pUpdate: Priority-based Scheduling for Continuous and Consistent Network Updates in SDN

Mehdi Nobakht
School of Systems and Computing
University of New South Wales, Canberra, Australia
mehdi.nobakht@unsw.edu.au
https://orcid.org/0000-0003-4676-536X

Arshiya Rezaie Hezaveh
Computer Engineering Department
Sharif University of Technology, Tehran, Iran
arshiya.rezaie@sharif.edu

***Abstract*—Preserving consistency properties, such as preventing forwarding loops, black-holes, and congestion, is crucial during the transitions in network forwarding state that occur between network updates. Nevertheless, maintaining these properties within SDN networks poses challenges due to flexibility and programmability of SDN, which can result in an increased frequency of network updates. While existing research has introduced mechanisms to uphold consistency during network transitions, they often overlook the consideration of flow priority when scheduling forwarding rules. We address this problem with pUpdate, a framework designed to facilitate consistent and continuous network updates while accounting for flow priority in the scheduling of forwarding rules within SDN. We have developed pUpdate and rigorously evaluated its performance through simulations. The measurements demonstrate that pUpdate adheres to priority-based scheduling without any degradation in network update performance when compared to state-of-the-art approaches in SDN network update.**

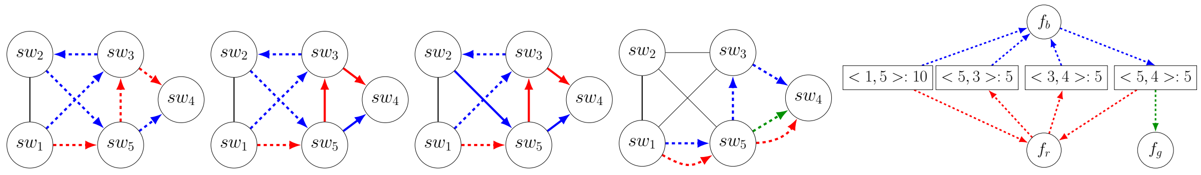
I. Introduction

In computer networking, the process of modifying the network forwarding state is typically known as a *network update*. These updates are triggered by events such as topology changes, failures, and policy adjustments, and they are initiated by network applications. In comparison to traditional non-SDN networks, SDN networks experience more frequent updates due to their enhanced flexibility, which allows for dynamic modifications initiated by network applications.

Maintaining consistency properites is crucial during network updates to avoid undesired behavior of network. The essential consistency properties in networking are: *i*) prevention of *black-holes*: ensuring that no packet are dropped within the network, *ii*) avoiding *forwarding loops*: ensuring that packets do not circulate endlessly in the network, *iii*) *per-packet consistency*: guaranteeing that packets are forwarded along either their initial or final routes, but not both, and *iv*) maintaining *congestion-free* condition: designing the network to avert excessive congestion. These properties are critical in ensuring the reliability and stability of a network during the *transitions* that occur between network updates. Proper management of network updates is vital to preserving consistency during updates and preventing performance degradation.

In the literature, research on consistent network updates can be classified into two primary categories. The first category revolves around the sequential execution of updates, one at a time, transitioning the network from an initial state to a final state in a step-by-step manner [1], [2], [3], [4]. In this approach, known as *single update*, a new update must patiently await the completion of the current update. This approach assists in preventing conflicts and guarantees that updates follow a predictable sequence, thereby minimizing the potential for inconsistencies during the transition between updates. Nevertheless, it faces challenges when confronted with sporadic bursts of update requests. In response to this challenge, a second category has emerged, focusing on the continuous execution of network updates [5], [6]. Within the *continuous update* category, the primary emphasis lies in the immediate execution of updates as they occur, accomplished by merging new updates with any previously unexecuted ones. This amalgamation results in operations akin to executing updates sequentially.

Despite considerable research efforts aimed at developing efficient algorithms for ensuring consistent network updates in SDN, the current solutions often neglect to account for flow priorities when updating rules. Our prior work [7] highlighted the repercussions of neglecting flow priorities during the installation of forwarding rules in SDN, all while striving to maintain consistency properties. Overlooking flow priority can result in *priority inversion*, where lower-priority flows are updated ahead of higher-priority ones. This situation can lead to extended network downtime for high-priority flows, particularly in events like link failures. Additionally, existing methods designed to preserve consistency during network updates may introduce *deadlocks*, where no progress can be made due to a scarcity of link resources occupied by flows. To mitigate deadlocks, existing solutions suggest reducing flow rates, which, unfortunately, comes at the cost of decreased network throughput. Nonetheless, this approach can potentially leave high-priority flows susceptible while providing relief to low-priority ones, thereby causing prolonged downtime and imposing resource constraints on the controller for high-priority flows.



