Towards Decentralized Fast Consistent Updates

Thanh Dang Nguyen Marco Chiesa Marco Canini
Université catholique de Louvain

ABSTRACT

Updating data plane state to adapt to dynamic conditions is a fundamental network control operation. Software-Defined Networking (SDN) offers abstractions for updating network state while preserving consistency properties. However, realizing these abstractions in a purely centralized fashion is inefficient, due to the inherent delays between switches and the SDN controller, we argue for delegating the responsibility of coordinated updates to the switches. To make our case, we propose *ez-Segway*, a mechanism that enables decentralized network updates while preventing forwarding anomalies and avoiding link congestion. In our architecture, the controller is only responsible for computing the intended network configuration. This information is distributed to the switches, which use partial knowledge and direct message passing to efficiently schedule and implement 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 extensive simulations show update speedups up to 2x.

CCS Concepts

•**Networks** → **Network dynamics; Network protocol design; Network manageability;**

Keywords

Software-defined networking; decentralized network update

1. INTRODUCTION

Many recent SDN systems have demonstrated the value of centrally controlling networks [6, 11, 13, 14, 19, 31]. We observe, like others before us [7, 16, 20, 29], that regardless of their goals, such systems operate by frequently updating the network configuration, either periodically or in reaction to events such as failures, load changes or routing policy changes. Updating network configuration is inherently challenging because it involves performing operations across different unsynchronized devices in multiple steps, each of which must be planned to avoid forwarding anomalies (such as loops and black-holes) or congestion.

Previous works have proposed mechanisms to update the network while retaining certain consistency properties during the configuration changes [16, 18, 20, 21, 24, 27, 29]. However, all these approaches have always commonly assumed that the SDN controller actively drives the update of network configuration by (*i*) scheduling each step, (*ii*) sending rule updates to the switches, and (*iii*) pausing when necessary to await acknowledgments from switches. In other words, the switches just behave as remote passive nodes that the controller writes state to and is notified from.

This controller-driven update process has three 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-art approaches [16], a network update typically takes seconds to be completed (recent results [16] shows 99^{th} percentiles as high as 4 seconds). Second, because scheduling updates is computationally expensive [16, 24, 27], *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.*

Performing network updates in a fast manner is a fundamental requirement in many critical scenarios. For example, when network devices fail, network operators rely on fast-failover techniques to preserve connectivity [8, 28]. These techniques consist in precomputing a backup forwarding state, which relies on local information at each switch. Since these backup forwarding states are not globally optimized, a fast network update towards a global optimal state reduces the period during which users would experience a worsened network service. Second, when network devices detect unusual traffic, there is a desire to reroute some flows through a set of security devices that would perform further analysis. By quickly updating the network state, the mitigation of an attack is more effective due to the faster adaption to the new network configuration.

In contrast with prior methods, we investigate the prospect of delegating the responsibility of consistent updates to the switches with the goal of achieving faster network updates completion time. We propose a distributed network update architecture wherein the controller is only responsible for computing the intended network configuration and pre-computing information needed by the

ANRW '16, July 16, 2016, Berlin, Germany

ISBN 978-1-4503-4443-2/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2959424.2959435

switches to schedule network update operations. The actual update function is realized by the switches, which coordinate execution of an update for the entire network using the information received by the controller and direct message passing among 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. We posit this approach leads to faster network updates, reduces the number of exchanged messages in the network, and has low complexity for the scheduling computation.

We argue that this approach is practical and supported by several recent trends that have demonstrated switch designs [1, 2, 4, 15] far more programmable when compared to OpenFlow switches (that are limited to a simple match-action paradigm). Moreover, networking industry is pursuing new open source operating systems and custom applications running on network switches [3, 10, 26]. For example, Facebook is already capable of executing custom software logic on switches to support their network management, automation and monitoring platform [9].

We formulate the distributed network update problem (§2) based on a model that has several distinctions from previous ones (e.g., [7, 16, 29]). Our model allows us to solve potential link congestion by carefully splitting traffic aggregates volumes, and to leverage "flow segmentation", a novel approach to speed up the update of a single flow by parallelizing its update operations.

