Survey of Consistent Network Updates

Article · September 2016

CITATIONS
8

READS
43

3 authors:

Klaus-Tycho Foerster
University of Vienna
32 PUBLICATIONS **97** CITATIONS
SEE PROFILE

Stefan Schmid
Aalborg University
40 PUBLICATIONS **305** CITATIONS
SEE PROFILE

Stefano Vissicchio
40 PUBLICATIONS **425** CITATIONS
SEE PROFILE

Survey of Consistent Network Updates

Klaus-Tycho Foerster[1] Stefan Schmid[2] Stefano Vissicchio[3]
[1] ETH Zurich, Switzerland [2] Aalborg University, Denmark [3] University of Louvain (UCL), Belgium

ABSTRACT

Computer networks have become a critical infrastructure. Designing dependable computer networks however is challenging, as such networks should not only meet strict requirements in terms of correctness, availability, and performance, but they should also be flexible enough to support fast updates, e.g., due to a change in the security policy, an increasing traffic demand, or a failure. The advent of Software-Defined Networks (SDNs) promises to provide such flexiblities, allowing to update networks in a fine-grained manner, also enabling a more online traffic engineering.

In this paper, we present a structured survey of mechansims and protocols to update computer networks in a fast and consistent manner. In particular, we identify and discuss the different desirable update consistency properties a network should provide, the algorithmic techniques which are needed to meet these consistency properties, their implications on the speed and costs at which updates can be performed. We also discuss the relationship of consistent network update problems to classic algorithmic optimization problems. While our survey is mainly motivated by the advent of Software-Defined Networks (SDNs), the fundamental underlying problems are not new, and we also provide a historical perspective of the subject.

Keywords

Network Updates, Algorithms, Software-Defined Networks, OpenFlow, NP-Hardness, TCAMs

1. INTRODUCTION

Computer networks such as datacenter networks, enterprise networks, carrier networks etc. have become a critical infrastructure of the information society. The importance of computer networks and the resulting strict requirements in terms of availability, performance, and correctness however stand in contrast to today's ossified computer networks: the techniques and methodologies used to build, manage, and debug computer networks are largely the same as those used in 1996 [4]. Indeed, operating traditional computer networks is often a cumbersome and error-prone task, and even tech-savvy companies such as GitHub, Amazon, GoDaddy, etc. frequently report issues with their network, due to misconfigurations, e.g., resulting in forwarding loops [25, 34, 74, 90]. Another anecdote reported in [4] illustrating the problem, is the one by a Wall Street investment bank: due to a datacenter outage, the bank was suddenly losing millions of dollars per minute. Quickly the compute and storage emergency teams compiled a wealth of information giving insights into what might have happened. In contrast, the networking team only had very primitive connectivity testing tools such as ping and traceroute, to debug the problem. They could not provide any insights into the actual problems of the switches or the congestion experienced by individual packets, nor could the team create any meaningful experiments to identify, quarantine and resolve the problem [4]. Given the increasing importance computer networks play today, this situation is worrying.

Software-defined networking is an interesting new paradigm which allows to operate and verify networks in a more principled and formal manner, while also introducing flexibilities and programmability, and hence faster innovations. In a nutshell, a Software-Defined Network (SDN) outsources and consolidates the control over the forwarding or routing devices (located in the so-called *data plane*) to a logically centralized controller software (located in the so-called *control plane*). This decoupling allows to evolve and innovate the control plane independently from the hardware constraints of the data plane. Moreover, OpenFlow, the de facto standard SDN protocol today, is based on a simple match-action paradigm: the behavior of an OpenFlow switch is defined by a set of forwarding rules installed by the controller. Each rule consists of a match and an action part: all packets matched by a given rule are subject to the corresponding action. Matches are defined over Layer-2 to Layer-4 header fields (e.g., MAC and IP addresses, TCP ports, etc.), and actions typically describe operations such as forward to a specific port, drop, or update certain header fields. In other words, in an SDN/OpenFlow network, network devices become simpler: their behavior is defined by a set of rules installed by the controller. This enables formal reasoning and verification, as well as flexible network update, from a logically centralized perspective [43, 46]. Moreover, as rules can be defined over multiple OSI layers, the distinction between switches and routers (and even simple middleboxes [17]) becomes blurry.

However, the decoupling of the control plane from the data plane also introduces new challenges. In particular, the switches and controllers as well as their interconnecting network form a complex asynchronous distributed system. For example, a remote controller may learn about and react to network events slower (or not at all) than a hardware device in the data plane: given a delayed and inconsistent view, a controller (and accordingly the network) may behave in an undesirable way. Similarly, new rules or rule updates communicated from the controller(s) to the switch(es) may take effect in a delayed and asynchronous manner: not only because these updates have to be transmitted from the controller to the switches over the network, but also the reaction time of the switches themselves may differ (depending on the specific hardware, data structures, or concurrent load).

Thus, while SDN offers great opportunities to operate a network in a correct and verifiable manner, there remains a fundamental challenge of how to deal with the asynchrony inherent in the communication channel between controller and switches as well as in

the switches themselves. Accordingly, the question of how to update network behavior and configurations correctly yet efficiently has been studied intensively over the last years. However, the notions of correctness and efficiency significantly differs across the literature. Indeed, what kind of correctness is needed and which performance aspects are most critical often depends on the context: in security-critical networks, a very strong notion of correctness may be needed, even if it comes at a high performance cost; in other situations, however, short transient inconsistencies may be acceptable, as long as at least some more basic consistency guarantees are provided (e.g., loop-freedom).

We observe that not only the number of research results in the area is growing very quickly, but also the number of models, the different notions of consistency and optimization objectives, as well as the algorithmic techniques. Thus, it has become difficult to keep an overview of the field even for active researchers. Moreover, we observe that many of the underlying problems are not entirely new or specific to SDN: rather, similar consistency challenges arose and have been studied already in legacy networks, although update algorithms in legacy protocols are often more distributed and indirect (e.g., based on IGP weights).

Accordingly, we believe that it is time for a comprehensive survey of the subject.

1.1 The Network Update Problem

Any dependable network does not only need to maintain a range of static invariants, related to correctness, availability, and performance, but also needs to be flexible and support reconfigurations and updates. Reasons for updating a network are manifold, including:

1. *Change in the security policy:* Due to a change in the enterprise security policy, traffic from one subnetwork may have to be rerouted via a firewall before entering another subnetwork. Or, in the wide-area network, the set of countries via which it is safe to route sensitive traffic may change over time.

2. *Traffic engineering:* In order to improve traffic engineering metrics (e.g., minimizing the maximal link load), a system administrator or operator may decide to reroute (parts of) the traffic along different links. For example, many Internet Service Providers switch between multiple routing patterns during the day, depending on the expected load. These patterns may be precomputed offline, or may be computed as a reaction to an external change (e.g., due to a policy change of a Content Distribution Provider).

3. *Maintenance work:* Also maintenance work may require the update of network routes. For example, in order to replace a faulty router, or to upgrade an existing router, it can be necessary to temporarily reroute traffic.

4. *Link and node failures:* Failures happen quite frequently and unexpectedly in today's computer networks, and typically require a fast reaction. Accordingly, fast network monitoring and update mechanisms are required to react to such failures, e.g., by determining a failover path.

Despite these changes, it is often desirable that the network maintains certain minimal consistency properties, *during the update*. For example, per-packet consistency (a packet should be forwarded along the old or the new route, but never a mixture of both), loop-freedom (at no point in time are packets forwarded along a loop), or waypoint enforcement (a packet should never bypass a firewall).

Moreover, while the reasons for network updates identified above are general and relevant in any network, both software-defined and traditional, we believe that the flexibilities introduced by programmable networks are likely to increase the frequency of network updates, also enabling, e.g., a more fine-grained and online traffic engineering [35].

1.2 Our Contributions

This paper presents a comprehensive survey of the consistent network update problem. We identify and compare the different notions of consistency as well as the different performance objectives considered in the literature. In particular, we provide an overview of the algorithmic techniques required to solve specific classes of network update problems, and discuss inherent limitations and trade-offs between the achievable level of consistency and the speed at which networks can be updated. In fact, as we will see, some update techniques are not only less efficient than others, but with them, it can even be impossible to consistently update a network.

While our survey is motivated by the advent of Software-Defined Networks (SDNs), the topic of consistent network updates is not new, and for example, guaranteeing disruption-free IGP operations has been considered in several works for almost two decades. Accordingly, we also present a historical perspective, surveying the consistency notions provided in traditional networks and discussing the corresponding techniques accordingly. Moreover, we put the algorithmic problems into perspective and discuss how these problems relate to classic optimization and graph theory problems, such as multi-commodity flow problems or maximum acyclic subgraph problems.

The goal of our survey is to (1) provide active researchers in the field with an overview of the state-of-the-art literature, but also to (2) help researchers who only recently became interested in the subject to bootstrap and learn about open research questions.

1.3 Paper Organization

The remainder of this paper is organized as follows. §2 presents a historical perspective and reviews notions of consistency and techniques both in traditional computer networks as well as in Software-Defined Networks. §3 then presents a classification and taxonomy of the different variants of the consistent network update problems. §4, §5, and §6 review models and techniques for connectivity consistency, policy consistency, and performance consistency related problems, respectively. §7 discusses proposals to further relax consistency guarantees by introducing tighter synchronization. In §8, we identify practical challenges. After highlighting future research directions in §9, we conclude our paper in §10.

2. THE NETWORK UPDATE PROBLEM FROM THE ORIGINS TO SDN

Any computer network needs to provide basic mechanisms and protocols to change forwarding rules and network routes, and hence, the study of consistent network updates is not new and the topic to some extent evergreen. For example, a forwarding loop can quickly deplete switch buffers and harm the availability and connectivity provided by a network, and protocols such as the Spanning Tree Protocol (STP) have been developed to ensure loop-free layer-2 forwarding at any time. However, consistency problems may also arise on higher layers in the OSI stack.

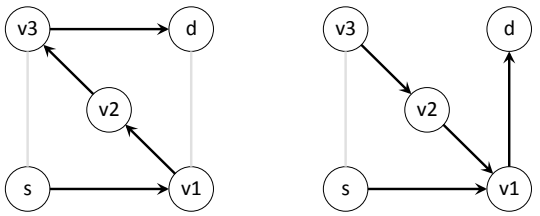
In this section, we provide a historical perspective on the many research contributions that lately focused on guaranteeing consistency properties during network updates, that is, while changing device configurations (and how they process packets).

We first discuss update problems and techniques in traditional networks (§2.1-2.2). In those networks, forwarding entries are computed by routing protocols that run standardly-defined distributed

algorithms, whose output is influenced by both network topology (e.g., active links) and routing configurations (e.g., logical link costs). Pioneering works then aimed at avoiding transient inconsistencies due to modified topology or configurations, mainly focusing on IGPs, i.e., the routing protocols that control forwarding within a single network. A first set of contributions tried to modify IGP protocol definitions, mainly to provide forwarding consistency guarantees upon link or node failures. Progressively, the research focus has shifted to a more general problem of finding a sequence of IGP configuration changes that lead to new paths while guaranteeing forwarding consistency, e.g., for service continuity (§2.1). More recent works have also considered reconfigurations of protocols different or deployed in addition to IGPs, mostly generalizing previous techniques while keep focusing on forwarding consistency (§2.2).

Subsequently (§2.3), we discuss update problems tackled in the context of logically-centralized networks, implementing the Software Defined Networking (SDN) paradigm. SDN is predicated around a clear separation between controller (implementing the control logic) and dataplane elements (applying controller's decision on packets). This separation arguably provides new flexibility and opens new network design patterns, for example, enabling security requirements to be implemented by careful path computation (done by the centralized controller). This also pushed network update techniques to consider additional consistency properties like policies and performance.

We rely on the generic example shown in Fig. 1 for illustration. The figure shows the intended forwarding changes to be applied for a generic network update. Observe that possible forwarding loops can occur during this update because edges (v_1, v_2) and (v_2, v_3) are traversed in opposite directions before and after the update.



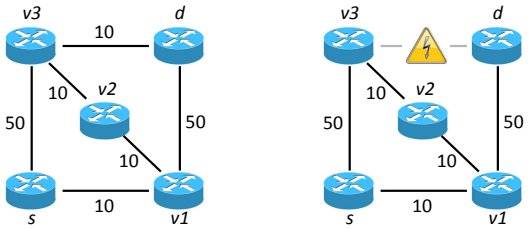
(a) Surpassed state (b) Down state

Figure 1: A network update example, where forwarding paths have to be changed from the Surpassed (Fig. 1a) to the Down (Fig. 1b) state. Arrows represent paths on which traffic (e.g., from s to d) is forwarded, while (gray) undirected edges between nodes represent unused links.

