Frontiers of Information Technology & Electronic Engineering
www.zju.edu.cn/jzus; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

A secure and high-performance multi-controller architecture for software-defined networking*

Huan-zhao WANG[†1,2], Peng ZHANG[†‡1,3], Lei XIONG[1], Xin LIU[1], Cheng-chen HU[†1,3]

([1]*Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China*)

([2]*Science and Technology on Information Transmission and Dissemination in Communication Networks Laboratory, Shijiazhuang 050081, China*)

([3]*MOE Key Laboratory for Intelligent Networks and Network Security, Xi'an Jiaotong University, Xi'an 710049, China*)

[†]E-mail: hzhwang@xjtu.edu.cn; p-zhang@xjtu.edu.cn; huc@ieee.org

Received Oct. 7, 2015; Revision accepted Jan. 26, 2016; Crosschecked June 8, 2016

Abstract: Controllers play a critical role in software-defined networking (SDN). However, existing single-controller SDN architectures are vulnerable to single-point failures, where a controller's capacity can be saturated by flooded flow requests. In addition, due to the complicated interactions between applications and controllers, the flow setup latency is relatively large. To address the above security and performance issues of current SDN controllers, we propose distributed rule store (DRS), a new multi-controller architecture for SDNs. In DRS, the controller caches the flow rules calculated by applications, and distributes these rules to multiple controller instances. Each controller instance holds only a subset of all rules, and periodically checks the consistency of flow rules with each other. Requests from switches are distributed among multiple controllers, in order to mitigate controller capacity saturation attack. At the same time, when rules at one controller are maliciously modified, they can be detected and recovered in time. We implement DRS based on Floodlight and evaluate it with extensive emulation. The results show that DRS can effectively maintain a consistently distributed rule store, and at the same time can achieve a shorter flow setup time and a higher processing throughput, compared with ONOS and Floodlight.

Key words: Software-defined networking (SDN), Security, Multi-controller, Distributed rule store
http://dx.doi.org/10.1631/FITEE.1500321 **CLC number:** TP393

1 Introduction

Software-defined networking (SDN) promises a centralized, flexible, and programmable control of computer networks. In a typical SDN, a controller compiles network policies into forwarding rules, and installs them at switches through a standard channel, e.g., OpenFlow (McKeown *et al.*, 2008). Switches simply enforce these rules, thereby realizing the intended network policies.

As a key component in SDN, the controller functions as a bridge between network applications and switches. However, current single-controller architectures suffer from the following drawbacks: (1) The controller can become a single-point failure under attack: an adversary can flood flow requests towards the controller, in order to saturate its processing capacity (Shin *et al.*, 2013); (2) The response time to flow requests is relatively long, due to the complicated interaction of applications through the northbound application programming interface

‡ Corresponding author

* Project supported by the National Natural Science Foundation of China (Nos. 61402357, 61272459, and 61402357), the China Postdoctoral Science Foundation (No. 2015M570835), the Fundamental Research Funds for the Central Universities, China, the Program for New Century Excellent Talents in University, and the CETC 54 Project (No. ITD-U14001/KX142600008)

ORCID: Peng ZHANG, http://orcid.org/0000-0001-7721-2675

(API). This can result in a large flow setup latency. Even though there are some multi-controller solutions (Tootoonchian and Ganjali, 2010; Yeganeh and Ganjali, 2012; Berde *et al.*, 2014), we find that they are still rooted in the traditional single-controller design philosophy, where applications and controllers are closely coupled. As a result, they still have a relatively large flow setup latency and low processing throughput.

To address the above issues of current SDN controllers, we propose the distributed rule store (DRS), a new multi-controller architecture that takes both security and performance into consideration. In DRS, flow rules calculated by applications are cached across multiple controllers, and each controller holds only a subset of all rules. The controllers then periodically check the consistency of their rules with each other, so that when rules at one controller are maliciously modified, they can be detected and repaired. When a flow request comes, the controller simply looks up in its cache for the corresponding rules, without the interaction of applications. Thus, the response time to flow requests can be significantly reduced.

