Decentralized Consistent Network Updates in SDN with ez-Segway

Thanh Dang Nguyen*
University of Chicago

Marco Chiesa
Université catholique de Louvain

Marco Canini*
KAUST

arXiv:1703.02149v1 [cs.NI] 6 Mar 2017

ABSTRACT

We present *ez-Segway*, a decentralized mechanism to consistently and quickly update the network state while preventing forwarding anomalies (loops and black-holes) and avoiding link congestion. In our design, the centralized SDN controller only pre-computes information needed by the switches during the update execution. This information is distributed to the switches, which use partial knowledge and direct message passing to efficiently realize the update. This separation of concerns has the key benefit of improving update performance as the communication and computation bottlenecks at the controller are removed. Our evaluations via network emulations and large-scale simulations demonstrate the efficiency of ez-Segway, which compared to a centralized approach, improves network update times by up to 45% and 57% at the median and the 99^{th} percentile, respectively. A deployment of a system prototype in a real OpenFlow switch and an implementation in P4 demonstrate the feasibility and low overhead of implementing simple network update functionality within switches.

1. INTRODUCTION

Updating data plane state to adapt to dynamic conditions is a fundamental operation in all centrally-controlled networks. Performing updates as quickly as possible while preserving certain consistency properties (like loop, black-hole or congestion freedom) is a common challenge faced in many recent SDN systems (e.g., [12, 14, 19, 32]).

Network updates are inherently challenging because rule-update operations happen across unsynchronized devices and the consistency properties impose dependencies among operations that must be respected to avoid forwarding anomalies and worsened network performance [7,15]. Yet, performing updates as fast as possible is paramount in a variety of scenarios ranging from performance to fault tolerance to security [15,23,28] and is a crucial requirement for "five-nines" availability of carrier-grade and mobile backhaul networks [25].

Ideally, a network would have the capability to instantaneously update its network-wide state while preserving consistency. Since updates cannot be applied at the exact same instant at all switches, recent work [25] explored the possibility of leveraging clock-synchronization techniques to minimize the transition time. This requires just a *single round of communication* between the controller and switches. However, with this "all-in-one-shot" update style, packets in flight at the time of the change can violate consistency properties due to the complete lack of coordination [15] or are dropped as a result of scheduling errors due to clock synchronization imprecisions [25].

The bulk of previous work in this area [15, 17, 20, 21, 24,28,29] focused on *maintaining consistency properties*. However, all these approaches require multiple rounds of communications between the controller (which drives the update) and the switches (which behave as remote passive nodes that the controller writes state to and is notified from). This controller-driven update process has four important drawbacks:

First, because the controller is involved with the installation of every rule, *the update time is inflated by inherent delays* affecting communication between controller and switches. As a result, even with state-of-the-art approaches [15], a network update typically takes seconds to be completed (results show 99^{th} percentiles as high as 4 seconds).

Second, because scheduling updates is computationally expensive [15,24,28], *the update time is slowed down by a centralized computation.*

Third, because the controller can only react to network dynamics (*e.g.*, congestion) at control-plane timescales, *the update process cannot quickly adapt to current data plane conditions.*

Fourth, in real deployments where the controller is distributed and switches are typically sharded among controllers, network *updates require additional coordination overhead among different controllers.* For wide-area networks, this additional synchronization can add substantial latency [8].

In this paper, we present ez-Segway, a new mechanism for network updates. Our key insight is to involve the switches as active participants in achieving

*Work performed at Université catholique de Louvain.

fast and consistent updates. The controller is responsible for computing the intended network configuration, and for identifying "flow segments" (parts of an update that can be updated independently and in parallel) and dependencies among segments. The update is realized in a *decentralized* fashion by the switches, which execute a network update by scheduling update operations based on the information received by the controller and messages passed among neighboring switches. This allows every switch to update its local forwarding rules as soon as the update dependencies are met (*i.e.*, when a rule can only be installed after dependent rules are installed at other switches), without any need to coordinate with the controller.

Being decentralized, our approach differs significantly from prior work. It achieves the main benefit of clock-synchronization mechanisms, which incur only a single round of controller-to-switches communication, with the guarantees offered by coordination to avoid forwarding anomalies and congestion. To the best of our knowledge, we are the first to explore the benefits of delegating network updates' coordination and scheduling functionalities to the switches. Perhaps surprisingly, we find that the required functionality can be readily implemented in existing programmable switches (OpenFlow and P4 [2]) with low overhead.

Preventing deadlocks is an important challenge of our update mechanism. To address this problem, we develop two techniques: (i) *flow segmentation*, which allows us to update different flow segments independently of one another, thus reducing dependencies, and (ii) *splitting volume*, which divides a flow's traffic onto its old and new paths, thus preventing link congestion.

We show that our approach leads to faster network updates, reduces the number of exchanged messages in the network, and has low complexity for the scheduling computation. Compared to state-of-the-art centralized approaches (*e.g.*, Dionysus [15]) configured with the most favorable controller location in the network, ez-Segway improves network update time by up to 45% and 57% at the median and the 99^{th} percentile, respectively, and it reduces message overhead by 65%. To put these results into perspective, consider that Dionysus improved the network update time by up to a factor of 2 over the previous state-of-the-art technique, SWAN [12]. Overall, our results show significant update time reductions, with speed of light becoming the main limiting factor.

Our contributions are as follows:

- We present ez-Segway (§4), a consistent update scheduling mechanism that runs on switches, initially coordinated by a centralized SDN controller.
- We assess our prototype implementation (§5) by running a comprehensive set of emulations (§6) and simulations (§7) on various topologies and traffic patterns.
- We validate feasibility (§8) by running our system prototype on a real SDN switch and present microbenchmarks that demonstrate low computational overheads. Our prototype is available as open source at `https://github.com/thanh-nguyen-dang/ez-segway`.

2. NETWORK UPDATE PROBLEM

We start by formalizing the network update problem and the properties we are interested in. The network consists of switches $\mathbb{S}=\{s_i\}$ and directed links $\mathbb{L}=\{\ell_{i,j}\}$, in which $\ell_{i,j}$ connects s_i to s_j with a certain capacity.

Flow modeling. We use a standard model for characterizing flow traffic volumes as in [7, 12, 15]. A flow F is an aggregate of packets between an ingress switch and an egress switch. Every flow is associated with a *traffic volume* v_F. In practice, this volume could be an estimate that the controller computes by periodically gathering switch statistics [7, 15] or based on an allocation of bandwidth resources [7, 14]. The *forwarding state* of a flow consists of an exact match rule that matches all packets of the flow. As in previous work [15], we assume that flows can be split among paths by means of weighted load balancing schemes such as WCMP or OpenFlow-based approaches like Niagara [16].

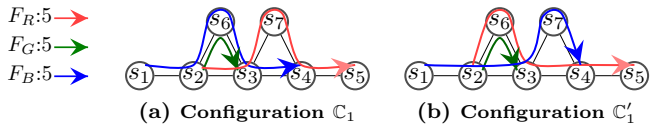
Network configurations and updates. A *network configuration* $\mathbb{C}$ is the collection of all forwarding states that determine what packets are forwarded between any pair of switches and how they are forwarded (*e.g.*, match-action flow table rules in OpenFlow). Given two network configurations $\mathbb{C}, \mathbb{C}'$, a *network update* is a process that replaces the current network configuration $\mathbb{C}$ by the target one $\mathbb{C}'$.

Properties. We focus on these three properties of network updates: (*i*) *black-hole freedom*: No packet is unintendedly dropped in the network; (*ii*) *loop-freedom*: No packet should loop in the network; (*iii*) *congestion-freedom*: No link should be loaded with a traffic greater than its capacity. These properties are the same as the ones studied in [7, 23].

Update operations. Due to link capacity limits and the inherent difficulty in synchronizing the changes at different switches, the link load during an update could get significantly higher than that before or after the update and all flows of a configuration cannot be moved at the same time. Thus, to minimize disruptions, it is necessary to decompose a network update into a set of *update operations* Π.

Intuitively, an update operation π denotes the operation necessary to move a flow F from the old to the new configuration: in the context of a single switch, this refers to the addition or deletion of F's forwarding state for that switch.

Thus, unlike per-packet consistent updates [29], we allow a flow to be routed through a mix of the old and new configuration, unless other constraints make it im-



(a) Configuration $\mathbb{C}_1$ **(b) Configuration $\mathbb{C}'_1$**

Figure 1: An example network update. Three flows F_R, F_G, F_B are to be updated in the new configuration $\mathbb{C}'_1$. Update operations must be ordered carefully in order to preserve consistency and avoid congestion.

possible (*e.g.*, a service chain of middle-boxes must be visited in reversed order in the two configurations).

Dependency graph. To prevent a violation of our properties, update operations are constrained in the order of their execution. These dependencies can be described with the help of a *dependency graph*, which will be explained in detail in §4. At any time, only the update operations whose dependencies are met in the dependency graph are possibly executed. That leads the network to transient intermediate configurations. The *update is successful* when the network is transformed from the current configuration $\mathbb{C}$ to the target configuration $\mathbb{C}'$ such that for all intermediate configurations, the aforementioned three properties are preserved.

3. OVERVIEW

Our goal is to improve the performance of updating network state while avoiding forwarding loops, blackholes, and congestion. In contrast with prior approaches – all of which use a controller to plan and coordinate the updates – we explore the question: can the network update problem be solved in a decentralized fashion wherein switches are delegated the task of implementing network updates?

3.1 Decentralizing for fast updates

Consider the example of seven switches $s_1, \cdots, s_7$ shown in Figure 1. Assume each link has 10 units of capacity and there are three flows F_R, F_G, F_B, each of size 5. This means that every link can carry at most 2 flows at the same time. We denote a path through a sequence of nodes $s1, \ldots, s_n$ by $(s1 \ldots s_n)$.

The network configuration needs to be updated from $\mathbb{C}_1$ to $\mathbb{C}'_1$. Note that we cannot simply transition to $\mathbb{C}'_1$ by updating all the switches at the same time. Since switches apply updates at different times, such a strategy can neither ensure congestion freedom nor loop- and black-hole- freedom. For example, if s_2 is updated to forward F_R on link $\ell_{2,6}$ before the forwarding state for F_R is installed at s_6, this results in a temporary blackhole. Moreover, if s_2 forwards F_R on link $\ell_{2,6}$ before F_B is moved to its new path, then $\ell_{2,6}$ becomes congested.

