On Consistent Updates in Software Defined Networks

Ratul Mahajan Roger Wattenhofer
Microsoft Research ETH Zurich

Abstract— We argue for the development of efficient methods to update the data plane state of an SDN, while maintaining desired consistency properties (e.g., no packet should be dropped). We highlight the inherent trade-off between the strength of the consistency property and dependencies it imposes among rules at different switches; these dependencies fundamentally limit how quickly data plane can be updated. For one basic consistency property—no packet should loop—we develop an update algorithm that has provably minimal dependency structure. We also sketch a general architecture for consistent updates that separates the twin concerns of consistency and efficiency.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks]: Network Architecture and Design
General Terms: Algorithms

1. INTRODUCTION

From early papers on the topic (e.g., [3, 1, 4, 2]), 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) [3, 1] proposed a centralized control plane architecture that directly configured the data plane of routers in an autonomous system.

4D aimed to simplify network management [4]. It observed that the "data plane needs to implement, in addition to next-hop forwarding, functions such as tunneling, access control, address translation, and queuing." In today's networks, this "requires complex arrangements of commands to tag routes, filter routes, and configure multiple interacting routing processes, all the while ensuring that no router is asked to handle more routes and packet filters than it has resources to cope with." Based on this observation, it argues for centrally computing data plane state in a way that obeys all concerns.

Similarly, ETHANE reasoned that for simplified management of enterprise networks "policy should determine the path that packets follow" [2]. It then argued for SDNs because the requirements of network management "are complex and require strong consistency, making it quite hard to compute in a distributed manner." These promises have led to SDNs garnering a lot of attention from both researchers and practitioners.

However, as we gain more experience with this paradigm, a nuanced story is emerging. Researchers have shown that, even with SDNs, packets can take paths that do not comply with policy [10] and that traffic greater than capacity can arrive at a link [5]. 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* data plane state has been changed, but inconsistencies emerge *during* data plane state changes.

Since changes to data plane state, in response to failures, load changes, and policy changes, are an essential part of an operational network, so will be the inconsistencies. Thus, successful use of SDNs requires not only methods to compute policy-compliant data plane state but also methods to change that state in a way that maintains desired consistency properties.

This paper takes a broad view of this aspect of SDNs. The key to consistently updating a network is carefully coordinating rule updates across multiple switches. Atomically updating the rules of multiple switches is difficult and uncoordinated updates can lead to inconsistencies such as packet loops and drops. Thus, when a rule can be safely updated at a switch depends on what rules are present at other switches, and in some cases, these dependencies can be circular. By analyzing a spectrum of possible consistency properties and the dependency structures they induce, we find an inherent trade-off. Dependency structures are simpler for weaker consistency properties and are more intricate for stronger properties. For instance, simply ensuring that no packet is dropped does not induce any dependency amongst switches, but ensuring that no packet sees a mix of old and new rules [10] makes rules at a switch depend on all other switches in the network.

We also take a detailed view of a basic consistency property—no packet loops—and develop two new update algorithms that induce less intricate dependency structures than currently known algorithms [10]. One of our algorithms induces provably minimal dependencies.

Motivated by our analysis, we sketch a general architecture for consistent network updates. It decouples across two modules the two concerns that are primary during network updates: consistency and speed. Given the consistency property of interest, the first module computes a correct plan for updating the network. This plan is represented as a directed acyclic graph in which nodes are rule updates and edges represent their dependencies. The second module is responsible for quickly applying the plan based on the properties of the network, e.g. the time for individual switches to apply updates and distances between the controller and switches. Preliminary experimental results highlight the value of this architecture as well as open challenges.

2. LOOP FREEDOM

To expose the nuances of consistently updating SDNs, we focus first on a basic consistency property—loop freedom. As the name suggests, it implies that no packet should loop in the network. In §3, we will analyze a broader set of consistency properties.

Consider the five-node network in Figure 1. Assume that we want to update the routing to Node d from the pattern on the left to that on the right. A naive method is to send out all forwarding 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.