When realizing DRS, we are faced with the following challenges: (1) How to partition flow rules among multiple controllers, such that each controller has roughly the same storage load? (2) How to check the consistency of multiple replicas of a single flow rule, such that faulty rules can be detected and recovered in time? (3) How to update flow rules efficiently, such that a single network event causes only a small number of rules to be updated? (4) How to assign multiple controllers to switches, such that each controller has roughly the same processing load?

We will show how we address the above challenges in the rest of this paper. In sum, our contribution is three-fold:

1. We propose DRS, a new multi-controller architecture, which can mitigate the controller capacity saturation threats and malicious rule modifications in SDN, and at the same time improve the performance of the SDN control plane.

2. We implement a distributed controller based on DRS, and design a controller assignment algorithm to distribute processing load evenly across multiple controllers.

3. We conduct extensive emulation to demonstrate that the DRS-based distributed controller can achieve a smaller flow setup latency, higher processing throughput, and better load balance than Floodlight and ONOS.

2 Motivation

In this section, we first validate the controller capacity saturation attack with a simple experiment. Then we show the limitation of flow rule installation mechanisms used in current SDN controllers.

2.1 Controller capacity saturation

In this subsection, we carry out an experiment to validate the possibility of controller capacity saturation, which was first studied by Shin *et al.* (2013). We use Mininet (Lantz *et al.*, 2010) to emulate a FatTree ($k = 4$), controlled by a single Floodlight controller (Floodlight Project, 2016). Mininet is a tool that uses process-based virtualization to emulate hosts and switches in Linux's network space. The emulated network and the controller reside on two different Linux servers.

Let several hosts flood User Datagram Protocol (UDP) packets to some other hosts using Trinoo (Dittrich, 1999), such that the top of rack (ToR) switches connected with these hosts will generate a large number of flow requests to the controller. At the same time, let one host periodically ping another host to check the availability of the Floodlight controller. To avoid letting the flooded UDP traffic interfere with the ping packets, we carefully arrange the paths of these UDP packets so that they do not share any common switch with those of the ping packets. Fig. 1 reports the results with different

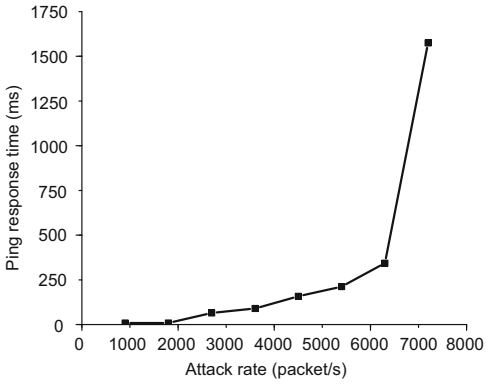

Fig. 1 Ping time vs. the tack rate

UDP sending rates. We can see that the ping time increases sharply when there are more than 6000 requests per second. This means that the controller can indeed cause a single point failure in SDN, which can be exploited by an adversary to launch a controller capacity saturation attack.

2.2 Proactive and reactive rule installation

In SDN, a controller controls the network mainly by installing flow rules at switches. Currently, there are two different modes to install flow rules: proactive/push mode and reactive/pull mode.

In the proactive mode, the controller pre-installs flow rules in the flow tables of switches. Then packets matching these rules will be forwarded at line speed, without interacting with the controller. However, since hardware rule tables (implemented with ternary content addressable memory (TCAM)) are rather scarce resources, it is impractical to pre-install all rules at switches.

In the reactive mode, a switch does not need to store all the rules. Instead, it requests flow rules whenever it does not know how to process a new flow. Even though the reactive mode does not require a large rule table at switches, it can incur a large flow setup latency. The root cause we found is the complicated interaction between controllers and applications.