Ensuring that the network stays congestion free and that the consistency of forwarding state is not violated requires us to carefully plan the order of update operations across the switches. In particular, we observe that certain operations depend on other operations, leading potentially to long chains of dependencies for non trivial updates. This fact implies that a network update can be slowed down by two critical factors: (*i*) the amount of computation necessary to find an appropriate update schedule, and (*ii*) the inherent latencies affecting communication between controller and switches summed over multiple rounds of communications to respect operation dependencies.

But is it necessary to have the controller involved at every step of a network update? We find that it is possible to achieve fast and consistent updates by minimizing controller involvement and delegating to the switch the tasks of scheduling and coordinating the update process.

We leverage two main insights to achieve fast, consistent updates. Our first insight is that we can complete an update faster by using *in-band messaging* between switches instead of coordinating the update at the controller, which pays the costs of higher latency. Our second insight is that it is not always necessary to move a flow as a whole. We can complete the update faster by using *segmentation*, wherein different "segments" of a flow can be updated in parallel. For example, in Figure 1, both flows F_R and F_B can be decomposed in two independent parts: the first between s_2 and s_3, and the second between s_3 and s_4.

Distributed update execution. Before we introduce our approach in detail, we illustrate the execution of a decentralized network update for the example of Figure 1. Initially, the controller sends to every switch a message containing the current configuration $\mathbb{C}_1$, and target configuration $\mathbb{C}'_1$.[1] This information allows every switch to compute *what* forwarding state to update (by knowing which flows traverse it and their sizes) as well as *when* each state update should occur (by obeying operation dependencies while coordinating with other switches via in-band messaging).

In the example, switch s_2 infers that link $\ell_{2,3}$ has enough capacity to carry F_B and that its next hop switch, s_3, is already capable of forwarding F_B (because the flow traverses it in both the old and new configuration). Hence, s_2 updates its forwarding state so as to move F_B from path $(s_2 s_6 s_3)$ to $(s_2 s_3)$. It then notifies s_6 about the update of F_B, allowing s_6 to safely remove the forwarding state corresponding to this flow.

Once notified by s_2, s_6 infers that link $\ell_{6,3}$ now has available capacity to carry F_R. So, it installs the corresponding forwarding state and notifies s_2, which is the upstream switch on the new path of F_R. Then, s_2 infers that the new downstream switch is ready and that $\ell_{2,6}$ has enough capacity, so it moves F_R to its new path.

Similarly, s_3 updates its forwarding state for F_R to flow on $\ell_{3,4}$, notifying s_7 about the update. Meanwhile, switch s_7 infers that link $\ell_{7,4}$ has enough capacity to

[1] We revisit later what the controller sends precisely.

carry F_B; then, it installs forwarding state for F_B and notifies s_3. Then s_3 moves F_B onto $\ell_{3,7}$.

Notice that several update operations can run in parallel at multiple switches. However, whenever operations have unsatisfied dependencies, switches must coordinate. In this example, the longest dependency chain involves the three operations that must occur in sequence at s_2, s_6, and s_2 again. Therefore, the above execution accumulates the delay for the initial message from the controller to arrive at the switches plus a round-trip delay between s_2 and s_6. In contrast, if a centralized controller performed the update following the same schedule, the update time would be affected by the sum of three round-trip delays (two to s_2 and one to s_6). We aim to replace the expensive communication (in terms of latency) between the switches and the controller with simple and fast coordination messages among neighboring switches.

3.2 Dealing with deadlocks

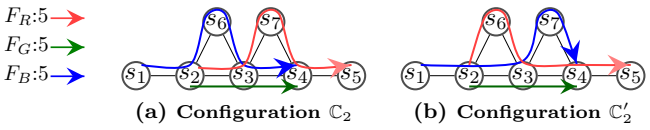
As in previous work on network update problems, it is worth asking: is it always possible to reach the target configuration while preserving all the desired consistency properties and resource constraints during the transition? Unfortunately, similarly to the centralized case [15], some updates are not feasible: that is, even if the target configuration do not violate any of the three properties, there exists no ordering of update operations to reach the target. For example, in Figure 2, moving first either F_B or F_R creates congestion.

Assume that a feasible update ordering exists. Then, will a decentralized approach always be able to find it? Unfortunately, without computing a possible schedule in advance [24], inappropriate ordering can lead to deadlocks where no further progress can be made. However, as discussed in [15], computing a feasible schedule is a computationally hard problem.

In practice, even in centralized settings, the current state-of-the-art approach, Dionysus [15], cannot entirely avoid deadlocks. In such cases, Dionysus reduces flow rates to continue an update without violating the consistency properties; however, this comes at the cost of lower network throughput.

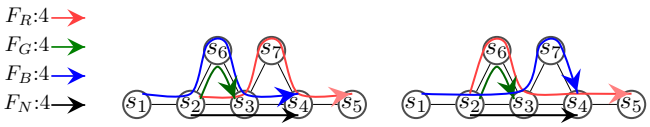
Deadlocks pose an important challenge for us as a decentralized approach is potentially more likely to enter deadlocks due to the lack of global information. To deal with deadlocks, we develop two techniques that avoid reducing network throughput (without resorting to rate-limit techniques [15] or reserving some capacities for the update operations [12]).

Our first technique is *splitting volume*, which divides a flow's traffic onto its old and new paths to resolve a deadlock. The second technique is *segmentation*, which allows to update different flow "segments" independently of one another. As we discussed, segmentation allows an update to complete faster via parallelization. In this case, it also helps to resolve certain deadlocks. Before presenting our techniques in detail (§4), we illustrate them with intuitive examples.



(a) Configuration $\mathbb{C}_2$ (b) Configuration $\mathbb{C}'_2$

Figure 2: An update with segment deadlock.



(a) Configuration $\mathbb{C}_3$ (b) Configuration $\mathbb{C}'_3$

Figure 3: An update with splittable deadlock.

Segmentation example. Consider the example in Figure 2, where every flow has size 5. This case presents a deadlock that prevents flows from being updated all at once. In particular, if we first move F_R, link $\ell_{3,4}$ becomes congested. Similarly, if we first move F_B, link $\ell_{2,3}$ is then congested.

We resolve this deadlock by segmenting these flows as: F_R=$\{s_2 \ldots s_3, s_3 \ldots s_4, s_4 \ldots s_5\}$, F_B=$\{s_1 \ldots s_2, s_2 \ldots s_3, s_3 \ldots s_4\}$. Then, switches s_2 and s_6 coordinate to first move segment $\{s_2 \ldots s_3\}$ of F_R followed by the same segment of F_B. Independently, switches s_3 and s_7 move segment $\{s_3 \ldots s_4\}$ of F_B and F_R, in this order.

Splitting volume example. Consider the example in Figure 3, where every flow has size 4. This case presents a deadlock because we cannot move F_R first without congesting $\ell_{2,6}$ or move F_B first without congesting $\ell_{2,3}$.

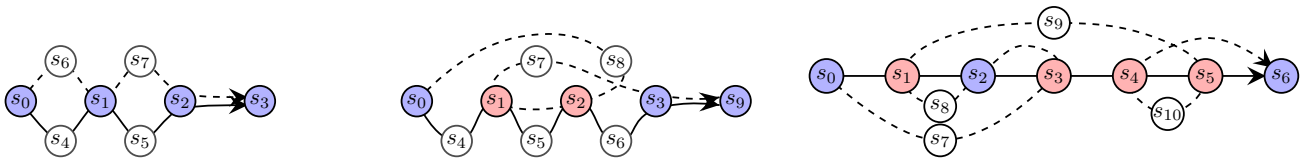
We resolve this deadlock by splitting the flows. Switch s_2 infers that $\ell_{2,3}$ has 2 units of capacity and starts moving the corresponding fraction of F_B onto that link. This movement releases sufficient capacity to move F_R to $\ell_{2,6}$ and $\ell_{6,3}$. Once F_R is moved, there is enough capacity to complete the mode of F_B.

Note that, we could even speed up part of the update by moving two units of F_R simultaneously to moving two units of F_B at the very beginning. This is what our decentralized solution would do.

4. EZ-SEGWAY

ez-Segway is a mechanism to update the forwarding state in a fast and consistent manner: it improves update completion time while preventing forwarding anomalies (*i.e.*, black-hole, loops), and avoiding the risk of link congestion.

To achieve fast updates, our design leverages a small number of crucial, yet simple, update functionalities that are performed by the switches. The centralized controller leverages its global visibility into network conditions to compute and transmit *once* to the switches the information needed to execute an update.



(a) No reversed order common switches **(b) A pair of reversed order common switches** **(c) Two pairs of reversed common switches**

Figure 4: Different cases of segmentation.

The switches then schedule and perform network update operations without interacting with the controller.

As per our problem formulation (§2), we assume the controller has knowledge of the initial and final network configurations $\mathbb{C}, \mathbb{C}'$. In practice, each network update might be determined by some application running on the controller (e.g., routing, monitoring, etc.). In our settings, the controller is centralized and we leave it for future work to consider the case of distributed controllers.

Controller responsibilities. In ez-Segway, on every update, the controller performs two tasks: (*i*) The controller identifies flow segments (*to enable parallelization*), and it identifies execution dependencies among segments (*to guarantee consistency*). These dependencies are computed using a fast heuristic that classifies segments into two categories and establishes, for each segment pair, which segment must precede another one. (*ii*) The controller computes the dependency graph (*to avoid congestion*), which encodes the dependencies among update operations and the bandwidth requirements for the update operations.

Switch responsibilities. The set of functions at switches encompasses simple message exchange among adjacent switches and a greedy selection of update operations to be executed based on the above information (*i.e.*, the dependencies among segments) provided by the controller. These functions are computationally inexpensive and easy to implement in currently available programmable switches, as we show in Section 8.

Next, we first describe in more detail our segmentation technique, which ez-Segway uses to speed up network updates while guaranteeing anomaly-free forwarding during updates. Then, we describe our scheduling mechanism based on segments of flows, which avoids congestion. Finally, we show how to cast this mechanism into a simple distributed setting that requires minimal computational power in the switches.

4.1 Flow segmentation

Our segmentation technique provides two benefits: it speeds up the update completion time of a flow to its new path and it reduces the risk of update deadlocks due to congested links by allowing a more fine-grained control of the flow update. Segment identification is *performed by the controller* when a network state update is triggered. We first introduce the idea behind segmentation and then describe our technique in details.

