P4Update: Fast and Locally Verifiable Consistent Network Updates in the P4 Data Plane

Zikai Zhou
Chair of Communication Networks
Technical University of Munich
zikai.zhou@tum.de

Mu He
Chair of Communication Networks
Technical University of Munich
mu.he@tum.de

Wolfgang Kellerer
Chair of Communication Networks
Technical University of Munich
wolfgang.kellerer@tum.de

Andreas Blenk
Chair of Communication Networks
Technical University of Munich
University of Vienna
andreas.blenk@tum.de

Klaus-Tycho Foerster
Technical University of Dortmund
University of Vienna
klaus-tycho.foerster@tu-dortmund.de

ABSTRACT

Programmable networks come with the promise of logically centralized control, in order to optimize the network's routing behavior. However, until now, controllers are heavily involved in network operations to prevent inconsistencies such as blackholes, loops, and congestion. In this paper, we propose the P4Update framework, based on the network programming language P4, to shift the consistency control and most of the routing update logic out of the overloaded and slow control plane. As such P4Update avoids high and unnecessary control plane delays by mainly scheduling and offloading the update process to the data plane.

P4Update returns to operating networks in a partially centralized and distributed manner — taking the best of both centralized and distributed worlds. The main idea is to flip the problem setting and see asynchrony as an opportunity: switches inform their local neighborhood on resolved update dependencies. What's more, our mechanisms are also provably resilient against inconsistent, reordered, or conflicting concurrent updates. Unlike prior systems, P4Update enables switches to locally verify and reject inconsistent updates, and is also the first system to resolve inter-flow update dependencies purely in the data plane, significantly reducing control plane preparation time and improving its scalability. Beyond verification, we implement P4Update in a P4 software-switch-based environment. Measurements show that P4Update outperforms existing systems with respect to update speed by 28.6% to 39.1% in average.

CCS CONCEPTS

• **Networks → Network architectures**; • **Theory of computation → Design and analysis of algorithms**.

KEYWORDS

P4, consistent network updates, loop freedom, verification

ACM Reference Format:
Zikai Zhou, Mu He, Wolfgang Kellerer, Andreas Blenk, and Klaus-Tycho Foerster. 2021. P4Update: Fast and Locally Verifiable Consistent Network Updates in the P4 Data Plane. In *The 17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '21), December 7–10, 2021, Virtual Event, Germany.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3485983.3494845

1 INTRODUCTION

Many networks exist in a constant state of change, *e.g.*, ISP and hyperscale clouds, where operators must continuously adapt and optimize the network's forwarding behavior to gain high network efficiency [33, 65]. Herein programmable networks significantly simplified network management by implementing logically centralized control planes. However, beyond scalability issues, a centralized control plane is also orders of magnitude slower than the data plane [52], and hence dynamic reactions can face comparatively long delays.

These dynamic reactions especially come into play when the consistency of the data plane has to be ensured. Simple pushing of network route updates to the data plane can critically impact network operation: the network's inherent asynchrony could introduce blackholes, forwarding loops, and congestion, preventing packet delivery and inducing large losses. These performance degradations are highly problematic in the face of today's delay-sensitive applications such as, *e.g.*, machine-learning, distributed computations, or remote procedure calls. As a consequence, various sophisticated methods are needed to tackle such challenges [21].

However, designing fast and provably consistent network route updates is non-trivial due to the inherent dependencies between data plane updates. Early work [57] favored a centralized approach, where subsets of to-be-deployed safe updates are pre-computed and rolled out iteratively, each after the whole switch subset acknowledge their implementation. Various extensions were proposed, *e.g.*, [23, 24, 42, 53, 55], however these mechanisms heavily invoke the control plane, as thus facing long delays and imposing ongoing load on the controller: even the interaction between neighbors needs to take a control plane detour.

CoNEXT '21, December 7–10, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-9098-9/21/12…$15.00
https://doi.org/10.1145/3485983.3494845

To resolve this downside, Nguyen *et al.* [63] proposed a hybrid approach: a centralized controller offloads the flow routing update logic back to the network's nodes: the centralized controller pre-computes an update order, but nodes notify their upstream neighbors of resolved dependencies via the data plane — detours over the centralized controller are avoided. To this end, they augment the switches with local controllers, improving the update times of centralized works.

Unfortunately, existing works [42, 57, 63] lack local verification aspects [19, 20]: In case the controller has an inconsistent view of the network [69, 71], the deployed updates can induce, *e.g.*, forwarding loops, which need to be detected and alleviated again by external mechanisms, otherwise leading to buffer overflow and packet loss. Additionally, current systems assume that prior updates of a flow need to be finished first, before the next updates can be deployed: stragglers, *e.g.*, switches that are currently overloaded or simply slow by design, can significantly delay the update progress [42]. How to quickly skip to the next update is still an open question. Network operators are hence stuck between a rock and a hard place, as they must decide either

(1) to risk inconsistencies, which potentially take long to resolve, but initiate updates quickly, or
(2) to maintain consistency, but delay updates and hence pay in terms of network performance.

P4Update. Our proposal, P4Update, resolves the above challenges and brings further performance benefits.

First, P4Update is implemented in the P4 programming language, making it easy to deploy on off-the-shelf hardware.

Second, we employ a local verification mechanism that allows switches to reject inconsistent updates, enabling rapid updates in succession: even if the controller has an inconsistent view or updates are reordered, the forwarding state is free from blackholes, loops, and congestion.

Third, we further improve the theoretical and practical performance of distributed consistent updates: to reduce the worst-case complexity, we introduce a parallelized update and verification approach, which accelerates the update time in complex update scenarios. In more simple scenarios, P4Update utilizes a serial update approach, which minimizes computation and message complexity, and thereby outperforms state-of-the-art systems [42, 63] as well. Accordingly, P4Update is a first step towards context-aware (both traffic and topology-aware) network update approaches.

Fourth, P4Update resolves inter-flow dependencies leveraging stateful processing capability in the P4 data plane, in a purely local and adaptive way. Whereas prior work attempts to resolve such dependencies in the control plane, possibly being outdated by the time they are being implemented and imposing heavy load and delays on the controller, P4Update computes and dynamically updates dependencies in the data plane, according to the current situation.

Main contributions. Our contributions are as follows:

- We propose a new network update architecture for programmable networks, which achieves consistency for route updates by means of local verification in the data plane.
- We show that P4 switches can locally verify the correctness of consistent route updates and prove its blackhole, loop, and congestion freedom globally in the network.
- Beyond a sequential route update approach, we also develop and implement parallel consistent update mechanisms, which speed up the verified update process by segmentation.
- Regarding congestion due to link capacity violations after flow updates, we also develop the first completely local and dynamic inter-flow dependency resolution in the data plane.
- We provide an open-source P4-based implementation of our proposal, P4Update, and evaluate its in-network update and control plane preparation time on real network topologies in a P4 software-switch-based environment against state-of-the-art centralized and decentralized approaches.

2 BACKGROUND

We first provide background on P4 (§2.1) and on the interplay between consistent updates and local verification (§2.2).

2.1 P4 Background

P4 is a domain-specific language to describe the packet processing pipeline of Programmable Data Planes (PDP) [7]. Each pipeline is composed of a *parser*, multiple *match-action units*, and a *deparser*. The parser can be customized to extract a set of fields from the packet header, and the deparser writes the updated values of the extracted fields and inserts new fields if needed to the packet header at the end of the pipeline. Each match action unit is comprised of a matching table and the associated action(s). Upon a successful matching (*e.g.*, exact or ternary), an action will be executed according to the input parameter(s) from the table entry and the *runtime metadata*. The runtime metadata carries information that is valid only for the current packet across the match-action units and gets refreshed for each packet.

Besides *tables*, similar as in Software Defined Networking (SDN) switches, P4 contains an additional feature, namely *registers*, for stateful processing. Unlike metadata, registers are persistent beyond the single iteration of packet processing. Each register is defined as a *register type*, an array that consists of multiple *register entries*. Because registers can be updated from both control and data plane, this work intensively uses them to apply the new routing configuration at the correct time, after the configuration is received from the control plane at the data plane. As P4 abstracts the packet processing atomic actions from the underlying entity, it is *target-independent*, where target means running device and platform. Moreover, this work intensively uses *clone* to generate packets in the data plane.

The same P4 program can be compiled by different compilers to generate configuration files for different devices. P4 supports both software targets, *i.e.*, software switches running in commodity off-the-shelf servers, such as BMv2 [13] and T4P4S [74], and hardware targets with hardware acceleration capabilities. Examples are Netronome SmartNIC [60], NetFPGA-SUME [79], and Barefoot Tofino [61], which leverage Network Processing Units (NPUs), Field-Programmable Gate Array (FPGA), and reconfigurable ASICs, respectively, to realize the customization of pipelines.

We describe our specific P4 implementation in more detail in §8.

2.2 Local Verification of Consistent Updates

When implementing network updates, operators have to commonly choose a trade-off w.r.t if every completed update operation should be acknowledged by the controller, before sending out the next batch, or not. In the latter case, updates can be rapidly deployed without inducing round trip delays to the control plane, but it also comes with the cost of inconsistency, unless maintaining that, *e.g.*, the controller's network view was consistent and all involved switches successfully implemented the updates. However, even in a network without malicious actors, there are "*many causes that may trigger inconsistencies at run time*" [69] between the data and the control plane, such as, e.g., priority, control, and software bugs, hardware failures, bit flips, and misconfigurations [69, 71].

Some of these inconsistencies may then need "manual" and slow controller involvement to fix. Blackholes, loops, and congestion might not just be transient but relatively long-term: they might not be fixable until obtaining a consistent network view, then potentially resolving them by rolling back or attempting to rapidly deploy new updates. We will demonstrate in §4 how P4Update does not incur either of these downsides, unlike previous work.

To this end, P4Update builds upon previous theoretical work that proposed the combination of local verification and consistent network updates. The local verification aspect relies on so-called *proof-labeling schemes* [28, 49]: herein, a prover assigns labels (proofs) to the nodes, which the nodes then have to verify. A node may only rely on its own and its direct neighbor's labels for its yes/no-decision. In case the proof is correct and was correctly computed, all nodes output yes and the proof is accepted, whereas if the property to be proved is incorrect, at least one node must output no. Schmid and Suomela [67] made the connection to SDNs in this context, where proof computation can be centralized, and verification made local, for checking spanning trees. Foerster et al. [19] then expanded this idea to consistent network updates, with the task of migrating between two rooted spanning trees: by checking that a node may only migrate if its new parents distance (as assigned in the proof) to the destination is by one smaller than its own, loops are prevented. Follow-up work by Foerster and Schmid [20] then showed how to incorporate multiple sequential updates by introducing version numbers: hereby, a node can skip to a later update, without handling the intermediate updates. However, these updates still propagate along a logically *single* sequential layer and allow for no parallelization. We next show how P4Update incorporates parallelization of verification to speed-up the update process.

3 ALGORITHMIC INTUITION OF P4UPDATE

P4Update relies on each node obtaining different kinds of information for an update, an update version number and distances to the destination herein. The underlying idea is that with the flow states, the nodes can verify and coordinate the update process in the data plane. The control plane generates this information for all nodes that are involved in the new flow path $\mathcal{P}_n$ as follows: Each such (verification) information contains the new version number V and new/old distances D_n/D_o, where

- The version number is V unique and increments automatically for each new configuration. It is used to reject out-of-date update commands.
- The new distance D_n is calculated as the number of hops from the node to the egress node in $\mathcal{P}_n$. It is used to guarantee that each hop is closer to the egress node.