(a) Update 1 (U_1) (b) State 1 (S_1) (c) State 2 (S_2) (d) Update 2 (U_2) (e) transition from S_1 to U_2

Figure 1: An Example of network updates (see §II for a detailed discussion of the example)

We present pUpdate, a framework designed to incorporate flow priority into the scheduling of forwarding rules within SDN, all while maintaining consistency during network updates. It combines the advantages of continuous updates with efficient scheduling algorithms to achieve fast and consistent network updates. We have implemented a prototype of pUpdate in Java code. We evaluate the performance of pUpdate by conducting experiments within a testbed topology that simulates two network topologies; a WAN and a three-layer FatTree datacenter topology. The WAN topology was constructed with a total of 8 switches, while the FatTree topology consisted 80 switches. Evaluation results demonstrates that pUpdate adheres to priority-based scheduling of forwarding rules across a variety of evaluation settings without any performance degradation. These settings include a mix-and-match of network topologies, varying update arrival rates, and varying network loads.

II. Motivating Example

To demonstrate the negative impact of not considering the priority of flow in scheduling rule updates in SDN, consider a five-node network illustrated in Fig. 1. The network topology has links with a capacity of 10 units each. Directed edges that are solid indicate executed forwarding rules, while dashed directed edges indicate unexecuted forwarding rules. All flows in the network have a demand of 5 units. U_1 installs two flows (f_b and f_r) with medium and low priorities and color-coded in blue and red, respectively (Fig. 1a).

A straightforward approach would be to send out all forwarding rule updates in one shot. For example, instructing all relevant switches to install the related forwarding rules for f_b and f_r at once. However, this approach could result in a black-hole in sw_5 if sw_1 updates its rule before sw_5. Although the black-hole will eventually disappear once sw_5 updates its rule, there is no guarantee when this will happen in an asynchronous system with possible message delays and losses. By properly ordering and timing updates, the consistency properties of the network can be maintained. Fig. 1 depicts several intermediate states (such as *State* 2 or S_2) during the update process to maintain consistency in the network.

Assume that the controller receives an update event U_2 while executing U_1. U_2 re-routes flows (f_b and f_r) to different paths and installs a new flow f_g with high priority, which is color-coded in green (Fig. 1d). In a single network update strategy, if update U_2 occurs while the controller is executing U_1 and the network state is *State 1* (Fig. 1b), U_2 must wait until the execution of U_1 is complete. This means that update events U_1 and U_2 will be executed sequentially, resulting in network forwarding state changes between the deterministic states given by the updates (e.g., $U_1 \rightarrow U_2$).

In contrast, the continuous network update method aims to transition from an intermediate state to the target state (e.g., $S_1 \rightarrow U_2$) without interruptions. The intermediate states contain unexecuted operations that will be queued at the controller. However, if not properly handled, the queuing can cause flow priority inversion, where a high-priority flow in a recent update event may be prevented from being updated due to low-priority flows waiting to be fully executed. This can negatively impact network performance.

Consider a scenario where U_2 arrives while the controller is executing U_1 and the network state is *State 1* (Fig. 1b). To change the network forwarding state from S_1 to U_2 ($S_1 \rightarrow U_2$), the controller has to execute the operations illustrated in Fig. 1e. If the controller does not prioritize flow updates and first executes flow f_r, flow f_g will have to wait until flow f_b releases the link between sw_5 and sw_4 (denoted by $\langle 5, 4\rangle$) because f_r has consumed all the remaining capacity of $\langle 5, 4\rangle$. This highlights the importance of considering flow priority as there is no guarantee that f_g will acquire its resources in the next step, as the controller may not update f_g if a new update arrives.

III. pUpdate Architecture

We present an overview of the pUpdate architecture, which consists of three key components, as illustrated in Fig. 2. Upon receiving an update event (e.g., U_{n+1}), containing new paths for a set of flows, pUpdate generates forwarding rule operations that are subsequently transmitted to the switches.

