Computer Networks 92 (2015) 287–299

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

SARD: A Smart Approach of Rule Division for fast flow-level consistent update in SDN

Qing Li[a,*], Kun Zhao[a], Yong Jiang[a], Mingwei Xu[b], Shu-Tao Xia[a]

[a] Graduate School at Shenzhen, Tsinghua University, Shenzhen, Guangdong, China
[b] Tsinghua University, Beijing, China

A R T I C L E I N F O

Article history:
Received 21 November 2014
Revised 30 May 2015
Accepted 20 September 2015
Available online 3 October 2015

Keywords:
Software Defined Networks
Configuration update
Flow-level consistency

A B S T R A C T

In Software Defined Networks (SDN), the configuration inconsistency during updates is one main source of network instability. Even if the validity of the initial and final configurations is guaranteed, the interim complex and inconstant network states might cause routing conflicts and transmission disruption. Therefore, an efficient updating scheme with configuration consistency is required. Current schemes well guarantee the packet-level consistent update, but perform poorly for the flow-level case.

In this paper, we present a Smart Approach of Rule Division (SARD) for fast flow-level consistent update in SDN. We first provide a simplified mathematical model of the network. Based on this model, we then propose SARD to guarantee the flow-level consistency during configuration updates. In SARD, the controller (1) collects the information of existing flows; (2) computes K optimal prefixes covering these flows, meanwhile with the minimized space; (3) installs the new rule and K old sub-rules with lower and higher priorities respectively. SARD preserves the flow-level consistent property and accelerates the process of configuration update in SDN. We evaluate the performance of SARD by comprehensive experiments. The results show that our scheme reduces the transition time to about 10% of the current method of periodical direct division.

1. Introduction

The network is a complicated and volatile system, i.e., the topology and the traffic are dynamically changing. To maintain network stability and provide reliable transmission service, the operators update the network configuration constantly according to network states. However, even if the static configurations are valid, some pathological states are inevitable during the indeterminate dynamic changes [1,2]. In a traditional network, updates of routes are independent and asynchronous among the forwarding devices when network events occur. The behaviors of the network are indefinite and uncontrollable.

Software Defined Networking (SDN) is proposed to enhance the controllability and manageability of the network [3]. SDN employs centralized controlling, and thus mitigates the problem of anomalies during network changes. SDN decouples the forwarding and controlling panels, centralizes the intelligence of the network into the controller. The controller determines the global routing policies and disseminates them to the switches. However, the controller still cannot ensure synchronous updates in the switches. As the network environment is extremely complicated, the update installation times in different switches are unpredictable. Therefore, the configurations in different switches might be inconsistent before the update is installed in the whole network, which may cause forwarding loops, packet dropping,

* Corresponding author. Tel.: +86 18038153239.
E-mail address: li.qing@sz.tsinghua.edu.cn (Q. Li).

http://dx.doi.org/10.1016/j.comnet.2015.09.025
1389-1286/© 2015 Elsevier B.V. All rights reserved.

service disruption, etc. The problem to be solved is ensuring that **each packet (or flow) is handled by the same configuration (the old one or the new one, but not the mixed) in the SDN network.**

In order to solve this problem of consistent configuration update, researchers have proposed several protocols and schemes, aiming to prevent transient anomalies during updating. A safe protocol in [4] argues that the packets affected by the network change should be sent to the controller, and then be re-leased to the network after the change process terminates. The two-phase updating scheme based on version controlling introduced in [5] solves the problem of packet-level configuration consistency in SDN very well. Katta et al. propose another scheme of incremental consistent update [6]. Hong et al. use a series of intermediate configurations to satisfy the rule space and bandwidth constraints during network changes [7]. Most of these proposed solutions can only guarantee the packet-level consistency. However, the flow-level consistency is required for all the connection-oriented applications. Reitblatt et al. propose the scheme of dividing rules into pieces and installing the pieces in switches periodically [5]. But an extremely long transition time is required in this scheme, which may cause severe transmission problems especially if the updating is caused by congestion. Therefore, an efficient (faster) scheme to guarantee the flow-level consistent update is required.

In this paper, we present a new Smart Approach of Rule Division (SARD) for fast flow-level consistent update in SDN. In SARD, we employ the algorithm of K optimal prefix covering to quickly decompose the old rules into small pieces. The K sub rules with the smallest space covering the current existing flows are generated, which guarantees the flow-level consistency. Meanwhile, the new rule takes effect in the other remained space (the space of the old rule minus the space of all the sub rules) immediately, which makes the updating more efficiently.

SARD mainly consists of four steps: (1) send the packets affected by the update to the SDN controller; (2) the controller computes K optimal prefixes covering the source IPs of the existing flows based on the information of these collected packets; (3) the controller generates K sub rules (old) according to the computed K prefixes; (4) these generated sub rules and the new rule are installed immediately into switches respectively with the higher and lower priorities (meanwhile the original old rule is deleted). After the four steps, the existing flows are forwarded by the generated sub rules (consistent with the original old rule) and the flow-level consistent property is preserved. The other new-coming packets are forwarded by the new rules immediately.

As Ternary Content Addressable Memory (TCAM) is an expensive and rare component in the switch, we further process the update by multiple rounds, with a smaller K for each round. To prevent some sub rules with bigger spaces from never disappearing, we divide these sub rules in the next round by K prefix covering algorithm.

We propose an update network model and use this model to prove that our scheme can preserve the flow-level consistent property. To construct the K sub rules, K-prefix covering algorithm uses the source IPs of existing flows to construct a binary tree and computes the K optimal prefixes with the minimal total covering space by dynamic programming.

We evaluate the performance of our scheme by comprehensive experiments. We implement SARD using the platform of POX controller and Mininet [8]. We also run the algorithm in the real topologies of ChinaNET, IBM and York [9]. The results show:

- SARD reduces the transition time (from the old configuration to the new configuration) to about 10% of the method of dividing periodically in [5], while maintaining the flow-level consistency.
- In our scheme, about 4012 flows are redirected to the controller during the update in the small topology as Fig. 2 shows. Although some extra burdens are brought to the controller, using the powerful controller to strengthen the network is reasonable and effective.

The remainder of this paper is organized as follows. In Section 2, we show the background and motivation. In Section 3, we develop a simple, formal network update model. This model allows us to describe the forwarding operation of packets and the update process of the configuration formally. In Section 4, we provide the flow-level consistent mechanism, and we also propose some optimization to avoid the network congestion and ease the burden of the controller. In Section 5, we present the minimum prefix covering algorithm and the corresponding complexity analysis. In Section 6, we discuss how our mechanism works in the SDN network with multi-controllers. In Section 7, we present the evaluation results of SARD and the scheme of dividing periodically. In Section 8, we review the related works. Finally, we end the paper by the conclusion and discussion in Section 9.

2. Background and motivation

Computer networks are composed of many different devices such as routers, switches and middleboxes, which are implemented with numerous complicated protocols. Simultaneously, operators have to accomplish some complex tasks with limited tools, i.e., translating high-level policies to low-level forwarding rules. As a result, the management of network is always a significant challenge with its error-prone property [10]. Another unsurmountable problem is that networks are extremely difficult to evolve with a wide deployment. These challenges promote the origin of openflow [11], which aims to provide a simple and feasible approach to manage the networks (especially the enterprise network) [12,13].

SDN allows network operators to manage networks like a local resource through the abstraction of devices [14]. This is achieved by decoupling the intelligence from the forwarding panel and centralizing it in the control panel. The control panel collects statistics of the network from the forwarding panel, provides the global network view and application interface to the operators. The forwarding panel is only responsible for packet forwarding. Network operators can implement their custom algorithms in the controller, generate rules to configure the switches and thus control traffic forwarding.

Contrast to the traditional network, SDN is a centralized system in the control panel. However, in the forwarding panel it is still a distributed system. Each switch forwards packets according to its flow table independently. This means there may also exist inconsistent states in SDN, which will cause some serious problems, like the security problem, network congestion and service interruption. Therefore, the configuration consistency must be guaranteed during updates in SDN. That is, each packet (or flow) must be handled by the same configuration (the old one or the new one).

