Software Defined Network's Garbage Collection with Clean-Up Packets

Md Tanvir Ishtaique ul Huque, Guillaume Jourjon, Craig Russell, and Vincent Gramoli

***Abstract*—Rule updates, such as policy or routing changes, occur frequently and instantly in software-defined networks managed by the controller. In particular, the controller software can modify the network routes by introducing new forwarding rules and deleting old ones in a distributed set of switches, a challenge that has received lots of attention in the last few years.**

In this paper, we present a problem that consists of determining the appropriate point in the rule update where it is safe to *garbage collect* old rules. To illustrate the difficulty of the problem, we list the previously proposed assumptions, like the upper-bound on the transmission delay of every packet through the network, and we offer a solution that alleviates these assumptions and significantly reduces the rule update time with a guarantee that no data packet is lost due to the rule alteration through the use of dedicated clean-up packets that detect the absence of in-flight packets. We then prove that the proposed technique guarantees *per-packet consistency*, *blackhole-freedom*, and *loop-freedom*. Our evaluations, via network emulations and real deployment in an SDN testbed, demonstrate that by using the proposed garbage collection solution the rule update times of the *two phase rule update* can be reduced by up to 99%.

***Index Terms*—SDN, Garbage collection, Consistency, Network Management, Rule update, and Asynchronous Network.**

I. Introduction

Software Defined Networking (SDN) eases the management and configuration of networks, leading to a more dynamic environment where forwarding rules can potentially be updated at a high pace. For example, OpenFlow switches commonly build upon Ternary Content Addressable Memory (TCAM) to match packets to some rule in a flow table entry. The rule update problem consists of two phases (1) installing a new forwarding rule version in the switches and (2) garbage collecting the old forwarding rule version. While various solutions were proposed to update rules [2], [3], to our knowledge there is no full-fledged solution for garbage collecting unused forwarding rules. Deleting the rule too early makes it impossible for a switch to match some packets to a flow table entry and hence leads to a packet loss or what is often called a *blackhole* [2], [4]. On the contrary, collecting the rule too late wastes the storage space of TCAM and even increases power consumption unnecessarily [5] by keeping multiple rule versions even though no in-flight packet matches the old rule version any longer.

Part of this work has been published in the IEEE Communications Magazine in 2017 [1].

M. T. I. ul Huque is with the University of New South Wales, Australia.
G. Jourjon, and C. Russell are with Data61-CSIRO, Australia.
V. Gramoli is with the University Sydney and Data61-CSIRO, Australia.

The crux of the problem is thus to decide the earliest point in the rule update execution when it is safe to garbage collect the old rule version. Time-based tentatives [6], [7] assume *synchronous* communications so that the rule update protocol knows a maximum period after which all in-flight packets will reach their destination. Other tentatives assume that paths between switches are redundant [3] or that all in-flight packets can be tracked [5]. Unfortunately, there is no way to implement these requirements practically: large-scale networks lack redundant paths between switches, too many packets can be in-transit at the same time, and their delay varies depending on external parameters, like switch firmware [8].

In this paper, we propose a path clean-up procedure to garbage collect the old version of a forwarding rule in the presence of *asynchronous* communications, where no upper-bound on the packet transmission delay is known. We complement our garbage collection technique with a rule installation procedure, similar to *two phase rule update* [9] technique, to offer a full-fledged rule update technique that guarantees *per-packet consistency* in that each packet is handled by a single rule version, *blackhole-freedom* in that no packet is dropped due to rule mismatch, and *loop-freedom* in that no loop is occurred in the network due to rule update. Note that the proposed garbage collection technique is general and could also be integrated with other rule update solutions. It is not restricted to any kind of network, as it does not assume special connectivity, instead, it can work in data center networks as well as wide area networks (WANs). The proposed technique creates "clean-up" data packets whose sole purpose is to help to detect that in-flight packets of a flow marked with the old rule version have left the core of the network. Provided that flows of a given rule version take a unique route in the core network, we guarantee that after the clean-up packet of a flow traversed the core network, the old rule version for this flow can be safely discarded. Note that this assumption prevents our solution from coping with flow splitting. At the cost of storing one additional forwarding rule per flow at some switches for a short period during garbage collection, our proposed technique guarantees that only one version of a given rule per flow is stored the rest of the time.

We demonstrate the performance improvement of the state-of-the-art method, the *two phase rule update*, using the proposed garbage collection solution, on both an emulation platform reproducing the *AGIS* network topology, and on an SDN testbed consisted of commercially available OpenFlow capable switches. Both emulated and experimental evaluations of the proposed technique indicate a significant reduction in rule update time and rule space overhead compared to the primitive form of *two phase rule update* method.[1] The proposed technique reduces the rule update time by up to 99% in case of both the *AGIS* network topology and the SDN testbed network, compared to *two phase rule update*. Moreover, both the emulation and the experimental evaluations show that the proposed technique offers two times better rule time-overhead efficiency [1] compared to *two phase rule update*.

Our key contributions in the paper are as follows:

- We illustrate the necessity of garbage collecting of old rules of the data plane of the asynchronous network which works as a motivation to have the realistic rule update technique for all types of networks (Section III).
- We introduce a garbage collection technique of forwarding rules that does not assume synchronous communications in that it does not impose an upper-bound on message transmission delay (Section IV).
- We introduce a formal model of the proposed technique capturing the well-informed behaviors of SDNs, and prove that the proposed technique guarantees *blackhole-freedom*, *loop-freedom*, and offers no data packet loss due to the path alteration of a flow (Section V).
- We evaluate the performance, in terms of the rule update time, the rule space overhead, and the rule time-overhead efficiency, of the proposed technique, on the *AGIS* emulated network topology and commercial OpenFlow switches based SDN testbed (Section VI).

II. Related Work

In this section, we survey the rule update techniques those typically consist of updating the flow table of distributed switches while guaranteeing that no rule mismatch translates into packet losses, blackholes, or loops.

Reitblatt et al. defined the rule update as the problem of updating a rule at distributed switches that applies to a flow of data packets [9]. The goal is to guarantee that either the old version or the new version is applied to each packet or even each flow but not a combination of the old and new versions. The proposed solution introduces new rule versions consistently by tagging each packet entering the network at an *ingress* switch with a version number to guarantee that each packet is always treated with the same version until it leaves the network at an *egress* switch. The authors suggest to wait for the sum of packet propagation and queuing delay, maximized over all paths before garbage collecting the old rule version. This remains difficult to implement given that rule update delays or switching performance are known to have high variances [4], [8]. As an alternative, Reitblatt et al. also recommend to wait for several seconds or even minutes [9] in practice. The problem is that waiting for too long wastes unnecessarily the limited storage space of the Ternary Content Addressable Memory (*TCAM*).

[1]The primitive form of *two phase rule update* method means the *two phase rule update* method without the asynchronous garbage collection solution. For rest of the paper, we have simply used *two phase rule update*, instead of repeating "the primitive form of *two phase rule update*" everytime.

To improve this rule update technique, Katta et al. [5] propose a sequential rule update technique that reduces the storage usage by up to $10\times$, and in that a rule can be garbage collected if it is not required. However, to do so, it requires to count all in-flight packets that remain to be migrated from the old to the new policy. While keeping track of all these in-flight packets can help garbage collecting in small networks, this may not be scalable in practice.

The proposed technique of Reitblatt et al. [9] is further developed by Mattos et al. [10]. They propose a sequential rule update technique that uses the reverse order of the flow path to update rules in switches, has less overheads, and faster, ~ 4 times, rule update time compared to Reitblatt et al. [9]. This simplified technique does not require packet tagging, however, the rule update consistency is not completely maintained.

Liu et al. [3] propose *zUpdate*, a multi-step loss-less migration technique to apply rule updates in switches. Like *zUpdate*, Jin et al. [4] and Zhou et al. [11] propose similar rule update techniques in those a global dependency graph is generated maintaining consistency among updates at different switches, and then update schedules of different switches are dynamically adjusted. Jin et al. [4] schedule updates based on the dynamic behavior of switches, a problem known as NP-complete and whose graph may be cyclic making it hard to find an ordering, and Zhou et al. [11] propose a model where confirmations of removals of forwarding links need to be delayed, and they also acknowledge the presence of in-flight packets that have been affected by rules no longer present in the network. However, using these techniques [3], [4], [11], some switches occasionally take much longer time to update the rules because of dynamic chage of network behaviour compared to other switches resulting a much longer overall rule update time. The proposed techniques of Xu et al. [12] and Gandhi et al. [13] overcome this problem. In their solutions they show that multiple alternative paths can be computed offline, and one of those paths can be chosen online to reduce the impact of the straggling switches. Although this solution reduces the rule update time in some cases, but it does not offer any certain guarantee in all cases since it works offline. Moreover, unlike our proposed technique, these solutions [3], [4], [12], [14] are limited to datacenter networks where high connectivity topology, like fat tree, can be considered. This high connectivity is necessary to offer multiple disjoint paths in order to deactivate some paths transiently during the rule update.

Other approaches proposed by Vissicchio et al. [15], [16], Brandt et al. [17] and Nguyen et al. [18] successfully alleviate the need for having multiple disjoint paths. Nguyen et al. [18] presents a decentralized, sequential rule update mechanism known as *ez-Segway* [18] in which switches are allowed to have intelligence to communicate with its neighboring switches to manage the intermediate states without any coordination from the controller, however, the controller decides the initial and the final states of the rule update. *ez-Segway* shows a significant improvement in the rule update time, up to 45%, compared to the centralized, sequential rule update mechanism of Jin et al. [4]. It however does not give the guarantee of the loss-less data packet transmission at the time of rule alteration.

The proposed solutions of Vissicchio et al. [15], [16], Brandt et al. [17] also poses the same drawback.

Like *zUpdate* and *Dionysus* [4], Wu et al. [19] proposes a sequential rule update technique, introducing network update parallelism and relaxing data packet tagging, in which a flow is partitioned into multiple segments, if possible, during the update to reduce the overall rule update time. However, the proposed technique is limited presenting the flow level analysis, and does not have packet level analysis using OpenFlow protocol supported tools.

Mizrahi et al. propose PTP [6] where the controller defines a time period during which the switches update the rule, and the ReversePTP [20] variation where switches inform the time period to the controller. Later on, Mizrahi et al. proposed a two-phase update [7] technique, followed by the k-phase update [21] technique, in which all switches update a rule following an ordered sequences specified by the controller. They compare the time complexity of usual updates to their timed update and conclude that their timed updates is faster. Like Mizrahi et al. [6], Zheng et al. [22] propose a rule update model for the synchronous networks which combines a decision algorithm to optimize the rule update time, and a greedy algorithm to find an appropriate update sequence, of switches during the update. All of these techniques [6], [7], [20]–[22] require synchronous communications, i.e., they assume an upper-bound on the time it takes for end-to-end communications between switches and controller-switch communications, an assumption that we do not need.

