Consistent Updates in Software Defined Networks: On Dependencies, Loop Freedom, and Blackholes

Klaus-Tycho Förster
ETH Zurich
foklaus@ethz.ch

Ratul Mahajan
Microsoft Research
ratul@microsoft.com

Roger Wattenhofer
ETH Zurich
wattenhofer@ethz.ch

***Abstract*—We consider the problem of finding efficient methods to update forwarding rules in Software Defined Networks (SDNs). While the original and updated set of rules might both be consistent, disseminating the rule updates is an inherently asynchronous process, resulting in potentially inconsistent states. We highlight the inherent trade-off between the strength of the consistency property and the dependencies it imposes among rule updates at different switches; these dependencies fundamentally limit how quickly the SDN can be updated. Additionally, we consider the impact of resource constraints and show that fast blackhole free migration of rules with memory limits is NP-hard for the controller. For the basic consistency property of loop freedom, we prove that maximizing the number of loop free update rules is NP-hard for interval-based routing and longest-prefix matching. We also consider the basic case of just one destination in the network and show that the greedy approach can be nearly arbitrarily bad. However, minimizing the number of not updated rules can be approximated well for destination-based routing. For applying all updates, we develop an update algorithm that has a provably minimal dependency structure. We also sketch a general architecture for consistent updates that separates the twin concerns of consistency and efficiency, and lastly, evaluate our algorithm on ISP topologies.**

I. Introduction

The Internet as a whole is a wild place, full of autonomous participants. As such, it is naturally difficult to control centrally; instead, routing and congestion control is achieved through a selection of distributed protocols such as BGP and TCP. However, distributed protocols degrade performance, BGP cannot find the least congested path, and TCP will only crudely approximate the available bandwidth on the path selected by BGP. As a result, a loss of performance is to be expected and accepted. Many desirable properties such as drop freedom of packets, good utilization of links, or packet coherence are not as important as robustness. In contrast, individual networks that make up the Internet are controlled by single administrative entites. These include enterprise networks, ISP networks, data center networks, and wide area networks that connect the data centers of large organizations. The owners of these networks want to get the maximum out of their massive financial investment, which often runs into hundreds of millions of dollars per year (amortized). Towards this end, they have started replacing inefficient distributed protocols.

The technological driver to this paradigm shift are so-called Software Defined Networks (SDNs): In an SDN, the data plane is separated from the control plane, allowing the decision of where and how much data is sent to be made independent of the system that forwards the traffic itself. A centralized controller monitors the current state of the network, then calculates a new set of forwarding rules, and distributes them to the routers and switches [1], [2], [3], [4].

Are centrally controlled SDNs the beginning of the end of distributed protocols? Not so fast! After all, the central SDN controller has to inform the switches about updates, and a network is an inherently asynchronous place, where nodes might even be temporarily not accessible to the controller [4]!

In this paper we will discuss the problems that arise when updating rules in an asynchronous SDN-based network. We will show that despite the central control, distributed computing will have an important role, depending on the kind of consistency model one expects from the network. One of the most basic consistency properties is that packets should not loop. As a result, this property, which we call "loop freedom," is the starting point of our discussion. We will then discuss the broader space of consistency properties and highlight the inherent trade-off between the strength of the property and the intricacy of dependencies it induces among the actions of different switches. These dependencies fundamentally limit how quickly the SDN can be updated.

We build on our prior work [5], which showed that single-destination networks can be updated loop free in a distributed fashion, but did not consider the inherent computational complexity or dynamic architectures. We also extend the view on the consistency space, especially regarding blackholes.

We start in Section III by formally modeling consistent single-/multi-destination network updates, and show that not all updates can be sent out in one flush. In Section IV, we follow up by studying the NP-hardness of loop free updates. In Section V, we study maximizing the number of sent out updates at once and how to build a minimal dependency structure for applying all updates. Afterwards, in Section VI, we reveal the trade-off between consistency properties and update dependencies. Additionally, we consider the impact of resource constraints and show that fast blackhole free migration of rules with memory limits, i.e., a packet arriving at a switch must always have a matching rule to handle it, is NP-hard. We sketch a general architecture for consistent network updates in Section VII and conclude with Section VIII, where we present practical evaluation results.

ISBN 978-3-901882-83-8

II. Background and Related Work

From early papers on the topic (e.g., [6], [7]), we can learn that the primary promises of SDNs were that i) centralized control plane computation can eliminate the ill-effects of distributed computation (e.g., looping packets), and ii) separating control and data planes simplifies the task of configuring the data plane in a manner that satisfies diverse policy concerns. For example, to eliminate oscillations and loops that can occur in certain iBGP architectures, the Routing Control Platform (RCP) [6], [7] proposed a centralized control plane architecture that directly configured the data plane of routers in an autonomous system. However, as we gain more experience with this paradigm, a nuanced story is emerging. Even with SDNs, packets can take paths that violate policy [8] and traffic greater than capacity can arrive at links [3]. What explains this gap between the promise and these inconsistencies? The root cause is that promises apply to the eventual behavior of the network, *after* the data plane state has been changed, but inconsistencies emerge *during* data plane state changes.

Recent works have tackled specific pieces of this consistent update problem. Reitblatt et al. [8], [9] propose a per-packet consistency solution that we call "packet coherence"—each packet is routed entirely using the old rules or the new rules, and never a mix of the two sets; Katta et al. [10] propose extensions to this solution to reduce switch memory overhead. SWAN [3] and [11], [12], [13] propose solutions to ensure that link capacity is not exceeded. The work of Moses et al. [14] discusses balancing update performance versus periods of inconsistencies in a time-based update approach.

We make two contributions to this nascent line of work. First, beyond looking at consistency properties in isolation, we outline the broader consistency space and the fundamental hardness of ensuring different consistency properties. This perspective helps uncover the trade-off between the strength of the consistency property and the difficulty of ensuring it. Second, we investigate in detail loop freedom, a property that has not been considered despite being basic, except for the recent parallel work of [15], [16], [17]. The packet stamping solution of Reitblatt et al. [8] can ensure loop freedom by adding version numbers to packets, but because it ensures the much stronger property of packet coherence, it is slow and has high memory overhead. The whole network needs to be updated first, before being able to use the system—a long delay in updating single node induces a long delay for the complete network. Further, despite the extensions of Katta et al. [10], which trade-off switch memory for speed, packet stamping has high memory overhead because it simultaneously stores both old and new rules. Switch memory is a scarce commodity, with even future generations of switches reaching their memory limit easily when optimizing the network [3]. Our solutions, designed specifically for loop freedom, are faster and memory efficient. Interestingly, a majority of the motivating examples in [8] do not need packet coherence, only loop freedom.

