced by the new forwarding rules. Their work was later extended to the dynamic setting in [26], [14], with the latter authors also showing the NP-hardness of the dynamic loop free update problem for $\Theta(n)$ destinations and an $O(\log n \log\log n)$ approximation algorithm for the case of one destination. Ludwig et al. [10] further studied the scheduling of loop free updates, and showed that this problem is NP-hard for 3 rounds, leaving the in this paper studied case of $r > 3$ as an open question. In parallel to our work, Dudycz et al. studied a setting similar to Section IV of this article, but allow to break up forwarding rules [27]: They prove that minimizing the number of interactions with the switches for loop freedom is NP-hard in this case, and also study how to efficiently compose schedules for single destinations. Vanbever et al. [13] consider a modified model for scheduling loop free updates for $O(n)$ destinations, in their variant all forwarding rules of a node have to be updated in the same round, making their decision variant NP-hard as well. Loop freedom can also be in conflict with waypoint enforcement, as shown in [28], [29], leading Vissicchio and Cittadini [9] to apply the idea of packet stamping (or 2-phase commit) from the seminal work of Reitblatt et al. [23] in the loop free setting, but they do not consider congestion. Furthermore, Mizrahi et al. [24] study the problem with a time-based approach, aiming to synchronize concurrent network updates to reduce the inherent asynchrony; their work is also applicable to the migration of flows.

Consistent Migration of flows *SWAN* [19] studied the consistent migration of flows in the presence of background traffic, they provided a system that allows to migrate in $\lceil 1/s \rceil - 1$ steps in the presence of a slack of s. Should no slack be available on some edges, they present a binary search heuristic to find a solution. *zUpdate* [18] uses their methods in a datacenter setting. In a similar line of work, Dionysus [7] tries to migrate via a dependency graph, violating some consistency rules if a heuristic search through the dependency graph does not yield a solution. Consistent flow migration in the model of *SWAN* was then considered in [16], where the authors showed how to decide if a consistent migration of flows is possible. A similar setting was studied in [15] for the case of a single logical

destination/source, improving the migration speed. The main difference to our work is that they allow the flows to use any (amount of) paths in the network, not just old and new, and thus also allow the flows to be arbitrarily splittable.

Lastly, in an orthogonal line of work, Jain et al. [17] aim to minimize the impact of congestion during the migration of flows by optimizing the speed of the used hardware and protocols, resulting in faster network update implementations.

X Summary

We studied the power of two in consistent network updates for loop freedom and flow migration in SDNs. We proved that adding a second destination turns scheduling sublinear loop free updates NP-complete, and that the two choices of old and new forwarding rule turns the dynamic loop free update problem NP-complete as well. For unsplittable flow migration, we showed that consistency respecting old and new flow paths is NP-hard. However, when the flows are two-splittable, we give a fast polynomial algorithm for consistent migration, and outlined alternatives when no consistent migration is possible.

Acknowledgements

We would like to thank Stefan Schmid for helpful discussions on the problem of loop freedom and network updates. We also thank the anonymous reviewers for their feedback. Klaus-Tycho Förster was partially supported by Microsoft Research.

References

[1] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, pp. 1094–1104, 2001.

[2] M. R. Garey and D. S. Johnson, *A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[3] G. Baier, E. Köhler, and M. Skutella, "The k-splittable flow problem," *Algorithmica*, vol. 42, no. 3-4, pp. 231–248, 2005.

[4] A. Vahdat, D. Clark, and J. Rexford, "A purpose-built global network: Google's move to SDN," *ACM Queue*, vol. 13, no. 8, p. 100, 2015.

[5] M. Casado, N. Foster, and A. Guha, "Abstractions for software-defined networks," *Commun. ACM*, vol. 57, no. 10, pp. 86–95, 2014.

[6] M. Kuzniar, P. Peresíni, and D. Kostic, "What you need to know about SDN flow tables," in *Passive and Active Measurement - 16th International Conference, PAM 2015, New York, NY, USA, March 19-20, 2015, Proceedings*, ser. Lecture Notes in Computer Science, J. Mirkovic and Y. Liu, Eds., vol. 8995. Springer, 2015, pp. 347–359.

[7] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, F. E. Bustamante, Y. C. Hu, A. Krishnamurthy, and S. Ratnasamy, Eds. ACM, 2014, pp. 539–550.

[8] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M. Thottan, "Measuring control plane latency in sdn-enabled switches," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15, Santa Clara, California, USA, June 17-18, 2015*, J. Rexford and A. Vahdat, Eds. ACM, 2015, pp. 25:1–25:6.

[9] S. Vissicchio and L. Cittadini, "Flip the (flow) table: Fast lightweight policy-preserving sdn updates," in *2016 IEEE Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-15, 2016*. IEEE, 2016.