Other approaches [23], [24] consist of detecting and resolving conflicts between concurrent updates by implementing transactions on top of the OpenFlow protocol. These approaches builds upon key advances in the context of distributed computing but considers a slightly different problems where distributed controllers try installing conflicting rules concurrently; our problem rather focuses on installing rules from one controller to distributed switches.

III. The Rule Update Problem without Synchrony

To illustrate the limitation of the common two-phase rule update technique [2], Fig. 1 depicts a simple network topology. This network consists of two edge switches (A, B) and four core switches (C, D, E, F). Let us consider that a single rule version (or, simply a forwarding rule) installed in a switch is responsible for a single flow. Now consider f unidirectional flows originated from switch A and destined to switch B going either through switches E then F, or C then D. If these flows are evenly distributed, the rule space overhead, presented in term of number of installed rules, of edge switches (A, B) and core switches (C, D, E, F) become f and $f/2$, respectively.

Typical rule update techniques follow a generic procedure similar to the one of the original two-phase rule update technique [2]. More specifically, when a new rule version is available, this new rule version is first installed in switches. Once the new version is installed in all switches, the switches start using the new rule version. Finally, the old rule version is deleted from switches. Without loss the generality, we can identify in this procedure three distinct phases: the pre-update phase (*phase 1*), the updating phase (*phase 2*), and the post-update phase (*phase 3*). These phases are depicted in Fig. 1.

In the aforementioned network scenario, let us consider the case where all f rules are updated at the same time. In this case, the rule update is simply the route alteration of flows, i.e., the data packet flow using route $A \rightarrow E \rightarrow F \rightarrow B$ changes its route to $A \rightarrow C \rightarrow D \rightarrow B$ and vice versa. On the one hand, the rule space overhead of edge switches (A, B) is equal to the number of flows f for *phase 1* and *phase 3*, and $2f$ for *phase 2*. On the other hand, the rule space overhead of core switches C, D, E, F becomes $f/2$ for *phase 1*, f for *phase 2*, and $f/2$ for *phase 3*. This rule space overhead analysis follows, as expected, from the rule alteration happening in *phase 2*. During the rule update time (T) of *phase 2*, the rule space overhead doubles for all switches of the network. This result is network topology dependent. As the rule space overhead of rule update is directly proportional to the duration T of *phase 2*, minimizing period T reduces the time during which the storage is heavily used.

To clarify the underlying challenge further, let us consider the example of Fig. 1 with the new constraint that each switch can install a maximum of $1.5f$ forwarding rules at any given time. In that case, all f flows cannot be updated at the same time (because to update f flows at the same time, switches need to have the capacity to install $2f$ forwarding rules at once). Instead, we need to update all flows per subset of maximum size of $f/2$. So, without the successful update of the first $f/2$ rules, the second $f/2$ rules cannot be updated. It is thus crucial to trigger the removal of the unused old versions of the rule as soon as possible.

IV. Rule Update with Clean-Up Packets

In this section, we present the proposed rule update technique following the assumptions of our system model. The proposed system model does not assume synchrony, i.e., there is no upper bound on the delay it takes for a packet to be delivered, that paths between switches are redundant or that in-flight packets can be tracked. The assumptions are as follows.

1 A flow is defined with the 5-tuple $\langle$ IP source, IP destination, protocol, port source, port destination $\rangle$, where any element of the tuple can be matched with a wildcard in a forwarding rule.

2 The communication between controllers and switches is reliable in that no packets are dropped between the controller and the switch, whereas the data plane offers a best effort type of service.

3 All switches manage and forward data packets of a flow in first-in-first-out (FIFO) per-rule basis. In particular, two distinct packets treated by distinct rules can be reordered as they typically take different routes. However, the order of packets treated by the same rule version is maintained.

4 We consider that the same flow may match different rule versions but not different rules, so that distinct rules cannot conflict with each other on the same flow.
In addition, we assume that the controller logic is in charge of constructing a set of rules, and that each rule version, taken individually is loop-free.

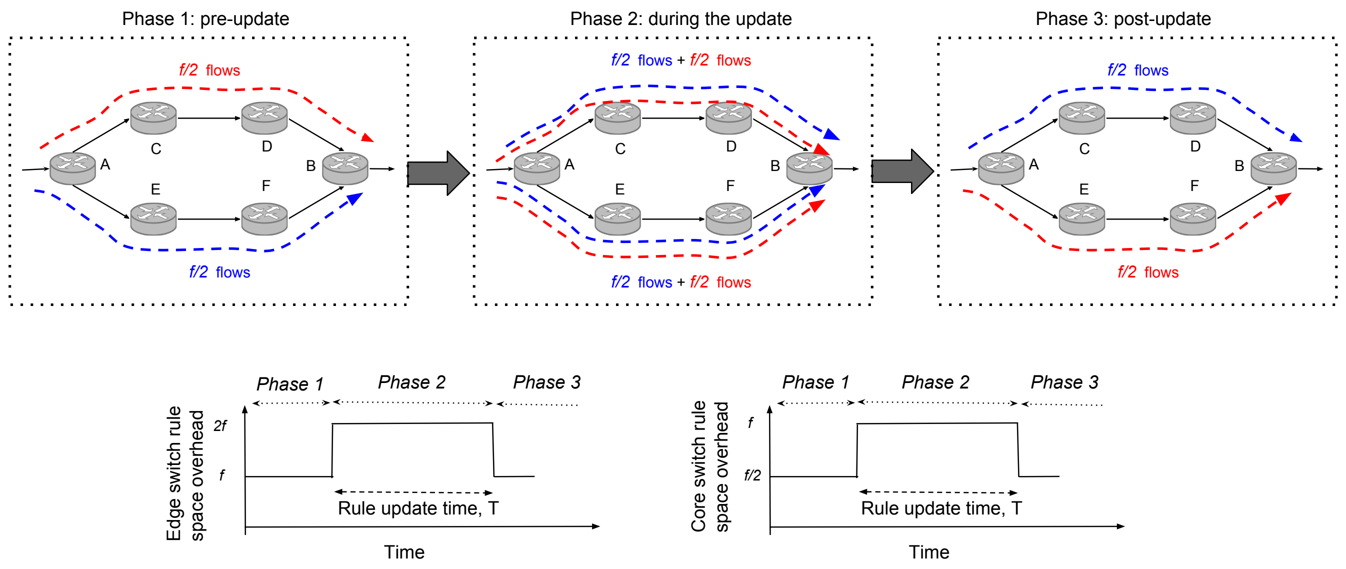


Fig. 1: Illustration of the rule update scenario

We offer a rule update technique that (i) installs the new version of the rule and (ii) garbage collects the unused version of the rule while guaranteeing network key invariants such as loop freedom, blackhole freedom and per-packet consistency. This technique minimizes both the rule space overhead and rule update time by using special "clean-up" packets and is applicable to all types of networks, e.g., sparse networks, datacenter networks or wide area networks. Point to be noted that the formation of rules is completely out of scope of the proposed technique, and, for instaling the new version of the rule it uses the same approach of the *two phase rule update* [9] technique.

The proposed rule update technique is illustrated in Fig. 3 showing it's big-picture view, and in Fig. 2 showing a partial but comprehensive view. As illustrated in both Fig. 3 and Fig. 2, the rule update technique works in four consecutive steps, i.e., *first step:* inserting the new version of a rule, *second step:* starting using this rule version, *third step:* starting the clean-up procedure, and *fourth step:* deleting the old version. Both Fig. 2 and Fig. 3 present a simplified SDN network, that consists of four switches (A, B, C and D), where two possible disjoint routes, route 1 ($A \rightarrow D \rightarrow B$) and route 2 ($A \rightarrow C \rightarrow B$), are available between switch A and switch B. This depicts a scenario where an SDN controller alternates a data packet flow from route 1 to route 2 and vice versa.

First step: Once the controller has a new version for the rule, it installs the new rule version in all necessary switches. Since each rule version manages a specific data packet flow and each flow is maintained in a single route in the network, the controller installs the new rule version in all switches of the new route (Fig. 3–$\textcircled{1}$). As an example, consider Fig. 2(a) where a data packet flow is available in route 1, and an ingress edge switch A uses $Rule$ 1 to manage this data packet flow. When the controller has the new rule version, $Rule$ 2, to forward the data packet flow through the new route, route 2, it installs $Rule$ 2 in all switches (A, C and B) of route 2 as depicted in Fig. 2(b). To keep the illustration simple, Fig. 2 presents the rule table (or, flow table) of switch A only, however, the sequential changes of the data packet flow through all switches, and the corresponding control message exchanges between the control plane and the data plane (all switches) are presented in Fig. 3. In our solution, we also follow a similar approach to Reitblatt et al. [2], where packets are marked at the ingress switch to differentiate them in both core and egress switches. Fig. 2(b) shows ingress switch A marks (tagging 2) all data packets, entering into the network, which uses $Rule$ 2.

Second step: When the new rule version is available at all switches, core switches followed by edge switches start using the new rule version as instructed by the controller. Although the edge switch has both the new and the old rule versions in its rule table, the core switch has either the new rule version or the old rule version in our example. In order to start using the new rule version, we simply set it in edge switches with a higher priority than the one of the old rule version (Fig. 3–$\textcircled{2}$). So, edge switches apply the new rule version to forward and mark the ingress traffic in a new route in the network. Note that, however, core switches in the old route can still use the old rule version when they get an in-flight data packet of the old route, to which the old rule version is applicable. As shown in Fig. 2(c), when $Rule$ 2 is available in all switches (A, C and B) of route 2, ingress switch A changes $Rule$ 2's priority to $High$ so that the ingress traffic can be forwarded to route 2.