Francois et al. [18],[19] consider avoiding transient loops during the convergence of link-state routing protocols. They argue that, due to high reliability requirements nowadays, one should try to avoid all packet losses. For the case of single-destination rules, they consider the routing tree T of the destination, layered into ranks equivalent to the depth. The ranks are then updated after another, causing depth(T) updates in total. Their mechanism design can achieve fast convergence even in tier-1 ISPs and is carefully fine-tuned for practical deployment [20]. Our work allows for updating nodes from different ranks in one update. As such, our number of updates is not linked to the maximum chain length in the tree, but rather on the maximum chain length in the dependencies imposed by the update in general.

Finally, Vanbever et al. [21] work on a related problem, and study the migration of a conventional (non-SDN) network to a new IGP protocol. The main differences to our work arise from the fact that they impose two restrictions on their model: First, every node must update all its rules at once. Second, only a single node may be updated at a time, one after another. In contrast, we can update individual forwarding entries for many nodes in parallel.

III. Model for Loop Free Routing Updates

We model a network as a set of connected routers and switches (from now on, *nodes*). Packets must be forwarded to their destination without loops. More formally, a network is a directed multi-graph with a set of nodes V, a set of destinations $D \subseteq V$, and a set of destination-labeled edges s.t. all edges labeled with the same set of destinations will not contain a directed loop. These edges form a directed spanning tree with d being the root and all edges being oriented towards d.

Definition 1: Let $T_d = (V, E_d)$ be a directed graph with V being the set of nodes, $d \in D$ being the sole destination, and E_d being the set of edges each labeled with d. The edge from $u \in V$ to $v \in V$ for destination d is noted as $(u, v)_d$. The labeled directed graph T_d is a *single-destination network*, if T_d is a spanning tree with all directed edges being oriented towards d.

Definition 2: Let V be a set of nodes and $D \subseteq V$ be a set of destinations. For all $d \in D$, let $T_d = (V, E_d)$ be a single-destination network and let $E_D = \bigcup_{d \in D} E_d$. Then the labeled directed multi-graph $T_D = (V, E_D)$ is a *multi-destination network*.

When a network needs to be updated, some (potentially all) nodes receive a new set of forwarding rules, leaving the network in a sort of limbo state. At some point all nodes will be updated, but until then, the network might not be consistent, i.e., it might induce loops.

Definition 3: Let $T_D^{old} = (V, E_D^{old})$ and $T_D^{new} = (V, E_D^{new})$ be multi-destination networks for the same set of nodes V and destinations D. Then $U_D = (V, E_D^{old}, E_D^{new})$ is called a *multi-destination network update*. If the labeled directed multi-graph $T_D = (V, E_D^{old} \cup E_D^{new})$ does not contain any loops of edges with the same label, then the update U_D is called *consistent* or *loop free*. A *single-destination network update* U_d can be defined analogously.

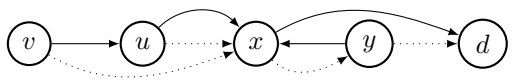

Fig. 1. Illustrating loop freedom. Not all updates can be sent out at once. Dotted edges are *new*, solid edges are *old*.

For an introductory example, consider the five-node single-destination network in Figure 1. Assume that we want to update the routing to destination d from the *old* pattern (solid edges) to the *new* pattern (dotted edges). A naïve method is to send out all updates (e.g., ask v to send packets destined to d to x) in one shot. However, during application of these updates, it might happen that x updates its rule before y, introducing a routing loop between x and y. This loop will eventually disappear, once y updates its rule, but in an asynchronous system with possible message delays and losses, we cannot guarantee when this will happen. Asynchronicity is not a technicality, as nodes in a production network can often react slowly (some switches might take up to $100\times$ longer than average to update [12]), or may not be accessible for some time to the controller [4]. Thus, solutions in which the network can quickly start using as many of the new rules as possible, while maintaining the consistency properties, are preferable.

IV. Updates and Default Rules

Interval routing and longest-prefix matching are common routing techniques for large networks. In interval routing (introduced in [22], cf. [23]), destinations $\{d_1, \ldots, d_{|D|}\}$ are ordered cyclically, and forwarding rules for a node are defined as disjoint intervals over the destinations, cf. [24], [25], [26]. In contrast, longest-prefix routing defines forwarding rules via prefixes of the destination IDs, which may overlap: If two rules are in conflict, the one with the longer matching prefix is chosen, cf. [27], [28]. Both techniques have great practical advantages, since multi-destination routing does not scale well: Even when considering just IPv4 (and not IPv6), no router on the market could store an individual rule for every IP–address. Furthermore, this fine–grained information is not available, since the complete knowledge over a network is usually restrained to one's own Autonomous System.

A subset of both techniques is multi-destination routing with the possibility of default routes. Nodes can either have individual forwarding rules for each destination or a default rule, cf. [29], i.e., all packets go to a specific other node (except for those that reached their destination at the current node). In this section, we show that maximizing a loop free update with default rules is an NP-hard problem – and therefore also NP-hard for both supersets.

Definition 4: Let $T_D = (V, E_D)$ be a multi-destination network and let $u, v \in V$. If all outgoing edges from u point at v in E_D, then those edges E_u may be merged into a *default edge*, labeled with all labels from D (but packets for a destination u do not get forwarded from u). We denote such an edge with $(u, v)_\forall$. I.e., we remove E_u from E_D and add $\{(u, v)_\forall\}$. Let the resulting set of edges of this iterated process be $E_{D,\forall}$. We call $T_{D,\forall} = (V, E_{D,\forall})$ a multi-destination network with default routes or *multi-default network*.

Definition 5: Let $T^{old}_{D,\forall} = (V, E^{old}_{D,\forall})$ and $T^{new}_{D,\forall} = (V, E^{new}_{D,\forall})$ be multi-default networks for the same set of nodes V and destinations D. Then $U_{D,\forall} = (V, E^{old}_{D,\forall}, E^{new}_{D,\forall})$ is called a *multi-default network update*. If the labeled directed multi-graph $T_{D,\forall} = (V, E^{old}_{D,\forall} \cup E^{new}_{D,\forall})$ does not contain any loops of edges with the same label, then the update $U_{D,\forall}$ is called *consistent* or *loop free*.