[10] A. Ludwig, J. Marcinkowski, and S. Schmid, "Scheduling loop-free network updates: It's good to relax!" in *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, C. Georgiou and P. G. Spirakis, Eds. ACM, 2015, pp. 13–22.

[11] P. François, C. Filsfils, J. Evans, and O. Bonaventure, "Achieving sub-second IGP convergence in large IP networks," *Computer Communication Review*, vol. 35, no. 3, pp. 35–44, 2005.

[12] P. François and O. Bonaventure, "Avoiding transient loops during the convergence of link-state routing protocols," *IEEE/ACM Trans. Netw.*, vol. 15, no. 6, pp. 1280–1292, 2007.

[13] L. Vanbever, S. Vissicchio, C. Pelsser, P. François, and O. Bonaventure, "Lossless migrations of link-state igps," *IEEE/ACM Trans. Netw.*, vol. 20, no. 6, pp. 1842–1855, 2012.

[14] K.-T. Förster, R. Mahajan, and R. Wattenhofer, "Consistent Updates in Software Defined Networks: On Dependencies, Loop Freedom, and Blackholes," in *Proceedings of the 15th IFIP Networking Conference, Networking 2016, Vienna, Austra, 17-19 May, 2016*. IEEE, 2016.

[15] S. Brandt, K.-T. Förster, and R. Wattenhofer, "Augmenting anycast network flows," in *Proceedings of the 17th International Conference on Distributed Computing and Networking, Singapore, January 4-7, 2016*. ACM, 2016, p. 24.

[16] S. Brandt, K.-T. Förster, and R. Wattenhofer, "On Consistent Migration of Flows in SDNs," in *2016 IEEE Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-15, 2016*. IEEE, 2016.

[17] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: experience with a globally-deployed software defined wan," in *ACM SIGCOMM 2013 Conference, SIGCOMM'13, Hong Kong, China, August 12-16, 2013*, D. M. Chiu, J. Wang, P. Barford, and S. Seshan, Eds. ACM, 2013, pp. 3–14.

[18] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz, "zupdate: updating data center networks with zero loss," in *ACM SIGCOMM 2013 Conference, SIGCOMM'13, Hong Kong, China, August 12-16, 2013*, D. M. Chiu, J. Wang, P. Barford, and S. Seshan, Eds. ACM, 2013, pp. 411–422.

[19] C. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *ACM SIGCOMM 2013 Conference, SIGCOMM'13, Hong Kong, China, August 12-16, 2013*, D. M. Chiu, J. Wang, P. Barford, and S. Seshan, Eds. ACM, 2013, pp. 15–26.

[20] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.

[21] G. Even, J. Naor, B. Schieber, and M. Sudan, "Approximating minimum feedback sets and multicuts in directed graphs," *Algorithmica*, vol. 20, no. 2, pp. 151–174, 1998.

[22] T. Lukovszki and S. Schmid, "Online admission control and embedding of service chains," in *Structural Information and Communication Complexity - 22nd International Colloquium, SIROCCO 2015, Montserrat, Spain, July 14-16, 2015*, ser. Lecture Notes in Computer Science, C. Scheideler, Ed., vol. 9439. Springer, 2015, pp. 104–118.

[23] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *ACM SIGCOMM 2012 Conference, Helsinki, Finland - August 13 - 17, 2012*, L. Eggert, J. Ott, V. N. Padmanabhan, and G. Varghese, Eds. ACM, 2012, pp. 323–334.

[24] T. Mizrahi, O. Rottenstreich, and Y. Moses, "Timeflip: Scheduling network updates with timestamp-based TCAM ranges," in *2015 IEEE Conference on Computer Communications, INFOCOM 2015, Kowloon, Hong Kong, April 26 - May 1, 2015*. IEEE, 2015, pp. 2551–2559.

[25] H. Ito, K. Iwama, Y. Okabe, and T. Yoshihiro, "Avoiding routing loops on the internet," *Theory Comput. Syst.*, vol. 36, no. 6, pp. 597–609, 2003.

[26] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *Twelfth ACM Workshop on Hot Topics in Networks, HotNets-XII, College Park, MD, USA, November 21-22, 2013*, D. Levine, S. Katti, and D. Oran, Eds. ACM, 2013, pp. 20:1–20:7.

[27] S. Dudycz, A. Ludwig, and S. Schmid, "Can't touch this: Consistent network updates for multiple policies," in *Proc. 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[28] A. Ludwig, M. Rost, D. Foucard, and S. Schmid, "Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks, HotNets-XIII, Los Angeles, CA, USA, October 27-28, 2014*, E. Katz-Bassett, J. S. Heidemann, B. Godfrey, and A. Feldmann, Eds. ACM, 2014, pp. 15:1–15:7.

[29] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid, "Transiently secure network updates," in *Proc. ACM SIGMETRICS*, 2016.