Update operation in the distributed approach. Consider the update problem represented in Figure 4a, where a flow F needs to be moved from the old (solid line) path $(s_0 s_4 \dots s_2 s_3)$ to the new (dashed-line) path $(s_0 s_6 \dots s_2 s_3)$. A simple approach would work as follows. A message is sent from s_3 to its predecessor on the new path (*i.e.*, s_2) for acknowledging the beginning of the flow update. Then, every switch that receives this message forwards it as soon as it installs the new rule for forwarding packets on the new path. Since the message travels in the reverse direction of the new path, the reception of such message guarantees that each switch on the downstream path consistently updated its forwarding table to the new path, thus preventing any risk of black-holes or forwarding loops anomalies. Once the first switch of the new path (*i.e.*, s_0) switches to the new path, a new message is sent from s_0 towards s_3 along the old path for acknowledging that no packets will be forwarded anymore along the old path, which can be therefore safely removed. Every switch that receives this new message removes the old forwarding entry of the old path and afterwards forwards it to its successor on the old path. We call this flow update technique from an old path to the new one Basic-Update. It is easy to observe that Basic-Update prevents forwarding loops and black-holes (proof in Appendix A).

Theorem 1. Basic-Update *is black-hole- and forwarding-loop-free.*

Speeding up a flow update with segmentation. It can be observed that the whole flow update operation could be performed faster. With segmentation the subpath $(s_0 s_4 s_1)$ of the old path can be updated with Basic-Update to $(s_0 s_6 s_1)$ while subpath $(s_1 s_5 s_2)$ of the old path is updated with Basic-Update to $(s_1 s_7 s_2)$. In fact, s_6 does not have to wait for s_1 to update its path since s_1 is always guaranteed to have a forwarding path towards s_3. We say that the pairs $(s_0 s_4 s_1, s_0 s_6 s_1)$ and $(s_1 s_5 s_2, s_1 s_7 s_2)$ are two "segments" of the flow update. Namely, a *segment* of a flow update from an old path P_o to a new path P_n is a pair of subpaths P'_o and P'_n of P_o and P_n, respectively, that start with the same first switch. We denote a segment by (P'_o, P'_n). The action of *updating a segment* consists in performing Basic-Update from P'_o to P'_n.

Dependencies among segments. In some cases, finding a set of segments that can be updated in par-

allel is not possible. For instance, in Figure 4b, we need to guarantee that s_2 updates its next-hop switch only after s_1 has updated its own next-hop; otherwise, a forwarding loop along (s_2, s_1, s_5) arises. While one option would be to update the whole flow using BASIC-UPDATE along the reversed new path, some parallelization is still possible. Indeed, we can update segments S_1=$(s_0 s_4 s_1, s_0 s_8 s_2)$ and S_2=$(s_1 s_5 s_2, s_1 s_7 s_3)$ in parallel without any risk of creating a forwarding loop. In fact, s_2, which is a critical switch, is not yet updated and the risk of forwarding oscillation is avoided. After S_2 is updated, also segment S_3=$(s_2 s_6 s_3, s_2 s_1)$ can be updated. Every time two switches appear in reversed order in the old and new path, one of the two switches has to wait until the other switch completes its update.

Anomaly-free segment update heuristic. It would be tempting to create as many segments as possible so that the update operation could benefit at most from parallelization. However, if the chain of dependencies among the segments is too long the updates may be unnecessary slowed down. In practice, computing a maximal set of segments that minimize the chain of dependencies among them is not an easy task. We rely on a heuristic that classifies segments into two categories called INLOOP and NOTINLOOP and a dependency mapping dep : INLOOP→NOTINLOOP that assigns each INLOOP segment to a NOTINLOOP segment that must be executed before its corresponding INLOOP segment to avoid a forwarding loop. This guarantees that the longest chain of dependencies is limited to at most 2. We call such technique SEGMENTED-UPDATE.

Our heuristic works as follows. It identifies the set of common switches $\mathbb{S}_C$ among the old and new path, denotes by $\mathbb{P}$ the pairs of switches in $\mathbb{S}_C$ that appear in reversed order in the two paths, and sets $\mathbb{S}_R$ to be the set of switches that appear in $\mathbb{S}_C$. It selects a subset $\mathbb{P}_R$ of $\mathbb{P}$ that has the following properties: (i) for each two pairs of switches (r, s) and (r', s') in $\mathbb{P}_R$ neither r' nor s' are contained in the subpath from r to s in the old path, (ii) every switch belonging to a pair in $\mathbb{S}_C$ is contained in at least a subpath from r to s for a pair (r, s) of $\mathbb{P}_R$, and (iii) the number of pairs in $\mathbb{P}_R$ is minimized. Intuitively, each pair (s_1, s_2) of $\mathbb{P}_R$ represents a pair of switches that may cause a forwarding loop unless s_2 is updated after s_1. The complexity for computing these pairs of switches is $O(N)$, where N is the number of switches in the network.

The segments are then created as follows. Let $\mathbb{S}^1_R$ ($\mathbb{S}^2_R$) be the set of switches that appear as the first (second) element in at least one pair of $\mathbb{P}_R$. Let $\mathbb{S}^*_C \subseteq \mathbb{S}_C$ be the set of common vertices that are contained in neither $\mathbb{S}^1_R$ nor $\mathbb{S}^2_R$. For each switch s in $\mathbb{S}^*_C \cup \mathbb{S}^1_R$ ($\mathbb{S}^2_R$), we create a NOTINLOOP (INLOOP) segment S=(P_o, P_n), where P_o (P_n) is the subpath of the old (new) path from s to the next switch r in $\mathbb{S}^*_C \cup \mathbb{S}^1_R \cup \mathbb{S}^2_R$. Morevoer, if S is an INLOOP segment, we set $dep(S)$ to be S_r, where S_r is the segment starting from r. The following theorem (proof in Appendix A) guarantees that SEGMENTED-UPDATE does not create black-holes or forwarding loops.

THEOREM 2. SEGMENTED-UPDATE *is black-hole- and forwarding-loop-free.*

As an example, consider the update depicted in Figure 4c. All the switches in common between the old and new path are colored in red or blue. Let $\mathbb{P}_R$ be $\{(s_1, s_3), (s_4, s_5)\}$ (colored in red) for which both properties (i), (ii), and (iii) hold. The resulting NOTINLOOP segments are S_1=$(s_1 s_2 s_3, s_1 s_9 s_5)$, S_2=$(s_4 s_5, s_4 s_6)$, S_3=$(s_0 s_1, s_0 s_7 s_3)$, the INLOOP segments are S_4=$(s_3 s_4, s_3 s_2 s_8 s_1)$, S_5 = $(s_5 s_6, s_5 s_{10} s_4)$, while the dependencies $dep(S_4)$ and $dep(S_5)$ are S_1 and S_2, respectively.

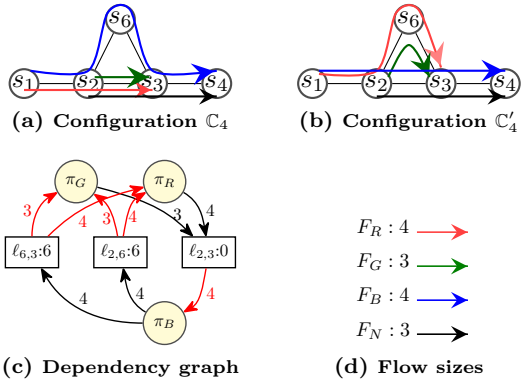
Segmentation and middleboxes. If the network policy dictates that a specific flow of data traffic must traverse one or more middleboxes, segmenting a flow must be carefully performed as the combination of fragments from the old and the new path may not traverse the set of middleboxes. In these cases, segmentation can only be applied if there are no INLOOP segments.

4.2 Scheduling updates

Performing a network update in a distributed manner requires coordination among the switches. As we already showed in Section 4.1, segments' updates that can cause forwarding loops must be executed according to a partial order of all the update operations.

We now consider the problem of performing a network update without congesting any link in the network. The main challenge of this procedure is how to avoid deadlock states in which any update operation would create congestion. We propose a heuristic that effectively reduces the risk of deadlocks, which we positively experimented in our evaluation section (Section 6). Our heuristic is correct, i.e., our heuristic finds a congestion-free migration only if the network update problem admits a congestion-free solution.

Figure 5 shows a case in which an inappropriate schedule of network operations leads to a deadlock situation. F_R and F_B have size 4 while F_G and F_N have size 3. The only operations that can be performed without congesting a link are updating either F_R or F_G. Which one to choose first? Choosing F_G will not allow to update F_B since F_G releases only 3 units of free capacity to $\ell_{2,3}$. This leads to a deadlock that can only be solved via splitting mechanisms, which increases the completion time of the update. Instead, choosing to update F_R releases 4 units of capacity to $\ell_{2,3}$, which allows us to update F_B. In turns, by updating F_B we have enough capacity to move F_G to its new path, and the update completes successfully.



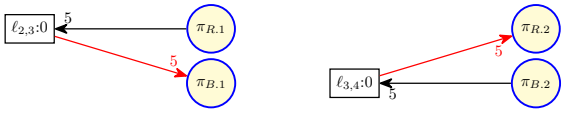
(a) Configuration $\mathbb{C}_4$ (b) Configuration $\mathbb{C}'_4$

(c) Dependency graph (d) Flow sizes

Figure 5: Choosing a correct update operation.

(a) Splittable deadlock (b) Deadlock ex. in Fig. 2

Figure 6: Dependency graph of deadlock cases.



(a) s_2 with segmentation (b) s_3 with segmentation

Figure 7: Dependency graph of deadlock cases.

In the previous example, we noted that splitting volumes helps avoiding deadlocks; however, this is less preferable than finding a sequence of update operations that do not split flows for two main reasons: (*i*) splitting flows requires more time to complete an update and (*ii*) the physical hardware may lack support for splitting flows based on arbitrary weights. In the last case, we stress the fact that ez-Segway can be configured to disable flow splitting during network updates.

Considering again the previous example, we observe that there are two types of deadlocks we must consider. If a network update execution reaches a state in which it is not possible to make any progress unless by splitting a flow, we say that there is *splittable deadlock* in the system. Otherwise, if even splitting flows cannot make any progress, we say that there is an *unsplittable deadlock* in the network.

Even from a centralized perspective, computing a congestion-free migration is not an easy task [3]. Our approach employs a mechanism to centrally precompute a *dependency graph* between links and segment updates that can be used to infer which updates are more critical than others for avoiding deadlocks.