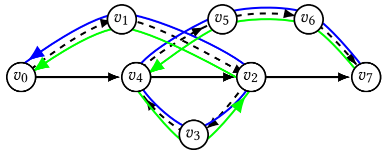


Figure 1: Illustration of SL- and DL-P4Update. The blue line represents the update in SL-P4Update and the green lines show the DL-P4Update segmentation.

We illustrate the different values in Fig. 1, where we denote the old path in solid lines by $\mathcal{P}_o$ and the new path in dashed lines by $\mathcal{P}_n$, respectively. In the old path $\mathcal{P}_o$ (v_0, v_4, v_2, v_7), we have $D_0(v_0) = 4, D_o(v_4) = 3, D_o(v_2) = 1, D_o(v_7) = 0$, and in the new path $\mathcal{P}_n$ $(v_0, \ldots, v_7)$, we have $D_n(v_0) = 7$, $D_n(v_1) = 6, \ldots, D_n(v_7) = 0$.

We next describe how P4Update incorporates a *single* sequential verification process (SL-P4Update), followed by a parallelized *dual* version (DL-P4Update), to speed up entangled updates. We will discuss in §7.5 how to select in practice which method to use.

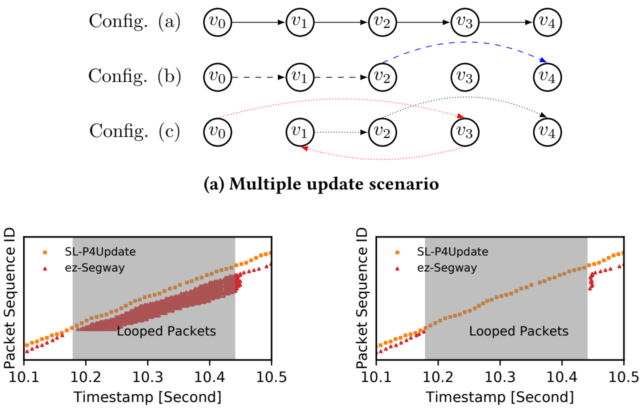
3.1 SL-P4Update Overview

The idea of the single-layer approach is to avoid complicated mechanisms and coordination procedures. To this end, we rely on the fact that by implementing the updates backward, from egress to ingress, no loops and blackholes will occur. As such, for the new version, the egress node can notify its child to update, which then notifies its child, and so on, until the ingress node is updated. In more detail, each node in this process can verify if the update is consistent, by checking that the new version is larger than the current and that its new parent has a smaller distance than its own — after which it can inform its child to update. If the data plane notification appears before the version from the control plane, the notification needs to wait in the node.

3.2 DL-P4Update Overview

Whereas the single-layer approach is efficient in the verification and coordination process, each node must update sequentially after another. In order to improve the update speed for intricate scenarios, we rely on the fact that some nodes update forward (downstream) along the flow path, which cannot induce loops, and hence updating these segments can be done in parallel [23]. On the other hand, when a node updates backward (upstream), we obtain dependencies on forward segments that have to be resolved first. The identification of the forward and backward segments is by comparison of the new distances w.r.t. initial or previous update distances. The decision whether to initiate a single- or dual-layer update is in the hands of the control plane and depends on the update scenario. As we will see in the evaluation later, simple update scenarios favor a single-layer approach, whereas complex scenarios benefit from parallelization.

Herein, in a dual-layer update, (path) segments are defined by the paths between node intersection of $\mathcal{P}_o$ (the old flow path) and $\mathcal{P}_n$ (the new flow path), where we call these shared nodes *gateway* nodes $\mathcal{G}$. More specifically, each segment has two gateway nodes, one closer to the global ingress w.r.t. its flow version, which we call ingress gateway (node), and one closer to the global egress, which we call egress gateway (node). For example, in Fig. 1, the



(a) Multiple update scenario

(b) Packets received at v_1.

(c) Packets received at v_4.

Figure 2: Demonstration of P4Update and ez-Segway for the scenario of inconsistent updates.

set of gateway nodes is $\mathcal{G} = \{v_0, v_4, v_2, v_7\}$, and the three segments are depicted in green: $\{v_0, v_1, v_2\}$, $\{v_2, v_3, v_4\}$, where, *e.g.*, v_2 is the ingress gateway and v_4 is the egress gateway, and $\{v_4, v_5, v_6, v_7\}$ as the last segment. Segments $\{v_0, v_1, v_2\}$ and $\{n_4, v_5, v_6, v_7\}$ are forward segments and can be updated independently: with respect to $\mathcal{G}$, they do not increase the distance and hence cannot produce a loop. On the other hand, the segment $\{v_2, v_3, v_4\}$ is a backward segment: w.r.t. $\mathcal{G}$, it increases the distance to the egress and hence needs to wait until the previous segment has finished updating, where v_7 (end of forward segment) is downstream of v_2 (start of backward segment): if v_2 updates before v_4, there will be a loop.

DL-P4Update Intuition. For ease of accessibility, we also use the above example of Fig. 1 to provide a different line of intuition how DL-P4Update works. Each gateway node has a distance in the *new* and *old* forwarding rules, and we can imagine that the old flow path distances represent segment IDs. As such, v_7 has a segment ID of 0, v_2 of 1, v_4 of 2, and v_0 of 3. A gateway node v can ask the next gateway node v' upstream (w.r.t. to the new forwarding rules) to join its segment and *inherit* its segment ID, which v' will only do, if the segment ID of v is smaller than the segment ID of v'. To give an example, at the beginning v_4 asks v_2, where v_2 will reject $(2 > 1)$, but v_4 accepts v_7 $(0 < 2)$ and v_0 accepts v_2 $(1 < 3)$, *i.e.*, backward proposals are rejected, but forward are accepted. We now only have 2 segment IDs left, 0 for v_0, v_4 and 1 for v_2, v_0. Next, v_2 accepts the proposal of v_4 $(0 < 1)$, inheriting the segment ID (or rather, old distance) of 0, and after, v_0 accepts the proposal of v_2 as well, *i.e.*, all gateway nodes are in the same segment ID. From a topological point of view, joining a segment with smaller ID (old distance) means that packets can only get routed closer to the destination, and by maintaining that invariant, no loops can appear.

4 CONSISTENCY OR UPDATE SPEED? PICK BOTH WITH P4UPDATE

To better motivate P4Update, we first evaluate a scenario where state-of-the-art work, in the form of ez-Segway [63], aims to be fast, but suffers from inconsistencies, and then where ez-Segway aims to be always consistent, but in turn faces long delays.

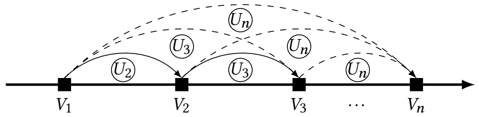


Figure 3: Illustration of the advantage of P4Update in terms of fast-forward updates. The x-axis shows the order of multiple configurations, each with a version number $V_1, \ldots, V_K$. Solid lines represent the updates of state-of-the-art proposals, and dashed lines represent the fast-forward possibilities of P4Update.

4.1 Risk Inconsistencies, Update Quickly?

To demonstrate P4Update's ability to avoid loops due to out-of-order updates from the control plane, we design the following scenario, depicted in Fig. 2. The initial configuration (a) of the forwarding path from v_0 to v_4 is depicted in solid lines. Afterwards, the new configuration (b) updates only the part from v_2 to v_4 (dashed lines). We assume messages of (b) from the control plane are, e.g., delayed, with the control plane being oblivious to it [69, 71]. The latest configuration (c) again updates some parts of the path (dotted lines). Note that if the updates for (b) are not lost or delayed, then ez-Segway will update in a consistent manner.

We run the experiment, starting from the configuration in Fig. 2a (a), where then the updates for Fig. 2a (c) are deployed, followed by the updates for Fig. 2a (b) shortly after. Regarding the experiment specifications, we generate data plane packets at a rate of 125 pps with a TTL of 64 at node v_0 in Fig. 2a, destined for the egress node v_4.

Fig. 2b shows the sequence ID of packets received at switch v_1 in Fig. 2a over time. The gray area depicts the time window after deploying the updates from Fig. 2a (c) until the missing updates from Fig. 2a (b) are sent. For ez-Segway, during this time window, all packets received by v_1 are trapped in a loop, namely v_1, v_2, v_3, as v_2 has not yet deployed its update from Fig. 2a (b). Only after v_2 has received its update and forwards to v_4 can the loop be resolved, and v_1 receives every packet only once. P4Update, on the other hand, by virtue of local verification, does not implement the inconsistent updates and receives every packet only once at the node v_1.

Fig. 2c shows the other side of the story, namely the sequence ID of the packets received at the egress node v_4. Whereas all packets arrive for P4Update, ez-Segway suffers losses due to packet TTL dropping to zero, after the v_1, v_2, v_3 loop has been traversed 21 times.

Hence, by optimizing for speed and not acknowledging updates before sending out the next set, prior work suffers from inconsistencies, which P4Update avoids by local verification.

4.2 Maintain Consistency, Delay Updates?

We next demonstrate P4Update's ability to safely skip ahead to the next update, while other approaches need to wait to maintain consistency. Fig. 3 illustrates this advantage. Assume a new configuration V_3 needs to be deployed, while the update of V_2, *i.e.*, U_2, is still ongoing. Here, unlike in the previous section, and in order to maintain consistency, ez-Segway makes the deliberate choice to wait until U_2 is completed before U_3 can be scheduled. P4Update, however, can trigger the update U_3 to directly go to V_3. For any number updates $V_3, \ldots, V_n$, P4Update can fast-forward to

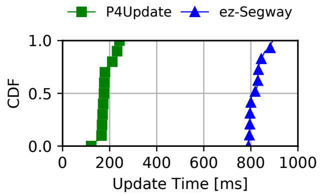


Figure 4: Demonstration of P4Update and ez-Segway for the (adversarial) scenario of two sequential updates.

V_n without creating loops and blackholes, *i.e.*, immediately deploy the latest update, as P4Update's verification ensures that every mix of implemented route update rules stays consistent.

To demonstrate the benefit of P4Update's ability to safely skip ahead, we design a two-consecutive-update scenario in a network with six nodes. Herein, the controller sends out updates for a complex update scenario U_2, but realizes only afterwards, due to a previously inconsistent view [69], that a simpler update U_3 would be more beneficial. As shown in §4.1, ez-Segway cannot jump ahead to U_3 without risking inconsistencies. P4Update however can locally decide in the data plane when nodes can safely jump ahead to U_3 and hence greatly improve the update time. To this end, we take an analogous setup as in §4.1 and measure the completion time for U_3. We measure 30 runs and plot the times in Fig. 4, dotting every third run. Our results show that P4Update is about 4× faster than ez-Segway in this setting.

4.3 Takeaway

As prior systems have no local verification for inconsistent routing updates, they can incur packet loss, by, *e.g.*, updates being reordered or an inconsistent view (§4.1), or must continuously involve the control plane loop (§4.2), which can take up to hundreds of miliseconds [52] each time. P4Update can recognize and prevent these inconsistencies with its verification capabilities, as shown in §4.1. Moreover, P4Update can safely jump to a later version of the configuration even if the ongoing update is not yet finished, hence providing significant speed-ups, as we illustrated in §4.2.