2.1 IGP Reconfigurations

In traditional (non-SDN) networks, forwarding paths are computed by distributed routing protocols. Link-state Interior Gateway Protocols (IGPs) are the most popular of those protocols used to compute forwarding paths within a network owned by the same administrative entity. Link-state IGPs are based on computing shortest-paths on a weighted graph, representing a logical view of the network, which is shared across routers. Parameters influencing IGP computations, like link weights of the shared graph, are set by operators by editing router configurations.

As an illustration, Fig. 2 shows a possible implementation for the update example presented in Fig. 1. In particular, Fig. 2 reports the IGP graph (consistent with the physical network topology) with explicit mention of the configured link weights. Based on those weights, for each destination (e.g., d in this example), all routers independently compute the shortest paths, and forward the corresponding packets to the next-hops on those paths. Consequently, the IGP configurations in Figs. 2a and 2b respectively produce the forwarding paths depicted in Figs. 1a and 1b.



(a) Surpassed state (b) Down state

Figure 2: Possible implementation of pre- and post-update forwarding paths for the update in Fig. 1 in a traditional, IGP-based network. Numbers close to network links represent the corresponding IGP weights.

When the IGP graph is modified (e.g., because of a link failure, a link-weight change or a router restart), messages are propagated by the IGP itself from node to node, so that all nodes rebuild a consistent view of the network: This process is called *IGP convergence*. However, IGPs do not provide any guarantee on the time and order in which nodes receive messages about the new IGP graphs. This potentially triggers transient forwarding disruptions due to temporary state inconsistency between a set of routers. For example, assume that we simply remove link (v_3, d) from the IGP graph shown in Fig. 2a. This will eventually lead us to the configuration presented in Fig. 2b. Before the final state is reached, the notification that (v_3, d) is removed has to be propagated to all routers. If v_3 receives such notification before v_2 (e.g., because closer to the removed link), then v_3 would recompute its next-hop based on the new information, and starts forwarding packets for d to v_2 (see Fig. 1b). Nevertheless, v_2 keeps forwarding packets to v_1 as it still forwards as (v_3, d) is still up. This creates a loop between v_3 and v_2: The loop remains until v_2 is notified about the removed link. A similar loop can occur between v_2 and v_1.

Guaranteeing disruption-free IGP operations has been considered by research and industry since almost two decades. We now briefly report on the main proposals.

Disruption-free IGPs have been studied. Early contributions focused on modifying the routing protocols, mainly to avoid forwarding inconsistencies. Among them, protocol extensions have been proposed [76, 86, 88] to gracefully restart a routing process, that is, to avoid forwarding disruptions (e.g., blackholes) during a software update of a router. Other works focused on preserving forwarding consistency, that is, avoiding loops, upon network failures. For example, François *et al.* [21] propose oFIB, an IGP extension that guarantees the absence of forwarding loops after topological changes (link/node addition or removal). The key intuition behind oFIB is to use explicit synchronization between routers in order to constrain the order in which each node changes its forwarding entries. In particular, each router (say, v_2 in our example) is forced not to update its forwarding entry for a given destination (d in our example) until all its final next-hops (v_1) use their own final next-hops (d in our case). Fu *et al.* [24] generalize the previous approach by defining a loop-free ordering of IGP-entry updates for arbitrary forwarding changes. Moreover, PLSN [87] specializes oFIB: It allows routers to dynamically avoid loops by locally delaying forwarding changes that are not safe. A variant of oFIB, studied by Shi *et al.* [89], also

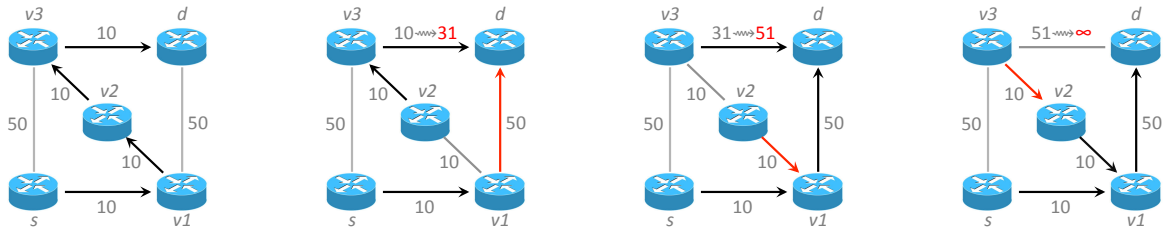


Figure 3: Illustration of how intermediate IGP weights (for link (v_3, d) in this case) enable to achieve a loop-free reconfiguration for the update example in Fig. 1.

extends the reconfiguration mechanism to consider traffic congestion in addition to forwarding consistency.

Modifying protocol specifications may seem the most straightforward solution to deal with reconfigurations in traditional networks, but it actually has practical limitations. First, this approach cannot accommodate custom reconfiguration objectives. For instance, ordered forwarding changes generally work only on a per-destination basis [21], which can make the reconfiguration process slow if many destinations are involved – while one operational objective could be to exit transient states as quickly as possible. Second, protocol modifications are typically targeted to specific reconfiguration cases (e.g., single-link failures), since it is technically hard to predict the impact of any configuration change on forwarding paths. Finally, protocol extensions are not easy to implement in current IGPs, because of the need for passing through vendors (to change proprietary router software), the added complexity and the potential overhead (e.g., load) induced on routers.

Limited practicality of protocol modifications quickly motivated new approaches, based on coordinating operations in order to eventually replace the initial configuration with the final one on all network nodes, while guaranteeing absence of disruptions throughout the process. Those approaches, summarized in the following, mainly focused on support for planned operations.

Optimization algorithms can minimize disruptions. As a first attempt, Keralapura *et al.* [45] studied the problem of finding the optimal way in which devices and links can be added to a network to minimize disruptions. Many following contributions focused on finer-grained operations to gain additional degrees of freedom in IGP reconfiguration problems.

A natural choice among finer-grained operations readily supported by traditional routers is tweaking IGP link weights. For example, in [80] and [81], Raza *et al.* propose a theoretical framework to formalize the problem of minimizing a certain disruption function (e.g., link congestion) when the link weights have to be changed. The authors also propose a heuristic to find an ordering in which to modify several IGP weights within a network, so that the number of disruptions is minimal.

While easily applicable to real reconfiguration cases, those approaches assume primitives which are quite coarse grained (e.g., addition of a link, or weight changes), and cannot guarantee the absence of disruptions in several cases: The scenario in Fig. 2 is an example where coarse-grained operations (link removal) cannot prevent forwarding loops.

Progressively changing link weights can avoid loops. Intermediate IGP link weights can be used during a reconfiguration to avoid disruptions – at the cost of increasing the size of the update sequence and slowing down the update. Consider again the example in Fig. 2, and let the final weight for link (v_3, d) conventionally be ∞. In this case, the forwarding loops potentially triggered by the IGP reconfiguration can be provably prevented by using two intermediate weights for link (v_3, d), as illustrated in Fig. 3. The first of those intermediate weights (see Fig. 3b) is used to force v_1 and only v_1 to change its next-hop, from v_2 to d: Intuitively, this prevents the loop between v_2 and v_1. The second intermediate weight (see Fig. 3c) similarly guarantees that the loop between v_3 and v_2 is avoided, i.e., by forcing v_2 to use its final next-hop before v_3. Of course, finding intermediate weights that guarantee the absence of disruptions becomes much trickier when multiple destinations are involved.

Such a technique can be straightforwardly applied to real routers. For example, an operator can progressively change the weight of (v_3, d) to 31 by editing the configuration of v_3 and d, then check that the all IGP routers have converged on the paths in Fig. 3b, repeat similar operations to reach the state in Fig. 3c, and safely remove the link. Even better, it has been shown [23] that a proper sequence of intermediate link weights can always avoid all possible transient loops for any single-link reweighting. Obviously, the weight can be changed on multiple links in a loop-free way, by progressively reweighting links one by one.

Additional research contributions then focused on minimizing the number of intermediate weights that ensure loop-free reconfigurations. Surprisingly, the problem is *not* computationally hard, despite the fact that all destinations have potentially to be taken into account when changing link weights. Polynomial-time algorithms have been proposed to support planned operations at the per-link [12, 23] (e.g., single-link reweighting) and at a per-router [13, 14] (e.g., router shutdown/addition) granularity.

Ships-in-the-Night (SITN) techniques generalize the idea of incremental changes to avoid loops. To improve the update speed in the case of many link changes and deal with generalized reconfigurations (from changing routing parameters to replacing an IGP with another), both industrial best practices and research works often rely on a technique commonly called Ships-in-the-Night [29]. This technique builds upon the capability of traditional routers to run multiple routing processes at the same time. Thanks to this capability, both the initial and final configurations can be installed (as different routing processes) on all nodes at the same time. Fig. 4 shows the setup for a Ships-in-the-Night reconfiguration for the reconfiguration case in Fig. 2.

In SITN, the reconfiguration process then consists in swapping the preference of the initial configuration with the final one on every node, potentially for a single destination. Hence, at any moment in time, every node uses only one of the two configurations, but different nodes can use different configurations. This implies that (1) for each destination, every switch either uses its initial next-hops or its final ones, meaning that the update does not add overhead to the hardware memory of any node; but (2) inconsistencies may

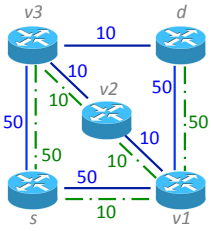


Figure 4: Ships-in-the-Night setup: All routers run two routing processes, one with the initial configuration (blue, solid lines) and the other with the final configuration (green, dashed and dotted segments).

arise from the mismatch between routing processes used by distinct nodes. For example, Fig. 5 shows a SITN-based reconfiguration that mimicks the progressive link weight increment depicted in Fig. 3.

SITN reconfiguration techniques are more powerful than IGP link reweighting. The Ships-in-the-Night framework enables to change the forwarding next-hop of each router independently from the others, hence providing a finer-grained reconfiguration primitive with respect to IGP weight modifications (which influence all routers for all destinations). Moreover, SITN techniques can be applied to arbitrary changes of the IGP configuration, rather than just link reweighting.

On the flip side, the Ships-in-the-Night approach also opens a new algorithmic problem, that is, to decide a safe order in which to swap preferences on a per-router basis. Indeed, naive approaches in swapping configuration preferences cannot guarantee disruption-free reconfigurations. For example, replacing the initial configuration with the final one on all nodes at once provides no guarantee on the order in which new preferences are applied by nodes, hence potentially triggering packet losses and service disruptions (in addition to massive control-plane message storms). Even worse, such an approach could leave the network in an inconsistent, disrupted and hard-to-troubleshoot state if any reconfiguration command is lost or significantly delayed. Similarly, industrial best practices (e.g., [29, 78]) only provide rules of thumb which do not apply in the general case, and do not guarantee lossless reconfiguration processes.

To guarantee the absence of disruptions, configuration preference must then be swapped incrementally, in a carefully-computed order [93]. This called for research contributions. Prominently, Vanbever *et al.* [93, 94] proposed various algorithms (based on Linear Programming, and heuristic ones) to deal with many more IGP reconfiguration scenarios, including the simultaneous change of multiple link weights, the modification of other parameters (e.g., OSPF areas) influencing IGP decisions, and the replacement of one IGP protocol with another (e.g., OSPF with IS-IS). To minimize the update time, the proposed algorithms also try to touch each router only once, i.e., modifying its forwarding entries to all possible destinations altogether. As such, they also generalize the algorithms behind protocol-modification techniques, especially oFIB [21], that restrict to per-destination operational orderings. Beyond providing ordering algorithms, [93, 94] also describe comprehensive system to carry out loop-free IGP reconfigurations in automatically or semi-automatically, i.e., possibly waiting the input from the operator to perform the next set of operations in the computed operational sequence.

2.2 Generalized Routing Reconfigurations in Traditional Networks

Research contributions have been devoted to reconfigurations in more realistic settings, including other protocols than just an IGP.

Enterprise networks, with several routing domains. As a first example, the Ships-in-the-Night framework has been used to carry out IGP reconfigurations in enterprise networks. Those networks typically use *route redistribution* [50], a mechanism enabling the propagation of information from one routing domain (e.g., running a given IGP) to another (e.g., running a different IGP). Unfortunately, route redistribution may be responsible for both routing (inability to converge to a stable state) and forwarding (e.g., loop) anomalies [50]. Generalized network update procedures have been proposed in [98] to avoid transient anomalies while (i) reconfiguring a specific routing domain without impacting another, and/or (ii) arbitrarily changing the size and shape of routing domains.

