FICUS: Fast Incremental Consistent Update in SDN based on Relation Graph

Qing Li*, Lei Wang*, Yong Jiang*, Guangwu Hu*, Mingwei Xu†,Qingmin Liao*
* Graduate School at Shenzhen, Tsinghua University, Shenzhen, China
† Tsinghua University, Beijing, China

***Abstract*—In Software Defined Networking (SDN), the configuration inconsistency during updates is one main source of network instability. An efficient updating scheme with configuration consistency is required. In this paper, we propose the scheme of Fast Incremental Consistent Update for SDN (FICUS) based on the relation graph (RG). In our scheme, we analyse the relation between update operations, construct the relation graph and find a proper order of these update operations to avoid inconsistency. To solve the problem, we define two types of relations: the path dependency relation and the path rejection relation. We evaluate our scheme and algorithms by comprehensive experiments. The results show that our scheme needs only 10%-40% of the rules compared with the two-phase update scheme and speeds up the update process by 40% in average.**

I. Introduction

As a future network architecture, Software Defined Networking (SDN) decouples the forwarding plane and the control plane, centralizes the intelligence of the network into the controller [1, 2]. The controller generates rules according to the routing/forwarding policies and distributes them into the switches. The switches are only responsible for forwarding packets according to the rules. SDN can be employed to simplify the management of the network [3–5]. In all of these systems, a reliable scheme is required to update the configuration (rules) fast and exactly.

Although SDN employs the approach of centralized controlling, it is still a distributed system in the respect of configuration updating. Every switch processes the packets according to its flow table. Therefore, during the configuration update, the flow tables of these devices may be inconsistent [6], which may cause forwarding loop, black hole, service disruption, etc. Therefore, to avoid these problems, consistency must be guaranteed during the network update.

Several schemes [7–11] have been proposed to guarantee the consistency of the configuration during the network transiting between two configurations. Two phase update (TP) [7] is proposed to divide the process of configuration update into two phases. This scheme can guarantee that every packet is processed by only one configuration (old or new) but never by the mixed during the network update. It ensures the consistent properties with no loop and black hole. However, it requires the network to reinstall the whole configuration (all rules) even if only a few rules are changed. Dionysus [11] improves the median update speed through the relations between the update operation, resource and path. But it only adapts to tunnel-based forwarding and WCMP forwarding. For data center networks, zUpdate [10] ensures congestion-free during network updates under asynchronous switches and traffic matrix changes. The core of zUpdate is a programming model which enables lossless transitions when the network changes.

According to the observations, consistency requirements impose some relations between the update operations. For example, switch S_n is the next hop of switch S_m in the configuration to be updated; S_m is the next hop of S_n in the old configuration. To avoid the forwarding loop, the update operation on S_n must be performed before S_m. Violating the updating order may lead to inconsistency and cause the forwarding loop. In this paper, we propose the scheme of Fast Incremental Consistent Update for SDN (FICUS), which is based on the relation graph (RG).

In FICUS, when a configuration update is invoked, we first analyse the relations between the update operations and generate a relation graph where the nodes are the update operations and the directed links are the relations between these operation. For the packet-level consistency, there are two types of relations: the path dependency relation and the path rejection relation. The path dependency is used to ensure the destination is reachable and the path rejection can guarantee that the packets are transferred correctly. We then design the efficient algorithms to find a proper order of these operations from the RG.

We implement our scheme using POX controller and Mininet [12] platform. The comprehensive experiment results show that: 1) FICUS needs only 10%-40% of the rules compared with the traditional update scheme; 2) FICUS installs the rules 40% faster on average.

The contributions of this paper can be concluded as follows:

- We analyse the relations between the update operations. For the packet-level consistent update, we define the path dependency relation and the path rejection relation.
- We propose a new consistent update scheme based on the relation graph. We construct a relation graph for the operations. With the help of relation graph, we can find a proper order to install these update.
- The comprehensive experiment results show that FICUS makes a significant improvement on reducing the required rules and the updating time.

The remainder of this paper is organised as follows. In Section II, we show the network update model. In Section III, we describe our consistent mechanism based on the relation graph. In Section IV, we present the preliminary results of our simulation. In Section V, we analyse the related works. Finally, we give the conclusion and discussion in Section VI.

II. The Update Model

In this section, we present the model to describe the network update process clearly. In our model, we abstract the packet as a point and the rule as a hypercube in the match space.