The paper contributions are summarized as follows:

- We introduce ez-Segway (§3-§4), a consistent update scheduling mechanism that runs as software on switches, initially coordinated by a centralized SDN controller. Our algorithms enable decentralized network updates that avoid any forwarding anomalies and avoid link congestion while allowing flexible scheduling according to dynamic traffic condition.

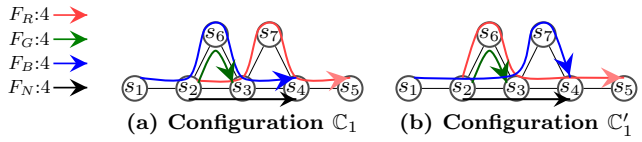
- We assess our system by running a comprehensive set of large scale simulations (§5) on various topologies and standard traffic patterns, which show that ez-Segway speeds up update time to 2x.

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 the standard model for characterizing flow traffic volumes as in [13, 16]. 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 [16] or based on an allocation of bandwidth resources [14]. The forwarding state of a flow consists of an exact match rule that matches all packets of the flow. As in previous work [16], 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 [17].

Network configurations and updates. A *network configuration* $\mathbb{C}$ is the collection of all forwarding states that determine what and how packets are forwarded between any pair of switches (*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 $\mathbb{C}'$.



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

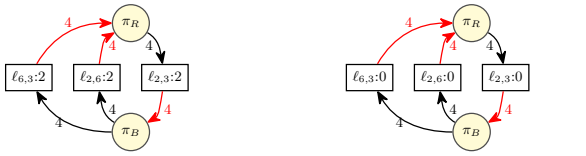
Figure 1: A network update with splittable deadlock.

Properties. We focus on three properties of network updates: *i*) *black-hole freedom*: For any flow, 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 [23]. We allow a flow to be routed through a mix of the old and new path configuration, unless other constraint make it impossible (*e.g.* set of middle-boxes must be visited in a reversed order in two configurations).

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 is 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. In order 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* (defined later). At anytime, 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 one $\mathbb{C}'$ such that for all intermediate configurations, the aforementioned three properties are preserved.

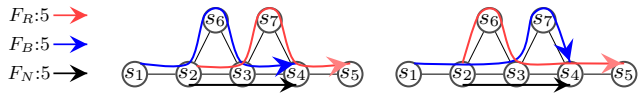
Dealing with deadlocks. We say that a network configuration is in a *deadlock* configuration if any further execution of an update operation will cause a violation of any desired property.

Unfortunately, similarly to the centralized setting [16], some network updates are not feasible: that is, even if the initial and target configuration do not violate any of the three property, there exists no ordering of update operations to reach the target. For example, consider an example based on the network of seven switches $s_1, \cdots, s_7$ shown in Figure 1. Assume each link has 10 units of capacity and there are several traffic aggregates F_R, F_G, F_B, F_N depicted with red, green, blue, and black colors, respectively, each of size 4. This means that every link can carry at most 2 flows at the same time, assuming that flows cannot be split.



(a) Splittable deadlock **(b) Unsplittable deadlock**

Figure 2: Dependency graph of deadlock cases.



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

Figure 3: A network update with segment deadlock.

We denote a path through a sequence of nodes $s_1, \ldots, s_n$ by $(s_1 \ldots s_n)$. The network configuration needs to be updated from $\mathbb{C}_1$ to $\mathbb{C}_1'$. If we first move F_R, $\ell_{2,6}$ becomes congested; if we first move F_B, $\ell_{2,3}$ becomes congested.

Even worse, computing a feasible schedule where flows cannot be fractionally split is a hard problem even in the centralized setting: it is NP-complete in the presence of both link capacity and switch memory constraints, and finding the fastest schedule with link capacity constraints is NP-complete [16]. To resolve deadlocks, typical approaches consist of reducing flow rates or pre-emptively reserving some spare capacity in order to continue an update without congesting a link; 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.

The dependency graph. The dependency graph captures the set of dependencies between network update operations and 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. The two subsets of edges 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.