The consistency of configuration update can be classified into two types: the packet-level consistent update and flow-level consistent update [15]. The packet-level consistency can only satisfy the requirements of the connectionless applications. For the connection-oriented applications, the flow level consistency is generally indispensable, as configuration switching from old to new may interrupt the current connection.

Two-phase commit [5] is employed to accomplish the packet-level consistent configuration update. In [5], the controller divides the network nodes into two domains: the external network domain and the internal network domain. Accordingly, the updates can be classified into internal updates and external updates. For each configuration change, (1) the internal part will be first updated with the new version of configuration; (2) then the external part will be updated. Before the second step, the ingress switch will forward the incoming packets by the old configuration and label the packets with the old version. All the internal switches will forward the packets according to the labeled version. In this way, the controller can distinguish the configurations and the packet-level consistent update is achieved. However, this is not enough for connection-oriented applications.

There are several approaches for flow-level consistent configuration update, including switch rules with timeouts, wildcard cloning, end-host feedback, etc. [5]. Wildcard cloning needs the switch to detect the existing flows and install the cloning rules for these flows. Nonetheless, as the switch is simple and has little intelligence in the SDN network, wildcard cloning is not feasible. End-host feedback needs the host to notify the controller about the existing flows. The communication between the host and controller will bring some security threats to the controller. Meanwhile, involving the end hosts for the network function is not reasonable.

In the approach of rules with timeouts, the old switch rule is divided into smaller sub rules and the sub rules will be erased periodically if no covered flow exists after the timeout. The controller refines the old rule and installs the smaller sub rules in the external switch with a soft timeout. After one period, the refined sub rules will be deleted as the existing flows drain out the network. If the refined sub rules have not been erased, the controller will refine them in the next period. The covering space size of a sub rule determines the possibility of erasure. Larger rules need smaller memory and longer transition time; smaller rules need larger memory and shorter transition time. Generally, the TCAM of a switch has limited memory. Therefore, this approach needs a long transition time, which may cause severe transmission problems. Especially when the updating is caused by congestion, the overloaded server may fail and the service may be interrupted.

To accelerate the flow-level consistent update in SDN, we present a Smart Approach of Rule Division (SARD) for fast flow-level consistent update in this paper. SARD improves the two-phase commit in [5]. It forwards the affected packets to the controller. Then the controller uses the minimum prefix covering algorithm to compute the K optimal prefixes and generates the corresponding K sub rules. SARD can achieve the fast flow-level consistent configuration update, meanwhile avoids the explosion of the flow tables in switches.

3. The update model

In this section, we present a simple mathematical model with the essential features of the SDN network. We also define the process and update operations of the network based on the model. The relation $N \xrightarrow{us} N'$ is used to describe the network update, where N is the network before the update and N' is the network after some number of update operations. Intuitively in SDN, the update operation should be regarded as a control message between the controller and the switch, which directs the switch to update its flow table or send some special packets. In this paper, it is the message to installing a new rule in the flow table. The main purpose of our model is to compute the traces of packets which contain the paths of the packets. According to these traces, we can find whether the consistent property is preserved during the network configuration update.

Notations. We use standard notations to denote a set: the notation $\{x_1, x_2, x_3\}$ means a set including the elements of x_1, x_2 and x_3. $[x_1, x_2, \ldots, x_k]$ is a list of k elements. A tuple which contains a pair of elements is expressed by the notation (x_1, x_2).

The operation $T_1 \to T_2$ denotes a function with the type of T_1 as the input and the type of T_2 as the output result, while $T_1 \xrightarrow{C} T_2$ means this function works under the condition of C. The operation $A++B$ means appending the element B to the list A.

Basic structures. As Fig. 1 shows, we define the syntax of the elements of the network update model. In the OpenFlow protocol, several fields (domains) are used to match the packets, including the ethernet source address, the ethernet destination address, the VLAN id, etc. The index key is a set composed of these domain names. The value is a sequence of bits, where a bit is either 0 or 1. A header is a list of fields and values, which is used to match the forwarding rules in the switch. A packet contains two parts: a header and the payload (data). Obviously, different types of packets have different headers which contain different domain values. Some headers may contain all the domains in the set *Key*, others may only contain some of the domains. For example, the header of a TCP packet has the domain transport source port, but the header of a LLDP packet does not. A rule consists of a match and some actions. A match is also a list of domains and values. There are some differences between a match and a header. For example, in a header the domain IP source address is a sequence of bits, but in a match it may contain a mask. In our model, we break a specific domain into two parts (one part is the value domain and another is the length domain which is never appears in a header). Apparently different rules have different domains. If a rule does not have a certain domain, it can match any value of the domain. In the SDN network, each switch or port has an unique number,

Bit	$b ::= 0 \mid 1$
Key	$k \in \{sip, dip, ...\}$
Value	$v ::= [b_1, ..., b_k]$
Header	$h ::= [k : v]$
Match	$m ::= [k : v] \mid \phi$
Packet	$pk ::= (h, data)$
Rule	$r ::= (m, action)$
Port	$p ::= 1 \mid ... \mid k \mid world \mid drop$
Switch	$s ::= 1 \mid ... \mid k$
Update	$u ::= (p, r)$
Trace	$t ::= [(s, rules)]$
Topology	$T \in p - \ p \rightarrow$
Configuration	$C ::= [(s, rules)]$
Located Packet	$lp ::= (p, pk, t)$
Switch Function	$SF \in lp - \ located\ packets$
Network	$N ::= (C, T, located\ packets)$

$\boxed{\text{Network PROCESS}}$

$if\ lp\ in\ N$
$and\ [lp_1, lp_2, ..., lp_k] = SF(lp)$
$and\ t_i = t + +(S, action\ of\ SF), for\ i\ in\ range(1, k)$
$and\ delete\ lp\ from lps$
$and\ lps' = override(lps, [lp_1, lp_2, ..., lp_k])$
$then\ (C, T, lps) \xrightarrow{lp} (C, T, lps')$

$\boxed{\text{Network Update}}$

$if\ C' = override(C, u)$
$then\ ((S, rs), T, lps) \xrightarrow{u} ((S, rs'), T, lps)$

$\boxed{\text{Packet-level Consistent Property}}$

$N \xrightarrow{us} N'$
$for\ any\ packet\ p,\ leave\ the\ network$
$t_p \in N\ or\ t_p \in N'$

$\boxed{\text{Flow-level Consistent Property}}$

$N \xrightarrow{us} N'$
$for\ all\ packets\ ps \in\ a\ flow,\ leave\ the\ network$
$t_{ps} \in N\ or\ t_{ps} \in N'$

Fig. 1. The update model: syntax and semantics.

which can be used to identify them. An update is a message which notifies a local switch to install a new rule.

The switch function is a packet processor that takes a packet coming from an inport as an input and generates several packets as a result and forwards these packets to outports. The switch function may modify the contents of the packet, generate one packet to forward, produce several packets to multicast or drop this packet. Following the work of Kazemian et al. in [16], a network is the composition of two simple behaviors: (1) forwarding the packet as the switch function directs and (2) moving the packet from one switch to another through the link. The topology function maps one switch to another if there is a link between these switches in the network. As the controller has a global view of the network, it knows whether a switch connects to the internal or external network and then divides the network into the egress network and the core network.

To ensure that topology and switch functions are reasonable, we make the following definitions for two special conditions:

(1) For any given packet p, $SF(drop, p) = [drop, p]$ and $SF(world, p) = [(world, p)]$;
(2) $T(drop) = drop$ and $T(world) = world$.

In these conditions, once a packet is dropped or forwarded beyond the perimeter of the network, it must stay dropped or beyond the perimeter of the network and never return. If a packet returns from another network, we treat the return packet as a “fresh” packet.