The *Initial Operations* component generates operations to transition from the current paths to the target paths associated with the new update event. The *Operation Composition* component compares the new update event with the ongoing update event, and then combines these operations with unexecuted operations to produce new operations that can change the network forwarding state from the current intermediate state to the

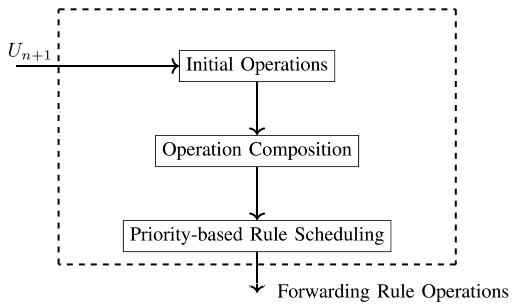


Figure 2: Priority-based Network Update Framework

target state. The *Priority-based Forwarding Rule Scheduling* component installs flow rules based on flow priority.

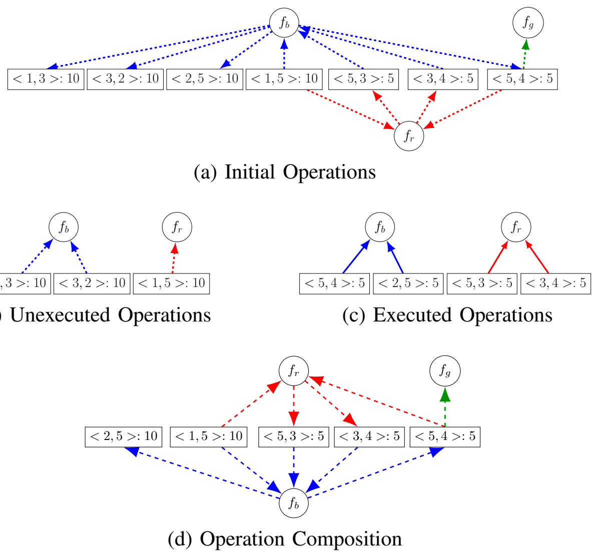
Initial Operations: When a new update event is received, pUpdate pauses the ongoing update process and begins to process the new event. Initially, it assumes that all pending forwarding rules have already been executed. Then, it compares the paths in the new event with the paths in the current update to identify the necessary initial operations to transition from the current paths to the target path. Subsequently, pUpdate compares the paths in the new event with the paths in the current update to identify the necessary initial operations for transitioning from the current paths to the target path. These initial operations are equivalent to the operations that the controller would execute in the *single update* method, meaning that arriving update events are executed sequentially or in a blocking manner.

Operation Composition: Operation composition aims to optimize network updates by reducing the number of required operations and minimizing controller-to-switch communication overhead. In the context of switch operations, there are three types: *add*, *delete* (*del*), and *modify* (*mod*), each with the following format:

- $add(target\_switch, flow, next\_hop)$
- $del(target\_switch, flow)$
- $mod(target\_switch, flow, new\_next\_hop)$

The $add(sw_i, f_k, sw_m)$ operation forwards flow f_k to switch sw_m after being executed in switch sw_i. This is achieved by adding the match-action fields in the forwarding table of switch sw_i that direct flow f_k towards sw_m. The $del(sw_i, f_k)$ operation removes the forwarding rule for flow f_k from the forwarding table of switch sw_i. Since each flow has at most one entry in each switch's forwarding table, only one entry will be removed. The $mod(sw_i, f_k, sw_j)$ operation changes the next hop of flow f_k to switch sw_j in the forwarding table of switch sw_i.

During this phase, the initial operations are combined with any unexecuted operations from the current update, considering allocation of the links in the data plane state, to generate new operations. As a result, new operations replace the existing unexecuted operations in the control plane. This step results



(a) Initial Operations

(b) Unexecuted Operations

(c) Executed Operations

(d) Operation Composition

Figure 3: State of proposed framework when U_2 arrives at S_2

in fewer equivalent operations capable of transitioning the network forwarding state from the current intermediate state to the target state specified by the new update. Table I outlines all possible combinations of new operations from a newly arrived update event and unexecuted operations from current update event, with taking into account the executed operations in the data plane.

Priority-based Forwarding Rule Scheduling: In the final phase, pUpdate pushes flow forwarding rules to individual switches according to the flow's priority level. Within the pUpdate framework, two priority scheduling policies have been integrated: Strict Priority and Weighted Round Robin (WRR).