Reitblatt et al.'s procedure [10] can provide loop freedom during updates. It relies on ingress nodes stamping packets with version numbers. Assuming the version currently being stamped is k, the procedure works as follows: *i*) send new rules at each node applicable to packets with version $k + 1$; *ii*) after all nodes have acknowledged that they have updated, ask the ingress nodes to switch to version $k + 1$; *iii*) after waiting for a time during which all version k packets should have left the network, delete the old rules.

This procedure is onerous because it requires that all nodes be updated (in step *i*) before new rules become usable (in step *ii*), which means that delays in updating even one node will delay the update. However, it also guarantees a consistency property that is stronger than loop freedom—each packet is routed entirely using the old rules or the new rules, and never a mix of the two sets. We call this property packet coherence in §3.

A natural question is: if we want only loop freedom (not packet coherence), is there an update procedure that does not rely on updating all nodes before starting to use any of the new forwarding rules?[1] This is not a technicality, as nodes in a production network can often react slowly, or they may even be temporarily unreachable via the controller [6]. Thus, solutions in which the network can quickly start using as many of the new forwarding rules as possible, while maintaining the consistency property, are preferable.

A related, fundamental question is: for a given consistency property, what is the minimal procedure? This is the question we raise in this paper. While §3 contains a broader discussion, we describe below two update procedures for loop freedom. Both have looser dependency requirements than Reitblatt et al.'s procedure; the first one is simpler and the second one is in fact minimal. Vanbever et al. [12] work on a related problem, and study the migration of a conventional (non-SDN) network to a new IGP protocol. The main differences in the two settings arise from the fact that they focus on updating an entire node (i.e., all its forwarding entries), while we can update individual forwarding entries.

2.1 A simple procedure

Our first procedure can be understood in terms of a *dependency tree* in which a node can safely update to new rules after its parent has switched. Thus, a node only depends on its ancestors, and any slowdowns elsewhere in the network have no impact on its ability to switch. For simplicity, we first describe our procedures as if there was a single destination d, and then discuss the case of multiple destinations. A valid dependency tree is, for instance, the destination rooted in-tree with respect to the new set of rules. In this tree, the destination is the root, and a node c is a child of p iff the new rule of c points to p. In Figure 1, this is basically the tree shown on the right.

A simple update procedure then is: start with the root of a destination-rooted dependency tree and successively update the children, pursuing the branches in parallel. This procedure guarantees loop freedom because nodes in the dependency tree of the destination will switch to the new rule only after all downstream nodes along the new path have switched.

2.2 A minimal procedure

While correct, the procedure above is not minimal. Observe that v can switch to new rules immediately, irrespective of whether x (its parent in the destination tree) is updated; no matter what, packets will end up at x, with no possibility to experience a loop.

One may wish for the fastest procedure, in the sense that dependencies are minimum. Surprisingly, this is not trivial. Consider the example in Figure 2, again with the old (new) rules on the left (right). 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. However, either one is fine, i.e., either u waits for w and v, w may switch immediately, *or* v waits for w and u, w may switch immediately. In other words, there are two valid dependency trees, one

[1]Interestingly, a majority of the motivating examples in [10] do not need packet coherence, but need only loop freedom.

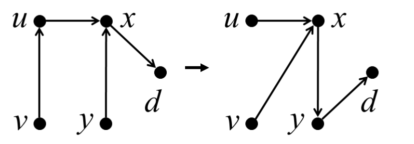

Figure 1: Illustrating loop freedom (§2).

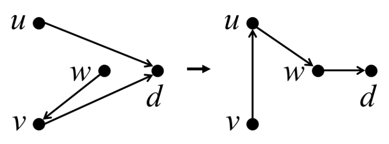

Figure 2: Illustrating multiple minimal solutions (§2.2).