The dependency graph. The dependency graph captures the complex set of dependencies between the network update operations and the available link capacities in the network. Given a pair of current and target configurations $\mathbb{C}, \mathbb{C}'$, any execution of network operation π (*i.e.*, updating a part of flow to its new path) requires some link capacity from the links on the new path and releases some link capacity on the old path. We formalize these dependencies in the *dependency graph*, which is a bipartite graph $\mathbb{G}(\Pi, L, E_{free}, E_{req})$, where the two subsets of vertices Π and L represent the *update operation set* and the *link set*, respectively. Each link vertex $\ell_{i,j}$ is assigned a value representing its current available capacity. Sets E_{free} and E_{req} are defined as follows:

• E_{free} is the set of directed edges from vertices in Π to vertices in L. A weighted edge with value v from π to a link $\ell_{i,j}$ represents the increase of available capacity of v units at $\ell_{i,j}$ by performing π.

• E_{req} is the set of directed edges from vertices L to vertices Π. A weighted edge with value v from link $\ell_{i,j}$ to π represents the available capacity at $\ell_{i,j}$ that is needed to execute π.

Consider the example update of Figure 5(b) and (d), where flows F_R, F_G, and F_B change paths. The corresponding dependency graph is shown in Figure 5(c). In the graph, circular nodes denote operations and rectangular nodes denote link capacities. E_{free} edges are shown in black and E_{req} edges are shown in red; the weight annotations reflect the amount of increased and requested available capacity, respectively. For instance, π_R requires 4 units of capacity from $\ell_{2,6}$ and $\ell_{6,3}$, while it increases available capacity of $\ell_{2,3}$ by the same units.

We now explore the dependency graph with respect to the two techniques that we introduced for solving deadlocks: segmentation and splitting flow volumes.

Deadlocks solvable by segmentation. Coming back to the example given in Section 3, we show its dependency graph in Figure 6b. This deadlock is unsolvable by the splitting technique. However, as discusses before, if we allow a packet to be carried in the mix of the old and the new path of the same flow, this kind of deadlock is solvable by using *segmentation*.

ez-Segway decomposes this deadlocked graph into two non-deadlocked dependency graphs as shown in Figure 7a and 7b, hence enabling the network to be updated.

Splittable deadlock. Assume that in our example we execute the update operation π_G before π_R. After F_G is moved to the new path, the dependency graph becomes the one in Figure 6a. In this case, every link has 3 units of capacity but it is impossible to continue the update. However, if we allow the traffic of flow F_R and F_B to be carried in both the old path and the new path at the same time, we can move 3 units of F_R and F_B to the new path and continue the update that enables updating the remaining part of flows. The deadlock would not be splittable if the capacity of the relevant links was zero, as shown in Figure 6b.

In the presence of a splittable deadlock, there exists

a splittable flow F_p and there is a switch s in the new segment of F_p. Switch s detects the deadlock and determines the amount of F_p's volume that can be split onto the new segment. This is taken as the minimum of the available capacity on the s's outgoing link and the necessary free capacity for the link in the dependency cycle to enable another update operation at s. An unsplittable deadlock corresponds to the state in which there is a cycle in the dependency graph where each link has zero residual capacity and it is not possible to release any capacity from the links in the cycle.

Congestion-free heuristic. Having categorized the space of possible deadlocks, we now introduce our scheduling heuristic, called ez-Schedule, whose goal is to perform a congestion-free update as fast as possible. The main goal is to avoid both unsplittable deadlocks, which can only be solved by violating congestion-freedom, and splittable deadlocks, which require more iterations to perform an update since flows are not moved in one single phase.

ez-Schedule works as follows. It receives as input an instance of the network update problem where each flow is already decomposed into segments. Each flow segmentation is updated with Segmented-Update, which means that each segment is updated directly from its old path to the new one if there is enough capacity. Hence, each segment corresponds to a network update node in the dependency graph of the input instance. Each switch assigns to every segment that contains the switch in its new path a priority level based on the following key structure in the dependency graph. An update node π in the dependency graph is *critical* at a switch s if (i) s is the first switch of the segment to be updated, and (ii) executing π frees some capacity that can directly be used to execute another update node operation that would otherwise be not executable (*i.e.*, even if every other update node operation could be possibly executed). A *critical* cycle is a cycle that contains a critical update node.

ez-Schedule assigns low priority to all the segment (i.e., a network update node) that do not belong to any cycle in the dependency graph. These update operations consume useful resources that are needed in order to avoid splittable deadlocks and, even worse, unsplittable deadlocks, which correspond to the presence of cycles with zero residual capacities in the dependency graph, as previously described. We assign medium priority to all the remaining segments that belong only to non-critical cycles, while we assign higher priority to all the updates that belong to at least one critical cycles. This guarantees that updates belonging to a critical cycle are executed as soon as possible so that the risk of incurring in a splittable or unsplittable deadlock is reduced. Each switch schedules its network operations as follows. It only considers segments that needs to be routed through its outgoing links. Among them, segment update operations with lower priority should not be executed before operations with higher priority unless a switch detects that there is enough bandwidth for executing a lower level update operations without undermining the possibility of executing higher priority updates when they will be executable. We run a simple Breadth-First-Search (BFS) tree rooted at each update operation node to determine which update operations belong to at least one critical cycle. In addition to the priority values of each flow, the updates must satisfy the constraints imposed by Segmented-Update, if there are any (i.e., there is at least one InLoop segment).

We can prove that ez-Schedule is *correct* (proof in Appendix A), i.e., as long as it there is an executable network update operation there is no congestion in the network, and that the worst case complexity for identifying a critical cycle for a given update operation is $O(|\Pi|+|L|+|\Pi|\times|L|)\simeq O(|\Pi|\times|L|)$. Consequently, for all update operations the complexity is $O(|\Pi|^2\times|L|)$.

Theorem 3. ez-Schedule *is correct for the network update problem.*

4.3 Distributed coordination

We now describe the mechanics of coordination during network updates.

First phase: the centralized computation. As described in Sect. 4.1, to avoid black-holes and forwarding loops, each segment can be updated using a Basic-Update technique unless there are InLoop segments. These dependencies are computed by the controller in the initialization phase and transmitted to each switch in order to coordinate the network update correctly.

As described in Sect. 4.2, the scheduling mechanism assigns one out of three priority levels to each segment update operation. The centralized controller is responsible for performing this computation and sending to each switch a `InstallUpdate` message that encodes the segment update operations (and their dependencies) that must be handled by the switch itself in addition to the priority level of the segment update operations.

Second phase: the distributed computation. Each switch s receives from the centralized controller the following information regarding each single segment S that traverses s in the new path: its identifier, its priority level (i.e., high, medium, or low), its amount of traffic volume, the identifier of the switch that precedes (succeeds) it along the new and old path, and whether the receiving switch is the initial or final switch in the new path of S and in the old and new path of the flow that contains S as a segment. If the segment is of type InLoop, the identifier of the segment that has to be updated before this one is also provided. Each switch in addition knows the capacity of its outgoing links and maintains memory of the amount of capacity that is

used by the flows at each moment in time.

Upon receiving this information from the controller, each switch performs the following initial actions for each segment S that it received: it installs the new path and it removes the old path. The messages exchanged by the switches for performing these three operations are described in detail the next three paragraphs. We want to stress the fact that all the functionalities executed in the switches consist of simple message exchanges and basic ranking of update operations that are computationally inexpensive and easy to implement in currently available programmable switches. Those operations are similar to those performed by MPLS to install labels in the switches [1]. Yet, MPLS does not provide any mechanism to schedule the path installation in a congestion-free manner.

Installing the new path of a segment. The installation of the new path is performed by iteratively reserving along the reversed new path the bandwidth required by the new flow. The last switch on the new path of segment S sends a `GoodToMove` message to his predecessor along the new path. The goal of this message is to acknowledge the receiver that the downstream path is set up to receive the traffic volume of S. Upon receiving a `GoodToMove` message for a segment S, a switch checks if there is enough bandwidth on the outgoing link to execute the update operation. If not, it waits that enough bandwidth will be available when some flows will be removed from the outgoing link. In that case, it checks if there are segment update operations that require the outgoing link and have higher priority than S. If not, the switch executes the update operation. Otherwise, it checks whether the residual capacity of the link minus the traffic volume of the segment is enough to execute in the future all the higher priority update operations. In that case, the switch executes the update operations. If the switch successfully performs the update operation, it updates the residual capacity of the outgoing link and it sends a `GoodToMove` message to its predecessor along the new path of S. If the switch has no predecessor along the new path of S, i.e., it is the first switch of the new path of S, it sends a `Removing` message to its successor in the old path. If the receiving switch is the last switch of an InLoop segment S', it sends a `GoodToMove` message to its predecessor on $dep(S')$.

Removing the old path of a segment. Upon receiving a `Removing` message for a segment S, if the receiving switch is not a switch in common with the new path that has not yet installed the new flow entry, it removes the entry for the old path and it forwards the message to its successor in the old path. Otherwise, it puts on hold the `Removing` message until it installs the new path. If the switch removed the old path, it updates the capacity of its outgoing links and checks whether there was a dependency between the segment that was removed and any segment that can be executed at the receiving switch. In that case, it executes these update operations according to their priorities and the residual capacity (as explained above) and propagates the `GoodToMove` that were put on hold.

4.4 Dealing with failures

While the network update process must deal with link and switch failures, we argue that doing so from within the switches simply introduces unwarranted complexity. Thus, we deliberately avoid dealing with failures in our distributed coordination update mechanism.

As with a centralized approach, if a switch or link fails during an update, a new target configuration must be computed. Recall that the ez-Segway is responsible for updating from an initial configuration to the final one but not for computing them. We believe that controller is the best place to re-compute a new global desired state and start a new update. Note that in the presence of a switch or link failure, our update process stops at some intermediate state. Once the controller is notified of the failure,it halts the updates and queries the switches to know which update operations were performed and uses this information to reconstruct the current network state and compute the new desired one.

This process can be optimized to minimize recovery delay. A natural optimization is to have switches asynchronously sending a notification to the controller upon performing an update operation, thus enabling the controller to keep closer in sync with the data plane state.

As for the messages between switches, we posit that these packets are sent with the highest priority so that they are not dropped due to congestion and that their transmission is retried if a timeout expires before an acknowledgment is received. When a message is not delivered for a maximum number of times, we behave as though the link has failed.