In the context of Strict Priority Scheduling, forwarding rules linked to flows of higher priority are first installed in the data plane, preceding the installation of forwarding rules related to lower-priority flows. The execution of forwarding rules associated with lower-priority flows occurs after the higher-priority flow has finished processing. This scheduling policy is commonly used in operating systems and real-time systems, where tasks are assigned priorities, and tasks with higher priorities are executed ahead of those with lower priorities. Its purpose is to allocate network resources to flows based on their priority, thereby mitigating priority inversion issues. However, a potential drawback is that low-priority flows might encounter *resource starvation* if a high-priority flow consistently obstructs the installation of low-priority flow rules.

Weighted Round Robin (WRR) stands as a scheduling algorithm used for distributing resources, such as CPU time, network bandwidth, or disk access time, among various competing processes or flows. WRR operates by assigning a weight to each priority class (N_{pr}), which then dictates the number of flows to be updated during each scheduling round. For instance, if we allocate $N_{high} = 10, N_{medium} = 8,$ and $N_{low} = 5$

Table I: Composition rules

New Operation	Unexecuted Operation	Allocation of Links in the Data Plane State	Composition Result	Scenario #
$add(sw_i, f_k, sw_m)$	$\emptyset$	$\{l_{i,m} \nrightarrow f_k\}$	$add(sw_i, f_k, sw_m)$	1
	$del(sw_i, f_k)$	$\{l_{i,m} \rightarrow f_k\}$	*Null*	2
	$del(sw_i, f_k)$	$\{l_{i,j} \rightarrow f_k\}$	$mod(sw_i, f_k, sw_m)$	3
$del(sw_i, f_k)$	$add(sw_i, f_k, sw_m)$	$\{l_{i,m} \nrightarrow f_k\}$	*Null*	4
	$\emptyset$	$\{l_{i,m} \rightarrow f_k\}$	$del(sw_i, f_k)$	5
	$mod(sw_i, f_k, sw_m)$	$\{l_{i,j} \rightarrow f_k\}$	$del(sw_i, f_k)$	6
$mod(sw_i, f_k, sw_q)$	$add(sw_i, f_k, sw_m)$	$\{l_{i,m} \nrightarrow f_k\}$	$add(sw_i, f_k, sw_q)$	7
	$\emptyset$	$\{l_{i,m} \rightarrow f_k\}$	$mod(sw_i, f_k, sw_q)$	8
	$mod(sw_i, f_k, sw_m)$	$\{l_{i,j} \rightarrow f_k\}$	$mod(sw_i, f_k, sw_q)$	9

Note: *Null* means:"do nothing", $\emptyset$ means there is no operation for the flow in the switch. $\{l_{i,m} \nrightarrow f_k\}$ means link $l_{i,m}$ is not allocated to flow f_k.

to high, medium, and low priority classes respectively, each round of scheduling would involve updating 10 high-priority flows, followed by 8 medium-priority flows, and finally, 5 low-priority flows. WRR is effective in mitigating the potential resource starvation issues linked with Strict Priority Scheduling, although it may introduce some degree of priority inversion as a trade-off.

An example. To illustrate the process in the architecture discussed above, consider the example introduced in §II. Imagine a scenario when the network state is S_2, the next update (U_2) will arrive. The architecture prioritizes flows while changing the network state from S_2 to U_2.

The *Initial Operations* compares the new paths in U_2 with the paths in the current update (U_1) to identify initial operations. For example, in U_1 policy, sw_1 forwards f_b to sw_3, but the next hop after sw_1 is sw_5 in U_2 policy. Therefore, if updates are executed sequentially, sw_1 must execute the operation $mod(sw_1, f_b, sw_5)$ to change the path of f_b (Fig. 3a).

In the next step, the *Operation Composition* changes the network forwarding state from the current intermediate state (S_2) to the target state (U_2). The operation $mod(sw_1, f_b, sw_5)$ can be merged with the unexecuted operation $add(sw_1, f_b, sw_3)$ (similarly, $\{l_{1,3} \nrightarrow f_b\}$), resulting in $add(sw_1, f_b, sw_5)$ (Figs. 3b and 3c). This scenario matches with scenario 7 in Table I. Instead of executing two update operations $add(sw_1, f_b, sw_3)$ and $mod(sw_1, f_b, sw_5)$ sequentially, only one operation is needed to achieve the same result (Fig. 3d).

if Strict Priority scheduling is used, flows will be updated in the following order: f_g, which has the highest priority, will be updated first. Next, f_b, a medium priority flow, will acquire its resources. Lastly, the flow with the lowest priority, f_r, will be updated.

