Consistency is Not Easy: How to Use Two-Phase Update for Wildcard Rules?

Shouxi Luo, Hongfang Yu, and Lemin Li

***Abstract*—The recent proposed *two-phase* mechanism is a provable theory to achieve consistent updates for SDN. However, how to make it work for practical rules is important yet unsolved—(1) two-phase mechanism requires that rules in the new configuration after an update are assigned with a distinct version number from rules in the old configuration before an update; but (2) setting rules in each configuration with a distinct version number causes serious rule-space overheads in practice due to the sophisticated "covered" relationships between practical wildcard rules. In this letter, we design a simple yet generic solution for the problem. By using well-designed wildcard-based version number matchings, we simplify the update procedure, make a stream of updates easy to be processed in parallel, and avoid all unwanted rule-space overheads. We think that our mechanism bridges the gap between the theory of two-phase consistent update and the practical issue of how to use it for today's networks.**

***Index Terms*—Consistency, planned change, software-defined networking, wildcard rule.**

I. Introduction

RECENT trends toward Software-Defined Networking (SDN) suffers from undesired transient behaviors during planned changes since the updates on switches take effect disorderly [1]–[3]. More specifically, when a network configuration changes from one to another, in-flight packets will encounter a mix of both the old configuration and the new configuration along their paths. The network configuration is inconsistent during the update; interim forwarding errors like *Reachability Failures*, *Forwarding Loops*, *Traffic Isolation* and *Leakage* are prone to occur.

The seminal work by Reitblatt *et al.* [2] has introduced the theory of *two-phase mechanism* to deal with such inconsistent problems (see Section II). By adopting version-based rule matchings, two-phase mechanism guarantees that a packet or a flow is handled either by the old configuration before an update or by the new configuration after an update, but never by some combination of the two. Nonetheless, there is not any literature that answers the non-trivial practical question of how to employ two-phase mechanism for current networks. As we will see, the straightforward implementation of two-phase mechanism presented in the literature [2] is impractical for today's networks that use wildcard rules—not only because it requires a complicated version number management but also because it causes serious rule-space overheads in practice.

In the original two-phase mechanism, each rule is assigned with the configuration's version number. Correspondingly, ingress switches tag each incoming packet with the current version number, telling which version of rules that packet should follow when traveling the network (The version numbers can be stored in OpenFlow header's unused fields in implementation, e.g., the VLAN tag [1], [2]). When an update occurs, two-phase mechanism first installs the new configuration's rules to the middle of the network and then flips packet version numbers at ingresses. Since only the "right" version of configuration takes effect along the path for each packet (or flow), the mechanism avoids the mix of configurations and preserves consistencies.

It is easy and cheap to employ the version-based method to update network configurations that consist of overlap-free rules (e.g., rules for per-flow/microflow based routing). In these networks, a switch's multiple rules have disjointed header spaces. The to-be-updated/updated rules and unmodified rules match with two disjointed sets of packets. Hence, the controller can easily carry out an update by only (1) setting a new version for the updated rules and (2) tagging involved packets with that new version number. However, this is not the case in most of current networks, in which rules contain wildcards and overlap [4], [5] (e.g. the prefix routing table or non-prefix firewall table). Because of the sophisticated dependency relationship between rules, a packet matching with an updated rule on a switch, may match with an unmodified rule in the next-hop switch; and *vice versa*. To carry out an update without omissions, all ingresses need to tag all incoming packets with the same new version number. This causes a complicated version number management and requires all rules to be duplicated for the new version number (i.e., causing *rule duplications*). As switches have to reserve the old version of rules until packets tagged with that version drain, the rule duplications cause serious rule-space overheads in practice.

In this letter, we propose a simple yet generic mechanism to simplify the process of two-phase update and to reduce rule-space overheads. Our mechanism is inspired by a critical observation—as a rule's duplications only differ in the field of version number, implementing the version number matchings using wildcards will avoid rule duplications. By using a well-designed version number setting policy and implementing the version number matchings with the help of wildcards, we improve the two-phase mechanism to get two crucial benefits:

- Each update is carried out by only operating its own to-be-updated rules and own involved ingresses—this greatly simplifies the version number management of a steam of updates and makes their processes natural to be handled in parallel.
- No rule duplication is needed—this avoids the serious rule-space overheads.