5 NETWORK UPDATE MODEL

Network Model. Following standard assumptions, we model the network topology as a connected graph. Each node denotes a P4 switch, and one designated node acts as the controller. Abstractly speaking, the control plane interacts with the P4 switches via predefined interfaces, where P4 switches communicate with each other via the data plane (see §6).

Flow and Routing Model. We consider traffic flows between ingress and egress switches, where each flow is routed from some source to some destination node along a specified path. In this context, we can order the nodes w.r.t. to a flow f as ingress$=v_0, v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_k$ =egress. Herein, w.r.t. f, we will call the next node downstream parent or successor, *e.g.*, v_{i+1} for v_i, and analogously, v_{i-1} is the child or predecessor of v_i. The routing of each flow is decided by its forwarding graph, implemented via forwarding rules that match flow packets on the traversed switches.

Consistent Update Model. We investigate three fundamental consistency properties [21]: *blackhole*, *loop*, and *congestion* freedom. Blackhole freedom is maintained if every packet arriving at a switch has a matching forwarding rule, loop freedom is maintained if the corresponding forwarding graph has no loops, and congestion freedom is maintained if each link capacity is respected under the total flow assignment. We assume the controller is aware of a maximum flow rate that is respected for each flow, respectively enforces it.[1] We want to maintain all three properties under network updates, where the controller decides on new forwarding rules and distributes them to the P4 switches to be implemented. However, even with (logically) centralized control, networks remain asynchronous distributed systems, and thus we cannot assume synchronous forwarding rule changes [21].

Verification Model. We consider the scenario where the distributed forwarding rules may be inconsistent, *e.g.*, due to an incorrect or outdated controller's network view [69, 71], dropped update packets, or update packet reordering [32, 50].[2] Even when the forwarding rules are consistent on their own, the network can still end up with, *e.g.*, forwarding loops when updates are sent in rapid succession (or some switches delay/fail their updates): the mix of two consistent forwarding graphs can easily contain inconsistencies such as forwarding loops. We hence study verification models that prevent conflicting/incorrect rules from being implemented.

We utilize a prover-verifier mechanism, where the prover (the controller) proposes a proof (network updates) that the verifiers (the switches) implement in a distributed and consistent way [19, 20]:

(i) If the network updates are consistent, sufficient resources are available (*e.g.*, link capacity) and follow the specified protocol, then the switches must eventually converge to the latest set of updates and maintain consistency properties.

(ii) If the network updates are inconsistent (*e.g.*, contain a loop) or do not follow the specified protocol, then switches may only implement them as long as the partial implementation is consistent (w.r.t. blackholes, loops, link capacity).

6 P4UPDATE ARCHITECTURE

The P4Update framework follows the general architecture of programmable networks. Based on runtime information, the control plane decides the new routing configuration, *i.e.*, flow paths, and pushes it to the data plane. The update decisions, *i.e.*, whether the new configuration is consistent and if yes when to update, are made locally on each data plane forwarding device/node. In P4Update, we alleviate the communication overhead between both planes while offering efficient and consistent network updates.

In Fig. 5, we illustrate the system components of P4Update. We define four types of *control* messages in different phases of P4Update's operation: Flow Report Messages (FRMs), Update Indication Messages (UIMs), Update Notification Messages (UNMs), and Update Feedback Messages (UFMs).

Control Plane Component. The control plane listens to the emergence of new flows reported by the data plane through FRMs (the green message in Fig. 5). The graph information is maintained in

[1] A common assumption for consistent updates [21], already present in Microsoft's SWAN [37]. Conceptually, it could also be replaced by other in-network approaches/measurements. However, we need some (guaranteed) bounds, as else congestion could already occur without any updates.

[2] We note that of course also the P4 programs themselves could have bugs [70], but we will assume for our proofs that our implementation will be done correctly, as the verification process itself is relatively simple, see, *e.g.*, Algorithm 1.

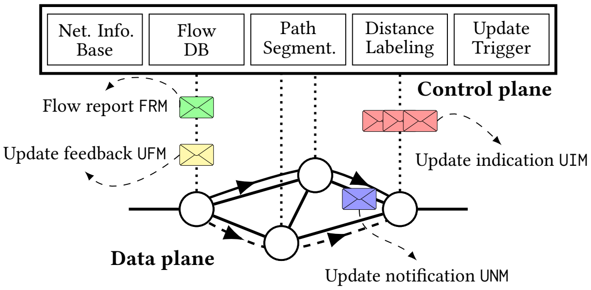


Figure 5: System component illustration of P4Update. Upon a new flow emerging in the data plane (DP), the UFM notifies the control plane (CP). The CP sends the UIM to the DP for a new configuration and triggers the update process. Then, the UNM in the DP coordinates verification and enables updates. Finally, the UFM from DP to CP reports on the update status.

the Network Information Base (Net. Info. Base). The flows are maintained in the Flow DB. When the network condition changes due to, *e.g.*, traffic load imbalance, the control plane computes a new configuration to reroute the traffic based on the pre-configured policy. For each flow update, the control plane decides the update and verification contents *e.g.*, distance, for each flow and encapsulates them into the UIM. To improve update efficiency, the new path can be segmented (*i.e.*, split into sub-paths) to enable parallelized updates (§3). After the control plane triggers the update, it pushes the UIMs to the data plane, and waits for the response in an UFM for update success or alarm (in case of an inconsistent update). Upon receiving UFM(s) from the data plane, it updates the flow state in the Flow DB.

Data Plane Component. To store flow information and assist future updates, the data plane maintains an Update Information Base (UIB). Once a node has received an UIM, it uses the label contents to update the per-flow state in the UIB. Before moving to the new forwarding state, it needs the help of an UNM to perform verification to satisfy blackhole and loop freedom (§7.1). Apart from processing UNM, it also coordinates the UNM transmission (§7.2). To maintain congestion freedom of a single flow, it dynamically calculates the remaining capacity and determines local priorities. To accelerate multiple flow updates, it schedules inter-flow dependencies based on the local priorities (§7.4). Whether moving to the new state or not, the data plane will generate UFM(s) to inform the controller on the recorded update state and prepare for the next update.

7 P4UPDATE ALGORITHMS

P4Update provides two update mechanisms, an efficient single-layer approach for common settings (SL-P4Update), and a more sophisticated dual-layer approach for intricate update scenarios (DL-P4Update). Both approaches structurally rely on egress distance and update version numbers. Moreover, P4Update supports update consistency w.r.t. topology (blackhole- and loop-freedom) and performance (congestion-freedom) properties.

However, as we will discuss and evaluate later on, DL-P4Update is not always superior to SL-P4Update, or vice versa. DL-P4Update implements a "dual" logical information flow, which speeds up entangled scenarios that can be segmented into independent update partitions: simplified, DL-P4Update updates these segments on their own, and passes on update information between the segments to resolve dependencies. On the other hand, in small and simple update scenarios, *e.g.*, in data center networks, the new route takes a small disjoint detour, this overhead is not necessary and SL-P4Update can update more quickly by just pushing the updates along the route.

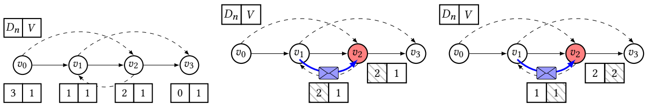
We provide the local topology verification procedures in §7.1, followed by the update and coordination mechanisms in §7.2, and their correctness in §7.3. Moreover, congestion-freedom depends on the interaction between different flows, whereas blackhole- and loop-freedom have no dependencies on other flows. We hence also show how to resolve such inter-flow dependencies in the data plane in §7.4. We then lastly discuss in §7.5 how to combine the single- and dual-layer approaches into a common framework and refer to §8 for implementation details.

7.1 Blackhole & Loop Freedom Verification

General Setting. After receiving the update and verification content encapsulated in the indication messages UIM from the control plane, all nodes involved in the new path $\mathcal{P}_n$ that need to be updated can start the local verification process in parallel. The egress node in the new path can apply the new configuration directly, and afterward, generate an update notification message UNM to trigger the update process of its child nodes. The UNM also encapsulates the information of the previous configuration, *i.e.*, old version number V_o and distance to egress node D_o, and the information of the current configuration, *i.e.*, new version number V_n and distance to egress node D_n. If a node receives a UNM for a version for which it has no UIM yet, it waits until the arrival of the UIM.

Inconsistent Updates. We note that the control plane can also output an *inconsistent configuration*, *e.g.*, in the following scenarios: (i) a node state update is (temporarily) invisible to the control plane due to software bugs; (ii) the new network state is miscomputed and contains loops or blackholes; (iii) subsequent updates are sent out in the wrong order. Furthermore, the content of UIM or UNM could also be corrupted, or the messages are lost completely. P4Update verifies the correctness of the update information before implementing an update to avoid an incorrect configuration.

SL-P4Update Verification. The verification procedure is described in Alg. 1, and on success, it outputs a verification state of $VS = 1$. For each non-egress node, it waits for the arrival of UNM from its parent. Upon receiving UNM, the node leverages D_n obtained from UIM to verify if UNM indeed comes from a possible parent, by checking that the versions match and that the new distance is smaller by one. Should the distance be incorrect, UNM is dropped, and the controller is informed for further optional analysis. If the corresponding UIM for the UNM has not yet arrived and a *waiting procedure* is necessary. The ability to perform such waiting locally in the data plane is one of our main motivations to use P4, see §8 for details. Else, if the version in UNM is outdated, then UNM could lead to inconsistency, *i.e.*, the message is dropped, and the controller is informed as well.
Example. Fig. 6 demonstrates three scenarios (i), (ii), (iii) to showcase data plane verification.



(a) Consistent. **(b) Distance errors.** **(c) Version # errors.**

Figure 6: Verifying update consistency, where the UNM is forwarded via v_3, v_1, v_2, v_0 to update from the solid old path to the dashed new path. Scenario (a) shows a successful update without inconsistency issues, whereas (b) and (c) have inconsistent update information.

Scen. (i): In Fig. 6a, assume all nodes have received their UIM for the new version $V(\text{UIM})$. When v_1 receives the UNM from v_3, the versions match and it remains to check the distance, where $D_n(v_3) = D_n(\text{UNM}) + 1$. v_1 can update and notify v_2, which succeeds in verification, and notifies v_0, which also updates. §7.2 covers the update process beyond verification.

Regarding updates that are inconsistent, we now present two examples, namely an erroneous new distance D_n or version number V, that is carried by UNM.

Scen. (ii): In the former case (Fig. 6b), the distance D_n of v_2 and its parent v_1 cannot be the same as its own: identical distances can cause a forwarding loop.

Scen. (iii): In the latter case (Fig. 6c), the node v_2 is not allowed to have a smaller version number than its parent node v_1: falling back to older updates could also induce loops.

For the two inconsistent cases, we make the design choice to report the inconsistencies to the control plane, which can, *e.g.*, schedule a new update to resolve it. From a *holistic* view, we could also try to run, *e.g.*, a distributed algorithm to resolve the situation. However, for efficiency purposes, we want each node to perform local verification entirely on its own without further interaction with other nodes or the control plane. Additionally, we also want to avoid nodes to "lock" the states of other nodes, while they gather new information: updates should be decided purely locally.

DL-P4Update Verification. In order to increase the update speed, we design P4Update's dual-layer verification to update *forward* segments earlier, as these segments cannot induce a loop. On the other hand, *backward* segments wait until preceding segments have updated. This identification of forward/backward segments relies on the comparison between old and new distances.