Rule, r. In our discussion, we define a rule as a tuple $r = (m, a, pri)$. The match field m identifies the rule uniquely. a is the action of the rule. The pri is an integer representing the priority of the rule. Let rs be the default rule set in our discussion. We use ms to denote the match field set and ms^i to denote the match set of the rules with the priority of i and use $\hat{m^i}$ to denote the match set of the rules with the priorities bigger than i.

Topology, T. We define the **network position** lp as a two-tuple (s, p) where s is the switch and p is the ingress port of s. Let LP be the set of all the possible network positions from the perspective of data plane (the controller is not included). We define the topology function as follows: $T(lp) = lp'$ denotes that lp and lp' are connected in the topology.

Configuration, C. In SDN, the configuration C is the collection of rules which are installed in the switches. Thus the configuration can be described as a set $\{(lp_i, rule_i)|lp_i \in LP, rule_i \in rs\}$.

Network, N. The network N can be defined as a tuple (C, T). Topology T denotes the physical connection relationships between the network devices, and Configuration C contains the forwarding behaviors in the data plane.

Network Update Process. In SDN, an update operation must have a location where the rule is installed or deleted. We define **update operation** u as (lp, r). It means that the new rule r should be installed at the network position of lp. For a packet pkt, after pkt leaves the network, it has a trace t. **Trace** t is an ordered list $([(lp_j, r_j)])$ where lp_j is the jth network position along the path of packet pkt and r_j is the corresponding rule for the packet at lp_j. The expression $lp_j \in t$ means lp_j appears in the trace t. It is similar for $r_i \in t$ and $u_i \in C_i$. Due to the definition of update operation, a trace can also be described as an ordered list of update operation, i.e., $[u_j]$. The trace of a packet shows the behaviors in the data plane, thus we can get the current trace and infer the trace after update operations.

The network update process can be defined as:

$$\begin{aligned}
&if\ C' = override(C, us)\\
&then\ N \xrightarrow{C'} N'\\
&where\ N = (C, T),\ N' = (C', T)
\end{aligned}$$

The function $override(C, us)$ means: for the two-tuple $u = (lp, r), u \in us$, if $u \notin C$, add u to C; if $u \in C$, do nothing. We set a higher priority for the updating (new) rules compared with the old ones. The expression $N \xrightarrow{C'} N'$ means under the configuration of C', the network N changes to N'.

Consistent property. In SDN, the update message is forwarded to the switch from the controller and then the switch updates its flow table accordingly. The switch is only responsible for forwarding packets. The consistent property should be guaranteed during the update as forwarding faults may occur in the chaos of old and new configurations.

For different networks, there are different consistent property requirements. Generally, the connectivity should be firstly guaranteed in the network. In our paper, we first propose the path dependency relation between the update operations, which is used to guarantee the switch can forward the packet correctly when the packet comes in. It can guarantee the loop and black-hole free property during the update. We then propose the path rejection relation between update operations to guarantee the packet-level consistent property. For different consistent properties, different relations can be provided. Then a relation graph (RG) can be generated based on these relations and a proper update order will be discovered.

III. Fast Incremental Consistent Update

In this section, we propose the scheme of Fast Incremental Consistent Update for SDN (FICUS). In software defined networking, the controller translates the policy into the configuration and then installs them into the switch. The basic idea of our mechanism is analysing the relationship between the update operations and finding a proper order to install them incrementally. In our mechanism, we guarantee the packet-level consistent property.

A. Construct Relation Graph

Let N and N' be the old and new network. Let us be an update sequence. Trace t' belongs to network N'. $location(u, t)$ is a function, which returns the network position lp of the input update operation u in the trace t.

Definition 1 (Path dependency). *We set the conditions as follows:*

- $C' = override(C, us)$
- $N \xrightarrow{C'} N'$
- $u_i,\ u_j \in\ us$
- $u_i,\ u_j \in\ t'$
- $location(u_j, t') < location(u_i, t')$

Then we define: $u_i \overset{d}{\rightarrow} u_j$*, i.e.,* u_j *depends on* u_i *and* u_i *should be installed before* u_j.