Internet Service Providers (ISPs), with BGP and MPLS. In ISP networks, the BGP and often MPLS protocols are pervasively used to manage transit Internet traffic, for which both the source and the destination is external to the network. Vanbever *et al.* [92] showed that even techniques guaranteeing safe IGP reconfigurations can cause transient forwarding loops in those settings, because of the interaction between IGP and BGP. They also proved conditions to avoid those BGP-induced loops during IGP reconfigurations, by leveraging MPLS or BGP configuration guidelines.

In parallel, a distinct set of techniques aimed at supporting BGP reconfigurations. Those contributions range from mechanisms to avoid churn of BGP messages during programmed operations (e.g., router reboots or BGP session maintenance [22]) to techniques for safely moving virtual routers [101] or part of physical-router configuration (e.g., BGP sessions) [44]. A framework that guarantees strong consistency for arbitrary changes of the BGP configuration is presented in [99]: It is based on implemeting Ships-in-the-Night in BGP and using packet tags to uniformly apply either the initial or the final forwarding at all routers.

Internet-level problems, like maintaining global connectivity upon failures, have also been explored (see, e.g., [48]).

Protocol-independent reconfiguration frameworks. By design, all the above approaches are dependent on the considered (set of) protocols and even on their implementation.

Protocol-independent reconfiguration techniques have also been proposed. Prominently, in [1], Alimi *et al.* generalize the ship-in-the-night technique, by re-designing the router architecture. This re-design would allow routers not only to run multiple configurations simultaneously but also to select the configuration to be applied on every packet based on the value of a specific bit in the packet header. The authors also describe a commitment protocol to support the switch between configurations without creating forwarding loops.

Mechanisms for consensus routing have been explored in [38].

2.3 Software-Defined Networks

Recently, Software Defined Networking (SDN) has grown in popularity, thanks to its promises to spur abstractions, mitigate compelling management problems and avoid network ossification [62].

Software-defined networks differ from traditional ones from an architectural viewpoint: In SDN, the control is outsourced and consolidated to a logically-centralized element, the network controller, rather than having devices (switches and routers) run their own distributed control logic. In pure SDN networks, the controller computes (according to operators' input) and installs (on the controlled devices) the rules to be applied to packets traversing the network: No message exchange or distributed computation are needed anymore on network devices.

Fig. 6 depicts an example of an SDN network, configured to implement the initial state of our update example (see Fig. 1). Beyond

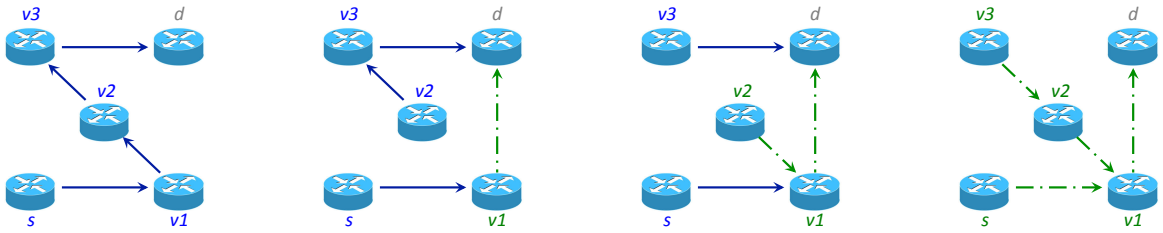


Figure 5: Illustration of a Ships-in-the-Night reconfiguration that mimicks the progressive link reweighting shown in Fig. 3. In each figure, the colors of router names indicate their respective control-plane preferences at the represented reconfiguration step.

the main architectural components, the figure also illustrate a classic interaction between them. Indeed, the dashed lines indicate that the SDN controller instructs the reprogrammable network devices, typically switches [62]), on how to process (e.g., forward) the traversing packets. An example command sent by the controller to switch s is also reported next to the dashed line connecting the two.

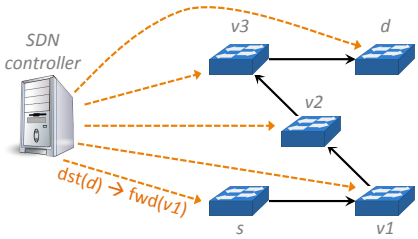


Figure 6: Implementation of the surpassed state in Fig. 1 in an SDN network.

The SDN architecture makes the role of network updates even more frequent and critical than in traditional networks. On the one hand, controllers are often intended to support several different requirements, including performance (like optimal choice of per-flow paths), security (like firewall and proxy traversal) and packet-processing (e.g., through the optimized deployment of virtualized network functions) ones. On the other hand, devices cannot provide any reaction (e.g., to topological changes) like in traditional networks. In turn, this comes at the risk of triggering inconsistencies, e.g., creating traffic blackholes during an update, that are provably impossible to trigger by reconfiguring current routing protocols [96]. As a consequence, the controller has to carry out a network update for every event (from failures to traffic surges and requirement modification) that can impact the computed forwarding entries, while typically supporting more critical consistency guarantees and performance objectives than in traditional networks.

An extended corpus of SDN update techniques have already been proposed in the literature, following up on the large interest raised by SDN in the last decade. This research effort nicely complements approaches to specify [20], compile [9, 75], and check the implementation of [42, 43] network requirements specified by operators in SDN networks.

Historically speaking, the first cornerstone of SDN updates is represented by the work by Reitblatt *et al.* in [82, 83]. This work provides a first analysis of the additional (e.g., security) requirements to be considered for SDN updates, extending the scope of consistency properties from forwarding to policy ones. In particular, it focuses on per-packet consistency property, imposing that packets have to be forwarded either on their initial or on their final paths (never a combination of the two), throughout the update.

The technical proposal is centered around the 2-phase commit technique, which relies on tagging packets at the ingress so that either all initial rules or all final ones can be consistently applied network-wide. Initially, all packets are tagged with the "old label" (e.g., no tag) and rules matching the old label are pre-installed on all the switches. In a first step, the controller instructs the internal switches to apply the final rule to packets carrying the "new label" (i.e., no packet at this step). After the internal switches have confirmed the successful installation of these new rules, the controller then changes the tagging policy at the ingress switches, requiring them to tag packets with the "new label". As a result, packets are immediately forwarded along the new paths. Finally, the internal switches are updated (to remove the old rules), and an optional cleaning step can be applied to remove all tags from packets. Fig. 7 shows the operational sequence produced by the 2-phase commit technique for the update case in Fig. 3.

Several works have been inspired by the 2-Phase technique presented in [82]. On the one hand, a large set of contributions focused on additional guarantees that can be provided by building upon that technique, e.g., to avoid congestion during SDN updates (from [31] to [6, 7, 19, 37, 52, 56, 102]). On the other hand, several algorithms [16, 18, 53, 54, 94] to compute a set of ordered rule replacements have been proposed to deal with specific SDN update cases (e.g., where only forwarding consistency is needed) avoid adding rules and wasting critical network resources (i.e., expensive and rare switch TCAM memory slots).

In the following sections, we detail most of those contributions and the insights on different update problems that globally emerge from them.

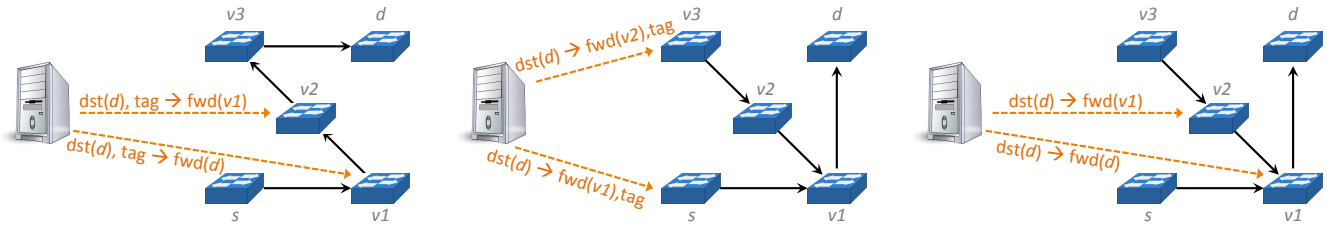
3. TAXONOMY

With this historic perspective and traditional network update problems and techniques in mind, we now present a general formulation of network update problem (§3.1), which abstracts from assumptions and settings considered in different works. This formulation enables us to classify research contributions on the basis of the proposed techniques (e.g., simultaneous usage of multiple configurations on nodes or not) and algorithms, independently of their application to traditional and SDN networks (§3.2).

3.1 Generalized Network Update Problems

In order to compare and contrast research contributions, we first provide a generalized statement for network update problems. We use again Fig. 1 for illustration.

Basic Problem. Generally speaking, a network update problem consists in computing a sequence of operations that changes the



(a) Step 1 (b) Step 2 (c) Step 3

Figure 7: Application of the 2-phase commit technique for carrying out our update example (see Fig. 3). A final (optional) step consists in cleaning the configuration by removing packet tags, i.e., reverting tagging at $v3$ and s as enforced by Step 2.

packet-processing rules installed on network devices. Consider any communication network: It is composed by a given set of interconnected devices, that are able to process (e.g., forwarding to the next hop) data packets according to rules installed on them. We refer to the set of rules installed on all devices at a given time as network state at that time. Given an initial and final state, a network update consists in passing from the initial state to the final one by applying operations (i.e., adding, removing or changing rules) on different devices. In Fig. 1, the initial state forces packets from source s to destination d along the path (s, v_1, v_2, v_3, d). In contrast, the final state forwards the same packets over (s, v_1, d), as well as packets from v_3 to d on (v_3, v_2, v_1, d). The network update problem consists in replacing the initial rules with the final ones, so that the paths for d are updated from (s, v_1, v_2, v_3, d) to (s, v_1, d) and (v_3, v_2, v_1, d).

Operations. To perform a network update, a sequence of operations has to be computed. By operation, we mean any modification of a device behavior in processing packets. As an example, an intuitive and largely-supported operation on all network devices is rule replacement: instructing any device (e.g., v_3) to replace an initial rule (e.g., forward the $s-d$ packet flow to v_2) with the corresponding final one (e.g., forward the $s-d$ flow to d).

Consistency. The difficulty in solving network update problems is that some form of consistency must be guaranteedly preserved throughout the update, for practical purposes like avoiding service disruptions and packet losses. Preserving consistency properties, in turn, depends on the order in which operations are executed by devices – even if both the initial and the final states comply with those properties. For example, if v_3 replaces its initial rule with its final one before v_2 in Fig. 1, then the operational sequence triggers a forwarding loop between v_2 and v_3 that interrupts the connectivity from s to d. In §3.2, we provide an overview of consistency properties considered in prior work.

The practical need for guaranteeing consistency has two main consequences. First, it forces network updates to be carried out incrementally, i.e., conveniently scheduling operations over time so that the installed sequence of intermediate states is provably disruption-free. Second, it requires a careful computation of operational sequences, implementing specific reasoning in the problem-solving algorithms (e.g., to avoid replacing v_3's rule before v_2's one in the previous example).

Performance. Another algorithmic challenge consists in optimizing network-update performance. As an example, minimizing the time to complete an update is commonly considered among those optimization objectives. Indeed, carrying out an update generally requires to install intermediate configurations, and in many cases it is practically desirable to minimize the time spent in such intermediate states. We provide a broader overview of performance goals considered by previous works in §3.2.

Final Operational Sequences. Generally, the solution for an update problem can be represented as a sequence of *steps* or *rounds*, that both guarantees consistency properties and optimizes update performance. Each step is a set of operations that can be started at the same time. Note that this does not mean that operations in the same step are assumed to be executed simultaneously on the respective devices. Rather, all operations in the same step can be started in parallel because target consistency properties are guaranteed irrespectively of the respective order in which those operations are executed. Examples of operational sequences, computed by different techniques, are reported in §2 (see Figs. 3 and 5).

3.2 Update Techniques

In this section, we provide an overview of the problem space and classify existing models and techniques. Previous contributions have indeed considered several variants of the generalized network update problem as we formulated in §3.1. Those variants differ in terms of both consistency constraints, performance goals and operations that can be used to solve an update problem.

Routing Model. We can distinguish between two alternative routing models: *destination-based* and *per-flow* routing.

1. **Destination-based Routing:** In destination-based routing, routers forward packets based on the destination only. An example for destination-based routing is IP routing, where routers forward packets based on the longest common IP destination prefix. In particular, destination-based routing describes confluent paths: once two flows from different sources destined toward the same destination intersect at a certain node, the remainder (suffix) of their paths will be the same. In destination-based routing, routers store at most one forwarding rule per specific destination.

2. **Per-flow Routing:** In contrast, according to *per-flow* routing, routes are not necessarily confluent: the forwarding rules at the routers are defined per-flow, i.e., they may depend not only on the destination but for example also on the source. In traditional networks, flows and per-flow routing could for example be implemented using MPLS: packets belonging to the same equivalence class resp. packets with the same MPLS tag are forwarded along the same path.

