Reverse Update: A Consistent Policy Update Scheme for Software Defined Networking

Diogo Menezes Ferrazani Mattos, Otto Carlos Muniz Bandeira Duarte, Guy Pujolle

***Abstract*—Policy and path updates are common causes of network instability, leading to service disruptions or vulnerable intermediate states. In this letter, we propose the Reverse Update, an update scheme for Software Defined Networking that guarantees to preserve properties of flows during the transition time. We prove through a formal model that the proposal achieves consistent policy updates, in which in-transit packets are always handled in the next forwarding hops by the same or a more recent policy. The main contributions are: (i) a relaxation of the concept of per-packet-consistency in the data plane of Software Defined Networking; and (ii) a policy update scheme, proved to be consistent and efficient. A Software Defined Networking simulator was developed and validated. The results of our simulations show that the proposed Reverse Update scheme is faster and has lower overhead than the current Two-Phase Update proposed in the literature.**

***Index Terms*—Software Defined Networking, Consistency, Policy Update, Network Security**

I. Introduction

POLICY updates[1] on Software Defined Networking (SDN) can lead to network instabilities, such as, outage, performance degradation, and inconsistent intermediate states [1], [2]. Policy update consistency is challenging, because there is no guarantee that the intermediate states are consistent, even when the initial and the final network states are correct and consistent. The transition from one network configuration to another should occur in an install and uninstall sequence of rules, switch after switch, to ensure that the network behaves properly during the procedure. The assumption that each SDN application should be responsible for its own policy update procedure is not feasible. The network control applications are usually error-prone when dealing with policy updates [3]. Moreover, the concept of per-packet consistency defines that packets traversing the network are handled by just one, single and consistent, global network configuration. Consequently, no network packet is ever processed by a mixture of network configurations [1].

The main proposals for updating policies in SDN are based on two ideas, the atomic update [4] or the Two-Phase Update [1], [5]. The atomic update considers that all network nodes accomplish an atomic update operation. The switch update process, however, cannot be performed atomically. As a consequence, in-transit packets, which are traversing the network while the update is being deployed, can be handled by inconsistent intermediate configurations of the network, forcing the update to roll back. The Two-Phase Update scheme ensures a per-packet consistency on the network by labeling each packet with a network configuration version tag. Hence, when the packets arrive on the network, the ingress node tags the packet with the current version and, then, all nodes process the tagged packets according to their version.

In this letter, we propose the Reverse Update scheme that guarantees a per-packet consistent policy update for Software Defined Networking. The proposed scheme is based on the relaxation of the per-packet consistency concept and on the installation of policy updates in a sequence that corresponds to the reverse path of the flows. The Reverse Update scheme neither introduces nor stores different versions of rules in the switches, as each switch atomically updates the previously installed rules instead. Our proposal, when compared to the Two-Phase update [1], also provides guarantees of consistency without introducing overhead on the flow table and complexity on rules composition by using wildcards. The proof of consistency of the Reverse Update scheme is performed using a formal model of Software Defined Networking. We simulate the Reverse Update scheme over a real network topology. Our results show that the installation overhead of the Reverse Update scheme is close to the ideal update scheme.

II. Related Works

The OF.CPP (Consistent Packet Processing for OpenFlow) proposal [4] argues that the SDN packet processing is error-prone because the decision making process and the implementation of forwarding rules are not atomic. Reiblatt *et al.* propose a universal per-packet consistent update scheme, Two-Phase Update, and defines an abstract formal model for Software Defined Networking [1]. The key idea of the Two-Phase Update is to associate each global network configuration with a version number and to tag all packets with a version number that determines the single global network view under which the packet should be handled. When a new version of the control plane is launched, the packets must be tagged with the new version of the configuration. The ingress edge switch tags the packets with the most current version tag and the egress edge switch removes the version tag when packets leave the network domain. The tagging procedure differentiates the entering packets, which will be processed by the new configuration, from the in-transit packets, which are

Manuscript received Freburary 06, 2016; revised March 15, 2016. This work was supported by CNPq, CAPES, and FAPERJ.