IV. Implementation and Evaluation

We have developed a pUpdate prototype, which consists of more than 3,000 lines of Java code. The prototype includes the implementation of the two previously discussed priority-based scheduling algorithms. We evaluate the performance of pUpdate prototype through simulation-based experiments. The simulation environment setup is a flow-level simulation.

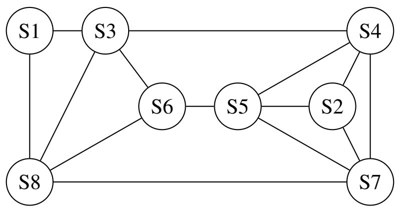


Figure 4: WAN topology as described in [2]

We first evaluate the performance of pUpdate with two priority-based scheduling algorithms when different classes of flows are waiting to be installed. Furthermore, to show that pUpdate is advantageous, we compare it against two state-of-the-art proposals, single update from Cupid [3] and continuous updates from Coeus [6]. We specifically chose Cupid as our benchmark due to its superior performance compared to its predecessors, including Dionysus [2] and SWAN [1]. By comparing our architecture against Cupid, we can showcase the advancements and improvements achieved in the realm of single update approaches. Similarly, we selected Coeus [6] as it represents the most recent work in continuous update methods. Coeus builds upon the foundation of Update Algebra [5] and incorporates considerations for the congestion freedom property.

A. Evaluation Setup

We evaluate pUpdate across a variety of settings that mix-and-match two network topologies, different workloads and traffic patterns, and varying update inter-arrival times. The evaluation cover a range of parameters, including topologies, workload, performance metrics, and network properties. To ensure robustness and accuracy, we conducted each experiment 10 times and present the average results obtained. The summary of all evaluation parameters are listed in Table II.

Topology. We consider two network topologies, a WAN topology from Dionysus [2] and a three-layer FatTree datacenter topology [10]. The WAN topology consists of 8 switches as shown in Fig. 4. The switches are connected by 10 Gbps links. The FatTree topology consists of Edge (or Top-of-Rack),

Table II: Evaluation parameters for simulation

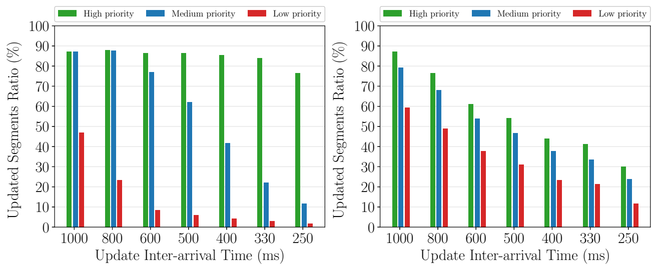
Topologies	**WAN** - A WAN topology consisting of 8 switches, which aligns with the same topology employed by [2], also used in prior works [6], [3], [8], [9] **DC** - A three-layer FatTree topology with 8 pods that consists of 80 switches [10] and 128 hosts.
Switch Properties	The average time required for inserting, removing, and modifying rules is 5 ms, 5 ms, and 10 ms, respectively. According to the findings of [2] on commodity switches [11]
Link properties	**WAN** - Link capacity: 10 Gbps, Propagation delay: 200 ns [12] **DC** - Link capacity:1 $\sim$ 10 Gbps, negligible propagation delay [2]
Traffic Classification	20% high-priority, 30% medium-priority and 50% low-priority flows according to [13], [1]
Workloads	The test is stressed by varying T from 100 $\sim$ 1000 ms, i.e., 1 to 10 updates per second [14], [15]. The average links' utilization varied between 20% to 80% [3]. We varied number of flows to update from 100 to 1000 flows per update [5], [6], [3].
Evaluated metrics	Ratio of updated to generated segments for flows with different priorities, CDF for time of updated flows, update completion time.

Aggregation, and Core layers. The FatTree testbed includes a total of 80 switches, with 32 edge switches, 32 aggregation switches, and 12 core switches. In terms of link capacity, the Aggregation and Core layers in the network are equipped with links with capacity of 10 Gbps. On the other hand, the links in the Edge layer have a capacity of 1 Gbps.

Workload. We evaluate pUpdate across various workloads, including a combination of all-to-all traffic and bursty traffic to simulate realistic network update scenarios. Consistent with recent studies [5], [6], we introduce variations in the number of flows to update, ranging from 100 to 1000 per update, as well as the update inter-arrival time (T), which spans from 250 ms to 1000 ms. While previous studies [5], [6] typically alter the path of 20% of flows during each update, we adopt a more comprehensive approach. In our evaluation, we modify the path of all flows in order to thoroughly assess the robustness of our approach and gauge its worst-case performance for every update. Moreover, the average link utilization is above 80%., which aligns with findings from a previous study [3].