Operations. Techniques to carry out network updates can be classified in broad categories, depending on the operations that they consider.

1. **Rule replacements:** A first class of network update algorithms is based on partitioning the total set of updates S

to be made at the different switches into different rounds: $S = (S_1, S_2, \ldots, S_k)$, where $S_i \cap S_j = \emptyset$ for all $i, j \in [1, k]$ and where S_t denotes the set of switches which is updated in round t. Consistent node ordering update schedules have the property that the updates in each round S_t may occur asynchronously, i.e., in an arbitrary order, without violating the desired consistency properties (e.g., loop-freedom). The next batch of updates S_{t+1} is only issued to the switches after the successful implementation of the S_t updates has been confirmed (i.e., ACKed) by the switches.

2. **Rule additions:** A second class of network update algorithms is based on adding rules to guarantee consistency during the update. The following two main variants of this approach have been explored so far.

   (a) **2-Phase commit:** In this case, both the initial and the final rules are installed on all devices in the central steps of the updates. Packet are tagged at the border of the network to enforce that the internal devices either (i) all use the initial rules, or (ii) all use the final rules. See Fig. 7 for an example.

   (b) **Additional helper rules:** For the purpose of the update, additional rules may be introduced temporarily, which do not belong neither to the old path nor to the new path. These rules allow to divert the traffic temporarily to other parts of the network, and are called *helper rules*.

Consistency properties. Another canonical classification can be defined along the fundamental types of consistency properties:

1. **Connectivity consistency:** The most basic form of consistency regards the capability of the network to continuously deliver packets to their respective destinations, throughout the update process. This boils down to guaranteeing two correctness properties: absence of blackholes (i.e., paths including routers that cannot forward the packets further) and absence of forwarding loops (i.e., packets bouncing back and forth over a limited set of routers, without reaching their destinations).

2. **Policy consistency:** Paths used to forward packets may be selected according to specific forwarding policies, for example, security ones imposing that given traffic flows must traverse specific waypoints (firewalls, proxies, etc.). In many cases, those policies have to be preserved during the update. Generally speaking, policy consistency properties impose constraints on which paths can be installed during the update (as a consequence of the partial application of an operational sequence). For example, a well-studied policy consistency property, often referred to as *strong consistency*, requires that packets are always forwarded along either the pre-update or the post-update paths, but never a combination of the two.

3. **Performance consistency:** A third class of consistency properties takes into account actual availability and limits of network resources. For instance, many techniques account for traffic volumes and the corresponding constraints raised by the limited capacity of network links: They indeed aim at respecting such constraints in each update step, e.g., to avoid *transient congestion* during updates.

This classification is also reflected in the structure of this survey.

Performance goals. We can distinguish between three broad classes of performance goals.

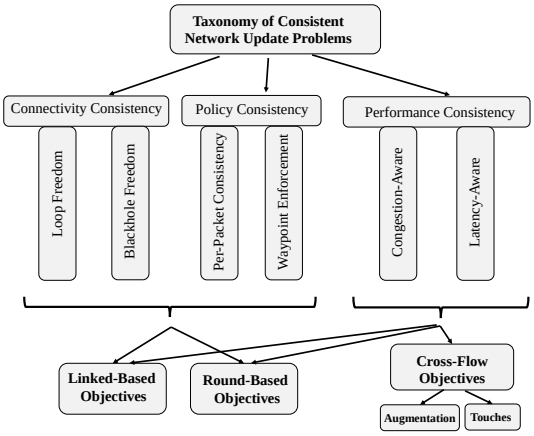


Figure 8: Types of consistent network update problems.

1. **Link-based:** A first class of consistent network update protocols aims to make new links available as soon as possible, i.e., to maximize the number of switch rules which can be updated simultaneously without violating consistency.

2. **Round-based:** A second class of consistent network update protocols aims to minimize the number of inter-actions between the controller and the switches.

3. **Cross-Flow Objectives:** A third class of consistent network update protocols targets objectives arising in the presence of multiple flows.

   (a) **Augmentation:** Minimize the extent to which link capacities are oversubscribed during the update (or make the update entirely congestion-free).

   (b) **Touches:** Minimize the number of interactions with the router.

Link-based and round-based objectives are usually considered for node-ordering algorithms and for weak-consistency models. Congestion-based objectives are naturally considered for capacitated consistency models.

Fig. 8 gives an overview of different types of network update problems.

4. CONNECTIVITY CONSISTENCY

In this section, we focus on update problems where the main consistency property to be guaranteed concerns the delivery of packets to their respective destinations. Packet delivery can be disrupted during an update by forwarding loops or blackholes transiently present in intermediate states. We separately discuss previous results on how to guarantee loop-free and blackhole-free network updates. We start from the problem of avoiding forwarding loops during updates, because they are historically the first update problems considered – by works on traditional networks (see §2). This is also motivated by the fact that blackholes cannot be created by reconfiguring current routing protocols, as proved in [96]. We then shift our focus on avoiding blackholes during arbitrary (e.g., SDN) updates.

4.1 Loop-Freedom

Loop-freedom is a most basic consistency property and has hence been explored intensively already.

4.1.1 Definitions

We distinguish between flow-based and destination-based routing: in the former, we can focus on a single (and arbitrary) path from s to d: forwarding rules stored in the switches depend on both s and d, can flows can be considered independently. In the latter, switches store a single forwarding rule for a given destination: once the paths of two different sources destined to the same destination intersect, they will be forwarded along the same nodes in the rest of their route: the routes are confluent.

Moreover, once can distinguish between two different definitions for loop-free network updates: *Strong Loop-Freedom (SLF)* and *Relaxed Loop-Freedom (RLF)* [54]. SLF requires that at any point in time, the forwarding rules stored at the switches should be loop-free. RLF only requires that forwarding rules stored by switches *along the path from a source s to a destination d* are loop-free: only a small number of "old packets" may temporarily be forwarded along loops.

4.1.2 Algorithms and Complexity

Node-based objective ("greedy approach"). Mahajan and Wattenhofer [58] initiated the study of destination-based (strong) loop-free network updates. In particular, the authors show that by scheduling updates across multiple rounds, consistent update schedules can be derived which do not require any packet tagging, and which allow some updated links to become available earlier. The authors also present a first algorithm that quickly updates routes in a transiently loop-free manner. The study of this model has been refined in [18, 19], where the authors also establish hardness results. In particular, the authors prove that for two destinations and for sublinear x, it is NP-hard to decide if x rounds (cf. round-based objectives) of updates suffice. Furthermore, maximizing the number of rules updated for a single destination is NP-hard as well, but can be approximated well.

Ludwig *et al.* [54, 55] initiated the study of arbitrary route updates: routes which are not necessarily destination-based. The authors show that the update problem in this case boils down to an optimization problem on a very simple directed graph: initially, before the first update round, the graph simply consists of two connected paths, the old and the new route. In particular, every network node which is not part of both routes can be updated trivially, and hence, there are only three types of nodes in this graph: the source s has out-degree 2 (and in-degree 0), the destination d has in-degree 2 (and out-degree 0), and every other node has in-degree and out-degree 2. The authors also observe that loop-freedom can come in two flavors, strong and relaxed loop-freedom [54].

Despite the simple underlying graph, however, Amiri *et al.* [2] show that the node-based optimization problem is NP-hard, both in the strong and the relaxed loop-free model (SLF and RLF). As selecting a maximum number of nodes to be updated in a given round (i.e., the node-based optimization objective) may also be seen as a heuristic for optimizing the number of update rounds (i.e., the round-based optimization objective), the authors refer to the node-based approach as the "greedy approach".

Amiri *et al.* [2] also present polynomial-time optimal algorithms for the following scenarios: Both a maximum SLF update set as well as a maximum RLF update set can be computed in polynomial-time in trees with two leaves. Regarding polynomial-time approximation results, the problem is 1/2-approximable in general, both for strong and relaxed loop-freedom. For additional approximation results for specific problem instances, we refer the reader to Amiri *et al.* [2].

Round-based objective ("greedy approach"). Ludwig *et al.* [54] initiate the study of consistent network update schedules which minimize the number of interaction rounds with the controller: *How many communication rounds k are needed to update a network in a (transiently) loop-free manner?*

The authors show that answering this question is difficult in the strong loop-free case. In particular, they show that while deciding whether a k-round schedule exists is trivial for $k = 2$, it is already NP-complete for $k = 3$. Moreover, the authors show that there exist problem instances which require $\Omega(n)$ rounds, where n is the network size. Moreover, the authors show that the greedy approach, aiming to "greedily" update a *maximum* number of nodes in each round, may result in $\Omega(n)$-round schedules in instances which actually can be solved in $O(1)$ rounds; even worse, a *single* greedy round may inherently delay the schedule by a factor of $\Omega(n)$ more rounds.

However, fast schedules exist for *relaxed loop-freedom*: the authors present a deterministic update scheduling algorithm which completes in $O(\log n)$-round in the worst case.

Hybrid Approaches. Vissicchio *et al.* presented FLIP [95], which combines per-packet consistent updates with order-based rule replacements, in order to reduce memory overhead: additional rules are used only when necessary. Moreover, Hua *et al.* [32] initiated the study of adversarial settings, and presented FOUM, a flow-ordered update mechanism that is robust to packet-tampering and packet dropping attacks.

Other Objectives. Dudycz *et al.* [16] initiated the study of how to update multiple policies simultaneously, in a loop-free manner. In their approach, the authors aim to minimize the number of so-called *touches*, the number of updates sent from the controller to the switches: ideally, all the updates to be performed due the different policies can be sent to the switch in one message. The authors establish connections to the *Shortest Common Supersequence (SCS)* and *Supersequence Run* problems [63], and show NP-hardness already for two policies, each of which can be updated in two rounds, by a reduction from *Max-2SAT* [51].

However, the authors also present optimal polynomial-time algorithms to combine consistent update schedules computed for individual policies (e.g., using any existing algorithm, e.g., [54, 58]), into a global schedule guaranteeing a minimal number of touches. This optimal merging algorithm is not limited to loop-free updates, but applies to any consistency property: if the consistency property holds for individual policies, then it also holds in the joint schedule minimizing the number of touches. the *Shortest Common Supersequence (SCS)* and *Supersequence Run* [63].

4.1.3 Related Optimization Problems

The link-based optimization problem, the problem of maximizing the number of links (or equivalently nodes) which can be updated simultaneously, is an instance of the maximum acyclic subgraph problem (or equivalently: dual minimum feedback arc set problem). For the NP-hardness, reductions from *SAT* and *Max-2SAT* are presented.

4.1.4 Concluding Remarks and Open Problems

Loop-free network updates still pose several open problems. Regarding the node-based objective, Amiri *et al.* [2] conjecture that update problems on bounded directed path-width graphs may still be solvable efficiently: none of the negative results for bounded degree graphs on graphs of bounded directed treewidth seem to be extendable to digraphs of bounded directed pathwidth with bounded degree. More generally, the question of on which graph families network update problems can be solved optimally in polynomial time in the node-based objective remains open. Regarding the round-based objective, it remains an open question whether strong loop-free updates are NP-hard for any $k \geq 3$ (but smaller than n): so far only

$k = 3$ has been proved to be NP-hard. More interestingly, it remains an open question whether the relaxed loop-free update problem is NP-hard, e.g., are 3-round update schedules NP-hard to compute also in the relaxed loop-free scenario? Moreover, it is not known whether $\Omega(\log n)$ update rounds are really needed in the worst-case in the relaxed model, or whether the problem can always be solved in $O(1)$ rounds. Some brute-force computational results presented in [54, 58] indicate that if it is constant, the constant must be large.

4.2 Blackhole-Freedom

Another consistency property is blackhole freedom, i.e., a switch should always have a matching rule for any incoming packet, even when rules are updated (e.g., removed and replaced). This property is easy to guarantee by implementing some default matching rule which is never updated, which however could in turn induce forwarding loops. A straightforward mechanism, if there is currently no blackhole for any destination, is to install new rules with a higher priority, and then delete the old rules [18, 58]. Nonetheless, in the presence of memory limits and guaranteeing loop-freedom, finding the fastest blackhole-free update schedule is NP-hard [18].

5. POLICY CONSISTENCY

While connectivity invariants are arguably the most intensively studied consistency properties in the literature, especially in traditional networks, operators often have additional requirements to be preserved. For example, operators want to ensure that packets traverse a given middlebox (e.g., a firewall) for security reasons or a chain of middleboxes (e.g., encoder and decoder) for performance reasons, or that paths comply with Service Level Agreements (e.g., in terms of guaranteed delay). In this section, we discuss studied problems and proposed techniques aiming at preserving such additional requirements.