In more detail, the ingress gateway node of a backward segment (v_2 in Fig. 6) may only update if the sequence of preceding segments resolves the loop potential of entering the backward segment. Furthermore, note that by definition of $\mathcal{G}$, all nodes between the last and first node of a segment lag at least one version number behind, or might yet not have any forwarding rules at all: hence, we update those sequentially inside their segment. Notwithstanding, these nodes in-between can already update their forwarding rules before the gateways have updated, to increase update speed.

With respect to verification, we first reject outdated updates and wait for future updates, for which no UIM is yet present. We next cover nodes inside segments, *e.g.*, the nodes v_1, v_3, v_5, v_6 in Fig. 1. These nodes are identified as their version number lags behind, *e.g.*, because they currently have no forwarding rules. Assuming the new distance is smaller, they can update, where they inherit their parents' old distance label, to pass on downstream. We next cover how to verify non-egress nodes that have the last version number, *e.g.*, the nodes v_0, v_4, v_2 in Fig. 1. For these, the new distance *and* the old distance must be smaller, and their previous update must not be dual-layer, only then do they update.

Algorithm 1: SL-Verification (Data Plane) at node v

```
Input  : Highest indication UIM, notification UNM
Output: Verification state $VS$, current state at $v$
$VS = 0$; // Initialize verification state
$D_n(v) = D_n(\text{UIM})$; // Get $v$'s new distance in UIM
$V(v) = V(\text{UIM})$; // Get $v$'s new version no. in UIM
if $V_n(\text{UNM}) = V(v)$ then
    if $D_n(v) = D_n(\text{UNM}) + 1$ then
        $VS = 1$; // Verification successful
    else
        Drop UNM, inform controller;
    // Incorrect distance, could cause a loop
else if $V_n(\text{UNM}) > V(v)$ then
    Wait for UIM; // Wait until new UIM arrives
else if $V_n(\text{UNM}) < V(v)$ then
    Drop UNM, inform controller; // Update outdated
```

7.2 Blackhole and Loop Coordination

We next show how the data planes coordinates the blackhole and loop-free network updates. Herein, each node v in the new flow path obtains its new information via an UIM, which becomes triggered and verified by an UNM message.

SL-P4Update Coordination. The single-layer starts at the egress node v_k, which upon receiving a correctly formed UIM message (*i.e.*, with higher version number and new distance of 0) sends an UNM message to its child v_{k-1} in the new version, which the child then verifies with Alg. 1. In case of positive verification ($VS = 1$), v_{k-1} updates the UNM message for its child v_{k-2}, in particular the version and distance of v_{k-1}, and v_{k-1} writes its new forwarding information from the UIM into its table entries. This process is iterated until (i) the ingress node v_0 is reached, followed by a positive verification outcome and v_0's forwarding rule changing to v_1, or (ii) verification fails, *e.g.*, because of inconsistent UIM messages or because nodes already updated to higher versions. See Fig. 1 for an example, where UNM messages are sent hop-by-hop along the blue path from v_7 to v_0.

DL-P4Update Coordination. The dual-layer update coordination is similar to the single-layer, but attempts to perform the updates in parallel for each segment. Herein, the individual segments can update conceptually as in the single-layer, with the exception of the (most upstream) gateway of each backward segment, e.g., v_2 in Fig. 1. These gateway nodes can only update when the received UNM message indicates that enough downstream segments are updated, such that their update will not form a loop. We defer the technical algorithm details to the appendix (Alg. 2) due to space constraints.

7.3 Blackhole and Loop Freedom Correctness

We next provide proof sketches (due to space constraints) for the correctness of the blackhole and loop freedom properties of P4Update.

Theorem 1. *Assuming the single-layer approach for loop and blackhole freedom is used, the network will be blackhole- and loop-free, even under inconsistent control plane messages.*

Proof Sketch. Blackhole freedom is maintained by only updating when informed by a parent, in turn guaranteeing rule existence downstream. For loop freedom, following the arguments in [20], we either increase version numbers or reduce distance, *i.e.*, eventually reach the destination. □

Theorem 2. *In the single-layer approach for blackhole and loop freedom, the flow path will eventually converge to the highest consistent version update pushed by the control plane.*

Proof Sketch. The highest version update travels from egress to ingress, updating the nodes along its path. □

Theorem 3. *The dual-layer approach is blackhole and loop-free, even under inconsistent control plane messages.*

Proof sketch. Blackholes are prevented analogously as for the single-layer, where correctness for loop freedom relies on inheriting the old distances (again as in the single-layer), with the new distances providing speed-up benefits. □

Theorem 4. *Assuming that the dual-layer approach for blackhole and loop freedom is used and that all impacted nodes were last updated by a corresponding single-layer approach, the flow path will eventually converge to the flow rules pushed by a dual-layer approach, if consistent and with highest version.*

Proof Sketch. The argument is analogous to the proof sketch for Theorem 2, using old distances, combined with the counter for symmetry breaking, for correctness. □

7.4 Congestion-Freedom: Resolving Dependencies in the Data Plane

Unlike blackhole and loop freedom, congestion freedom has inter-flow dependencies when scheduling multiple flow updates in parallel, due to limited link capacities.[3] The latter imposes resource scheduling challenges, in some sense already seen in the classical 15-puzzle [43]: as flows are moved atomically, we need capacity on the old and the new link (path) for a single flow update, and finding the correct order is intractable already for relatively simple update scenarios [21]. In fact, it is easy to show that already scheduling flow updates for a single node, with just two outgoing links, is NP-hard via a reduction from the partition [27] problem. Due to the inherent intractability of consistent congestion-free flow update scheduling [17] and in order to alleviate the control plane load, we hence propose a dynamic and efficient heuristic scheduler in the data plane in P4Update. Correctness details are deferred to §A.2.

The main idea is that if flow f_1 wants to move from link e_1 to e_2, it might be blocked from doing so until some flow f_2 moves from e_2 to e_3, freeing up some capacity on e_2 in the process. Conceptually similar to the dependencies between forward and backward segments for loop freedom, we can now define inter-flow dependencies: if some flow f cannot move to link e due to e's insufficient remaining capacity, then all flows that desire to move away from e obtain *high priority*, whereas flows without such dependencies stay at *low priority*. In more detail, when a low priority flow desires to move to a link e with sufficient capacity, it can only do so if there is no high priority flow that desires to move to e. On the other hand, high priority flows can move immideately with sufficient capacity, in order to resolve dependencies.

Note that this process is completely local to a node and that the priorities are dynamic: the node v needs no pre-computed flow priorities, which might be outdated over time and incur control plane load, and adjusts the priorities based on the currently waiting flows.

Moreover, we need some mechanism such that the nodes know the flow sizes, such that we can perform local computation if the remaining capacity suffices. For our purposes, we follow the assumption, often made in this context [21], that each flow has an immutable and, *e.g.*, by the ingress enforced, upper size bound known by the controller,[4] where congestion freedom is maintained if the sum of these size bounds does not violate link capacity. We are not aware of prior work on local and dynamic flow update scheduling in the data plane, and show its efficiency in §9.

7.5 Single and Dual-Layer Combination

Conceptually, SL- and DL-P4Update represent two extreme ends of the spectrum: SL-P4Update forgoes parallelization by combining all updates into a single segment, whereas DL-P4Update attempts to create as many segments as possible. At first, it might seem that DL-P4Update is strictly superior, as it maximizes the number of parallel updateable segments. However, this comes with an overhead for each segment, which in the end can negate all parallelization benefits, depending on the considered scenario.

We as thus propose to deploy P4Update as follows, which we evaluate in §9: 1) Updates, which install new forwarding rules on relatively few nodes in forward segments, are handled by SL-P4Update. 2) All other updates are handled by DL-P4Update.

8 P4UPDATE IMPLEMENTATION

In this section, we describe the implementation details of P4Update based on the P4 programming language [7]. Taking a look back at Fig. 5, we describe the details of the data plane, *i.e.*, storing the information needed for updates, generating UNM, and handling UIM and UNM. For the control plane, we describe the flow information database (Flow DB) and the generation of UIM to schedule an update. We develop a prototype of P4Update, consisting of a P4 data plane and a control plane in Python. The data plane algorithms of P4Update as described in §7 are implemented in > $1k$ lines of $P4_{16}$ [14] code. The BMv2 switch [13] is chosen as the P4 target.

Data Plane To implement the update procedure, we need to store the update information in the UIB (introduced in §6) as registers, which are summarized in Table 1 in Appendix B. The data plane performs the following tasks: (1) generate FRM, (2) process UIM, (3) generate and process UNM, (4) generate UFM.

We use registers to temporarily maintain current and new version of routing and graph information (e.g., distance). The new

[3] Enabling congestion-freedom for a single flow is relatively simple: the node checks if the capacity on the outgoing link suffices.

[4] Conceptually, other methods to determine/limit flow sizes would also be possible, as long as the nodes know the respective flow size bounds.

version is assigned by controller via UIM. We use the *clone* primitive to generate UNM at gateways. A one-to-one port-based forwarding table is used to determine the clone session of a UNM. The UNM header is populated in the egress-pipeline to inherit the label contents of updated version. For the SL-P4Update, only one layer of UNM is generated. For the DL-P4Update, inter-segment UNM is generated at the egress node of the flow while the intra-segment of UNM is generated at the egress node of each segment. Upon the arrival of UNM, the neighbor's state is unpacked from the notification header which is compared with the current node's state. If the first-layer UNM arrives at the ingress node, it is then transformed to UFM and sent to the controller. The second-layer UNM is dropped at gateway nodes.

Control Plane The control plane stores the flow IDs in the flow DB, and uses it to manage flow state and make fine-grained update control for each flow. The graph information is maintained in the NIB which represents the current view of the network. Having calculated a new path for a flow and UNM with the help of NIB, the control plane generates a unique UIM for each switch. The UIM contains the distance, the version number, the flow size and the egress port of the new path. Sending the UIMs triggers the update process in the data plane. After the update has been finished in the data plane, the control plane receives the UFM. The controller then sets the update state to finished in the NIB. For the dual-layer approach, the controller calculates the segmentation described in §7.

9 EVALUATION

We evaluate P4Update through emulation in Mininet and show that P4Update (i) is advantageous in terms of the overall update time, and (ii) has scalable control plane algorithms.

9.1 Setup

We evaluate P4Update on a dedicated workstation on an 8 core Intel Xeon E3-1275 CPU at 3.6 GHz, 32 GB of RAM, and Ubuntu Linux 16.04. The control plane runs in a single thread. Beyond the example in Fig. 1, we consider two WAN topologies, namely B4 [39] and Internet2 [1], and a fat-tree with $K = 4$ as a DC topology. B4 is Google's private backbone network connecting its facilities across the globe, whereas Internet2 is a research network connecting several sites in the US. The latency of each link is calculated based on the geographical distance and the propagation speed through optical cables, *i.e.*, around $2 \cdot 10e6$ km/s. For the topology in Fig. 1, we assume a homogeneous link latency of 20 ms. For the WAN topologies, the physical controller resides at the centroid node, to minimize worst-case control latency. For the fat-tree topology, the control plane latency is randomly sampled from a normal distribution measured by [38].