In this definition, we impliedly define a partial ordered relation of network position. $lp_j < lp_i$ means if a packet flow according to the trace, it will come to network position lp_j earlier than lp_i. A path dependency ($u_i \overset{d}{\rightarrow} u_j$) means if two update operations belong to the same trace, the later operation depends on the former one. This relation exists because the former operation works before the later one. During the updating, u_i should be installed before u_j.

For example, in Figure 1, the updating rule at $S2$ depends on the updating rule at $S1$. For simplification, we use $[X, Y]$ to denote the rules. $[X, Y]$ means if the destination of the packet is X, forward it to the switch Y. Therefore, if $[S4, S1]$ at $S2$ is installed before $[S4, S3]$ at switch $S1$, the old configuration with low priority at $S1$ still works. Then the packets will be forwarded back to $S2$, and it will cause the forwarding loop between $S1$ and $S2$. According to the Definition 1, in the new trace t' after update, the packet will pass network position $(S2, x)$ earlier than $(S1, y)$, we should update the rules in $S1$ first. x and y are the ingress port of the switches for the incoming packet.

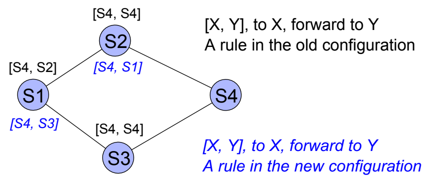

[S4, S4]
S2
[S4, S1]
[X, Y], to X, forward to Y
A rule in the old configuration
[S4, S2]
S1
[S4, S3]
S4
[S4, S4]
S3
[X, Y], to X, forward to Y
A rule in the new configuration

Figure 1. Path Dependence of Two Updating Rules.

Beside of the update operations appearing in the same trace, we show another scenario. Let N and N' be the old and new network. Let us be an update operation sequence. There are two update operations $u_i = (lp_i, r_i)$ and $u_j = (lp_j, r_j)$. The packet p follows trace t_p in network N and t'_p in N'.

Definition 2 (Path rejection). *The conditions are set as follows:*

- $C' = override(C, us)$
- $N \xrightarrow{C'} N'$
- $u_i,\ u_j \in\ us$
- $u_i \in\ t'_p,\ u_j \notin\ t'_p, r_j \in u_j$
- $p \rhd\ r_j$
- $u_i,\ u_j \in\ t_p$
- $location(lp_i, t_p) < location(lp_j, t_p)$

Then we define: $u_i \xrightarrow{r} u_j$, *i.e.,* u_i *rejects* u_j *and* u_i *should be installed before* u_j.

The symbol $p \rhd r_j$ (defined in the previous section) denotes the rule r_j has the highest priority among all the rules which can match the packet p, and it performs when the packet p arrives. Suppose in switch s_j, there exist the certain packets following the old configuration. An update operation u_j at s_j may influence these packets. If another update operation u_i at switch s_i is performed, the packets that belong to this flow will not pass the switch s_j any more. We say that u_i rejects u_j. The relation between these two operations is path rejection. For example, in Figure 2, the new routing policies include: the packets to $S4$ from $S5$ should be dropped and the packets to $S4$ from $S1$ should be forwarded to $S3$. Two updating the update operations including $[S4, S3]$ at $S1$ and $[S4, Drop]$ at $S2$ can achieve the goal of these policies. By Definition 2, $[S4, S3]$ at $S1$ rejects $[S4, Drop]$ at $S2$. If the later one is installed first, packets from $s1$ will be dropped at $S2$. This behavior violates the routing policy, and generates a black hole during the process of upgrading rules.

Given the initial network configuration C, based on C, if any two update operations u_i and u_j have the relation of $u_i \xrightarrow{d/r} u_j$ ($u_i \xrightarrow{d} u_j$ or $u_i \xrightarrow{r} u_j$), we say that a partial order exists between them: $u_i \prec u_j$.

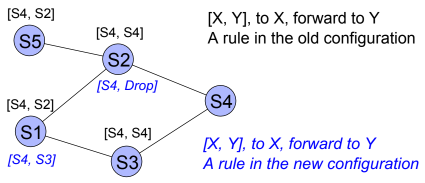

[S4, S2]
S5
[S4, S4]
S2
[S4, Drop]
[X, Y], to X, forward to Y
A rule in the old configuration
[S4, S2]
S1
[S4, S3]
S4
[S4, S4]
S3
[X, Y], to X, forward to Y
A rule in the new configuration

Figure 2. Path Rejection of Two Updating Rules.

