Survey of Consistent Software-Defined Network Updates

Klaus-Tycho Foerster, Stefan Schmid, and Stefano Vissicchio

***Abstract*—Computer networks have become a critical infrastructure. In fact, networks should not only meet strict requirements in terms of correctness, availability, and performance but they should also be very flexible and support fast updates, e.g., due to policy changes, increasing traffic, or failures. This paper presents a structured survey of mechanism and protocols to update computer networks in a fast and consistent manner. In particular, we identify and discuss the different desirable consistency properties that should be provided throughout a network update, the algorithmic techniques which are needed to meet these consistency properties, and the implications on the speed and costs at which updates can be performed. We also explain the relationship between consistent network update problems and classic algorithmic optimization ones. While our survey is mainly motivated by the advent of software-defined networks and their primary need for correct and efficient update techniques, the fundamental underlying problems are not new, and we provide a historical perspective of the subject as well.**

***Index Terms*—Computer network management, software defined networking, algorithms.**

I. 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 [1]. 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 [2]–[5]. An anecdote reported in [1] 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 [1].

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 foster 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 [6], [7]. Moreover, as rules can be defined over multiple OSI layers, the distinction between switches and routers (and even simple middleboxes [8]) 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

Manuscript received February 8, 2018; revised August 12, 2018; accepted October 15, 2018. Date of publication October 19, 2018; date of current version May 31, 2019. *(Corresponding author: Klaus-Tycho Foerster.)*

K.-T. Foerster and S. Schmid are with the Faculty of Computer Science, University of Vienna, 1080 Vienna, Austria (e-mail: klaus-tycho.foerster@univie.ac.at; stefan_schmid@univie.ac.at).

S. Vissicchio is with the Department of Computer Science, University College London, London WC1E 6BT, U.K. (e-mail: s.vissicchio@cs.ucl.ac.uk).

Digital Object Identifier 10.1109/COMST.2018.2876749

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 differ 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 is the number of research results in the area 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, many of the underlying update problems are not entirely new or specific to SDN: rather, techniques to consistently update legacy networks have been studied in the literature, although they are based on the (more restrictive) primitives available in traditional protocols (e.g., IGP weights).

We therefore believe that it is time for a comprehensive survey of the subject.

A. The Network Update Problem

Any dependable network does not only need to maintain certain invariants, related to correctness, availability, and performance, but also needs to be flexible in how it process packets.

1) Flexibility: Flexibility implies that networks have to be updated, e.g., to support the following use cases.

a) Security policy changes: For example, in enterprise networks, traffic from a specific subnetwork may have to be routed via a firewall if specific alarms are raised. Similarly, in wide-area networks, the countries that must be avoided by sensitive traffic can change over time.

b) Traffic engineering: To improve performance metrics (e.g., minimizing the maximal link load), a network operator may decide to reroute some traffic to different paths. For example, many Internet Service Providers change their paths during the day, depending on the expected load or in reaction to external changes (e.g., a policy modification from a content provider).

c) 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.

d) 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.

e) Service relocation: Networks typically run several services, from in-network packet processing functions (e.g., virtualized middleboxes) to applications (like data storage or application servers). Addition, removal or relocation of any of those services would require a network update, i.e., to reroute traffic for the affected service.

2) Maintaining Consistency: It is often desirable that the network maintains certain consistency properties *throughout the update*. Those properties may include per-packet path consistency (a packet should be forwarded along the old or the new route, but never a mixture of both), waypoint enforcement (a packet should never bypass a firewall), or at least correct packet delivery (at no point in time packets should be dropped or trapped in a loop).

3) Towards SDNs: While the above reasons for network updates are relevant independently of the adopted paradigm, the decoupling of control- and data-plane, as well as the flexibility allowed by the SDN architecture are likely to increase the frequency of network updates, e.g., for supporting more fine-grained and frequent optimization of traffic paths [9].

B. Our Contributions

This paper presents a comprehensive survey of the consistent network update problem in Software-Defined Networks (SDNs). In the basic scenario assumed by most prior contributions, an SDN network is controlled by a single controller, which needs to preserve specific consistency properties at each and every moment during the update. Preserving such properties is often argued to be more important that the induced inability to guarantee perfect network availability or partition tolerance simultaneously—e.g., to avoid packet losses or security breaches. This impossibility result follows from the celebrated CAP theorem [10], which also applies to control algorithms used in networks [11]. Throughout this paper, we first consider this basic scenario and then extend the discussion to network updates with distributed SDN controllers and different consistency models [12]–[14]. The goal of our survey is to both (1) provide active researchers in the field with an overview of the state-of-the-art literature, and (2) help researchers who only recently became interested in the subject bootstrap and learn about open research questions.

In discussing the literature, we identify and compare the consistency properties (absence of forwarding loops and blackholes, policy preservation, congestion avoidance, etc.) and performance objectives (update duration, maximum link overload during the update, etc.) considered by the scientific literature. We provide an overview of the algorithmic techniques required to solve specific classes of network update problems, and discuss the inherent tradeoffs 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.

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.

C. Paper Organization

The remainder of this paper is organized as follows. Section II presents a historical perspective and reviews notions of consistency and techniques both in traditional computer networks as well as in Software-Defined Networks. Section III then presents a classification and taxonomy of the different variants of the consistent network update problems. Sections IV–VI review models and techniques for connectivity consistency, policy consistency, and capacity consistency related problems, respectively. Section VII-A discusses proposals to further relax consistency guarantees by introducing tighter synchronization. In Section VIII, we identify practical challenges. After highlighting future research directions in Section IX, we conclude our paper in Section X.

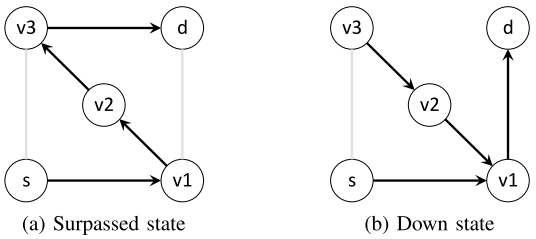


Fig. 1. A network update example, where forwarding paths have to be changed from the Surpassed (Fig. 1(a)) to the Down (Fig. 1(b)) state. Arrows represent paths on which traffic (e.g., from s to d) is forwarded, while (gray) undirected links between nodes represent unused links.

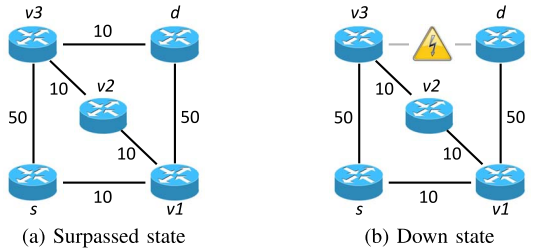


Fig. 2. Possible implementation of pre- and post-update forwarding paths for the update in Fig. 1 in a traditional, IGP-based network. Numbers close to network links represent the corresponding IGP weights.

II. History of the Network Update Problem From the Origins to SDN

Any computer network must guarantee some consistency properties for the configured forwarding rules and paths. For example, forwarding loops must be avoided, as they can quickly deplete switch buffers and harm the availability and connectivity provided by a network.

It is eminently desirable to preserve consistency properties *during network updates*—i.e., while changing packet-processing rules on network devices. In fact, early studies on consistent network updates date back long before the advent of software-defined networking. In this section, we provide a historical perspective on the many research contributions that can be considered as the main precursors of the state of the art for SDN updates.

We first discuss update problems and techniques in traditional networks (Sections II-A and II-B). In those networks, forwarding rules are computed by routing protocols that run standardly-defined distributed algorithms, whose output is influenced by both physical topology (e.g., active links) and routing configurations (e.g., logical link costs). Pioneering update works aimed at avoiding transient inconsistencies due to modified topology or configurations, mainly focusing on the Interior Gateway Protocols (IGPs) that are commonly used to 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 sequences of IGP configuration changes that modify forwarding while guaranteeing forwarding consistency, e.g., for service continuity (Section II-A). 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 (Section II-B).

Subsequently (Section II-C), we discuss update problems in the context of SDNs. Those networks are based on a clear separation between controller (implementing the control logic) and dataplane elements (applying controller's decision on packets). This separation indisputably 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.

Throughout the section, 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 when we update nodes one by one, because links ($v1$, $v2$) and ($v2$, $v3$) are traversed in opposite directions before and after the update.

A. IGP Reconfigurations

In traditional (non-SDN) networks, forwarding paths are computed by distributed routing protocols. Among them, link-state IGPs are typically used to compute forwarding paths within a network owned by the same administrative entity. They are based on computing shortest-paths on a logical view of the network, that is, a weighted graph which is shared across routers. Parameters influencing IGP computations, like link weights, can be set by operators by editing router configurations.

As an illustration, Fig. 2 shows a possible IGP implementation for the network states shown 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. 2(a) and 2(b) respectively produce the forwarding paths depicted in Figs. 1(a) and 1(b).

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 timing and ordering 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 ($v3$, d) from the IGP graph shown in Fig. 2(a). This will eventually lead us to the configuration presented in Fig. 2(b). Before the final state is reached, the notification that ($v3$, d) is removed has to be propagated to all routers. If $v3$ receives such notification before $v2$ (e.g., because closer to the removed link), then $v3$ would recompute its next-hop based on the new information, and starts forwarding packets for d to $v2$ (see Fig. 1(b)). Nevertheless, $v2$ keeps forwarding packets to $v3$ as it considers that ($v3$, d) is still up. This creates a loop between $v3$ and $v2$: The loop remains until $v2$ is notified about the removed link. A similar loop can occur between $v2$ and $v1$.

Guaranteeing disruption-free IGP operations has been considered by research and industry since almost two decades. We now briefly report on the main proposals in the area, which share the focus on support for planned operations.

1) Protocol Extensions: Early contributions focused on the modification of IGPs, mainly to avoid forwarding inconsistencies. Among them, protocol extensions have been proposed [15]–[17] to gracefully restart a routing process, that is, to avoid forwarding disruptions (e.g., blackholes) during the software upgrade of a router. Other works focused on avoiding forwarding loops during configuration changes. For example, François and Bonaventure [18] propose oFIB, an IGP extension that guarantees the absence of forwarding loops upon manually managed topological changes, e.g., to propagate information about a link or a router that has to be shut down for maintenance. The key intuition behind oFIB is to use explicit synchronization between routers to constrain the order in which each router changes its forwarding entries. Namely, each router is forced not to update its forwarding entry for a given destination until all its final next-hops switched to their respective final next-hops for that destination. Consider again Fig. 2, assuming that oFIB is deployed. To prepare the shutdown of link ($v3$, d), oFIB ensures that $v1$ is the only router changing its forwarding entry to d at first: this is safe because $v1$'s final next-hop is directly the destination d. All the other routers (e.g., $v2$) do not update yet until their final next-hops (e.g., $v1$) use their respective final paths. In fact, after $v1$ starts forwarding traffic through the ($v1$, d) link, it notifies its neighbors about its updated state. At that point, $v2$ and s can update their respective forwarding entries for d. The whole network is eventually updated by iterating this process.

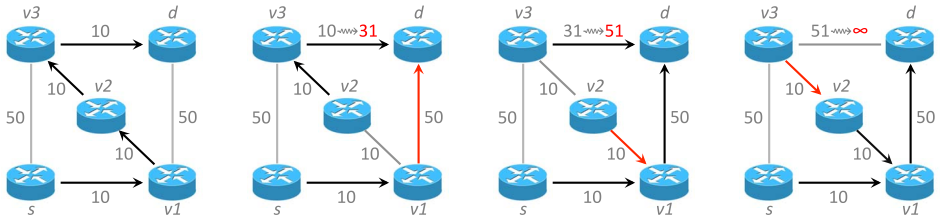
oFIB inspired a number of variants, aiming at applying explicit notification to more generic updates. Fu *et al.* [19] generalizes the approach by defining a loop-free ordering of IGP-entry updates for arbitrary forwarding changes. Shi *et al.* [20] also extends the reconfiguration mechanism to avoid traffic congestion in addition to forwarding incorrectness. A broader overview of loop avoidance and mitigation techniques mostly inspired by oFIB is reported in [21].

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 [18], 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 targeted to specific reconfiguration cases (e.g., single-link failures), since it is intrinsically hard to predict the impact of any possible configuration change on forwarding paths. Finally, protocol extensions are not easy to implement in practice, because of the reluctance of vendors to change their proprietary router software, as well as the additional complexity and potential overhead (e.g., load) induced on routers.

Limited practicality of protocol modifications quickly motivated new approaches, based on coordinating operations readily available in deployed routers, at a per-router level.

2) Coarse-Grained Operation Scheduling: Planned reconfigurations may encompass several coarse-grained operations. Consider the case where in Fig. 2(a), the weight of link (s, $v1$) has to be set to 70 in addition to removing the link ($v3$, d). The link reweighting might be desirable to improve load balancing across the network, e.g., adapting to a permanently increased volume of traffic from s to d. Such a reconfiguration effectively consists of two macro operations: removing ($v3$, d) and changing the weight of (s, $v3$). Even assuming that each coarse-grained operation is atomic, the order in which distinct operations are performed can have an impact on how much the network traffic is disrupted during the reconfiguration. Assume that link (s, $v3$) can sustain no more than 50% of the $s{-}d$ traffic volume. Reweighting (s, $v1$) before removing ($v3$, d) forces all the $s{-}d$ traffic on the path (s, $v3$, d) which overloads link (s, $v3$), while removing ($v3$, d) first would not cause congestion.

Several works propose to use optimization techniques to compute the order of macro-operations so as to guarantee given consistency properties. As a first approach, Keralapura *et al.* [22] formalized the problem of finding the optimal order in which nodes can be added to a network, one by one, so as to minimize an objective function modeling typical costs of connectivity and traffic disruptions in Internet Service Providers. The following contributions encompass additional operations. In 2009 [23] and 2011 [24], for instance, Raza *et al.* propose a theoretical framework to schedule link weight changes in a way that minimizes a generic disruption function. This approach enables to formulate our reconfiguration example as a formal optimization problem, where



(a) Initial (b) Step 1 (c) Step 2 (d) Final

Fig. 3. Intermediate IGP weights that enable a loop-free reconfiguration for the example in Fig. 1.

constraints enforce that the reweighting of links (s, $v1$) and ($v3$, d) (from 10 to 70 and from 10 to ∞, respectively) are both scheduled, and the objective functions aggregates the cost of every step in the schedule. The works also describe two algorithms to solve the formalized problems, one based on dynamic programming and the other on an ant colony optimization heuristic.

The approaches just described basically spread coarse-grained operations over time. This is not sufficient to deal with many update scenarios. Fig. 2 displays one of such scenarios: since the reconfiguration includes a single coarse-grained operation (link removal), previous approaches cannot prevent forwarding loops possibly occurring when that single operation is performed.