Third step: When edge switches start using the new rule version to forward the data packet flow through a new route, the controller starts the cleaning procedure by installing an additional rule at each edge switch (Fig. 3–$\textcircled{3}$), and sending the clean-up data packet to the ingress edge switch (Fig. 3–$\textcircled{4}$). This new rule aims to forward the clean-up data packet through

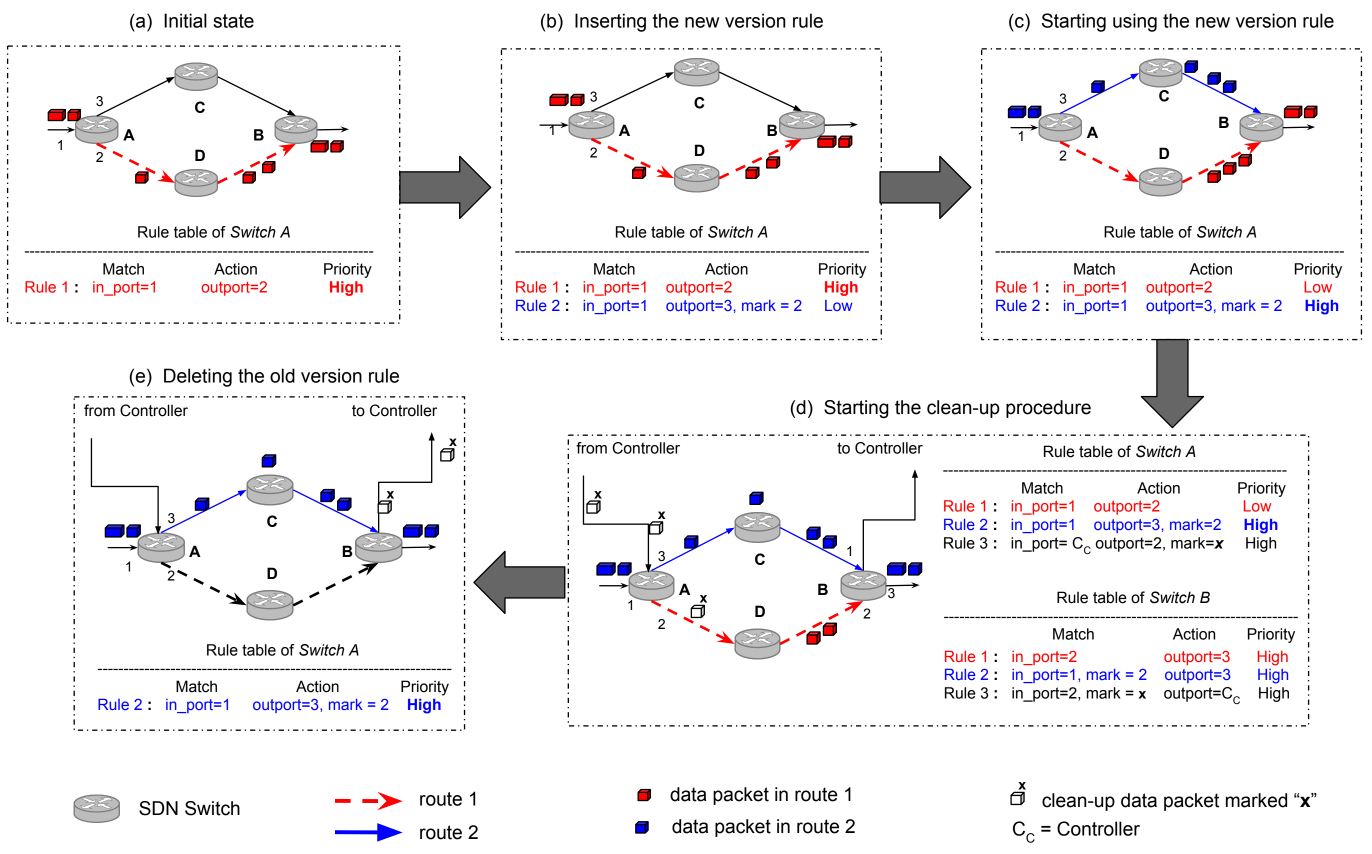

(a) Initial state
(b) Inserting the new version rule
(c) Starting using the new version rule
(d) Starting the clean-up procedure
(e) Deleting the old version rule
A
B
C
D
1
2
3
x
from Controller
to Controller
Rule table of Switch A
Rule table of Switch B
Match Action Priority
Rule 1 : in_port=1 outport=2 High
Rule 2 : in_port=1 outport=3, mark = 2 Low
Rule 1 : in_port=1 outport=2 Low
Rule 2 : in_port=1 outport=3, mark = 2 High
Rule 3 : in_port= C_C outport=2, mark=x High
Rule 1 : in_port=2 outport=3 High
Rule 2 : in_port=1, mark = 2 outport=3 High
Rule 3 : in_port=2, mark = x outport=C_C High
SDN Switch
route 1
route 2
data packet in route 1
data packet in route 2
clean-up data packet marked "x"
C_C = Controller

Fig. 2: Step-by-step workflow of the proposed rule update technique

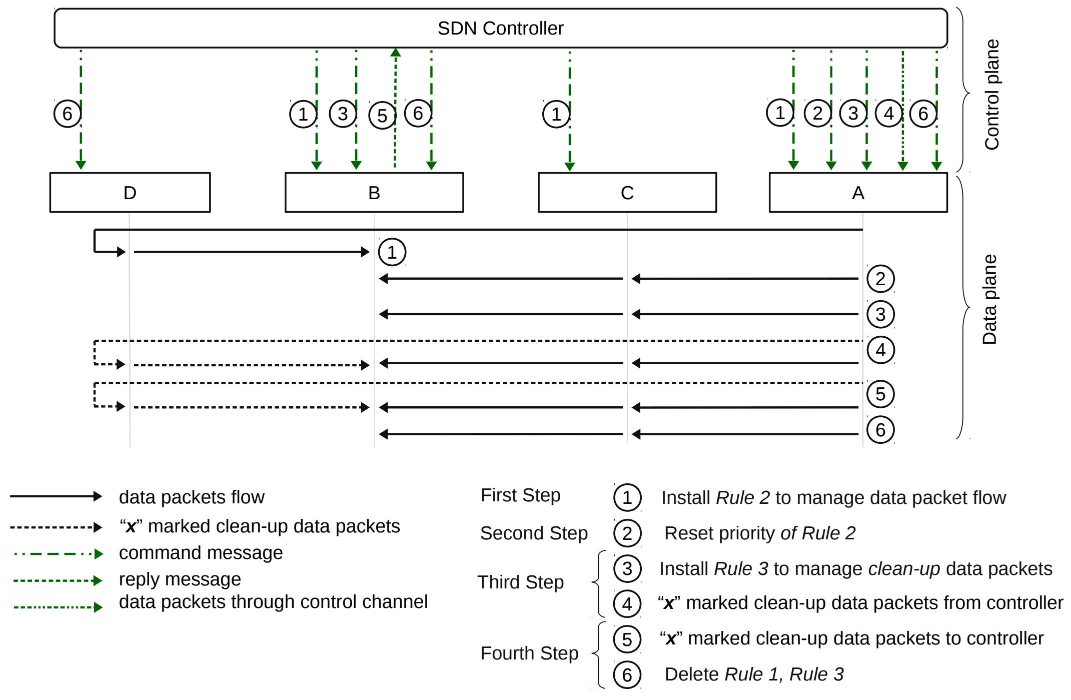

SDN Controller
D
B
C
A
1
2
3
4
5
6
Control plane
Data plane
data packets flow
"x" marked clean-up data packets
command message
reply message
data packets through control channel
First Step
Second Step
Third Step
Fourth Step
1 Install Rule 2 to manage data packet flow
2 Reset priority of Rule 2
3 Install Rule 3 to manage clean-up data packets
4 "x" marked clean-up data packets from controller
5 "x" marked clean-up data packets to controller
6 Delete Rule 1, Rule 3

Fig. 3: The sequence diagram of the proposed rule update technique

the old route. In Fig. 2(d), we see that the controller starts sending clean-up data packets continuously to switch A, and a new rule, $Rule$ 3, is installed in both switch A and switch B. $Rule$ 3 of switch A forwards clean-up data packets through the old route ($A \rightarrow D \rightarrow B$) with a known marking, whereas $Rule$ 3 of switch B forwards these marked clean-up data packets to the controller. $Rule$ 3 of edge switches (A, B) is thus used to guarantee that the clean-up data packet, sent by the controller, is also received by the controller itself. Note that, during the cleaning procedure, the in-flight data packets of the flow, passing through the old route, are managed by the already installed $Rule$ 1 of switch A, switch D and switch B.

Fourth step: As soon as the controller gets the first of the clean-up data packets from the egress edge switch (Fig. 3−⑤), it immediately stops sending clean-up data packets to the ingress edge switch and deletes the old rule version, responsible for the data packet flow in the old route, from all switches in the old route. It also deletes rules from edge switches those are used to manage the clean-up data packet flow (Fig. 3−⑥). As depicted in Fig. 2(e), when the controller gets the clean-up data packet from switch B, it (i) stops sending the clean-up data packet to ingress edge switch A, and (ii) instructs to delete the old rule version, $Rule$ 1, responsible for the data packet flow in route 1, from switches A, D and B. Edge switches (A and B) also delete $Rule$ 3 that is responsible for the clean-up data packet flow from and to the controller. At the end, only the new rule version, $Rule$ 2, becomes available in switches A, C and B to manage the data packet flow in route 2.

Using the proposed cleaning procedure, the clean-up data packet, originated from the controller and then passed through the old route, is finally received by the controller itself. Since (i) switches follow FIFO policy to manage and forward data packets and (ii) no data packet flow, except the clean-up packet flow, is sent through the old route using the old rule version, the clean-up data packet reception in the egress edge switch ensures the absence of other in-flight data packets, i.e., in-flight packets on which the old version rule is applicable, in the old route.

After deleting the old rule version, controller can simply modify the new rule version, $Rule$ 2, installed in switches of route 2, relaxing the data packet marking and matching operations to minimize the packet processing costs. To do so, controller instructs all switches of route 2, except ingress edge switch A, to modify the `Match` field of $Rule$ 2 in which the matching operation, i.e., mark= 2, is avoided; and then it modifies the `Action` field of $Rule$ 2 of the ingress edge switch A in that the marking action, i.e., mark= 2, is avoided. These instructions, to modify $Rule$ 2, are updated in switches of route 2 instantly without causing any packet loss.

V. Network Model

A. Network Model

In this section, we present the network model, that is based on the network models proposed by Reitblatt et al. [9], and Noyes et al. [25], and reuses most of the notations, terminology, and syntax used in the network models of Reitblatt et al. [9], and Mizrahi et al. [7].

We consider a SDN network consisted of a controller, and multiple packet forwarding paths (or, simply *Path*). Each packet forwarding path, i.e., $Path\_a, Path\_b, ...$, presents a subset of switches, i.e., $\mathbb{S}_a$, $\mathbb{S}_b, ...$, respectively, that manages data packet flows passing through the SDN network. Here, $\mathbb{S}_a$ is the set of switches of $Path\_a$. The formation of packet forwarding path is further discussed in *Lemma 5.1*.