Table I
UPDATE OPERATIONS

Name	Position	Rule		Priority
		Match	Action	
U_1	(S1,3)	SIP:192/8	To 1	H
U_2	(S2,1)	SPORT:22	To 2	H
U_3	(S3,1)	SPORT:22	To 3	H
U_4	(S1,3)	SIP:128/8	To 2	H

Definition 3 (Update Partial Order $\prec$). *The update partial order* $\prec$ *between two update operations* u_i *and* u_j *means that* $u_i \prec u_j \Leftrightarrow u_i \xrightarrow{d/r} u_j$.

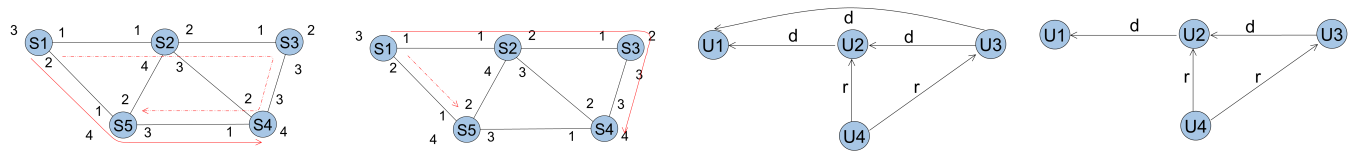
To be simple, we use $(u_i, u_j)_C$ to represent the partial order of u_i and u_j based on the initial network configuration C.

The partial order $u_i \prec u_j$ means u_i should be updated before u_j. If the relation is dependency, the packets are forwarded from lp_j to lp_i. Otherwise the relation is rejection, the packets will pass lp_i and do not pass lp_j. However, we will always update u_i firstly regardless the flow direction. In other words, we can construct a directed graph based on the partial order. This directed graph is the **relation graph (RG)**. According to the RG, we can schedule the order of update operations.

As Figure 3 shows, there are five switches and two flows in the network. The match fields of the two flows are source IP: 192/8, port: 22 (denoted by solid line) and source IP: 128/8, port: 22 (denoted by dotted line). The behaviors of these two flows are shown in Figure 3(a) and after the configuration updates, they change as Figure 3(b) shows. The priority of rules in the old configuration is L. The update operations are in Table I. The relation of these update operations are shown in Figure 3(c).

It is straightforward that given an initial configuration and the update configuration, we can analyse the path dependency relation and the path rejection relation. Then we can construct the relation graph accordingly, as Figure 3(c) shows. In Figure 3(c), the update operations are nodes and the relations are edges. In the RG, the relationship in the edges is either path dependency or path rejection. The edge direction is set according to the partial order of the update operations. We design the algorithm of **Generate-RG** to describe the relations between the update operations and generate the RG.

In Algorithm 1, the node of the directed graph is a tuple of network position and rule. Every node has the match space as the attributes denoted by m_i. We use a queue to store the visited nodes and traverse the other nodes by BFS (Breadth First Search). If we initialize the queue with update operations



(a) Current State (b) Target State (c) Relation Graph (d) Simplified Relation Graph

Figure 3. A configuration update example.

us and assign the new configuration to rs, we will get the traces which belong to the new network. Through these traces, we can get the path dependency relations between these update operations. If we initialize the queue with the old rules which will be replaced by the update operations and assign the old configuration to rs, we will get the old traces. Comparing with the old and new traces, we can get the path rejection relation. Based on the two relations, we can add the link and construct the RG. In this algorithm, the operation judging the relation type of the link is included in the function addLink.

Algorithm 1 Generate-RG

```
1: # Graphic Node:$(lp_i, r_i)$, Node attribute: $m_i$
2: # Queue:$st = [(lp_i, r_i)]$
3: # The start points of the path are stored in the queue
4: while st.front()$\neq \phi$ do
5:     $(lp_i, r_i)$ =st.dequeue()
6:     # traverse all the outports of packets
7:     for $lp'_i = T(lp)$ do
8:         if $lp'_i \neq \phi$ then
9:             # traverse all the next rules
10:            for $r'_i \in \bigcap_{rs}(r_i)$ do
11:                RG.addNode($lp'_i, r'_i, re(m'_i)$)
12:                # $re(m'_i) = m'_i - m'_i \bigcap m_i'^{pri}$
13:                RG.addLink($(lp_i, r_i), (lp'_i, r'_i)$)
14:                st.enqueue($lp'_i, r'_i$)
15:            end for
16:        end if
17:    end for
18: end while
```