3) Progressive Link Reweighting: Intermediate link weights can be set to avoid disruptions during a reconfiguration, even if it includes a single link weight change. Consider again the example in Fig. 2, and let the final weight for link ($v3$, 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 ($v3$, d), as illustrated in Fig. 3. The first of those intermediate weights (see Fig. 3(b)) is used to force $v1$ and only $v1$ to change its next-hop, from $v2$ to d: Intuitively, this prevents the loop between $v2$ and $v1$. The second intermediate weight (see Fig. 3(c)) similarly guarantees that the loop between $v3$ and $v2$ is avoided, i.e., by forcing $v2$ to use its final next-hop before $v3$.

Of course, computing intermediate weights that guarantee the absence of disruptions becomes 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 ($v3$, d) to 31 by editing the configuration of $v3$ and d, then check that the all IGP routers have converged on the paths in Fig. 3(b), repeat similar operations to reach the state in Fig. 3(c), and finally remove the link safely. Even better, François *et al.* [25] have proved that it is always possible to compute a sequence of intermediate link weights that provably avoids all transient loops when a single link has to be reweighted. Obviously, the weight of multiple links can be changed in a loop-free way, by safely reweighting links one by one.

Additional research contributions 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

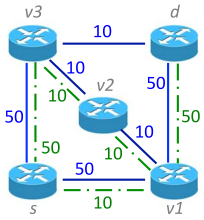


Fig. 4. Ships-in-the-Night setup: All routers run two routing processes, one with the initial configuration (blue, solid lines) and the other with the final configuration (green, dashed and dotted segments).

potentially to be taken into account when changing link weights. Polynomial-time algorithms have been proposed to support planned operations at the per-link [25], [26] (e.g., single-link reweighting) and at a per-router [27], [28] (e.g., router shutdown/addition) granularity.

4) Ships-in-the-Night (SITN) Techniques: To improve the update speed in the case of simultaneous link weight changes and to deal with additional reconfiguration scenarios (from changing routing parameters like OSPF areas 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. When multiple configurations are installed on the same node, only one of them is preferred and used. 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 between the initial and the final configurations on every node, typically one by one. Configuration preference can be swapped at a per-destination granularity. This means that (1) for each destination, every node either forwards packets to its initial next-hops or its final ones; (2) at any time during the reconfiguration, distinct nodes can use different configurations; hence, (3) inconsistencies may arise from the mismatch between the configurations used by distinct nodes.

Because of those potential inconsistencies, the Ships-in-the-Night approach opens a new algorithmic problem, that is, to decide a safe order in which to swap preferences on a per-router basis. For example, if the configuration preference is

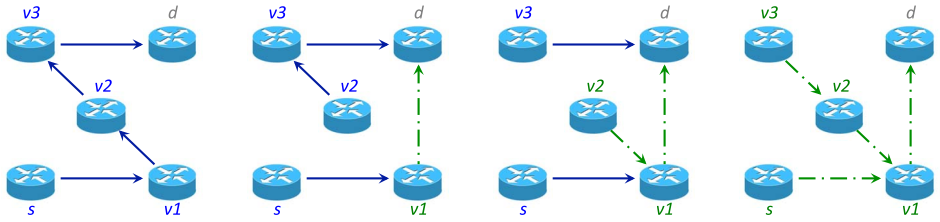


Fig. 5. Ships-in-the-Night reconfiguration that mimics the progressive link reweighting shown in Fig. 3. In each figure, the colors of router names indicate their respective control-plane preferences at the represented reconfiguration step.

swapped on $v3$ before doing the same on $v2$ in Fig. 4, we end up with a loop between $v2$ and $v3$. In contrast, Fig. 5 shows a SITN-based safe reconfiguration that mimics the progressive link weight increment depicted in Fig. 3.

Naive algorithms for 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 they potentially trigger packet losses and service disruptions (in addition to massive control-plane message exchanges). Such an approach will also leave the network in an inconsistent, disrupted and hard-to-troubleshoot state if any reconfiguration command is lost or significantly delayed. Industrial best practices (e.g., [29] and [30]) only provide rules of thumb which do not apply in the general case, and do not guarantee lossless reconfiguration processes.

Hence, the problem of computing a safe per-router reconfiguration order within SITN called for new research contributions. Prominently, [31], [32] show that no SITN-based update order guarantees the absence of forwarding loops in some cases, and it is NP-complete to even assess whether a loop-free order exists. Those papers also propose two algorithms to compute a loop-free order. The first algorithm is based on applying traditional optimization algorithms to solve a Linear Program (LP) that models the input update problem. Such an LP is derived from enumerating all possible loops, and encoding all the possibilities to avoid every loop as LP constraints (that must be alternatively satisfied). For the example in Fig. 4, this algorithm would enumerate the two potential loops (between $v1$ and $v2$, and between $v2$ and $v3$), and formulate LP constraints forcing the control-plane preference to be swapped at $v1$ before $v2$ (to break the first loop) and at $v2$ before $v3$ (to break the second loop). The second algorithm is a heuristic based on reconfiguring routers according to the final paths (e.g., $v1$ before $v2$ and s in our example, because $v1$ is closer to d), aimed to avoid the scalability problems of the LP-based approach. Finally, [31], [32] describe a comprehensive system to carry out loop-free SITN-based reconfigurations. The system computes the operational sequence to perform an input reconfiguration, and directly interacts with the routers to modify their configurations, and check when every operation in the sequence is completed. The system was envisioned to work semi-automatically, waiting for an explicit confirmation from the operator before performing the next operation in the computed sequence.

B. Generalized Routing Reconfigurations in Traditional Networks

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

1) 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* [33], a mechanism enabling the propagation of information from one routing domain (e.g., running an IGP) to another (e.g., running another IGP). Route redistribution may be responsible for both routing (inability to converge to a stable state) and forwarding (e.g., loop) anomalies [33]. SITN-based update procedures have been proposed in [34] to avoid transient anomalies while (i) reconfiguring a specific routing domain, and/or (ii) arbitrarily changing the size and shape of routing domains.

2) Internet Service Providers (ISPs), With BGP and MPLS: In ISP networks, the Border Gateway Protocol (BGP) and often the Multi-Protocol Label Switching (MPLS) protocol are pervasively used to manage transit traffic, for which both the source and the destination is external to the network. Vanbever *et al.* [35] 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 the presence of MPLS or carefully configuring BGP (according to some guidelines).

In parallel, a distinct set of techniques aimed at supporting BGP reconfigurations. François *et al.* [36] propose a solution to avoid churn and loss of connectivity due to planned BGP session shutdown: This solution is based on modifying BGP to distribute alternate routes and move traffic on them *before* the target BGP session is actually removed. Wang *et al.* [37] present an approach, based on extending virtual machine migration techniques, to quickly transfer virtual routers from one physical device to another. Keller *et al.* [38] address the more general problem of fastly migrating parts of the BGP configuration (e.g., transferring a BGP session from one router to another), with a technique that takes care of moving the BGP state to the new route and reduce the impact of the migration on both BGP peers and other routers. Vissicchio *et al.* [39] describe a framework that enables radical re-organizations of BGP sessions (e.g., changing several of

them, or transforming a full-mesh into a route reflector topology) while guaranteeing the absence of forwarding and routing anomalies: The framework is based on implementing Ships-in-the-Night in BGP (with a minimal extension to existing routers), and tagging packets so that routers can uniformly apply a single BGP configuration to every packet.

Finally, Internet-level problems, like maintaining global connectivity upon failures, have also been explored (see [40]).

3) 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 been studied as well in the literature. Mainly, [41] generalizes SITN by proposing a new design for the internal router architecture. This re-design would allow routers not only to run multiple configurations simultaneously, and to select the configuration to apply for every packet on the basis of a specific bit in the packet header. The work also describes a commit protocol to support the switch between configurations without creating forwarding loops. General mechanisms for consensus routing have also been explored in [42].

C. Updates of Software-Defined Networks

Recently, software-defined networking has grown in popularity, thanks to its promises to spur abstractions, mitigate compelling management problems and avoid network ossification [43]. SDN is currently used or discussed in a wide range of contexts, e.g., to improve network virtualization in datacenters, generalize traffic engineering in the wide-area network, or enable slicing in emerging 5G applications, to just name a few.

In pure SDN networks, rather than having devices (switches and routers) run their own distributed control logic, 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. This is very different from the existing decentralized control planes typically used in traditional networks (as well as in many Ad Hoc and P2P networks) and allows, e.g., to overcome the notoriously slow reaction to changes (e.g., link failures) and rerouting of flows in those networks: one of the key reasons behind Google's move to SDN [45].

Fig. 6 depicts an example of an SDN network, configured to implement the initial state of our update example (see Fig. 1). Beyond the main architectural components, the figure also illustrates a classic interaction between them. Indeed, the dashed lines indicate that the SDN controller instructs the programmable network devices, typically switches [43]), on how to process (e.g., forward) the traversing packets. An example command sent by the controller to switch s is reported next to the dashed line connecting the two: This command instructs s to use $v1$ as next-hop for any packet destined to d.

The SDN architecture is expected to make network updates more frequent and more critical than in traditional networks. On the one hand, controllers are often intended to support

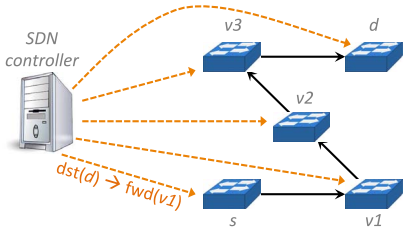


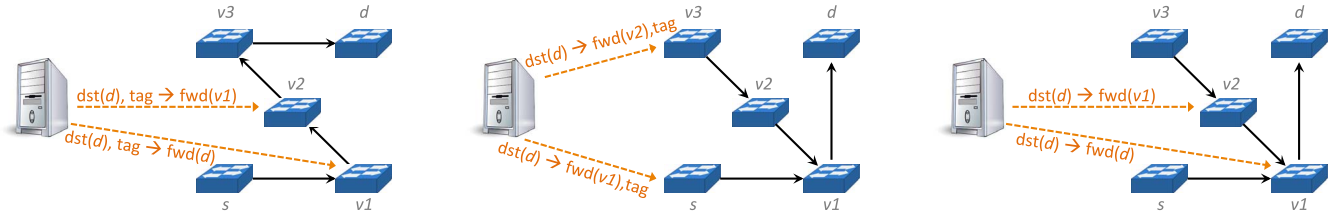
Fig. 6. Implementation of the surpassed state in Fig. 1 in an SDN network.

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 [46]. 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 forwarding rules installed on the switches; additionally, it should perform such updates while typically supporting more critical consistency guarantees (e.g., security-related ones) and performance objectives (e.g., for prompt reaction to failures) 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 few years. This research effort nicely complements approaches to specify [47], compile [48], [49], and check the implementation of [6] and [50] network requirements that operators may want to implement in their (SDN) networks.

The first cornerstone of SDN updates is represented by the work by Reitblatt *et al.* in 2011 [51] and 2012 [52]. 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 an 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 a given "old label" (e.g., no tag) and rules matching the old label are pre-installed on the switches. In a first step, the controller then instructs the internal switches to apply the final forwarding rules to packets carrying a "new label" – even if no packet carries such label 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



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

Fig. 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.

(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 [52]. One first line of research focuses on providing additional guarantees, e.g., congestion-freedom (from [53]–[60]). In a second line of research, several algorithms [32], [61]–[64] 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), avoiding the need for additional rules and hence enabling more resource-efficient approaches (e.g., TCAM memory slots are expensive and precious).

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

For a tutorial description of a few major works in the area, mainly [57], [65], we refer to the recent article of Li *et al*. [66].

III. Taxonomy of Update Techniques

We now present a general formulation of network update problem (Section III-A), which abstracts from assumptions and settings considered in the literature. 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 (Section III-B).

A. Generalized Network Update Problem (GNUP)

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.

1) Basic Problem: Generally speaking, a network update problem consists in computing a sequence of operations that changes the packet-processing rules installed on network devices. Consider any communication network: It is composed by a given set of inter-connected devices, that are able to process data packets (e.g., forwarding them to a next-hop) 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, $v1$, $v2$, $v3$, d); the final state forwards the same packets over (s, $v1$, d), and packets from $v3$ to d on ($v3$, $v2$, $v1$, 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, $v1$, $v2$, $v3$, d) to (s, $v1$, d) and ($v3$, $v2$, $v1$, d).

2) Operations: To perform a network update, a sequence of operations has to be computed. By operation, we mean a (direct or indirect) modification of packet-processing rules installed on one or more devices. As an example, an intuitive and largely-supported operation on network devices is rule replacement, which consists in instructing a device (e.g., $v3$) to replace an initial rule (e.g., forward the $s{-}d$ packet flow to $v2$) with the corresponding final one (e.g., forward the $s{-}d$ flow to d). Operations can be coarse-grained and indirect, as IGP link reweighting or configuration swapping in legacy networks that imply multiple rule replacements at distinct devices (see Section II).

3) Consistency: The difficulty in solving network update problems is that some form of consistency must be guaranteedly preserved *during* the update, for practical purposes (e.g., avoiding service disruptions and packet losses). Preserving consistency properties, in turn, depends on the order in which operations appear in the computed sequence and are executed by network devices. For example, if $v3$ replaces its initial rule with its final one before $v2$ in Fig. 1, then the operational sequence triggers a forwarding loop between $v2$ and $v3$ that interrupts the connectivity from s to d. In Section III-B, we provide an overview of consistency properties considered in the literature.

The practical need for guaranteeing consistency has two main consequences (as shown in Section II). First, it forces network updates to be performed incrementally, i.e., appropriately scheduling operations over time so that the installed intermediate states are 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 $v3$'s rule before $v2$'s one in the previous example).

4) 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 incrementally requires to install intermediate configurations, and in many cases it is practically desirable

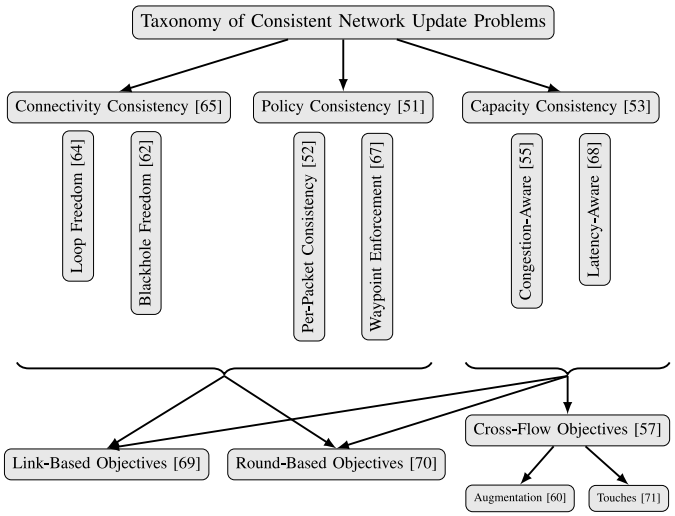


Fig. 8. Types of consistent network update problems, defined independently of the network setting (rule granularity and supported operations).

to minimize the time spent in such intermediate states. We provide a broader overview of performance goals considered by previous works in Section III-B.