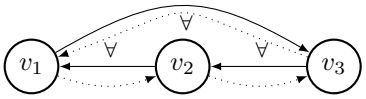

Fig. 2. Illustrating circular dependencies with default routes. Note that both in the old and new rules, no packet will loop: E.g., in the old rules, a packet sent out from v_1 will be forwarded to v_3, and possibly to v_2, but never to v_1 again - as all possible destinations were already reached on the path.

Let us start with an example of just three nodes in Figure 2. We want to update the three old default edges (drawn solid) to the three new default edges (drawn dotted). However, due to circular dependencies, not even a single edge can be updated without causing a loop. This problem can be handled by relaxing the constraints of default routing: One can prevent loops by breaking a single (default) rule into one helper rule for each of the two other destinations, introducing these rules during the update process and then removing them later. In general, this is not desirable, as memory constraints on routers can easily prevent introducing these additional helper rules, cf. [3]. Nonetheless, one can directly check if a non-empty update exists: Check each new edge individually, since adding more edges cannot remove existing cycles. However, even if a multi-default network can be updated with some edges, it is a hard optimization problem. We define the problem of updating multi-default networks as finding the maximum number of edges that can be included in an update at once:

Problem 1: Let $U_{D,\forall} = (V, E^{old}_{D,\forall}, E^{new}_{D,\forall})$ be a multi-default network update. Find a set $E^{max}_{D,\forall} \subseteq E^{new}_{D,\forall}$, s.t. i) $U^{max}_{D,\forall} = (V, E^{old}_{D,\forall}, E^{max}_{D,\forall})$ is a loop free multi-default network update ii) for all loop free multi-default network updates $U^{other}_{D,\forall} = (V, E^{old}_{D,\forall}, E^{other}_{D,\forall})$ with $E^{other}_{D,\forall} \subseteq E^{new}_{D,\forall}$ it holds that they do not contain more edges, i.e., $|E^{other}_{D,\forall}| \leq |E^{max}_{D,\forall}|$.

Theorem 1: Problem 1 is NP-hard.

Proof: Our proof is a reduction from the classic NP-complete satisfiability problem 3-SAT, in the variant with exactly three pairwise different variables per clause [30]:

1) Consider the routes for destination Y in the triangle-gadget from Figure 3. If node $\overline{X}_i$ updates, then node X_i cannot update without inducing a loop for Y, and vice versa. Choosing one of the two update rules corresponds to a variable assignment for a variable x_i in the instance I of 3-SAT: x_i is either *true* or *false*, but not both.
2) Let C be a clause in the instance I of 3-SAT. If there is a variable assignment S that satisfies I, then updating the triangle-gadgets for the variables according to S does not induce a loop for any destination C in the cycle-gadget for the corresponding clause in Figure 4. If no such variable assignment S exists, then at least one triangle-gadget cannot be updated at all without causing a loop for a destination representing a clause.

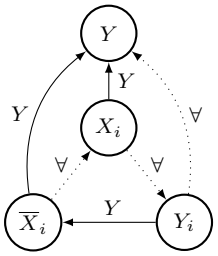

Fig. 3. Triangle-gadget for a variable x_i. New edges are drawn dotted, old solid.

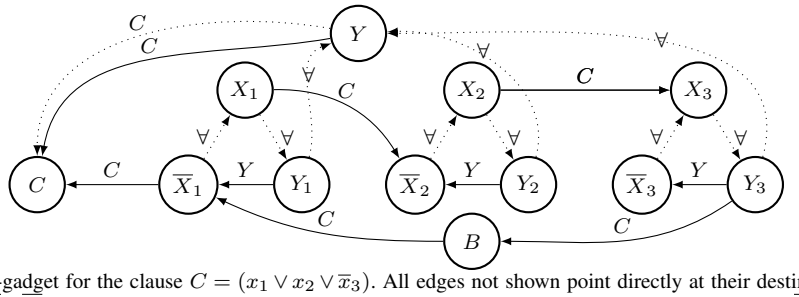

Fig. 4. Cycle-gadget for the clause $C = (x_1 \vee x_2 \vee \overline{x}_3)$. All edges not shown point directly at their destination. Only if all three nodes $\overline{X}_1, \overline{X}_2, X_3$ update their forwarding rule for C, then there is a loop for the label C (via $B - \overline{X}_1 - X_1 - \overline{X}_2 - X_2 - X_3 - Y_3 - B$). E.g., $C = (\overline{x}_1 \vee \overline{x}_2 \vee x_3)$ could only induce a cycle via $B - X_1 - Y_1 - X_2 - Y_2 - \overline{X}_3 - X_3 - B$.

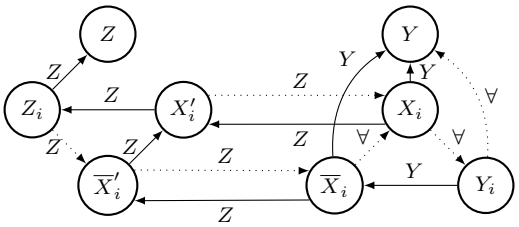

Fig. 5. Extension of the triangle-gadget for a variable x_i from Figure 3. New edges are drawn dotted, old solid. Edges not shown point at their destination. The four possible cycles for destination Z are $i)$ X_i, X'_i, $ii)$ $\overline{X}_i, \overline{X}'_i$, $iii)$ $\overline{X}'_i, X'_i, Z'_i$, $iv)$ $\overline{X}'_i, \overline{X}_i, X_i, X'_i, Z_i$. No other new cycles are introduced.

# in sequence	conflicting clauses	variable false	variable true
1	Y, Z, Y_i	$Y, Z, Y_i, \overline{X}_i$	Y, Z, Y_i, X_i
2	$X_i, \overline{X}_i$	$\overline{X}'_i, X_i$	$X'_i, \overline{X}_i$
3	$X'_i, \overline{X}'_i$	X'_i, Z_i	$\overline{X}'_i, Z_i$
4	Z_i	$\emptyset$	$\emptyset$

Fig. 6. Table depicting the fastest possible migration scenarios for the nodes in Figure 5. $i)$ X_i cannot update before X'_i, $ii)$ $\overline{X}_i$ not before $\overline{X}'_i$, $iii)$ Z'_i not before $\overline{X}'_i$ or X'_i, and $iv)$ X_i or X'_i must update before $\overline{X}'_i$ and $\overline{X}_i$ and Z_i can all three be updated. Note that Y, Z, Y_i can always update right away. However, if there are conflicting clauses (i.e., the corresponding instance is not satisfiable), then neither $\overline{X}_i$ nor X_i can update right away, but must wait for the next update to be sent out – after the conflicts with the clauses have been cleared, thus requiring a sequence of length four. Else, one could update with a sequence of length three, as shown in the two rightmost columns.