B. Process Simplified Relation Graph

Theorem 1. *Suppose configuration C turns into configuration C' as a set of update operations us are installed. If the relation graph is a Directed Acyclic Graph (DAG), topological sorting for this DAG can get an ordered list of update operation nodes. The order of this list is a proper order for scheduling the update operations.*

The proof of this theorem is straightforward. The definition of partial order determines the installation order of two update operations. Based on the topological sorting, the ordered node list do not break the partial order. Because the node in the list with the a larger index is impossible to have a directed edge pointing to the node with a smaller index. That means the order of this list conforms with the partial order strictly.

Lemma 1. *Only take into account path dependency relation, there are no loop in the RG for a single trace.*

Proof. According to the definition of trace, it is a ordered list of $[(lp_j, r_j)]$ where lp_j is the jth network position along the path of packet pkt_i and r_j is the corresponding rule for the packet at lp_j. Because black hole and loop free are the basic properties for packet forwarding and the trace is the equivalent of the data plane path, the network positions are different from each other in the trace, i.e., there is no repeated network position for a single trace. It is straightforward that loop free can be guaranteed in the RG for a single trace. □

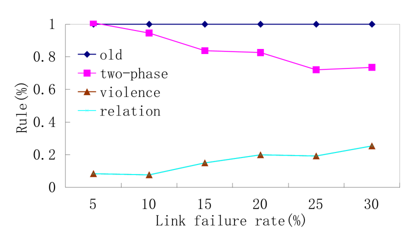
Loop free property cannot always be guaranteed especially for multiple traces. However, for some networks, if the strong consistency property is not required, the path rejection relation can be ignored. The path dependency relation can guarantee that there are no loops or disconnections. If a weaker consistency property is required and some inconsistencies can be accepted, we can treats the path rejection relation as the path dependency relation. In these situations, we can ignore the rejection relation in the loop. For some networks that need a very strong consistency property, we use the two-phase scheme [7] to update the left operations. We omit the details for this step due to the space limitation.

IV. Evaluation

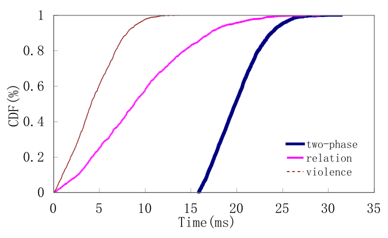
In this section, we use the platform Mininet[12] and POX controller [13] to emulate the performance of our work FICUS (marked as relation in the figures).

The topology of the network is ”Chinanet” from the topology zoo[14]. In this topology, there are 43 nodes and 66 links. First, we put 12 hosts into this network and connect each one to a random switch in the topology. First, we use the prefixes in the BGP RIB table (Jan 1st, 2015) from the RouteView Project [15] and the shortest routing application to generate the configuration (We add 12 outports to the network and distribute these prefixes to the outports). Second, we put 12 hosts into the network and use the shortest routing (broadcast) application to generate the configuration. The shortest routing application computes the shortest path between the hosts and generates the configuration. The broadcast application uses the spanning tree algorithm to generate the configuration. Third, the traffic subjects to Poisson distribution. The inter-arrival times are exponentially distributed and the durations are power law distribution.

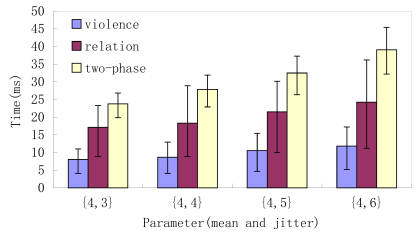
To simulate the configuration update, we disable the links to trigger the controller to regenerate the configuration using