In Figure 2a, we show a subgraph of the dependency graph derived from the initial configuration of Figure 1 in which we only highlight links $\ell_{6,3}$, $\ell_{2,6}$, and $\ell_{2,3}$ plus the update operation for moving the blue flow F_B and the red flow F_R. Observe that none of the update operations can be fully performed in single update operations since the required bandwidth (i.e., 4 units) is higher that the available one (i.e., 2 units) and partially moving a fraction of a flow is not allowed. We will show how splitting traffic volume can solve these types of deadlocks in the next section.

3. EZ-SEGWAY

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

To speed up network updates, we leverage two new ideas. First, 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. Second it uses *flow segmentation*, which allows us to split a single flow update into different independent chunks that can be updated independently from each other.

To deal with deadlocks, we both use flow segmentation and *volume splitting*, which divides a flow's traffic onto its old and new paths to resolve a deadlock. We discuss the aforementioned techniques in the rest of this section.

3.1 Decentralizing Consistent Updates

ez-Segway advocates that a small number of crucial update operations should be performed within the switches. The central controller pre-computes information needed by the switches to schedule network update operations and transmit it to them. The actual update function is realized by the switches, which are then able to schedule network update operations without interacting anymore with the controller. 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 scheduling information provided by the controller. These functions are computationally inexpensive and easy to implement in currently available programmable switches.

We describe the execution of a decentralized network update for the example of Fig. 3, where each flow has size 5. The network configuration needs to be updated from $\mathbb{C}_2$ to $\mathbb{C}_2'$. Note that we cannot simply transition to $\mathbb{C}_2$ by updating all the switches at the same time. Since switches apply updates at different time, such a strategy can neither ensure congestion freedom nor loop- and black-hole- freedom. For example, if s_2 forwards F_B on link $\ell_{2,3}$ before F_R is updated, then that link would become congested. Moreover, 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 blackhole.

In ez-Segway, initially, the controller sends to every switch a message containing the dependency graph of the network update from the current configuration $\mathbb{C}_2$, and the target configuration $\mathbb{C}_2'$. 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).

In the example, switch s_6 infers that link $\ell_{6,3}$ has enough capacity to carry F_R and that its next hop switch, s_3, is already capable of forwarding F_R (because the flow traverses it in both old and new configurations). Therefore, s_6 reserves 5 units of bandwidth for F_R and notifies s_2 about this information. As a consequence, s_2 also infers that link $\ell_{2,6}$ has enough capacity to carry F_R so it updates its forwarding state so as to move F_R from path $(s_2 s_3)$ to $(s_2 s_6 s_3)$. Similarly, s_3 updates its forwarding state for F_B to flow on $(s_3 s_7 s_4)$ instead of $(s_3 s_4)$.

Now, s_2 infers that link $\ell_{2,3}$ has enough capacity to carry

F_B. So, s_2 updates its forwarding state so as to move F_B from path $(s_2 s_6 s_3)$ to $(s_2 s_3)$ and communicate to s_6 that it is now safe to remove its forwarding entry for F_B. Similarly, s_3 infers that link $\ell_{3,4}$ has enough capacity to carry F_R. Therefore, s_3 updates its forwarding state so as to move F_R from path $(s_3 s_7 s_4)$ to $(s_3 s_4)$ and communicate to s_7 that it is now safe to remove its forwarding entry for F_R.

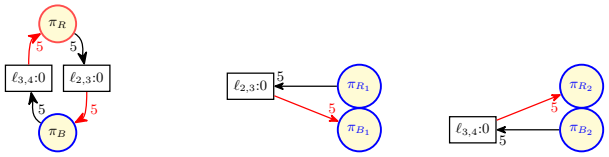
Notice that several update operations can run in parallel at multiple switches. However, whenever operations have unsatisfied dependencies, switches must coordinate. In this example, the longest dependency chain involves the three operations that must occur in sequence at s_2, s_6, and s_2 again. So, the above strategy 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 approach 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).

3.2 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 with simple examples and then provide formal definitions. The example discussed in the previous paragraphs already leveraged the idea of partitioning an update operation into several different independent "segments". We first discuss a simple distributed approach to update a flow and we then show how segmentation can help in speeding up a flow update.