Manuscript received September 4, 2014; accepted December 30, 2014. Date of publication January 7, 2015; date of current version March 6, 2015. This work is supported in part by the 973 Program under Grant No. 2013CB329103, and the National Natural Science Foundation of China under Grant No. 61271171. The associate editor coordinating the review of this paper and approving it for publication was P. Serrano.

The authors are with the Key Laboratory of Optical Fiber Sensing and Communications, Ministry of Education, University of Electronic Science and Technology of China, Chengdu 611731, China (e-mail: rithmns@gmail.com; yuhf@uestc.edu.cn; lml@uestc.edu.cn).

Digital Object Identifier 10.1109/LCOMM.2015.2388754

We think that our mechanism bridges the gap between the theory of two-phase consistent update method and the practical issue of how to use it for practical wildcard-based networks.

II. Background

In this section, we give a brief review of how the two-phase mechanism [2] processes and why it works.

How a Two-Phase Update Works: Generally, when using two-phase mechanism, each packet is tagged with the configuration's version number when it enters the network (done at each ingress switch); then the subsequent switches it encountered use this version number to find the "right" version of configuration to apply; finally, egress switches strip the version number when the packet leaves. Suppose the controller is to update the network configuration from current C_{old} to another C_{new}, where C_{old} tags packets with version 1 and C_{new} tags packets with version 2, the two-phase update mechanism involves three passes of operations as follows:

1) **Pass-1**: Install C_{new}'s rules in the middle of the network;
2) **Pass-2**: Install C_{new}'s rules at the ingress switches after all the operations in Pass-1 complete; these ingress rules tag all the matched packets with the new version number 2;
3) **Pass-3**: Remove all C_{old}'s rules after all packets tagged with version number 1 drain from the network.

Why Two-Phase Mechanism Guarantees Consistency: For update techniques, the authors [2] summarize the fundamental building blocks into two types—the *one-touch update* and the *unobservable update*; and prove two theorems—(1) If an update is a one-touch update then it is a per-packet consistent update; (2) If an update is a series of an unobservable update and a per-packet consistent update then it is also a per-packet update.

Roughly, a one-touch update is an update with the property that no packet will meet/match with the updated rules more than once, e.g., the sub-update of Pass-2, where involved packets only meet with the updated rules at ingresses; and an unobservable update is an update that does not change the network's forwarding behaviors, e.g., the sub-updates of Pass-1 and Pass-3, where no packet traces have been changed after the updates. The two theorems give us a simple way to prove and design consistent update techniques. For instance, as the original two-phase mechanism's 3 sub-passes are *unobservable*, *one-touch* and *unobservable*, respectively, it is proved to guarantee the per-packet consistency.

III. Problem Statement

As the procedure of two-phase mechanism in Section II shows, the key to make consistent updates is employing distinct version numbers to distinguish between old/current rules and new rules. It is easy to determine what an update's "old" rules and "new" rules are, if the match fields of rules on a switch are overlap-free. In such cases, each class[1] of packets has its explicit path(s), and rules belonging to its path(s) would not match with other class of packets. Then, the update's "old" rules and "new" rules are just defined by the to-be-updated/modified rules. The controller can simply assign a new version number to the new rules to make consistent updates. For example, the networks that use per-flow/microflow or per-tunnel routing fall into this category.

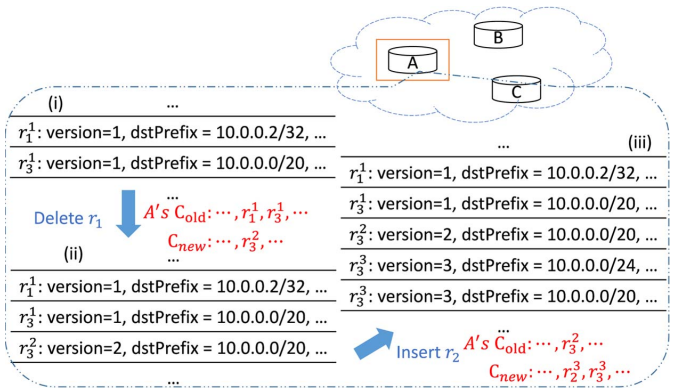

Fig. 1. The example of that the controller installs duplicated rules to switch A for two successive updates: the prior update (i.e., i $\rightarrow$ ii) is to delete r_1 and the later (i.e., ii $\rightarrow$ iii) is to insert r_2. As table-(ii) and table-(iii) show, A has to hold the old versions of rules (i.e., version 1 and 2) until packets tagged with that versions drain. Note that, we omit both the complete workflow of two-phase updates and the operations on other switches.