(a) Rule VS Link Failure Rate

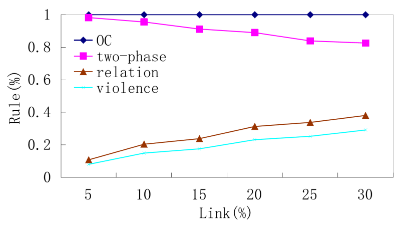


(b) Completion Time

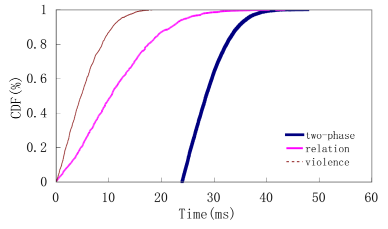


(c) Completion Time of 50th, 90th, 99th

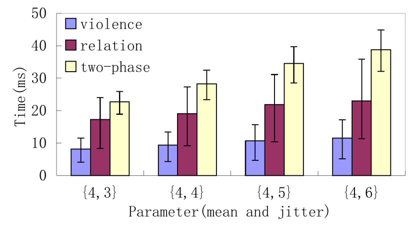
Figure 4. Shortest Path Routing



(a) Rule VS Link Failure Rate



(b) Completion Time



(c) Completion Time of 50th, 90th, 99th

Figure 5. Broadcast Routing

the shortest path routing or the broadcast application. For each experiment, we tested three update mechanisms: (1) the violence mechanism is that the controller installs the update operations in the network, which is optimal in terms of update speed and does not care the consistent property. These operations are in the new configuration and not in the old configuration; (2) the two-phase mechanism proposed in [7]; (3) our incremental mechanism based on the relation graph (RG).

As Figure 4(a) shows, we count the update operations installed of three mechanisms. We use the update operations of the old configuration as the basis. As the link failure rate increases, the network will be cut into more connected components and the update operations of the new configuration will be less. The number of rules in our mechanism is nearly the number in the violence mechanism. Even the link failure rate is 30%, the number of rules in our mechanism or the violence mechanism is less than half of rules in the two-phase mechanism. As the shortest path routing application computes the shortest path between the hosts and generates rules for each path. In other words, there are no path rejection relations in this test. So the number of rules in violence and our relation graph schemes are equal.

In Figure 4(b), we set the parameters of the normal distribution are 4 and 5 (the mean is 4 and the jitter is 5). The link failure rate in this test is 10%. The violence mechanism costs 14 ms to update the configuration. The time of our relation graph mechanism is 29 ms and the two-phase mechanism is about 31 ms. When the two-phase mechanism begins, 85% rules in our mechanism have been installed.

We also compare the three mechanisms, by varying the jitter of the normal distribution. Figure 4(c) shows the 50th, 90th and 99th percentile update completion time, under different parameters of the normal distribution. The pair $\{a, b\}$ means a is the mean and b is the jitter. For example in $\{4, 3\}$ case, our mechanism is 115%, 38%, 15% faster than the two-phase mechanism. As the jitter increases, the completion time of these three mechanisms all increases, but our mechanism is still faster than the two-phase mechanism.

We regenerate the new configuration using the broadcast application during the network changes. In our relation graph mechanism, we break the circle of the dependency path relation, find the proper order through the relation graph and update the left update operations using the two-phase mechanism. In Figure 5(a), the rules installed in our scheme or violence scheme are both less than the two-phase scheme. Figure 4(a) and 5(a) show that the incremental scheme is more suitable for the small-scale update. That is because in the incremental scheme most of the rules which are both in the old and new configurations are not installed again. In Figure 5(b), we set the parameters of the normal distribution are 4 and 5 too. The link failure rate in this test is 10%. When the two-phase mechanism begins, 85% rules in our mechanism have been installed. Figure 5(c) shows the 50th, 90th and 99th percentile update completion time under different parameters of the normal distribution. As the jitter increases, the completion time of all the three mechanisms increase, but our mechanism is still faster than the two-phase mechanism.

V. Related Work

As SDN is used in the cloud computing and data centers [16, 17], many works have been proposed to maintain consistent properties during configuration update in SDN.

In the paper of [6], the consistent update has been explored first and two criteria of consistent update mechanism (packet consistency and flow consistency) are proposed. Reitblatt [7] provides a network model and the two-phase commit for the packet-level consistent update. To guarantee the flow-level consistency property, the dividing periodically mechanism is proposed. But during the configuration update, the two-phase commit needs double FIB because it installs the old and new configuration in the network at the same time. An incremental update algorithm [18] which makes a trade-off between the time required and the TCAM needed. It divides the update into several rounds and completes a part of configuration in each round.