An update message is forwarded to a switch from the controller and then directs the switch to update its rule table by installing the new rule in the message. On receipt of a packet, the switch performs the table lookup. Then the packet matches some rules according to the corresponding fields (if the packet matches no rule, it will be sent to the controller). These matched rules can be used to generate the trace of this packet. The configuration of the network is a set of rules which compose the switch function. The located packet has three parts: location (port), content (packet) and trace. A network is composed of the configuration which directs the switches to forward packets, the topology and the packets with their trace.

Network process and update. The network process is an operation of packet forwarding in the network. In a network process, the switch function takes a packet as an input and returns a list of packets which may contains zero, one or several packets. These packets are forwarded through the link and reach the adjacent switches. After the network process, the network comes into a new state. So the network process will not change the topology and configuration of the network.

The network update is an update operation of the network. The controller generates an update message and forwards it to a switch. Then the switch changes its rule table according to this message. This update operation changes the configuration of the network only.

Flow-level consistent property. In the SDN network, the switch only has some simple functions such as forwarding packets according to the rules in the flow table and the controller directs the switch how to forward packets. So the trace which contains the rules matched by the packet can be used to verify many priorities, including access control, loop-freedom, consistency, etc. The packet-level consistent property means any packet is forwarded by the same configuration (the old one or the new one, but not the mixed) during the network configuration update. The flow-level consistent property means all packets which belong to a flow are forwarded by the consistent rules from the same configuration. Once a packet is forwarded by a configuration (the new or old one), the remaining packets of the flow should be forwarded by the same configuration.

4. SARD: fast flow-level consistent update

In the openflow network forwarding devices are simple, but the controller is intelligent and powerful. So the basic idea of our mechanism is redirecting the packets affected by the configuration update to the controller and utilizing the

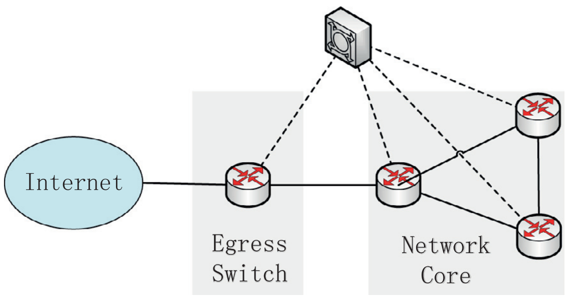

Fig. 2. Two-phase commit.

controller to accelerate the update. As the flow-level consistency is stronger than packet-level consistency, we use the two-phase commit [5] to guarantee packet-level consistency, based on which we then design the scheme of SARD to achieve the flow-level consistent update.

To prepare update, (1) we divide the network devices and ports into two parts: the egress network and the core network as Fig. 2 shows (as in [5]); (2) the controller adds the version number as a domain of the rules and the switch in the egress network tags the version number in the packets entering the network; (3) the core network uses the version number to guarantee the packet is forwarded according to the same version of configuration; (4) the egress switch strips the version number when the packet leaves the network.

As there are some unused matching fields in SDN packet, we can store the version number in one of them (VLAN id, MPLS label, etc.). In our mechanism, we use the VLAN id field. The ingress (egress) switch can modify the version number with the push (pop) operation which is well supported in the standard Openflow protocol. Therefore, no protocol change is required for our mechanism.

Definition 1 (two-phase commit [5]). Let C_1=[(s,rules)] be the old configuration with version number 1 and C_2 =[(s,rules')] be the new configuration with version number 2. Let $us = [u^i_1, \ldots, u^i_m, u^e_1, \ldots, u^e_n]$ be an update sequence:

- $C_2 = override(C_1, us)$
- $\forall p \in$ *the domain of* u^i_j, *p is in the core network*
- $\forall p \in$ *the domain of* u^e_j, *p is in the egress network*
- u^i_j, u^e_j *with version number* 2
- u^e_j *add version number* 2

Then us is a two-phase commit from C_1 to C_2.

Theorem 1. *Two-phase commit can preserve the packet-level consistent property.*

In the first phase, we update the core network. In the second phase, we update the egress network. When the packet entering the network, it is tagged with a version number which will not be changed. Using this version number to distinguish the old and new configuration, the packet is only forwarded by one configuration. If the packet has version number 1, the old configuration decides the path of the packet. If the packet has version number 2, the new configuration forwards the packet.

The problem of preserving the flow-level consistent property is equivalent to the problem of tagging the same version number to the packets of a flow. If the version number is the same, all packets will be forwarded by the same configuration.

Based on the two-phase commit, we propose the controller-partaking mechanism which uses the controller to accelerate the speed of configuration update. Considering the TCAM in switches is a scarce and expensive resource, we restrict the number of rules we can use during the update. The steps of our controller-partaking mechanism for SARD are as follows.

1. The controller installs the new configuration with a new version number in the core network. This operation does not affect the traffic forwarding because there are no packets with the new version number in the network.
2. The controller installs new configuration with a lower priority in the egress switch after the core network completes update. This operation guarantees when a packet comes in, the switch in the core network knows how to deal with it.
3. The controller installs a temporary rule with a higher priority in the egress switch and deletes the old rule. This temporary rule redirects all the affected flows to the controller when they come in the network during the flow heart break. The flow heart break means the biggest interval between two continuous packets in a flow. If there is no incoming packet of the flow during the flow heart break, we think the flow expires. During the period, the controller gets the information of existing flows and forwards these flows by the old configuration.
4. After a heart break, the controller computes *K* optimal prefixes according to the collected information of the existing flows and accordingly generates K sub rules covering all the existing flows. Then the controller installs these sub rules with a higher priority. Meanwhile the temporary rule is removed. The action of these sub rules are the same with the old rule.
5. The sub rules in the switches are timed out and removed as the existing flows expire. The old configuration disappears as time goes by and the new configuration takes place.

Definition 2 (controller-partaking update). Let $C_1 = [(s, rules)]$ be the old configuration with the version number of 1 and $C_2 = [(s, rules')]$ be the new configuration with the version number of 2. Let $us = [u^i_1, \ldots, u^i_m, u^e_1, \ldots, u^e_n, u^c_1, \ldots, u^c_h, u^o_1, \ldots, u^o_k]$ be an update sequence:

- $C_2 = override(C_1, us)$
- $\forall p \in$ *the domain of* u^i_j, *p is in the core network*
- $\forall p \in$ *the domain of* u^e_j, *p is in the egress network*
- $\forall p \in$ *the domain of* u^c_j, *p is in the egress network*
- $\forall p \in$ *the domain of* u^o_j, *p is in the egress network*
- u^i_j *with version number* 2
- u^e_j, *add version number* 2
- u^o_j, *add version number* 1
- u^c_j, *forward packets to the controller*
- $\forall actions \in u^o_j$, $actions \in C_1$

Then us is a controller-partaking update from C_1 to C_2.

Theorem 2. *Controller-partaking update in SARD can preserve flow-level consistent property.*

Before the configuration update, all packets which belong to existing flows are tagged with the version number 1 and forwarded by C_1. During the configuration update, the packets of existing flows match the sub rules of the old configuration in the egress network and are tagged with the version number of 1 and directed by C_1. The new flows beyond the covering space of the sub rules are tagged with the number of 2 and forwarded by C_2. As the pieces of the old configuration can cover all existing flows and are deleted after the contained flows disappear, the flow-level consistent property is preserved.

In the controller-partaking mechanism, all the packets affected by the update are forwarded to the controller. Then the controller can get the information of the existing flows and generate sub rules according to this information. So this mechanism accelerates the process of updating. However, as some sub rules installed in the egress switch may be still too "big", some new flows covered by these rules may come before the existing flows expire, they may never disappear. To solve these problem, we propose an incremental controller-partaking mechanism for SARD.