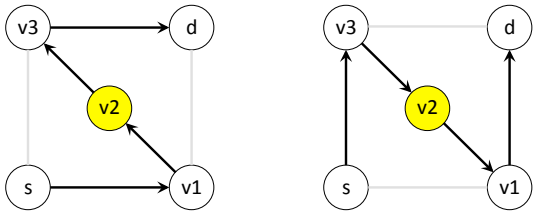
5.1 Definitions

Additional requirements on forwarding paths that may have to be respected during a network update can be modeled by *routing policies*, that is, sub-paths that have to be traversed by transient paths installed during network updates.

Over the years, several contributions have targeted policy-preserving updates, typically focusing on specific policies. Historically, the first policy considered during network updates is *per-packet consistency (PPC)*, which ensures that every packet travels either on its initial or on its final paths, never on intermediate ones. This property is the most natural to (try to) preserve. Assume indeed that both the initial and the final paths comply with high-level network requirements, e.g., security, performance, SLA policies. The most straightforward way to guarantee that those requirements are not violated is to constrain all paths installed during the update to always be either initial paths or final ones.

Nonetheless, guaranteeing per-packet consistency may be an unnecessarily strong requirement in practice. Not always it is strictly needed that transient paths must coincide with either the initial or the final ones. For example, in some cases (e.g., for enterprise networks), security may be a major concern, and many security requirements may be enforced by guaranteeing that packets traverse a firewall. We refer to this specific case where single nodes (waypoints) have to be traversed by given traffic flows as *waypoint enforcement (WPE)*. An example WPE-consistent update is displayed in Fig. 9

More complex policies (i.e., beyond WPE) may also be needed in general. Indeed, policies to be satisfied in SDN networks tend to grow in number and complexity over time, because of both new requirements (e.g., as dictated by use cases like virtualized infrastructure and network functions) and novel opportunities (e.g., programmability and flexibility) opened by SDN. For example, it may desirable that specific traffic flows follow certain sub-paths (e.g., with low delay for video streaming and online gaming applications) or are explitictly denied to pass through other sub-paths (e.g., because of political or economical constraints). Such *arbitrary policies* are also considered in recent SDN update works.



(a) Surpassed state (b) New state

Figure 9: A WPE-consistent update example, taken from [55], where forwarding paths have to be changed from the Surpassed (Fig. 9a) to the New (Fig. 9b) state, while preserving traversal of the waypoint $v2$ (highlighted in the figure) at any time during the update.

5.2 Algorithms and Complexity

2-Phase commit techniques. As described in §3.2, 2-Phase commit techniques deploy carry out updates by (1) tagging packets at their ingress in the network, and (2) using packet tags to use initial or final paths consistently network-wide. Unsurprisingly, this approach guarantees per-packet consistency (hence, potentially any policy satisfied by both pre- and post-update paths).

While the idea is quite intuitive, some support is needed on the devices, e.g., to tag packets and match packet tags. A framework to implement this update approach in traditional networks has been proposed by Alimi *et al.* in [1]. It requires invasive modification of router internals, to manage tags and run arbitrary routing processes in separate process spaces. The counterpart of such a framework for SDN networks is presented in [82, 83]. Those works avoid the need for changing device internals since it relies on OpenFlow, the protocol classically used in SDN networks. They also argue on the criticality of supporting PPC in the SDN case and the advantages of integrating 2-phase commit techniques within an SDN controller.

A major downside of 2-phase commit is that it doubles the consumed memory on switches, along with requiring header space, tagging overhead, and complications with middleboxes changing tags. It indeed requires devices to maintain both the initial and final sets of forwarding rules throughout the update, in order to possibly apply any of the two sets according to packet tags. To mitigate this problem, a variant of the basic approach has been studied in [41]. The authors of the latter work proposed to break a given update into several sub-updates, such that each sub-update changes the paths for a different set of flows. Of course, this approach would make it longer for the full update to be completed. In other words, it can limit the memory overhead on each switch at any moment in time but at the price of slowing down the update.

Actually, the switch-memory consumption of 2-phase commit techniques remains a fundamental limitation of the approach, which also motivated the exploration of alternatives.

SDN-based update protocols. McGeer [60, 61] presented two protocols to carry out network updates and defined on top of OpenFlow. The first update protocol [60] saves switch resources by sending packets to the controller during updates. As a result, switch resources (like precious TCAM entries) are saved, at the cost of

Model	NP-hard	Polynomial time	Remarks
# Rounds, strong LF	Is there a 3-round loop-free update schedule? [54] *For 2-destination rules and sublinear x: Is there a x-round loop-free update schedule? [19]*	Is there a 2-round loop-free update schedule? [54]	In the worst case, $\Omega(n)$ rounds may be required. [54], *[18]*. $O(n)$-round schedules always exist *[58]*. Both applies to flow-based & *destination-based* rules.
# Rounds, relaxed LF	No results known.	$O(\log n)$-round update schedules always exist. [54]	It is not known whether $o(\log n)$-round schedules exist (in the worst case). No approximation algorithms are known.
# Links, strong LF	Is it possible to update x nodes in a loop-free manner? [2], *[19]*	Polynomial-time optimal algorithms are known to exist in the following cases: A maximum SLF update set can be computed in polynomial-time in trees with two leaves. [2]	The optimal SLF schedule is 2/3-approximable in polynomial time in scenarios with exactly three leaves. For scenarios with four leaves, there exists a polynomial-time 7/12-approximation algorithm. [2] Approximation algorithms from maximum acyclic subgraph [2] and minimum feedback arc set *[18]* apply.
# Links, relaxed LF	Is it possible to update x nodes in a loop-free manner? [2]	Polynomial-time optimal algorithms are known to exist in the following cases: A maximum RLF update set can be computed in polynomial-time in trees with two leaves. [2]	No approximation results known. [2]

Table 1: Overview of results for loop-freedom. Results/references in *italics* are in the destination-based model.

adding delay on packet delivery, and consuming network bandwidth and controller memory. The second update protocol [61] is based on a logic circuit for the update sequence which requires neither rule-space overhead nor transferring the packets to the shelter during the update.

Both proposals need a dedicated protocol which is not currently supported by devices out of the box.

Rule replacement ordering. Some works explored which policies can be supported, and how, by only relying on (ordered) rule replacements, given that this both (i) comes with no memory overhead and (ii) is supported by both traditional and SDN devices.

Some works noticed that PPC can be an unnecessarily strong requirements in several practical cases. Initial contributions mainly focused on WPE consistency, e.g., to preserve security policies. Prominently, [55] studies how to compute quick updates that preserve WPE by only replacing initial with final rules, when any given flow has to traverse a single waypoint. The authors propose WayUp, an algorithm that guarantees WPE during the update and terminates in 4 rounds. However, they also show that it may not be possible to ensure waypointing through a single node and loop-freedom at the same time. Fig. 9 actually shows one case in which any rule replacement ordering either causes a loop or a WPE consistency violation. Those infeasibility results are extended to waypoint chains in [53]. In that work, in particular, the authors show that flexibility in ordering and placing virtualized functions specified by a chain do not make the update problem always solvable. The two works also show that it is computationally hard (NP-hard) to even decide if an ordering preserving both WPE and loop-freedom exists. Mixed integer program formulations to find an operational are proposed and evaluated in both cases.

The more general problem of preserving policies defined by operators is tackled in [59]. That paper describes an approach to (i) model update-consistency properties as Linear Temporal Logical formulas, and (ii) automatically synthesize SDN updates that preserve input properties. Such a synthesis is performed by an efficient algorithm based on counterexample-guided search and incremental model checking. Experimental evidence is provided about the scalability of the algorithm (up to one-thousand node networks).

Finally, [97] explores algorithmic limitations of guaranteeing per-packet consistency without relying on state duplication. The work shows that a greedy strategy implements a correct and complete approach in this case, meaning that it finds the maximal sequence of rule replacements that do not violate PPC. Cerny *et al.* [10] complement those findings, by presenting a polynomial-time synthesis algorithm that preserves PPC while allowing the maximal parallelism between per-switch updates. Also, an evaluation on realistic update cases is presented in [97]. It shows that PPC can be preserved while replacing many forwarding entries on the majority of the switches, despite updates can rarely be completed this way. However, this observation motivates both approaches tailored to a more restricted family of policies (like WPE-preserving ones, described above), and efforts for mixed approaches (mixing rule replacements and duplication, see below).

Mixed approaches. In [97], a basic mixed approach is considered to ensure PPC in generalized networks running both traditional and SDN control-planes (or any of the two). This approach consists in first computing the maximal sequence of rule replacements that preserve PPC, and then applying a restricted 2-phase commit procedure on a subset of (non-ordered) devices and flows.

Vissicchio *et al.* [95] propose an algorithm addressing a larger set of update problems with a more general algorithmic approach, but restricting to SDN networks. This work focuses on the problem of preserving generic policies during SDN updates. For each flow, a policy is indeed defined as a set of paths so that the flow must traverse any of those paths in each intermediate state. The proposed algorithm interleaves rule replacements and additions (i.e., packet tagging and tag matching) in the returned operational sequences and during its computation – rather than considering the two primitives in subsequent steps as in [97].

Both works argue that it is practically profitable to combine rule replacements and additions, as it greatly reduces the amount of memory overhead while keeping the operational sequence always computable.

5.3 Related Optimization Problems

Many policy-preserving algorithms face generalized versions of the optimization problems associated to connectivity-preserving updates (see §4): While the most common objective remains the maximization of parallel operations (to speed-up the update), policy consistency requires that all possible intermediate paths comply with certain regular expressions in addition to being simple (that is, loop-free) paths. Mixed policy-preserving approaches focus on

even more general problems where (i) different operations can be interleaved in the output operational sequence (which provides more degrees of freedom in solving the input problems), and (ii) multiple optimization objectives are considered at the same time (typically, maximizing the update parallelism while also minimizing the consumed switch memory).

5.4 Concluding Remarks and Open Problems

Unsurprisingly, preserving policies requires more sophisticated update techniques, since it is generally harder to extract policy-induced constraints and model the search space. Two major families of solutions have been explored so far. On the one hand, 2-phase commit techniques and update protocols sidestep the algorithmic challenges, at the cost of relying on specific primitives (packet tagging and tag matching) that comes with switch memory consumption. On the other hand, ordering-based techniques directly deal with problem complexities, at the cost of algorithmic simplicity and impossibility to always solve update problems. Finding the best balance between those two extremes is an interesting research direction. Some initial work has started in this direction, with the proposal of algorithms that can interleave different kinds of operations within the computed sequence (see mixed approaches in §5.2). However, many research questions are left open. For example, the computational complexity of solving update problems while mixing rule additions (for packet tagging and matching) with replacements is unknown. Moreover, it is unclear whether the proposed algorithms can be improved exploiting the structure of specific topologies or the flexibility of new devices (e.g., P4-compatible ones [5]), e.g., to achieve better trade-offs between memory consumption and update speed.

6. PERFORMANCE-AWARE CONSISTENCY

Computer networks are inherently capacitated, and respecting resource constraints is hence another important aspect of consistent network updates. Congestion is known to significantly impact throughput and increase latency, therefore negatively impacting user experience and even leading to unpredictable economic loss.

6.1 Definitions

The capacitated update problem is to migrate from a multi-commodity flow $\mathcal{F}_{old}$ to another multi-commodity flow $\mathcal{F}_{new}$, where consistency is defined as not violating any link capacities and not rate-limiting any flow below its demand in $\min(\mathcal{F}_{old}, \mathcal{F}_{new})$. In few works, e.g., [6], $\mathcal{F}_{new}$ is only implicitly specified by its demands, but not by the actual flow paths. Some migration algorithms will violate consistency properties to guarantee completion, as a consistent migration does not have to exist in all cases.

Typically, four different variants are studied in the literature: First, individual flows may either only take one path (unsplittable) or they may follow classical flow-theory, where the incoming flow at a switch must equal its outgoing flow (splittable). Secondly, flows can take any paths via helper rules in the network during the migration (intermediate paths), or may only be routed along the old or the new paths (no intermediate paths).

To exactly pinpoint congestion-freedom, one would need to take many detailed properties into account, e.g., buffer sizes and ASIC computation times. As such, the standard consistency model does not take this fine-grained approach, but rather aims at avoiding ongoing bandwidth violations and takes a mathematical flow-theory point of view. Introduced by [31], consistent flow migration is captured in the following model: No matter if a flow is using the rules before the update or after the update, the sum of all flow sizes must be at most the links capacity.