5. IMPLEMENTATION

We implemented an unoptimized ez-Segway prototype written as 6.5K LoC in Python. The prototype consists of a *global controller* that runs centrally as a single instance, and a *local controller* that is instantiated on every switch. The local controller is built on top of the Ryu controller [31]. The local controller, which executes our ez-Schedule algorithm, connects and manipulates the data-plane state via OpenFlow, and sends/receives UDP messages to/from other switches. Moreover, the local controller communicates with the global controller to receive the pre-computed scheduling information messages and to send back an acknowledgment once an update operation completes.

6. PROTOTYPE EVALUATION

We now evaluate the update time performance of our prototype through emulation in Mininet [11]. We

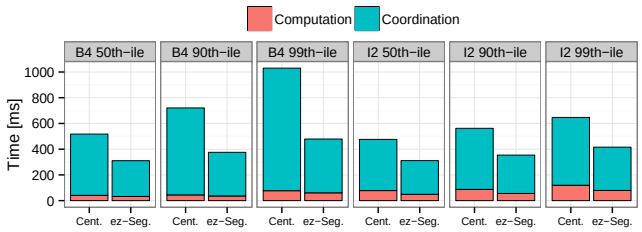


Figure 8: Update time of ez-Segway versus a Centralized approach for $B4$ and $Internet2$.

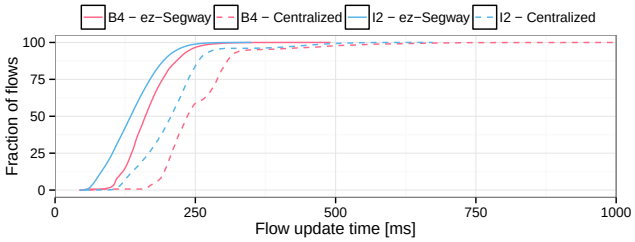


Figure 9: Flow update time of ez-Segway versus a Centralized approach for $B4$ and $Internet2$.

compare ez-Segway with a *Centralized* approach, inspired by Dionysus [15], that runs the same scheduling algorithm of ez-Segway but coordination among the switches is delegated to a central controller. To make the comparison fair, we empower Centralized with segmentation and splitting volume support, both of which do not exist in Dionysus.

Experimental setup. We run all our experiments on a dedicated server with 16 cores at 2.60 GHz with hyperthreading, 128 GB of RAM and Ubuntu Linux 14.04. The computation in the central controller is parallelized across all cores. Deadlocks are detected by means of a timeout, which we set to 150 ms.

We consider two real WAN topologies: $B4$ [14] – the globally-deployed software defined WAN at Google – and the layer 3 topology of the $Internet2$ network [13]. Without loss of generality, we assume link capacities of 1 Gbps. We place the controller at the centroid switch of the network, *i.e.*, the switch that minimizes the maximum latency towards the farthest switch. We compute the latency of a link based on the geographical distance between its endpoints and the signal propagation speed through optical fibers (*i.e.*, ∼200,000 km/s).

Similarly to the methodology in [9], we generate all flows of a network configuration by selecting non-adjacent source and destination switch pairs at random and assigning them a traffic volume generated according to the gravity model [30]. For a given source-destination pair (s, d), we compute 3 paths from s to d and equally spread the (s, d) traffic volume among these 3 paths. To compute each path, we first select a third transit node t at random and then we compute a cycle-free shortest path from s to d that traverses t. If it does not exist, we choose a different random transit node. We guarantee that the latency of the chosen paths is at most a factor of 1.5 greater than the latency of the source-destination shortest path. Starting from an empty network configuration that does not have any flow, we generate $1,000$ continuous network configurations. If the volumes of the flows in a configuration exceed the capacity of at least one link, we iteratively remove flows until the remaining ones fit within the given link capacities. The resulting network updates consist of a modification of at least (at most) 176 and 188 (200 and 276) flows in B4 and Internet2, respectively. The experiment goes through these $1,000$ sequential update configurations and we measure both the time for completing each network update and each individual flow update. The network update completion time corresponds to the slowest flow completion time among the flows that were scheduled in the update.

We break down the update time by the amount of computation performed at the central controller for the scheduling *computation* of the network updates and due to *coordination* time among the switches to perform the network update. Our coordination time is measured as the interval between the time when the first controller-to-switch is sent until when the last switch notifies the controller that it has completed all update operations. We report the 50^{th}, 90^{th}, and 99^{th} percentile update time across all the update configurations.

ez-Segway outperforms Centralized. Figure 8 shows the results for the update completion time. For B4, ez-Segway is 45% faster than Centralized in the median case and 57% faster for the 99^{th} percentile of updates, a crucial performance gain for the scenarios considered. We observed a similar trend for I2, where the completion time is 38% shorter than Centralized at the 90^{th} percentile. It should be noted that we have adopted almost worst conditions for ez-Segway in this comparison in that the central controller is ideally placed at the network centroid. Importantly, our approach is able to complete all the network updates, despite overcoming 203 and 68 splittable deadlocks in B4 and Internet2, respectively. Also, ez-Segway exchanges 35% of the messages sent by Centralized.

Controller-to-switch communication is the Centralized bottleneck. We note that the computation time at the controller represents a small fraction of the overall update time (*i.e.*, $\leq$ 20%), showing that the main bottleneck of a network update is the coordination time among the network devices, where ez-Segway significantly outperforms Centralized. As we show later with micro-benchmarks on a real switch in §8, the computation within the switch is in the order of 3−4 ms. This means that our approach is essentially limited by the propagation latency of the speed of light, which cannot be improved with better scheduling algorithms.

ez-Segway speeds up the slowest flows. Figure 9 shows the distribution of the update completion time

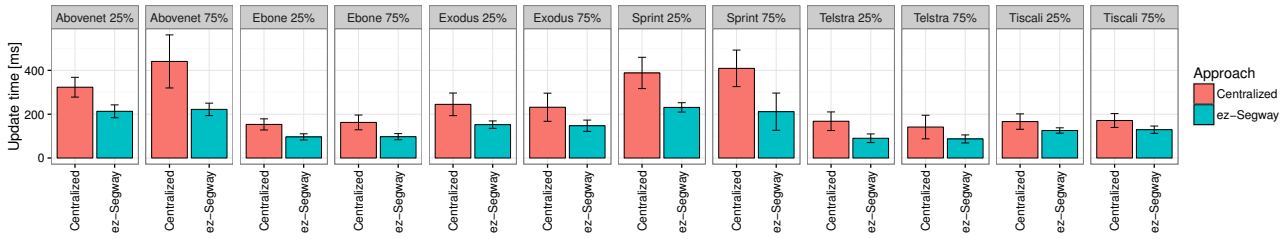


Figure 10: Update time of ez-Segway versus Centralized for various simulation settings.

Overhead	Mean	Max.
Rule	1.37(±0.57)	4
Message	3.16(±3.62)	41

Table 1: Overheads for flow splitting operations.

of individual flows across all network updates. We observe that ez-Segway (solid lines) not only consistently outperforms the centralized approach (dashed lines) in both B4 and Internet2, but it also reduces the long-tail visible in Centralized, a fundamental improvement to network's responsiveness and recovery after failures. We observed that the maximum flow update time is 456 ms in ez-Segway and 1634 ms in Centralized for B4 and 350 ms in ez-Segway and 673 ms in Centralized for Internet2. The mean flow update time is 26% slower and 51% slower with Centralized for both B4 and Internet2, respectively.

ez-Segway overcomes deadlocks with low overheads. We now measure the cost of resolving a deadlock by splitting flows in terms of forwarding table overhead and message exchange overhead in the same experimental setting used so far. Splitting flows increases the number of forwarding entries since both the new and the old flow entries must be active at the same time in order to split a flow among the two of them. Moreover, splitting flows requires coordination among switches, which leads to an overhead in the message communication. We compare ez-Segway against a version of ez-Segway that does not split flows and completes an update by congesting links. Table 1 shows that the average (maximum) number of additional forwarding entries that are installed in the switches is 1.37 (4). As for the message exchange overhead, we observe that the average (maximum) number of additional messages in the entire network is 3.16 (41). Thus, we conclude that the overheads are negligible.

7. LARGE-SCALE SIMULATIONS

We turn to simulations to study the performance of our approach on large scale Internet topologies. We compare ez-Segway against the Centralized Dionysus-like approach, which we defined in §6. We measure the total update time to install updates on 6 real topologies annotated with link delays and weights as inferred by the RocketFuel engine [22]. We set link capacities between 1∼100 Gbps, inversely proportional to weights. We place the controller at the centroid switch.

We generate flows using the same approach as in our prototype experiments. We run simulations for a number of flows varying from 500 to 1,500 and report results for 1,000 flows as we observed *qualitatively* similar behaviors. Since the *absolute* values of network update time strongly depend on the number of flows, we focus on the performance gain factor. We generate updates by simulating link failures that cause a certain percentage p of flows to be rerouted along new shortest paths. We ran experiments for 10%, 25%, 50%, and 75%; for brevity, we report results for 25% and 75%. For every setting of topology, flows, and failure rate, we generate 10 different pairs of old and new network configurations, and report the average update completion time and its standard deviation. Figure 10 shows our results, which demonstrate that ez-Segway reduces the update completion time by a factor of $1.5 - 2$. In practice, the ultimate gains of using ez-Segway will depend on specific deployment scenarios (e.g., WAN, datacenter) and might not be significant in absolute terms in some cases.

8. HARDWARE FEASIBILITY

OpenFlow. To assess the feasibility of deploying ez-Segway on real programmable switches, we tested our prototype on a Centec $V580$ switch [5], which runs a low-power 528MHz core Power PC with 1GB of RAM, wherein we execute the ez-Segway local-controller. Our code runs in Python and is not optimized. We observe that executing the local controller in the switch required minimal configuration effort as the code remained unchanged.

We first run a micro-benchmark to test the performance of the switch for running the scheduling computation. Based on Internet2 topology tests, we extract a sequence of 79 updates with 50 flows each, that we send to the switch while we measure compute time. We measure a mean update processing time of 3.4 ms with 0.5 ms standard deviation. This indicates that our approach obtains satisfactory performance despite the low-end switch CPU. Moreover, the observed CPU utilization for handling messages is negligible.

As a further thought for optimizing ez-Segway, we asked ourselves whether current switches are suitable for moving the scheduling computation from the cen-

tralized controller to the switches themselves. This would be a natural and intuitive optimization since our scheduling algorithm consists of an iterative computation of the update operations priorities for each switch. This would be beneficial to the system when the number of switches in the network is much larger than the number of cores in the central controller. We found that processing the dependency graph for the same sequence of 79 updates of the Internet2 topology with ∼200 flows in the entire network with our heuristic takes on avarage 22 ms with 6.25 ms standard deviation.