In the incremental controller partaking mechanism, the controller completes the configuration update in several rounds. In the next round, the controller updates the "big" sub rules which are installed in the previous rounds and still alive. Suppose K is the max number of sub rules that the switch can provide for the update. The steps of incremental controller-partaking mechanism are as follows:

1. The controller installs the new configuration with a new version number in the core network.
2. The controller installs the new rule with a lower priority in the egress switch.
3. The controller installs a temporary rule which redirects the affected flows to the controller.
4. After a flow heart break, the controller generates M number of sub rules covering all the existing flows using the minimum prefix covering algorithm (M equals K minus the number of rules in the sub rule table). The controller then installs these M sub rules in the egress switch.
5. The controller records these sub rules in its sub rule table. Some sub rules in the egress switch may expire as time goes. The controller catches the flow-removed message from the egress switch and removes the corresponding entry in the sub rule table.
6. After some flow heart breaks, if there still exist some "big" sub rules (bigger than a given threshold) in the sub rule table, the controller chooses one and starts another round of update (Jump to step 3).

The controller-partaking mechanism may install some rule whose space is huge and never expires. The incremental controller-partaking mechanism can solve this problem, where the "big" sub rules will be updated in the following periods.

Theorem 3. *Incremental controller-partaking mechanism can preserve flow-level consistent property.*

For the packets of the existing flows, no matter whether or not they are forwarded to the controller, they will be tagged with the old version number and forwarded by the old configuration. As for the new flows, some of them (covered by the sub old rules) will be tagged with the old version number and the others will be tagged with the new one. The flow-level consistent property is preserved as the core network will forward the flow packets according their tagged version number.

5. Algorithm design

In this section, we propose the algorithm of K-prefix covering, which computes the optimal K prefixes. These K prefixes can be used to divide an old rule into K optimal sub rules with the minimized space and the sub rules cover all the existing flows. We also prove the validity of our algorithm and analyze the complexity.

5.1. Definitions and theorems

In SARD, the controller gets the details of existing flows and then installs the corresponding rule for each flow. To avoid the nontrivial memory overhead in TCAM, the controller must control the number of rules. In openflow network, there are 12 match fields for an exact rule. In this section, we only use the field of source or destination IP to divide an old rule.

Definition 3 (Prefix Space)**.** For a prefix p, $Space(p)$ is the number of IP addresses covered by p. For a prefix set PS, $Space(PS)$ is the number of IP addresses covered by $\forall p \in PS$.

For example, the prefix space of 192.168.0.0/16 is 2^{16}. For any rule defined by a specific prefix, we can use the prefix space to evaluate its space size. If the prefix space is smaller, the rule contains fewer flows and it will be timed out and removed faster. But if the prefix space is larger, the rule may be difficult to disappear, as new flows come before the current flows expire.

Definition 4 (Cover relation $\preceq$)**.** Let p_1 and p_2 be any prefixes and $\preceq$ be a relation of two prefixes, $p_1 \preceq p_2$ means that all the IP addresses covered by p_1 are covered by p_2.

For the prefix sets PS_1 and PS_2, $PS_1 \preceq PS_2$ means each IP address covered by the prefixes in PS_1 is covered by some certain prefix in PS_2. In this paper, the prefix set we discuss has no cover relation among its prefixes.

Strong cover ($\prec$): $p_1 \prec p_2$ means $p_1 \preceq p_2$ and $Space(p_1) < Space(p_2)$. $PS_1 \prec PS_2$ means $PS_1 \preceq PS_2$ and $Space(PS_1) < Space(PS_2)$.

A trie (binary tree) can be used to represent all IP addresses and prefixes, as Fig. 4 shows. In the trie, for two prefixes p_2 and p_1, $p_1 \prec p_2$ means that p_2 is an ancestor node of p_1. It is straightforward that if $p_2 \bigcup p_1$ is not empty, the relation between them must be one of the three: $p_2 \prec p_1$, $p_1 \prec p_2$ or $p_2 = p_1$.

Definition 5 (Merge Operation)**.** Given two any different prefixes p_1 and p_2, $p_3 = Merge(p_1, p_2)$ means finding a prefix p_3 that: $p_1 \preceq p_3$, $p_2 \preceq p_3$ and $Space(p_3)$ is minimized.

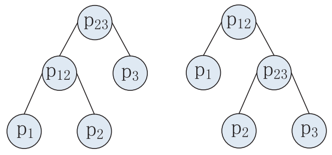

Fig. 3. A binary tree for three ordered prefixes.

In the complete binary tree, the merge operation means finding the lowest common ancestor of two prefixes. If $p_1 \prec p_2$ or $p_2 \prec p_1$, the merge operation returns p_2 or p_1 correspondingly.

Given three ordered[1] prefixes p_1, p_2 and p_3, suppose there are no cover relation among them. Let $p_{12} = Merge(p_1, p_2)$ and $p_{23} = Merge(p_2, p_3)$. According to Definition 4, p_{12} and p_{23} both cover p_2. We can further get the conclusion that $p_{12} \prec p_{23}$ or $p_{23} \prec p_{12}$. That is to say, we can use these five prefixes to construct a binary tree as shown in Fig. 3 shows. This is a significant conclusion for our algorithm.

Definition 6 (Min K-Prefix Set $Min_k(PS)$)**.** Given a prefix set PS and $k \le |PS|$, $PS' = Min_k(PS)$ is the min k-prefix set for PS iff $PS \preceq PS'$, $|PS'| = k$ and $Space(PS')$ is minimized.

From Definition 5, if two prefixes do not cover any common IP address, they can be merged and we can get a prefix covering both of them. For a set which covers N IPs, we sort the IPs and merge the adjacent IPs to get the lowest common ancestor. The N IPs and $N-1$ prefixes can be constructed as a binary tree.

According to Definition 3, the rules defined by the prefixes with smaller space are more possible to disappear. From the constructed binary tree, we need to choose K prefixes with the minimal space and covering all the IPs (leaf nodes).

Let $Space_k(PS) = Space(Min_k(PS))$.

Theorem 4. *Given any prefix set PS,* $Space_k(PS) \ge Space_{k+1}(PS)$.

From Definition 6, $Space_k(ps)$ is the minimal space and all K prefixes have no common elements. We can choose one from K prefixes and split it into two prefixes. Then the *Space* of this new prefixes set is not larger than $Space_k(ps)$ and not smaller than $Space_{k+1}(ps)$.

5.2. Minimum prefix covering algorithm

All flows affected by the update are forwarded to the controller, and then the controller gets the details (like the source or destination IPs) of these flows. On one hand, to avoid the explosion of the flow table in the switch, we limit the number of rules used during the configuration update. On the other hand, to complete the update fast, we use the rules as many as we can. So even $Space_{k-1}(ps)$ equals $Space_k(ps)$, the controller still uses K prefixes to cover the IP addresses and divide the old rule.

Our algorithm uses dynamic programming and derives the optimal prefixes from bottom to top: (1) sort N IPs and get new $N-1$ prefixes by merging the adjacent prefixes; (2) construct a binary tree using N IPs and $N-1$ merged prefixes; (3) find K nodes from the tree with the minimized space. Table 1 defines the set of variables and constants which appear in this K-prefix covering algorithm.

[1] The prefixes are sorted by the inorder traversal in the trie.

Table 1
Variables and constants.

IPs	The source IPs of flows
Height	32—the mask of prefix
K	The number of prefixes
item = <*s*, *l*, *r*>	The left child has *l* IP addresses
	The right one has *r* IP addresses
	The minimal total space is *s*
$IS = [item_i]$	An item set

In the step 2, we use *mergeNode* to compute *IS* in every node as the Eq. (1) shows. In the step 3, we get an allocation scheme *P.IS* (*dlist.start.value.IS*) and then get K prefixes from up to bottom. If the scheme allocates m prefixes to a node p, $p.IS[m]$ is the allocation of its children. Additionally, l in $p.IS[m]$ is the number of prefixes allocated to the left child, r is the number of prefixes allocated to the right child and s is the space of this allocation scheme. If l and r in $p.IS[m]$ are 0, it means p is selected and not divided. At the end, we find the best allocation scheme.