A switch $S \in \mathbb{S}$, managed by the controller in the network, regulates the data packet flows passing through it. Here, $\mathbb{S}$ is the set of all switches in the SDN network. There are two types of switches: edge switches ($\mathbb{S}_E$), and core switches ($\mathbb{S}_C$). Each switch has at least a port, $P \in \mathbb{P}$. Port of a switch is used as the mean of physical inputs/outputs in which data packets of flows are queued for processing. Here, $\mathbb{P}$ is the set of all ports of switches in the network. A data packet, $p_k \in \mathbb{P}_K$, is considered as the smallest unit of a flow $f \in \mathbb{F}$; where $\mathbb{P}_K$, and $\mathbb{F}$, are sets of all packets, and flows, in the network respectively. The final source and destination of each packet of a flow are beyond the confinement of the network.

There are five different types of ports: ingress ports ($\mathbb{P}_{E_{ing}}$), egress ports ($\mathbb{P}_{E_{eg}}$), ordinary ports ($\mathbb{P}_O$), *World* port, *Drop* port. Ingress ports and egress ports are commonly known as edge ports ($\mathbb{P}_E$), i.e., $\{\mathbb{P}_{E_{ing}}, \mathbb{P}_{E_{eg}}\} \in \mathbb{P}_E$. Data packets of a flow queued at ingress ports are received from outside of the network, i.e., *World* port, at egress ports are forwarded to outside of the network, i.e., *World* port, at ordinary port are traversed in the network, and at *Drop* port are discarded from the network. Note that an edge switch with the ingress port is known as the ingress edge switch, and with the egress port is known as the egress edge switch, of the packet forwarding path.

In this network model, Controller and switch, two primary network elements, are managed by the following controller function and the switch function[2] respectively.

Switch function: The switch function $SF \in \mathbb{SF}$, presented in (4), takes a packet $p_k \in \mathbb{P}_K$ from a port $P \in \mathbb{P}$ as input and results a set of packets destined at different ports as output. Here, $\mathbb{SF}$ is the set of all possible switch functions of switches $\mathbb{S}$ in the network. For any port P and packet $p_k \in \mathbb{P}_K$, switch function $SF \in \mathbb{SF}$ of switch $S \in \mathbb{S}$ satisfies following conditions,

(i) $P \in \{\mathbb{P}_O, \mathbb{P}_{E_{ing}}\}, \rho \in \mathbb{P}\backslash\{World\}, \forall SF : SF(P, p_k) = [(\rho, p_k)]$

(ii) $P \in \mathbb{P}\backslash\{World, Drop\}, \forall SF : SF(P, p_k) = [(Drop, p_k)]$

(iii) $P \in \mathbb{P}_{E_{eg}}, \forall SF : SF(P, p_k) = [(World, p_k)]$

(iv) $P \in \{Drop\}, \forall SF : SF(P, p_k) = [\ ]$

First condition states that applying the switch function to a packet in a port (limited to *ordinary* port, and *ingress* port), the switch must generate a new packet in a different port. Although in real network, switches can generate multiple packets at different ports, for simplicity in this paper we consider that for each $< port, packet >$ pair another corresponding $< port, packet >$ pair is generated by all switches.

[2]In Section V, the term *switch function* is used to present *rule* of the switch, whereas we have used the term *rule* for rest of the paper.

$$\mathbb{S} = \{\mathbb{S}_E, \mathbb{S}_C\} = \{n \geqslant 2 : S^1,\ S^2,\ S^3, ...\ S^n\} \quad (1)$$

$$f = \{z \geqslant 1, \forall p_{k_z} \in \mathbb{P}_K, f \in \mathbb{F} : p_{k_1}, p_{k_2}, ...p_{k_z}\} \quad (2)$$

$$f_S = \{f | z \geqslant 1, \forall p_{s_z} \in \mathbb{P}_S, f_S \in \mathbb{F} : p_{s_1}, p_{s_2}, ...p_{s_z}\} \quad (3)$$

$$\mathbb{SF} := \mathbb{P} \times \mathbb{P}_K \rightharpoonup \mathbb{P} \times \mathbb{P}_K \quad (4)$$

Second condition presents that all ports, except *World* port and *Drop* port, of a switch can forward packets to *Drop* port. Third condition implies that only *egress* ports ($\mathbb{P}_{E_{eg}}$) of an edge switch in the network are allowed to forward packets beyond the confinement of the network, i.e., *World* port. Fourth condition means that if a packet is dropped, then there is no way to return it back in the network.

There is an special switch function. It is applicable on flow f_S, presented in (3), and discussed in *Definition 1*. Flow f_S is consisted of specially marked data packets, $p_s \in \mathbb{P}_S$, generated by the controller. This switch function implies that when egress ports of an edge switch S gets marked packet $p_s \in \mathbb{P}_S$, $\mathbb{P}_S$ is the set of all marked packets, of flow f_S, it immediately forwards it to the controller, i.e.,

$$\forall S \in \mathbb{S}_E, P \in \mathbb{P}_{E_{eg}}, \forall SF : SF(P, p_s) = [(Controller, p_s)]$$

Controller function: The controller is responsible to set switch functions in switches to manage data packet flows passing through the packet forwarding paths in the network. When a new packet of a new flow arrives in the ingress edge switch, the controller sets switch functions in switches of the *packet forwarding* path in the network to forward packets of that flow. To manage flow $f \in \mathbb{F}$ passing through switches $\mathbb{S}_a$ of $Path\_a$, controller function $C_f \in \mathbb{C}_f$ is used to apply the switch function $SF_a \in \mathbb{SF}_a$ in switch $S_a \in \mathbb{S}_a$ of $Path\_a$, i.e.,

$$i = [1, 2, ..N], \forall S_a \in \mathbb{S}_a, \forall SF_a \in \mathbb{SF}_a, \forall C_f \in \mathbb{C}_f : C^i_f = [(S^i_a, SF^i_a)]$$

Here, $\mathbb{S}_a$ is the set of all (N) switches of $Path\_a$, $\mathbb{SF}_a$ is the set of all switch functions of switches $\mathbb{S}_a$, and, $\mathbb{C}_f$ is the set of all controller functions.

Controller uses another function called as *controller state* function that keeps track of all flows, keeping the information of flows as *3-tuple instance* $< switch, switch\ function, flow >$, passing through switches in the network. *Controller state* function, ${}^{state}C_f$, presents that switch $S_a \in \mathbb{S}_a$, using switch function $SF_a \in SF_a$, manages flow f, i.e.,

$$i = [1, 2, ..N], \forall S_a \in \mathbb{S}_a, \forall SF_a \in \mathbb{SF}_a,$$
$$\forall {}^{state}C_f \in\ {}^{state}\mathbb{C}_f :{}^{state}C^i_f = [(S^i_a, SF^i_a, f)$$

Here, ${}^{state}\mathbb{C}_f$ is the set of all *controller state* functions.

Although in the network a flow can pass through multiple *packet forwarding* paths, in this paper we consider that one flow passes through a *packet forwarding* path at a given time. However, multiple flows can pass thorough a *packet forwarding* path at a time. Based on the aforementioned network model, we have proposed a rule update model, that is a simplified formal model of the proposed rule update technique presented in Section IV, for the SDN network. The objective of the rule update model is to offer a way to change the *packet forwarding* path of a flow with a guarantee that no data packet is lost due to the path alteration of that flow. The formation of switch functions and the bandwidth constraints of the *packet forwarding* path in the network are out of scope of this rule update model.

This proposed rule update model is based on two assumptions (*Axiom 1, and Axiom 2*), and takes into account of five propositions (*Lemma 5.1, Lemma 5.2, Lemma 5.3, Lemma 5.4, and Lemma 5.5*). Finally, it is shown that the proposed rule update model is *black hole* free (*Corollary 1 of Theorem 5.6*), *loop* free (*Corollary 1 of Theorem 5.6*), and there is no data packet loss due to the transition of the *packet forwarding* path of a flow in the network (*Theorem 5.7*).

In the proposed model, *Axiom 1*, *Axiom 2* represent Assumptions *3* and *4*, discussed in Section IV, respectively; *Lemma 5.1*, *Lemma 5.2* specify the network topology on which the rule update model is applicable; and, *Lemma 5.3*, *Lemma 5.4*, *Lemma 5.5* maintain the execution order of the rule update model. Pointedly, *Lemma 5.3* represents *Second step* ($Fig.$ 3-②) of the proposed technique, *Lemma 5.4* ensures the execution order of *Second step* and *Third step* of the proposed technique that *Second step* must come before *Third step*, *Definition 1 (Path clean-up)* represents *Third step* ($Fig.$ 3-③,④), completely, and *Fourth step* ($Fig.$ 3-⑤,⑥), partially, of the proposed technique, discussed in Section IV. *Definition 1 (Path clean-up)* along with *Lemma 5.5* complete the representation of *Fourth step* ($Fig.$ 3-⑤,⑥) of the proposed technique.

TABLE I: The key notations of the rule update model

$\mathbb{S}_a$	The set of all switches of $Path\_a$
$\mathbb{S}_b$	The set of all switches of $Path\_b$
S^1_a	The ingress edge switch of $Path\_a$
S^N_a	The egress edge switch of $Path\_a$
S^1_b	The ingress edge switch of $Path\_b$
S^N_b	The egress edge switch of $Path\_b$
$t^f_{S^i_a,S^{i+1}_a}$	The traffic load, due to flow f, between two consecutive switches S^i_a and S^{i+1}_a of the packet forwarding path
$L_{S^i_a,S^{i+1}_a}$	The link between two consecutive switches S^i_a and S^{i+1}_a of the packet forwarding path. If `True`, then switches are connected.

B. Rule Update Model:

In this model, rule update (R) is used to change the packet forwarding path of a flow in the network and presented as *3-tuple* $< f, C_f, Cmd >$, where C_f is the controller function to manage flow f in the network and Cmd is the action command applicable on that rule update. There are three different action commands: *push.Cmd*, *apply.Cmd*, and *delete.Cmd*, where *push.Cmd* is used to push new switch functions in switches, *apply.Cmd* is used to activate the newly pushed switch function in switches, and *delete.Cmd* is used to delete the switch functions from switches.[3] For these three action commands, there are three different rule updates ($R \in \mathbb{R}$): R^p for *push.Cmd*, R^a for *apply.Cmd*, and R^d for *delete.Cmd*.