3) Let k be the number of variables in I. If k rules from the nodes $X_i, \overline{X}_i$ in the triangle-gadgets can be updated loop free, then there exists a variable assignment S that satisfies the instance I of 3-SAT. If less than k rules can be updated from the nodes $X_i, \overline{X}_i$ in the triangle-gadgets, then I cannot be satisfied. ∎

We now examine interval routing updates: Since the forwarding rules have to be disjoint, we may only apply updates that result in a valid state for each node. I.e., after applying an update, the forwarding rules have to cover all destinations and be disjoint. Removing all current rules and replacing them with a default rule matches this requirement. In a similar fashion, we specify longest-prefix matching updates: A new prefix rule may contain a set of rules it overrides when the rule is inserted at a node. Else, applying an "update" might not change the routing behavior of a node at all.

Corollary 1: Maximizing loop free updates for interval routing or longest-prefix matching is NP-hard.

A. Future Hardware

Even though asynchronicity is inherent in current hardware solutions (e.g., node failures [4] or highly deviating update times [12]), one could imagine these issues being tackled in future work. For example, the method of updating routing information could be decoupled from the remaining computational load of a node, resulting in roughly the same update time for all nodes in a network. Then one would want to find a shortest sequence of precomputed updates that migrate the network from the current old to the desired new routing rules. I.e., the controller will send out a first loop free multi-default update and wait until all affected edge changes are confirmed. This sending out of updates is iterated until all nodes switched their edges to the new desired routing rules. Nonetheless, this problem of updating a network remains hard, i.e., how long is the sequence of updates that are sent out:

Problem 2: Let $U_{D,\forall} = (V, E^{old}_{D,\forall}, E^{new}_{D,\forall})$ be a multi-default network update. Find a sequence of r loop free multi-default network updates $U^1_{D,\forall} = (V, E^{old}_{D,\forall}, E^{new_1}_{D,\forall}), U^2_{D,\forall}, \ldots, U^r_{D,\forall}$ with vertex sets V and corresponding pairwise disjoint new edge sets $E^{new_1}_{D,\forall}, E^{new_2}_{D,\forall}, \ldots, E^{new_r}_{D,\forall}$ s.t. $E^{new_1}_{D,\forall} \cup E^{new_2}_{D,\forall} \cup \cdots \cup E^{new_r}_{D,\forall} = E^{new}_{D,\forall}$ s.t. $r \in \mathbb{N}$ is minimal.

Theorem 2: Problem 2 is NP-hard.

Proof: Note that the construction for the proof of Theorem 1 is not enough to show that Problem 2 is NP-hard: While it is NP-hard to decide if k rules from the nodes $X_i, \overline{X}_i$ in the triangle-gadgets can be updated, the whole network in the proof can always be updated in a sequence of just two updates. In the first step, one would update all nodes (except for the nodes $X_i, \overline{X}_i$ in the triangle-gadgets). Then, in the second step, all the nodes $X_i, \overline{X}_i$ in the triangle-gadgets can be updated, since the possibility of loops in the gadgets created from variables and clauses have vanished after the first update. However, we can extend our construction s.t. for a solution of sequence-length three, all k triangle-gadgets need to update either $X_i, \overline{X}_i$ in the first element of the sequence of updates. Else, a sequence of length four would be needed. The construction is described in the Figures 5 and 6. ∎

Corollary 2: It is NP-hard to approximate the length of the sequence of updates needed for Problem 2 with an approximation ratio strictly better than $4/3$.

V. Algorithms for Loop Free Routing Updates

We first consider variants for single-destination updates and then extend the discussion to the other models. While dynamic updates (i.e., update as much as you can at once) are desirable due to fault-tolerance (see Section I, e.g., a node might be

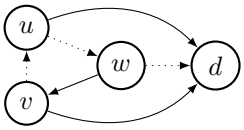

Fig. 7. Illustrating multiple maximal solutions. The nodes u and v cannot update together.

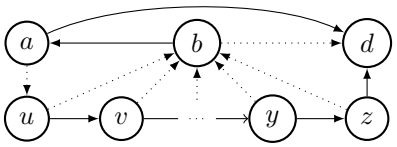

Fig. 8. An update of the nodes a and b is a maximal update, but an update of the nodes $u, v, \ldots, y, z$ and b would be a maximum update.

temporarily unable to update), we also study how to apply all updates in this section. Some proofs are in the Appendix.

We start with single-destination updates: Given an update $U_d = (V, E_d^{old}, E_d^{new})$, find a loop free update $U_d = (V, E_d^{old}, E_d^{'})$ with $E_d^{'} \subseteq E_d^{new}$. We begin by setting $E_d^{'} = \emptyset$:

An update is *maximal*, if adding more edges from E_d^{new} to $E_d^{'}$ violates loop freedom. Maximal updates do not have to be unique, see Figure 7. Node w may switch to the new rule immediately, but not nodes u and v. If they both switch immediately, and w is still using the old rule, we get a loop. So, one of them must wait for w to switch. Either one is fine, i.e. either u must wait for w (and v, w may switch immediately), *or* v must wait for w (and u, w may switch immediately).

Algorithm 1:

1) *Check for an edge $(u,v)_d = e \in E_d^{new}$ if the update $U_d = (V, E_d^{old}, E_d^{'} \cup \{e\})$ is loop free. This loop test can be performed, e.g., by a DFS from node v to find node u on edges with label d.*
2) *If adding e does not introduce a loop, set $E_d^{'} = E_d^{'} \cup \{e\}$.*
3) *Repeat step 1 until all edges were checked.*

Lemma 1: The update calculated by Algorithm 1 is loop free and maximal.