Mattos, D.M.F., and Duarte, O.C.M.B are with Grupo de Teleinformática e Automação, Universidade Federal do Rio de Janeiro (COPPE/UFRJ), Rio de Janeiro, Brazil. Mattos, D.M.F., and Pujolle, G. are with Laboratoire d'Informatique de Paris 6, Sorbonne Universities, UPMC Univ Paris 06, Paris, France. (e-mail: menezes@gta.ufrj.br; otto@gta.ufrj.br; Guy.Pujolle@lip6.fr).

[1]In this letter, we the terms configuration update and policy update interchangeably.

processed by the preceding configuration. Luo *et al.* argue that the Two-Phase Update can lead to the definition of subsets of rules, when new and current rules are defined with overlapping sets of wildcard fields [6].

To avoid the Two-Phase Update overhead on switch memory, McGeer proposes to buffer in-transit packets on the controller, when deploying an update, and then reinject these packets on other network node after concluding the update [7]. By their turn, Katta *et al.* propose to split the policy update into rounds, and each round acts on a set of flows on the network [8]. Indeed, each round performs a Two-Phase update to a partial subset of flows. After a round, the old installed policies are removed of the network. Both proposals reduces the overhead on the network switches at the cost of a longer update process, as well as a more complex update, because both activities depend on analyzing the new policies before performing them. McClurg *et al.* propose an algorithm for searching for an order to deploy the policy update on the network [9]. The main idea is to automatically seek for an update order that keeps invariant specific properties during the update procedure. This proposal focuses on path updates, although it imposes a higher complexity for the update procedure as it adds the algorithm complexity of searching an update deployment order. Canini *et al.* argue that the simultaneous updates can lead the network to inconsistent states, and propose a transactional interface for policy updates, which applies the update on the network if there is no conflict with other policies [5].

III. SDN Model Background

The notation used in this letter is based on the one applied by Reitblatt *et al.* [1]. We consider us as an update sequence of observable events that modifies the initial state N of the network to the final state N', after an execution. The update observable events, u in us, are messages exchanged between controller and switches that change the network state. We consider as network state the set of the global network view and all flow forwarding rules. The basic structures of the model are packets and ports. The packet pk is the network data transmission unit. The port p represents the location on the network where a packet waits to be processed.

The network is modeled as a packet processor, which is able to forward packets and, in some cases, to modify the content of the packet at each hop [1]. Therefore, packet processing is modeled as a composition of two simple functions, to process a packet on the switch and to move a packet on a link. The execution of the network is represented by a switch function S that inputs lp, representing a located packet, i.e., the tuple containing the packet and the switch port where the packet is, and returns a list of located packets. The return indicates the future location of the packets on the network. Other important definitions are the trace, t, and port queue, Q. A trace is a list of located packets that keeps track of the sequence of switch hops that a packet passes through. The port queue stores packets waiting to be processed at each port on the network. The network state, N, is a pair (Q, C), containing the port queue, Q, and a network configuration, C. The configuration, C, comprises a switch, S, and a topology, T, functions.

Reitblatt *et al.* state that two traces are equivalent if the packets in the two traces have the same characteristics. Per-packet consistent updates occur when the traces generated by the network, at the time of an update, is equivalent to traces generated either by the initial configuration before the update, or by the final configuration after the update. In other words, there is no trace corresponding to a transition state, mixing two different configurations. Reitblatt *et al.* also consider two other settings: one-touch and unobservable updates.

Definition: One-Touch Update. The one-touch update is an update that does not generate traces with intermediate states, i.e., packets are handled either by the initial configuration (C_1) or by the final configuration (C_2).

Definition: Unobservable Update. The unobservable update does not affect the set of traces generated by the network.

Definition: Two-Phase Update. It introduces the network configuration version tag. Thus, the version becomes a property of the network trace. A trace may differ one from another just by the version tag. The configuration C is a n-version of the network configuration if $C = (S, T)$ and S modify the processed packets in any ingress port on the network changing the packets to include the version tag set to n. The version tag does not change in any point on the network.

As an one-touch or an unobservable update are per-packet consistent updates, Reitblatt *et al.* prove that if us_1 is an unobservable update from C_1 to C_2 and us_2 is a one-touch update from C_2 to C_3, then the concatenation of $us_1 + +us_2$ is a per-packet consistent update from C_1 to C_3. Therefore, Two-Phase Update is defined as a concatenation of updates.