For a better understanding of the proposed rule update model let's consider flow f is passing through switches $\mathbb{S}_a$ of $Path\_a$ in the network. The objective is to change the *packet forwarding* path, from $Path\_a$ to $Path\_b$, of flow f using the proposed rule update model. Now, to alter the *packet forwarding* path, from $Path\_a$ to $Path\_b$, of flow f, at first, the rule update R^p is applied. R^p simply pushes, but does not activate, switch functions ($\mathbb{SF}_b$) installed in switches ($\mathbb{S}_b$) of $Path\_b$ to manage flow f in $Path\_b$. The newly pushed switch functions of $Path\_b$ become active when the rule update R^a is applied. After applying R^a, flow f starts passing through switches of $Path\_b$ (*Lemma 5.3*). But, there are still in-flight data packets of flow f in $Path\_a$. Then path clean-up event ($pClean_a$) for flow f in $Path\_a$ is applied, which guarantees that no data packet of flow f is present in $Path\_a$ (*Theorem 5.7*). Finally, the rule update R^d is applied to delete switch functions responsible to manage flow f of switches $\mathbb{S}_a$ of $Path\_a$ (*Lemma 5.5*). The alteration of packet forwarding path, from $Path\_a$ to $Path\_b$, of flow f using the rule update model is depicted below.

$$\xrightarrow[Path\_a]{} R^p \xrightarrow[\substack{Path\_a\\Path\_b}]{} R^a \xrightarrow[\substack{Path\_b\\Path\_a}]{pClean_a} R^d \xrightarrow[Path\_b]{}$$

In the depiction, *blue* marked path is used to present the active packet forwarding path of flow f at a given time. The illustration shows that, after applying rule update R^a, flow f starts passing through switches of $Path\_b$, however, there are in-flight data packets of flow f in $Path\_a$. Since, the use of the path clean-up event gives the guarantee of the absence of the data packets of flow f in $Path\_a$, the deletion of switch functions, following the path clean-up event, of switches of $Path\_a$ does not cause any data packet drop of flow f in $Path\_a$. Therefore, no data packet is lost due to the path alteration of a flow because of applying the proposed rule update model.

Axiom 1: All packets are processed and forwarded by switches first-in-first-out (FIFO) basis in the network.

Axiom 2: A switch function of a switch is responsible to manage only one flow in the network, i.e.,

$$\forall S \in \mathbb{S}, \forall SF \in \mathbb{SF}, \forall f \in \mathbb{F} : f_1 \neq f_2 \rightarrow SF^1 \neq SF^2$$

It presents that switch functions SF^1, and SF^2 are responsible to manage flow f_1, and f_2 respectively. This assumption is used to confine the property of switch functions.

Definition 1 (Path clean-up): Path clean-up ($pClean$) event is used to check the absence of in-flight data packet of a flow in the *packet forwarding* path in the network. $pClean$ consists of applying three consecutive actions in the network: *firstly*, it installs rules, responsible to manage flow f_S, in both edge switches of the *packet forwarding* path, *secondly* it allows the controller to send data packet flow f_S in that *packet forwarding* path, and *thirdly*, it deletes installed rules in both edge switches whenever the data packet of flow f_S is detected in the egress edge switch of the *packet forwarding* path. For a better understanding let's consider, flow f has started passing from $Path\_a$ to $Path\_b$ in the network. Path clean-up event of flow f in $Path\_a$ ($pClean_a$) is performed by applying following three successive actions.

(i) $i = [1, N], C^i_{f_S} = (S^i_a, SF^i_a), R^p := (f_S, C^i_{f_S}, push.Cmd),$
$R^a := (f_S, C^i_{f_S}, apply.Cmd) : (R = R^p) \rightarrow (R = R^a)$

(ii) $i = [1, N], C^i_{f_S} = (S^i_a, SF^i_a), R^a := (f_S, C^i_{f_S}, apply.Cmd) :$
$(R = R^a) \rightarrow inject(f_S, S^1_a)$

(iii) $i = [1, N], C^i_{f_S} = (S^i_a, SF^i_a), R^d := (f_S, C^i_{f_S}, delete.Cmd) :$
${}^{state}C^N_{f_S} = (S^N_a, SF^N_a, f_C) \rightarrow \neg inject(f_S, S^1_a) \wedge (R = R^d)$

The first action implies that the controller pushes and then activates switch functions, by applying rule updates R^p and then R^a, in both ingress and egress edge switches of $Path\_a$ to manage flow f_S. Rule updates R^p and R^a installs and activates switches functions SF^1_a and SF^N_a in switches S^1_a and S^N_a of $Path\_a$ respectively to manage flow f_S. Here, the specially marked data packet flow f_S is generated by the controller in such a way that switch functions already installed in core switches of $Path\_a$ to manage flow f can also be applied on it. Flow f_S is used only for path clean-up event in our proposed model.

The second action states that, after applying rule update R^a, the controller starts sending flow f_S to the ingress edge switch S^1_a of $Path\_a$. Here, $inject(f_S, S^1_a)$ is an event which injects the data packet flow f_S from controller to the ingress edge switch S^1_a of $Path\_a$ in the network [26].

The third action presents that if the controller gets the data packet of flow f_S in the egress edge switch S^N_a of $Path\_a$ (${}^{state}C^N_{f_S} = (S^N_a, SF^N_a, f_S)$), it stops sending data packets of flow f_S to the ingress edge switch S^1_a ($\neg inject(f_S, S^1_a)$) and applies rule update R^d ($R = R^d$) immediately. Here, rule update R^d is used to delete switch functions SF^1_a and SF^N_a of switches S^1_a and S^N_a of $Path\_a$ respectively; and ${}^{state}C^N_{f_S} = (S^N_a, SF^N_a, f_S)$ says that the data packet of flow f_S is available in switch S^N_a using the switch function SF^N_a.

Lemma 5.1: A subset of switches, say, $\mathbb{S}_a = \{S^1_a, S^2_a, ...S^N_a\}$, *is said to be a packet forwarding path, Path_a, if it, i.e.,* $\mathbb{S}_a$, *satisfies following two conditions,*

(i) *A packet forwarding path starts and ends with the ingress/egress edge switches*, i.e,

$$j = \{|j| \geq 0 : [2, 3, ..(N-1)]\}, \{S^1_a, S^N_a\} \in \mathbb{S}_E,$$
$$\forall S_a \in \mathbb{S}_a : \mathbb{S}_a = \{S^1_a, S^j_a, S^N_a\}$$

(ii) *All switches of a packet forwarding path are connected consecutively*, i.e.,

$$i = \{N \geq 2 : [1, 2, ..N]\}, \forall S_a \in \mathbb{S}_a : L_{S^i_a, S^{i+1}_a} = True$$

[3]OpenFlow protocol has $OFPIT\_WRITE\_ACTIONS$, $OFPIT\_APPLY\_ACTIONS$, and $OFPIT\_CLEAR\_ACTIONS$ commands, and these commands are used to push, activate, and delete switch function in switches respectively.

First condition states that the packet forwarding path has, at least, two ingress/egress edge switches (S^1_a, S^N_a), and other switches of it can be either the edge switch or the core switch. Second condition presents that in a packet forwarding path, a switch is connected to its next switch, i.e., in $Path\_a$, S^1_a is connected to S^2_a, S^2_a is connected to S^3_a, and so on.

Lemma 5.2: The transition of the packet forwarding path, from $Path\_a$ to $Path\_b$, of flow f in the network is feasible if and only if the paths have the same ingress and egress edge switches, i.e.,

$$\mathbb{S}_a = \{S^1_a, \mathbb{S}^2_a, ..S^N_a\}, \mathbb{S}_b = \{S^1_b, \mathbb{S}^2_b, ..S^N_b\} : S^1_b = S^1_a, S^N_b = S^N_a$$

Lemma 5.3: During the transition of the packet forwarding path, from Path_a to Path_b, of flow f in the network, flow f starts passing through Path_b, instead of path_a, if rule update R^a has already been applied, i.e.,

$$R^a := (f, \mathbb{C}_f, apply.Cmd), \mathbb{C}_f = (\mathbb{S}_b, \mathbb{SF}_b) : (R = R^a) \rightarrow \\ (\,^{state}C^1_f \neq (S^1_a, SF^1_a, f)) \wedge (\,^{state}C^1_f = (S^1_b, SF^1_b, f))$$

Rule update R^a activates switches functions $\mathbb{SF}_b$ of switches $\mathbb{S}_b$ of $Path\_b$ to manage flow f. After applying R^a, flow f starts passing through switches of $Path\_b$. In that case, $^{state}C^1_f = (S^1_b, SF^1_b, f)$, i.e., flow f is available in the ingress edge switch $S^1_b \in \mathbb{S}_b$ using the switch function $SF^1_b \in \mathbb{SF}_b$. On the contrary, after applying R^a, no data packets of flow f goes to $Path\_a$. In that case $^{state}C^1_f \neq (S^1_a, SF^1_a, f)$, i.e., there is no data packets of flow f in the ingress edge switch $S^1_a \in \mathbb{S}_a$ using switch function $SF^1_a \in \mathbb{S}_a$.

Lemma 5.4: Path clean-up is applied if rule update R^a has already been applied in the network, that is,

$$R^a := (f, \mathbb{C}_f, apply.Cmd), \mathbb{C}_f = (\mathbb{S}_b, \mathbb{SF}_b) : \\ (R = R^a) \rightarrow (pClean = pClean_a)$$

It states that during the transition of the packet forwarding path, from $Path\_a$ to $Path\_b$, of flow f, *path clean-up* of flow f in $Path\_a$ ($pClean_a$) can only be applied if rule update R^a has already been applied in the network.

Lemma 5.5: Rule update R^d is applied if path clean-up event has already been applied, i.e.,

$$R^d := (f, \mathbb{C}_f, delete.Cmd), \mathbb{C}_f = (\mathbb{S}_a, \mathbb{SF}_a) : \\ (pClean = pClean_a) \rightarrow (R = R^d)$$

It presents that during the transition of *packet forwarding path*, from $Path\_a$ to $Path\_b$, of flow f, rule update R^d is applied if path clean-up event of flow f in $Path\_a$ ($pClean_a$) has already been conducted. Rule update R^d deletes switch functions $\mathbb{SF}_a$, managing flow f in $Path\_a$, of switches $\mathbb{S}_a$.