We further illustrate how the interaction between controllers and applications can slow down the processing of flow requests. When a flow request comes to a controller, the controller will first trigger an internal event, which will be dispatched to all applications that have registered for that kind of event. These applications will then query the network view application programming interface (API) maintained by the controller for network status (e.g., topology and link status). Next, the applications will compute a set of flow rules, and then call the controller API again to install these rules. Note that these interactions take place in sequence, and can hardly be made parallel in time.

As stated above, both the proactive and reactive modes of rule installation have their respective problems. This motivates us to find a more efficient approach to speed up the processing of flow requests in SDN.

3 Design of distributed rule store

In this section, we first present the architecture of DRS, and then elaborate the design details.

3.1 Architecture

Fig. 2 shows the architecture of DRS, which consists of three layers. At the bottom layer are the controller instances (or simply controllers), each of which controls a subset of switches. These controllers manage connections with switches via standard southbound APIs (e.g., OpenFlow), collect topology information and statistics, accept requests from switches, and install flow rules. At the top layer are applications that realize various network policies, including routing, network slicing, access control, and traffic engineering. In between the controllers and applications are two modules: the global network view and the distributed rule store.

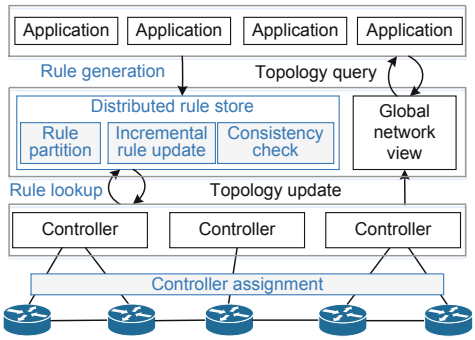

Fig. 2 Architecture of the distributed rule store

The global network view is a database that stores the network states, including network topology, link status, and switch information. It is shared by all controllers: each controller propagates its local network updates to other controllers, and constructs the global network view by assembling all network updates received from other controllers.

The DRS is a distributed key-value store for flow rules, acting as a medium between controllers and applications. It intercepts flow rules pre-computed or installed by applications, and the controllers look up in the DRS for flow rules that can satisfy the flow requests. In this way, applications are decoupled from the flow request processing, and thus the reactive rule installation can become faster.

Note that we are not using off-the-shelf solutions like Cassandra (Lakshman and Malik, 2010) or

RAMCloud (Ousterhout *et al.*, 2010) as in ONOS. The reason is that we want to customize our key-value store so as to support our consistency check algorithms, as will be shown in Section 3.5. In addition, a distributed key-value store does not take locality into account. When a controller needs to retrieve a set of rules, these rules may be physically stored at many other controller instances. On the other hand, DRS can partition rules in a more flexible manner so as to minimize the possibility of remote retrieval.

There are four modules in the DRS architecture: (1) The rule partition module partitions and distributes rules among multiple controllers; (2) The consistency check module checks whether multiple replicas of each rule are consistent, and repairs faulty rules when detecting them; (3) The incremental rule update module updates rules when the network status changes, in an incremental way; (4) The controller assignment module assigns a controller for each switch so that the request processing load is balanced among multiple controllers. In the following, we give design details of these modules.

3.2 Rule generation and update

There are two ways that rules are generated and cached in DRS. For the first one, applications can pre-compute all the rules and push them into the rule store. This method requires enumeration of all the possible requesting flows, and thus is not suitable for all applications. For the second one, when an application installs a new rule, it attaches an expire time to it. The controller then caches this rule together with the expiration time. When there is a flow that requests the same rule, the controller checks whether the rule has expired. If not, the controller directly installs this rule on switches. This method is less demanding for applications. We implemented the above two methods in our DRS-based distributed controller. Note that DRS can work seamlessly with legacy applications, which can neither pre-compute rules, nor attach expiration time to enable rule caching. For such applications, DRS simply handles the flow request to applications as usual, without caching any rule installed by them.