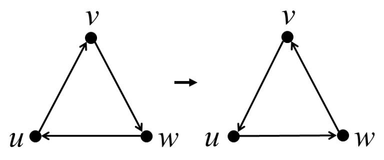

Figure 3: Illustrating circular dependencies in prefix routing (§2.3).

that is fast node u (at the expense of v), and one which is fast for v (at the expense of u). Thus, *the minimum* solution does not exist, and we must instead look for *a minimal* solution in which no node can improve its dependencies without some other node getting worse.

In designing such a solution, we observe that there may be many nodes that can switch immediately. In Figure 1, v and y can switch immediately, for different reasons. We thus expand the concept of a dependency tree into a *dependency forest*. As before, children wait for their parents before switching, but we now have multiple roots and trees. Our goal is to look for a dependency forest that is minimal in the sense that one cannot attach a node at an ancestor of its current parent.

In our algorithm, each node is in one of three states, *old*, *new*, or *limbo*, depending on whether the node is guaranteed to only use its old rule or new rule, or whether it might use both rules, respectively. Since the destination d does not have any rules, neither old nor new, it is by definition in state new.

We construct the dependency forest as follows. We start out with only the old rules, by definition a loop-free in-tree to destination d. Now, for each node u, we test whether adding u's new rule will introduce a loop, using a loop-detection subroutine (described below). If not, node u is entering the state limbo, and added as a root in the dependency forest. On the other hand, if u's new rule introduces a loop, node u remains old, and must wait until we find its parent in the dependency forest. The initialization is completed after we processed all the nodes, and found all the roots (which are now leaves in our incomplete dependency forest). Next we add the children to the dependency forest, one after another. In each step, we choose a limbo (dependency forest leaf) node u. We remove u's old rule from the network (putting u in the new state), and then check upon all the old nodes, trying to find nodes v where the new rule of v does not introduce a loop (using the same loop-detection subroutine), thanks to removing u's old rule. If we find such a node v, then node v is a child of u in the dependency forest, and a new leaf of the dependency forest, in limbo state. If no such node v exists, node u remains a childless leaf of the dependency forest. We can show that at least one old node is eligible for the limbo state, and as such we can always make progress until all nodes are in the new state.

The loop-detection subroutine can be implemented in various ways, e.g., using Tarjan's algorithm [11]. But this algorithm does not only detect loops; it finds strongly connected components. We employ a simpler solution for loop-detection. Our solution is recursive, starting at the new rule $u.new = v$. Nodes are in one of three states, *unknown*, *seen*, and *visited*. Both u, v are seen, all other nodes are unknown. Now we do a depth-first search (DFS), starting at v. If the DFS visits w, depending on the state of w, we do the following: if w is unknown, we mark w as seen and continue DFS. If w is visited, we backtrack DFS. If w is seen, we found a loop. When backtracking to w, we mark w as visited. If the DFS does not find a loop, and we backtrack all the way to the starting node v, the network is loop-free and we can safely add the new rule.

Once a dependency forest is computed, we update the individual dependency trees in parallel. For this procedure, we can prove that the computed dependency forest guarantees loop-freedom and is minimal [9].

2.3 Multiple destinations

While we described the procedures above in terms of a single destination node, their correctness and optimality hold in the presence of multiple destinations if rules are per-destination. This setting maps to layer 2 routing, which uses MAC addresses as destinations, or tunnel-based routing, which users tunnel identifiers as destinations. (Both large-scale SDNs described in recent literature use tunnel-based routing [5, 6].) Here, we can compute dependency trees or forests for each destination separately and apply updates in parallel.