Theorem 5.6: A packet forwarding path, say $\mathbb{S}_a$, in the network is black hole free and loop free if it satisfies following two conditions,

(i) *A flow, say f, entering into a switch, say S^i_a, of a packet forwarding path, $\mathbb{S}_a$, is fully delivered to its next switch, S^{i+1}_a, of that packet forwarding path*, i.e.,

$$i = [1, 2, ..N], \forall S_a \in \mathbb{S}_a, \forall f \in \mathbb{F} : t^f_{S^i_a, S^{i+1}_a} = t^f_{S^{i+1}_a, S^{i+2}_a}$$

(ii) *The switch functions are installed in edge switches (S^1_a, S^N_a) followed by core switches $(S^2_a, S^3_a, ..S^{N-1}_a)$ of the path forwarding path* ($\mathbb{S}_a$), i.e.,

$$j = [2, 3, ..(N-1)], \forall C_f \in \mathbb{C}_f = (\forall S_a \in \mathbb{S}_a, \forall SF_a \in \mathbb{SF}_a) : \\ C^j_f \rightarrow (C^1_f \wedge C^N_f)$$

First condition presents that for all switches $\mathbb{S}_a$ of $Path\_a$, traffic load, due to flow f, entering into switch S^{i+1}_a, from switch S^i_a, is equivalent to traffic load going into switch S^{i+2}_a, from switch S^{i+1}_a. Here, S^i_a, S^{i+1}_a, and S^{i+2}_a are three consecutively connected switches of $Path\_a$. This condition indirectly satisfies the network property that the traffic load entering into a switch is equivalent to the traffic load going out of that switch of the packet forwarding path.
Second condition states that to manage flow f in switches $\mathbb{S}_a$ of $Path\_a$, controller functions C^1_f, C^N_f are executed after executing controller functions C^2_f,C^3_f, $..C^{N-1}_f$. Controller functions C^1_f and C^N_f install ingress and egress edge switch functions SF^1_a and SF^N_a in ingress and egress edge switchs S^1_a and S^N_a of $Path\_a$ respectively. On the contrary, C^2_f,C^3_f, $..C^{N-1}_f$ install switch functions SF^2_a,SF^3_a, $..SF^{N-1}_a$ in switches S^2_a,S^3_a, $..S^{N-1}_a$ of $Path\_a$ respectively. This condition ensures that a flow always encounters the correct switch functions in switches of a packet forwarding path.

Proof (by contradiction): Let's consider, $\mathbb{S}_a = \{S^1_a, S^2_a, ...S^N_a\}$ is a set of N switches of $Path\_a$ in the network, and flow f is passing through the switches of $Path\_a$. All of these switches ($\mathbb{S}_{P_a}$) of $Path\_a$ have the common property that (i) flow f entering into a switch of $Path\_a$ is fully delivered to its next switch of that path, (ii) switch functions are installed in the edge switches (S^1_a, S^N_a) followed by the core switches $(S^2_a, S^3_a, ..S^{N-1}_a)$ to manage flow f of $Path\_a$.

Now, assume that there is a *black hole*, or, a *loop*, in $Path\_a$. Because of having a *black hole*, or, a *loop*, due to flow f, the traffic load entering into $Path\_a$ can not be equivalent to the traffic loads going out of $Path\_a$. In this case, $\{S^1_a, S^N_a\} \in \mathbb{S}_a, \forall f \in \mathbb{F} : t^f_{world, S^1_a} \neq t^f_{S^N_a, World}$, where t^f_{world, S^1_a} is the traffic load, due to flow f, from `World` port to switch S^1_a and $t^f_{S^N_a, World}$ is the traffic load, due to flow f, from switch S^N_a to `World` port. But, we know that (i) switches $\mathbb{S}_a$ of $Path\_a$ are sequentially connected ($lemma$ 5.1), (ii) a switch of $Path\_a$ delivers the traffic load of flow f to its next switch, (iii) egress edge switch S^N_a of $Path\_a$ is only allowed to forward the flow f outside of the network (*Switch function*, *Section* V-A), and (iv) flow f does no encounter any incorrect switch functions in switches of $Path\_a$ so that data packet cannot be dropped or misguided. Therefore, , due to flow f, the traffic load entering into $Path\_a$ is equivalent to the traffic load going out of $Path\_a$. it is not possible to have a *black hole*, or a loop in $Path\_a$. It shows contradiction with the assumption, and reveals that *packet forwarding* paths are *black hole* free and loop free in the network. □

Corollary 1 of Theorem 5.6: The path clean-up event is black-hole free, and loop free.

Path clean-up event of flow f in $Path\_a$ ($pClean_a$) sends flow f_S in the ingress edge switch S_a^1 of $Path\_a$, and detects its presence in the egress edge switch S_a^N of $Path\_a$ (*Definition 1*)- it verifies that data packets of flow f_S passes through $Path\_a$. Since switches of $Path\_a$ are consecutively connected ($lemma$ 5.1) and flow f_S passes through $Path\_a$, a switch forwards flow f_S to the next switch of $Path\_a$. It satisfies the first condition of *Theorem 5.6*

Again, the path clean-up event installs the switch functions in edge switches followed by core switches in the packet forwarding path (*Definition 1*) $-$ it satisfies the second condition of *Theorem 5.6*. Note that to manage data packets of flow f_S, path clean-up event of flow f in $Path\_a$ ($pClean_a$) installs the new switch functions in edge switches (S_a^1, S_a^N), and uses the pre-installed switch functions rather than installing the new switch functions in core switches $(S_a^2, S_a^3, ..S_a^{N-1})$ in the $Path\_a$. Since, the path clean-up event in the network satisfies both conditions of *Theorem 5.6*, the path clean-up event is black hole free, and loop free.

Theorem 5.7: During the transition of the packet forwarding path, from the old path to the new path, of a flow, the application of path clean-up (pClean) event in the old path guarantees the absence of data packets of that flow in that path, i.e.,

$$i = [1, 2, ..N], \forall S_a \in \mathbb{S}_a, \forall SF_a \in \mathbb{SF}_a, \forall {}^{state}C_f \in {}^{state}\mathbb{C}_f : \\ (pClean = pClean_a) \rightarrow ({}^{state}C_f^i \neq (S_a^i, SF_a^i, f))$$

It presents that during of the transition of the *packet forwarding path*, from $Path\_a$ to $Path\_b$, of flow f, there is no data packet of flow f in switches $\mathbb{S}_a$, using switch functions $\mathbb{SF}_a$, of $Path\_a$, i.e., ${}^{state}C_f \neq (S_a, SF_a, f)$, if *path clean-up* event of flow f in $Path\_a$ has already been applied, i.e., $pClean = pClean_a$. It offers the per packet consistency property [9] of rule update in the network.

Proof (by contradiction): Let's consider, switches $\mathbb{S}_a$ of $Path\_a$ in the network use switch functions $\mathbb{SF}_a$ to manage flow f passing through that path. Now consider that during the change of the packet forwarding path, from $Path\_a$ to $Path\_b$, *path clean-up* ($pClean_a$) event has already been applied in $Path\_a$. Assume that after applying $pClean_a$, there is data packet of flow f in $Path\_a$ in the network.
We know that $pCheck_a$ is applied when the rule update R^a has already been applied in the network (*Lemma 5.4*). After applying R^a, flow f starts passing through switches of $Path\_b$, instead of $Path\_a$, in the network (*lemma 5.3*). So, when $pClean_a$, following R^a, is applied, no new data packet of flow f goes to $Path\_a$, i.e., ${}^{state}C_f^1 \neq (S_a^1, SF_a^1, f)$ (*lemma 5.3*, *lemma 5.4*). Again, $pClean_a$ sends data packet flow f_S in $Path\_a$ (*Definition 1*), and switches of $Path\_a$ forward the data packets of flow f before forwarding the data packets of flow f_S (*Axiom 1*). So, we can say that if egress switch of $Path\_a$ gets data packet of flow f_S, it ensures that no packets of flow f is left in $Path\_a$ (*Definition 1, and Axiom 1*). Thus there is no data packet of flow f in $Path\_a$, and it shows a direct contradiction with the assumption. Thus $pClean_a$ guarantees the absence of the data packet of flow f in $Path\_a$. □

VI. Implementation and Performance Evaluation

In this section we have presented the comparative performance analysis of the proposed solution to the rule update technique of Reitblatt et al. [9], addressed as *two phase rule update*[1] technique. The performance of both the proposed technique and the two phase rule update technique were evaluated based on the following three metrics,

- *Rule update time*: Rule update time of a switch is defined as the time period which starts with the insertion of the new version of a rule, and ends with the deletion of the old version of that rule in the switch.
- *Rule space overhead*: Rule space overhead of a switch is defined as the number of rules set in it's rule-space, i.e., ternary content addressable memory.
- *Rule time-overhead efficiency*: Rule time-overhead efficiency (E_R) of a switch using a specific rule update technique is defined in (5) [1]. This metric presents the cost, considering both the rule update time and the rule space overhead, of using a rule update technique for each switch of the network. The lower value of E_R of a switch for a rule update technique presents the lower level of efficiency of that rule update technique, and vice versa. For a given rule update technique, RUT and RSO are the rule-update time and the rule-space overhead of the switch, and RUT_{max} and RSO_{max} are the maximum observed rule-update time and rule-space overhead of the network, respectively.

$$E_R = \left(1 - \frac{RUT \times RSO}{RUT_{max} \times RSO_{max}}\right) \tag{5}$$

We considered two different scenarios, presented below, to show the performance evaluations.

1 In *Emulation-based Evaluation*, we considered the system model of Huque et al. [1] in which *AGIS* topology was considered to evaluate the performance of the two-phase rule update technique and the proposed technique.
2 In *Experimental Evaluation*, a SDN testbed, consisted of four Allied Telesis AT-x930-28gTX switches [27], was considered to show the comparative performance analysis of both of these rule update techniques.

In the following performance evaluation, at time $t = 10sec$, we initiated the rule update in case of both the proposed technique and the *two phase rule update* technique in the network. We then reverted to the original rule set at $t = 170sec$. Please note that, the *two phase rule update*[1] technique cannot detect when an old rule version can be deleted and thus deletes a rule after a fixed period of time of 2 minutes, following the

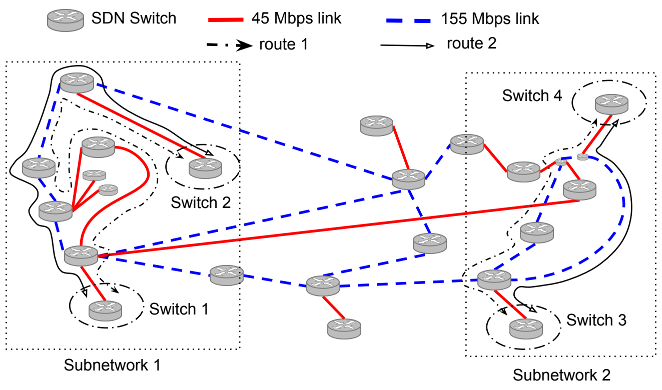


Fig. 4: *AGIS* network topology

recommendation from [2] and assuming that no packets are in-transit after that.[4]

A. Emulation-based Evaluation

1) Emulation Set-up: In the emulation, we used the *Mininet* virtual network, *Ryu* controller framework [28], the Open vSwitch (ver 2.10) to run the *AGIS* network topology [29]. We run our emulations on an Intel(R) Core(TM) i7-5600U CPU at 2.60GHz with Ubuntu 14.04LTS and 8GB RAM.