When the network topology changes, e.g., a link becomes up or down, rules in DRS should be updated. To save update cost, DRS updates only those rules that are affected by the topology changes. For routing applications, we achieve this as follows. The controller maintains a table to record all link-to-rule mappings. The key of the table is the link ID, and the value is the set of rules which use that link. When the link status changes, only those flow rules that use this link will be re-computed. Node failures can be seen as a special case where multiple links (those associated with this node) are down. Then all rules associated with these links are recomputed.

3.3 Rule partition

DRS partitions flow rules and stores them in a distributed hash table (DHT), which is inspired by Chord (Stoica *et al.*, 2001). The major difference from Chord is that DRS stores multiple replicas for each flow rule, and maintains consistency among these replicas.

The detailed procedure of rule partition is as follows. First, for each controller with identifier C, we calculate its index as $H(C)$, where H is a consistent hash function (Karger *et al.*, 1997). Controllers are arranged in a ring clockwise in ascending indices. Second, for each rule r with matching field $r.match$, we calculate its index as $H(r.match)$. Then r is assigned to a controller C if C has the smallest index that is larger than $H(r.match)$. For redundancy, r is also replicated at another $k-1$ controllers, where k is termed the redundancy rate. In this case, we say controller C assumes the primary role for rule r, and the other $k-1$ controllers assume a secondary role for rule r.

Fig. 3 is an example where $k=3$. Apart from controller C_2, rule r is also stored at two neighboring controllers, i.e., C_1 and C_3. Here, C_2 is the primary controller; C_1 and C_3 are secondary controllers, with respect to rule r. As the number of controllers is relatively small (unlikely to exceed 10), we let each controller store the IP addresses and the indices of all other controllers. In this way, it is to locate the controllers where a rule is stored.

Note that multiple applications may generate rules with the same matching field. Even for the same application, it may also generate multiple rules for the same matching field. For example, a routing application will generate a forwarding rule for each switch along the forwarding path, and all these forwarding rules have the same matching field (e.g.,

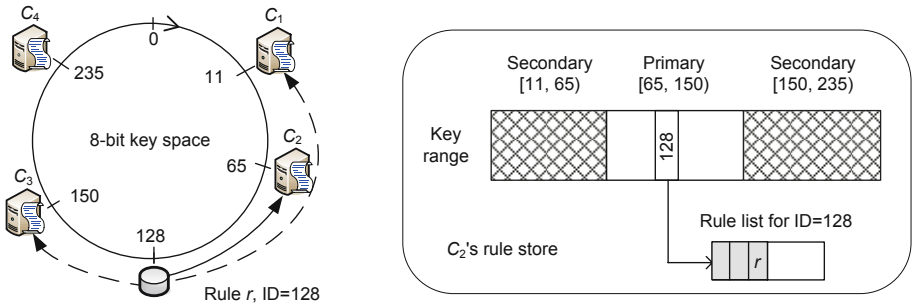

Fig. 3 An example for rule partition in DRS. The output of the hash function is eight bits, and there are four controllers. Rule r has a hash value of 128 and is stored at controller C_2 and C_2's two neighbors (C_1 and C_3)

destination IP address). Thus, inside a controller's rule store, each entry is actually a list of rules with the same index. Apart from matching fields and actions, each entry also stores meta data including switch ID, priority, and expiration time.

3.4 Request processing

The processing of a flow request can be divided into two phases: rule retrieval phase and rule installation phase. Each phase requires cooperation among multiple controllers. The whole process is summarized in Algorithm 1.

1. Rule retrieval phase

When receiving a flow request f, a controller C calculates the rule index as $H(f.match)$, where $f.match$ is the matching field of f. Based on the rule index, the controller identifies the primary controller and all other secondary controllers. If C itself is among these controllers, then it just retrieves the rules from its local rule store. Otherwise, C tries to retrieve the rules from the primary controller. If the primary controller is not responding, then C retrieves them from other secondary controllers.