MergeNode merges two prefixes into one and computes *IS* of the new node. As Eq. (1) shows, i is the number of prefixes allocated in this node and $IS[i]$ means the allocation scheme with minimal space among all possible schemes. $IS[1]$ means the number of prefixes allocated is 1 and there is no dividing. So $IS[1]$ equals 2^{height}.

$$IS[i] = \begin{cases} < min(IS1[j].s + IS2[m].s),\ j, m >, \\ i = j + m, K \ge i > 1 \\ < 2^{height}, 0, 0 >,\ \ i = 1 \end{cases} \tag{1}$$

For example, given five IP addresses (59.78.45.192, 59.78.45.195, 59.78.45.199, 59.78.45.203, 59.78.45.207) and get four prefixes (59.78.45.192/30, 59.78.45.192/29, 59.78.45.192/28, 59.78.45.200/29). We can construct a binary tree as Fig. 4 shows and initialize the leaf nodes a, b, c, d, e: $height$=0, $IS = [< 1, 0, 0 >]$. *IS* shows the space of these nodes is one and they have no children. Through merging node a and b, we can get the value of node T: $height$=2, $IS = [< 4, 0, 0 >, < 2, 1, 1 >]$. The *IS* of node T means if allocating 1 prefix, the prefix is T; if allocating 2 prefixes, they are a and b. When we compute $IS[3]$ of R, we can allocate the left child S two prefixes and the right child W one prefixes or opposite. Then we get the allocation with minimal space. After building the binary tree, we get the K prefixes from up

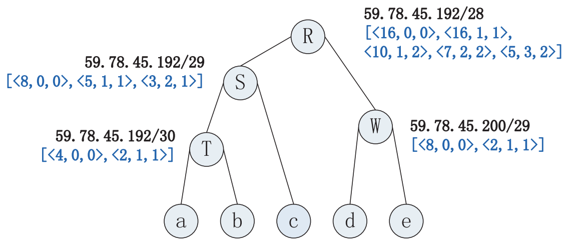

Fig. 4. An example for the algorithm.

to bottom. Assume K is 3, we get the allocation from $IS[3]$ in the root R. Then we find it allocates one prefix to its left child and two prefixes to its right child. Finally we will get S, d and e.

5.3. Algorithm validity and complexity

In our algorithm, we construct a binary tree using $2N-1$ prefixes and get K prefixes from the tree to compose the final allocation scheme. Let $PS = [p_1, p_2, \ldots, p_N, \ldots, p_{2N-1}]$ be the list of prefixes and r be the index of the root in the list PS. Assume that *flag* is an index of a prefix in the tree, $Left(p_{flag})$ is the index of the left child, $Right(p_{flag})$ is the index of the right child and the final list of K prefixes is $PK = [p'_1, \ldots, p'_k]$ which is gained by our k-prefix covering algorithm. Then we prove $PK = Min_k(PS)$.

First, we prove $Min_k(PS)$ must be a subset of PS: Assume that the prefix p_{2N} is not in PS but in $Min_k(PS)$.

(1) **Compare p_{2N} and p_r**: If $p_r \prec p_{2N}$, then $Space(p_r) < Space(p_{2N})$. So p_{2N} will never be selected and the process ends; If $p_{2N} \prec p_r$, then $flag = r$, $left = Left(p_{flag})$, $right = Right(p_{flag})$ and jump to step 2.

(2) **Compare** p_{2N} and p_{child}**:** If $p_{2N} \prec p_{child}$, $child = left|right$, then $flag = child, left = Left(p_{child}), right = Right(p_{child})$, and jump to step 2; If $p_{child} \prec p_{2N}$ and $p_{2N} \bigcap p_{child'} = \phi$, $(child, child') = (left, right)$, then $Space(child) < Space(p_{2N})$. So p_{2N} will never be selected and the process ends.

Therefore, the prefix p_{2N} does not exist and $Space_k(PS) \subseteq PS$. Given p_i, $p_i \bigcap p_{left} \neq \phi$ and $p_i \bigcap p_{right} \neq \phi$, we can get that $Space(p_{flag}) \leq Space(p_i)$. Thus, there does not exist p_{2N} that $p_{any} \bigcap p_{left} \neq \phi$ and $p_{any} \bigcap p_{right} \neq \phi$.

Now we prove there does not exist a better allocation. Given the prefix list $PT = [p^t_1, \ldots, p^t_m]$ with m prefixes.

(1) $p^t_i \in PT$, $p^t_j \in PT$ and $p^t_k = Merge(p^t_i,\ p^t_j)$.

(2) Compute the list IS of p^t_k according to Eq. (1). Assume $IS^t_k[(a+b)] = IS^t_i[a] + IS^t_j[b]$. If there exist smaller $IS^t_i[a]$, $IS^t_k[(a+b)]$ will be smaller. Thus in our algorithm, the optimal solution covers the optimal solution of sub-problem.

(3) The final list PK is the list which has K elements, covers all the N *ips* and has the minimal space.

Therefore, there does not exist a better allocation. $Min_k(PS) \subseteq PS$ and our algorithm can get the best allocation, so $PK = Min_k(PS)$.

The algorithm has three steps: (1) sort the prefixes and get new $N-1$ prefixes through merging the adjacent prefixes; (2) construct a binary tree using $2N-1$ prefixes; (3) find K nodes from the tree. In the first step, the time complexity of sorting is $O(NlogN)$ and getting new prefixes is $O(N)$. In the second step constructing the tree just needs to iterate all the prefixes, so the time complexity is also $O(N)$. In the third step we need to build IS in every node and the complexity is $O(NK^2)$. So the time complexity of our algorithm is $O(NlogN + NK^2)$.

6. SDN with multiple controllers

In SDN, one single centralized controller with limited computation capacity and bandwidth will become the bottleneck of the network. Besides, the controller may crash due to diverse vicious attacks. To solve these problems, some logically centralized but physically distributed SDN architectures have been proposed [17,18]. Hyperflow [19] unifies the global view through a subscription system. Onix [20] uses a database to synchronize the strong consistent information like the global view and employs the distributed hash table to share the weak consistent information like link utilization. Kandoo [21] proposes a multi-layer SDN architecture to overcome the problem. In these schemes, the control plane contains multiple controllers. As Fig. 5 shows, each switch connects to one master controller. A controller controls the switches in its domain directly and affects the switches beyond its domain by the communication with other controllers. When the topology changes, the master controller receives the event and then notifies the other controllers.

To guarantee the flow level consistent update for the multi-controller scenario, SARD needs the cooperation among these controllers. (1) When the network configuration update occurs, the main controller (the controller which triggers the configuration update) divides the network into the egress network and the core network. (2) The main controller installs the new configuration into the core network. It installs the rules into the switches in its domain directly. If the switch is out of its domain, it sends the rules to the master controller of the switch. The corresponding master controller programs the switches in its domain and notifies the main controller after the rules have been installed. (3) Then the main controller notifies the egress controller (the master controller of the egress switches) to update the egress network. (4) In the following steps, we only need to program the egress switch to find the existing flows and install some rules for them. The egress controller can handle all the remaining work as if it is the only one single controller of the network. It installs some rules in the egress switches to collect the information of the packets affected by the update. At the same time, it installs the new configuration with a lower priority in these egress switches. Then it generates the K optima sub rules of the old configuration. The new rule configuration are immediately triggered for all the flows except those covered by the K sub rules. Finally, these sub rules will disappear as the existing flows expire.