In total update time evaluation, we compare P4Update to two state-of-the-art proposals, namely centralized updates as in [57] and the distributed updates from *ez-Segway* [63]. In §9.2, two scenarios are considered. For the single flow scenario, the old and new paths of a flow have been intentionally selected to traverse a long distance within the topology and to trigger segmentation. The intention is to highlight the differences on a per-flow basis, focusing on blackhole and loop freedom, as the new path is assumed to have sufficient capacity. For the multiple flow scenario, each node selects another node uniformly at random as a destination, where the old flow path is a shortest path and the new flow path is the 2nd shortest path. The flow sizes are generated according to the Gravity Model, as proposed by Roughan [66]. The generated traffic aims to be close to the network's capacity, where if the new flow paths are not feasible w.r.t. to capacity, we repeat the traffic generation. The intention of this scenario is to test the impact of multiple inter-flow dependency and accordingly the effect of P4Update's data plane flow scheduler.

In all scenarios evaluated in this section, we do not assume the adversarial effects studied in §4, as we do not update a flow multiple times: Our goal is to show that P4Update even outperforms prior work under their assumed evaluation settings. Hence, our multi-flow setup assumes no extra, *e.g.*, control load delay on the switches, to give a direct comparison to the evaluation of *ez-Segway* [63]. In order to better compare with the setting of *Dionysus* [57], which is motivated by variations in node update time, we modify the single-flow setup s.t. for each run, each node is slowed by a random delay when updating rules, generated by exp(100)ms in *NumPy* [2].

The update of a flow is considered to be complete when the whole ingress-to-egress flow path is established for the new rules, which we record with a packet traversal, whose success is reported to the controller. For multiple flows, we take the completion time of the last flow update. With respect to single- and dual-layer combinations, we follow the strategy proposed in §7.5, choosing the single-layer approach when we have only forward segments with at most five nodes to be updated.

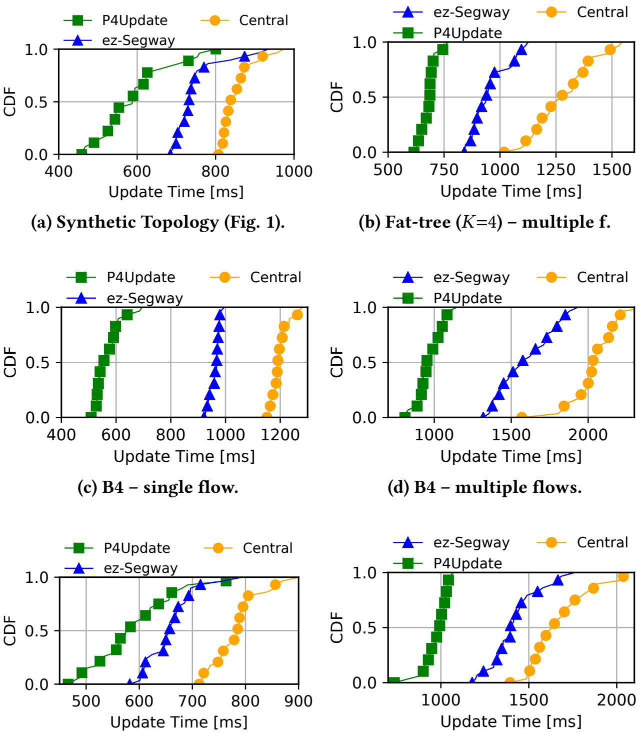
Computation Time Measurement. The evaluation in §9.3 focuses on the computation time of the control plane algorithms, in comparison to *ez-Segway* that uses a similar approach. The tests are performed on B4 and Internet2, as well as another two networks from the Topology Zoo [48]. The computation time is recorded for 1000 updates.

Previous Work. In order to make a fair comparison, we adapt two state-of-the-art proposals for our evaluation framework, called Central and ez-Segway, described next.

Centralized Updates. State-of-the-art centralized systems utilize a dependency graph to greedily update as many nodes as possible. In our implementation, the control plane computes the dependency relationship and then schedules updates to the nodes that could be updated in parallel (denoted as $\mathcal{P}_n^1$). After applying the new configuration, each node in $\mathcal{P}_n^1$ sends a notification back to the control plane. Upon receiving the notifications, the control plane computes a new dependency relationship based on the current state in the data plane. Afterward, a new set of nodes is updated (denoted as $\mathcal{P}_n^2$), and the notifications are generated accordingly. After N rounds, all nodes are updated, *i.e.*, $\mathcal{P}_n = \mathcal{P}_n^1 \cup \mathcal{P}_n^2 \cup \cdots \cup \mathcal{P}_n^N$.

Due to asynchronous update speeds and heterogeneous link delays, the control plane is expected to receive update notifications at different times. Besides the network update, the control plane is also responsible for other tasks such as new path setup and flow monitoring, which can involve a huge amount of control messages that need to be processed. Therefore, the *intermediate* update notification messages, among other control messages, will experience both queuing delay and processing delay [40].

ez-Segway. Our implementation follows [63]. We start with the update mechanisms for a single flow with enough capacity on the new path. Based on the old and new flow paths, the control plane



(a) Synthetic Topology (Fig. 1).

(b) Fat-tree (K=4) – multiple f.

(c) B4 – single flow.

(d) B4 – multiple flows.

(e) Internet2 – single flow.

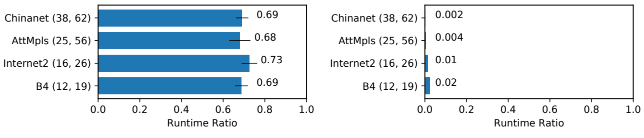
(f) Internet2 – multiple flows.

Figure 7: Update time CDF (using `Central` as a baseline): Single flow scenarios are shown on the left, multiple flow scenarios are on the right.

groups switches into *in_loop* and *not_in_loop* segments, where the update order within each segment is encoded into the egress of each segment. If two connected segments are both *not_in_loop*, they will be updated in parallel. The *in_loop* segment is updated after the *not_in_loop* segment. Therefore, the nodes that connect the *in_loop* and *not_in_loop* segment are marked as gateways, and they have to resolve the dependency between *in_loop* and *not_in_loop* segments. The nodes inside the *in_loop* segment wait for the finished updates of dependent *not_in_loop* segments. Once receiving the notifications of finished updates of *not_in_loop* segments, the gateways start to update switches within *in_loop* segments. Instead of using a local controller to encode the predecessor-successor relationship, we encapsulate the current state of switches into the notification message, and the nodes can locally determine when to update. For the inter-flow dependency resolution under additional congestion freedom, ez-Segway implements a centralized dependency graph generation, which assigns three types of update priorities along nodes in segments.

9.2 Total Update Time

We measure the update time from the sending of UIM messages to the receiving of UFM messages. Fig. 7 shows the empirical distribution of the update time for 30 runs in single flow scenarios (left column) and multiple flow scenarios (right column) The results in



(a) Preparation w/o cong.-fdm.

(b) Preparation with cong.-fdm.

Figure 8: Ratio of the control plane preparation time between DL-P4Update and ez-Segway. The 2-tuple inside the brackets denotes the # of nodes and edges in each topology [48].

Fig. 7a correspond to the topology in Fig. 1. P4Update is considerably faster than ez-Segway, and Central both in Data Center and WAN topologies.

In our experiments, DL-P4Update was picked for the segmented single-flow scenarios and SL-P4Update for the multiple-flow scenarios. For the latter, all P4Update deployments complete before the other two competitors, ez-Segway and Central—but P4Update is also significantly faster for the single-flow scenarios.

SL-P4Update performs roughly equal to DL-P4Update for Internet2, but is slower than DL-P4Update for Synthetic by 31.5% and for B4 by 12.5%. In particular, P4Update (by choosing DL) has strongly improved performance over ez-Segway for all three single flow settings (Synthetic: −18.5%, B4: −40.9%, Internet2: −9.3%), which in turn also outperforms Central—which comes due to two reasons:

First, P4Update involves fewer rounds of communication between the control and the data plane; Second, P4Update leverages high degrees of parallelism when updating multiple segments, especially the coordination between first-layer (blue message in Fig. 1) and second-layer (green message in Fig. 1) notification. The path segmentation of DL-P4Update is applied to the whole topology, and there is no dependency within all forward segments, whereas in ez-Segway, potential dependencies between segments delay updates inside some segments. The Fat-Tree topology (Fig. 7b) is interesting to mention since it only has forward segments. Even though, the update speed of P4Update is still faster since it can choose SL-P4Update without segmentation based on the old and new flow conditions. In the multiple flow scenario in Fig. 7 (right column), P4Update shows advantages (fat-tree: −28.6%, B4: −39.1%, Internet2: −31.4%) over ez-Segway. Herein, the picked SL-P4Update improves over DL by −27.3% for Fat-Tree, −39.2% for B4, and −27.2% for Internet2.

Takeaway. For multiple flow scenarios, P4Update outperforms ez-Segway w.r.t. total update time in average by at least 28.6% to 39.1%, depending on the chosen topologies, with even significantly more improvements over Central.

9.3 Control Plane Preparation Time

Fig. 8 shows the ratio of control plane preparation time between P4Update and ez-Segway, where 8b accounts for congestion freedom and 8a does not. Here a value of 1.0 means that both take the same amount of time, whereas values below 1.0 indicate DL-P4Update being faster.

The bars correspond to the average of 30 runs with 99% confidence interval. The numbers inside the brackets to the left of each bar report the number of nodes (left) and edges (right) of each topology, where the number to the right reports the mean ratio.

This experiment reflects the cost for the control plane to prepare the respective configuration when the flow routing paths changes.

For all topologies in 8a, the ratio stands near 0.7, meaning the control plane preparation of P4Update saves around 30% time compared to ez-Segway. These results show a similar level of scalability for P4Update's and ez-Segway's control plane algorithms, when not accounting for congestion freedom. Figure 8b paints a very different picture however. When ez-Segway needs to compute the dependency graph in the control plane, unlike P4Update's distributed offloading to the data plane, especially resolving congestion without global and local controller involvement, P4Update scales better by a factor of 50× (smallest) to 500× (largest network).

Takeaway. The P4Update control plane computation is scalable in terms of runtime w.r.t. topology size, with significant advantages over ez-Segway under congestion freedom.

10 RELATED WORK

Consistent Updates. Consistent network updates in SDNs have been widely studied in the last decade, we refer to a recent survey for an overview [21]. Reitblatt et al. [64, 65] kick-started the field by proposing a 2-phase commit to maintain consistency, where flow rules pertain to tags or version numbers, stamped in each packet. Once the new updates are successfully deployed network-wide, ingress nodes apply new flow tags for each packet, guaranteeing consistency. However, straggling nodes severely delay updates [42], packet headers change, and the required rule space can double.