While a maximal solution might seem like a good approach at first glance, it can be far from optimal regarding the number of updates sent out in one flush, see Figure 8: Even for just one destination, a maximum update can be of size $|E_d^{new}| - 1$, but a maximal might just be 2 edges. Can we do better? Since we want to include as many edges as possible, we are essentially solving restricted instances of the NP-complete Feedback Arc Set Problem (FASP) [30]: Given a directed graph, what is the minimum number of edges that needs to be removed to break all cycles. FASP can also be considered in a variant with weighted edges: This allows us to exclude old edges from removal, by giving all old edges an arbitrarily high weight, and all new edges a weight of just 1. The best known approximation algorithm for weighted FAS has an approximation ratio of $O(\log n \log\log n)$ [31], allowing us to enhance the greedy algorithm for maximal updates:

Algorithm 2:

1) *Set the weight of all edges contained in E_d^{old} to ∞, and the weight of all other edges to just* 1.
2) *Calculate a FAS F for the weighted graph $(V, E_d^{old} \cup E_d^{new})$ according to [31].*
3) *Set $E_d^{'} = E_d^{new} \setminus F$.*
4) *Apply Algorithm 1 to make the update maximal.*

Lemma 2: The update calculated by Algorithm 2 is loop free and maximal. The number of removed edges from E_d^{new} can at most be reduced by a factor of $O(\log n \log\log n)$.

Proof: The removal of a FAS implies by definition loop freedom for the network. However, old edges are not allowed to be removed: But since all edges contained in the set of old edges $E_d^{old} = E_d^{old} \cup \left(E_d^{old} \cap E_d^{new}\right)$ have their weight set to infinity, there is always an infinitely better solution than removing any old edge. One would just set the edges being in $E_d^{'}$ to $\emptyset$, which results in a loop free network by definition.

Maximality is ensured by applying Algorithm 1 afterward, which also preserves the loop free property for the network, see Lemma 1. Since Algorithm 1 can only add more edges to the update, and not remove any, the approximation ratio of $O(\log n \log\log n)$ from [31] is still valid. ■

Let us now consider how to apply the whole desired update for a single destination via sending out multiple smaller loop free updates. In the worst case, we will need $|E_d^{new}|$ loop free updates, for example when reversing the links in a ring – only one edge can be updated loop free at a time.

Algorithm 3:

1) *Use Algorithm 1/2 to send out a first update E_{d,g_1}.*
2) *Once a set of nodes has reported back to the central controller that they have performed the rule updates $E'_{d,g_1} \subseteq E_{d,g_1}$ for destination d (and discarded their old rules $E_{d,g_1}^{old'}$), the controller can calculate a current set of old rules. Take into account that the nodes applying the rules $E_{d,g_1} \setminus E'_{d,g_1}$ are still in a limbo state: Either they applied the update already or not, but it is not known due to the asynchronicity until they report in.*
3) *Calculate and send out the next set of update edges $E_{d,g_2} \subseteq \left(E_d^{new} \setminus E_{d,g_1}\right)$ with Algorithm 1/2, which are derived from $\left(V, \left(E_d^{old} \setminus E_{d,g_1}^{old'}\right) \cup E_{d,g_1}, E_d^{new} \setminus E_{d,g_1}\right)$.*
4) *Iterate the process until all new edges are sent out.*

Algorithm 3 computes a series of loop free updates E_{d,g_1}, $\ldots$,E_{d,g_k}, with $\bigcup_{i=1}^{k} E_{d,g_i} = E_d^{new}$. For Algorithms 1 and 2, this can be understood as a dynamic *dependency forest*, which is minimal in the sense that an edge $e \in E_{d,g_j}$ cannot be added to E_{d,g_i}, if $i < j$.

Lemma 3: Iterating either Algorithm 1 or 2 to construct a dynamic dependency forest needs at most $|E_d^{new}|$ non-empty updates to switch the network to the new rules in E_d^{new}.

Proof: If an update is non-empty, then it contains at least one new edge. Thus, $|E_d^{new}|$ non-empty updates suffice to update the network to only new rules. We now show that we can always include at least one new edge in an update, once all sent out rules are applied. Assume that there is no node that is currently applying a new rule, i.e., all nodes that received a new rule for d applied it and reported back to the controller. Thus, no node is in a *limbo* state, where the node was ordered to apply a new rule, but has not successfully reported back yet. For contradiction, let us now assume that Algorithm 1 does not find any new edge to be sent out as an update. Thus, all not yet applied edges were checked, and each would induce a loop when adding it to the network in an update.

However, at least one edge exists that would not induce a loop. For ease of notation, let us call nodes that still need to apply a new rule *old*, and *new* elsewise. Note that currently no

nodes are in limbo. Start from an arbitrary old node, and move along the set of new rules towards the destination d. Since the destination is (by definition) new, along this new-rules path, there must be a last pair of nodes c, p, where the new edge of c points at p, and c is old and p is new. The edge $(c, p)_d$ cannot induce a loop: It points only to nodes which are in the new state already, that is, there are no more old rules which can cause loops. Therefore, Algorithm 1 would have found at least one more edge to be included in a non-empty update to be sent out (and thus, Algorithm 2 as well). ■

Lemma 4: The structure of the dynamic dependency forest is minimal: Any $e \in E_{d,g_j}$ cannot be added to E_{d,g_i}, if $i < j$.

Proof: W.l.o.g. let $e \in E_{d,g_j}$ and consider any update E_{d,g_i} with $i < j$. The set of edges for E_{d,g_i} was maximal, i.e., no more edges could have been added, see the Lemmas 1 and 2. ■

Note that the Algorithms 1, 2, and 3 can be applied to multi-destination network updates by treating them as a set of single-destination network updates: We can compute the variants separately for each label and apply updates in parallel, as edges with different labels will not interfere with each other regarding loop freedom.

A more complex case is where individual rules control routing to multiple destinations and different rules control overlapping sets of destinations. (For non-overlapping destination sets, the situation is similar to above; replace destination sets with a virtual destination.) This situation can emerge in interval-based routing and longest-prefix matching. One can still use adapted versions of Algorithm 1 within Algorithm 3 for maximal loop free updates, but those updates might be empty: In this case, no (loop free) dependency forests to apply all new rules may exist (cf. the network in Figure 2).

We note that in practice, one should divide the multi-graphs $G = (V, E^{old} \cup E^{new})$ into strongly connected components (SCCs), e.g., by implementing Tarjan's algorithm [32]: Edges from different SCCs cannot be part of the same loop, allowing to partition the problem into smaller instances. However, this does not lead to better theoretical approximation bounds.

Also, if we were able to calculate the set of all loops for each label in the multi-graph G induced by an update $G = (V, E^{old} \cup E^{new})$, then we can even improve the approximation ratio for some cases: First, consider each loop for each label as a set of edges, but only add new edges to the sets. The set of old edges was loop free, meaning there are no empty sets. Second, solve the Minimum Hitting Set Problem (MHSP) [30] by choosing a minimum set of new update edges s.t. each loop is broken. MHSP is NP-complete as well, but a greedy approach yields an approximation ratio of $H(|E^{new}|)$ (with some improvement possible [33]), where $H(n)$ is the n^{th} harmonic number, $H(n) \approx \ln n$, cf. [34].