However, the per-flow or per-tunnel routing is only a special case in current practical networks. In today's networks, the match field of a rule is a ternary string consisting of 0, 1 and wildcard $*$. The ternary string denotes an original network address (e.g. a prefix destination address, a source-destination pair, or a 5-tuple etc.) or an aggregated address [5]. In each switch, rules commonly overlap and are ordered according to their priorities [4], [5]. Entered packets are processed as the first matched rule specifies.

When the controller is to insert, delete or modify a set of rules, the new rules for the update are not only defined by the to-be-update rules, but also by the "covered" rules. Take the toy network shown in Fig. 1 as an example, where rules $\{\cdots, r_i^k, r_i^l, \cdots\}$ denote r_i's duplications for different version numbers; and the update's operation(s) on switch A is to delete rule r_1 from $Table$: $[\cdots, r_1, r_3, \cdots]$. As the match field of $r_1 - 10.0.0.2/32$ covers that of $r_3 - 10.0.0.0/20$, A's new configuration after r_1 being deleted is C_{new} : $[\cdots, r_3^2, \cdots]$, while A's old configuration is C_{old}: $[\cdots, r_1^1, r_3^1, \cdots]$.That is to say, to make two-phase consistent updates, the controller needs to install a duplication of the "covered" r_3 for the new version number even though r_3 is not an updated rule (see table-ii in Fig. 1). Moreover, an update generally involves a suit of rule operations on several switches. The involved rules on different switches usually have distinct match fields.[2] To avoid omissions when upgrading to the new configuration, all ingress switches need to tag all incoming packets with the new version number. Accordingly, all the unmodified rules also need to be duplicated for the new version number even if they are not "covered" by any updated rules. Since switches must hold both versions of rules during the update procedure, the rule duplications of two-phase mechanism causes serious rule-space overheads.

[1]The item *class* here is similar to the concept of Forwarding Equivalence Class (FEC) in MPLS. It may describe a set of packets with a same destination IP address, a same 5-tuples, or a same MPLS label, etc.

[2]This may be defined by the update's demands or caused by flow table aggregation techniques. In an aggregation-enabled network, the controller usually needs to insert and delete several aggregated rules to merge an (original) update, and the rule operations in switches differ from one another [5].

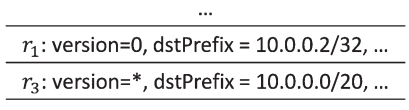

Fig. 2. Employing 0 and 1 as the two version numbers and implementing r_3's version-based matching using $*$ avoid r_3's rule duplications.

In addition, a dynamic network's configuration is volatile. Once a new update occurs before the current two-phase procedure completes, switches have to hold all old duplicated rules if packets tagged with that version(s) have not drained yet. As the example in Fig. 1 shows, switch *A* needs to hold two old versions of rules temporarily (version 1 and 2), which both contain r_3's duplications. Besides, the controller must guarantee that the new version number is not in use as well.

In short, to achieve consistent updates, the original two-phase mechanism needs a complicated version number management and causes serious rule-space overheads.

IV. Handling One Update

Key Ideas: As discussed in Section III, setting wildcard rules with explicit version numbers complicates the version number management and causes serious rule duplications. Motivated by the observation that a rule's multiple duplications only differ in the fields of version number, we plan to use *wildcard*s for the match of version numbers to avoid rule duplications. Then, the update procedure needs to be redesigned carefully. We first investigate how to handle a single update in this section and study the handling of a stream of updates in Section V.

Design Details: When processing an update, we use {0,1} as the two version numbers which only cost one bit in the packet header. In implementation, the version number is stored in unused header fields like VLAN tags or MPLS labels [2]. In consideration of that the value of unused fields in a packet header is set to 0 and the value of unused fields in a rule's match fields is set to $*$ by default, we assign 0 to the old/current configuration and 1 to the new configuration. Accordingly, we use $*$ as the match fields of the unmodified rules; then they can match with both versions without duplications.