Mahajan and Wattenhofer proposed to build dependency graphs for loop freedom [57], further studied in, *e.g.*, [24, 42]. Beyond weaker consistency instantiations [23] and intractable integer programs, dependency graphs form the state-of-the-art approach for blackhole and loop freedom, *e.g.*, implemented by Google's B4 [36]. We hence compare P4Update with centralized dependency graphs and show significant speed-ups, as also seen by ez-Segway [63].

ez-Segway [63] is the state-of-the-art for decentralized consistent network updates in SDNs. The authors depart from the concept of update ordering via the control plane and deploy the network updates in the data plane. Herein the controller partitions the updates into segments and sends them all out at the same time, upon which the update process propagates upstream through the data plane. Whereas the feasibility in P4 is only considered theoretically, they demonstrate deployment in an OpenFlow setting with per-switch local controllers. As a contribution to the community, we implement their algorithm in P4 as well and confirm the results of their simulations over centralized approaches. Due to more fine-grained parallelization for complex scenarios and speed-up updates for common settings, we improve upon their accelerated update times. For example, DL-P4Update provides a larger number of segments that can be updated simultaneously, instead of sequentially, and can also update the forwarding rules of nodes inside backward segments right away. Unlike ez-Segway, P4Update provides verification and hence can also handle, *e.g.*, multiple flow version updates in a faster and more consistent fashion. Moreover, ez-Segway performs a static dependency graph calculations in the control plane, which we offload to the data plane and dynamically adapt to the current state, where P4Update incurs significantly less control plane computation and achieves better scalability, while maintaining faster update speed.

Even though no practical work currently provides distributed verification of updates, there is further work on other consistency properties, such as congestion freedom [3, 9, 37] or waypoint traversal [55]. The concept of time has also been used in network updates to combat asynchrony [58, 76] or by direct insertion into packets [53]. Notwithstanding, perfect timing is not the silver bullet to solve all consistency issues, already for updating a single forwarding rule [18, 25].

Zhou et al's [78] work can be seen as complementary to P4Update. Their consistency generator could be used to, *e.g.*, generate the new flow paths and to further increase the network performance: even though P4Update will reject inconsistent updates, recomputing and redeploying those updates via the controller can take significant overhead. On the other hand, P4Update's verification is purely local at the switches, without any data or control plane invocation, also takes lost/reordered updates into account, and hence a combination of both systems would be interesting.

Local Verification. Local verification of consistent network updates was first proposed in [19], extended to multiple updates in [20]. Expanding on proof-labeling schemes [28, 49] in SDNs [67], a distributed proof is deployed by the controller, and the switches migrate towards newer proofs.

However, prior work is theoretical, requires a sequential traversal of the new updates, and does not account for congestion. Conceptually, we build upon this approach in the single-layer P4Update setting for loop freedom, which is faster than a centralized approach, where the latter and ez-Segway both do not provide verification. A further major difference in this context is P4Update's dual-layer update approach, the first locally verifiable network update method that allows for parallelization to increase the update speed.

Beyond Consistent Updates: PDP & P4. The number of works focusing on leveraging PDP for different network functions [35, 41, 59] is constantly growing. Some of them emphasize on the acceleration capability of PDP to solve known networking problems efficiently [41, 45, 59], while others leverage the flexibility of PDP to address new challenges [29, 46, 68]. Our work falls in the former class and targets the consistent network update problem. We also observe works that leverage registers for low-latency failure recovery [68], heavy hitter detection [31, 54, 72], time synchronization [45], and load balancing [46]. P4Update makes use of the registers similarly to that of the work of heavy hitter detection, i.e., the distance, version number, and other helping variables are defined per-flow and indexed by the flow ID. From a syntax perspective, register usage can incur logical errors and consistency problems and therefore, should be addressed properly [34, 62].

11 DISCUSSION

Distributed Control Planes. For our evaluation, we assume only one physical controller. A major component of the overall update time is the control plane latency, which can be even diminished with a distributed control plane in reality [16]. This might introduce consistency challenges within the control plane [51], which P4Update can locally resolve.

Data Plane Overhead. The overhead mainly comprises the processing of UIM and UNM, and the resubmission of these messages to ensure the correct update logic. Even though our prototype is based on a software target, we envision that with a hardware target, the processing can be accelerated with negligible overhead [59]. For the resubmission, the switch can reserve a slice with, *e.g.*, virtualization techniques [30, 77] with isolated resources to eliminate the impact on data plane processing.

2-Phase Commit Updates. The 2-phase commit protocol for network updates by Reitblatt *et al.* [64], discussed at the beginning of §10, can also directly be integrated into P4Update. We can deploy the rules for the new flow tags by a single-layer update, upon whose completion the ingress can switch to the new tags, maintaining policy consistency.

Rule Cleanup. We can implement rule cleanup in reverse, from ingress to egress, respectively also in reverse between gateway nodes: after an update, if the old link e is different to the new one e', a cleanup packet is sent via e, informing the old parent node that no further packets will be sent

Failures in the Update Process. While P4Update prevents inconsistent updates from happening, it still represents a significant architectural deviation for SDN, where, unbeknownst to the controller, the switches coordinate with each other. However, here we can leverage that in P4Update, the ingress switch will eventually inform the controller of a successful flow update process, and we could build on this to protect against packet loss. Then, the gateway nodes would periodically monitor the arrival of UNM. If the gateway nodes do not receive packets within a specified time window, they assume that UNM loss happens during transmission. Then the child gateway node notifies controller and the controller will re-trigger the updates. For the SL-P4Update, the controller sends UIM to the egress node and UNM is re-generated from egress node. Then the UNM transmission and Alg. 1 will be executed again. For the DL-P4Update, the update is re-triggered partially and UNM only needs to be retransmitted from gateway nodes.

Destination-Based Routing. We present P4Update as a system to update flow-based routing, but P4Update can also be adapted to different routing paradigms. For example, in destination based-routing, forward edges upstream and backward edges downstream exist, whereas basic distance labeling can be used for new forwarding rules being neither. W.r.t. to network functions using, *e.g.*, segment routing, P4Update can update the segment routes in parallel.

Subsequent Dual-Layer Updates. P4Update's dual-layer update approach relies on old distances previously established by single-layer updates or initial deployment. Hence, currently, a dual-layer update needs the next update to be single-layer, before a dual-layer update can be deployed again. We can remove this restriction by enforcing that nodes use their old distances based on their last single-layer update. We give further details in Appendix §C.

Reducing the Number of Control Plane Messages. In our current implementation, the controller sends the new forwarding state to each affected switch, but the number of these messages could be reduced by larger messages. Taking it to the extreme, it suffices if the controller sends out only one large message, which is propagated upstream, leveraging a source-routing paradigm where the notifications also include the new forwarding states. However, such an approach removes any parallelism and will perform similar to single-layer P4Update in §9. Instead, to retain parallelism, the controller could adapt the above idea and send out messages to exactly those switches that may immediately notify their children, *e.g.*, only to v_7, v_4, v_2 in Fig. 1.

Removing Version Numbers. The invariant that a parent v has a smaller distance suffices to avoid loops, even without version numbers. Yet, asynchrony and delays might induce a mixture of different forwarding paths $\mathcal{P}_1, \ldots, \mathcal{P}_k$ to be deployed in the end, whereas P4Update converges to $\mathcal{P}_k$.

12 CONCLUSION

This paper presents P4Update, a framework for efficient distributed consistent network routing updates with local verification. Our approach flips the problem setting of network routing updates and leverages the asynchronous updates of multiple switches to its advantage. The control plane only schedules the new configuration and dependency once to the data plane, and each switch verifies on its own and informs its neighbor to carefully respect the consistency properties. The verification also helps to resolve inconsistent updates, and we provide the first locally verifiable update method that can update its nodes in parallel. Our evaluation on a prototype implementation shows that P4Update can improve the update time on average from 28.6% to 39.1%, in comparison to the state-of-the-art. The control plane algorithm remains scalable, in particular by dynamically generating inter-flow dependencies in the data plane.

Future Work. In general, we regard our work as a first step and believe that it opens several interesting avenues for future research. In particular, it remains to explore the verification of further consistency properties [21], *e.g.*, related to policy, such as per-packet consistency or (congestion-aware) network function traversal [4, 5, 11, 56], but also relaxed or combined notions of, *e.g.*, loop freedom [6, 22, 23]. In this context, there can also be multiple ways to update a network [26], and hence one could just specify desired performance properties (*e.g.*, for throughput [8] and latency [25]), without fixing the routes, bringing us a step closer towards self-driving networks [44, 47]. A further issue is the impact on other types of routing [15], *e.g.*, when lying or fibbing is part of the routing mechanism [12, 73], standing in contrast to verification. Another direction is an implementation on, e.g., Tofino [61] switches, but also to take considerations for (distributed) in-band control [10, 75].

Reproducibility. To facilitate reproducibility, our source code will be made available at https://p4update.lkn.ei.tum.de/.

Acknowledgements. We thank our shepherd Shir Landau Feibish and the reviewers for their valuable feedback. This work received funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 316878574 and 397973531. This work is part of a project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 647158 - FlexNets "Quantifying Flexibility for Communication Networks").

REFERENCES

[1] 2016. Internet2. https://www.internet2.edu/.

[2] 2021. numpy.random.exponential. https://numpy.org/doc/stable/reference/random/generated/numpy.random.exponential.html.

[3] Saeed Akhoondian Amiri, Szymon Dudycz, Stefan Schmid, and Sebastian Wiederrecht. 2018. Congestion-Free rerouting of flows on DAGs. In *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP) (LIPIcs, Vol. 107)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 143:1–143:13.

[4] Saeed Akhoondian Amiri, Klaus-Tycho Foerster, Riko Jacob, and Stefan Schmid. 2018. Charting the Algorithmic Complexity of Waypoint Routing. *Comput. Commun. Rev.* 48, 1 (2018), 42–48.

[5] Saeed Akhoondian Amiri, Klaus-Tycho Foerster, and Stefan Schmid. 2020. Walking Through Waypoints. *Algorithmica* 82, 7 (2020), 1784–1812.

[6] Arsany Basta, Andreas Blenk, Szymon Dudycz, Arne Ludwig, and Stefan Schmid. 2018. Efficient Loop-Free Rerouting of Multiple SDN Flows. *IEEE/ACM Trans. Netw.* 26, 2 (2018), 948–961.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[8] Sebastian Brandt, Klaus-Tycho Foerster, and Roger Wattenhofer. 2017. Augmenting flows for the consistent migration of multi-commodity single-destination flows in SDNs. *Pervasive Mob. Comput.* 36 (2017), 134–150.

[9] Sebastian Brandt, Klaus-Tycho Foerster, and Roger Wattenhofer. 2016. On consistent migration of flows in SDNs. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 1–9.

[10] Marco Canini, Iosif Salem, Liron Schiff, Elad Michael Schiller, and Stefan Schmid. 2017. A Self-Organizing Distributed and In-Band SDN Control Plane. In *ICDCS*. IEEE Computer Society, 2656–2657.

[11] Pavol Cerný, Nate Foster, Nilesh Jagnik, and Jedidiah McClurg. 2016. Optimal Consistent Network Updates in Polynomial Time. In *DISC (Lecture Notes in Computer Science, Vol. 9888)*. Springer, 114–128.

[12] Marco Chiesa, Gábor Rétvári, and Michael Schapira. 2016. Lying your way to better traffic engineering. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 391–398.

[13] The P4 Language Consortium. 2016. P4 behavioral-model. https://github.com/p4lang/behavioral-model/.

[14] The P4 Language Consortium. 2020. P4_16 Language Specification, version 1.2.1. https://p4.org/p4-spec/docs/P4-16-v1.2.1.html.

[15] Arnaud Dethise, Marco Chiesa, and Marco Canini. 2018. Prelude: Ensuring inter-domain loop-freedom in SDN-enabled networks. In *Proceedings of the 2nd Asia-Pacific Workshop on Networking (APNet)*. ACM, 50–56.

[16] Advait Dixit, Fang Hao, Sarit Mukherjee, TV Lakshman, and Ramana Kompella. 2013. Towards an elastic distributed SDN controller. In *Proceedings of the second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*. ACM, 7–12.