VI. Consistency Space

We now take a broader view of the range of consistency properties. Table 9 helps frame this view. Its rows correspond to consistency properties. We defined loop freedom in Section III; the others are:

	None	Self	Downstream subset	Downstream all	Global
Eventual consistency	Always guaranteed				
Blackhole freedom	Impossible	Add before remove			
Loop freedom (Section V)	Impossible (Lemma 5)		Rule dep. forest		
Packet coherence	Impossible (Lemma 6)			Per-flow ver. numbers	Global ver. numbers [8]
Congestion freedom	Impossible (Lemma 7)				Staged partial moves [3], [11], [12], [13]

TABLE 9
Basic consistency properties & their dependencies.

- **Eventual consistency** No consistency is provided during updates. If the new set of rules computed by the controller are consistent (by any definition), the network will be eventually consistent.
- **Blackhole freedom** No packet should be blackholed during updates. Blackholes occur if a packet arrives at a switch when there is no matching rule to handle it.
- **Packet coherence** The set of rules seen by a packet should not be a mix of old and new rules; they should be either all old or all new rules.
- **Congestion freedom** The amount of traffic arriving at a link should not exceed its capacity. Physical link capacity is a natural limit, but other limits may be interesting as well (e.g., margin for burstiness). Congestion freedom must be maintained without dropping traffic; otherwise, we can trivially meet any limit.

The consistency properties are listed in rough order of strength, and satisfying a property lower on the list often (but not always) satisfies a property above it. Obviously, packet coherence implies blackhole and loop freedom (assuming that the old and new rules sets are free of blackholes and loops). Perhaps less obviously, congestion freedom implies loop freedom because flows in a loop will likely surpass any bandwidth limit. Note that flows may be splittable [35].

However, these properties cannot be totally ordered. Packet coherence and congestion freedom are orthogonal, as packet coherence does not address congestion, and congestion freedom can be achieved with solutions beyond packet coherence. Blackhole freedom and loop freedom are also orthogonal. In fact, trivial solutions for one violates the other—dropping packets before they enter a loop guarantees loop freedom, and just sending packets back to the sender provides blackhole freedom but creates loops.

The columns in Table 9 denote dependency structures. They capture rules at which other switches must be updated before a new rule at a switch can be used safely. Thus, the dependency is at rule level, not switch level; dependencies are often circular at switch level—a rule on switch u depends on a rule on v, which in turn depends on u for other rules. The structures in Table 9 are:

- **None** The rule does not depend on any other update.
- **Self** The rule depends on updates at the same switch.
- **Downstream subset** The rule depends on updates at a subset of switches downstream for impacted packets.
- **Downstream all** The rule depends on updates at all

switches downstream for impacted packets.

- **Global** The rule depends on updates even at potentially all switches, including those that are not on the path for packets that use the rule.

These dependency structures are qualitative, not quantitative. For instance, they do not capture the time it might take for the update to complete. They also assume that switch resources, such as forwarding table memory or internal queues for unfinished updates, are not a bottleneck. Resource limitations induce additional dependencies on the order in which updates can be applied (see below).

In general, update procedures with fewer dependencies (i.e., to the left) are preferable. The cells in Table 9 denote whether a procedure exists to update the network with the corresponding consistency property and dependency structure. We can prove that certain combinations are impossible (proofs are in the Appendix). For example, packet coherence cannot be achieved in a way that rules depend on updates at only a subset of downstream switches.

As we can see, weaker consistency properties (towards the top of Table 9) need weaker dependency structures (towards the left). At one extreme, eventual consistency (i.e., no consistency during updates) has no dependencies at all. Slightly stronger properties, such as blackhole freedom, have dependencies on other rules at the switch itself. A simple procedure for blackhole freedom is to add the new rule in the switch before the old rule is removed. When installed with higher priority, the new rules become immediately usable, without wait.

At the other extreme, maintaining congestion freedom requires global coordination. The intuition here is that maintaining congestion freedom at a link requires coordinating all flows that use it, and some of these flows share links with other flows, and so on.

Interestingly, all cells to the immediate right of impossible cells are occupied in Table 9, which implies that, across past work and this paper, (qualitatively) optimal algorithms for maintain all these consistency properties are known. However, one must not infer from this observation that finding consistent update procedures is a "solved problem," for three reasons. First, some networks may need different properties, for which effective procedures or even best-case structures are unknown (e.g., load balancing across links and maintaining packet ordering within a flow).

Second, even for the properties in Table 9, the picture looks rosy partly because it assumes plentiful switch resources (e.g., forwarding table memory). If switch resources are constrained, maintaining consistency becomes harder. For instance, maintaining blackhole freedom with plentiful switch memory is straightforward and induces no dependencies across switches—we can just add all new rules with high priority before deleting any old rules. But in the presence of switch memory limits, this becomes challenging because introducing a new rule at a switch might require removing another rule first, which can only be removed after having added a new rule at some other switch.

In fact, we can show that in the presence of memory limits, even maintaining a simple property like blackhole freedom is NP-hard. Formally:

Problem 3: Let $c_i \in \mathbb{N}$ be the total interval rule memory of a switch v_i, the combined number of interval rules in current use and the interval rules it can receive in one update. Let $G = (V, E)$ be the directed graph on which packets can be routed, with the destinations $D \subseteq V$ and the sources $S \subseteq V$ for the packets. In one round, a central controller can send out a set of any interval rules as an update to each node in the network. What is the minimum number of rounds, to migrate the network from a set of blackhole free old rules to a new set of blockhole free rules, if no blackholes should be introduced during migration and routing should be possible at all times?

Theorem 3: Problem 3 is NP-hard.

Proof: The proof for Theorem 3 is based on a reduction from the NP-hard directed Hamiltonian Cycle problem (HC), cf. [30]: Given a directed graph $G = (V, E)$, is there a cycle that visits each node exactly once? The construction with further details is shown in Figure 10: It is possible to migrate blackhole free in two rounds if and only if there is a Hamiltonian Cycle in G, thus allowing to first use the cycle for intermediate routing via default rules, and then installing the new rules; Else it will take three rounds, one for each new rule. Thus, it is NP-hard to decide whether one can migrate in two or three rounds, even if the diameter is just two. The construction for the memory limit of $c = 4$ for all nodes in V can be directly extended to any $c \in \mathbb{N}$ with $c \geq 4$. ■

Furthermore, note that blackhole freedom is easy to guarantee for each node in the presence of default rules, if one does not care about routing: Just set a default rule to any neighboring node. While packets might not arrive at all (and in addition violate other consistency properties, e.g., congestion freedom), blackhole freedom is guaranteed.