Traffic Flow. In order to generate the datasets for our experiments, we followed a specific procedure. Firstly, we randomly selected two nodes that were not adjacent to each other to form a source-destination pair (s, d). Next, we chose another random node t and determined the shortest loop-free path from s to d that passes through t. After that, we assigned a demand and priority to the flow based on our network traffic classification policy, which is discussed in detail below. Finally, we verified that the link capacity was not exceeded, ensuring a congestion-free condition. We created the datasets using over 600 lines of Python code. To generate these datasets, we used the computational cluster Katana supported by Research Technology Services at UNSW Sydney [16], employing 24 GB of RAM and 12 CPU cores. It is important to note that generating datasets for large-scale topologies can be resource-intensive. For instance, creating datasets for the FatTree topology required approximately 12 hours to complete due to its complexity and scale.

Flow Priority Levels. In real-world networks, traffic is carried with different priorities. Building on prior research [1],



(a) Strict Priority Scheduling (b) Weighted Round Robin

Figure 5: Ratio of updated flows with different scheduling algorithms in WAN topology [2]

[13], we implement a network traffic classification policy to allocate a priority level and corresponding demand to each flow. Consistent with previous findings [1], we estimate that 20%, 30%, and 50% of network traffic belongs to high, medium, and low priority flows, respectively. In line with these observations, we implemented a prototype that takes into account three distinct traffic classes. Our classification system categorizes flows into three priority levels represented by L = 3 and labeled as {HIGH, MEDIUM, LOW}. Although additional traffic classes could be incorporated into our architecture, we only consider three traffic classes in our implementation and evaluation for simplicity.

B. Evaluation Results

We devise experiments aimed at assessing the pUpdate prototype through simulation. To carry out this evaluation, we execute a simulation run encompassing 15 consecutive network update events. Within these events, we allocate flow priority levels, as previously described, and measure the update time for the identified flows linked to these update events.

1) Priority-based Scheduling Policies: The purpose of this experiment is to examine how different priority-based scheduling policies affect the network performance. We focus on two scheduling policies: *strict priority scheduling* and *weighted round-robin*, and measure the ratio of updated flows for each.

Fig. 5 shows the ratio of updated flows to the total number of flows for various update inter-arrival times (T), ranging from 250 to 1000 ms. Previous studies [5], [6] typically conducted their experiments with the fastest update inter-arrival time set at 330 ms. In contrast, we intentionally introduce a higher level of stress during testing by reducing the update inter-arrival time to 250 ms.

Strict Priority Scheduling. Fig. 5a demonstrates that the ratio of updated flows with higher priority is consistently greater than or equal to the ratio of updated flows with lower priority. Specifically, the ratio for high-priority flows never falls below 75%, meaning that the pUpdate prototype prioritizes the updating of high-priority flows over medium and low-priority flows.

Note that due to the limited availability of link capacity, not all high-priority flows have been updated even with strict priority scheduling. In situations of high link utilization, pUpdate defers the updating of flows with insufficient resources until adequate resources become available. Furthermore, during the time period T = 600 to 1000 ms, there is sufficient time to update medium-priority flows once all possible high-priority flows have been updated and before the next update arrives.

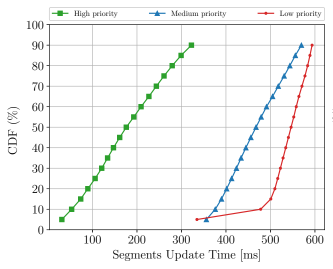
Weighted Round Robin. We used WRR scheduling algorithm with the following parameters: $N_{high} = 10$, $N_{medium} = 12$ and $N_{low} = 20$. Fig. 5b shows that the number of lower-priority updated flows is higher than in the strict priority scheduling (Fig. 5a), for example, when T = 250 ms, the ratio of updated low-priority flows is less than 10% in Fig. 5a, while in Fig. 5b, the ratio for low-priority flows is almost 20%. However, the precedence of higher-priority classes is preserved.

It is important to note that not all high priority flows are updated in Fig. 5. This can be attributed to several factors. Firstly, the arrival of a new update pauses the execution of the current update, causing any unexecuted operations to be combined with the new ones. Secondly, if there is a shortage of link capacity, deadlock resolution is triggered in, which prioritizes updating and limits the throughput of low priority flows. This can cause our architecture to spend more time on low priority flows than it would during the regular update process, leading to potential delays in updating higher priority flows.