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 prefix-based routing and longest-prefix matching. In this case, no (loop-free) dependency forests may exist. Consider, for instance, the network in Figure 3. Each node has two rules: one for itself as the destination and one default rule (which will cover the other two destinations). In the old routing, default rules point clockwise. In the new routing, they point counter-clockwise. Now, no matter which of the new default rules is changed first, we immediately cause a loop for some destination. We can capture such circular dependencies using *rule dependency graphs*. We

	None	**Self**	**Downstream subset**	**Downstream all**	**Global**
Eventual consistency	Always guaranteed				
Drop freedom	Impossible	Add before remove			
Memory limit	Impossible	Remove before add			
Loop freedom	Impossible (Lemma 6)		Rule dep. forest (§2.2)	Rule dep. tree (§2.1)	
Packet coherence	Impossible (Lemma 7)			Per-flow ver. numbers	Global ver. numbers [10]
Bandwidth limit	Impossible (Lemma 8)				Staged partial moves [5]

Table 1: Some basic consistency properties (rows) and their dependencies (columns). Proofs of lemmas are in [9].

discuss how to handle them in §4, after getting a better understanding of the consistency space.

3. CONSISTENCY SPACE

Thus far, we have focused on loop freedom; we now take a broader view of the range of consistency properties. Table 1 helps frame this view. Its rows correspond to consistency properties. We defined loop freedom and packet coherence in §2; the others are:

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.

Drop freedom No packet should be dropped during update. Drops will occur if a switch lacks a rule to handle a packet and, for scalability, it is not configured to send unmatched packets to the controller [5, 6].

Memory limit The number of rules that a switch is required to hold is always below a certain limit. A natural limit is the physical capacity of the flow table, but other limits may also be enforced.

Bandwidth limit The amount of traffic arriving at a link should not exceed a certain limit. Physical link capacity is a natural limit, but other limits may be interesting as well (e.g., margin for burstiness). The limit must be maintained without dropping traffic; otherwise, we can trivially meet any limit.

The consistency properties we list are not the only ones of interest. Some networks may require different properties (e.g., balanced load across two links), and some others may require guarantees that combine two or more properties (e.g., packet coherence + memory limits [7]). We chose these consistency properties because they are basic and natural, capturing the basic expectations of the experience of packets and network elements.

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 drop and loop freedom (assuming that the old and new rules sets are free of drops and loops). Perhaps less obviously, bandwidth limits imply loop freedom because flows in a loop will likely surpass any bandwidth limit.

However, these properties cannot be totally ordered. Packet coherence and bandwidth limits are orthogonal, as packet coherence does not address bandwidth, and bandwidth limits can be achieved with solutions beyond packet coherence. Drop 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 drop freedom but creates loops.

The columns in Table 1 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 the 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. Further, the dependency captures when a new rule can be installed and used safely, not when an old rule can be safely removed. Even after all new rules are being used, the rule set in the network may not be the same as the new rule set; additional (unused) rules may still exist. Such rules will be removed in a clean-up phase. The safe usage time of a new rule is important because it determines when the network is carrying traffic in the new pattern (which may have been necessitated by a failure).

The different structures in Table 1 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 the switches that lie downstream for impacted packets.

Downstream all The rule depends on updates at all switches that lie downstream for impacted packets.

Global The rule depends on updates at potentially all switches, even those not on the path for impacted packets.

These dependency structures are qualitative, not quantitative (e.g., time it takes for the update), but in general, update procedures with fewer dependencies (i.e., to the left) are preferable. The cells in Table 1 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 [9]. 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) need weaker dependency structures (towards the left). At one extreme, eventual consistency has no dependencies. Slightly stronger properties, drop freedom and memory limit, have dependencies on other rules at the switch itself. A simple procedure for drop 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. A simplistic method for maintaining memory limits is to remove an old rule on the switch before adding the new rule. But this method may cause drops or loops.

At the other extreme, maintaining bandwidth limit requires global coordination. The intuition here is that maintaining bandwidth limits at a link requires coordinating all flows that use it, and some of these flows share links with other flows, and so on. Hong et al. [5] describe a procedure to effect such transitions by moving flows partially across multiple stages.

Interestingly, all cells to the immediate right of impossible cells are occupied, which implies that, across past work and this paper, qualitatively optimal algorithms for maintaining all these consistency properties are known. However, this does not imply 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 1, the picture looks rosy partly because the table focuses on consistency properties in isolation. The combinations are hard to ensure, and efficient algorithms are not known. For instance, drop freedom and memory limit, while easy to ensure individually, are challenging to ensure in combination. Maintaining the combination requires global dependencies, as introducing some rule at a switch might need to remove another rule first, which can only be removed after having added a new rule somewhere else.