Corollary 3: It is NP-hard to approximate the number of rounds needed for Problem 3 with an approximation ratio

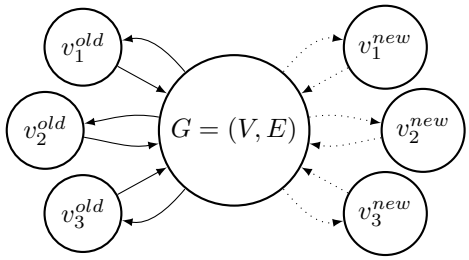

Fig. 10. The center node represents the graph $G = (V, E)$ from an instance I of the directed Hamiltonian Cycle problem, with nodes $v_1, \ldots, v_n$. The sets of edges to (n each) and from ($n/3$ each) the outer six nodes are bundled into single edges in this figure. Each node in $V = S = D$ has a memory limit c of four rules, with S being the set of packet sources and D being the set of packet destinations. The solid edges represent the edges used for the three old rules $\forall v \in V$, the dotted edges the edges used for the three new rules $\forall v \in V$. All nodes in V currently use the three nodes v_1^{old} (for $v_1, \ldots, v_{(n/3)}$), v_2^{old} ($v_{(n/3)+1}, \ldots, v_{(2n/3)}$), v_3^{old} ($v_{(2n/3)+1}, \ldots, v_n$) on the left for 2-hop routing to the respective destinations in $D = V$, and want to migrate to use the nodes v_1^{new} (for $v_1, \ldots, v_{(n/3)}$), v_2^{new} ($v_{(n/3)+1}, \ldots, v_{(2n/3)}$), v_3^{new} ($v_{(2n/3)+1}, \ldots, v_n$) on the right for 2-hop routing.

strictly better than $3/2$.

Third, the table only shows the qualitative part of the story, ignoring quantitative effects, which may be equally important. Even though [8] and [3] both have global dependencies, [8] can always resolve the dependencies in two rounds, whereas [3] may need more stages. Because of these three reasons, we believe that what is presented in this paper is just the tip of the iceberg for consistent updates in Software Defined Networks.

VII. An Architecture for SDN Updates

We have argued that maintaining consistency during rule updates is a key hurdle towards realizing the promise of SDNs. The question is: how can we accomplish this in a flexible, efficient manner? A straightforward possibility is that a single software module (controller) decides on new rules and then micro manages the update process in a way that maintains consistency. However, this monolithic architecture is undesirable because it mixes three separable concerns — i) the rule set should be policy-compliant; ii) rules updates should maintain the desired consistency property; iii) the update process should be efficient, which depends on the asynchronicity in the network.

We propose an alternative architecture (Figure 11) with three parts, one for each concern above: i) the *rule generator* produces policy-compliant rules; ii) the *update method selector* chooses the method of how to apply the rules, based on data from past updates; and iii) the *update executor* schedules the updates efficiently in a dynamic fashion, taking current asynchronicity into account.

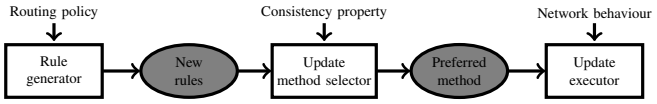

Fig. 11. Proposed dynamic architecture for SDN updates

The update method selector proceeds in two steps. It first generates, using the old rules and collected data from past updates of the network, a model of the current state of the network. This includes, e.g., the mean and variance of applying an update to a switch or the amount of unallocated memory/bandwidth. In the second step, multiple methods of applying the update are checked and simulated on the model of the network. Depending on the outcome, a preferred method for updates is selected: For example, if the current amount of free memory on switches is small, packet stamping is not a viable update method. However, if a long chain of links needs to be reversed loop free, and memory is not an issue, packet stamping might be the best way to proceed. In this step, it is also possible to issue *helper rules*, that are neither in the old or new set of rules, but allow consistent updates via a specific method. Consider the network in Figure 2: One can prevent loops by breaking a single (default) rule into one for each of the other destinations, introducing these rules during the update process and then removing them later.

The update executor computes a maximal set of updates that can be sent out immediately with the selected method, using the old rules, the new rules, and the desired consistency property. Once a set of nodes reported back on the successful implementation of the new rules, another batch of updates can be sent out into the network. Since the update process is a dynamic one, faulty nodes only induce a limited delay, independent parts of the network can still be updated. Nodes that did not report back yet have to be considered in a limbo state: Either they applied the new rules already or not, but to not break consistency properties, one has to assume that they are in both the new and the old state at the same time.

An example for an update executor would be Algorithm 3: Maximal sets of loop free updates are sent out each time nodes report back about the successful implementation of rules, inducing a minimal dependency structure in form of a dynamic dependency forest.

VIII. Evaluation

We took Rocketfuel ISP topologies with intra-domain routing weights [36] and considered link failures in these topologies, with our goal being loop free network updates from pre- to post-failure least-cost routing.

Figure 12 plots the distribution of the length of dependency chains that emerge across ten trials, where a randomly selected link was failed in each. We see that roughly half of the updates depended on 0 or 1 other switch, and 90% of all forwarding rules were dependent on at most 3 other switches. In contrast, had we used Reitblatt's procedure [8], which ensures the stronger property of packet coherence, rules would have had to wait for all other switches (well over a hundred in some cases), and a single slow switch can impede everyone.

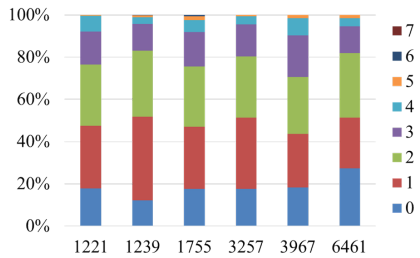

Fig. 12. Chain lengths in loop free updates in six Rocketfuel topologies. The x-axis label denotes the ASN.

Francois et al. [18] evaluated their work on a tier-1 ISP with 200 nodes and 800 links, resulting in chain lengths of 14. We had a chain length of at most 7, even for tier-1 ISPs such as ASN 1239 (Sprintlink) with 547 nodes and 1647 links.