For example, for the case of deleting r_1 from *A*'s table shown in Fig. 1, as Fig. 2 shows, we set r_1 to version 0 since it is an old rule to be deleted, and set r_3 to $*$ since it is an unmodified rule existing in both configurations. Following this principle, we further design the complete procedure of deleting rules from a network as follows:

1) For the rules to be deleted, modify their match fields of the version number from $*$ to 0;
2) Install action "tagging version-1" to the ingress switches to tag the incoming packets with version number 1;
3) (After all the version-0 packets drain from the network) Remove all the to-be-deleted rules;
4) Remove the version-tagging actions at ingresses.

Note that, before the update begins, all packets are tagged with version-0 and all rules' version fields are set with $*$ by default. Thus, at the first step-Step (1), we modify the match fields of these to-be-deleted rules from $*$ to 0 to start the update procedure. And at the end of the procedure-Step (4), we annul the tagging of version-1 to make all packets to use the default version number 0 again.

Similarly, for an update consisting of only insertions, we can schedule its procedure as follows:

1) For the rules to be inserted, set their match fields of the version number to 1 and install them to the switch;
2) Install action "tagging version-1" to the ingress switches to tag all incoming packets with version number 1;
3) (After all the version-0 packets drain from the network) Modify the match fields of the version number in inserted rules from 1 to $*$;
4) Remove the version-tagging actions at ingresses.

Like the case of deletion, at the end of insertion's procedure, we first reset the version number fields of each inserted rules back to default $*$ in Step-(3), and then annul the version tagging actions in Step-(4) to make packets reuse the default version number 0.

The Generic Update Mechanism: Generally, an update consists of insertions, deletions and modifications. The modification of a rule is equivalent to inserting the new rule and deleting the old rule. Accordingly, we can design a generic update scheduling for two-phase updates by making a synthesis of both the deletion operations and insertion operations:

Step 1) For rules to be inserted, set their match fields of the version number to 1 and install them to switches;
for rules to be deleted, modify their match fields of the version number from $*$ to 0;
for rules to be modified, treat each modification as an insertion plus a deletion.
Step 2) Install action "tagging version-1" to the ingress switches to tag all incoming packets with version-1;
Step 3) (After all version-0 packets drain from the network) Modify the match fields of the version number in installed rules to $*$; remove all the to-be-deleted rules.
Step 4) Remove the version-tagging actions at ingresses (And wait until all version-1 packets drain).

As the procedure shows, besides the change of version-tagging actions at ingress switches, our method only operates the to-be-updated rules to process an update. The procedure is quite simple and no rule duplications are needed.

On the Correctness of the Mechanism: For each unmodified rule whose match field of the version number is $*$, we can expand $*$ into 0 and 1 to split the rule into two disjointed rules without changing the forwarding semantics. The two sub-rules belong to the old configuration and the new configuration respectively. In such an imaginary table, each tagged packet only matches with rules that have the "right" version number when traveling through the network; thus per-packet consistency is guaranteed. Actually, since the Step-1, Step-3, and Step-4 are unobservable updates and Step-2 is a one-touch update, refer to the two theorems mentioned in Section III, the whole update is a per-packet consistent update.

However, similarly to [2], to achieve per-flow consistencies, our mechanism also needs to install a single rule for each flow at ingress switches and set a timeout to that rule, or employ other techniques like wildcard cloning.

V. Handling a Stream of Updates

In a dynamic network, the forwarding configuration is volatile and updates occurs one after another. As a two-phase

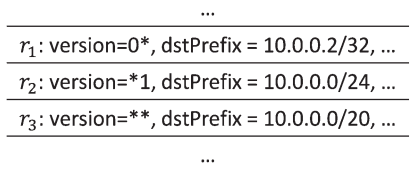

Fig. 3. The example of using separate bits for different updates' version numbers to perform multiple updates in parallel.

update takes a significant duration, controllers need to deal with a new coming update before the current update completes.