6.2 Algorithms and Complexity

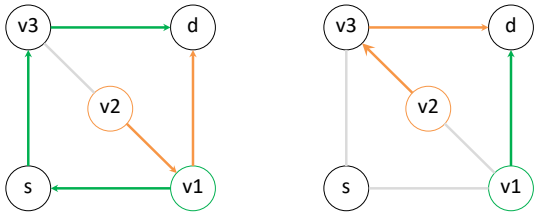
6.2.1 Algorithms

Current algorithms for capacitated updates of network flows use the seminal work by Reitblatt *et al.* [82] as an update mechanism. Analogously to *per-packet consistency* (cf. §5), one can achieve *per-flow consistency* by a 2-phase commit protocol. While this technique avoids many congestion problems, is not sufficient for bandwidth guarantees: When updating the two flows in Fig. 10, the lower green flow could move up before orange flow is on its new path, leading to congestion.

An overview over all algorithmic approaches discussed here can be found in Table 2.

Mizrahi *et al.* [65] prove that flow swapping is necessary for throughput optimization in the general case, as thus algorithms are needed that do not violate any capacity contraints during the network update, beyond simple flow swapping as well.

The seminal work by Hong *et al.* [31] on *SWAN* introduces the current standard model for capacitated updates. Their algorithmic contribution is two-fold, and also forms the basis for *zUpdate* [52]: First, the authors show that if all flow links have free capacity *slack* s, consistent migration is possible using $\lceil 1/s \rceil - 1$ updates: E.g., if the free capacity is 10%, 9 updates are required, always moving 10% of the links' capacity to the new flow paths. If the network contains non-critical background traffic, free capacity can be generated for a migration by rate-limiting this background traffic temporarily, cf. Fig. 11: removing some background traffic allows for consistent migration.

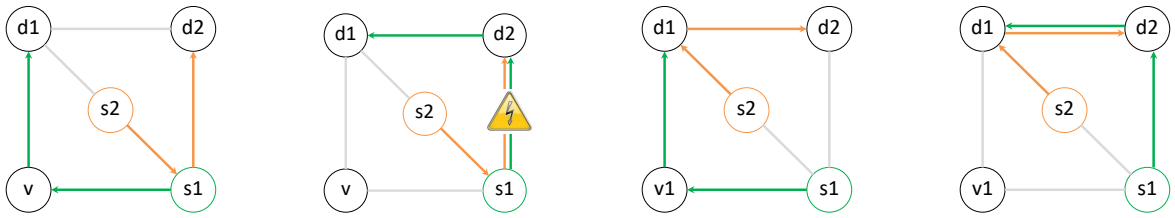


(a) Initial (b) Final

Figure 11: In this network the task is again to migrate consistently from the initial to the final state. If all flows and links have unit size, no consistent migration is possible: The destination has just two incoming links of combined size two. Should the flows just have a size of $2/3$, one can migrate consistently in $\lceil 1/(1/3) \rceil - 1 = 2$ updates by moving half of the flow size of $1/3$ each time in parallel.

Second, the authors provide an LP-formulation for splittable flows which provides a consistent migration schedule with x updates, if one exists. By performing a binary search over the number of updates, the number of necessary updates can be minimized. This approach allows for intermediate paths, where the flows can be re-routed anywhere in the network. E.g., consider the example in Fig. 11 with all flows and links having unit size. If there was an additional third route to d, the orange flow could temporarily use this intermediate path: we can then switch the green flow, and eventually the orange flow could be moved to its desired new path.

This second LP-formulation was extended by Zheng *et al.* [102] to include unsplittable flows as well via a MIP. Furthermore, using randomized rounding with an LP, Zheng *et al.* can approximate



(a) Initial (b) Congestion! (c) Orange first (d) Final

Figure 10: In this introductory network for flow migration, all links have unit bidirectional capacity, and both orange and green flows have unit size as well. The task is to move both the green and orange flows from their initial paths in Fig. 10a to their final ones shown in Fig. 10d. Updating both flows together could lead to the green flow being moved first, inducing congestion, see Fig. 10b. However, this can be avoided by using succinct updates, first moving the orange flow as in Fig. 10c, and then moving the green flow.

the minimum congestion that will occur if the migration has to be performed using x updates. Should intermediate paths be allowed however, then their LP is of exponential size. Paris *et al.* [77] also consider the tradeoff between reconfiguration effects and update speed in the context of dynamic flow arrivals. In terms of tradeoffs, Luo *et al.* [57] allow for user-specified deadlines (e.g., a flow has to be updated until some time t) via an MIP or an LP-based heuristic.

The work by Brandt *et al.* [7] tackles the problem of deciding in polynomial time if consistent migration is possible at all for splittable flows with intermediate paths allowed. By iteratively checking for augmenting flows that create free capacity (slack) on fully-capacitated links, it is possible to decide in polynomial time if slack can be obtained on all flow links. If yes, then the first technique of [31] can be used, else no consistent migration is possible. Should the output be no, they also provide an LP-formulation to check to which demands it is possible to migrate consistently.

Jain *et al.* [37] also consider the variable update times of switches in the network. For both splittable and unsplittable flows without intermediate paths, they build a dependency graph for the update problem. Then, this dependency graph is traversed in a greedy fashion, updating whatever flows are currently possible. E.g., in Fig. 10, the orange flow would be moved first, then the green flow next. Should this traversal result in a deadlock, flows are rate-limited to guarantee progress. Wang *et al.* [100] improve the local dependency resolving to improve the greedy traversal. Luo *et al.* [56] provide a MIP-formulation of the problem, and also provide a heuristic framework using tiny MIPs.

Foerster and Wattenhofer [19] consider an alternative approach to migrating unsplittable flows without intermediate paths: They split each flow along its old and new path, changing the size allocations during the updates, until the migration is complete. Their algorithm has polynomial computation time, but has slightly stronger consistency requirements than the model of [31].

Lastly, Brandt *et al.* [6] consider a modified migration problem by not fixing the new multi-commodity flow, but just its demands. If the final (and every intermediate) configuration has no congestion then the locations of the flows in the network do not matter. In scenarios with a single destination (or a single source), augmenting flows can be used to compute the individual updates: Essentially, the flows are changed along the routes of the augmenting flows, allowing for a linear number of updates for splittable flows with intermediate paths. The augmentation model cannot be extended to the general case of multi-source multi-destination network flows.

6.2.2 Complexity

The complexity of capacitated updates can roughly be summarized as follows: Problems involving splittable flows can be decided in polynomial time, while restrictions such as unsplittable flows or memory limits turn the problem NP-hard, see Table 3. In a way, the capacitated update problems differs from related network update problems in that it is not always solvable in a consistent way. On the other hand, e.g., per-packet/flow consistency can always be maintained by a 2-phase commit, and loop-free updates for a single destination can always be performed in a linear number of updates.

One standard approach in recent work for flow migration is linear (splittable flows) or integer programing (unsplittable flows): With the number of intermediate configurations x as an input, it is checked if a consistent migration with x intermediate states exists. Should the answer be yes, then one can use a binary search over x to find the fastest schedule. This idea originated in *SWAN* [31] for splittable flows, and was later extended to other models, cf. Table 2.

However, the LP-approach via binary search (likewise for the integer one) suffers from the drawback that it is only complete if the model is restricted: If x is unbounded, then one can only decide whether a migration with x updates exists, but not whether there is no migration schedule with y steps, for some $y > x$. Additionally, it is not even clear to what complexity class the general capacitated update problem belongs to, cf. the decision problem hardness column of Table 3.

The only exception arises in case of splittable flows without memory restrictions, where either an (implicit) schedule or a certificate that no consistent migration is possible, is found in polynomial time [7]. The authors use a combinatorial approach not relying on linear programming. Adding memory restrictions turns this problem NP-hard as well [37].

If the model is restricted to allow every flow only to be moved once (from the old path to the new path), then the capacitated update problem becomes NP-complete [19, 37]: Essentially, as the number of updates is limited by the number of flows, the problem is in NP. In this specific case, one can also approximate the minimum congestion for unsplittable flows in polynomial time by randomized rounding [102].

Hardly any (in-)approximability results exist today, and most work relies on reductions from the Partition problem, cf. Table 4. The only result that we are aware of is via a reduction from MAX 3-SAT, which also applies to unit size flows [7].

6.3 Related Optimization Problems

In a practical setting, splitting flows is often realized via deploying multiple unsplittable paths, which is an NP-hard optimization problem as well, both for minimizing the number of paths and for maximizing k-splittable flows, cf. [3, 26]. Another popular option is to split the flows at the routers using hash functions; other major techniques are flow(let) caches and round-robin splitting, cf. [27].

Nonetheless, splitting flows along multiple paths can lead to packet reordering problems, which need to be handled by further techniques, see, e.g., [39].

Many of the discussed flow migration works rely on linear programming formulations: Even though their runtime is polynomial in theory, the timely migration of large networks with many intermediate states is currently problematic in practice [31]. If the solution takes too long to compute, the to-be solved problem might no longer exist, a problem only made worse when when resorting to (NP-hard) integer programming for unsplittable flows. As such, some tradeoff has to be made between finding an optimal solution and one that can actually be deployed.

Orthogonal to the problem of consistent flow migration is the approach of scheduling flows beforehand, not changing their path assignments in the network during the update. We refer to the recent works by Kandula *et al.* [40] and Perry *et al.* [79] for examples. Game-theoretic approaches have also been considered, e.g., [30].

Lastly, the application of model checking to consistent network updates does not cover bandwidth problem restrictions yet [59, 103].

6.4 Concluding Remarks and Open Problems

The classification of the complexity of flow migration still poses many questions, cf. Table 4: If every flow can only be moved once, then the migration (decision) problem is clearly in NP. However, what is the decision complexity if flows can be moved arbitrarily often, especially with intermediate paths? Is the "longest" fastest update schedule for unsplittable flows: linear, polynomial or exponential, or even worse? Related questions are also open for flows of unit or integer size in general.

The problem of migrating splittable flows without memory limits and without intermediate paths is still not studied either: It seems as if the methods of [7] and [19] also apply to this case, but a formal proof is missing.

7. RELAXING SAFETY GUARANTEES

So far we studied network updates from the viewpoint that consistency in the respective model must be maintained, e.g., no forwarding loops should appear at any time. In situations where the computation is no longer tractable or the consistency property cannot be maintained at all, some of the discussed works opted to break consistency in a controlled manner.

An orthogonal approach is to relax the consistency safety guarantees, and try to minimize the time the network is in an inconsistent state, with underlying protocols being able to correct the induced problems (e.g., dropped packets are re-transmitted), as done in a production environment in Google *B4* [35].

One idea mainly investigated by Mizrahi *et al.* [73] is to synchronize the clocks in the switches s.t. network updates can be performed simultaneously: With perfect clock synchronization and switch execution behavior, at least in theory, e.g., loop freedom could be maintained. As the standard Network Time Protocol (NTP) does not have sufficient synchronization behavior, the Precision Time Protocol (PTP) was adapted to SDN environments in [66, 67], achieving microsecond accuracy in experiments. However, even if the time is synchronized well enough, there will be unpredictable variations of command execution time from network switches [37], motivating the need for prediction-based scheduling methods [69, 71]. Even worse, if a switch fails to update at all, the network can stay in an inconsistent state until the controller is notified, then either rolling back the update on the other switches or computing another update. Additionally, ongoing message overhead for time synchronization is required in the whole network, and controller-to-switch messages can be delayed/lost. In contrast, at the expense of additional updates, sequential approaches can verify the application of sent network updates one by one, possibly moving forward (to the next update) or back (if a command is not received or not yet applied) with no risk of incurring ongoing safety violations.

Nonetheless, in some situations synchronized updates can be considered optimal: E.g., consider the case in Fig. 11 where two unsplittable flows need to be swapped [65], with no alternative paths in the network available for the final links. Then, synchronizing the new flow paths can minimize the induced congestion [70].

Still, timed updates cannot guarantee packet consistency on their own, as packets that are currently on-route will encounter changed forwarding rules at the next switch. In [64] some additional methods are discussed how to still guarantee packet consistency by, e.g., temporarily storing traffic at the switches.

Time can be used similarly to a 2-phase commit though, by analogously using timestamps in the packet header as tags during the update [72], with [72] also showing an efficient implementation using timestamp-based TCAM ranges. Additional memory, as in the 2-phase commit approach of Reitblatt *et al.* [82], will be used for this method, but packets only need to be tagged implicitly by including the timestamp (where often 1 bit suffices [68, 72]).

8. FROM THEORY TO PRACTICE

As a complement to the previously-described theoretical and algorithmic results, we now provide an overview on practical challenges to ensure consistent network updates. We also describe how previous works tackled those challenges in order to build automated systems that can automatically carry out consistent updates.

1. **Ensuring basic communication with network devices:** Automated update systems classically rely on a logically-centralized coordinator, which must interact with network devices to both instruct them to apply operations (in a given order). Such a device-coordinator interaction requires a communication channel. Update coordinators in traditional networks typically exploit the command line interface of devices [11, 93]. In SDN networks, the interaction is simplified by their very architecture, since the coordinator is typically embodied by the SDN controller which must be already able to program (e.g., through OpenFlow [62] or similar protocols) and monitor (e.g., thanks to a Network Information Base [47]) the controlled devices.