In SDN, the consistent update has been explored in [19]. The authors establish two criteria of consistent update mechanism: per-packet consistency and per-flow consistency.To implement per-packet consistency, the controller stamps packets with their configuration version at ingress switches and tests for the version number in the core network. [7] provides a network model and the two-phase commit for per-packet consistency. When packets arrive the network, the egress switch stamps packets with a version number. In the core of the network, all rules in switches use the version number as a match field and affect packets with the same version number. After packets walk through the network, the egress switch strips the version number from packets. A weakness of this mechanism is that during the configuration update, the old and new rules are both installed in the network, and consume too much TCAM.

VI. Conclusion and Future Work

In this paper, we present a consistent mechanism based on the relation graph, which makes a significant improvement on reducing the rules installed and the update time required. We define the path dependency relation and the path rejection relation. Then we analyse the relations between the update operations and generate the relation graph. Using RG, we can find a proper order of these operations. The results of our simulation show that our work makes a significant influence on reducing the update time required and rule space used.

In the future, we will process the circle in the RG in serval situations. Efficient circle divide algorithm needs to be design. Besides, we will test our mechanism under more real topologies and more complex update process in both simulation environment and test bed.

VII. Acknowledgement

This work is supported by the National Natural Science Foundation of China under grant No. 61402255, the Guangdong Natural Science Foundation under grand No. 2015A030310492, the R&D Program of Shenzhen under grant No. ZDSYS20140509172959989, No. JCYJ20150630170146830.

References

[1] W. Xia, Y. Wen, C. Foh, D. Niyato, and H. Xie, “A survey on software-defined networking,” *IEEE Communications Surveys Tutorials*, vol. 17, no. 1, pp. 27–51, 2015.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “Openflow: enabling innovation in campus networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[3] H. Kim and N. Feamster, “Improving network management with software defined networking,” *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[4] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, “Achieving high utilization with software-driven wan,” in *Proceedings of ACM SIGCOMM*, Hong Kong, Aug. 2013.

[5] M. Bredel, Z. Bozakov, A. Barczyk, and H. Newman, “Flow-based load balancing in multipathed layer-2 networks using OpenFlow and multipath-TCP,” in *Proceeding of ACM SIGCOMM HotSDN*, Chicago, USA, Aug. 2014.

[6] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, “Consistent updates for software-defined networks: Change you can believe in!” in *Proceedings of ACM Workshop on HotNets*, Cambridge, USA, Nov. 2011.

[7] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, “Abstractions for network update,” in *Proceedings of ACM SIGCOMM*, Felsinki, Finland, Aug. 2012.

[8] P. Peresini, M. Kuzniar, N. Vasic, M. Canini, and D. Kosti, “OF.CPP: Consistent packet processing for openflow,” in *Proceeding of ACM SIGCOMM HotSDN*, Hong Kong, Aug. 2013.

[9] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, “Software transactional networking: Concurrent and consistent policy composition,” in *Proceeding of ACM SIGCOMM HotSDN*, Hong Kong, Aug. 2013.

[10] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, “zUpdate: Updating data center networks with zero loss,” in *Proceedings of ACM SIGCOMM*, Hong Kong, Aug. 2013.

[11] 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 ACM SIGCOMM*, Chicago, USA, Aug. 2014.

[12] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: Rapid prototyping for software-defined networks,” in *Proceedings of ACM Workshop on HotNets*, Monterey, USA, Oct. 2010.

[13] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky, “Advanced study of SDN/OpenFlow controllers,” in *Proceedings of the 9th Central Eastern European Software Engineering Conference in Russia*, Moscow, Russia, Oct. 2013.

[14] “The internet topology zoo.” [Online]. Available: http://topology-zoo.org

[15] “The route views project.” [Online]. Available: http://www.routeviews.org

[16] M. Banikazemi, D. Olshefski, A. Shaikh, J. Tracey, and G. Wang, “Meridian: an SDN platform for cloud network services,” *IEEE Communications Magazine*, vol. 51, no. 2, pp. 120–127, 2013.

[17] J. Zheng, H. Xu, G. Chen, and H. Dai, “Minimizing transient congestion during network update in data centers,” in *Proceedings of ACM CoNEXT*, Sydney, Australia, Dec. 2014.

[18] N. P. Katta, J. Rexford, and D. Walker, “Incremental consistent updates,” in *Proceeding of ACM SIGCOMM HotSDN*, Hong Kong, Aug. 2013.

[19] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, “Consistent updates for software-defined networks: Change you can believe in!” in *Proceedings of ACM Workshop on HotNets*, Cambridge, USA, Nov. 2011.