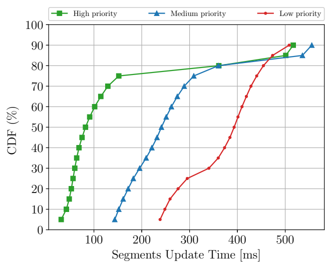
Takeaway. pUpdate adheres to priority-based scheduling and the ratio of updated flows with a higher priority is always greater than or equal to the ratio of lower priority flows.

2) Priority-based SDN Update: The purpose of this experiment is to demonstrate pUpdate's capability to schedule forwarding rules based on flow priority.

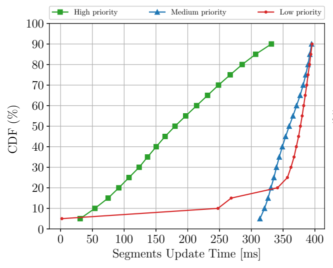
Methodology. For this experiment, we consider two topologies: FatTree and WAN topology. Each update event involves assigning a new path for all flows in the network, placing substantial pressure on the network update scheduling algorithm. The average link utilization exceeds 80%, increasing the likelihood of deadlocks and priority inversion. In this experiment, we use strict priority scheduling algorithm. Furthermore, the traffic pattern is highly imbalanced, resulting in significant load on certain links (nearly at full capacity).



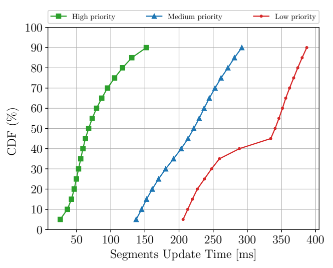
(a) T = 600 ms, #flows = 600, WAN



(b) T = 600 ms, #flows=1000, FatTree



(c) T = 400 ms, #flows = 600, WAN



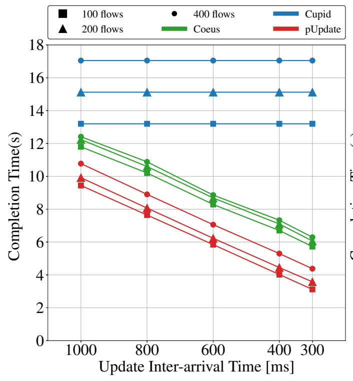
(d) T = 400 ms, #flows=1000, FatTree

Figure 6: CDF of flow update times with priority-based scheduling forwarding rule updates; WAN topology on the left, and FatTree topology on the right.

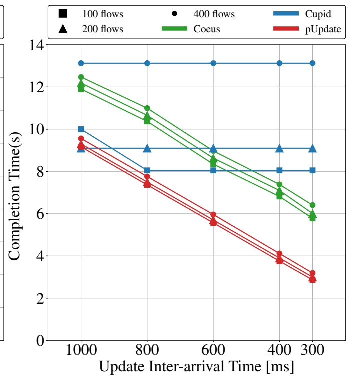
Fig. 6 shows the Cumulative Distribution Function (CDF) of flows update time. As illustrated in the figures, higher priority flows have shorter update times compared to lower priority flows, with an exception, which will be discussed below. It is important to note that high link utilization ($> 80\%$) can result in multiple calls to deadlock resolution. Consequently, this can limit the throughput of lower-priority and bandwidth-intensive flows, and cause shorter update times for some low priority flows to allocate resources to high and medium-priority flows.

Fig. 6a displays CDF in a WAN with an inter-arrival time of 600 ms for update events. As depicted in the figure, 90% of high-priority flows have received updates before lower-priority ones. Fig. 6c corresponds to an experiment with a 400 ms update inter-arrival rate, indicating that approximately 10% of the updated flows with low priority exhibit shorter update times compared to the medium-priority flows. Our analysis revealed that during the final stages of updating flows, they were contending for some shared links that were close to full utilization, and it was not possible to allocate the link's full bandwidth without reducing the bandwidth of the low priority flows.

Figs. 6b and 6d show CDF in FatTree topology. Despite the higher number of flows comparing with WAN topology, flows generally have less update time. This is because of more links and switches in FatTree topology which give pUpdate the ability to update more operations simultaneously. As shown in



(a) WAN topology



(b) FatTree topology

Figure 7: Update Completion Time with Strict priority scheduling

Fig. 6b, a minor priority inversion occurred when update time exceeds 500 ms. We observed that when $T = 600$ ms the behavior of our architecture is similar to when $T = 400$ ms until the time exceeds 400 ms. At $T = 400$ ms, an unbalanced traffic pattern causes deadlocks. To resolve the deadlocks, pUpdate had to update (and limit) the low priority flows, causing a minor priority inversion.