2. **Applying operational sequences, step by step:** Both devices and the device-coordinator communication are not necessarily reliable. For example, messages sent by the coordinator may be lost or not be applied by all devices upon reception [37]. Those possibilities are typically taken into account in the computation of the update sequence (see §3). However, an effective update system must also ensure that operations are actually applied as in the computed sequences, e.g., before sending operations in the next update step. To this end, a variety of strategies are applied in the literature, from dedicated monitoring approaches (based on available network primitives like status-checking commands and protocols [11] or lower-level packet cloning mechanisms [93]) of traditional networks to acknowledgement-based protocols implemented by SDN devices [49].

3. **Working around device limitations:** Applying carefully-computed operational sequences ensures update consistency but not necessarily performance (e.g., speed), as the latter also depends on device efficiency in executing operations. This aspect has been analyzed by several works, especially focused

Reference	Approach	(Un-)splittable model	Intermediate paths	Computation	# Updates	Complete (decides if consistent migration exists)
[82]	Install old and new rules, then switch from old to new	Both, move each flow only once	No	Polynomial	1	No bandwidth guarantees
[31]	Partial moves according to free slack capacity s	Splittable	No	Polynomial	$\lceil 1/s \rceil - 1$	Requires slack on flow links
[37]	Greedy traversal of dependency graph	Both, move each flow only once	No	Polynomial	Linear	No (rate-limit flows to guarantee completion)
[56]	MIP of [37]	Both, move each flow only once	No	Exponential	Linear	Yes
[102]	Minimize transient congestion for fixed number of x intermediate states via LP	Both	No	Polynomial	Any $x \in \mathbb{N}$	For any given x, approx. min. transient congestion by $\log n$ factor
			Yes	*Exponential*		
[102]	... via MIP	Both	Both	*Exponential*	Any $x \in \mathbb{N}$	For any given x yes, but cannot decide in general
[31]	Binary search of intermediate states via LP	Splittable	Yes	Polynomial in # of updates	Unbounded	Cannot decide if migration possible
[7]	Create slack with intermediate states, then use partial moves of [31]	Splittable	Yes	Polynomial	Unbounded	Yes
[19]	Split unsplittable flows along old and new paths	2-Splittable	No	Polynomial	Unbounded	Yes
[6]	Use augmenting flows to find updates	Splittable, 1 dest., paths not fixed	Yes	Polynomial	Linear	Yes
Further practical extensions						
[52]	Extends approach of *SWAN* [31] in a data center setting					
[100]	Extends approach of *Dionysus* [37] with local dependency resolving					
[77]	Considers reconfiguration for dynamic flow arrivals					
[57]	Allows for (un-)splittable flow migration (move each once) with user-specified deadlines and requirements via MIP (or LP heuristic)					

Table 2: Compact overview of flow migration algorithms

Flow migration problem	Intermediate paths	Memory restrictions	Decision problem hardness
Unsplittable	Yes	Yes	NP-hard [7]
		No	
	No	Yes	NP-hard [19]
		No	
Unit size	Yes	Yes	NP-hard [7]
		No	
	No	Yes	Open (also for integer size)
		No	
Splittable	Yes	Yes	NP-hard [37]
		No	P [7]
	No	Yes	NP-hard [37]
		No	Open
Move every flow only once	Yes	Yes	Not allowed (model)
		No	
	No	Yes	NP-complete [37]
		No	NP-complete [19]

Table 3: Table summarizing decision problem results for flow migration. In general, it is unknown if flow migration is in NP if flows can be moved more than once, except for the case of splittable flows without memory restrictions. We note that if a problem is NP-hard without memory restrictions, it is also NP-hard with memory restrictions.

Ref.	Reduction via	(Un-)splittable model	Intermediate paths	Memory limits	Decision problem in general	Optimization problems/remarks
[37]	Partition	Splittable	No	Yes	NP-hard	NP-complete if every flow may only move once
[37]	Partition	Splittable	No	No	–	NP-hard (fewest rule modifications)
[7]	–	Splittable	Yes	No	P	Fastest schedule can be of unbounded length, LP for new reachable demands if cannot migrate
[19]	–	2-Splittable	No	No	P	studies slightly different model
[7]	(MAX) 3-SAT	Unsplittable	Yes	No	NP-hard (also for unit size flows)	NP-hard to approx. additive error of flow removal for consistency better than $7/8 + \varepsilon$
[102]	Partition	Unsplittable	Yes & No	No	–	NP-hard (fastest schedule)
[19]	Partition	Unsplittable	No	No	NP-hard	stronger consistency model, but proof carries over
[57]	Part. & Subset Sum	Unsplittable	No	No	–	NP-hard (does a 3-update schedule exist?)

Table 4: Compact overview of flow migration hardness techniques and results

on SDN updates which are more likely to be applied in real-time (e.g., even to react to a failure). It has been pointed out that SDN device limitations impact update performance in two ways. First, SDN switches are not yet fast to change their packet-processing rules, as highlighted by several measurement studies. For example, in the Devoflow [15] paper, the authors showed that the rate of statistics gathering is limited by the size of the flow table and is negatively impacted by the flow setup rate. In 2015, He *et al.* [28] experimentally demonstrated the high rule installation latency of four different types of production SDN switches. This confirmed the results of independent studies [33, 84] providing a more in-depth look into switch performance across various vendors. Second, rule installation time can highly vary over time, independently on any switch, because it is a function of runtime factors like already-installed rules and data-plane load. The measurement campaign on real OpenFlow switches performed in Dionysus [37] indeed shows that rule installation delay can vary from seconds to minutes. Update systems are therefore engineered to mitigate the impact of those limitations – despite not avoiding per-rule update bottlenecks. Prominently, Dionysus [37] significantly reduces multi-switch update latency by carefully scheduling operations according to dynamic switch conditions. CoVisor [36] and [16] minimize the number of rule updates sent to switches through eliminating redundant updates.

4. **Avoiding conflicts between multiple control-planes:** For availability, performance, and robustness, network control-planes are often physically-distributed, even when logically centralized as in the cases of replicated SDN controllers or loosely SDN controller applications. For updates of traditional networks, the control-plane distribution is straightforwardly taken into account, since it is encompassed in the update problem definition (see §2). In contrast, additional care must be applied to SDN networks with multiple controllers: if several controllers try to update network devices at the same time, one controller may override rules installed by another, impacting the correctness of the update (both during and after the update itself). This requires to solve potential conflicts between controllers, either by pro-actively specifying how the final rules have to computed (e.g., [75]) or by reactively detecting and possibly resolving conflicts (e.g., [8]). A generalization of the above setting consists in considering multiple control-planes that may be either all distributed, all centralized, or mixed (some distributed and some centralized). Potential conflicts and general meta-algorithms to ensure consistent updates in those cases are described in [96].

5. **Updating the control-plane:** In traditional networks, data-plane changes can only be enforced by changing the configuration of control-plane protocols (e.g., IGPs). In contrast, the most studied case for SDN updates considers an unmodified controller that has to change the packet-processing rules on network switches. Nevertheless, a few works also considered the problem of entirely replacing the SDN controller itself, e.g., upgrading it to a new version or replacing the old controller with a newer one. Prominently, HotSwap [91] describes an architecture that enable the replacement of an old controller with a new one, by relying on a hypervisor that maintains a history of network events. As an alternative, explicit state transfer is used to design and implement the Morpheus controller platform in [85].

6. **Dealing with events occurring during an update:** Operational sequences computed by network update algorithms forcedly assume stable conditions. In practice, however, unpredictable concurrent events like failures can modify the underlying network independently from the operations performed to update the network. While concurrent events can be very unlikely (especially for fast updates), by definition they cannot be prevented. A few contributions assessed the impact of such unpredictable events on the update safety. For instance, the impact of link failures on SITN-based IGP reconfigurations is experimentally evaluated in [94]. Another example is represented by the recent FOUM work [32], that aims at guaranteeing per-packet consistency in the presence of an adversary able to perform packet-tampering and packet-dropping attacks.

9. FUTURE RESEARCH DIRECTIONS

While we have already identified specific open research questions in the corresponding sections, we now discuss more general areas which we believe deserve more attention by the research community in the future.

1. **Charting the complexity landscape:** Researchers have only started to understand the computational complexities underlying the network update problem. In particular, many NP-hardness results have been derived for general problem formulations for all three of our consistency models: connectivity consistency, policy consistency, and performance consistency. So far, only for a small number of specific models polynomial-time optimal algorithms are known. Even less is known about approximation algorithms. Accordingly, much research is required to chart a clearer picture of the complexity landscape of network update problems. We expect that some of these insights will also have interesting implications on classic optimization problems.

2. **Refining our models:** While we believe that today's network models capture well the fundamental constraints and trade-offs in consistent network update problems, these models are still relatively simple. In partiular, we believe that there is room and potential for developing more refined models. Such models could for example account for additional performance aspects (e.g., the impact of packet reorderings on throughput). Moreover, they could e.g., better leverage predictable aspects and models, e.g., empirical knowledge of the network behavior. For example, the channel between SDN controller and OpenFlow switches may not be completely asynchronous, but it is reasonable to make assumptions on the upper and lower bound of switch update times.

3. **Considering new update problems:** We expect future update techniques to ensure consistency of higher-level network requirements (like NFV, path delay, etc.), the same way as recent SDN controllers are supporting them.

4. **Dealing with distributed control planes:** We believe that researchers have only started to understand the design and implication of more distributed SDN control planes. In particular, while for dependability and performance purposes, future SDN control planes are likely to be distributed, this also introduces additional challenges in terms of consistent network updates and controller coordination.

10. CONCLUSION

The purpose of this survey was to provide researchers active in or interested in the field of network update problems with an overview of the state-of-the-art, including models, techniques, impossibility results as well as practical challenges. We also presented a historical perspective and discussed the fundamental new challenges introduced in Software-Defined Networks, also relating them to classic graph-theoretic optimization problems. Finally, we have identified open questions for future research.

11. REFERENCES

[1] R. Alimi, Y. Wang, and Y. R. Yang. Shadow configuration as a network management primitive. In *Proc. ACM SIGCOMM*, 2008.

[2] S. Amiri, A. Ludwig, J. Marcinkowski, and S. Schmid. Transiently consistent sdn updates: Being greedy is hard. In *Proc. SIROCCO*, 2016.

[3] G. Baier, E. Köhler, and M. Skutella. The k-splittable flow problem. *Algorithmica*, 42(3-4):231–248, 2005.

[4] Barefoot Networks. The world's fastest and most programmable networks (white paper). https://barefootnetworks.com/white-paper/the-worlds-fastest-most-programmable-networks/, 2016.

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[6] S. Brandt, K.-T. Förster, and R. Wattenhofer. Augmenting anycast network flows. In *Proc. ICDCN*, 2016.

[7] S. Brandt, K.-T. Förster, and R. Wattenhofer. On Consistent Migration of Flows in SDNs. In *Proc. IEEE INFOCOM*, 2016.

[8] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. Software transactional networking: Concurrent and consistent policy composition. In *Proc. ACM SIGCOMM HotSDN*, August 2013.

[9] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proc. ACM SIGCOMM*, 2007.

[10] P. Cerny, N. Foster, N. Jagnik, and J. McClurg. Optimal consistent network updates in polynomial time. In *Proc. DISC*, 2016.

[11] X. Chen, Z. M. Mao, and J. Van der Merwe. PACMAN: a platform for automated and controlled network operations and configuration management. In *Proc. ACM CoNEXT*, 2009.

[12] F. Clad, P. Merindol, J.-J. Pansiot, P. Francois, and O. Bonaventure. Graceful Convergence in Link-State IP Networks: A Lightweight Algorithm Ensuring Minimal Operational Impact. *IEEE/ACM Transactions on Networking (TON)*, 22(1):300–312, February 2014.

[13] F. Clad, P. Merindol, S. Vissicchio, J.-J. Pansiot, and P. Francois. Graceful Router Updates for Link-State Protocols. In *Proc. ICNP*, 2013.

[14] F. Clad, S. Vissicchio, P. Mérindol, P. Francois, and J.-J. Pansiot. Computing minimal update sequences for graceful router-wide reconfigurations. *IEEE/ACM Transactions on Networking (TON)*, 23(5):1373–1386, 2015.

[15] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):254–265, 2011.

[16] S. Dudycz, A. Ludwig, and S. Schmid. Can't touch this: Consistent network updates for multiple policies. In *Proc. IEEE/IFIP DSN*, 2016.