P4. To further illustrate the feasibility of realizing simple network update functionality in switches, we explored implementing ez-Segway in P4 [2]. We regard this as a thought experiment that finds a positive answer. However, the need to delegate update functions to the switch data plane is unclear since the main performance bottleneck in our results is due to coordination.

Using P4, we are able to express basic ez-Segway functionality as a set of handlers for `InstallUpdate`, `GoodToMove`, and `Removing` messages. The `InstallUpdate` handler receives a new flow update message and updates the switch state. If the switch is the last switch on the new path, it performs the flow update and sends a `GoodToMove` message to its predecessor. Upon receiving a `GoodToMove` message, a switch checks whether the corresponding update operation is allowed by ensuring that the available capacity on the new path is higher than the flow volume. In that case, the switch performs the update operation, updates the available capacity, and forwards the `GoodToMove` message to its predecessor (until the first switch is reached). When the lack of available capacity prevents an update operation, the switch simply updates some bookkeeping state to be used later. Once a `GoodToMove` message is received, the switch also attempts to perform any update operation which was pending due to unavailable capacity. Upon receiving a `Removing` message, the switch simply uninstalls the flow state and propagates the message to the downstream switch till the last one. For sake of presentation, the above does not describe priority handling.

While the above functionality is relatively simple, some of the language characteristics of P4 make it non-trivial to express it. P4 does not have iteration, recursion, nor queues data structures. Moreover, P4 switches cannot craft new messages; they can only modify fields in the header of the packet that they are currently processing [6]. To overcome these limitations of the language, we perform loop unrolling of the ez-Segway logic and adopt a header format that contains the union of all fields in all ez-Segway messages. We report the detailed P4 instructions needed to implement ez-Segway in Appendix B.

9. RELATED WORK

The network update problem has been widely studied in the recent literature [3,4,15,17,20,24–26,28,29,33,35]. These works use a centralized architecture in which an SDN controller computes the sequence of update operations and actively coordinates the switches to perform these operations. The work in [25] relies on clock-synchronization for performing fast updates, which are, however, non-consistent due to synchronization imprecisions and non-deterministic delays in installing forwarding rules [10,15,18]. In contrast, ez-Segway speeds up network updates by delegating the task of coordinating the network update to the switches. To the best of our knowledge, ez-Segway is the first work that tackles the network update problem in a decentralized manner.

In a sense, our approach follows the line of recent proposals to centrally control distributed networks like Fibbing [34], where the benefits of centralized route computation are combined with the reactiveness of distributed approaches. We discuss below the most relevant work with respect to our decentralized mechanism.

Dionysus [15] is centralized scheduling algorithm that updates flows atomically (*i.e.*, no traffic splitting). Dionysus computes a graph to encode dependencies among update operations. This dependency graph is used by the controller to perform update operations based on dynamic conditions of the switches. While ez-Segway also relies on a similar dependency graph, the controller is only responsible for constructing the graph whereas the switches use it to schedule update operations in a decentralized fashion.

Recent work [35] independently introduced flow segmentation and traffic splitting techniques similar to ours [27]. However, their work and problem formulation focuses on minimizing the risk of incurring in a deadlock in the centralized setting. In contrast, ez-Segway develops flow segmentation to increase the efficiency of network update completion time.

From an algorithmic perspective, [15] and [3] showed that, without the ability to split a flow, several formulations of the network update problem are NP-hard. The work in [3] is the only one that provides a poly-time algorithm that is correct and complete for the network update problem. However, their model allows flows to be moved on any possible intermediate path (and not only the initial and final one). Moreover, there are several limitations. First, the complexity of the algorithm is too high for practical usage. Consequently, the authors do not evaluate their algorithm. Last, this work also assumes a centralized controller that schedules the updates.

10. CONCLUSION

This paper explored delegating the responsibility of executing consistent updates to the switches. We proposed ez-Segway, a mechanism that achieves faster com-

pletion time of network updates by moving simple, yet clever coordination operations in the switches, while the more expensive computations are performed in the centralized controller. Our approach enables switches to engage as active actors in realizing updates that provably satisfy three properties: black-hole freedom, loop freedom, and congestion freedom.

In practice, our approach leads to improved update times, which we quantified via emulation and simulation on a range of network topologies and traffic patterns. Our results show that ez-Segway improves network update times by up to 45% and 57% at the median and the 99^{th} percentile, respectively. We also deployed our approach on a real OpenFlow switch to demonstrate the feasibility and low computational overhead, and implemented the switch functionality in P4.

Acknowledgements. We are thankful to Xiao Chen, Paolo Costa, Huynh Tu Dang, Petr Kuznetsov, Ratul Mahajan, Jennifer Rexford, Robert Soulé, and Stefano Vissicchio for their helpful feedback on earlier drafts of this paper. This research is (in part) supported by European Union's Horizon 2020 research and innovation programme under the ENDEAVOUR project (grant agreement 644960).

11. REFERENCES

[1] R. 3209. RSVP-TE: Extensions to RSVP for LSP Tunnels.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3), July 2014.

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

[4] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. A Distributed and Robust SDN Control Plane for Transactional Network Updates. In *INFOCOM*, 2015.

[5] Centec Networks. GoldenGate 580 Series. http://www.centecnetworks.com/en/SolutionList.asp?ID=93.

[6] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos Made Switch-y. *SIGCOMM Comput. Commun. Rev.*, 46(2), Apr 2016.

[7] K. Foerster, S. Schmid, and S. Vissicchio. Survey of Consistent Network Updates. *CoRR*, abs/1609.02305, 2016.

[8] M. Gerola, F. Lucrezia, M. Santuari, E. Salvadori, P. L. Ventre, S. Salsano, and M. Campanella. ICONA: a Peer-to-Peer Approach for Software Defined Wide Area Networks using ONOS. In *EWSDN*, 2016.

[9] J. He, M. Suchara, M. Bresler, J. Rexford, and M. Chiang. Rethinking Internet Traffic Management: From Multiple Decompositions to a Practical Protocol. In *CoNEXT*, 2007.

[10] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M. Thottan. Measuring Control Plane Latency in SDN-Enabled Switches. In *SOSR*, 2015.

[11] B. Heller, N. Handigol, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible Network Experiments using Container Based Emulation. In *CoNEXT*, 2012.

[12] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving High Utilization with Software-Driven WAN. In *SIGCOMM*, 2013.

[13] Internet2. Internet2 Network Infrastructure Topology. https://www.internet2.edu/media/medialibrary/2015/08/18/Internet2-Network-Infrastructure-Topology-201508.pdf.

[14] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.

[15] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic Scheduling of Network Updates. In *SIGCOMM*, 2014.

[16] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford. Efficient Traffic Splitting on Commodity Switches. In *CoNEXT*, 2015.

[17] N. P. Katta, J. Rexford, and D. Walker. Incremental Consistent Updates. In *HotSDN*, 2013.

[18] M. Kuźniar, P. Perešíni, and D. Kostić. What You Need to Know About SDN Flow Tables. In *PAM*, 2015.

[19] D. Levin, M. Canini, S. Schmid, F. Schaffert, and A. Feldmann. Panopticon: Reaping the Benefits of Incremental SDN Deployment in Enterprise Networks. In *USENIX ATC*, 2014.

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

[21] A. Ludwig, J. Marcinkowski, and S. Schmid. Scheduling Loop-free Network Updates: It's Good to Relax! In *PODC*, 2015.

[22] R. Mahajan, N. T. Spring, D. Wetherall, and T. E. Anderson. Inferring Link Weights using End-to-End Measurements. In *IMW*, 2012.

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

[24] J. McClurg, H. Hojjat, P. Černý, and N. Foster. Efficient Synthesis of Network Updates. In *PLDI*, 2015.

[25] T. Mizrahi and Y. Moses. Software Defined Networks: It's About Time. In *INFOCOM*, 2016.

[26] T. Mizrahi, O. Rottenstreich, and Y. Moses. TimeFlip: Scheduling Network Updates with Timestamp-based TCAM Ranges. In *INFOCOM*, 2015.

[27] T. D. Nguyen, M. Chiesa, and M. Canini. Towards Decentralized Fast Consistent Updates. In *ANRW*, 2016.

[28] P. Perešíni, M. Kuźniar, M. Canini, and D. Kostić. ESPRES: Transparent SDN Update Scheduling. In *HotSDN*, 2014.

[29] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.

[30] M. Roughan. Simplifying the Synthesis of Internet Traffic Matrices. *SIGCOMM Comput. Commun. Rev.*, 35(5):93–96, Oct. 2005.

[31] Ryu SDN Framework. http://osrg.github.io/ryu/.

[32] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A Language for Provisioning Network Resources. In

CoNEXT, 2014.
[33] S. Vissicchio and L. Cittadini. FLIP the (Flow) Table: Fast LIghtweight Policy-preserving SDN Updates. In *INFOCOM*, 2016.
[34] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford. Central Control Over Distributed Routing. In *SIGCOMM*, 2015.
[35] W. Wang, W. He, J. Su, and Y. Chen. Cupid: Congestion-free Consistent Data Plane Update In Software Defined Networks. In *INFOCOM*, 2016.

APPENDIX

A. CORRECTNESS

We first remind the three properties that ez-Segway aim to preserve during a network update:

- *black-hole freedom*: For any flow, no packet is unintendedly dropped in the network.
- *loop-freedom*: No packet should loop in the network.
- *congestion-freedom*: No link has to carry a traffic greater than its capacity.

Notation and terminology. We first introduce some useful notation and terminology. Let $u_0, \ldots, u_k$ be the sequence of k update operations executed during a network update and et $t_0, \ldots, t_k$ be the sequence of time instants where these operations are performed, with $t_i \leq t_j$ for any $i < j$. Since each switch performs a single update operations at a time, w.l.o.g., we assume that at each time instant only one operation is performed. We assume that, at time t_0, the controller sends the precomputed information to all the switches. Let F be a flow from a switch s_i to s_f that is being updated from an old path $P_{old} = (s_o^1 \ldots s_o^n)$ to a new path $P_{new} = (s_n^1 \ldots s_n^m)$. For each switch $s \in \mathbb{S}$, let $next(F, s, t_j)$ be the next-hop of F according to the rules installed at s at time t_j, where $next(F, s, t_j) = \epsilon$ if there is no rule installed for that specific flow. Let $path\_forwarding(F, s, t_j)$ be the (possibly infinite) sequence of switches that is traversed by a packet belonging to F that is received by switch s at time t_j. Observe that the incoming port of a packet is not relevant in ez-Segway.