Takeaway. pUpdate respects flow priority, which is measured by flow update time, ensuring that flows with higher priority receive updates earlier. However, it is important to note that deadlocks can become inevitable in cases of severe link utilization, leading to priority inversion.

3) Update Completion Time Performance: The purpose of this experiment is to benchmark pUpdate against other existing solutions for achieving consistent SDN network updates in terms of update completion time. The update completion time represents the amount of time required to complete all updates.

Benchmarking. We compare pUpdate with two state-of-the-art proposals, single update from Cupid [3] and continuous update from Coeus [6]. Figs. 7a and 7b depict update completion time in WAN and FatTree topologies, respectively. The figures clearly demonstrate that pUpdate outperforms the alternative approaches in terms of update completion time. This superiority can be attributed to the advanced update algorithm integrated into pUpdate, which efficiently schedules a higher number of operations to be executed in switches in parallel, resulting in a reduced update completion time.

Takeaway. pUpdate is able to maintain flow priority without any update performance degradation in the term of update completion time in both WAN and FatTree topologies.

V. Summary

We have proposed pUpdate for achieving efficient consistent and continuous network updates in SDN by incorporating priority-based scheduling algorithms to install forwarding rules. This feature ensures that critical flows receive appropriate treatment and are given higher priority during forwarding rule installation, leading to a more optimized and efficient network. We have developed a prototype of pUpdate using Java code and evaluated its ability to handle continuous and consistent SDN update using experiment simulations. Two scheduling algorithms, namely *Strict Priority Scheduling* and *Weighted Round Robin (WRR)*, are utilized in this study. Furthermore, we have evaluated the effectiveness of pUpdate using a simulation-based testbed that utilizes a WAN topology and FatTree datacenter topology. Moving forward, we plan to extend our evaluation to include other network topologies, such as large-scale Google's B4 topology.

References

[1] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving High Utilization with Software-Driven WAN," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013, pp. 15–26.

[2] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic Scheduling of Network Updates," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 539–550, 2014.

[3] W. Wang, W. He, J. Su, and Y. Chen, "Cupid: Congestion-free Consistent Data Plane Update in Software Defined Networks," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[4] T. D. Nguyen, M. Chiesa, and M. Canini, "Decentralized Consistent Updates in SDN," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 21–33.

[5] G. Li, Y. R. Yang, F. Le, Y.-s. Lim, and J. Wang, "Update Algebra: Toward Continuous, Non-blocking Composition of Network Updates in SDN," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1081–1089.

[6] X. He, J. Zheng, H. Dai, C. Zhang, W. Rafique, G. Li, W. Dou, and Q. Ni, "Coeus: Consistent and Continuous Network Update in Software-Defined Networks," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 1509–1518.

[7] A. Rezaie Hezaveh and M. Nobakht, "On Priority-Based Scheduling for Network Updates in SDN," in *2023 IEEE 48th Conference on Local Computer Networks (LCN)*, 2023.

[8] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey, "Enforcing Customizable Consistency Properties in Software-Defined Networks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 73–85.

[9] K.-R. Wu, J.-M. Liang, S.-C. Lee, and Y.-C. Tseng, "Efficient and consistent flow update for software defined networks," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 411–421, 2018.

[10] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," *ACM SIGCOMM computer communication review*, vol. 38, no. 4, pp. 63–74, 2008.

[11] A. Networks. Arista 7500r series. [Online]. Available: https://www.arista.com/en/products/7500r-series/literature

[12] Q. Cai, M. T. Arashloo, and R. Agarwal, "dcPIM: Near-optimal proactive datacenter transport," in *ACM SIGCOMM*, 2022.

[13] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined WAN," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013.

[14] M. Kuźniar, P. Perešíni, D. Kostić, and M. Canini, "Methodology, measurement and analysis of flow table update characteristics in hardware openflow switches," *Computer Networks*, vol. 136, pp. 22–36, 2018.

[15] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, "Rules placement problem in openflow networks: A survey," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1273–1286, 2015.

[16] "Computation cluster Katana," https://researchdata.edu.au/katana/1733007, 2023, [Online; accessed 1-July-2023].