Fortunately, for each update, our method only needs one bit to store version numbers and the update procedure only operates its own to-to-updated rules. Therefore, by simply using disparate fields for different updates, the controller can process their 4-step update procedures independently and simultaneously. Take the two coexisting updates discussed in Section III and Fig. 1 as an example, where one update is to delete r_1 from A's table and the other is to insert r_2 to the same table (operations on other switches' tables are not shown); it is easy to make parallel updates by using two bits for their version numbers as Fig. 3 shows.

Suppose that the version numbers are stored in VLAN tags (32 bit) in implementation, then the controller can recycle these fields to support 32 ongoing updates simultaneously. Moreover, for a group of updates that occur in a short time, they can be treated as a hybrid update to reduce the demands of fields.

VI. Discussion

About Ingress Switches: To make consistent updates, both the original two-phase mechanism and our mechanism tag packets with version numbers at ingress switches. In our mechanism, the ingress switches are not limited to the switches at the border of the network; they are a group of switches that can tag all the packets involved in the update before the packets encounter the to-be-updated rules/switches. So, there are multiple chooses of ingress switches for an update (but the "simplest" choice is to use border switches). The selection of ingress switches defines how many switches need to install version-tagging actions in Step-2, and affects how long the process should wait for packet draining in Step-3 and Step-4. It is an open problem.

Update Durations: The procedure of our mechanism consists of 4 steps and the process would not go to the next step until all operations in the current step complete (the operations in each step can proceed in parallel). Suppose that the operation set for Step-i is O_i ($i = 1, 2, 3, 4$), the time cost of operation e is $c(e)$, and the (upper bound) time for packet draining is t_o. Then the update duration can be estimated as $\sum_{i=1}^{4} \max\{c(e)|e \in O_i\} + 2t_o$, where one t_o is used to ensure that all version-0 packets have drained in Step-3, and the other is used to ensure that all version-1 packets have drained from the network in Step-4. Further, let $t_\lambda = \max\{c(e)|e \in \cup_{i=1}^{4} O_i\}$; accordingly, the update duration can be roughly estimation as $4t_\lambda + 2t_o$, which has a weak correlation to the scale of update.

VII. Related Work

This work builds on the seminal work by Reitblatt *et al.* [2], which introduces the inconsistent problem during planned updates and presents the two-phase theory to guarantee per-packet and per-flow consistencies for updates. However, the raw two-phase update mechanism they proposed is not friendly to current network that use wildcard rules—because their method assigns distinct version numbers to rules, which causes complicated version number managements and serious rule-space overheads. Based on their theory, we propose a simple yet generic method to achieve the version-based matching using wildcards, which makes the update procedure much easier and avoids all rule duplications.

When modifying pre-exiting rules, the two-phase mechanism needs $2\times$ rule-space because it leaves the old rule on the switch as it installs the new one. To this problem, Katta *et al.* [6] break a bulky update into k rounds and introduce algorithms to trade the time required to perform a consistent update against the rule-space overheads required to implement it. Our mechanism can improve their update of each round. Different from Katta *et al.*, the work by Ratul Mahajan and Roger Wattenhofer [3] gives up the strong consistency to avoid the rule-space overheads. However their algorithms only guarantee loop-free and are hard to work for non-prefix rules. Moreover, for updates with large scales, their methods have huge updating durations since the rule operations have dependencies and need to be enforced in the planned sequence. Suppose O_L is the set of operations on the longest path in the *dependency tree* [3], $t(e)$ is the time cost of operation e, their methods will cost more than $\sum_{e \in O_L} t(e)$ to complete the update.

VIII. Summary

In this letter, we answered the practical question of *how to employ two-phase update theory for networks that use wildcard rules*. Our simple yet generic solution simplifies the update procedure and makes consistent updates easy to achieve.

Acknowledgment

We thank the anonymous reviewers and editors for useful feedback.

References

[1] N. McKeown *et al.*, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.
[2] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. ACM SIGCOMM*, 2012, pp. 323–334.
[3] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *Proc. ACM HotNets*, 2013, pp. 20:1–20:7.
[4] X. Meng *et al.*, "Ipv4 address allocation and the bgp routing table evolution," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 1, pp. 71–80, Jan. 2005.
[5] S. Luo, H. Yu, and L. M. Li, "Fast incremental flow table aggregation in SDN," in *Proc. 23rd ICCCN*, Aug. 2014, pp. 1–8.
[6] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *Proc. ACM HotSDN*, 2013, pp. 49–54.