2. Rule installation phase

After retrieving the rules, C begins to install these rules one by one. For each rule r, if r has not expired, C checks whether the switch S at which to install r is in its own site (i.e., directly connected with controller C). If so, C installs r directly; otherwise, C asks the remote controller which manages S to install r.

Note that rules should be installed in the reverse order as compared to the order by which they appear on the path, so as to guarantee consistency. Specifically, suppose controller C_1 needs to set up a path $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow$ Dst, where S_1 is controlled by controller C_1 and the other two switches are controlled by controller C_2. Then C_1 would remotely call C_2 to install the rules for $S_3 \rightarrow$ Dst and $S_2 \rightarrow S_3$ in sequence, and then install the rules for $S_1 \rightarrow S_2$. In this way, the packet will be sent only when all rules along the path have been installed. Even so, inconsistency is still possible due to transmission delays from controllers to switches. There are many previous works that tried to guarantee strong consistency during network updates (McGeer, 2012; Reitblatt *et al.*, 2012; Katta *et al.*, 2013; Mahajan and Wattenhofer, 2013; Perešíni *et al.*, 2013;

Algorithm 1 Flow request processing

Input: the set of switches managed by the controller, S; the flow request sent from switch sw and with matching field $f.match$, f.

```
1: rid ← H(f.match)
2: C ← LocateByRule(rid)
3: R ← {}
4: for all c ∈ C do
5:    if c is available then
6:       R ← SearchRules(c, rid)
7:       break
8:    end if
9: end for
10: if R == {} then
11:    Handle the flow request to applications
12: end if
13: for all r ∈ R do
14:    if r.switch ∈ S then
15:       Install rule r at switch r.switch
16:    else
17:       c ← LocateBySwitch(r.switch)
18:       Call controller c to install rule r
19:    end if
20: end for
```

Paul, 2014). We will not go into details here, since the focus of this study is to improve the control plane performance rather than guarantee strong consistency on the data plane.

3.5 Consistency check

As DRS maintains multiple replicas for each rule, it should ensure that these replicas are consistent over time. This can prevent faulty rules from being installed at a switch causing problems in the network. In addition, consistency checks can defend controllers against malicious rule modifications at a single controller.

To maintain consistency, controllers cooperatively check each rule periodically. When a new round of consistency checks begins, each secondary controller sends its rules to the primary controller, which checks whether all replicas of each rule are consistent. If not, it resolves the conflicts and repairs the faulty rules.

Currently, we use a simple majority voting method to resolve conflicts. Specifically, supposing the redundancy rate is k, this approach can detect up to $\lfloor (k-1)/2 \rfloor$ faulty replicas. If any faulty replica is detected, the primary controller sends the correct copy to controllers that hold faulty replicas for correction.

We use the Merkle hash tree (MHT) (Merkle, 1988) to reduce the overhead of consistency check. Take Fig. 4 as an example, where there are four controllers, and the redundancy rate is $k=3$. Controller C_2 assumes the primary role for rules R_1 through R_4. The hash values of these rules are made as the leaf nodes. The parent node of two adjacent leaf nodes is the hash of their concatenation. This process continues iteratively until the root node (MHT22) is obtained. Controller C_2 also maintains another two MHTs for rules for which C_2 assumes a secondary role (C_1 and C_3 assume the primary role for these rules, respectively). Similarly, each of the other three controllers also maintains three MHTs.

When consistency check begins, C_1 and C_3 send MHT12 and MHT32 to C_2, respectively. Then $C2$ checks whether MHT22=MHT12=MHT32. If the equation does not hold, then C_2 continues to check these MHTs to locate the faulty rules. As shown in Fig. 4, C_2 would search MHT22 down through A22 and A13, and finally find that R_3 is faulty. After locating the faulty rule R_3, C_2 would replace the faulty rule with the correct replica from C_1 or C_3. Since controllers exchange only a small number of hash values in each round, the MHT-based consistency check can significantly reduce the overhead if faults are not too frequent. We will evaluate the overhead of the MHT-based consistency check in Section 4.3.