Third, the table only shows the qualitative part of the story and ignores quantitative effects that may be equally important. Even though [10] and [5] both have global dependencies, [10] can resolve the dependencies in two rounds, whereas [5] may need more stages. Because of these reasons, what is presented in this paper is just the tip of iceberg for consistent updates in SDNs.

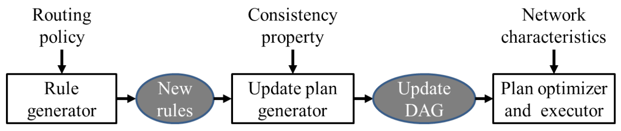

Figure 4: Proposed architecture

4. 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 for the same software module (controller) to decide on new rules and then micro manage the update process in a way that maintains consistency. This monolithic architecture is undesirable because it mixes three separable concerns— *i*) the rule set should policy-compliant; *ii*) rules updates should maintain the desired consistency property; *iii*) the updates process should be efficient, which depends intimately on network characteristics (e.g., the mean and variance of applying an update to a switch).

We propose an alternative architecture (Figure 4) with three parts, one for each concern above: *i*) the *rule generator* produces policy-compliant rules; *ii*) the *update plan generator* produces a plan for applying those rules in a way that ensures consistency; and *iii*) the *plan optimizer and executor* executes the plan efficiently.[2]

We represent the update plan using an *update DAG* (directed acyclic graph) in which nodes are updates (i.e., rule additions, deletions, or changes) and directed edges are dependencies between updates. Updates that do not have an inbound edge can be implemented immediately.

Using only updates as nodes does not capture all types of dependencies. Safe application of some updates requires time delays; e.g., in [10], rules with old version numbers can be removed only after time to drain in-transit packets has elapsed. Further, some updates may have dependencies such that they can be applied when any one of their parents have been applied; e.g., switch the rules of node u in Figure 2 such that $u.old = w$ and $u.new = d$. Then, in order to prevent a loop, node v must wait for either u *or* w, not both.

To generally handle such dependencies, we introduce *combinator nodes* into the update DAG. Current combinators include delay and logical functions. A delay combinator is considered applied (i.e., its dependent updates can be now applied) after the specified time has elapsed since its parent updates were applied. A logical combinator is a logical function (e.g., AND or OR) over the binary state (applied or not) of its parents. It is

[2]In analogy to programs, compilers, and runtimes, the first part declaratively specifies the desired network state, the second produces instructions to safely update the network, and the third part executes the instructions efficiently, taking into account the properties of the underlying hardware.

considered applied when the function evaluates to true. These two combinators suffice to efficiently represent update plans for all procedures in Table 1; future work may uncover the need for additional combinators.

The update plan generator proceeds in two steps. It first computes, using the old rules, new rules, and the desired consistency property, a *rule dependency graph* where nodes correspond to deletion of old rules or addition of new rules. It then converts this graph into an update DAG such that, starting from the old rules, applying the DAG leads to new rules.

This second step is straightforward if the graph is cycle-free; otherwise, we must break cycles. How this should be done depends on the consistency property, but a few general tools exist. One is using version numbers [10], which help when a new rule must wait for an old rule to be removed. Introducing version numbers makes it clear which of the (otherwise) conflicting rules should be used. As an extension, if one is not willing to wait for a rule to be inserted, additional information (similar to source routing) may be embedded in the packet as to how it should be handled downstream and the circular dependency vanishes.

Another tool for breaking cycles is using *helper updates*, which exist in neither the old or new rule sets but help with consistent updates. SWAN's staged partial moves [5] can be expressed using helper updates that move subsets of flows to avoid overloading any link. Circularity due to prefix-based routing (§2.3) can also be broken using helper updates. In Figure 3, we can eliminate the cycle by breaking a single (default) rule into one for each of the two destinations covered by the default rule, introducing these rules during the update process and then removing them later.