[17] N. Feamster, J. Rexford, and E. Zegura. The road to sdn. *Queue*, 11(12):20:20–20:40, Dec. 2013.

[18] K.-T. Förster, R. Mahajan, and R. Wattenhofer. Consistent Updates in Software Defined Networks: On Dependencies, Loop Freedom, and Blackholes. In *Proc. IFIP Networking*, 2016.

[19] K.-T. Förster and R. Wattenhofer. The Power of Two in Consistent Network Updates: Hard Loop Freedom, Easy Flow Migration. In *Proc. ICCCN*, 2016.

[20] N. Foster, A. Guha, M. Reitblatt, A. Story, M. J. Freedman, N. P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison. Languages for software-defined networks. *IEEE Communications Magazine*, 51(2):128–134, 2013.

[21] P. Francois and O. Bonaventure. Avoiding Transient Loops During the Convergence of Link-State Routing Protocols. *IEEE/ACM Transactions on Networking (TON)*, 15(6):1280–1292, December 2007.

[22] P. Francois, O. Bonaventure, B. Decraene, and P.-A. Coste. Avoiding disruptions during maintenance operations on bgp sessions. *IEEE Transactions on Network and Service Management*, 4(3):1–11, 2007.

[23] P. Francois, M. Shand, and O. Bonaventure. Disruption-free topology reconfiguration in OSPF Networks. In *Proc. IEEE INFOCOM*, 2007.

[24] J. Fu, P. Sjodin, and G. Karlsson. Loop-Free Updates of Forwarding Tables. *IEEE Transactions on Network and Service Management*, 5(1):22–35, 2008.

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

[26] T. Hartman, A. Hassidim, H. Kaplan, D. Raz, and M. Segalov. How to split a flow? In *Proc. IEEE INFOCOM*, 2012.

[27] J. He and J. Rexford. Toward internet-wide multipath routing. *IEEE Network*, 22(2):16–21, 2008.

[28] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M. Thottan. Measuring control plane latency in sdn-enabled switches. In *Proc. ACM SIGCOMM SOSR*, 2015.

[29] G. Herrero and J. van der Ven. *Network Mergers and Migrations: Junos Design and Implementation.* Wiley, 2010.

[30] M. Hoefer, V. S. Mirrokni, H. Röglin, and S. Teng. Competitive routing over time. *Theor. Comput. Sci.*, 412(39):5420–5432, 2011.

[31] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):15–26, 2013.

[32] J. Hua, X. Ge, and S. Zhong. FOUM: A Flow-Ordered Consistent Update Mechanism for Software-Defined Networking in Adversarial Settings. In *Proc. IEEE INFOCOM*, 2016.

[33] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity switch models for software-defined network emulation. In *Proc. ACM SIGCOMM HotSDN*, pages 43–48, 2013.

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

[35] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):3–14, 2013.

[36] X. Jin, J. Gossels, J. Rexford, and D. Walker. Covisor: A compositional hypervisor for software-defined networks. In *Proc. USENIX NSDI*, pages 87–101, 2015.

[37] X. Jin, H. Liu, R. Gandhi, S. Kandula, R. Mahajan, J. Rexford, R. Wattenhofer, and M. Zhang. Dionysus: Dynamic scheduling of network updates. In *Proc. ACM SIGCOMM*, 2014.

[38] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani. Consensus routing: The internet as a distributed system. In *Proc. USENIX NSDI*, 2008.

[39] S. Kandula, D. Katabi, S. Sinha, and A. Berger. Dynamic load balancing without packet reordering. *SIGCOMM Comput. Commun. Rev.*, 37(2):51–62, 2007.

[40] S. Kandula, I. Menache, R. Schwartz, and S. R. Babbula. Calendaring for wide area networks. In *Proc. ACM SIGCOMM*, 2014.

[41] N. P. Katta, J. Rexford, and D. Walker. Incremental consistent updates. In *Proc. ACM SIGCOMM HotSDN*, 2013.

[42] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proc. USENIX NSDI*, 2013.

[43] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proc. USENIX NSDI*, 2012.

[44] E. Keller, J. Rexford, and J. Van Der Merwe. Seamless BGP migration with router grafting. In *Proc. USENIX NSDI*, 2010.

[45] R. Keralapura, C.-N. Chuah, and Y. Fan. Optimal Strategy for Graceful Network Upgrade. In *Proc. ACM SIGCOMM INM*, 2006.

[46] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. *SIGCOMM Comput. Commun. Rev.*, 42(4):467–472, Sept. 2012.

[47] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *Proc. USENIX OSDI*, 2010.

[48] N. Kushman, S. Kandula, D. Katabi, and B. M. Maggs. R-BGP: Staying connected in a connected world. In *Proc. USENIX NSDI*, 2007.

[49] M. Kuzniar, P. Peresini, and D. Kostić. Providing reliable fib update acknowledgments in sdn. In *Proc. ACM CoNEXT*, pages 415–422, 2014.

[50] F. Le, G. G. Xie, D. Pei, J. Wang, and H. Zhang. Shedding Light on the Glue Logic of the Internet Routing Architecture. In *Proc. ACM SIGCOMM*, 2008.

[51] M. Lewin, D. Livnat, and U. Zwick. Improved rounding techniques for the MAX 2-SAT and MAX DI-CUT problems. In *Integer Programming and Combinatorial Optimization*, pages 67–82. Springer, 2002.

[52] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz. zUpdate: Updating Data Center Networks with Zero Loss. In *Proc. ACM SIGCOMM*, 2013.

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

[54] A. Ludwig, J. Marcinkowski, and S. Schmid. Scheduling loop-free network updates: It's good to relax! In *Proc. ACM PODC*, 2015.

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

[56] L. Luo, H. Yu, S. Luo, and M. Zhang. Fast lossless traffic migration for SDN updates. In *Proc. IEEE ICC*, pages 5803–5808. IEEE, 2015.

[57] S. Luo, H. Yu, L. Luo, and L. Li. Arrange your network updates as you wish. In *Proc. of IFIP Networking*, 2016.

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

[59] J. McClurg, H. Hojjat, P. Cerny, and N. Foster. Efficient Synthesis of Network Updates. In *ACM SIGPLAN PLDI*, 2015.

[60] R. McGeer. A safe, efficient update protocol for openflow networks. In *Proc. SIGCOMM HotSDN*, pages 61–66, 2012.

[61] R. McGeer. A correct, zero-overhead protocol for network updates. In *Proc. SIGCOMM HotSDN*, pages 161–162, 2013.

[62] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

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

[64] T. Mizrahi and Y. Moses. Time-based updates in software defined networks. In *Proc. ACM HotSDN*, pages 163–164, 2013.

[65] T. Mizrahi and Y. Moses. On the necessity of time-based updates in SDN. In *Proc. USENIX ONS*, 2014.

[66] T. Mizrahi and Y. Moses. Reverseptp: A software defined networking approach to clock synchronization. In *Proc. ACM HotSDN*, 2014.

[67] T. Mizrahi and Y. Moses. Using REVERSEPTP to Distribute Time in Software Defined Networks . In *Proc. IEEE ISPCS*, 2014.

[68] T. Mizrahi and Y. Moses. The case for data plane timestamping in sdn. In *Proc. IEEE INFOCOM SWFAN*, 2016.

[69] T. Mizrahi and Y. Moses. Oneclock to rule them all: Using time in networked applications. In *Proc. IEEE/IFIP NOMS*, 2016.

[70] T. Mizrahi and Y. Moses. Software defined networks: It's about time. *Proc. IEEE INFOCOM*, 2016.

[71] T. Mizrahi and Y. Moses. Time Capability in NETCONF. In *RFC 7758*, 2016.

[72] T. Mizrahi, O. Rottenstreich, and Y. Moses. Timeflip: Scheduling network updates with timestamp-based tcam ranges. In *Proc. IEEE INFOCOM*, pages 2551–2559, 2015.

[73] T. Mizrahi, E. Saat, and Y. Moses. Timed consistent network updates in software defined networks. *Trans. on Netw.*, 2016.

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

[75] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *Proc. USENIX NSDI*, 2013.

[76] J. Moy, P. Pillay-Esnault, and A. Lindem. Graceful OSPF Restart. RFC 3623, 2003.

[77] S. Paris, A. Destounis, L. Maggi, G. S. Paschos, and J. Leguay. Controlling flow reconfigurations in sdn. In *Proc. IEEE INFOCOM*, 2016.

[78] I. Pepelnjak. Changing the Routing Protocol in Your Network, 2007.

[79] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized "zero-queue" datacenter network. *SIGCOMM Comput. Commun. Rev.*, 44(4):307–318, Aug. 2014.

[80] S. Raza, Y. Zhu, and C.-N. Chuah. Graceful Network Operations. In *Proc. IEEE INFOCOM*, 2009.

[81] S. Raza, Y. Zhu, and C.-N. Chuah. Graceful Network State Migrations. *IEEE/ACM Transactions on Networking (TON)*, 19(4):1097 –1110, 2011.

[82] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proc. ACM SIGCOMM*, pages 323–334, 2012.

[83] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent updates for software-defined networks: Change you can believe in! In *Proc. ACM HotNets*, 2011.

[84] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. Oflops: An open framework for openflow switch evaluation. In *Proc. PAM*, 2012.

[85] K. Saur, J. Collard, N. Foster, A. Guha, L. Vanbever, and M. Hicks. Safe and Flexible Controller Upgrades for SDNs. In *SOSR*, 2016.

[86] A. Shaikh, R. Dube, and A. Varma. Avoiding instability during graceful shutdown of multiple OSPF routers. *Trans. on Netw.*, 14:532–542, June 2006.

[87] M. Shand and S. Bryant. A Framework for Loop-Free Convergence. RFC 5715, IETF, January 2010.

[88] M. Shand and L. Ginsberg. Restart Signaling for IS-IS. RFC 5306, 2008.

[89] L. Shi, J. Fu, and X. Fu. Loop-Free Forwarding Table Updates with Minimal Link Overflow. In *Proc. IEEE ICC*, 2009.

[90] United. United Airlines Restoring Normal Flight Operations Following Friday Computer Outage. http://newsroom.united.com/news-releases?item=124170, 2011.

[91] L. Vanbever, J. Reich, T. Benson, N. Foster, and J. Rexford. HotSwap: Correct and Efficient Controller Upgrades for Software-defined Networks. In *Proc. ACM SIGCOMM HotSDN*, 2013.

[92] L. Vanbever, S. Vissicchio, L. Cittadini, and O. Bonaventure. When the cure is worse than the disease: the impact of graceful igp operations on bgp. In *Proc. IEEE INFOCOM*, 2013.

[93] L. Vanbever, S. Vissicchio, C. Pelsser, P. François, and O. Bonaventure. Seamless network-wide igp migrations. In *Proc. ACM SIGCOMM*, pages 314–325, 2011.

[94] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure. Lossless migrations of link-state igps. *IEEE/ACM Transactions on Networking (TON)*, 20(6):1842–1855, 2012.

[95] S. Vissicchio and L. Cittadini. FLIP the (Flow) Table: Fast LIghtweight Policy-preserving SDN Updates. In *Proc. IEEE INFOCOM*, 2016.

[96] S. Vissicchio, L. Cittadini, O. Bonaventure, G. G. Xie, and L. Vanbever. On the Co-Existence of Distributed and Centralized Routing Control-Planes. In *Proc. IEEE INFOCOM*, 2015.

[97] S. Vissicchio, L. Vanbever, L. Cittadini, G. Xie, and O. Bonaventure. Safe Update of Hybrid SDN Networks. Technical report, UCLouvain, 2013.

[98] S. Vissicchio, L. Vanbever, L. Cittadini, G. Xie, and O. Bonaventure. Safe Routing Reconfigurations with Route Redistribution. In *Proc. IEEE INFOCOM*, 2014.

[99] S. Vissicchio, L. Vanbever, C. Pelsser, L. Cittadini, P. Francois, and O. Bonaventure. Improving network agility with seamless bgp reconfigurations. *IEEE/ACM Transactions on Networking (TON)*, 21(3):990–1002, June 2013.

[100] W. Wang, W. He, J. Su, and Y. Chen. Cupid: Congestion-free consistent data plane update in software defined networks. In *Proc. IEEE INFOCOM*, 2016.

[101] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, and J. Rexford. Virtual routers on the move: live router migration as a network-management primitive. In *Proc. ACM SIGCOMM*, 2008.

[102] J. Zheng, H. Xu, G. Chen, and H. Dai. Minimizing transient congestion during network update in data centers. In *Proc. ICNP*, 2015.

[103] W. Zhou, D. K. Jin, J. Croft, M. Caesar, and P. B. Godfrey. Enforcing customizable consistency properties in software-defined networks. In *Proc. USENIX NSDI*, pages 73–85. USENIX Association, 2015.