IV. The Reverse Update Scheme

Reverse Update is a policy update scheme for Software Defined Networking that ensures consistency of policy commitment. The key advantage of our proposal when compared with the Two-Phase Update is the lower overhead for configuring flow tables, as it does not depend on packet tagging. The Reverse Update is based on updating flow processing and forwarding rules in the reverse path of the already installed flow, to assure that a flow always reaches the most current network configuration.

We relax the concept of per-packet consistency. We assume that a packet may be processed by more than one global network configuration if, and only if, it is always processed by the most recent network configuration. In other words, the relaxed concept of per-packet consistency avoids that a packet, which has already been processed by a recent configuration, be processed by a previous configuration in the next hops. The relaxed concept is important to assure that a packet is never forwarded by unexpected network states. We guarantee this property because every packet that reaches an already updated switch will always be handled by the most recent network configuration, in which the invariant properties are assured. Moreover, the relaxed concept enables the fast deployment of updates, as it updates even the in-transit packets.

Definition: Reverse Update. Let $dom(u_i)$ be the domain of a update u_i and $us = [u_1, \dots, u_k]$ be a sequence of switch updates, ordered to be committed in the reverse sense of the flow path on the network. The the update order is given by

$$\forall(p, pk) \in dom(u_i),\ \nexists(p, pk) \in dom(u_j), \text{ for } i < j \leq k.$$

A packet that travels from the source to the destination should never traverse a switch that still presents a previous configuration state. This occurs because the configuration updates are applied on the reverse sense, from the destination to the source. Thus, for all:

- initial states Q;
- executions $(Q, C_1) \overset{us}{\rightarrow} \star(Q', C_2)$;

packets that are processed by C_i will nevermore be processed by C_j, where $j < i$, according to the relaxation of the concept of per-packet consistency. When using the Reverse Update scheme, every packet will not be processed by a preceding configuration state and, thus, we have the following theorem.

Theorem: If an update sequence us is a Reverse Update, then us is a per-packet consistent update.
Proof: The proof follows the induction mechanism over us. We first consider an update sequence $us = [u_1, \dots, u_i, \dots, u_k]$. We induct over k.
Base Case (for $k = 1$): In the base case, $us = u_1$, there is only one update to be performed. In this case, the flow path length is one. Thus, there is just one hop on the network to apply the update. Considering this update by itself an one-touch update and, then, for every execution $(Q, C_1) \rightarrow \star(Q', C_2)$ that generates trace t, we have $(Q, C_1) \rightarrow \star(Q'', C_1)$ or $(Q, C_2) \rightarrow \star(Q'', C_2)$ where Q'' contains the trace t' that is equivalent to trace t. Therefore, according to the definition, the base case is per-packet consistent.
Induction Hypothesis (for $k = i$): We assume that the update sequence $us = [u_1, \dots, u_i]$ is a Reverse Update and it is per-packet consistent.
Induction Step (for $k = i + 1$): For $k = i + 1$, we assume that $us = [u_1, \dots, u_i, u_{i+1}]$. Therefore, we define $us_i = [u_1, \dots, u_i]$. We also consider:

$$Dom(us_i) = \cup_{j \leq i} dom(u_j), \text{ and } us = us_i + +u_{i+1}.$$

From Induction Hypothesis, us_i is a Reverse Update and it is per-packet consistent. Thus, we have just to prove that the concatenation of us_i with u_{i+1} is also a Reverse Update that maintains per-packet consistency property. Therefore, we check that:

- If $(p, pk) \in Dom(us_i)$, then $(p, pk) \notin dom(u_{i+1})$, by the definition of Reverse Update,
- t contains (p_1, pk_1) and (p_2, pk_2),
- There is no $(p, pk) \in Dom(us_i)$ and, at the same time, $(p, pk) \in dom(u_{i+1})$,

then, for all:

- Initial state Q_i,
- Executions $(Q_i, C_i) \overset{u_{i+1}}{\rightarrow} \star(Q_{i+1}, C_{i+1})$

the execution is an One-touch Update and, thus, it is per-packet consistent. Following the Concatenation Theorem [1], as us_i is consistent and the u_{i+1} update is also consistent, the update sequence us is a per-packet consistent update.