Update operation in the distributed approach. Consider the same update problem depicted in Figure 3 where a flow F_B needs to be moved from the old path $(s_1 s_2 s_6 s_3 s_4)$ to the new path $(s_1 s_2 s_3 s_7 s_4)$. A simple approach would work as follows. A message is sent from s_4 to its predecessor on the new path (*i.e.*, s_7) 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 traverses 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. Once the first switch of the new path (*i.e.*, s_1) alters to the new path, a new message is sent from s_1 towards s_2 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 migration 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.

Segmentation approach. It can be observed that the whole flow update operation for F_B could be performed faster. With segmentation the subpath $(s_2 s_6 s_3)$ of the old path can be updated with Basic-Update to $(s_2 s_3)$ while at the same time the subpath $(s_3 s_4)$ of the old path is updated with Basic-Update to $(s_3 s_7 s_4)$. In fact s_2 does not have to wait for s_3 to update its path since s_3 is guaranteed to have a forwarding path towards s_4 both in the old and new configuration. Moreover, since flow F_B does not change between s_1 and s_2, we can ignore that part of the update. This technique allows us to update a flow into several independent "flow segments", which can be treated as if they are independent flows to be updated. Segments can be created by looking for common vertices in the initial and target configuration, paying attention to some edge cases in which forwarding loops may arise[1]. By splitting a flow update operation into multiple parallel flow update operations we can speed up the network update completion time. Coming back to the example given in Section 3, we show a subset of its dependency graph in Figure 4a where π_R (π_B) represents the update operation of the red (blue) flow and π_B. This dependency graph is in a deadlock state since neither the red nor the blue flow can be updated. 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 4b and 4c, where $\pi_{R,1}$ and $\pi_{B,1}$ (respectively $\pi_{R,2}$ and $\pi_{B,2}$) represent the update operation of the red and blue segments from s_2 to s_3 (resp. from s_3 to s_4). Segmentation can therefore be used to both speed up an update and resolve deadlocks.



(a) Ex. in Fig. 3 (b) s_2 with segments (c) s_3 with segments

Figure 4: Deadlock solvable by segmentation

3.3 Splitting Volume

In Figure 1, 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 gives sufficient capacity to move F_R to $\ell_{2,6}$. Once F_R is moved, there is sufficient capacity to complete the mode of F_B. Note that, we could even move 2 units of F_R simultaneously to moving two units of F_B, before the update completes as before. This is what our decentralized solution would actually do. The deadlock would not be splittable if the capacity of the links were 8, as shown in Figure 4a.

3.4 Scheduling

Before introducing our scheduling algorithm, we first categorize the space of possible deadlocks. If a deadlock can be solved by splitting volumes, we say that the deadlock is *splittable*. In ez-Segway, if a switch s detects a deadlock, it looks for a flow F_p 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

[1]Due to space constraints we do not discuss these cases here.

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. In these cases, ez-Segway relies of rate limiting to solve the deadlock.

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 segment is updated with Basic-Update, which means that each flow segment is updated directly from its old path to the new one as long as there is enough spare 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 operation 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 operation that would otherwise be not executable (*i.e.*, even if every other update operation could be possibly executed). A *critical* cycle is a cycle that contains a critical update operation.

ez-Schedule assigns low priority to all the update operations 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.

We can prove that ez-Schedule is *correct*, 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 $O(|\Pi|+|L|+|\Pi|\times|L|)\simeq O(|\Pi|\times|L|)$. So, for all update operations the complexity is $O(|\Pi|^2\times|L|)$.

Theorem 1. ez-Schedule *is correct for the network update problem.*

4. DISTRIBUTED COORDINATION

This section describes the mechanics of coordination during network updates.

First phase: the centralized computation. As described in § 3.2, to avoid black-holes and forwarding loops, each single segment can be updated using a Basic-Update. Dependencies among update operations are enforced by their priorities (*i.e.*, low, medium, high), which are computed by the controller and transmitted to each switch.

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, 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. 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, for each segment S, each switch prepares for performing two actions: installing the new path of S and removing the old one. The messages exchanged by the switches for performing these two operations are described in the next two paragraphs.