Let $prec_{old}(s)$ ($prec_{new}(s)$) be the predecessor vertex of s in P_{old} (P_{new}) and $succ_{old}(s)$ ($succ_{new}(s)$) be the successor vertex of s in P_{old} (P_{new}). Let $prec^*_{old}(s)$ ($prec^*_{new}(s)$) be the set of switches between s_i and s in P_{old} (P_{new}) and let $succ^*_{old}(s)$ ($succ^*_{new}(s)$) be the set of switches between s_f and s.

A.1 Basic-Migration Correctness

We first prove that the Basic-Update operation is guaranteed to complete in the absence of congestion. Let F be any flow that has to be updated from P_{old} to P_{new}.

Lemma 1. Basic-Update *guarantees that every switch on the new path eventually receives a* `GoodToMove` *message.*

Proof. Suppose, by contradiction, that the statement is not true, i.e., there exists a switch on the new path that does not receive a `GoodToMove` message. Let s_n^h be the closest switch to s_f on the new path that does not receive a `GoodToMove` message. This means that s_n^{h+1} received a `GoodToMove` message but it did not send it to s_n^h. This is a contradiction since we assumed that it is not possible to congest a link during the update. Hence, the statement of the lemma is true. □

Lemma 2. Basic-Update *guarantees that every switch on the old path eventually receives a* `Removing` *message.*

Proof. By Lemma 1, s_i is guaranteed to receive a `GoodToMove` message. When s_1 receives such messages, it switches its forwarding path to the new one, it removes the old path entry, and it sends a `Removing` message to its successor of the old path. Suppose, by contradiction, that the statement is not true, i.e., there exists a switch of the old path that does not receive a `Removing` message. Let s_o^h be the closest switch to s_i on the old path that does not receive a `Removing` message. This means that s_n^{h-1} received a `Removing` message but it did not send it to s_n^h, which is a contradiction. Hence, the statement of the lemma is true. □

Lemma 3. Basic-Update *guarantees that any packet routed from a vertex that received a* `GoodToMove` *reaches s_f in a finite number of steps.*

Proof. We prove by induction on $t_0, \ldots, t_k$ that for each switch s_n^h of the new path that received a `GoodToMove` message at time $t_j > t_0$ we have that $next(s_n^h, t_k) = s_n^{h+1}$ holds, with $t_k \geq t_j$. That is, $path\_forwarding(s, t_k) = (s_n^h \ldots s_n^m)$. At time t_0, the statement trivially holds since none of the switches received a `GoodToMove`. At time $t_j > t_0$, if s_n^h received a `GoodToMove` message at any time before t_j, then by induction hypothesis $next(s_n^h, t_j) = s_n^{h+1}$ holds. If s_n^h receives a `GoodToMove` at t_j, it installs the new forwarding rules $next(s_n^h, t_j) = s_n^{h+1}$. Hence, the statement of the lemma holds. □

Lemma 4. Basic-Update *is black-hole-free.*

Proof. A black-hole is created if a switch removes the old path while there are some packets that still needs to be routed through that old path. Since the `Removing` is sent once s_i switches to the new path, any packet sent after that message is create is forwarded on the new path. Hence, this guarantees that a packet is never dropped. □

Theorem 1. Basic-Update *is black-hole- and forwarding-loop-free.*

PROOF. We first show that BASIC-UPDATE prevents forwarding loops. Consider a packet that is routed along the old path. Two cases are possible: it either reaches s_f along it or it reaches a switch that installed the new path forwarding entry. In the latter case, by Lemma 3, it is guaranteed to reach s_f.

By Lemma 4, we have that BASIC-UPDATE guarantees that black-holes are preventing.

We now prove that the update operation terminates, we first need to show that s_i eventually switches to the new path. By Lemma 1, s_i is guaranteed to receive the `GoodToMove` message and therefore it will switch to the new path. We then need to prove that the old path is eventually removed from the network. By Lemma 2, each switch of the old path is guaranteed to receive the `Removing` message and, in turn, to remove the old path entry.

Hence, the statement of the theorem is true. □

A.2 Segmented-Migration Correctness

Heuristic reminder. Our heuristic works as follows. It identifies the set of common switches $\mathbb{S}_C$ among the old and new path, denotes by $\mathbb{P}$ the pairs of switches in $\mathbb{S}_C$ that appear in reversed order in the two paths, and sets $\mathbb{S}_R$ to be the set of switches that appear in $\mathbb{S}_C$. It selects a subset $\mathbb{P}_R$ of $\mathbb{P}$ that has the following properties: (i) for each two pairs of switches (r,s) and (r',s') in $\mathbb{P}_R$ neither r' nor s' are contained in the subpath from r to s in the old path, (ii) every switch belonging to a pair in $\mathbb{S}_C$ is contained in at least a subpath from r to s for a pair (r,s) of $\mathbb{P}_R$, and (iii) the number of pairs in $\mathbb{P}_R$ is minimized. Intuitively, each pair (s_1,s_2) of $\mathbb{P}_R$ represents a pair of switches that may cause a forwarding loop unless s_2 is updated after s_1. The complexity for computing these pairs of switches is $O(N)$, where N is the number of switches in the network.

The segments are then created as follows. Let $\mathbb{S}^1_R$ ($\mathbb{S}^2_R$) be the set of switches that appear as the first (second) element in at least one pair of $\mathbb{P}_R$. Let $\mathbb{S}^*_C \subseteq \mathbb{S}_C$ be the set of common vertices that are contained in neither $\mathbb{S}^1_R$ nor $\mathbb{S}^2_R$. For each switch s in $\mathbb{S}^*_C \cup \mathbb{S}^1_R$ ($\mathbb{S}^2_R$), we create a NOTINLOOP (INLOOP) segment $S{=}(P_o,P_n)$, where P_o (P_n) is the subpath of the old (new) path from s to the next switch r in $\mathbb{S}^*_C \cup \mathbb{S}^1_R \cup \mathbb{S}^2_R$. Morevoer, if S is an INLOOP segment, we set $dep(S)$ to be S_r, where S_r is the segment starting from r.

For each pair $(r,s) \in \mathbb{P}_R$, let $\mathbb{S}_{(r,s)}$ be the set of switches contained in the subpath from r to s of P_{old}, endpoints included. We order the pairs in $\mathbb{P}_R$ as follows. We have that $(r,s) < (r',s')$ if r is closer to s_i than r' on the old path. Recall that the pairs in $\mathbb{P}_R$ are chosen in such a way that the common switches that connects each pair of switches do not overlap each other.

LEMMA 5. *For any possible execution of* SEGMENTED-UPDATE*, for any time $t_j > t_0$, for any pair $(r,s) \in \mathbb{P}_R$, for any switch $u \in \mathbb{S}_{(r,s)}$, $path\_forwarding(r,t_j)$ does not traverse any switch in $(prec^*_{old}(r) \cup prec^*_{old}(r)) \setminus \mathbb{S}_{(r,s)}$.*

PROOF. Suppose, by contradiction, that the statement of the lemma is not true, *i.e.*, there exists an execution of SEGMENTED-UPDATE such that at a certain time instant $t_j > t_o$, there exists a pair $(r,s) \in \mathbb{P}_R$ such that for a switch $u \in \mathbb{S}_{(r,s)}$ we have that $path\_forwarding(r,t_j)$ traverses a switch w in $(prec^*_{old}(r) \cup prec^*_{old}(r)) \setminus \mathbb{S}_{(r,s)}$. Let w be the closer vertex to u on $path\_forwarding(r,t_j)$. Let $(v_1 \ldots, v_n)$ be the subpath of $path\_forwarding(u,t_j)$ from u to w. Since w is a closer to s_i than u and routing along the old path cannot get closer to s_i, $path\_forwarding(r,t_j)$ must contain a subpath P^* that connects a switch x in $\mathbb{S}_{(r,s)}$ to w. This means that x and w appear in reverse order in the new and old path. Let (r',s') be a pair such that $w \in \mathbb{S}_{(r',s')}$. We can then create a new pair (r',s) that spans at least both $\mathbb{S}_{(r,s)}$ and $\mathbb{S}_{(r',s')}$, which is a contradiction since, by Property (iii) of the set of pairs $\mathbb{P}_R$, the cardinality of $\mathbb{P}_R$ is minimal. Hence, the statment of the lemma is true. □

Lemma 5 guarantees that if a forwarding loop exists, it must be confined within the two segments created by the switches of a single pair in $\mathbb{P}_R$. We now prove that the dependencies among these segments prevent any forwarding loop.

LEMMA 6. *For any possible execution of* SEGMENTED-UPDATE*, for any time $t_j > t_0$, for any pair $(r,s) \in \mathbb{P}_R$, for any switch $u \in \mathbb{S}_{(r,s)}$, $path\_forwarding(r,t_j)$ is finite.*

PROOF. By Lemma 5, a forwarding loop can only traverse switches that belong to the old path of the NOTINLOOP segment S_r of a switch r and the old path of the INLOOP segment S_s of a switch s for a pair $(r,s) \in \mathbb{P}_R$. However, s only switches to its new path when it receives the `Removing` message of S_r because $dep(S_s) = S_r$. Since Theorem 1 guarantees that BASIC-UPDATE is forwarding-loop-free, a forwarding loop cannot happen and the statement of the theorem is true. □

LEMMA 7. SEGMENTED-UPDATE *is black-hole free.*

PROOF. Since SEGMENTED-UPDATE is an execution of independent BASIC-UPDATE, by Lemma 4, we have that a packet is never dropped during an update. We only need to check that the last switches of each segment do not drop a packet. By construction of the segments, each of these switches is a common switch between the old and the new path, thus they always have either the old or new forwarding entry installed, which proves the statement of the theorem. □

Theorem 2. SEGMENTED-UPDATE *is black-hole- and forwarding-loop-free.*

PROOF. It easily follows from Lemma 6 and 7. □

A.3 Congestion-freedom Correctness

LEMMA 8. EZ-SCHEDULE *is correct for the network update problem.*

PROOF. Since each switch moves to the new path only if its outgoing capacity is at least the volume of the flow that has to be routed through the new path, it immediately follow that the EZ-SCHEDULE heursistic is correct. □

B. IMPLEMENTATION IN P4

In this section, we illustrate in detail our implementation of ez-Segway in P4. We remind the reader that a P4 program consists of a "control" part, a set of "tables", and a set of "actions". The control part is responsible for executing the logic of the packet processing opertations. A table consists of a set of entries that perform an action if the processed packet header matches some conditions. Finally, an action consists of a set of read/write operations performed on the packet and on the internal state of the switch.