IX. Summary

We argued that consistent updates in Software Defined Networks is an important and rich area for future research. We highlighted the trade-off between the strength of the consistency property and the dependency structure it imposes, and developed minimal algorithms for loop freedom. For the basic consistency properties of loop and blackhole freedom, we showed that fast updates are NP-hard optimization problems. We also sketched an architecture for consistent updates and showed that our loop freedom algorithm performs well in evaluations on ISP topologies.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their helpful comments, which helped us to improve the presentation of this paper. We would also like to thank Stefan Schmid and Stefano Vissicchio for pointing us to [15], [16], [17] shortly before this article was accepted for publication. Klaus-Tycho Förster was supported in part by Microsoft Research.

REFERENCES

[1] M. Borokhovich and S. Schmid, "How (Not) to Shoot in Your Foot with SDN Local Fast Failover," in *OPODIS*, 2013.
[2] M. Casado, N. Foster, and A. Guha, "Abstractions for software-defined networks," *Commun. ACM*, vol. 57, no. 10, pp. 86–95, Sep. 2014.
[3] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *SIGCOMM*, 2013.
[4] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hoelzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined WAN," in *SIGCOMM*, 2013.
[5] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *HotNets*, 2013.
[6] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe, "Design and implementation of a routing control platform," in *USENIX NSDI*, 2005.
[7] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and K. van der Merwe, "The Case for Separating Routing from Routers," in *SIGCOMM Workshop on Future Directions in Network Architecture (FDNA)*, 2004.
[8] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *SIGCOMM*, 2012.
[9] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent updates for software-defined networks: Change you can believe in!" in *10th ACM Workshop on Hot Topics in Networks*, 2011.
[10] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *HotSDN*, 2013.
[11] S. Brandt, K.-T. Förster, and R. Wattenhofer, "Augmenting anycast network flows," in *ICDCN*, 2016.
[12] X. Jin, H. Liu, R. Gandhi, S. Kandula, R. Mahajan, J. Rexford, R. Wattenhofer, and M. Zhang, "Dionysus: Dynamic Scheduling of Network Updates," in *SIGCOMM*, 2014.
[13] S. Brandt, K.-T. Förster, and R. Wattenhofer, "On Consistent Migration of Flows in SDNs," in *INFOCOM*, 2016.
[14] T. Mizrahi, O. Rottenstreich, and Y. Moses, "TimeFlip: Scheduling network updates with timestamp-based TCAM ranges," in *INFOCOM*, 2015.
[15] A. Ludwig, J. Marcinkowski, and S. Schmid, "Scheduling Loop-free Network Updates: It's Good to Relax!" in *PODC*, 2015.
[16] S. Vissicchio and L. Cittadini, "FLIP the (Flow) Table: Fast LIghtweight Policy-preserving SDN Updates," in *INFOCOM*, 2016.
[17] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid, "Transiently Secure Network Updates," in *Sigmetrics*, 2016.
[18] 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.
[19] 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.
[20] P. Francois and O. Bonaventure, "Loop-free convergence using oFIB," *Internet-Draft, IETF*, 2011.
[21] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure, "Lossless Migrations of Link-state IGPs," *IEEE/ACM Trans. Netw.*, vol. 20, no. 6, pp. 1842–1855, Dec. 2012.
[22] N. Santoro and R. Khatib, "Routing without routing tables," SCS-TR-6, Carleton University, Ottawa, Tech. Rep., 1982.
[23] C. Gavoille, "A survey on interval routing," *Theor. Comput. Sci.*, vol. 245, no. 2, pp. 217–253, 2000.
[24] M. Flammini, G. Gambosi, and S. Salomone, "Boolean Routing," in *WDAG*, 1993.
[25] P. Fraigniaud and C. Gavoille, "A characterization of networks supporting linear interval routing," in *PODC*, 1994.
[26] J. Van Leeuwen and R. B. Tan, "Interval routing," *The Computer Journal*, vol. 30, no. 4, pp. 298–307, 1987.
[27] D. Comer, Ed., *Internetworking with TCP/IP - Principles, Protocols, and Architectures, Fourth Edition*. Prentice-Hall, 2000.
[28] A. S. Tanenbaum and D. J. Wetherall, *Computer Networks*, 5th ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2010.
[29] V. Fuller and T. Li, "RFC 4632, Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan," 2006.
[30] M. R. Garey and D. S. Johnson, *A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.
[31] 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.
[32] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.
[33] A. Srinivasan, "Improved approximations of packing and covering problems," in *STOC*, 1995.
[34] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1994.
[35] R. Cohen and G. Nakibly, "Maximizing restorable throughput in mpls networks," *IEEE/ACM Trans. Netw.*, vol. 18, no. 2, pp. 568–581, 2010.
[36] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson, "Inferring link weights using end-to-end measurements," in *IMW*, 2002.

X. APPENDIX FOR SECTION 5

Proof of Lemma 1:
We start with loop freedom: The invariant of the algorithm is that the current edges in the network are without loops. The invariant is true at the beginning, since no new edge is included, and the old edges form an in-tree to the destination d. When a new egde $(u,v)_d$ is added, a now existing loop must contain this edge, i.e., there is a path from v to u. If a DFS starting at v cannot reach u, then there is no path from v to u, and the network is loop free. We now look at maximality: The algorithm checks each edge once if it can be added without inducing a loop. Consider an edge $e = (x,y)_d$, that is being tested w.l.o.g. as the i-th edge, but cannot be added to the network, because it would induce a loop $x, y, z, \ldots, x$. If e is being tested again after the $(j-1)$-th edge, with $i < j$, could e be added to a loop free network without inducing a loop in the network? No, because it would still induce the same loop, as edges were never removed, only possibly added. ■

XI. APPENDIX FOR SECTION 6

Lemma 5: Loop freedom depends on other nodes.

Proof: In Figure 1, node x depends on node y. ■

Lemma 6: Packet coherence depends on all non-trivial downstream switches.

Proof: Let u be a switch router that is non-trivial, in the sense that u is affected by a rule change, i.e. u's old rule differs from its new rule. If the source starts to route packets according to the new rule, switch u will forward the packets wrongly, or drop them, which is not packet coherent. ■

Lemma 7: Congestion freedom depends on all switches.

Proof: Let f be a flow that wants to use a new path p, or increase its capacity on an existing path. The network may be able to adapt to flow f, however, only if other flows use different paths as well, which in turn may (recursively) move even other flows (some of which have no single switch/link in common with the new path p). As such, any f may potentially depend on any single switch in the network. ■