5) Final Operational Sequences: Generally, the solution for an update problem can be represented as a sequence of *steps* or *rounds*, that both (i) guarantees consistency properties and (ii) 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 the target consistency properties are guaranteed independently of the relative order in which those operations are executed. Examples of operational sequences, computed by different techniques, are reported in Section II (see Figs. 3 and 5).

B. Classifying Update Techniques According to the Addressed GNUP Instance

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 Section III-A. Those variants basically differ in terms of the update problem on which they focus. Update problems in turn define both (1) the network setting, including admitted rule granularity and operations, (2) consistency properties, and (3) performance goals. An overview of the main update problems considered previously is depicted in Fig. 8, where we skipped the orthogonal network setting dimensions, rule granularity and supported operations, for clarity.

1) Rule Granularity: Network update techniques assume that the underlying devices support rules that implement one of the two alternative routing models: *destination-based* and *per-flow* routing.

a) 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.

b) 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 respectively packets with the same MPLS tag are forwarded along the same path.

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

a) Rule replacements: A first class of update techniques is based on computing an order in which initial rules are replaced by the corresponding final ones on the devices. Depending on the target setting (e.g., legacy networks or SDNs), such replacement can be admitted at different granularity, i.e., on a per-rule and per-device basis (as in OpenFlow networks) or on a per-group of rules and devices (link reweighting in IGP).

b) 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.

1) *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. Packets 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.

2) *Helper Rules:* Some techniques introduce additional rules, which do not belong neither to the old state nor to the new one, in some intermediate update step. These rules allow to divert the traffic temporarily to other parts of the network, and are called *helper rules*.

c) Mixed: Recently, some update techniques combine rule replacements and additions, in order to reduce the update overhead (especially in terms of device-memory consumption) while keeping the flexibility provided by adding rules.

3) Consistency Properties: Update techniques typically target to preserve one (or more) of the following consistency properties.

a) Connectivity consistency: The most basic form of consistency regards the capability of the network to keep delivering 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 on a limited set of routers, without reaching their destinations).

b) 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. For example, an already-mentioned policy consistency property (see Section II) is *per-packet consistency*, requiring that packets are always forwarded along either the pre-update or the post-update paths, but never a combination of the two.

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

Note: As mentioned in the introduction, most work on SDN network updates argues for the need to preserve the considered consistency properties at each and every moment during an update. In the following sections, we therefore assume a strong consistency model. We survey approaches relaxing this consistency model beginning in Section VII.

4) Performance Goals: We can distinguish between three broad classes of performance goals.

a) 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.

b) Round-based: A second class of consistent network update protocols aims to minimize the total makespan, by computing a schedule of rounds or steps, each consisting of switch rules that are safe to update simultaneously.

c) Cross-flow objectives: A third class of consistent network update protocols targets objectives arising in the presence of multiple flows.

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

2) *Touches:* Minimize the number of interactions with the switch, i.e., the sent messages.

Note: 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.

C. Summary and Insights

We formulated a generalized version of consistent network update problems studied by prior work. This generalization enables us to create a taxonomy of network update techniques, where previous contributions are classified along four dimensions: assumed rule granularity, operations allowed on the switches, consistency properties to be preserved and optimization goals. We structure this survey according to the dimension of the consistency properties, because of its importance in the definition of the addressed update problem: the following sections reflect this choice.

IV. Update Techniques to Guarantee 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 Section II). This is also motivated by the fact that blackholes cannot be created by reconfiguring current routing protocols, as proved in [46]. We then shift our focus on avoiding blackholes during arbitrary (e.g., SDN) updates.

A. Guaranteeing Loop-Freedom

Loop-freedom is a most basic consistency property and has hence been explored intensively already in the network update literature. So far in this work, we presented the notion of loop-freedom in the following framework: 1) routing based on the destination, and 2) avoiding all transient loops. Current research extends the loop-freedom model in both dimensions, introduced next, beginning with the routing model.

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, and 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.

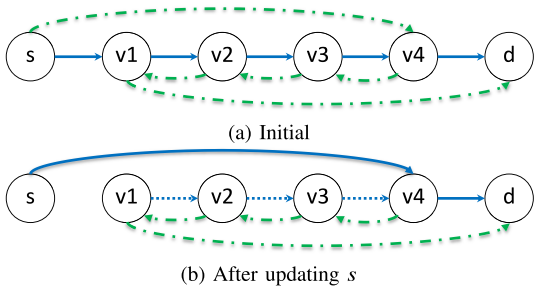
Moreover, one can distinguish between two different definitions for loop-free network updates: *Strong Loop-Freedom (SLF)* and *Relaxed Loop-Freedom (RLF)* [64]. SLF requires that at any point in time, the forwarding rules stored at the switches should be loop-free. RLF only requires that forwarding 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. RLF can significantly speed up the consistent migration process, as illustrated in Fig. 9.

2) Algorithms and Complexity: Two performance objectives are investigated in the literature, node-based und round-based, to be discussed in turn. Node-based objectives were studied first in the literature: the goal is to update as many nodes/links at a time as possible. Round-based objectives can be seen as more intuitive, aiming at minimizing the number of (controller-switch interaction) rounds. Node-based approaches are also used in round-based contexts, with the following intuition: by updating as many nodes as possible, the total makespan is hopefully minimized – coining the notion of *greedy approaches*. However, it has been shown that node- and round-based objectives can conflict and lead to vastly different schedules.

TABLE I
OVERVIEW OF RESULTS FOR LOOP-FREEDOM

Model	NP-hard	Polynomial time	Remarks
# Rounds, strong LF	Is there a 3-round loop-free update schedule? [64] *For 2-destination rules and sublinear x: Is there a x-round loop-free update schedule? [56]*	Is there a 2-round loop-free update schedule? [64]	In the worst case, $\Omega(n)$ rounds may be required. [64], *[62]*. $O(n)$-round schedules always exist *[65]*. Both applies to flow-based & *destination-based* rules.
# Rounds, relaxed LF	No results known.	$O(\log n)$-round update schedules always exist. [64]	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? [69], *[56]*	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. [69]	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. [69] Approximation algorithms from maximum acyclic subgraph [69] and minimum feedback arc set *[62]* apply.
# Links, relaxed LF	Is it possible to update x nodes in a loop-free manner? [69]	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. [69]	No approximation results known. [69]

Note: Results/references in *italics* are in the destination-based model.



(a) Initial

(b) After updating s

Fig. 9. Example to illustrate the differences between strong and relaxed loop-freedom. The current rules are drawn in solid blue, the new rules are drawn in dash-dotted green. In the left initial state, the task is to update all nodes to use the new forwarding rules. The right figure shows the network state after updating the node s. Note that the current rules for $v1$, $v2$, $v3$ are no longer on a path from s to t, and are hence drawn dotted. Thus, under RLF, $v1$, $v2$, $v3$ can be updated next, as possible loops are not connected to the source. Using SLF, $v3$ must be updated *after* $v2$, which in turn must be updated *after* $v1$. Hence, by updating $v4$ last, the RLF schedule length is just three, even when the construction is extended from 4 nodes to up to vx nodes. On the other hand, SLF requires at least x rounds. Hence, RLF can yield a speedup linear in the number of nodes.

a) Node-based objective ("greedy approach"): Mahajan and Wattenhofer [65] initiated the study of destination-based (strong) loop-free network updates. In particular, the authors show that by scheduling updates based on combinatorial dependencies, 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: based on the current state, the controller greedily attempts to update as many nodes as possible. E.g., in Fig. 9(a), their algorithm picks the nodes s and $v1$, as $v2$-4 have unresolved dependencies. The study of this model has been refined in [56] and [62], where the authors also establish hardness results, see Table I. A related variant using so-called proof labeling schemes [72] was proposed in [73].

Ludwig *et al.* [74], extended in [64], and [67] 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, as shown in Fig. 9. The authors also observe that loop-freedom can come in two flavors, strong and relaxed loop-freedom [64].

Despite the simple underlying graph, however, Amiri *et al.* [69] show that the node-based optimization problem is NP-hard, both in the strong and the relaxed loop-free model (SLF and RLF). In the example of Fig. 9(a) the maximization problem is easy though, clearly no more than two nodes (s and $v1$) can be updated initially. 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.* [69] also present polynomial-time optimal algorithms for specific scenarios, and both[1] [56], [69] provide further insights into approximability properties, see Table I.

b) Round-based objective: Foerster *et al.* [64], [74] 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

[1]We note that there exists a subtle difference between the approximation results by Foerster *et al.* and Ludwig *et al.*: the former authors usually aim to minimize the number of links which *cannot* be updated [56] (a feedback arc set problem), while Amiri *et al.* [69] consider the dual problem variant and aim to maximize the links which *can* be updated in the given round (the maximum acyclic subgraph problem). The approximation guarantees of the two problems differ: for the former model, the best known approximation bound is $O(\log n \log\log n)$ [75] while for the latter, constant approximation results exist [76].

is the network size, see Fig. 9. Furthermore, 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.

c) Other objectives: Dudycz *et al.* [61], detailed in [71], 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 total number of update messages the switches receive from the controller. The number of touches can be reduced if controllers *bundle* the updates of multiple flows to a given switch into a single message. However, consistency requirements impose limits on the extent to which updates can be bundled: e.g., in order to preserve loop-freedom, the update of a flow f_1 needs to take place at node $v1$ *before* $v2$, while the update of a flow f_2 needs to take place at node $v1$ *after* $v2$. The authors establish connections to the *Shortest Common Supersequence (SCS)* and *Supersequence Run* problems [77], and show NP-hardness already for two policies, each of which can be updated in two rounds, by a reduction from *Max-2SAT* [78].

Notwithstanding, Dudycz *et al.* [61] also present optimal polynomial-time algorithms to combine consistent update schedules computed for individual policies (e.g., using any existing algorithm, e.g., [64] and [65]), 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.

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; equivalently, the dual problem of minimizing the number of links which cannot be updated is a minimum feedback arc set problem. For showing NP-hardness, reductions from the *minimum hitting set problem* [69] and the *feedback arc set problem* [56] are used.

The problem of reconfiguring routes in a network can be seen as a special case of *combinatorial reconfiguration theory*: an abstract reconfiguration framework to transform a feasible solution of a problem (e.g., shortest path routing) into another solution of the same problem, e.g., while ensuring shortest path routing during the update [79]).

B. Guaranteeing 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 [62], [65]. Nonetheless, in the presence of memory limits and guaranteeing loop-freedom, finding the fastest blackhole-free update schedule is NP-hard [62].

C. Summary and Insights

Loop- and blackhole-freedom are both fundamental consistency properties, as their violation disconnects the logical routing graph, with loops additionally creating congestion. Both are easy to maintain, but hard to optimize regarding makespan or resource consumption. Of the two, loop-freedom is better understood, as already simple greedy approaches perform relatively well in simulations [62], where the node-based objective can also be approximated well [56], [69]. However, regarding the makespan, greedy approaches perform poorly in the adversarial scenarios [64]. Still, both round- and node-based objectives are NP-hard to optimize [56], [64], [69]. So far, to obtain a logarithmically competitive algorithm for the number of rounds, the consistency guarantees have to be slightly relaxed [64]. We summarize current hardness and algorithmic results in Table I, denoting results of the destination-based model in *italics*, whereas the remaining entries refer to route updates which are not necessarily destination-based.

D. Open Problems

Loop-free network updates still pose several open problems. Regarding the node-based objective, Amiri *et al.* [69] conjecture that update problems on bounded directed pathwidth 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 [64] indicate that if it is constant, the constant must be large. Regarding blackhole-freedom, the possible speedup while maintaining loop-freedom is inherently connected to the available memory, but a deeper algorithmic understanding is still missing [62].

V. Update Techniques to Guarantee Policy Consistency

Modern requirements go often beyond connectivity. For example, operators may want to ensure that packets traverse a given middlebox (e.g., a firewall) for security reasons,

TABLE II
OVERVIEW OF RESULTS FOR POLICY-PRESERVING UPDATES

Ref.	Approach	Guarantees	Computation	Remarks
[51], [52]	2-phase commit	PPC	Constant	Always applicable (if switches have free memory slots); requires packet tagging and additional rules on internal switches
[80]	incremental 2-phase commit	PPC	Exponential	Always applicable (if switches have at least one free memory slot); spreads switch-memory overhead over time
[81]	packet storing protocol	PPC	Constant	Dedicated protocol, based on storing packets at the controller
[82]	per-switch update protocol	PPC	Polynomial	Dedicated protocol, based on the translation of updates into logic circuits
[83]	anti-tampering update protocol	PPC	Polynomial	Dedicated protocol, based on updating switches according to paths; Robust to tampering and dropping attacks.
WayUp [67]	ordered rule replacements	WPE	Polynomial	Finishes in 4 rounds; Does not guarantee connectivity (e.g., for loops)
MIP of [67]	ordered rule replacements	WPE	Exponential	Optimizes update time; Guarantees absence of loops
MIP of [63]	ordered rule replacements	WPE chains	Exponential	Optimizes update time; Guarantees absence of loops
[84], [85, §2]	ordered rule replacements	arbitrary	Exponential	Update synthesis based on linear temporal logic and model checking
[86]	ordered rule replacements	PPC	Polynomial	Algorithm based on necessary conditions for PPC-preserving switch updates; Minimizes update rounds
GPIA [87]	ordered rule replacements	PPC	Polynomial	Greedy algorithm; Minimizes update rounds; Applicable to hybrid SDNs (becomes exponential)
GPIA+FED [87]	mixed	PPC	Polynomial	Applies restricted 2-phase commit after rule replacement sequence; Aims at reducing # of update rounds; Applicable to hybrid SDNs (becomes exponential)
[88]	mixed	arbitrary	Exponential	Optimizes the interleaving of rule replacements and additions; Aims at reducing # of update rounds

Note: Contributions are grouped by approach and year.

or a chain of middleboxes (e.g., encoder and decoder) for performance reasons; and/or they might like to enforce that paths comply with Service Level Agreements (e.g., in terms of delay). In this section, we discuss studied problems and proposed techniques aiming at preserving such requirements during network updates.

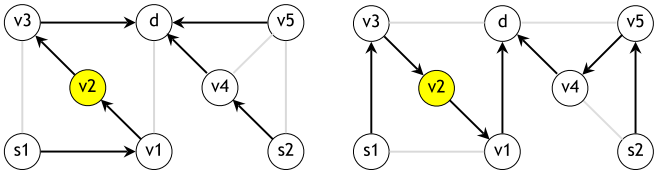
A. Definitions

Requirements on forwarding paths additional to connectivity can be modeled by routing *policies*, that is, links, nodes or sub-paths that have to be traversed by certain traffic flows at any moment in time.