Figure 11 shows the definition in P4 of all the header types used in the messages exchanged in ez-Segway, *i.e.*, `InstallUpdate`, `GoodToMove`, and `Removing`. Namely, since P4 allows the programmer to define a single header type, we define `inst_t` so as to contain the union of all the needed fields. Observe that we define a `msg_type` field so that a switch can distinguish among the different types of messages. For convenience, we also define two additional `header_type`, called `cap_t` and `flow_id_t`, that are simply used as structured variable types and not as packet headers. These two variables are instantiated with the `metadata` instruction as the `cap_meta` and `id_meta`, respectively. We remind the reader that `metadata` state is destroyed as soon as the processed packet leaves the switch. To maintain state within a switch, we use registers, which are arrays of bits. We use multiple registers – one for each state variable – because P4 does not allow to create a register of composite data types. Each `register` instruction is instantiated FLOW_COUNT or LINK_COUNT times, where those constants denote the maximum number of flows and links at a switch, respectively. We remind the reader that only registers can be accessed from the "action" part of the P4 program, which we describe later in this section.

In our P4 program, ez-Segway packets are processed by a set of *control* functions, which form the so-called *control flow*. In Figure 12, we describe all the control functions needed in ez-Segway. When a packet

```
header_type inst_t{
  fields{
    msg_type: 2; /* 1: InstallUpdate,
                    2: GoodToMove,
                    3: Removing */
    id: 16;
    vol: 4;
    old_lnk: 8;
    new_lnk: 8;
    pre_lnk: 8;
    is_end_sw: 1;
    version: 8;
  }
}

header inst_t inst;
metadata inst_t inst_meta;

header_type cap_t{
  fields{
    old_cap: 16;
    new_cap: 16;
  }
}
metadata cap_t cap_meta;

header_type flow_id_t{
  fields{
    id: 8;
    gtm_recv: 1;
    done: 1;
  }
}
metadata flow_id_t id_meta;

register vols_register{
  width: 16; instance_count : FLOW_COUNT;
}
register out_lnks_register{
  width: 16; instance_count : FLOW_COUNT;
}
register pre_lnks_register{
  width: 16; instance_count : FLOW_COUNT;
}
register old_lnks_register{
  width: 16; instance_count : FLOW_COUNT;
}
register new_lnks_register{
  width: 16; instance_count : FLOW_COUNT;
}
register old_pre_register{
  width: 16; instance_count : FLOW_COUNT;
}
register is_end_sws_register{
  width: 1; instance_count : FLOW_COUNT;
}
register is_init_sws_register{
  width: 1; instance_count : FLOW_COUNT;
}
register versions_register{
  width: 8; instance_count : FLOW_COUNT;
}
register gtm_recvs_register{
  width: 1; instance_count : FLOW_COUNT;
}
register dones_register{
  width: 1; instance_count : FLOW_COUNT;
}
register caps_register{
  width: 16; instance_count : LINK_COUNT;
}
```

Figure 11: Header type, metadata and registers in P4.

enters a switch, the `ingress` control function is the first being called. We use this control function to distinguish among the different message types. The three controls *ctrl_inst*, *ctrl_gtm*, and *ctrl_rm* are used to process `InstallUpdate`, `GoodToMove` and `Removing`, respectively. The control function *ctrl_move_flow* (*ctrl_rm_flow*) is used in *ctrl_gtm* (*ctrl_rm*) to move a flow to its new path (to remove the flow state from its old path). We remind the reader that logic statements can only be used in the control flow program and they are not allowed to be used in the rest of the program. The `apply` instruction is used to redirect the execution to the table part of the P4 program.

We now show in Figure 13) the different table functions that are called in the control part of the P4 program. In our program, each `table` statement simply consists of a series of actions that are applied to the processed packet. According to *P4*'s convention, every table must be called only once in the whole program.

Finally, Figure 14 shows all the action functions that are used to process a packet. In P4, each action consists of a sequence of read/write operations from/to the registers. For instance, the `add_flow` action performs the following three actions: i) it updates the new outgoing link of the processed flow, ii) it registers that a flow update operation has been performed, and iii) it updates the residual bandwidth capacity of the new outgoing link.

P4 does not allow the flow execution to call the same `table` instruction multiple times. As such, we generated our code in such a way that the preprocessor will generate a table for each specific flow id, as shown in Figure 15.

```
control ingress{
  if (valid(inst)){
    if (inst.msg_type==1){
      ctrl_inst();
    }
    else if (inst.msg_type==2){
      ctrl_gtm();
    }
    else if (inst.msg_type==3){
      ctrl_rem();
    }
  }
}

control ctrl_inst{
  apply(save_inst);
  if (inst.is_end_sw == 1){
    apply(read_inst);
    apply(send_gtm_inst);
  }
}

control ctrl_rm{
  if (inst_meta.version==inst.version){
    ctrl_rm_flow();
  }
}

control ctrl_gtm{
  apply(tbl_read_inst_gtm);
  if (inst_meta.version==inst.version){
    apply(tbl_wrt_gtm_recv);
    apply(tbl_read_cap);
    if (cap_meta.new_cap > inst.vol){
      ctrl_move_flow();
      apply(tbl_read_inst_id_0);
      if (id_meta.gtm_recv==1 and
          cap_meta.new_cap > inst_meta.vol){
        ctrl_move_flow_id_0();
      }
      [...]
      apply(tbl_read_inst_id_N);
      if (id_meta.gtm_recv==1 and
          cap_meta.new_cap > inst_meta.vol){
        ctrl_move_flow_id_N();
      }
    }
  }
}

control ctrl_move_flow{
  apply(tbl_move_flow);
  if (inst.pre_lnk!=-1){
    apply(tbl_send_gtm);
  }
  if (inst.old_link!=-1){
    apply(tbl_rm_flow_fr_add);
    apply(tbl_send_rm_fr_add);
  }
}

control ctrl_rm_flow{
  apply(tbl_rm_flow);
  if (inst.old_lnk!=-1){
    apply(tbl_send_rm);
  }
}
```

Figure 12: Controls in P4.

```
table tbl_wrt_gtm_recv{
  actions{ write_gtm_recv; }
}

table tbl_save_inst{
  actions{ save_inst; }
}

table tbl_send_gtm{
  actions{ send_gtm; }
}

table tbl_send_rm{
  actions{ send_rm; }
}

table tbl_send_rm_fr_add{
  actions{ send_rm; }
}

table tbl_send_gtm_inst{
  actions{ update_id_meta; send_gtm; }
}

table tbl_rem{
  actions{ send_rem; }
}

table tbl_read_inst{
  actions{ read_inst; }
}

table tbl_read_inst_gtm{
  actions{ read_inst; }
}

table tbl_read_cap{
  actions{ read_cap; }
}

table tbl_move_flow{
  actions{ add_flow; }
}

table tbl_rm_flow{
  actions{ remove_flow; }
}

table tbl_rm_flow_fr_add{
  actions{ remove_flow; }
}

table tbl_save_gtm_recv{
  actions{write_gtm_recv; }
}
```

Figure 13: All tables in P4.

```
action save_inst(){
 vols_reg[inst.id]=inst.vol;
 old_lnks_reg[inst.id]=inst.old_lnk;
 new_lnks_reg[inst.id]=inst.new_lnk;
 is_end_sws_reg[inst.id]=inst.is_end_sw;
 pre_lnks_reg[inst.id]=inst.pre_lnk;
 versions_reg[inst.id]=inst.version;
 gtm_recvs_reg[inst.id]=0;
 dones_reg[inst.id]=0;
}

action forward(port){
 standard_metadata.egress_spec=port;
}

action send_gtm(){
 inst.msg_type=2;
 standard_metadata.egress_spec=
     inst_meta.pre_lnk;
}

action send_rm(){
 inst.msg_type=3;
 standard_metadata.egress_spec=
     inst_meta.pre_lnk;
}

action update_id_meta(){
 id_meta.id=inst.id;
}

action read_inst(){
 inst_meta.vol=vols_reg[id_meta.id];
 inst_meta.old_lnk=old_lnks_reg[id_meta.id];
 inst_meta.new_lnk=new_lnks_reg[id_meta.id];
 inst_meta.is_end_sw=is_end_sws_reg[id_meta.id];
 inst_meta.pre_lnk=pre_lnks_reg[id_meta.id];
 inst_meta.version=versions_reg[id_meta.id];
 id_meta.gtm_recv=gtm_recvs_reg[id_meta.id];
 id_meta.done=dones_reg[id_meta.id];
}

action write_gtm_recv(){
 gtm_recvs_reg[id_meta.id]=1;
}

action read_cap(){
 cap_meta.old_cap=caps_reg[inst.old_lnk];
 cap_meta.new_cap=caps_reg[inst.new_lnk];
}

action add_flow(){
 out_lnks_reg[id_meta.id]=inst_meta.new_lnk;
 caps_reg[inst_meta.new_lnk]=cap_meta.new_cap
 - inst_meta.vol;
 dones_reg[id_meta.id]=1;
}

action remove_flow(){
 out_lnks_reg[id_meta.id]=-1;
 caps_reg[inst_meta.old_lnk]=cap_meta.old_cap
 +inst_meta.vol;
 dones_reg[id_meta.id]=1;
}
```

Figure 14: Actions in P4.

```
#define tbl_send_gtm_id(x)\
table tbl_send_gtm_id_ ## x{\
  actions{ send_gtm; }\
}

#define tbl_read_inst_id(x)\
table tbl_read_inst_id_ ## x}\
  actions{ update_id_meta; read_inst; }\
}

#define table tbl_move_flow_id(x)\
table tbl_move_flow_id_ ## x{\
  actions{ move_flow; }\
}

#define ctrl_move_flow_id(x)\
control ctrl_move_flow_id_ ## x{\
  apply(tbl_move_flow_id_ ## x);\
  if (inst.pre_lnk == -1){\
    apply(tbl_send_gtm_id_ ## x);\
  }\
}
```

Figure 15: Preprocessing code.