The plan executor applies the update DAG correctly and quickly. In deciding the order and timing of updates, it needs to factor in several concerns, including the delays from the controller to the switch, the mean and variance of the time a switch takes to apply an update, and limiting load on a switch. Further, in some cases, the update DAG may contain a long chain of dependencies. In the worst case, with n nodes in the network, the chain can be $O(n)$ long (e.g., ensuring loop freedom for changing direction in a ring network.) Long chains may be shortened, for instance, using version numbers. Alternatively, we may also introduce a new primitive in switches by which a switch informs its dependents when it is safe to apply an update, which would enable updates to be done without $O(n)$ round trip exchanges with the controller. Developing efficient algorithms for applying update DAGs, while accounting for all these concerns, is a rich area for future work.

Sometimes, in the middle of forwarding state changes, we may need to change the network to yet another state (e.g., due to a failures). Such events are easier to handle in our architecture. The rule generator can compute the new state of the network, without worrying about the current, transient state. The plan executor knows the current state of the network, with some small uncertainty that corresponds to update messages that have been sent to the switches but have not been acknowledged. Using this state as the starting point, the plan generator can generate a new update DAG, which may cancel out updates in the old DAG. Then, the plan executor can start applying this new update DAG.

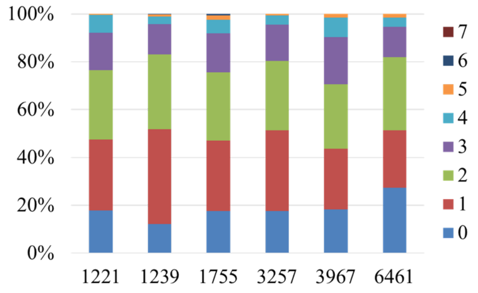

Figure 5: Chain lengths in update DAGs in six Rocketfuel topologies. The x-axis label denotes the ASN.

5. PRELIMINARY EVALUATION

We have a preliminary implementation of the architecture above. Our focus thus far has been loop freedom and the update DAGs that emerge for it. In one experiment, we took Rocketfuel ISP topologies with intradomain routing weights [8]. We considered link failures in these topologies, and our goal was loop free network updates from pre- to post-failure least-cost routing.

Figure 5 plots the distribution of the length of dependency chains that emerge across ten trials, where a randomly selected link was failed in each, and update DAGs were computed using the procedure in §2.2. We see that roughly half of the updates depend on 0 or 1 other switch, and 90% of all rules are dependent on at most 3 other switches. In contrast, had we used Reitblatt's procedure [10], 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.

We also see a small fraction of cases with chain lengths greater than 5. These are prime candidates for implementation through localized chain shortening optimizations by the plan executor.

6. SUMMARY

We argued that consistent updates in SDNs 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. We also sketched an architecture for consistent and quick updates in SDNs.

7. REFERENCES

[1] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *NSDI*, 2005.

[2] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Mckeown, and S. Shenker. ETHANE: taking control of the enterprise. In *SIGCOMM*, 2007.

[3] 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.

[4] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. In *SIGCOMM CCR*, 2005.

[5] 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.

[6] 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 *SIGCOMM*, 2013.

[7] N. P. Katta, J. Rexford, and D. Walker. Incremental consistent updates. In *HotSDN*, 2013.

[8] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. Inferring link weights using end-to-end measurements. In *Internet Measurement Workshop*, 2002.

[9] R. Mahajan and R. Wattenhofer. On consistent updates in software defined networking (extended version). Technical Report MSR-TR-2013-99, Microsoft Research, 2013.

[10] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, 2012.

[11] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[12] L. Vanbever, S. Vissicchio, C. Pelsser, P. François, and O. Bonaventure. Lossless migrations of link-state IGPs. *IEEE/ACM Trans. Netw.*, 2012.