Over the years, several contributions have targeted updates focusing on preserving specific policies. Historically, the first policy considered during SDN 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. PPC seems a natural choice to comply with high-level network requirements. Assume indeed that both the initial and the final paths accommodate requirements like security, performance, and SLA compliance. 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 PPC 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 the only major concern, and it may translate into simply enforcing that some flows traverse a firewall. Fig. 10 shows an example of this case, where the flow from $s1$ to d has to traverse $v2$. We refer to the property of forced traversal of a given node (waypoint) as *waypoint enforcement (WPE)*.

Policies more complex than WPE (but less constraining than PPC) may also be needed in general. For example, it may be desirable in large Internet Service Providers 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) Extended Surpassed state (b) Novel state

Fig. 10. A WPE-consistent update example where forwarding paths have to be changed from the Surpassed (Fig. 10(a)) to the New (Fig. 10(b)) state, while preserving traversal of the waypoint $v2$ (highlighted in the figure) at any time during the update.

B. Algorithms and Complexity

Table II overviews solving algorithm and complexity of policy-preserving update problems, further discussed in the following.

1) 2-Phase Commit Techniques: As described in Section III-B, 2-phase commit techniques carry out updates by setting an initial or final tag on packets at ingress devices (e.g., on $s1$ and $s2$ in Fig. 10), and maintaining two forwarding rules at internal nodes (e.g., $v1$, $v2$, $v3$, $v4$, and $v5$ in Fig. 10) so that each packet is forwarded over either its initial or final paths, according to the carried tag. This approach guarantees per-packet consistency by design.

A framework to implement 2-phase commit in traditional networks has been proposed by Alimi *et al.* [41] (see also Section II). It requires invasive modification of router internals, to manage tags and run arbitrary routing processes in separate process spaces. Such modifications are not needed in SDNs, where data-plane devices exhibit finer-grained programmability. Beyond presenting an implementation of 2-phase commit in OpenFlow, pioneering SDN update works [51], [52] argue

for the criticality of ensuring PPC in the SDN case, and of providing programmatic support for consistent updates within SDN controllers.

A downside of 2-phase commit techniques is that they require internal switches (e.g., $v1$, $v2$, $v3$, and $v5$ in Fig. 10) to maintain an additional rule every time their initial and final rules differ. This overhead requires all switches to have free memory slots (whose number depends on the update case), generally wastes memory resources, and might hamper other applications to properly work during updates (e.g., fast rerouting or security ones that might need to also install new flow rules in reaction to sudden traffic changes). To mitigate those problems, an incremental version of the original technique has been studied in [80]. This work proposes to divide the input update into sub-updates that can be carried out one after the other. Consider for example a variant of Fig. 10 where d is replaced by a set $\mathcal{D}$ of N replicas $d_1, \ldots, d_N$, and the update consists in changing the paths for every $d_i \in \mathcal{D}$ as shown in the figure. The incremental 2-phase commit technique enables to break down such an update into a sequence of sub-updates, where each sub-update modifies the paths of a distinct set of destinations $\mathcal{D}_j \subset \mathcal{D}$. This will limit the rules added to every switch at any time, at the price of a longer update completion time.

Ultimately, switch-memory consumption remains a fundamental limitation of 2-phase commit techniques, since tag-matching rules must be added to internal switches sooner or later during an update (or sub-updates). Both the original and the incremental techniques also exhibit other limitations, like the need for packet-header space, the tagging overhead, and complications with middleboxes modifying packet headers and tags [89], [90].

2) SDN-Based Update Protocols: McGeer [81], [82] presented two protocols to carry out network updates defined on top of OpenFlow. The first update protocol [81] is based on sending packets to the controller during updates, so that the controller can locally store packets until the final flow rules are installed on all the switches. As a result, switch resources (especially, TCAM entries) are saved, at the cost of adding delay on packet delivery, consuming network bandwidth, and requiring the controller to temporarily store packets. The second protocol [82] implements sequences of per-switch rule updates that guarantee PPC: It updates one switch at the time, ensuring that PPC is preserved at every step. In addition, Hua *et al.* [83] initiate a study on how to support PPC in an adversarial setting. They present FOUM, an update protocol where the update sequence is encoded in a packet signed by the controller, sent to one switch, and passed among switches at runtime. FOUM is shown to be robust to packet-tampering and packet dropping attacks. All those works assume dedicated protocols that are not supported by devices out of the box.

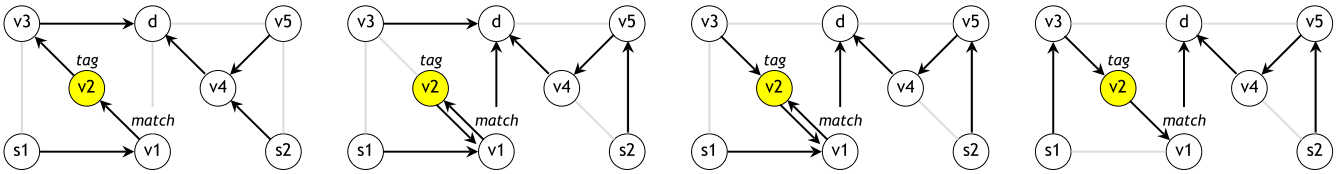
3) Rule Replacement Ordering: Further works explore which policies can be supported by only relying on carefully-computed sequences of rule replacements, so as to (i) introduce no memory overhead, and (ii) be readily supported by all (traditional and SDN) devices, without tag-related issues.

Initial contributions mainly focused on WPE consistency. Prominently, [67] studies how to compute short sequences of rule replacements, for quick updates ensuring that any given flow traverse a single waypoint. The authors propose WayUp, an algorithm that computes WPE-preserving updates spanning 4 rounds. However, they also show that it is not possible to guarantee both WPE and loop-freedom in some cases. Fig. 10 actually shows one case in which any sequence of rule replacements either causes a loop or a WPE consistency violation. Those infeasibility results have been extended to chains of waypoints in [63]. The latter work shows that flexibility in ordering and placing virtualized functions specified by a chain do not make the update problem always solvable. Those two works also prove that it is NP-hard to even decide if there exists a sequence of rule replacements preserving both loop-freedom and WPE (or waypoint chain traversal). Mixed Integer Program (MIP) formulations to find a safe sequence of rule replacements (when any exists) are proposed and evaluated in both cases.

The more general problem of preserving arbitrary policies defined by operators is tackled in [84]. This 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 algorithm based on counterexample-guided search and incremental model checking. Experimental results are provided about the scalability of the algorithm (up to networks with 1,000 nodes).

More recent works finally consider the problem of guaranteeing PPC by ordering rule replacements. Vissicchio *et al.* [87] show that this problem can be solved efficiently; they prove that a polynomial-time greedy algorithm called GPIA finds the sequence of per-switch rule replacements that does not violate PPC while updating the maximal number of switches and allowing the maximal parallelism between per-switch updates. The algorithm is based on iteratively simulating the update of every switch which has not been updated yet. For instance, in Fig. 10, GPIA would initially simulate the replacement of next-hops on every node. It would then define the first round of the update under computation by collecting all the switches ($v4$ and $v5$ in this example) whose update does not violate PPC. Then, it would iterate on the remaining nodes, discovering that $s2$ can be updated in the second round without violating PPC. GPIA terminates when there are no switches that can be safely updated. Cerný *et al.* [86] describe a refined version of this algorithm that avoids the simulation of switch updates thanks to the identification of necessary conditions for safely updating the switches.

Opportunities and limitations of rule replacement ordering for PPC-preserving updates have also been evaluated in [87], by simulating realistic update scenarios, on real network topologies. Results show that ordered rule replacements can rarely complete an update (for example, it could not in Fig. 10, but they can safely update many switches (typically, even more than the 3 out of 7 that it would update in Fig. 10)). Those results further motivate rule-replacement algorithms tailored to a more restricted family of policies (like WPE-preserving



(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4

Fig. 11. Sequence generated by the FLIP algorithm proposed in [88]. Note that the loop between $v1$ and $v2$ in Step 2 and 3 does not break connectivity, as it is traversed *only once* by packets. This is because $v1$ matches the tag set by $v2$, hence it forwards packets that have already traversed $v2$ directly to d.

ones) as well as mixed approaches (employing both rule replacements and duplication, as described below).

4) Mixed Approaches: Following up on their experimental results, Vissicchio *et al.* [87] argue for sequentially computing a safe rule replacement ordering, and applying a scope-limited 2-phase commit variant afterwards. In the example of Fig. 10, this approach would then lead to concatenating the ordered rule replacements on $v4$, $v5$ and $s2$ with the application of a 2-phase commit technique to the sub-network of non-updated switches (i.e., $s1$, $v1$, $v2$ and $v3$). This combination ensures the possibility to always perform the update (contrary to pure rule-replacement approaches) while reducing the update overhead (in switch memory and data-plane) with respect to the original 2-phase commit technique. The same work also generalizes this strategy to hybrid SDNs, potentially running any number both traditional and/or SDN control-planes – a setting in which a brute force (exponential) algorithm might be needed, depending on the nature of control-planes involved in the update.

FLIP, a different algorithm implementing a mixed approach, is described in [88] and detailed in [91]. FLIP jointly optimizes the interleaving of rule replacements and additions (for matching packet tags) so as to preserve arbitrary policies, including PPC. For instance, in Fig. 10, FLIP would compute a sequence of operations (illustrated in Fig. 11) where only $v1$ matches a tag set by $v2$: That is, FLIP needs only 1 additional rule when 2-phase commit would add 4, and sequentially combining replacements and additions would result in 3 additional rules. This implies that FLIP is strictly more powerful (i.e., solves a higher number of update cases) than only relying on rule replacements, exclusively using 2-phase commit, and sequential combining those two approaches. However, it is unclear how FLIP can be used in hybrid SDNs, with more than one control-plane. Also, its time complexity is not polynomial – even if the experiments suggest that FLIP quickly computes short update sequences in realistic networks.

C. Related Optimization Problems

Many policy-preserving algorithms face generalized versions of the optimization problems associated to connectivity-preserving updates (see Section IV): 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).

D. Summary and Insights

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. Initial work has been done on mixed approaches, relying on algorithms that can interleave different kinds of operations within the computed operational sequence.

E. Open Problems

Finding the best balance between the two extremes of relying on protocols on one hand, and on ordering algorithms on the other hand is an interesting direction. Despite some initial work has started towards this goal (see Section V-B), 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 (like those implementing P4 [92]) – for example, to achieve a better trade-off between switch memory consumption and update speed.

VI. Update Techniques to Guarantee Congestion-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.

TABLE III
COMPACT OVERVIEW OF FLOW MIGRATION ALGORITHMS

Ref.	Approach	(Un-)splittable model	Interm. paths	Computation	# Updates	Complete (decides if consistent migration exists)
[52]	Install old and new rules, then switch from old to new	Both, move each flow only once	No	Polynomial	1	No bandwidth guarantees
[53]	Partial moves according to free slack capacity s	Splittable	No	Polynomial	$\lceil 1/s \rceil - 1$	Requires slack on flow links
[57]	Greedy traversal of dependency graph	Both, move each flow only once	No	Polynomial	Linear	No (rate-limit flows to guarantee completion)
[59]	MIP of [57]	Both, move each flow only once	No	Exponential	Linear	Yes
[60]	Fix number of x intermediate states ahead of time, optimize via LP	Both	No	Polynomial	Any $x \in \mathbb{N}$	For a given number of intermediate states x, approximate minimum transient congestion (if > 0) by $\log n$ factor
			Yes	*Exponential*		
[60]	... via MIP	Both	Both	*Exponential*	Any $x \in \mathbb{N}$	For any given x yes, but not in general
[53]	Binary search of intermediate states via LP	Splittable	Yes	Polynomial in # of updates	Unbounded	Cannot decide if consistent migration is possible
[55]	Create slack with intermediate states, then use partial moves of [53]	Splittable	Yes	Polynomial	Unbounded	Yes
[56]	Split flows along old and new paths	2-Splittable	No	Polynomial	Unbounded	Yes
[54]	Use augmenting flows to find updates	Split., 1 dest., paths not fixed	Yes	Polynomial	Linear	Yes
[70]	Dynamic programming	Unsplittable	Both	*Exponential*	Exponential	Yes
Further practical extensions						
[58]	Extends approach of *SWAN* [53] in a data center setting					
[93]	Extends approach of *Dionysus* [57] with local dependency resolving					
[94]	Extends approach of *Dionysus* [57] with circuit nodes for optical wavelengths					
[95]	Extends approach of *Dionysus* [57] by using switch buffers to break deadlocks					
[96]	Extends approach of *Dionysus* [57] by allowing multiple target states					
[97]	Considers reconfiguration for dynamic flow arrivals					
[98]	Allows (un-)splittable flow migration (move once) with user-spec. deadlines & requirements via MIP (LP heuristic)					
[99], [100], [101]	Proposes multi-casting on portions of routes for faster updates, also to break deadlocks by using out-of-band capacities					
[102], [103], [104]	Does not require tagging of flows in the packet header, flows may take a mix of the old and new paths. For a constant number of flows on directed acyclic graphs (DAGs), a linear-time (fixed parameter tractable) algorithm is provided.					

A. 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., [54], $\mathcal{F}_{new}$ 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. It can then be useful to investigate, e.g., what type of rate-limiting is performed [105].

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 [53], 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 respect the link capacity.

B. Algorithms

Most current algorithms for capacitated updates of network flows use the seminal work by Reitblatt *et al.* [52] as an update mechanism. Analogously to *per-packet consistency* (see Section V), 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 flows in Fig. 12, if the green flow moves up before orange flow is on its new path, congestion occurs.

Mizrahi and Moses [106] prove that flow swapping is necessary for throughput optimization in the general case, as thus algorithms are needed that do not violate any capacity constraints during the network update, beyond simple flow swapping as well.

An overview of current algorithmic approaches can be found in Table III. In particular, we briefly describe the technique used, e.g., partial moves using slack capacity or dependency graphs. Furthermore, we categorize the algorithm techniques according to their model assumptions (splittability, helper rules), their complexity (computation and # updates), and if they can decide the underlying decision problem under their model assumptions, see also the later Table IV. Note that small changes in the model can lead to different complexities. For example, by not allowing intermediate paths as

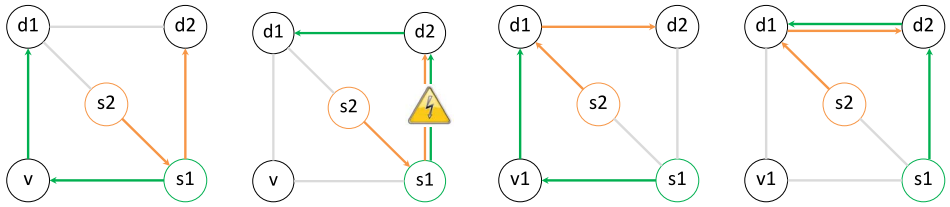


Fig. 12. In this flow migration example, all links have unit bidirectional capacity, and both orange and green flows have unit size as well. The task is to move both the green and orange flows from their initial paths in Fig. 12(a) to their final ones shown in Fig. 12(d). Updating both flows together could lead to the green flow being moved first, inducing congestion, see Fig. 12(b). However, this can be avoided by using succinct updates, first moving the orange flow as in Fig. 12(c), then the green flow.