It is worth noting that the definition of a Reverse Update is restricted to policy updates that act on disjoint and loop-free paths on the network. Moreover, the composition of new and old paths should be loop-free. Taking these restrictions into consideration, Reverse Update fits well for path and action updates on the flow paths. Moreover, when considering switches that process packets through multiple tables, the Reverse Update remains the same, as the processing pipeline acts as a loop-free path inside each switch. The scheme acts in each switch on the reverse path, updating each table in the reverse order of packet-processing pipeline. Another important consideration is that, as wildcard flow entries have different granularity, policy updates can incur on the definition of overlapping policies. Updating overlapping-policies is a complex challenge [6]. In this paper we assume that updates do not overlap with already defined policies.

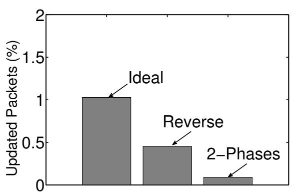
V. Simulation and Results

We evaluate the proposed Reverse Update scheme by simulating an SDN. We developed a discrete event simulator that considers the SDN model summarized in this letter and proposed by Reitblatt *et al.* [1]. The simulator was written in Python. The evaluation of the proposed scheme considers the real topology of the National Research and Educational Network (*Rede Nacional de Ensino e Pesquisa* - RNP), from Brazil, with 31 nodes. The graph topology was obtained from *The Internet Topology Zoo*[2]. The inter-arrival time between new flows follows *log-normal* distribution, with average equals to 7 ($\mu = 7$) and standard deviation equals to 2 ($\sigma = 2$) [10]. The arrival of new flows happens during 900 simulation steps and the simulation ends when all port queues of all switches are already empty. Each flow lasts for 50 simulation steps.

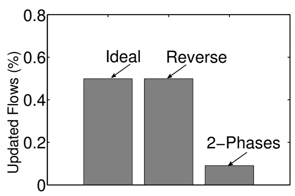
Simulation of the Reverse Update (`Reverse`) was compared to the Two-Phase Update (`2-Phases`) [1] and to an Ideal Update (`Ideal`). The Ideal Update is only feasible through simulation and it was introduced by comparison purpose. The Ideal Update stops packet forwarding on the network for accomplishing the update procedure. The Ideal Update is similar to an atomic update [4] that always achieves to commit updates. Update events change actions over the flows and were held after each 300 simulation steps. New flow arrivals were the same for all schemes.

The first experiment, shown in Figure 1(a), measures the percentage of packets that are processed by two different network configurations. For Ideal Update, 1% of packets is processed by two different configurations. This means that 1% of the packets were in-transit when the network accomplishes configuration updates. Even though an atomic update procedure does not introduce transitional nor inconsistent states, in-transit packets may be processed by two different configurations. Our Reverse Update proposal practically halves percentage of packet processed by two different configurations because the hop-by-hop update, from the destination to origin, halves the effect on in-transit packets. It is worth to note that

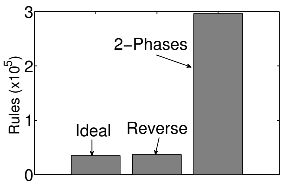
[2]http://www.topology-zoo.org/.



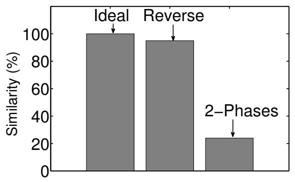
(a) Percentage of packets handled by different network configurations.



(b) Percentage flows that are affected by the policy updates.



(c) Total number of installed rules on the network during the simulation.



(d) Similarity index of the update schemes with the Ideal Update.

Figure 1. Effect of policy updates on packets and flows forwarded on the network. The simulation runs on a real topology of a Brazilian Research (RNP) network. The average flow arrival interval is 1 s and the flow duration is 50 simulation steps. a) Two-Phase Update has a few packets forwarded by two different configurations due to first-packet handling by OpenFlow controller during the update procedure; b) Ideal and Reverse Updates update the same number of flows; c) Two-Phase Update installs the greatest number of rules; and d) Reverse Update has a similarity of 94% with Ideal Update.