[17] Klaus-Tycho Foerster. 2017. On the consistent migration of unsplittable flows: Upper and lower complexity bounds. In *NCA*. IEEE, 153–156.

[18] Klaus-Tycho Foerster. 2018. On the Consistent Migration of Splittable Flows: Latency-Awareness and Complexities. In *NCA*. IEEE, 1–4.

[19] Klaus-Tycho Foerster, Thomas Luedi, Jochen Seidel, and Roger Wattenhofer. 2018. Local checkability, no strings attached: (A)cyclicity, reachability, loop free updates in SDNs. *Theoretical Computer Science* 709 (2018), 48–63.

[20] Klaus-Tycho Foerster and Stefan Schmid. 2019. Distributed Consistent Network Updates in SDNs: Local Verification for Global Guarantees. In *NCA*. IEEE, 1–4.

[21] Klaus-Tycho Foerster, Stefan Schmid, and Stefano Vissicchio. 2019. Survey of consistent Software-Defined Network updates. *IEEE Communications Surveys & Tutorials* 21, 2 (2019), 1435–1461.

[22] Klaus-Tycho Foerster and Roger Wattenhofer. 2016. The Power of Two in Consistent Network Updates: Hard Loop Freedom, Easy Flow Migration. In *ICCCN*. IEEE, 1–9.

[23] Klaus-Tycho Foerster, Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. 2018. Loop-Free route updates for Software-Defined Networks. *IEEE/ACM Transactions on Networking* 26, 1 (2018), 328–341.

[24] Klaus-Tycho Foerster, Ratul Mahajan, and Roger Wattenhofer. 2016. Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes. In *Proceedings of the 2016 IFIP Networking Conference*. IEEE, 1–9.

[25] Klaus-Tycho Foerster, Laurent Vanbever, and Roger Wattenhofer. 2019. Latency and consistent flow migration: Relax for lossless updates. In *Proceedings of the 2019 IFIP Networking Conference*. IEEE, 1–9.

[26] Rohan Gandhi, Ori Rottenstreich, and Xin Jin. 2017. Catalyst: Unlocking the Power of Choice to Speed up Network Updates. In *CoNEXT*. ACM, 276–282.

[27] Michael R Garey and David S Johnson. 1979. *Computers and intractability*. Vol. 174. Freeman San Francisco.

[28] Mika Göös and Jukka Suomela. 2016. Locally checkable proofs in distributed computing. *Theory of Computing* 12, 1 (2016), 1–33.

[29] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 357–371.

[30] David Hancock and Jacobus Van der Merwe. 2016. Hyper4: Using P4 to virtualize the programmable data plane. In *Proceedings of the 12th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 35–49.

[31] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. 2018. Network-wide heavy hitter detection with commodity switches. In *Proceedings of the 2018 ACM Symposium on SDN Research (SOSR)*. ACM, 1–7.

[32] Keqiang He, Junaid Khalid, Aaron Gember-Jacobson, Sourav Das, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. 2015. Measuring control plane latency in SDN-enabled switches. In *SOSR*. ACM, 25:1–25:6.

[33] Mu He, Alberto Martínez Alba, Arsany Basta, Andreas Blenk, and Wolfgang Kellerer. 2019. Flexibility in softwarized networks: Classifications and research challenges. *IEEE Communications Surveys & Tutorials* (2019).

[34] Mu He, Andreas Blenk, Wolfgang Kellerer, and Stefan Schmid. 2019. Toward consistent state management of adaptive programmable networks based on P4. In *Proceedings of the ACM SIGCOMM 2019 Workshop on Networking for Emerging Applications and Technologies*. ACM, 29–35.

[35] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: Fast connectivity recovery entirely in the data plane. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 161–176.

[36] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. 2018. B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined WAN. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 74–87.

[37] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving high utilization with software-driven WAN. In *Proceedings of the 2013 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 15–26.

[38] Danny Yuxing Huang, Ken Yocum, and Alex C. Snoeren. 2013. High-fidelity switch models for software-defined network emulation. In *HotSDN*. ACM, 43–48.

[39] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a globally-deployed software defined WAN. In *Proceedings of the 2013 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 3–14.

[40] Michael Jarschel, Simon Oechsner, Daniel Schlosser, Rastin Pries, Sebastian Goll, and Phuoc Tran-Gia. 2011. Modeling and performance evaluation of an OpenFlow architecture. In *Proceedings of the 2011 23rd International Teletraffic Congress (ITC)*. IEEE, 1–7.

[41] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. 2019. Fast string searching on pisa. In *Proceedings of the 2019 ACM Symposium on SDN Research (SOSR)*. ACM, 21–28.

[42] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. 2014. Dynamic scheduling of network updates. In *SIGCOMM*. ACM, 539–550.

[43] Wm Woolsey Johnson, William Edward Story, et al. 1879. Notes on the" 15" puzzle. *American Journal of Mathematics* 2, 4 (1879), 397–404.

[44] Patrick Kalmbach, Johannes Zerwas, Péter Babarczi, Andreas Blenk, Wolfgang Kellerer, and Stefan Schmid. 2018. Empowering Self-Driving Networks. In *SelfDN@SIGCOMM*. ACM, 8–14.

[45] Pravein Govindan Kannan, Raj Joshi, and Mun Choon Chan. 2019. Precise time-synchronization in the data-plane using programmable switching ASICs. In *Proceedings of the 2019 ACM Symposium on SDN Research (SOSR)*. ACM, 8–20.

[46] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the 2016 ACM Symposium on SDN Research (SOSR)*. ACM, 1–12.

[47] Wolfgang Kellerer, Patrick Kalmbach, Andreas Blenk, Arsany Basta, Martin Reisslein, and Stefan Schmid. 2019. Adaptable and Data-Driven Softwarized Networks: Review, Opportunities, and Challenges. *Proc. IEEE* 107, 4 (2019), 711–731.

[48] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. 2011. The internet topology zoo. *IEEE Journal on Selected Areas in Communications* 29, 9 (2011), 1765–1775.

[49] Amos Korman, Shay Kutten, and David Peleg. 2010. Proof labeling schemes. *Distributed Computing* 22, 4 (2010), 215–233.

[50] Maciej Kuzniar, Peter Peresíni, and Dejan Kostic. 2015. What You Need to Know About SDN Flow Tables. In *PAM (Lecture Notes in Computer Science, Vol. 8995)*. Springer, 347–359.

[51] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. 2012. Logically centralized? State distribution trade-offs in Software Defined Networks. In *Proceedings of the first workshop on Hot topics in Software Defined Networks (HotSDN)*. ACM, 1–6.

[52] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. 2013. Ensuring connectivity via data plane mechanisms. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 113–126.

[53] Sheng Liu, Theophilus A Benson, and Michael K Reiter. 2019. Efficient and safe network updates with suffix causal consistency. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM.

[54] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 101–114.

[55] Arne Ludwig, Szymon Dudycz, Matthias Rost, and Stefan Schmid. 2016. Transiently secure network updates. *ACM SIGMETRICS Performance Evaluation Review* 44, 1 (2016), 273–284.

[56] Arne Ludwig, Szymon Dudycz, Matthias Rost, and Stefan Schmid. 2018. Transiently Policy-Compliant Network Updates. *IEEE/ACM Trans. Netw.* 26, 6 (2018), 2569–2582.

[57] Ratul Mahajan and Roger Wattenhofer. 2013. On consistent updates in Software Defined Networks. In *Proceedings of the 12th ACM Workshop on Hot Topics in Networks (HotNets)*. ACM, New York, NY, USA, 20:1–20:7.

[58] Tal Mizrahi, Ori Rottenstreich, and Yoram Moses. 2017. TimeFlip: Using timestamp-based TCAM ranges to accurately schedule network updates. *IEEE/ACM Transactions on Networking* 25, 2 (2017), 849–863.

[59] Edgar Costa Molero, Stefano Vissicchio, and Laurent Vanbever. 2018. Hardware-accelerated network control planes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNets)*. ACM, 120–126.

[60] Netronome. 2017. Netronome SmartNIC. https://www.netronome.com/products/smartnic/overview/.

[61] Barefoot Networks. 2016. TOFINO: World's fastest P4-programmable Ethernet switch ASICs. https://www.barefootnetworks.com/products/brief-tofino/.

[62] Miguel Neves, Lucas Freire, Alberto Schaeffer-Filho, and Marinho Barcellos. 2018. Verification of P4 programs in feasible time using assertions. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 73–85.

[63] Thanh Dang Nguyen, Marco Chiesa, and Marco Canini. 2017. Decentralized consistent updates in SDN. In *Proceedings of the 2017 ACM Symposium on SDN Research (SOSR)*. ACM, 21–33.

[64] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for network update. In *Proceedings of the 2012 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, New York, NY, USA, 323–334.

[65] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. 2011. Consistent updates for software-defined networks: change you can believe in!. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks (HotNets)*. ACM, 7.

[66] Matthew Roughan. 2005. Simplifying the synthesis of Internet traffic matrices. *ACM SIGCOMM Computer Communication Review* 35, 5 (2005), 93–96.

[67] Stefan Schmid and Jukka Suomela. 2013. Exploiting locality in distributed SDN control. In *Proceedings of the second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*. ACM, 121–126.

[68] Roshan Sedar, Michael Borokhovich, Marco Chiesa, Gianni Antichi, and Stefan Schmid. 2018. Supporting emerging applications with low-latency failover in P4. In *Proceedings of the ACM SIGCOMM 2018 Workshop on Networking for Emerging Applications and Technologies*. ACM, 52–57.

[69] Apoorv Shukla, Seifeddine Fathalli, Thomas Zinner, Artur Hecker, and Stefan Schmid. 2020. P4Consist: Toward Consistent P4 SDNs. *IEEE J. Sel. Areas Commun.* 38, 7 (2020), 1293–1307.

[70] Apoorv Shukla, Kevin Nico Hudemann, Zsolt Vági, Lily Hügerich, Georgios Smaragdakis, Artur Hecker, Stefan Schmid, and Anja Feldmann. 2021. Fix with P6: Verifying Programmable Switches at Runtime. In *INFOCOM*. IEEE, 1–10.

[71] Apoorv Shukla, Said Jawad Saidi, Stefan Schmid, Marco Canini, Thomas Zinner, and Anja Feldmann. 2020. Toward Consistent SDNs: A Case for Network State Fuzzing. *IEEE Trans. Netw. Serv. Manag.* 17, 2 (2020), 668–681.

[72] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. 2017. Heavy-hitter detection entirely in the data plane. In *Proceedings of the 2017 ACM Symposium on SDN Research (SOSR)*. ACM, 164–176.

[73] Stefano Vissicchio, Olivier Tilmans, Laurent Vanbever, and Jennifer Rexford. 2015. Central control over distributed routing. In *Proceedings of the 2015 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 43–56.

[74] Péter Vörös, Dániel Horpácsi, Róbert Kitlei, Dániel Leskó, Máté Tejfel, and Sándor Laki. 2018. T4P4S: A target-independent compiler for protocol-independent packet processors. In *Proceedings of the 2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 1–8.

[75] Shuyuan Zhang, Sharad Malik, Sanjai Narain, and Laurent Vanbever. 2014. In-Band Update for Network Routing Policy Migration. In *ICNP*. IEEE Computer Society, 356–361.