Compared with the single-controller case, SARD will not hinder the performance or increase update time in

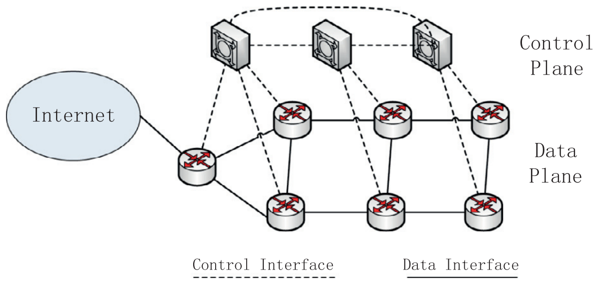

Fig. 5. Multi-controller SDN.

multi-controller SDN. That is because when the controller installs the new configuration in the core network, it distributes the rules to all the other controllers that will program the corresponding switches. If the egress switches belong to different master controllers, these controllers may commit the update concurrently. We only need to guarantee the version number is unique in the whole network. There are some consensus algorithms [22] which can solve this problem.

7. Evaluation results

We evaluate the performance of SARD and the algorithm with the popular POX controller and Mininet. We develop a load balance application that finds server replicas and then divides traffic between them. The topology is illustrated in Fig. 2. The egress switch broadcasts an external IP address using ARP protocol. The destination (the external IP) of an in packet will be replaced with the IP address of a certain server replica by the egress switch. When receiving an out packet with the source IP of a server replica, the egress switch replaces the source IP with the external IP. In short, the egress switch integrates the functions of NAT and load-balancer.

The traffic confirms to Poisson distribution [23]. The inter-arrival times are exponentially distributed and the durations are power law distribution [24]. All IPs are evenly distributed across the whole IP space. To simulate the network change and the configuration update, one server is provided at the first and another wakes up after a while. At beginning, the egress switch forwards all packets to the single server. After another server wakes up, the controller detects the network change and notifies the load-balance program. Then the program divides the space into pieces and shifts traffic to the new server. In our experiment, we implement the method that divides the space and installs the pieces periodically as a comparison. To avoid the expansion of the flow table, the number of rules we use during the update is limited. We collect the data every 30 s.

Algorithm 1 *K*-PrefixCover(IPs).

```
sort IPs
# build a double linked list using IPs
initialize dlist
for all ip in IPs do
    initialize a node for ip
    node.value=$Tree_{ip}$ with only the root
    $Tree_{ip}.height = 0$
    $Tree_{ip}.prefix = IP$
    $Tree_{ip}.IS = <1, 0, 0>$
    dlist.add(node)
end for
for node in dlist do
    node.height=32-Mask(node,node.next)
end for
#the bits of IP is 32,the height is less than 33
dlist.end.height=33
#construct a binary tree
flagNode=dlist.start
while flagNode.next!=dlist.end do
If flagNode.height < flagNode.next.height
    mergeNode(flagNode,flagNode.next)
    dlist.delete(flagNode.next)
    flagNode=flagNode.before
Else
    flagNode=flagNode.next
EndIf
end while
IPs merge into one prefix
dlist.start.value is the allocation scheme
```

Fig. 6 shows how the factors (length of prefix, number of rules, number of flows) affect the speed of configuration update. In Fig. 6(a), the number of rules we use during the configuration update is 128 and the number of flows per second is 120. It shows when the length of the prefix

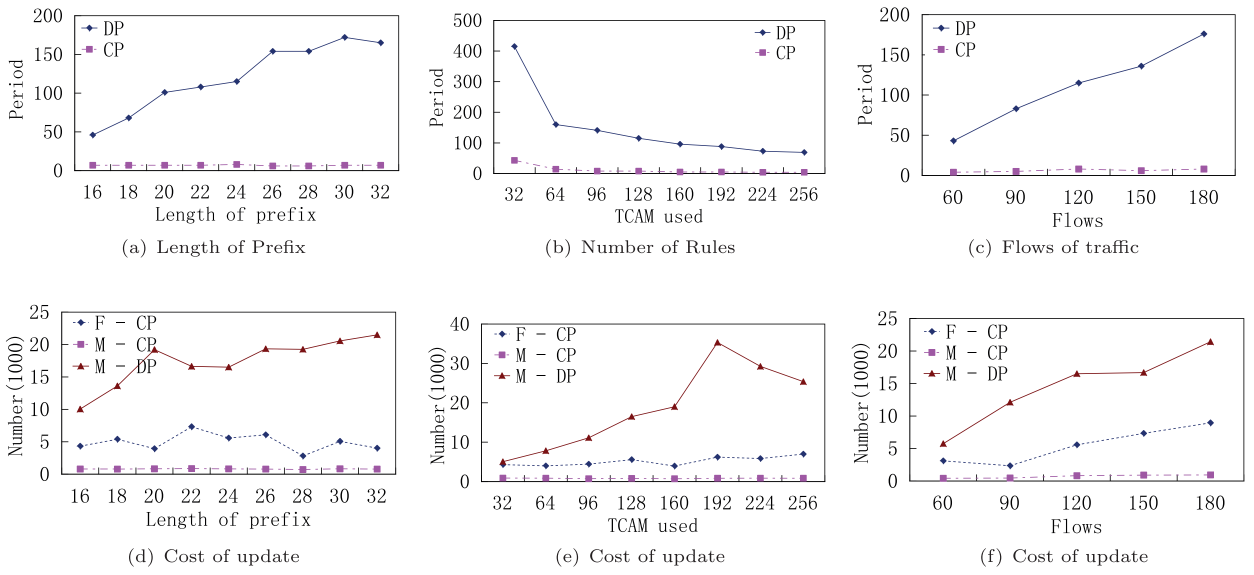

Fig. 6. DP is an abbreviation of divide periodically and CP is short of controller partaking. F means the flows through the controller and M means the control message between the switch and controller. So M–CP is the control message in the scheme of controller partaking during the configuration update.

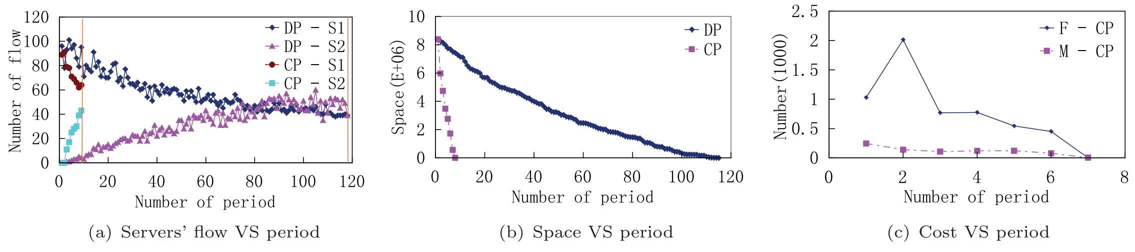

Fig. 7. Load balance.

increases, the update period also increases for the method of periodical division (DP). While SARD (Controller Partaking, CP) is quite stable with the prefix length. In the method of dividing periodically, the prefixes will be divided and installed in the egress switch. So if the prefix is longer, it will be split more times. But in SARD, we generate sub rules according to the IP addresses of the existing flows. The space sizes of the sub rules are mainly determined by the flow number. Thus SARD is almost not affected by size of the original old rule (prefix length). Correspondingly, the number of flows forwarded to the controller in SARD and the control messages in these two schemes are shown in Fig. 6(d). The control messages in dividing periodically increase as the length of prefix increases, but not in SARD.

In Fig. 6(b), the prefix length is 24 and the number of flows per second is 120. In the method of SARD and periodical division, the more rules (larger TCAM) in the egress switch, the fewer periods are needed. As larger TCAMs mean the prefixes can be divided into smaller pieces. Thus the sub rules will take less time to disappear (time out). Fig. 6(e) shows the number of flows forwarded to the controller in SARD and the number of control messages.

Fig. 6(c) presents the relation of the traffic and update period (the prefix length is 24 and the rule number is 128). More flows the traffic contains, more periods the method of periodical division need. As more flows means before the rule piece is deleted, new flows are more likely to arrive. Thus there will be fewer usable rules in the next period. In contrast, although SARD is also influenced by these factors, it is more steady and takes fewer periods. Fig. 6(f) is the number of flows forwarded by the controller and the number of control messages during the configuration update in SARD. We can see, when the number of flows increases, there are more flows forwarded to the controller. That is because: although the periods increase as the flows increase, the flows matching the sub rules increase too. As Fig. 6 shows, our scheme is better than periodical division.