the Two-Phase Update still presents a number of packets processed by two different configurations, which is an unexpected behavior. When a packet is the first of a flow, it does not have a flow entry on all switches on the path and it is sent to the controller. As our simulation adopted a naive controller model[3], if a packet reaches an already updated controller, the new flow entry is calculated based on the updated network configuration. This behavior is reflected by the number of packets handled by two different configurations, even on the Two-Phase Update scheme [1]. According to the relaxation of the per-packet consistency concept, it is clear that the Reverse Update has a percentage of packets affected by updates that is very close to the Ideal Update, as shown in Figure 1(a). It also indicates that Reverse Update has a faster reaction time to the update when compared to the Two-Phase Update. We also evaluated the number of affected flows by the updates, i.e., flows that were in-transit while the updates take place on the network. This experiment measures how fast is the update completion. It is noticed that the Reverse Update schemes and Ideal Update have the same number of updated flows, Figure 1(b), while the delay for committing updates introduced by Two-Phase Update reduces the number of flows that are handled by the most current configuration. Another important feature to evaluate is the number of installed rules in the switches, as depicted in Figure 1(c). We verify that the number of rules installed by the Two-Phase Update is almost eight times higher than other update schemes, due to the addition of new rules on the network-core ports on each switch, for each flow on the network. The Reverse Update only updates the existing rules, as well as the Ideal Update.

Finally, we evaluated, for each update scheme, the percentage of forwarded packets that follows the same configuration when compared to the Ideal Update. This metric is important to measure the update proportionality. As stated by Reitblatt *et al*, a proportional update is the one in which the cost of installing a new policy is proportional to the applied change [1]. We consider the Ideal Update as the reference of a proportional update scheme. According to Figure 1(d), the Reverse Update scheme reaches up to 94% of similarity with the Ideal Update, while the Two-Phase Update features only 24%.

[3]Our naive controller does nothing to assure the consistency of the update. All consistency warranties are assured by the evaluated update schemes.

VI. Conclusion

In this letter, we propose the Reverse Update scheme. The proposed scheme updates switch policies, switch-by-switch, on a Software Defined Networking, in the reverse sense of flow paths. We prove that our policy update scheme is per-packet consistent and, thus, the flow properties are preserved. It is worth mentioning that our scheme is simple and does not require packet tagging, which guarantees low processing overhead and reduced number of installed rules on the core of the network. The simulation of Reverse Update scheme in a SDN showed that the configuration overhead is close to an ideal update scheme. Moreover, the Reverse Update promptly updates the rules, presenting a similarity with the Ideal Update of 94%, which is up to four times higher when compared to the Two-Phase Update scheme.

References

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

[2] N. C. Fernandes, M. D. D. Moreira, I. M. Moraes, L. H. G. Ferraz, R. S. Couto, H. E. T. Carvalho, M. E. M. Campista, L. H. M. K. Costa, and O. C. M. B. Duarte, "Virtual networks: isolation, performance, and trends," *Annals of Telecommunications - Annales des Télécommunications*, vol. 66, no. 5, pp. 339–355, 2010.

[3] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test openflow applications," in *Proceedings of the USENIX NSDI'12*. Berkeley, CA, USA: USENIX Association, 2012, pp. 127–140.

[4] P. Perešíni, M. Kuzniar, N. Vasić, M. Canini, and D. Kostiū, "OF.CPP: Consistent packet processing for openflow," in *ACM SIGCOMM - HotSDN'13*. Hong Kong, China: ACM, 2013.

[5] M. Canini, P. Kuznetsov, D. Levin, S. Schmid *et al.*, "A distributed and robust SDN control plane for transactional network updates," in *The IEEE INFOCOM 2015*, Apr. 2015.

[6] S. Luo, H. Yu, and L. Li, "Consistency is not easy: How to use two-phase update for wildcard rules?" *Communications Letters, IEEE*, vol. 19, no. 3, pp. 347–350, Mar. 2015.

[7] R. McGeer, "A safe, efficient update protocol for openflow networks," in *ACM SIGCOMM - HotSDN'12*. Helsinki, Finland: ACM, 2012.

[8] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *ACM SIGCOMM - HotSDN'13*. Hong Kong, China: ACM, 2013.

[9] J. McClurg, H. Hojjat, P. Cerny, and N. Foster, "Efficient synthesis of network updates," in *ACM SIGPLAN - PLDI*. Portland, USA: ACM, Jun. 2015.

[10] L. H. G. Ferraz, D. M. F. Mattos, and O. C. M. B. Duarte, "A two-phase multipathing scheme based on genetic algorithm for data center networking," in *IEEE GLOBECOM 2014*, Dec. 2014, pp. 2270–2275.