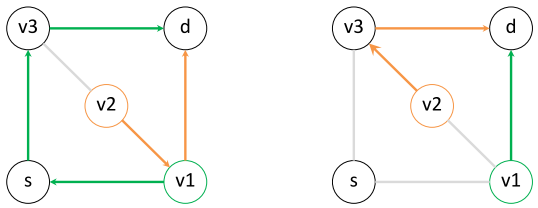


Fig. 13. In this network the task is to migrate consistently from the initial to the final state. If all flows and links have unit size, no consistent migration is possible: the destination has just incoming links of size two. If the flows just have a size of 2/3, one can migrate consistently in $\lceil 1/(1/3) \rceil - 1 = 2$ updates by moving half of the flow size of 1/3 each time in parallel.

in [57] and [59], flow migration is easier to handle, but is also less powerful. In the following, we focus on selected works that introduce general techniques or model ideas.

1) Slack: The seminal work by Hong *et al.* [53] on *SWAN* introduces the current standard model for capacitated updates. Their algorithmic contribution is two-fold, and also forms the basis for *zUpdate* [58]: 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, see Fig. 13: removing some background traffic allows consistent migration.

2) LP-Formulation: Second, Hong *et al.* [53] present 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. 13 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.

3) Spread Flows Over the Network: Brandt *et al.* [55] prove that splittable flow migration is always decidable in polynomial time, by providing an algorithm that attempts to create slack capacity on all links. The fundamental idea is to keep splitting flows along new paths, until slack is obtained such that the algorithm of Hong *et al.* [53] is applicable. For example in Fig. 12(a), one can proceed as follows: 1) split the orange flow equally along the old and new path, and afterwards 2) route a quarter of the green flow via either $s2$ or $d2$. The correctness of their approach relies on an augmenting flow techniques, we refer the reader to [55] for the intricate details.

4) Dynamicity & Dependency Graphs: Jin *et al.* [57] also consider the variable update times of switches in the network. To this end, inspired by [65], they build a dependency graph of the individual updates, greedily sending out updates once the respective pre-conditions are satisfied. For example, assume a flow f_1 can be migrated to its new path when least one of the flows f_2, f_3 was moved, but f_2, f_3 take different (unknown) time-spans to complete the move: the fastest method is to dynamically wait until either f_2, f_3 moves, any pre-computed schedule will have a longer makespan under adversarial conditions. When this greedy traversal of the dependency graph results in a deadlock, flows are rate-limited to guarantee progress. We note that it is not clear how to extend the dependency graph idea to intermediate paths.

5) Jointly Optimize Migration & New Paths: Brandt *et al.* [54] introduce the idea that the new flow paths should not be part of the problem input, but rather be computed jointly with the migration schedule. This idea aims at 1) speeding up the migration process and 2) allowing more problem instances to be solved. As an illustration, consider the problem in Fig. 13: from a pure admission perspective, no migration is needed, both flow demands are already satisfied in the initial state. Interestingly, for a single destination (but multiple commodities), consistent migration is *always* possible in this model, as long as there is some way to admit all flows without violating capacities. A framework for general multi-commodity flows is still missing.

Ghandi *et al.* [96] also observe that flows can have multiple migration options, as networks are often built with redundancy in mind. To this end, they first compute multiple choices for new flow paths, optimizing for close-to-optimal path properties and few stages in the resulting dependency graph. This allows them to dynamically speed up the execution of the consistent network updates, depending on the runtime conditions.

6) Node-Ordering Instead of 2-Phase Commit: Amiri *et al.* [102]–[104] propose to identify flows only by their source and destination, removing flow version numbers

TABLE IV
TABLE SUMMARIZING DECISION PROBLEM RESULTS FOR FLOW MIGRATION

Flow migration problem	Intermediate paths	Memory restrictions	Decision problem hardness
Unsplittable	Yes	Yes	NP-hard [55], EXPTIME [70]
		No	
	No	Yes	NP-hard [56], EXPTIME [70]
		No	
Unit size	Yes	Yes	NP-hard [55], EXPTIME [70]
		No	
	No	Yes	Open (also for integer size splitting)
		No	
Splittable	Yes	Yes	NP-hard [57]
		No	P [55]
	No	Yes	NP-hard [57]
		No	Open
Move every flow only once	Yes	Yes	Not allowed (model)
		No	
	No	Yes	NP-complete [57]
		No	NP-complete [56]
Node-ordering	Mix of old and new	Yes	NP-hard [102]
		No	

Note: 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, as providing sufficient memory is a special case of memory restrictions.

from the packet header ("tagging"). Their approach reduces complexity overhead, but does not permit the use of 2-phase commit techniques. Conceptually, each node has an old and a new forwarding rule for each flow, where the challenge is how to *order* these updates, without inducing congestion or forwarding loops. For an intuition, recall Fig. 5(d), and let the old rules be marked in solid blue, with the new rules being in dash-dotted green: over multiple rounds, the routing rules converge to the new state.

C. 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 IV. For unsplittable flows, an exponential time algorithm exists. 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* [53] for splittable flows, and was later extended to other models, see Table III.

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, see the decision problem hardness column of Table IV.

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 [55]. The authors use a combinatorial approach not relying on linear programming. Adding memory restrictions turns this problem NP-hard as well [57].

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 [56], [57]: 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 [60].

Hardly any (in-)approximability results exist today, and most work relies on reductions from the Partition problem, see Table V. The only result that we are aware of is via a reduction from MAX 3-SAT, which also applies to unit size flows [55].

D. 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, see [107], [108]. 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, see [109]. Nonetheless, splitting flows along multiple paths can lead to packet reordering problems, which need to be handled by further techniques, see [110].

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 [53]. If the solution takes too long to compute, the to-be solved problem might no longer exist, a problem only made worse when resorting to (NP-hard) integer programming for unsplittable flows. As such, some tradeoff has to be made

TABLE V
COMPACT OVERVIEW OF FLOW MIGRATION HARDNESS TECHNIQUES AND RESULTS

Ref.	Reduction via	(Un-)splittable model	Interm. paths	Memory limits	Decision prob. in general	Optimization problems/remarks
[57]	Partition	Splittable	No	Yes	NP-hard	NP-complete if every flow may only move once
[57]	Partition	Splittable	No	No	–	NP-hard (fewest rule modifications)
[55]	–	Splittable	Yes	No	P	Fastest schedule can be of unbounded length, LP for new reachable demands if cannot migrate
[56]	–	2-Splittable	No	No	P	studies slightly different model
[55]	(MAX) 3-SAT	Unsplittable	Yes	No	NP-hard (also for unit size)	NP-hard to approx. additive error of flow removal for consistency better than $7/8 + \varepsilon$
[60]	Partition	Unsplittable	Yes/No	No	–	NP-hard (fastest schedule)
[56]	Partition	Unsplittable	No	No	NP-hard	Stronger consistency model, but proof carries over
[98]	Partition & Subset sum	Unsplittable	No	No	–	NP-hard for 3-update schedule
[70]	Disjoint paths	Unsplittable	Yes	No	NP-hard (also for unit size)	NP-hard for already 2 unit size flows
[102], [103], [104]	3-SAT	Node-ordering	Mix of old & new	No	NP-hard (also for unit size)	NP-hard for 6 unit size flows if the pair of old and new path forms a DAG, on general graphs 2 flows suffice

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.* [111] and Perry *et al.* [112] for examples. Game-theoretic approaches have also been considered, e.g., [113]. Lastly, the application of model checking does not cover bandwidth restrictions yet [84], [114].

E. Summary and Insights

Congestion-aware consistency is stronger than loop-freedom [62], however it can be seen as orthogonal to policy consistency. Notwithstanding, most algorithms tag individual flows and perform 2-phase commits [52], thereby achieving some levels of policy consistency. Overall, four algorithmic techniques are currently studied: 1) If flows must remain on the paths defined by their old or new state, dependency graph approaches [57] are popular, which can also capture dynamic network conditions [96]. 2) If flows may be spread over the network, then insights from flow augmentation algorithms can be applied [54], [55]. 3) If non-polynomial runtime is acceptable, the problem of finding short congestion-aware schedules can be formulated as a MIP [60]. 4) Modifying the packet headers via tagging can be omitted in many cases by carefully tailoring the update schedules [102]–[104], however possibly at the cost of some policy consistency. In order to break deadlocks, the flows themselves may be rate-limited [57] respectively oversubscribed [60], or the background traffic is to be reduced [53]. A summary of all discussed algorithmic approaches can be found in Table III. Regarding the complexity classification of flow migration, one can think of it being analogous to classic multi-commodity flow problems: discrete constraints are NP-hard, whereas continuous constraints permit polynomial runtime [55]. However, while the problem remains in NP if every flow may only be touched once [57], the complexity is unknown otherwise [70]. All complexity results are summarized in Tables IV and V.

F. Open Problems

The classification of the complexity of flow migration still poses many questions, see Table V: 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? In other words, is the problem complete in NP, PSPACE, or EXPTIME? 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 [55] and [56] also apply to this case, but a formal proof is missing. Another open issue which researchers recently started to consider concerns how to exploit traffic engineering flexibilities and helper rules to jointly optimize update scheduling and route selection [115]. It is also yet unclear how to integrate such helper rules into dependency graphs, beyond manually defining intermediate states that differ from old and new as in [116] and [117].

Lastly, it would be interesting to compare the power and performance of the node-ordering approach introduced by Amiri *et al.* [102], further detailed in [103] and [104], to using the 2-phase commit of Reitblatt *et al.* [52]. Such a comparison could use involve the work of Zheng *et al.* [68], [118], under relaxed consistency guarantees, see the next section.

VII. FURTHER CONSIDERATIONS IN NETWORK UPDATES

We have so far assumed a "logically-centralized" perspective on the algorithmic network update problem, and mainly focused on strong notions of consistency. This is also the focus in most existing literature on the topic. However, there also exist interesting first work on solutions trying to relax these assumptions, by studying relaxed notions of consistency and aspects of distributed control planes. In the following, we summarize the most important work.

A. Relaxing Consistency Guarantees

So far we studied network updates assuming that consistency in the respective model must be maintained, e.g., no

forwarding loops must appear at any time. There are cases where consistency properties fundamentally cannot be guaranteed across an SDN network. For example, Panda *et al.* [11] noted that consistency (in terms of consistent application of some policies), availability and partition tolerance cannot be all guaranteed at the same time in an SDN network. In situations where the consistency property cannot be maintained at all or the computation of consistent updates is not tractable, some works proposed to break consistency in a controlled manner.

A first approach in this direction consists in trying to minimize the time spent in an inconsistent state, with underlying protocols being able to correct the induced problems (e.g., dropped packets are re-transmitted), as done in Google's *B*4 network [9], [119]. This can be understood as a very relaxed form of consistency, eventual consistency [13], [62], [120].

For less relaxed guarantees, i.e., beyond eventual consistency, Mizrahi and Moses propose to synchronize the clocks in the switches so that network updates can be performed simultaneously: With perfect clock synchronization, lossless communications and switch execution behavior, 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 [121] and [122], achieving microsecond accuracy in experiments. This obviously comes with additional message overhead for time synchronization in the whole network. An introduction and overview of so-called timed consistent updates is provided in [123]. Zheng *et al.* [68], [118] study the use of timed consistent updates in order to prevent congestion in the context of flow migration, in combination with latency considerations [124], [125, Sec. 6].

Nonetheless, in some situations synchronized updates can be considered optimal: e.g., consider the case in Fig. 13 where two unsplittable flows need to be swapped [106], with no alternative paths in the network available for the final links. Then, synchronizing the new flow paths can minimize the induced congestion [126]. Synchronized updates cannot guarantee packet consistency on their own, as packets that are currently en-route may still encounter changed forwarding rules at the next switch. Time can also be used similarly to a 2-phase commit though, by analogously using timestamps in the packet header as tags during the update [127], with [127] also showing an efficient implementation using timestamp-based TCAM ranges. Additional memory, as in the 2-phase commit approach of Reitblatt *et al.* [52], will be used for this method, but packets only need to be tagged implicitly by including the timestamp (where often 1 bit suffices [127], [128]). In [129] some additional methods are discussed on how to guarantee packet consistency by temporarily storing traffic at the switches.

Despite all those advantages, the proposed clock synchronization approaches do not prevent unpredictable variations of command execution time on network switches [57], motivating the need for prediction-based scheduling methods [130], [131]. Even worse, failures have an intrinsic, unavoidable cost in this approach. If a switch fails to update at all, the network can stay in an inconsistent state until the controller is notified and takes appropriate actions (e.g., computing another update). The same risk of inconsistencies holds if controller-to-switch messages are delayed or lost. In contrast, techniques based on sequential approaches can verify the application of sent update commands 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 safety violations.

B. Updates in Distributed Control Planes

Emerging large-scale SDNs will need to rely on scalable architectures and distributed control planes. Besides scalability, control planes need to be physically distributed to ensure availability and fault-tolerance, to improve load-balancing, and to reduce overheads. Distributed control planes can be organized differently, e.g., be either partitioned vertically (e.g., [132] and [133]) or horizontally (e.g., [134]–[136]), where switches are typically sharded among controllers (e.g., accounting for geographic location or latency), or where different controllers are in charge of different flow spaces. The design of a distributed control plane is a distributed systems problem and different designs come at different tradeoffs [137].

Which specific architecture is used also has implications on the network update problem. In general, to ensure consistency in network updates, additional coordination among different controllers may be required which comes with overhead: to guarantee consistency of network operation, actions performed on the data plane by different controllers may need to be synchronized. One option to this end are distributed data stores [138], so that applications would ideally remain unaware of any inconsistency [44]. Especially for wide-area networks, such additional synchronization can add substantial latency, where the notion of "continuous consistency" can be of use for parametrization in the application design of geo-replicated services [139]. On the other hand, strongly consistent network updates are unlikely possible if the control plane itself is only weakly consistent. *STN* [140] relies on a replicated state machine to update networks, which provides strong consistency guarantees, namely linearizability: thus, the distributed controller can emulate any existing network update algorithm designed for a single controller. However, it requires consensus. In contrast, *Ravana* [141]'s consistency is based on a weaker notion of "observational indistinguishability", and *Onos* [133] and *Net-Paxos,netpaxos* rely on partial event ordering.