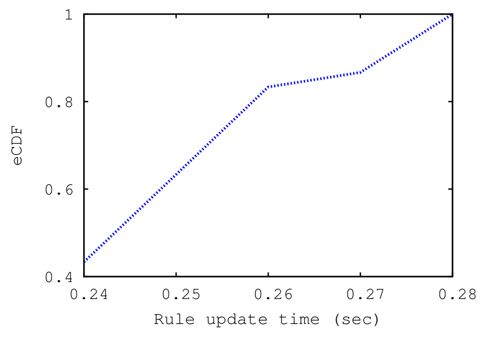
In *AGIS* network, depicted in Fig. 4, we considered two subnetworks (subnetwork 1 and subnetwork 2), each subnetwork having two routes (route 1, route 2) to forward data packet flows. The emulation was consisted of sending a data packet flow from switch 1 to switch 2 of subnetwork 1, and from switch 3 to switch 4 of subnetwork 2, using either route 1 or route 2 at the maximum limit of the corresponding links.

2) Emulation-based Result Analysis:

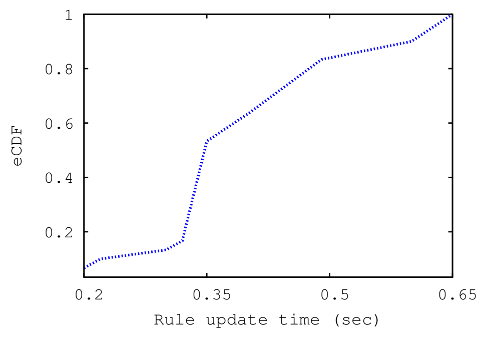
Rule update time: Fig. 5(a) presents the empirical cumulative distribution function (eCDF) of the rule update time of the switches of the *AGIS* network. The rule update from route 1 to route 2 (or, simply the change of packet forwarding path of a data packet flow from route 1 to route 2) means the data packet flow was passing through route 1 and, because of applying the rule update technique, the data packet flow has started passing through route 2 instead of route 1, and vice versa. We got the rule update time of switches of *AGIS* network, shown in Fig. 5(a), in the range of $0.24sec$ to $0.28sec$, which 75^{th} $percentile$ value is $0.255sec$. We also observed that the rule update time of all switches, both the edge switches and the core switches, of a route of the network was same. This emulation-based result presents that because of having the efficient garbage collection technique, the proposed technique offers almost 99% less rule update time to switches compared to the *two phase rule update*[1] technique.

Rule space overhead: In our experiment, we found that at the time of the rule update from route 1 to route 2 (or, from route 2 to route 1) of subnetwork 1 (and, subnetwork 2) of the *AGIS* network, the rule space overheads of the edge

[4]Due to our asynchronous communications hypothesis, we followed the recommendation of Reitblatt et al. [2], "*....the controller can be quite conservative in estimating the delays and simply wait for several seconds (or even minutes) before removing the old rules*" and configured the deletion timeout of the two-phase rule update technique to 2 minutes.



(a) Rule update time of the switches, from route 1 to route 2 (or, from route 2 to route 1), of subnetwork 1 (and, subnetwork 2) of the *AGIS* network.



(b) Rule update time of the switches, from route 1 to route 2 (or, from route 2 to route 1), of the SDN testbed network.

Fig. 5: Rule update time of the network

switches were 2, using the *two phase rule update* technique, and 3, using the proposed technique. We also found that during the rule update from route 1 to route 2 (or, from route 2 to route 1) of subnetwork 1, the rule space overheads of the core switches of the old route were 9, using both the proposed technique and the *two phase rule update* technique. Again, for subnetwork 2, during the rule update from route 1 to route 2 (or, from route 2 to route 1), the rule space overheads of the core switches of the old route were 6, using both of the techniques. Therefore, by using the proposed technique, for updating the rule from the route 1 to route 2 (or, from route 2 to route 1), the edge switches of route 1 (or, route 2) of the *AGIS* network required to hold one additional rule during the rule update time. It was due to conducting the garbage collection technique as explained in Section V-B. However, because of offering the shorter rule update time, *i.e.*, $0.255sec$, compared to the *two phase rule update* technique, the proposed technique minimized the overall rule space overhead of the switches, which can be understood easily noticing the rule time-overhead efficiency metric.

Rule time-overhead efficiency: The rule time-overhead efficiencies of the switches of the *AGIS* network using both the proposed technique and the *two phase update*[1] technique are listed in Table II, where $RUT_{max} = 120sec$ and $RSO_{max} = 3$. Table II shows that for all switches, the proposed technique offers the higher rule time-overhead efficiency compared to the *two phase update* technique. We got that the rule time-overhead efficiency of the edge switches of subnetwork 1 (and,

TABLE II: The rule time-overhead efficiency of the switches of subnetwork 1 and subnetwork 2 of the *AGIS* network

Rule update type	Switch	Proposed technique	Two-phase rule update technique
Update 1 → 2	Edge	99%	33%
	Core	99%	33% − 66%
Update 2 → 1	Edge	99%	33%
	Core	99%	33% − 66%

Update 1 → 2 : Rule update from route 1 to route 2
Update 2 → 1 : Rule update from route 2 to route 1

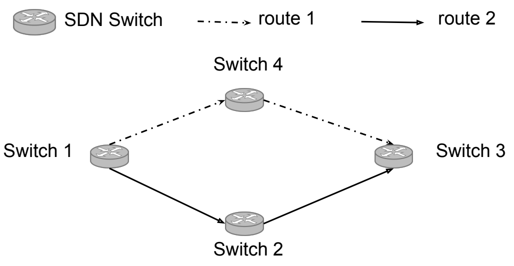


Fig. 6: The SDN testbed topology

subnetwork 2), during the rule update from route 1 to route 2 (or, from route 2 to route 1), were $[(1-\frac{0.255\times3}{120\times3})100\%] = 99\%$, using the proposed technique, and $[(1-\frac{120\times2}{120\times3})100\%] = 33\%$, using the *two-phase rule update* technique.

We found that, during the rule update, edge switches of the *AGIS* network held same number of rules, whereas different core switches held different number of rules. As an example, in subnetwork 2 during the rule update from route 1 to route 2, each edge switch held 3 rules but the core switches of route 1 based on their position in the network topology held either 2 rules or 3 rules in their rule-space, using the proposed technique, resulting the rule time-overhead efficiency to almost 99%. In contrast, during the rule update, from route 1 to route 2, using the *two-phase rule update* technique, the rule time-overhead efficiency of the core switch of route 1 having 1 rule and 2 rules became 66% and 33% respectively. Hence, we can say that the listed results of Table II lead us to the conslusion that the proposed technique offers around 2 times better rule time-overhead efficiency compared to the *two-phase rule update* technique.

B. Experimental Evaluation

1) Experimental Set-up: A SDN testbed, shown in Fig. 6, was consisted of a Ryu controller and four Allied Telesis AT-x930-28gTX switches, in which any two switches were connected by a link having capacity of 1 *Gbps*. Ryu controller was run on a virtual machine (VM), the specification of the VM was Ubuntu 14.04LTS, 1GB RAM and 1 VCPU, in an OpenStack cloud. In this network topology, we considered switch 1 sent the data packet flow to switch 3 using either route 1 (switch 1 → switch 4 → switch 3) or route 2 (switch 1 → switch 2 → switch 3) at the maximum limit of the corresponding links.

2) Experimental Result Analysis:

Rule update time: Fig. 5(b) presents the empirical cumulative distribution function (eCDF) of the rule update time

TABLE III: The rule time-overhead efficiency of the switches of the SDN testbed network

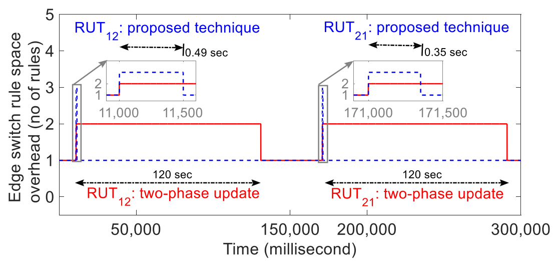
Rule update type	Switch	Proposed technique	Two-phase rule update technique
Update 1 → 2	Edge	99%	33%
	Core	99%	66%
Update 2 → 1	Edge	99%	33%
	Core	99%	66%

Update 1 → 2 : Rule update from route 1 to route 2
Update 2 → 1 : Rule update from route 2 to route 1

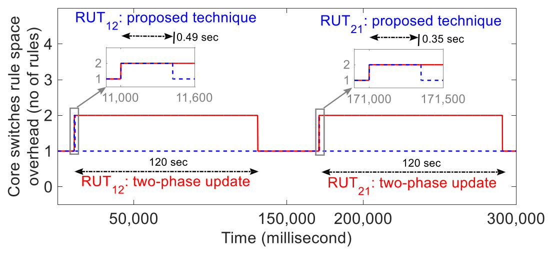
of the switches of the SDN testbed network. We got the rule update time of switches of the SDN testbed network, shown in Fig. 5(b), in the range of $0.2sec$ to $0.65sec$, which 75^{th} $percentile$ value is $0.45sec$. We also found that the rule update time of all switches, both the edge switches and the core switches, of a route of the SDN testbed network was same. Similar to the emulation-based results of Fig. 5(a), this experimental results of Fig. 5(b) lead us to the conclusion that because of having the efficient garbage collection technique, the rule update time of the switches is reduced to almost 99% compared to the *two phase rule update*[1] technique. Note that the rule update from route 1 to route 2 means the data packet flow was passing through route 1 and because of applying the rule update technique the data packet flow has started passing through route 2 instead of route 1, and vice versa.

Rule space overhead: In our experiment, we found that at the time of the rule update from route 1 to route 2 (or, from route 2 to route 1) of the SDN testbed network, the rule space overheads of the edge switches were 2, using the *two phase rule update* technique, and 3, using the proposed technique. Whereas, the rule space overheads of the core switches of route 1 (or, route 2), during the rule update from route 1 to route 2 (or, from route 2 to route 1) of the SDN testbed network, were 2, using the proposed technique, and 1, using the *two phase rule update* technique. Fig. 7(a) and Fig. 7(b) present the rule space overhead variation, using both the proposed technique and the two-phase rule update technique, of the edge switches and the core switches respectively. Both of these figures depict that the rule update time of both core and edge switches were $0.49sec$ in case the packet forwarding path of a data packet flow was changed from route 1 to route 2, and $0.35sec$ in case the packet forwarding path of a data packet flow was changed from route 2 to route 1. We know that, by using the proposed technique, for updating the rule from the route 1 to route 2 (or, from route 2 to route 1), the switches of route 1 (or, route 2) of the SDN testbed network required to hold one additional rule during the rule update time due to conducting the garbage collection technique. However, compared to the *two phase rule update* technique, the proposed technique significantly minimized the overall rule space overhead and the rule update time of the switches. In that context, the following rule time-overhead efficiency (E_R) was used to present an easier understanding.