In Fig. 7, the prefix length and the rule number are respectively set to be 24 and 128. The number of flows per second is 120. We present the change of servers' load, space and number of flows through the controller.

Fig. 7(a) shows the number of flows through the server at each period. The load balance program uses periodical division and controller partaking to balance traffic between servers. The method of periodical division takes 50 min to complete update. Controller partaking of SARD takes only 4 min. Periodical division takes a longer time because it divides the space of prefix and installs all the small pieces in switches periodically regardless of the existing flows. After the configuration update, the flows to server 1 are more than server 2. As there are still some existing flows forwarded according to the old configuration. During update, we shift some traffic from the busy sever to another, that means moving space between servers. As Fig. 7(b) shows, the controller partaking method in SARD is much faster than periodical division.

However, there are also some problems in SARD. In SARD, the flows affected by the configuration update must be forwarded to the controller before arriving at the final server. Thus controller partaking of SARD consumes some extra memory, CPU and bandwidth of the controller. Fig. 7(c) shows the number of flows through the controller in every period. In the first period, the number of flows is about 1000 and in the second period, it is about 2000. But in the remaining periods, the number becomes smaller and smaller. Because as the configuration update proceeds, SARD gets the network state and evaluate the volume of traffic. It only chooses some appropriate pieces to update according to these information which is collected in the previous periods.

We also evaluate our algorithm on other real-world topologies [9], such as Chinanet, IBM and York. The topology properties, including the number of switches, links and hosts, are shown in Table 2. We obtain the BGP table from [25] and then generate an IP set accordingly. As the length of these prefixes is not constant, we fix these prefixes with the length 24 and set the left bits as zero. The results show SARD is faster than the method of periodical division, which is consistent with the above results. SARD takes less than 20 periods to complete the update and shift the traffic, but the method of periodical division needs more than 300 periods. During the update, SARD adds/deletes about 1300/900 rules

Table 2
The real topologies.

Algorithm	Topology			Controller partaking				Divide periodically			
	Switches	Links	Hosts	Periods	Add	Delete	Flows	Periods	Add	Delete	Flows
Chinanet	42	66	16	17	1357	816	29,067	266	32,748	2852	0
Ibm	18	24	16	18	1365	872	27,933	296	37,448	3303	0
York	23	24	16	16	1297	800	30,010	263	33,662	2781	0

in the egress switch, while periodical division adds/deletes about 32,000 and 3000 rules. That is to say, in the method of periodical division, there are many rules that are timed out and deleted automatically. Most of these added rules do not play any role in packet forwarding. The controller in our SARD needs to forward about 30000 flows during the update. In SDN, there may be several hundreds of switches but only several controllers. Using several powerful controllers to strengthen the network ability is acceptable.

8. Related works

SDN is proposed to facilitate the evolution of network and enable simple management. As it is developed intensively and widely, SDN demonstrates the formidable vitality and the superiority in some aspects, including meeting the demand of network virtualization and cloud computing [26,27], combining with Information-Centric Networking [28,29], composing hybrid network with traditional network [7,30,31], etc.

There are various protocols and mechanisms of avoiding transient undesired states when the planned update happens. Most of them focus on concrete protocol and only a few are about SDN configuration update. In the paper [32], the authors progressively adjust the metric associated with this link to minimize disruption and ensure transient consistency during the convergence of link-state interior gateway protocols like OSPF. These methods only preserve the basic properties like no-loop.

In SDN, the consistent update has been explored in [15]. 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. Reitblatt et al. [5] provide a network model and the two-phase commit for per-packet consistency. When packets arriving 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. In the worst case, this brings a 100% consumption of TCAM. Frenetic [33] uses this scheme to guarantee consistency.

To reduce the cost of TCAM, an incremental update algorithm [6], which splits the update into several rounds, trade update time for rule-space overhead. In each round: (1) choose a subset of flows which are moved from the old configuration to the new one; (2) find new rules to be installed according to the flows; (3) install rules using the two-phase commit mechanism; (4) find old rules that are not used by any flows and delete them. Multi-commit transactions [34] are proposed to solve the problem which stems from the inconsistent packet processing during network change. This mechanism improves the classic techniques of the database and provides the essential serialization and isolation properties. Vissicchio et al. [35] develop a method to complete anomaly-free update in hybrid networks. The trade-off between the strength of consistent property and dependencies among rules is argued in [36]. The controller generates an update DAG according to the consistent property and topology, which is complex and needs a number of computation. All these methods concentrate on per-packet consistency.

The basic idea of combining version number with rules timeout is proposed in [15]. Based on the paper, Reitblatt et al. [5] propose three mechanisms to achieve the flow-level consistency: (1) Switch rules with timeouts. This method divides the rule space into pieces and installs the pieces in switches dynamically over time. (2) Wildcard cloning. When a packet coming into the network, the switch will generate a small new rule to match the domain exactly. During the update, existing flows are handled by the cloning rules. (3) End-host feedback. The end host or server reports information of existing flows to the controller and the controller use this information to determine which flow is still alive. As openflow switch is very simple and has little intelligence, only the switch rules with timeouts mechanism can be achieved. But this mechanism takes a long time to complete the update. A safe protocol [4] is suggested that the packets affected by the network change should be sent to the controller, and then be re-leased to the network after the change terminates. But this mechanism will bring many burdens to the controller and the control plane bandwidth. During the configuration update, it will also cause the disruption of transmission and bring some delay to the packet transmission.

Our mechanism is developed from the model proposed in [5,16]. The model in [5,16] only focuses on a static configuration and is used to help describe their mechanism. The network formalism has been explored in the verification of network functions [37]. We extended the network semantics to include switch, updates and consistent priorities, so we could prove flow-level consistent properties of our mechanism.

9. Conclusion

In this paper, we explore a new mechanism (SARD) that makes the flow-level consistent configuration update in SDN fast and efficient, by limiting the number of rules used during the update and redirecting the packets affected to the controller. We have presented a network update model which captures the essential features of the SDN network and controller-partaking method which preserves the flow-level consistency. To avoid the explosion of the flow table, we propose the *K*-prefix covering algorithm to computes K optimal prefixes which cover all the existing flows with the minimized space. We also do some optimization of shifting a part of traffic from the old policy to the new one in each round, only using controller-partaking approach to update the overlap between the old and the new rules. In addition, we also discuss how to realize our mechanism in multi-controller SDN. The results of our simulation show that our work makes a significant influence on reducing the configuration update time.

The incremental controller-partaking mechanism is powerful and the network operators (programmers) do not need to verify the properties during the transition between the old configuration and the new one, as the property which is preserved in both configurations holds for the flow during the configuration update. The operators do not need to consider the interim complex and inconstant network states during the configuration update. They can use our incremental controller-partaking mechanism to design arbitrary update. They only need to guarantee the initial and final configurations are correct.

Acknowledgments

This work is supported by the National Research Program of China (973) with no. 2012CB315803, the National High-tech R&D Program of China (863) with no. 2014ZX03002004, the National Science Foundation of China with no. 20149110081 and Shenzhen Key Laboratory of Software Defined Network with no. ZDSYS20140509172959989.