Panda *et al.* [11] argue that linearisability is often unnecessary for ensuring correct application of most network policies as the investigated policies often have simple correctness conditions. Motivated by this observation, Sakic *et al.* [12] aim to overcome the blocking process in strongly consistent distributed control planes, and propose an adaptive, eventually consistent model. Levin *et al.* show in [143] that distributed network functions such as load-balancers can work around eventual consistency and still deliver performance sufficient for production deployments. Guo *et al.* [144] further expand the work of Levin *et al.* by reducing synchronization overhead.

TABLE VI
TABLE INDICATING THE CURRENT STATUS OF PRACTICAL CHALLENGES TO ENSURE CONSISTENT UPDATES

Challenge	Approach / Measurement	Status
(VIII-A) Ensuring basic communication	A: command line interface of devices	Available in (traditional) networks [147], [31]
	A: SDN controller programs and monitors	Deployed via, e.g., OpenFlow [43] and Network Information Bases [148]
(VIII-B) Applying operational sequences	M: coordinator messages lost respectively not applied by all devices	Measured on commodity hardware [57], [149]
	A: status-checking commands and protocols	Evaluated in testbed [147]
	A: lower-level packet cloning mechanisms	Evaluated in simulations [31], data set at https://inl.info.ucl.ac.be/softwares
	A: active probing packets	Evaluated in (small) testbed [150]
	A: acknowledgement-based protocols	Evaluated in small testbed [149]
(VIII-C) Working around device limitations	M: flow table size and setup rate limits statistics gathering	Measured on a common OpenFlow implementation on a switch [134]
	M: high rule installation latency	Measured on various types of (SDN) switches [151], [152], [153], [57]
	A: adapt schedules dynamically	Evaluated in testbed [57]
	A: eliminate redundant updates	Evaluated in Mininet [154], algorithms [61], SDN testbed [71]
(VIII-D) Multiple control-plane conflicts	A: pro-actively specifying computation of final rules	Implementation of [49] available at http://www.frenetic-lang.org
	A: implementing coordination and locking primitives on switches	Implementation of [155] available at https://github.com/lironsc/of-sync-lib
	A: reactively detecting and possibly resolving conflicts	Algorithms [140]
	A: meta-algorithms	General theory [46]
(VIII-E) Updating the control-plane	A: hypervisor maintains history	Evaluated in Mininet [156]
	A: explicit state transfer	Evaluated in Mininet [157]
(VIII-F) Events occurring during updates	M: impact of link failures (IGP)	Evaluated in simulations [32], data set at https://inl.info.ucl.ac.be/softwares
	A: enforcing per-packet consistency against packet-tampering adversary	Evaluation in small testbed [83]

Nguyen *et al.* [145] present ez-Segway, a decentralized mechanism to consistently and quickly update the network state while preventing forwarding anomalies (loops and black-holes) and avoiding link congestion. In their design, the centralized SDN controller only pre-computes information needed by the switches during the update execution. This information is distributed to the switches, which use partial knowledge and direct message passing to efficiently realize the update. This separation of concerns has the benefit of improving update performance as the communication and computation bottlenecks at the controller are removed.

Related to the question of how to perform updates in distributed control planes is the issue of how to perform such updates *in-band*: how to preserve connectivity between control and data plane if the updates performed by a remote controller also affects its own paths? Guaranteeing that each switch is managed, at any time, by at least one controller is challenging if control is in-band, and only recently, a first solution has been presented by Canini *et al.* [146], based on self-stabilization principles.

C. Summary and Insights

Even though SDN comes with the promise of centralized control, the network itself remains a distributed system—a fact which is responsible for many of the difficulties encountered in the previous sections. If one assumes perfect availability and/or partition tolerance, in clear contradiction to [11], then providing consistency becomes much easier: updates can be assumed to be executed at perfectly synchronized points in time, without any faults. Technical steps in this direction have already been performed by improving time synchronization protocols [121] and implementing timestamp-based TCAMs [127], allowing for new algorithmic directions [68], [118], [123]. Another hurdle is that at scale, the control is just *logically* centralized. The underlying distributed control plane can be implemented in various ways, each with its own set of benefits and tradeoffs [137]; an additional challenge is introduced if control is in-band [146].

VIII. 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. A brief overview indicating the current status is presented in Table VI.

A. Ensuring Basic Communication With Network Devices

Automated update systems classically rely on a logically-centralized coordinator, which must interact with network devices to 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, as noted in [31] and [147]. For SDNs, 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 [43] or similar protocols) and monitor (e.g., thanks to a Network Information Base [148]) the controlled devices.

B. 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 [57], [149]. Those possibilities are typically taken into account in the computation of the update sequence (see Section III). However, an effective update system must also ensure that operations are actually applied as in the computed sequences, e.g., that all operations in one update step are actually executed on the switches before sending

operations in the next 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 [147], lower-level packet cloning mechanisms [31], or active probing packets [150]) to acknowledgment-based protocols implemented by SDN devices [149].

C. 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 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 current 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 [134] 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.* [151] experimentally demonstrated the high rule installation latency of four different types of production SDN switches. This confirmed the results of independent studies [152], [153] 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 [57] 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 [57] significantly reduces multi-switch update latency by carefully scheduling operations according to dynamic switch conditions. In addition, CoVisor [154] and [61] minimize the number of rule updates sent to switches through eliminating redundant updates.

D. 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 case of SDNs with replicated controllers). For updates of traditional networks, the control-plane distribution is straightforwardly taken into account, since it is encompassed in the update problem definition (see Section II). 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, e.g., by pro-actively specifying how the final rules have to be computed (e.g., [49]), by implementing coordination and locking primitives on switches (e.g., [155]), or by reactively detecting and possibly resolving conflicts (e.g., [140]). A generalization of the above setting consists in considering multiple control-planes that may be either all distributed, all centralized, or hybrid (some distributed and some centralized). Potential conflicts and general meta-algorithms to ensure consistent updates in those cases are described in [46].

E. 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 an old controller with a newer one. In particular, HotSwap [156] describes an architecture that enables the replacement of an SDN controller, 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 [157].

F. Dealing With Events Occurring During an Update

Operational sequences computed by update algorithms forcedly assume stable network conditions. In practice, however, unpredictable events, like failures, can modify the network behavior concurrently and 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 [32]. A more systematic approach is taken by the recent FOUM work [83], that aims at guaranteeing per-packet consistency in the presence of an adversary able to perform packet-tampering and packet-dropping attacks.

IX. Future Research Directions

We have identified and already discussed (in Sections IV-D, V-E, and VI-F) several open research questions to preserve each of the consistency properties considered by prior work. We now describe more general areas which we believe deserve more attention by the research community in the future.

A. 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 capacity consistency. So far, only for a small number of specific models polynomial-time optimal algorithms are known. Even less is known about approximation algorithms. Hence, more research efforts would be needed to chart a clearer picture of the complexity landscape for network update

problems. We expect that some of these insights will also have interesting implications on classic optimization problems, such as combinatorial reconfiguration [103] problems.

B. Tailoring Update Mechanisms to Specific Networks

Datacenter topologies are usually highly connected and regular while wide-area networks are more sparse and organically grown: properties which may be exploitable towards more efficient and faster network update algorithms. Today, hardly anything is known about mechanisms for such more specific graph classes. A conceptually similar challenge arises in the context of reconfigurable links, e.g., "*how to gracefully transition between two topologies*" [158] or dynamic (e.g., wireless or cellular) networks. Additionally, the latter environments are appropriate for network coding [159], [160], where to the best of our knowledge current work is oblivious to ensuring consistency for codes during updates, respectively how to facilitate network coding itself for, e.g., faster updates. Tailoring existing algorithm and designing new algorithms for network topologies arising in practice hence constitutes a theoretically and practically very interesting area for future research.

C. Refining Our Models for Specific Technologies

While today's network models capture well the *fundamental* constraints and tradeoffs in consistent network update problems, these models are relatively simple. Today, SDN is used and discussed in various contexts and for different reasons, from network virtualization in datacenters to network slicing in emerging 5G applications [161]–[163], but also in the context of smart grids [164], [165], wireless (sensor) networks [166]–[168], and (hybrid) enterprise and ISP networks [48], [169], [170]. These use cases come with different *specific* requirements on the consistency (e.g., whether per-packet consistency is strictly needed) and performance (e.g., tolerable number of rounds), and also differ in terms of the available *knobs* which can be used for the network update (e.g., helper rules may undesirable in the context of network slicing). Accordingly, our models need to be refined and tailored towards specific use cases.

D. New Update Problems Raised by Stateful Applications

Network update schemes also depend on the application, e.g., traffic engineering applications can come with very different requirements than load-balancing [171]. Especially the development of more advanced and complex applications (e.g., [172], [173], [85, Sec. 3]) on top of SDN networks may create the need to support new consistency properties, and to potentially preserve these new properties during network updates. This is particularly true for stateful applications, whose state is built or modified by multiple flows. As an illustration, consider a stateful security appliance (e.g., a firewall) that checks if traffic is illegitimate on the basis of the exchange of information between flow sources and destinations over multiple interaction rounds – e.g., checking that TCP connections are never started by machines external to an enterprise network. In this case, network updates must consistently move all the flows needed for the security appliance to work correctly. For example, consistent updates should ensure that flows from any source s to any destination d cross the same security appliance traversed by flows from d to s; otherwise, the appliance can incorrectly interrupt connections during the update (and potentially block the corresponding traffic for some time). An interesting avenue for future research is represented by studying if and how constraints related to stateful applications change the complexity of update problems, as well as by developing algorithms and strategies to efficiently deal with such additional constraints.

E. Update Frequency

Another research direction is with respect to the frequency of network updates. Regarding inter-datacenter networks, *SWAN* [53] proposes to update the network every few minutes; a choice heavily influenced by the computation time needed. As such response times are not acceptable for, e.g., applications requiring interactive traffic, part of the network resources cannot be optimized on the fly. Similarly, in a smart/micro grid environment, security concerns demand policy consistency, but at the same time, update speed is critical for power grid applications after network failures. Hence, Jin *et al.* [174] employ the fast 2-phase commit protocol [52], which enforces policy consistency, but not, e.g., capacity consistency. Security policy concerns also prevail in enterprise networks, where dynamic updates are required to facilitate users' changing devices as part of BYOD (Bring Your Own Device) or workplace timetables [175]. It would be interesting to see how computation (and deploy) times can be significantly improved for more complex consistency properties. One option could be to employ machine learning, as already used to obtain routing configurations [176]. On the other hand, if the updates are not time-critical but just augment the system's behavior, e.g., by improving throughput, the tradeoff between update frequency and performance would be of further research interest. Fundamental groundwork on this perspective has already been presented by Destounis *et al.* [97], but, as the authors point out, "*the network updates problem is orthogonal and complementary*" [97] to this problem angle. Another dimension to explore is how to reduce update frequency by improving in-band mechanisms [134].

F. Dealing With Distributed Control Planes

We believe that researchers have only started to understand the design and implication of SDN control planes which are distributed not only vertically but also horizontally [137], [177], [178]. The research work developed so far mainly focuses on how to ensure that concurrent update operations performed by different distributed controllers reach a consistent state. Depending on the specific setting, many more problems have to be solved. For example, if each distributed controller controls a different subset of devices, it may only change subpaths traversed by some traffic: How to perform updates that are consistent network-wide, in this case? Introducing synchronization between controllers might be a building block towards the final solution, i.e., for the

distributed controllers to agree on which operation to perform when. Which kind of synchronization is needed in this case? Are there lightweight forms of synchronization that preserve some consistency properties with low impact on control overhead and update speed?

G. Supporting In-Band Updates

Challenges of guaranteeing consistency throughout network updates are further exacerbated if the controllers have in-band control over the data plane devices, since they are also affected by the effects of update operations. As an example of additional constraints to be taken into account, the distributed controllers may lose connectivity to some switches that still need to be updated, in the case of in-band connectivity between the controller and the devices. How does this constraint affect the type and amount of update scenarios that can be carried out by different strategies (e.g., ordering-based algorithms)? Are there practical workarounds to ensure that controllers always have connectivity with the controlled switches, irrespectively of the update ordering and uncontrollable events like possible packet losses?

X. Conclusion

The purpose of this survey was to provide researchers active in or interested in the field of network update problems (and in particular Software-Defined Networks) with an overview of the state-of-the-art. To this end, besides summarizing and tabularizing current results, we also presented a classification of consistent network updates in form of a taxonomy. As such, the multitude of different models, techniques, impossibility results, and practical challenges are put into context and also allow direct comparisons. Furthermore, we identified and listed many at present open technical and algorithmic problems, but also pointed out currently overlooked gaps in the framework of consistent network updates. Additionally, we presented a historical perspective, showcasing the possibilities in traditional networks, which can be of interest to operators investigating the migration to Software-Defined Networking. Subsequently, we discussed the fundamental new challenges introduced in Software-Defined Networks, also relating them to classic graph-theoretic optimization problems. These new challenges open a wide landscape of possibilities for future research, ranging from the integration of new technologies, adapting to particular network types, or the rise of complex (stateful) applications on top of current deployments, to list just a few.

Acknowledgment

For inputs and feedback, the authors would like to thank Marco Canini, Nate Foster, Dejan Kostić, Ratul Mahajan, Roger Wattenhofer, and Jiaqi Zheng, as well as the anonymous reviewers of this article.

References

[1] Barefoot Networks. (2016). *The World's Fastest and Most Programmable Networks (White Paper)*. [Online]. Available: https://barefootnetworks.com/white-paper/the-worlds-fastest-most-programmable-networks/

[2] M. Imbriaco. (Dec. 2012). *Network Problems Last Friday, GitHub*. [Online]. Available: https://github.com/blog/1346network problemslastfriday

[3] J. Jackson. (Sep. 2012). *Godaddy Blames Outage on Corrupted Router Tables, PCWorld*. [Online]. Available: http://www.pcworld.com/article/262142/godaddy_blames_outage_on_corrupted_router_tables.html

[4] R. Mohan. (Jun. 2011). *Storms in the Cloud: Lessons From the Amazon Cloud Outage, Security Week*. [Online]. Available: http://www.securityweek.com/storms-cloud-lessons-amazon-cloud-outage

[5] United. (Jun. 2011). *United Airlines Restoring Normal Flight Operations Following Friday Computer Outage*. [Online]. Available: http://newsroom.united.com/news-releases?item=124170

[6] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. USENIX NSDI*, 2012, p. 9.

[7] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," *Comput. Commun. Rev.*, vol. 42, no. 4, pp. 467–472, 2012.

[8] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN: An intellectual history of programmable networks," *Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, 2014.

[9] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined WAN," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, Oct. 2013.

[10] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[11] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker, "CAP for networks," in *Proc. ACM HotSDN*, 2013, pp. 91–96.

[12] E. Sakic, F. Sardis, J. W. Guck, and W. Kellerer, "Towards adaptive state consistency in distributed SDN control plane," in *Proc. IEEE ICC*, 2017, pp. 1–7.

[13] E. Brewer, "Cap twelve years later: How the 'rules' have changed," *Computer*, vol. 45, no. 2, pp. 23–29, Feb. 2012.