Rule time-overhead efficiency: For both the proposed technique and the two-phase update technique, the rule time overhead efficiencies of the switches of the SDN testbed network are listed, using (5), in Table III, where $RUT_{max} =$



(a) Rule update time of the edge switches of route 1 and route 2



(b) Rule update time of the core switches of route 1 and route 2

Fig. 7: Rule update time of switches of the SDN testbed network

$120sec$ and $RSO_{max} = 3$. Table III presents that the proposed technique offers $1.5\times$ better the rule time-overhead efficiency to the core switch, and $3\times$ better the rule time-overhead efficiency to the edge switch compared to the two-phase rule update technique.

C. Discussion

In our experiments we got that the insertion and the start utilization of the new version of a rule were required around $0.5msec$ for the SDN testbed network. This time period is almost negligible compared to whole rule update time, i.e., $\sim 0.45sec$ for the SDN testbed network. Thus the rule update time of the network using the proposed technique mostly depends on the successful rule deletion time, i.e., reception of the "clean-up" data packet flow by the controller.

We observed that 1 clean-up data packet (each data packet was 120 bytes) was sufficient for both the *AGIS* network and the SDN testbed network to trigger the rule deletion process.

We run the Mininet-based system model, presented in Section VI-A1, to observe the behavior of TCP (Transmission Control Protocol) based data packet flow using the proposed technique. We found that there is no data packet re-transmission during the rule update for the present link capacity (45 $Mbps$ and 155 $Mbps$) of *AGIS* network topology. However, we noticed the data packet re-transmission occasionally during the rule update, when we increase the link capacity of *AGIS* network topology to 1 $Gbps$. Because of the increasing data rate and the limited buffer size of the switches, the data packet re-transmission rate went to 8 data packets, in the worst case scenario, during the rule update. But, in all cases, we did not find any data packet loss using the proposed technique due to the rule alteration.

Considering the rule deletion timeout of 2 minutes, we found that, using the proposed garbage collection solution, the rule update time of the two-phase technique can be reduced by up to 99%. We also observed that, in our experimental analysis, if we further reduce the rule deletion timeout to 10 seconds (or, even 1 second), the rule update times can be reduced by up to 95% (or, 55%). Therefore, reducing the benchmark of the rule deletion time, the rule update time can be reduced, however, the rule deletion time using the proposed garbage collection solution can be considered as the minimum one ensuring the *per-packet consistency* property. Note that, the reduction of the benchmark of the rule deletion time also reduces the *rule time-overhead efficiency* of the switches proportionally.

We already know that less efficient controllers slow down network performance [30]. Therefore, the computational resources allocated for the controller play a vital role to result the rule update time of the network. In our experiments, we have found that the limited computational efficiency of the controller leads to a relatively longer rule update time of the network.

VII. Conclusion

In this paper we have explored the garbage collection problem of old rules of the data plane of the asynchronous networks, and, to tackle this problem, we have proposed a novel clean-up mechanism for garbage collecting old rules. We have formally proven that our proposed technique guarantees *blackhole-freedom*, *loop-freedom*, and offers no data packet loss due to the path alteration of a flow. To demonstrate the real world feasibility, we deployed our solution on commercially available OpenFlow switch based SDN testbed and achieved an improvement in rule update time by up to 99% compared to *two phase rule update* that in turn minimized the rule space overhead significantly. Our solution can be used as a standalone technique or as a subsidiary technique with any existing rule update technique for any SDN network topology with a guarantee that no data packet is lost due to the rule alteration.

As a future work, the proposed solution can further be extended in the context of garbage collection of forwarding rules of the multiple packet forwarding paths, in which a data packet flow has to be migrated from a set of packet forwarding paths to a new set of packet forwarding paths in the network. Besides, the multiple controller scenario can also be an interesting aspect to investigate considering the real world scenario in which multiple operators are involved to manage a single flow in a large scale network. Finally, we plan to investigate the application of our technique to flow splitting scenarios where the same flow can use multiple routes at the same time.

References

[1] M. T. I. ul Huque, G. Jourjon, and V. Gramoli, "Garbage collection of forwarding rules in software defined networks," *IEEE Communications Magazine*, vol. 55, no. 6, pp. 39–45, 2017.

[2] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent updates for software-defined networks: change you can believe in!" in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks (HotNets)*, ACM, Ed., 2011, pp. 1–6.

[3] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zupdate: updating data center networks with zero loss," *SIGCOMM Comput. Commun. Rev.*, vol. 43, 2013.

[4] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 539–550.

[5] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN)*, ACM, Ed., 2013, pp. 49–54.

[6] T. Mizrahi and Y. Moses, "Time-based updates in software defined networks," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN '13*, ACM, Ed., 2013, pp. 163–164.

[7] T. Mizrahi, E. Saat, and Y. Moses, "Timed consistent network updates," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*, 2015.

[8] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "Oflops: An open framework for openflow switch evaluation," in *Proceedings of the 13th International Conference on Passive and Active Measurement (PAM'12)*, 2012, pp. 85–95.

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

[10] D. M. F. Mattos, O. C. M. B. Duarte, and G. Pujolle, "Reverse update: A consistent policy update scheme for software-defined networking," *IEEE Communications Letters*, vol. 20, no. 5, pp. 886–889, May 2016.

[11] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey, "Enforcing customizable consistency properties in software-defined networks," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 73–85.

[12] H. Xu, Z. Yu, X.-Y. Li, L. Huang, C. Qian, T. Jung, H. Xu, Z. Yu, X.-Y. Li, L. Huang, C. Qian, and T. Jung, "Joint route selection and update scheduling for low-latency update in sdns," *IEEE/ACM Trans. Netw.*, vol. 25, no. 5, pp. 3073–3087, Oct. 2017.

[13] R. Gandhi, O. Rottenstreich, and X. Jin, "Catalyst: Unlocking the power of choice to speed up network updates," in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '17. New York, NY, USA: ACM, 2017, pp. 276–282.

[14] A. Basta, A. Blenk, S. Dudycz, A. Ludwig, and S. Schmid, "Efficient loop-free rerouting of multiple sdn flows," *IEEE/ACM Transactions on Networking*, vol. PP, no. 99, pp. 1–14, 2018.

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

[16] S. Vissicchio and L. Cittadini, "Flip the (flow) table: Fast lightweight policy-preserving sdn updates," in *Proceedings of the 35th Annual IEEE INFOCOM*, IEEE, Ed., 2016.

[17] S. Brandt, K.-T. Förster, and R. Wattenhofer, "On consistent migration of flows in sdns," in *Proceedings of the 35th Annual IEEE INFOCOM*, IEEE, Ed., 2016.

[18] 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: ACM, 2017, pp. 21–33.

[19] 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. PP, no. 99, pp. 1–1, 2018.

[20] T. Mizrahi and Y. Moses, "ReversePTP: a software defined networking approach to clock synchronization," in *Proceedings of the third workshop on Hot topics in software defined networking (HotSDN '14)*, ACM, Ed., 2014, pp. 203–204.

[21] T. Mizrahi, E. Saat, and Y. Moses, "Timed Consistent Network Updates in Software-Defined Networks," *IEEE/ACM Transactions on Networking*, vol. 99, pp. 1–14, 2016.

[22] J. Zheng, G. Chen, S. Schmid, H. Dai, J. Wu, and Q. Ni, "Scheduling congestion- and loop-free network update in timed sdns," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 11, pp. 2542–2552, Nov 2017.

[23] L. Schiff, S. Schmid, and P. Kuznetsov, "In-band synchronization for distributed SDN control planes," *SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 1, pp. 37–43, Jan. 2016.

[24] H. Zhou, C. Wu, Q. Cheng, and Q. Liu, "SDN-LIRU: A lossless and seamless method for sdn inter-domain route updates," *IEEE/ACM Trans. Netw.*, vol. 25, no. 4, pp. 2473–2483, Aug. 2017.

[25] B. Finkbeiner and A. Solar-Lezama, Eds., *Proceedings Second Workshop on Synthesis, SYNT 2013, Saint Petersburg, Russia, July 13th and July 14th, 2013*, ser. EPTCS, vol. 142, 2014.

[26] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "A distributed and robust sdn control plane for transactional network updates," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, April 2015, pp. 190–198.

[27] A. Telesis, "Allied Telesis x930 series (28gTX)," 2017. [Online]. Available: https://www.alliedtelesis.com/products/x930-series

[28] "Ryu SDN Framework," 2016, accessed 2013. [Online]. Available: https://osrg.github.io/ryu/

[29] AGIS, "The Internet Topology Zoo," 2013, accessed 2011-01. [Online]. Available: http://topology-zoo.org/

[30] S. Mallon, V. Gramoli, and G. Jourjon, "Are today's sdn controllers ready for primetime?" in *2016 IEEE 41st Conference on Local Computer Networks (LCN)*, Nov 2016, pp. 325–332.

Md Tanvir Ishtaique ul Huque is a research associate at UNSW Canberra Cyber Center, UNSW, Canberra and has been with Data61-CSIRO, Australia, and the University of Helsinki, Finland in the past. He has received his Master degree and PhD degree both in electrical engineering from the University of Sydney in 2014 and the University of New South Wales (UNSW) in 2019, respectively. He is actively working on projects of time-sensitive networks and software-defined networks.

Guillaume Jourjon is senior researcher at Data61-CSIRO. He received his PhD from the University of New South Wales and the Toulouse University of Science in 2008. Prior to his PhD, he received a Engineer Degree from the ENSICA. He also received a DEUG in Physics and Chemistry (Major) and Mathematic (Minor) from the University of Toulouse III. His research areas of interest are related to the development of new teaching and learning facilities, measurement architecture and testbeds as well as Software Defined Network and Network economics.

Craig Russell received the Ph.D. degree in applied mathematics from Macquarie University, Sydney, Australia, in 1997. He is currently a Principal Research Engineer with the Cyber-Physical Systems Program of CSIRO Data61. He has design, implementation, and operational experience in a wide range of advanced telecommunications equipment and protocols. His professional interest is the application of advanced Ethernet and IP-based technologies to Australian industries.

Vincent Gramoli is Associate Professor in Computer Science at the University of Sydney and a senior researcher at Data61-CSIRO. Prior to this, he was affiliated with INRIA, University of Connecticut, Cornell University, University of Neuchâtel and EPFL. Vincent received his PhD from Université de Rennes and his Habilitation from Sorbonne University. He is the Chair of the Blockchain Technical Committee for the Australian Computer Society and a Future Fellow of the Australian Research Council.