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 its predecessor in the new path, which acknowledges the receiver that the downstream path is ready to carry the traffic volume of S. Upon receiving a `GoodToMove` message for S, a switch checks if there is enough bandwidth on the outgoing link to execute the update operation. If not, it freeze the update operation until there is enough spare bandwidth. If there is enough bandwidth, it checks if there are update operations that require the outgoing link and have higher priority than S. If not, the switch executes the update operation. Otherwise, if the residual capacity of the link minus the traffic volume of S does not prevent any higher level update operation to be executed in the future, it executes the update operation. If the switch 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.

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, it removes the entry for the old path and it forwards the message to its successor in the old 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.

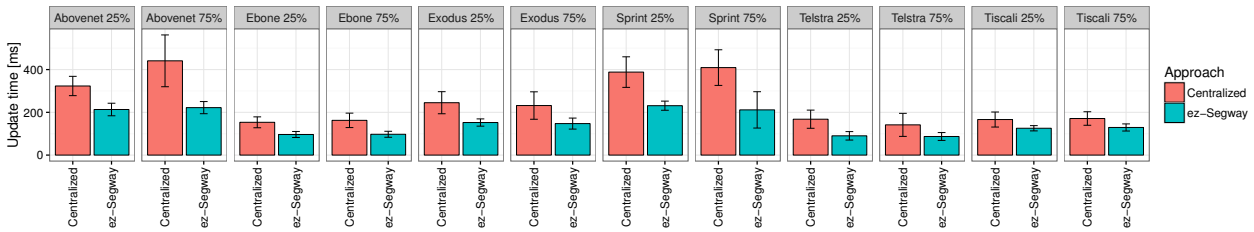


Figure 5: Update time of ez-Segway versus Centralized for various settings.

Dealing with failures. As with a centralized approach, if a switch or link fails during an update, a new valid target configuration must be computed. We stress the fact that the decentralized approach of ez-Segway is responsible for moving from one 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 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.

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. LARGE-SCALE SIMULATIONS

Using simulations, we compare ez-Segway versus a centralized approach that runs the same scheduling algorithm of ez-Segway but coordination among the switches is delegated to a central controller. This approach is a close approximation of Dionysus [16], the only deviation is segmentation to speed up update time.

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 $1 \sim 100$Gbps, inversely proportional to weights. We place the controller at the centroid switch. We stresst the fact that this is the best setting for centralized approach, which are unlikely to happen in practice.

Following the methods in [12], 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 based on the gravity model [30]. For a given source-destination pair, we compute a path by first selecting a third transit node at random and then computing a cycle-free shortest path that contains the three nodes 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 and that there is sufficient capacity to route the flow on the chosen path.

We run simulations for a number of flows varying from 500 to 1500 and report results for 1000 flows as we observed qualitatively similar behaviors. We generate updates by simulating link failures such that they 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. Fig. 5 shows our results, which demonstrate that ez-Segway reduces the update completion time by a factor of $1.5 - 2$. We want to stress the fact that the main limiting factor for further reducing the update completion time in ez-Segway is the latency due to the physical propagation delay of the messages, which cannot be reduced.

6. RELATED WORK

The network update scheduling problem has been widely studied in the last years [5,7,16,18,20,24,25,27,29,32]. These works use *centralized* approaches based on the SDN control plane to preserve the logical constraints of network update. The approaches only consider the case where every flow can be atomically updated as a whole, which increases the number of deadlock scenarios. In contrast, ez-Segway tackles the network update problem in a decentralized manner, allows flow segmentation, splitting of traffic volumes. To the best of our knowledge, ez-Segway is the first system solving the network update problem with a distributed approach.

The works in [5,16] show that without splitting a flow the general network update problem, and some of its variations, are NP-hard. A centralized scheduling algorithm, called Dionysus [16], that updates flows atomically. It computes a dependency graph that represents the dependencies of the flow update operations with respect to the link capacity resource available in the network. This dependency graph is used by the *centralized* SDN control plane to perform update operations with a flexible scheduling based on the actual finishing time of update operations across switches.

In a very recent work, both segmentation and splitting volumes techniques have been independently proposed to solve deadlocks [32]. However, these techniques are used in a centralized setting, hence missing the importance of using segmentation for speeding up a distributed network update.