[14] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*, 2nd ed. Upper Saddle River, NJ, USA: Pearson Educ., 2007.

[15] J. Moy, P. Pillay-Esnault, and A. Lindem, "Graceful OSPF restart," Internet Eng. Task Force, Fremont, CA, USA, RFC 3623, 2003.

[16] A. Shaikh, R. Dube, and A. Varma, "Avoiding instability during graceful shutdown of multiple OSPF routers," *IEEE/ACM Trans. Netw.*, vol. 14, no. 3, pp. 532–542, Jun. 2006.

[17] M. Shand and L. Ginsberg, "Restart signaling for IS–IS," Internet Eng. Task Force, Fremont, CA, USA, RFC 5306, 2008.

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

[19] J. Fu, P. Sjodin, and G. Karlsson, "Loop-free updates of forwarding tables," *IEEE Trans. Netw. Service Manag.*, vol. 5, no. 1, pp. 22–35, Mar. 2008.

[20] L. Shi, J. Fu, and X. Fu, "Loop-free forwarding table updates with minimal link overflow," in *Proc. IEEE ICC*, 2009, pp. 1–6.

[21] M. Shand and S. Bryant, "A framework for loop-free convergence," Internet Eng. Task Force, Fremont, CA, USA, RFC 5715, Jan. 2010.

[22] R. Keralapura, C.-N. Chuah, and Y. Fan, "Optimal strategy for graceful network upgrade," in *Proc. ACM INM*, 2006, pp. 83–88.

[23] S. Raza, Y. Zhu, and C.-N. Chuah, "Graceful network operations," in *Proc. IEEE INFOCOM*, 2009, pp. 289–297.

[24] S. Raza, Y. Zhu, and C.-N. Chuah, "Graceful network state migrations," *IEEE/ACM Trans. Netw.*, vol. 19, no. 4, pp. 1097–1110, 2011.

[25] P. François, M. Shand, and O. Bonaventure, "Disruption free topology reconfiguration in OSPF networks," in *Proc. IEEE INFOCOM*, 2007, pp. 89–97.

[26] F. Clad, P. Mérindol, J.-J. Pansiot, P. François, and O. Bonaventure, "Graceful convergence in link-state IP networks: A lightweight algorithm ensuring minimal operational impact," *IEEE/ACM Trans. Netw.*, vol. 22, no. 1, pp. 300–312, Feb. 2014.

[27] F. Clad, P. Mérindol, S. Vissicchio, J.-J. Pansiot, and P. François, "Graceful router updates in link-state protocols," in *Proc. IEEE ICNP*, 2013, pp. 1–10.

[28] F. Clad, S. Vissicchio, P. Mérindol, P. François, and J.-J. Pansiot, "Computing minimal update sequences for graceful router-wide reconfigurations," *IEEE/ACM Trans. Netw.*, vol. 23, no. 5, pp. 1373–1386, Oct. 2015.

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

[30] I. Pepelnjak. (Oct. 2007). *Changing the Routing Protocol in Your Network, ipSpace*. [Online]. Available: http://blog.ipspace.net/2007/10/change-routing-protocol-in-your-network.html

[31] L. Vanbever, S. Vissicchio, C. Pelsser, P. François, and O. Bonaventure, "Seamless network-wide IGP migrations," in *Proc. ACM SIGCOMM*, 2011, pp. 314–325.

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

[33] 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, pp. 39–50.

[34] S. Vissicchio, L. Vanbever, L. Cittadini, G. G. Xie, and O. Bonaventure, "Safe routing reconfigurations with route redistribution," in *Proc. IEEE INFOCOM*, 2014, pp. 199–207.

[35] 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, pp. 2220–2228.

[36] P. François, O. Bonaventure, B. Decraene, and P.-A. Coste, "Avoiding disruptions during maintenance operations on BGP sessions," *IEEE Trans. Netw. Service Manag.*, vol. 4, no. 3, pp. 1–11, Dec. 2007.

[37] 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," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 231–242, Oct. 2008.

[38] E. Keller, J. Rexford, and J. E. van der Merwe, "Seamless BGP migration with router grafting," in *Proc. USENIX NSDI*, 2010, p. 16.

[39] S. Vissicchio *et al.*, "Improving network agility with seamless BGP reconfigurations," *IEEE/ACM Trans. Netw.*, vol. 21, no. 3, pp. 990–1002, Jun. 2013.

[40] N. Kushman, S. Kandula, D. Katabi, and B. M. Maggs, "R-BGP: Staying connected in a connected world," in *Proc. USENIX NSDI*, 2007, p. 25.

[41] R. Alimi, Y. Wang, and Y. R. Yang, "Shadow configuration as a network management primitive," in *Proc. ACM SIGCOMM*, 2008, pp. 111–122.

[42] 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, pp. 351–364.

[43] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[44] D. Kreutz *et al.*, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.

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

[46] 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, pp. 469–477.

[47] N. Foster *et al.*, "Languages for software-defined networks," *IEEE Commun. Mag.*, vol. 51, no. 2, pp. 128–134, Feb. 2013.

[48] M. Casado *et al.*, "Ethane: Taking control of the enterprise," in *Proc. ACM SIGCOMM*, 2007, pp. 1–12.

[49] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software-defined networks," in *Proc. USENIX NSDI*, 2013, pp. 1–14.

[50] P. Kazemian *et al.*, "Real time network policy checking using header space analysis," in *Proc. USENIX NSDI*, 2013, pp. 99–112.

[51] 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, p. 7.

[52] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. ACM SIGCOMM*, 2012, pp. 323–334.

[53] C.-Y. Hong *et al.*, "Achieving high utilization with software-driven WAN," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 15–26, Oct. 2013.

[54] S. Brandt, K.-T. Foerster, and R. Wattenhofer, "Augmenting flows for the consistent migration of multi-commodity single-destination flows in SDNs," *Pervasive Mobile Comput.*, vol. 36, pp. 134–150, Apr. 2017.

[55] S. Brandt, K.-T. Föerster, and R. Wattenhofer, "On consistent migration of flows in SDNs," in *Proc. IEEE INFOCOM*, 2016, pp. 1–9.

[56] K.-T. Foerster and R. Wattenhofer, "The power of two in consistent network updates: Hard loop freedom, easy flow migration," in *Proc. ICCCN*, 2016, pp. 1–9.

[57] X. Jin *et al.*, "Dynamic scheduling of network updates," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 539–550, Oct. 2014.

[58] H. H. Liu *et al.*, "zUpdate: Updating data center networks with zero loss," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 411–422, Oct. 2013.

[59] L. Luo, H. Yu, S. Luo, and M. Zhang, "Fast lossless traffic migration for SDN updates," in *Proc. IEEE ICC*, 2015, pp. 5803–5808.

[60] J. Zheng, H. Xu, G. Chen, and H. Dai, "Minimizing transient congestion during network update in data centers," in *Proc. IEEE ICNP*, 2015, pp. 1–10.

[61] S. Dudycz, A. Ludwig, and S. Schmid, "Can't touch this: Consistent network updates for multiple policies," in *Proc. IEEE/IFIP DSN*, 2016, pp. 133–143.

[62] K.-T. Föerster, R. Mahajan, and R. Wattenhofer, "Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes," in *Proc. IFIP Netw.*, 2016.

[63] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid, "Transiently secure network updates," in *Proc. ACM SIGMETRICS*, 2016, pp. 273–284.

[64] K.-T. Foerster, A. Ludwig, J. Marcinkowski, and S. Schmid, "Loop-free route updates for software-defined networks," *IEEE/ACM Trans. Netw.*, vol. 26, no. 1, pp. 328–341, Feb. 2018.

[65] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *Proc. ACM HotNets*, 2013, p. 20.

[66] D. Li, S. Wang, K. Zhu, and S. Xia, "A survey of network update in SDN," *Front. Comput. Sci.*, vol. 11, no. 1, pp. 4–12, 2017.

[67] 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, p. 15.

[68] J. Zheng *et al.*, "Scheduling congestion-free updates of multiple flows with chronicle in timed SDNs," in *Proc. IEEE ICDCS*, 2018, pp. 12–21.

[69] S. A. Amiri, A. Ludwig, J. Marcinkowski, and S. Schmid, "Transiently consistent SDN updates: Being greedy is hard," in *Proc. SIROCCO*, 2016, pp. 391–406.

[70] K.-T. Foerster, "On the consistent migration of unsplittable flows: Upper and lower complexity bounds," in *Proc. IEEE NCA*, 2017, pp. 1–4.

[71] A. Basta, A. Blenk, S. Dudycz, A. Ludwig, and S. Schmid, "Efficient loop-free rerouting of multiple SDN flows," *IEEE/ACM Trans. Netw.*, vol. 26, no. 2, pp. 948–961, Apr. 2018.

[72] A. Korman, S. Kutten, and D. Peleg, "Proof labeling schemes," *Distrib. Comput.*, vol. 22, no. 4, pp. 215–233, 2010.

[73] K.-T. Foerster, T. Luedi, J. Seidel, and R. Wattenhofer, "Local checkability, no strings attached: (A)cyclicity, reachability, loop free updates in SDNs," *Theor. Comput. Sci.*, vol. 709, pp. 48–63, Jan. 2018.

[74] A. Ludwig, J. Marcinkowski, and S. Schmid, "Scheduling loop-free network updates: It's good to relax!" in *Proc. ACM PODC*, 2015, pp. 13–22.

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

[76] R. Hassin and S. Rubinstein, "Approximations for the maximum acyclic subgraph problem," *Inf. Process. Lett.*, vol. 51, no. 3, pp. 133–140, 1994.

[77] M. Middendorf, "Supersequences, runs, and CD grammar systems," in *Developments in Theoretical Computer Science*, vol. 6. Amsterdam, The Netherlands: Gordon Breach Sci., 1994, pp. 101–114.

[78] M. Lewin, D. Livnat, and U. Zwick, "Improved rounding techniques for the MAX 2-sat and MAX DI-CUT problems," in *Proc. IPCO*, 2002, pp. 67–82.

[79] M. Kamiński, P. Medvedev, and M. Milanič, "Shortest paths between shortest paths," *Theor. Comput. Sci.*, vol. 412, no. 39, pp. 5205–5210, 2011.

[80] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *Proc. ACM HotSDN*, 2013, pp. 49–54.

[81] R. McGeer, "A safe, efficient update protocol for OpenFlow networks," in *Proc. ACM HotSDN*, 2012, pp. 61–66.

[82] R. McGeer, "A correct, zero-overhead protocol for network updates," in *Proc. ACM HotSDN*, 2013, pp. 161–162.

[83] 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, pp. 1–9.

[84] J. McClurg, H. Hojjat, P. Cerný, and N. Foster, "Efficient synthesis of network updates," in *Proc. ACM SIGPLAN PLDI*, 2015, pp. 196–207.

[85] J. McClurg, "Program synthesis for software-defined networking," Ph.D. dissertation, Dept. Comput. Sci., Univ. Colorado Boulder, Boulder, CO, USA, Jan. 2018.

[86] P. Cerný, N. Foster, N. Jagnik, and J. McClurg, "Optimal consistent network updates in polynomial time," in *Proc. DISC*, 2016, pp. 1–14.

[87] S. Vissicchio, L. Vanbever, L. Cittadini, G. G. Xie, and O. Bonaventure, "Safe update of hybrid SDN networks," *IEEE/ACM Trans. Netw.*, vol. 25, no. 3, pp. 1649–1662, Jun. 2017.

[88] S. Vissicchio and L. Cittadini, "FLIP the (flow) table: Fast lightweight policy-preserving SDN updates," in *Proc. IEEE INFOCOM*, 2016, pp. 1–9.

[89] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul, "FlowTags: Enforcing network-wide policies in the presence of dynamic middlebox actions," in *Proc. ACM HotSDN*, 2014, pp. 19–24.

[90] Z. A. Qazi *et al.*, "Simple-fying middlebox policy enforcement using SDN," in *Proc. ACM SIGCOMM*, 2013, pp. 27–38.

[91] S. Vissicchio and L. Cittadini, "Safe, efficient, and robust SDN updates by combining rule replacements and additions," *IEEE/ACM Trans. Netw.*, vol. 25, no. 5, pp. 3102–3115, Oct. 2017.

[92] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[93] 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, pp. 1–9.

[94] X. Jin *et al.*, "Optimizing bulk transfers with software-defined optical WAN," in *Proc. ACM SIGCOMM*, 2016, pp. 87–100.

[95] Y. Chen and J. Wu, "Max progressive network update," in *Proc. IEEE ICC*, 2017, pp. 1–6.

[96] R. Gandhi, O. Rottenstreich, and X. Jin, "Catalyst: Unlocking the power of choice to speed up network updates," in *Proc. ACM CoNEXT*, 2017, pp. 389–401.

[97] A. Destounis, S. Paris, L. Maggi, G. S. Paschos, and J. Leguay, "Minimum cost SDN routing with reconfiguration frequency constraints," *IEEE/ACM Trans. Netw.*, vol. 26, no. 4, pp. 1577–1590, Aug. 2018.

[98] S. Luo, H. Yu, L. Luo, and L. Li, "Arrange your network updates as you wish," in *Proc. IFIP Netw.*, 2016, pp. 10–18.

[99] S. Delaët, S. Dolev, D. Khankin, S. Tzur-David, and T. Godinger, "Seamless SDN route updates," in *Proc. IEEE NCA*, 2015, pp. 120–125.

[100] Y. Dinitz, S. Dolev, and D. Khankin, "Dependence graph and master switch for seamless dependent routes replacement in SDN (extended abstract)," in *Proc. IEEE NCA*, 2017, pp. 1–7.

[101] S. Delaët, S. Dolev, D. Khankin, and S. Tzur-David, "Make&activate-before-break for seamless SDN route updates," *Comput. Netw.*, vol. 147, pp. 81–97, Dec. 2018.

[102] S. A. Amiri, S. Dudycz, S. Schmid, and S. Wiederrecht, "Congestion-free rerouting of flows on DAGs," *CoRR*, vol. abs/1611.09296, p. 28, 2016.

[103] S. A. Amiri, S. Dudycz, S. Schmid, and S. Wiederrecht, "Congestion-free rerouting of flows on DAGs," in *Proc. ICALP*, 2018, pp. 1–13.

[104] S. A. Amiri, S. Dudycz, M. Parham, S. Schmid, and S. Wiederrecht, "Short schedules for fast flow rerouting," *CoRR*, vol. abs/1805.06315, p. 22, 2018.

[105] J. Zheng *et al.*, "Hermes: Utility-aware network update in software-defined WANs," in *Proc. IEEE ICNP*, 2018, pp. 1–10.

[106] T. Mizrahi and Y. Moses, "On the necessity of time-based updates in SDN," in *Proc. USENIX ONS*, 2014, p. 2.

[107] G. Baier, E. Köhler, and M. Skutella, "The k-splittable flow problem," *Algorithmica*, vol. 42, nos. 3–4, pp. 231–248, 2005.