References

[1] V. Gill, D. McPherson, A. Retana, D. Walton, Border Gateway Protocol (BGP) Persistent Route Oscillation Condition, 2001, Internet draft.
[2] T.G. Griffin, G. Wilfong, Analysis of the MED oscillation problem in BGP, in: Proceedings of the 10th IEEE International Conference on Network Protocols (ICNP), Paris, France, 2002.
[3] N. McKeown, Keynote talk: software-defined networking, in: Proceedings of INFOCOM the 28th IEEE International Conference on Computer Communications, Rio de Janeiro, Brazil, 2009.
[4] R. McGeer, A safe, efficient update protocol for OpenFlow networks, in: Proceedings of the First ACM SIGCOMM Workshop on Hot Topics in Software Defined Networks, Helsinki, Finland, 2012.
[5] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, D. Walker, Abstractions for network update, in: Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), Helsinki, Finland, 2012.
[6] N.P. Katta, J. Rexford, D. Walker, Incremental consistent updates, in: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networks, Hong Kong, 2013.
[7] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, R. Wattenhofer, Achieving high utilization with software-driven WAN, in: Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), Hong Kong, 2013.
[8] B. Lantz, B. Heller, N. McKeown, A network in a laptop: rapid prototyping for software-defined networks, in: Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks, Monterey, USA, 2010.
[9] The Internet Topology Zoo, http://topology-zoo.org/.
[10] D. Oppenheimer, A. Ganapathi, D.A. Patterson, Why do internet services fail, and what can be done about it? in: Proceedings of USENIX Symposium on Internet Technologies and Systems (USITS), Seattle, USA, 2003.
[11] The openflow switch specification, http://OpenFlowSwitch.org.
[12] M. Casado, M.J. Freedman, J. Pettit, J. Luo, N. McKeown, S. Shenker, Ethane: taking control of the enterprise, ACM SIGCOMM Comput. Commun. Rev. 37 (4) (2007) 1–12.
[13] M. Casado, T. Garfinkel, A. Akella, M.J. Freedman, D. Boneh, N. McKeown, S. Shenker, SANE: a protection architecture for enterprise networks, in: Proceedings of the 15th USENIX Conference on Security Symposium, Vancouver, Canada, 2006.
[14] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, S. Shenker, NOX: towards an operating system for networks, ACM SIGCOMM Comput. Commun. Rev. 38 (3) (2008) 105–110.
[15] M. Reitblatt, N. Foster, J. Rexford, D. Walker, Consistent updates for software-defined networks: change you can believe in!, in: Proceedings of the 10th ACM SIGCOMM Workshop on Hot Topics in Networks, 2011.
[16] P. Kazemian, G. Varghese, N. McKeown, Header space analysis: static checking for networks, in: Proceedings of the Ninth USENIX Conference on Networked Systems Design and Implementation, San Jose, USA, 2012.
[17] D. Levin, A. Wundsam, B. Heller, N. Handigol, A. Feldmann, Logically centralized?: state distribution trade-offs in software defined networks, in: Proceedings of the First ACM SIGCOMM Workshop on Hot Topics in Software Defined Networks, Helsinki, Finland, 2012.
[18] A. Dixit, F. Hao, S. Mukherjee, T.V. Lakshman, R. Kompella, Towards an elastic distributed SDN controller, in: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networks, Hong Kong, 2013.
[19] A. Tootoonchian, Y. Ganjali, HyperFlow: a distributed control plane for openflow, in: Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking, San Jose, USA, 2010.
[20] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, Onix: a distributed control platform for large-scale production networks, in: Proceeding of the Ninth USENIX Symposium on Operating Systems Design and Implementation (OSDI), Vancouver, Canada, 2010.
[21] S. Hassas Yeganeh, Y. Ganjali, Kandoo: a framework for efficient and scalable offloading of control applications, in: Proceedings of the First ACM SIGCOMM Workshop on Hot Topics in Software Defined Networks, Helsinki, Finland, 2012.
[22] L. Lamport, Paxos made simple, ACM Sigact News 32 (4) (2001) 18–25.
[23] C. Williamson, Internet traffic measurement, IEEE Internet Comput. 5 (6) (2001) 70–74.
[24] Z. Xiao, L. Guo, J. Tracey, Understanding instant messaging traffic characteristics, in: Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria, 2007.
[25] Cidr Report, http://www.cidr-report.org/.
[26] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, G. Parulkar, Flowvisor: A Network Virtualization Layer, Technical Report, Deutsche Telekom Inc. R&D Lab, Stanford University, Nicira Networks, 2009.
[27] Z. Liu, Y. Li, L. Su, D. Jin, L. Zeng, M2cloud: software defined multi-site data center network control framework for multi-tenant, in: Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), Hong Kong, 2013.
[28] D. Syrivelis, G. Parisis, D. Trossen, P. Flegkas, V. Sourlas, T. Korakis, L. Tassiulas, Pursuing a software defined information-centric network, in: Proceedings of European Workshop on Software Defined Networking (EWSDN), Darmstadt, Germany, 2012.
[29] L. Veltri, G. Morabito, S. Salsano, N. Blefari-Melazzi, A. Detti, Supporting information-centric functionality in software defined networks, in: Proceedings of IEEE International Conference on Communications (ICC), Ottawa, Canada, 2012.
[30] P. Lin, J. Hart, U. Krishnaswamy, T. Murakami, M. Kobayashi, A. Al-Shabibi, K.-C. Wang, J. Bi, Seamless interworking of SDN and IP, in: Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), Hong Kong, 2013.
[31] B. Rais, M. Mendonca, T. Turletti, K. Obraczka, Towards truly heterogeneous internets: bridging infrastructure-based and infrastructure-less networks, in: Proceedings of the Third International Conference on Communication Systems and Networks (COMSNETS), Bangalore, India, 2011.
[32] P. Francois, M. Shand, O. Bonaventure, Disruption free topology reconfiguration in OSPF networks, in: Proceedings of the 26th IEEE International Conference on Computer Communications (INFOCOM), Anchorage, USA, 2007.
[33] N. Foster, A. Guha, M. Reitblatt, A. Story, M.J. Freedman, N.P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, others, Languages for software-defined networks, IEEE Commun. Mag. 51 (2) (2013) 128–134.
[34] P. Peresini, M. Kuzniar, N. Vasic, M. Canini, D. Kostic, OF. CPP: consistent packet processing for openflow, in: Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networks, Hong Kong, 2013.
[35] S. Vissicchio, L. Vanbever, L. Cittadini, G. Xie, O. Bonaventure, Safe updates of hybrid SDN networks, Technical Report, UCL, 2013.
[36] R. Mahajan, R. Wattenhofer, On consistent updates in software defined networks, in: Proceedings of the 12th ACM SIGCOMM Workshop on Hot Topics in Networks, College Park, USA, 2013.
[37] R. McGeer, Verification of switching network properties using satisfiability, in: Proceedings of IEEE International Conference on Communications (ICC), Ottawa, Canada, 2012.

Qing Li received the B.S. degree (2008) from Dalian University of Technology, Dalian, China, the Ph.D. degree (2013) from Tsinghua University, Beijing, China; all in computer science and technology. He is currently an assistant researcher in the Graduate School at Shenzhen, Tsinghua University. His research interests include reliable and scalable routing of the Internet, Software Defined Networks and Information Centric Networks.

Kun Zhao received his B.S. degree (2012) from Northwestern Polytechnical University, Xian, P.R. China, the M.S. degree from Graduate School at Shenzhen, Tsinghua University, Shenzhen, China; both in computer science and technology. He is now an IT engineer in Hangzhou, China. His research interests include Future Internet and Software Defined Networking.

Yong Jiang received his M.S. and Ph.D. degrees in computer science from Tsinghua University, Beijing, P.R. China, in 1998 and 2002, respectively. Since 2002, he has been with the Graduate School of Shenzhen of Tsinghua University, Guangdong, China, where he is currently a professor. His research interests include Internet architecture and its protocols, IP routing technology, etc.

Mingwei Xu received the B.S. degree (1994) and Ph.D. degree (1998) both from Department of Computer Science and Technology, Tsinghua University. Now he is a professor in Tsinghua University. His research interests include future network architectures, Internet Routing and high-speed router architectures.

Shutao Xia received the B.S. degree in mathematics and the Ph.D. degree in applied mathematics from Nankai University, Tianjin, China, in 1992 and 1997, respectively. Since January 2004, he has been with the Graduate School of Shenzhen of Tsinghua University, Guangdong, China, where he is currently a professor. His current research interests include coding theory, information theory, and networking.