[76] Jiaqi Zheng, Bo Li, Chen Tian, Klaus-Tycho Foerster, Stefan Schmid, Guihai Chen, Jie Wu, and Rui Li. 2019. Congestion-Free rerouting of multiple flows in timed SDNs. *IEEE Journal on Selected Areas in Communications* 37, 5 (2019), 968–981.

[77] Peng Zheng, Theophilus Benson, and Chengchen Hu. 2018. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 98–111.

[78] Wenxuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and P Brighten Godfrey. 2015. Enforcing customizable consistency properties in Software-Defined Networks. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 73–85.

[79] Noa Zilberman, Yury Audzevich, G Adam Covington, and Andrew W Moore. 2014. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE Micro* 34, 5 (2014), 32–41.

A APPENDIX: ALGORITHMS & PROOFS

A.1 DL-P4Update Coordination

Algorithm 2: DL-Verification (Data Plane) at node v

```
Input  : Highest indication UIM, notification UNM, state at $v$ (old version
         $V_o(v)$ and distance $D_o(v)$, new version $V_n(v)$ and distance $D_n(v)$,
         counter $C(v)$, last update type $T(v)$ (dual or empty/single
         otherwise))
Output : Verification state $VS$, current state at $v$
$VS = 0$; // Initialize verification state
if $T$(UIM) $\neq$ dual or $T$(UNM) $\neq$ dual then
    Switch to Alg. 1 for single-layer updates;
if $V_n$(UNM) $>$ $V_n$(UIM) then
    Wait for UIM; // Wait until new UIM arrives
else if $V_n$(UNM) $<$ $V_n$(UIM) then
    Drop UNM, inform controller; // Update outdated
else if $V_n$(UNM) $=$ $V_n$(UIM) then
    if $V_n(v) + 1 < V_n$(UNM) then
        // nodes inside segment
        if $D_n$(UIM) $= D_n$(UNM) $+ 1$ then
            $V_n(v) = V_n$(UNM); // update new version
            $D_n(v) = D_n$(UIM); // update new distance
            $V_o(v) = V_n$(UNM) $- 1$; // update old version
            $D_o(v) = D_o$(UNM); // inherit distance
            $C(v) = C$(UNM) $+ 1$; // Counter+1
            $T$ = dual; $VS = 1$;
    else if $V_n(v) + 1 = V_n$(UNM) $= V_o$(UNM) $+ 1$ then
        // gateway node (end/start of segment)
        if $D_n$(UIM) $= D_n$(UNM) $+ 1$ and $T \neq$ dual then
            if $D_n(v) > D_o$(UNM) then
                $V_n(v) = V_n$(UIM); $D_n(v) = D_n$(UIM);
                $V_o(v) = V_o$(UNM); $D_o(v) = D_o$(UNM); // Inherit
                    distance
                $C(v) = C$(UNM) $+ 1$; // Counter+1
                $T$ = dual; $VS = 1$;
    else if $V_n(v) = V_n$(UNM) and $V_o(c) = V_o$(UNM) then
        // node that was already updated and is used to pass
            old distances upstream
        if $D_n(v) = D_n$(UIM) $= D_n$(UNM) $+ 1$ then
            if $D_o(v) > D_o$(UNM) or ($D_o(v) = D_o$(UNM) and
              $C(v) > C$(UNM) ) then
                $D_o(v) = D_o$(UNM);// Inherit dist.
                $VS = 1$; $C(v) = C$(UNM) $+ 1$;
```

A.2 Congestion Freedom

We now show how to extend SL- and DL-P4Update to also account for congestion freedom. To this end, recall that congestion freedom is defined s.t. for every link, the combined flow sizes do not violate the link's capacity, and that the sizes are known for each flow. We moreover recall two (implicit) observations:

Observation 1. *Nodes only update to higher versions.*

Moreover, observe that the change of a forwarding rule must be triggered by a node's new parent:

Observation 2. *A node v may only change its forwarding to w upon or after receiving a message from w.*

The consistency property of congestion freedom is maintained, if, for every flow, there is enough reserved capacity to reach its egress node. We can hence use ideas analogous as for blackhole and loop freedom, and ensure that a node may only update if the downstream capacity is guaranteed.

Still, as congestion freedom is not just a topological property, but also relies on the capacity remaining due to other flows, we need an additional check to ensure that the capacity suffices when updating. However, when a node v obtains the information that there is enough capacity from its parent onward downstream, all that is left to do is to check the remaining capacity on the flow's new outgoing link from v.

We extend the verification procedures for P4Update by also checking that the outgoing link capacity suffices and that the flow size stays identical. For the latter, we compare it with the previous flow size and else discard it, respectively skip this check if the flow is newly deployed at this node. Regarding the capacity check, when an update to move a flow f to the outgoing link e at node v passes all checks for blackhole and loop freedom we compute the remaining capacity on e, to see if it suffices for f. This check can be done locally as v is aware of all updated flow sizes on e, and may also be limited to elephant flows. If capacity suffices, the update can be performed, and if not, it is deferred until enough capacity is available. Note that if the flow f was routed on e under the prior forwarding rules, with the same or larger size, then capacity is already allocated and similarly, the egress node does not need to check for outgoing capacity.

The update coordination is analogous to blackhole and loop-freedom, with the only exception that nodes also locally check flow size and if the outgoing link capacity suffices.

Observe that P4Update's data plane scheduler only imposes delays in the order of the flow updates, due to its implicit scheduling via high or low priorities, all (verification) checks are not altered. Hence, consistency cannot be violated by the scheduler. We start with the single-layer update and first expand Theorem 1. Now, with congestion freedom as a consistency property in mind, each switch checks that the new outgoing link has enough capacity, before updating the flow. Hence, there is enough capacity to send to the parent, which in turn has enough capacity to its parent etc. As the flow only receives data from its ingress node, there needs to be a node downstream that terminates the flow, the egress node, and as such there is sufficient capacity along the path.

Corollary 1. *Assuming the single-layer approach is used, the network will be blackhole-, loop-, and congestion-free, even under inconsistent control plane messages.*

If there is enough capacity, the flow can update eventually at every switch, and a flow cannot steal capacity from its own higher version number. If the new parent is also the old parent, the capacity check succeeds automatically.

Corollary 2. *Assuming the single-layer approach is used and there is enough capacity for the highest version updates pushed by the control plane, the flow path will eventually converge to the highest version update, if it is consistent.*

We next consider the dual-layer update and extend Theorems 3 and 4. Here the extension argument is as before, by guaranteeing that the downstream path has sufficient capacity, an ingress cannot induce link capacity violations, and moreover, a flow cannot steal its own capacity.

Name	Explanation
`new_distance`	D_n specified in $\mathcal{P}_n$
`new_version`	V_n specified in $\mathcal{P}_n$
`egress_port_updated`	Egress port in $\mathcal{P}_n$
`old_distance`	D_o specified in $\mathcal{P}_o$
`old_version`	V_o specified in $\mathcal{P}_o$
`egress_port`	Egress port in $\mathcal{P}_o$
`flow_size`	Per-flow Size
`flow_priority`	Per-flow Priority
`t`	Last update type
`counter`	Counter for hops

Table 1: Registers defined in `P4Update`.

Corollary 3. *Assuming the dual-layer approach is used, the network will be blackhole-, loop-, and congestion-free, even under inconsistent control plane messages.*

Corollary 4. *When the dual-layer approach is used and enough capacity on the new paths and all impacted nodes were last updated by a corresponding single-layer approach, the flow path will eventually converge to the flow rules pushed by a dual-layer approach, if consistent and highest version.*

B APPENDIX: DATA PLANE & REGISTERS

FRM. The ingress switch generates a FRM when a new flow f emerges. It calculates a hash value $H(f)$ based on the source-destination pair of f. The hash value is the identifier of the flow, called Flow ID. As P4 cannot generate packets from scratch, the ingress switch *clones* the first flow packet, inserts the Flow ID into the clone, sending it to the control plane.

UIM and UNM. After receiving an UIM from the control plane, a switch first verifies whether it is the egress node of a path. If yes, the switch takes the egress port information from the UIM and writes it into the corresponding entry of `egress_port_updated` (listed in Table 1) according to the flow ID. Then it generates a UNM by leveraging the ongoing packets of that flow. The packet header fields of the UNMs include the old and new distances, the old and new versions, the Flow ID, and the update type, *e.g.*, SL or DL.

When changing to a new path, a switch cannot apply the value of `egress_port_updated` immediately, only after the arrival of UNM. If the UNM arrives earlier, it needs to wait for UIM. As the P4 data plane does not natively support a timer for waiting, `P4Update` uses *packet resubmission* to check repeatedly if UIM has arrived while processing UNM. We make modifications on the BMv2 to reduce the performance overhead of *resubmission.* After UNM and UIM are present, it starts the verification algorithm. Before checking the remaining capacity for the outgoing link, it checks if there are some high priority flows which are already waiting. If the capacity is not enough, the UNM will be *resubmitted* and flows will be marked as high priority, recall §7.4. Upon obtaining positive verification results, the value of `egress_port_updated` can be deployed. We use two registers `egress_port_updated` and `egress_port` to realize the update. The value of the former is read from UIM into metadata and then written into the latter when processing UNM. The value of the `egress_port` is then read into metadata as well and fed as the input parameter of the forwarding table. After the `egress_port` has been updated, the new configuration replaces the old one, and the following data plane packets can be forwarded with the new configuration. Accordingly, the `old_distance` and `old_version` will also be updated to the corresponding value in `new_distance` and `new_version`.

UFM. After receiving the UNM, the ingress switch clones a packet to create the UFM and sends it to the control plane.

DL-`P4Update`. In contrast to the SL approach, each egress gateway has to conceptually undertake the egress node role for each segment. As gateways are on both old and new paths, they can leverage ongoing packets of that flow to generate UNMs. After ingress gateways receive the UNMs from the egress gateways, they can directly update if they are in forward segments. Otherwise, they have to wait until the loop is resolved. As explained in §3, the ingress gateways need to iteratively inherit the old distance of downstream gateways to realize loop freedom. Therefore, they can receive UNMs repeatedly and use the old distance to identify UNM origins.

C COMBINATION OF SINGLE- AND DUAL-LAYER UPDATES.

As discussed before in §11, single-layer updates are oblivious to whether the previous update was single- or dual-layer, their correctness stems from creating new downstream forwarding rules coming upstream from the egress. However, so far dual-layer updates require the nodes to be previously updated by a single-layer approach or the dual-layer approach being the first one. We can alleviate this requirement, to allow dual-layer updates to follow on dual-layer updates.

A first practical fix would be to utilize the UNM message sent by the egress switch. For all intents and purposes, it can be understood and modified to be a single-layer update that can *clean-up* any dual-layer dependencies. On the other hand, this clean-up is sequential, meaning that no new dual-layer updates can be implemented along the flow path until the clean-up packet reached the respective node. In order to retain parallelism, we want to avoid waiting for a sequential clean-up packet though. To this end, observe that the consistency of the dual-layer update approach relies on the old-distance, which uses single-layer verification ideas and relies on hooking onto nodes with smaller distance. As thus, we can modify the dual-layer update approach to keep the old (possibly decreased) distances obtained from the previous single-layer approaches for verification, and use the new distances from the new and higher version just for the decision where change the new forwarding rule to. To guarantee updating, newer versions can implement counters dependent on the version number for symmetry breaking.