[108] T. Hartman, A. Hassidim, H. Kaplan, D. Raz, and M. Segalov, "How to split a flow?" in *Proc. IEEE INFOCOM*, 2012, pp. 828–836.

[109] J. He and J. Rexford, "Toward Internet-wide multipath routing," *IEEE Netw.*, vol. 22, no. 2, pp. 16–21, Mar./Apr. 2008.

[110] S. Kandula, D. Katabi, S. Sinha, and A. Berger, "Dynamic load balancing without packet reordering," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 2, pp. 51–62, 2007.

[111] S. Kandula, I. Menache, R. Schwartz, and S. R. Babbula, "Calendaring for wide area networks," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 515–526, Aug. 2014.

[112] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized 'zero-queue' datacenter network," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 307–318, Aug. 2014.

[113] M. Hoefer, V. S. Mirrokni, H. Röglin, and S. Teng, "Competitive routing over time," *Theor. Comput. Sci.*, vol. 412, no. 39, pp. 5420–5432, 2011.

[114] 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*, 2015, pp. 73–85.

[115] H. Xu *et al.*, "Real-time update with joint optimization of route selection and update scheduling for SDNs," in *Proc. IEEE ICNP*, 2016, pp. 1–10.

[116] R. Singh, M. Ghobadi, K.-T. Foerster, M. Filer, and P. Gill, "Run, walk, crawl: Towards dynamic link capacities," in *Proc. ACM HotNets*, 2017, pp. 143–149.

[117] R. Singh, M. Ghobadi, K.-T. Foerster, M. Filer, and P. Gill, "RADWAN: Rate adaptive wide area network," in *Proc. ACM SIGCOMM*, 2018, pp. 547–560.

[118] J. Zheng *et al.*, "Scheduling congestion-and loop-free network update in timed SDNs," *IEEE J. Sel. Areas Commun.*, vol. 35, no. 11, pp. 2542–2552, Nov. 2017.

[119] C. Hong *et al.*, "B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN," in *Proc ACM SIGCOMM*, 2018, pp. 74–87.

[120] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond," *Commun. ACM*, vol. 56, no. 5, pp. 55–63, 2013.

[121] T. Mizrahi and Y. Moses, "ReversePTP: A clock synchronization scheme for software-defined networks," *Int. J. Netw. Manag.*, vol. 26, no. 5, pp. 355–372, 2016.

[122] T. Mizrahi and Y. Moses, "Using reversePTP to distribute time in software defined networks," in *Proc. ISPCS*, 2014, pp. 112–117.

[123] T. Mizrahi, E. Saat, and Y. Moses, "Timed consistent network updates in software-defined networks," *IEEE/ACM Trans. Netw.*, vol. 24, no. 6, pp. 3412–3425, Dec. 2016.

[124] K.-T. Foerster, "On the consistent migration of splittable flows: Latency-awareness and complexities," in *Proc. IEEE NCA*, 2018, p. 4.

[125] K.-T. Foerster, "Don't disturb my flows: Algorithms for consistent network updates in software defined networks," Ph.D. dissertation, Dept. Inf. Technol. Elect. Eng., ETH Zürich, Zürich, Switzerland, Sep. 2016.

[126] T. Mizrahi and Y. Moses, "Software defined networks: It's about time," in *Proc. IEEE INFOCOM*, 2016, pp. 1–9.

[127] T. Mizrahi, O. Rottenstreich, and Y. Moses, "TimeFlip: Using timestamp-based TCAM ranges to accurately schedule network updates," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 849–863, Apr. 2017.

[128] T. Mizrahi and Y. Moses, "The case for data plane timestamping in SDN," in *Proc. IEEE INFOCOM SWFAN*, 2016, pp. 856–861.

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

[130] T. Mizrahi and Y. Moses, "OneClock to rule them all: Using time in networked applications," in *Proc. IEEE/IFIP NOMS*, 2016, p. 9.

[131] T. Mizrahi and Y. Moses, "Time capability in NETCONF," IETF, Fremont, CA, USA, RFC 7758, 2016.

[132] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. R. Kompella, "ElastiCon: An elastic distributed SDN controller," in *Proc. ACM/IEEE ANCS*, 2014, pp. 17–27.

[133] P. Berde *et al.*, "ONOS: Towards an open, distributed SDN OS," in *Proc. ACM HotSDN*, 2014, pp. 1–6.

[134] A. R. Curtis *et al.*, "DevoFlow: Scaling flow management for high-performance networks," in *Proc. ACM SIGCOMM*, 2011, pp. 254–265.

[135] S. H. Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proc. ACM HotSDN*, 2012, pp. 19–24.

[136] K. Phemius, M. Bouet, and J. Leguay, "DISCO: Distributed multi-domain SDN controllers," in *Proc. IEEE NOMS*, 2014, pp. 1–4.

[137] S. Schmid and J. Suomela, "Exploiting locality in distributed SDN control," in *Proc. ACM HotSDN*, 2013, pp. 121–126.

[138] F. A. Botelho, F. M. V. Ramos, D. Kreutz, and A. N. Bessani, "On the feasibility of a consistent and fault-tolerant data store for SDNs," in *Proc. IEEE EWSDN*, 2013, pp. 38–43.

[139] H. Yu and A. Vahdat, "Design and evaluation of a conit-based continuous consistency model for replicated services," *ACM Trans. Comput. Syst.*, vol. 20, no. 3, pp. 239–282, 2002.

[140] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "A distributed and robust SDN control plane for transactional network updates," in *Proc. IEEE INFOCOM*, 2015, pp. 190–198.

[141] N. P. Katta, H. Zhang, M. J. Freedman, and J. Rexford, "Ravana: Controller fault-tolerance in software-defined networking," in *Proc. ACM SOSR*, 2015, Art. no. 4.

[142] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "NetPaxos: Consensus at network speed," in *Proc. ACM SOSR*, 2015, Art. no. 5.

[143] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically centralized? State distribution trade-offs in software defined networks," in *Proc. ACM HotSDN*, 2012, pp. 1–6.

[144] Z. Guo *et al.*, "Improving the performance of load balancing in software-defined networks through load variance-based synchronization," *Comput. Netw.*, vol. 68, pp. 95–109, Aug. 2014.
[145] T. D. Nguyen, M. Chiesa, and M. Canini, "Towards decentralized fast consistent updates," in *Proc. ACM ANRW*, 2016, pp. 19–25.
[146] M. Canini, I. Salem, L. Schiff, E. M. Schiller, and S. Schmid, "Renaissance: A self-stabilizing distributed SDN control plane," in *Proc. IEEE ICDCS*, 2018, pp. 1–11.
[147] X. Chen, Z. M. Mao, and J. E. van der Merwe, "PACMAN: A platform for automated and controlled network operations and configuration management," in *Proc. ACM CoNEXT*, 2009, pp. 277–288.
[148] T. Koponen *et al.*, "Onix: A distributed control platform for large-scale production networks," in *Proc. USENIX OSDI*, 2010, pp. 351–364.
[149] M. Kuzniar, P. Peresíni, and D. Kostic, "Providing reliable FIB update acknowledgments in SDN," in *Proc. ACM CoNEXT*, 2014, pp. 415–422.
[150] P. Peresíni, M. Kuzniar, and D. Kostić, "Rule-level data plane monitoring with monocle," in *Proc. ACM SIGCOMM*, 2015, pp. 595–596.
[151] K. He *et al.*, "Measuring control plane latency in SDN-enabled switches," in *Proc. ACM SOSR*, 2015, Art. no. 25.
[152] D. Y. Huang, K. Yocum, and A. C. Snoeren, "High-fidelity switch models for software-defined network emulation," in *Proc. ACM HotSDN*, 2013, pp. 43–48.
[153] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An open framework for OpenFlow switch evaluation," in *Proc. PAM*, 2012, pp. 85–95.
[154] X. Jin, J. Gossels, J. Rexford, and D. Walker, "Covisor: A compositional hypervisor for software-defined networks," in *Proc. USENIX NSDI*, 2015, pp. 87–101.
[155] L. Schiff, S. Schmid, and P. Kuznetsov, "In-band synchronization for distributed SDN control planes," *Comput. Commun. Rev.*, vol. 46, no. 1, pp. 37–43, 2016.
[156] L. Vanbever, J. Reich, T. Benson, N. Foster, and J. Rexford, "HotSwap: Correct and efficient controller upgrades for software-defined networks," in *Proc. ACM HotSDN*, Hong Kong, 2013, pp. 133–138.
[157] K. Saur *et al.*, "Safe and flexible controller upgrades for SDNs," in *Proc. ACM SOSR*, 2016, p. 8.
[158] K.-T. Foerster, M. Ghobadi, and S. Schmid, "Characterizing the algorithmic complexity of reconfigurable data center architectures," in *Proc. ACM/IEEE ANCS*, 2018, pp. 89–96.
[159] D. Szabó, F. Németh, B. Sonkoly, A. Gulyás, and F. H. P. Fitzek, "Towards the 5G revolution: A software defined network architecture exploiting network coding as a service," *Comput. Commun. Rev.*, vol. 45, no. 5, pp. 105–106, 2015.
[160] D. Szabo, A. Gulyas, F. H. P. Fitzek, and D. E. Lucani, "Towards the tactile Internet: Decreasing communication latency with network coding and software defined networking," in *Proc. Eur. Wireless*, 2015, pp. 1–6.
[161] X. Zhou, R. Li, T. Chen, and H. Zhang, "Network slicing as a service: Enabling enterprises' own software-defined cellular networks," *IEEE Commun. Mag.*, vol. 54, no. 7, pp. 146–153, Jul. 2016.
[162] I. da Silva *et al.*, "Impact of network slicing on 5G radio access networks," in *Proc. IEEE EuCNC*, 2016, pp. 153–157.
[163] J. Ordonez-Lucena *et al.*, "Network slicing for 5G with SDN/NFV: Concepts, architectures, and challenges," *IEEE Commun. Mag.*, vol. 55, no. 5, pp. 80–87, May 2017.
[164] N. Dorsch, F. Kurtz, H. Georg, C. Hägerling, and C. Wietfeld, "Software-defined networking for smart grid communications: Applications, challenges and advantages," in *Proc. IEEE SmartGridComm*, 2014, pp. 422–427.
[165] J. Zhang, B. Seet, T. T. Lie, and C. H. Foh, "Opportunities for software-defined networking in smart grid," in *Proc. IEEE ICICS*, 2013, pp. 1–5.
[166] T. Luo, H.-P. Tan, and T. Q. S. Quek, "Sensor OpenFlow: Enabling software-defined wireless sensor networks," *IEEE Commun. Lett.*, vol. 16, no. 11, pp. 1896–1899, Nov. 2012.
[167] B. T. de Oliveira, C. B. Margi, and L. B. Gabriel, "TinySDN: Enabling multiple controllers for software-defined wireless sensor networks," *IEEE Latin America Trans.*, vol. 13, no. 11, pp. 3690–3696, Nov. 2015.
[168] C. J. Bernardos *et al.*, "An architecture for software defined wireless networking," *IEEE Wireless Commun.*, vol. 21, no. 3, pp. 52–61, Jun. 2014.
[169] D. Levin, M. Canini, S. Schmid, F. Schaffert, and A. Feldmann, "Panopticon: Reaping the benefits of incremental SDN deployment in enterprise networks," in *Proc. USENIX ATC*, 2014, pp. 333–345.
[170] D. K. Hong, Y. Ma, S. Banerjee, and Z. M. Mao, "Incremental deployment of SDN in hybrid enterprise and ISP networks," in *Proc. ACM SOSR*, Santa Clara, CA, USA, 2016, p. 1.
[171] M. Aslan and A. Matrawy, "On the impact of network state collection on the performance of SDN applications," *IEEE Commun. Lett.*, vol. 20, no. 1, pp. 5–8, Jan. 2016.
[172] S. Ghorbani and B. Godfrey, "Towards correct network virtualization," in *Proc. ACM HotSDN*, 2014, pp. 109–114.
[173] J. McClurg, H. Hojjat, N. Foster, and P. Černý, "Event-driven network programming," in *Proc. ACM SIGPLAN PLDI*, 2016, pp. 369–385.
[174] D. Jin *et al.*, "Toward a cyber resilient and secure microgrid using software-defined networking," *IEEE Trans. Smart Grid*, vol. 8, no. 5, pp. 2494–2504, Sep. 2017.
[175] "Understanding enterprise SDN," Bothell, WA, USA, Allied Telesis, White Paper, 2015. [Online]. Available: https://www.alliedtelesis.com/sites/default/files/wp_understanding_enterprise_sdn_reva.pdf
[176] A. Valadarsky, M. Schapira, D. Shahaf, and A. Tamar, "Learning to route," in *Proc. ACM HotNets*, 2017, pp. 185–191.
[177] K. Phemius, M. Bouet, and J. Leguay, "DISCO: Distributed SDN controllers in a multi-domain environment," in *Proc. IEEE NOMS*, 2014, pp. 1–2.
[178] T. D. Nguyen, M. Chiesa, and M. Canini, "Decentralized consistent updates in SDN," in *Proc. ACM SOSR*, 2017, pp. 21–33.

Klaus-Tycho Foerster received the Diploma degrees in mathematics and computer science from the Braunschweig University of Technology, Germany, in 2007 and 2011, respectively, and the Ph.D. degree from ETH Zurich, Switzerland, in 2016, advised by R. Wattenhofer. He has been a Post-Doctoral Researcher with the Faculty of Computer Science, University of Vienna, Austria, since 2018. In 2016, he was a Visiting Researcher with Microsoft Research Redmond with R. Mahajan and joined Aalborg University, Denmark, as a Post-Doctoral Researcher with S. Schmid in 2017. His research interests revolve around algorithms and complexity in the areas of networking and distributed computing.

Stefan Schmid received the M.Sc. and Ph.D. degrees from ETH Zurich, Switzerland, in 2004 and 2008, respectively. He is a Professor with the Faculty of Computer Science, University of Vienna, Austria. In 2009, he was a Post-Doctoral Fellow with TU Munich and the University of Paderborn, from 2009 to 2015, a Senior Research Scientist with Telekom Innovations Laboratories, Berlin, Germany, and an Associate Professor with Aalborg University, Denmark, from 2015 to 2018. His research interests revolve around fundamental and algorithmic problems arising in networked and distributed systems.

Stefano Vissicchio received the master's degree and the Ph.D. degree in computer science from Roma Tre University in 2008 and 2012, respectively. He has been Post-Doctoral Researcher with the Université catholique of Louvain. He is a Lecturer with University College London. His research interests span network management, routing theory, algorithms and protocols, measurements, and network architectures. He was a recipient of the awards, including the ACM SIGCOMM 2015 Best Paper Award, the ICNP 2013 Best Paper Award, and two IETF/IRTF Applied Networking Research Prizes.