7. CONCLUSION

This paper explored delegating the responsibility of consistent updates to the switches. We proposed *ez-Segway*, a decentralized mechanism where the controller only computes the desired network configuration and switches have an active role to realize consistent network updates that provably satisfy four properties: black-hole freedom, loop freedom, and congestion freedom. In practice, this approach leads to improved update times, which we quantified via simulation on a range of network topologies and traffic patterns. As part of our ongoing work, we are also deploying our approach on a real switch to investigate the feasibility and low computational overhead.

8. ACKNOWLEDGMENTS

We would like to thank Xiao Chen, Paolo Costa, Tu Dang, Ratul Mahajan, Petr Peresini, Jennifer Rexford, 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).

9. REFERENCES

[1] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. OpenState: Programming Platform-Independent Stateful Openflow Applications Inside the Switch. *SIGCOMM Comput. Commun. Rev.*, Apr. 2014.

[2] R. Bifulco, J. Boite, M. Bouet, and F. Schneider. Improving SDN with InSPired Switches. In *SOSR'15*.

[3] BigSwitch. Switch Light. http://www.bigswitch.com/products/switch-light.

[4] 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.*, July 2014.

[5] S. Brandt, K.-T. Förster, and R. Wattenhofer. On Consistent Migration of Flows in SDNs. In *INFOCOM'16*.

[6] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and Implementation of a Routing Control Platform. In *NSDI'05*.

[7] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. A Distributed and Robust SDN Control Plane for Transactional Network Updates. In *INFOCOM'15*.

[8] M. Chiesa, S. Mitrovic, I. Nikolaevskiy, A. Gurtov, A. Panda, A. Madry, M. Schapira, and S. Shenker. The Quest for Resilient (Static) Forwarding Tables. In *INFOCOM'16*.

[9] Facebook Ops Director On Breaking Open The Switch. http://goo.gl/GpjEQF.

[10] Facebook. Introducing "Wedge" and "FBOSS," the next steps toward a disaggregated network. https://goo.gl/WUIztC.

[11] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthy. Participatory Networking: An API for Application Control of SDNs. In *SIGCOMM'13*.

[12] J. He, M. Suchara, M. Bresler, J. Rexford, and M. Chiang. Rethinking Internet Traffic Management: From Multiple Decompositions to a Practical Protocol. In *CoNEXT'07*.

[13] 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'13*.

[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'13*.

[15] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *SIGCOMM'14*.

[16] 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'14*.

[17] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford. Efficient Traffic Splitting on Commodity Switches. In *CoNEXT'15*.

[18] N. P. Katta, J. Rexford, and D. Walker. Incremental Consistent Updates. In *HotSDN'13*.

[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'14*.

[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'13*.

[21] A. Ludwig, J. Marcinkowski, and S. Schmid. Scheduling Loop-free Network Updates: It's Good to Relax! In *PODC'15*.

[22] R. Mahajan, N. T. Spring, D. Wetherall, and T. E. Anderson. Inferring Link Weights Using End-to-end Measurements. In *SIGCOMM Internet Measurement Workshop, IMW'02*.

[23] R. Mahajan and R. Wattenhofer. On Consistent Updates in Software Defined Networks. In *HotNets'13*.

[24] J. McClurg, H. Hojjat, P. Černý, and N. Foster. Efficient Synthesis of Network Updates. In *PLDI'15*.

[25] T. Mizrahi, O. Rottenstreich, and Y. Moses. TimeFlip: Scheduling Network Updates with Timestamp-based TCAM Ranges. In *INFOCOM'15*.

[26] Open Network Linux. http://opennetlinux.org/.

[27] P. Perešíni, M. Kuźniar, M. Canini, and D. Kostić. ESPRES: Transparent SDN Update Scheduling. In *HotSDN'14*.

[28] M. Reitblatt, M. Canini, A. Guha, and N. Foster. FatTire: Declarative Fault Tolerance for Software-defined Networks. In *HotSDN'13*.

[29] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM'12*.

[30] M. Roughan. Simplifying the Synthesis of Internet Traffic Matrices. *SIGCOMM Comput. Commun. Rev.*

[31] 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'14*.

[32] W. Wang, W. He, J. Su, and Y. Chen. Cupid: Congestion-free Consistent Data Plane Update In Software Defined Networks. In *INFOCOM'16*.