3.6 Controller assignment

In the following, we show how the DRS assigns controllers to switches, such that each controller has relatively the same processing load. Suppose there are n controllers and m switches. Then the controller assignment problem can be formulated as the following mixed integer program:

$$\min \max_{j\in\{1,n\}} \sum_{i\in\{1,m\}} x_{i,j}\omega_i$$

$$\text{s.t. (1) } \omega_k = \sum_{i\in\{1,m\}} \sum_{j\in\{1,m\}} r_{i,j} d_{i,j}^k,\ \forall k \in \{1,m\},$$

$$(2)\ \sum_{j\in\{1,n\}} x_{i,j} = 1,\ \forall i \in \{1,m\},$$

$$(3)\ \sum_{j\in\{1,n\}} x_{i,j} \prod_{k\in a(i)} x_{k,j} = 1,\ \forall i \in \{1,m\},$$

$$(4)\ x_{i,j} \in \{0,1\},\ \forall i \in \{1,m\}, \forall j \in \{1,n\}.$$

Here $x_{i,j}=1$ if switch i is assigned to controller

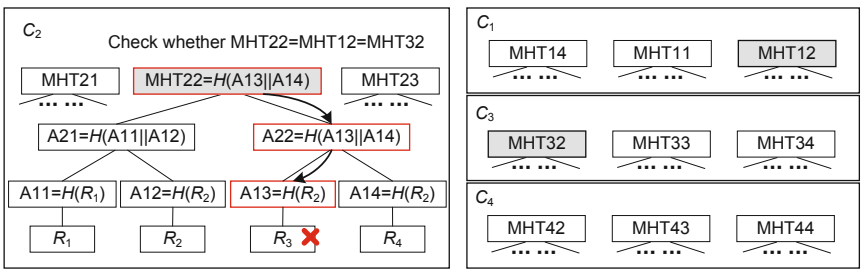

Fig. 4 An example for consistency check in DRS based on the Merkle hash tree

j, or 0 otherwise; $d^k_{i,j} = 1$ if switch k is on the path of a flow from switch i to switch j, or 0 otherwise; $r_{i,j}$ is the flow setup rates from switch i to switch j, i.e., the number of new flows per unit time; $a(i)$ is the set of switches that should be assigned to the same controller with switch i. Constraint (3) ensures that once switch j is assigned to controller j, all switches in $a(i)$ should also be assigned to controller j. Since the problem is NP, we use a simple heuristic algorithm (Algorithm 2) to find an approximate solution.

Algorithm 2 Heuristic controller assignment

Input: the flow request rate from switch i to switch j, $r(i,j)$; a variable indicating whether switch k is on the path of flow from switch i to switch j, $d^k_{i,j}$.

Output: controller assignment strategy, Ctr.

```
1: s(i) ← 0, ∀i ∈ [1, m]
2: c(i) ← 0, ∀i ∈ [1, n]
3: for i, j, k ∈ [1, m] do
4:     if d^k_{i,j} == 1 then
5:         s(k) ← s(k) + r(i, j)
6:     end if
7: end for
8: Sort switches by descending order of s(i)
9: for i ∈ [1, m] do
10:    j ← argmin_k c(k)
11:    Ctr(i) ← j
12:    c(j) ← c(j) + s(i)
13: end for
```

In Algorithm 2, lines 3–7 compute the flow request rate for each switch, given the flow setup rates of all source-destination switch pairs. Line 8 sorts switches by descending order of request rates. Lines 9–13 assign each switch to a controller with the minimum accumulated request rates.

4 Implementation and evaluation

In this section, we first present the implementation of our DRS controller, and then evaluate its performance with emulation.

4.1 Implementation

We implement the DRS distributed controller based on Floodlight version 1.0, which supports the OpenFlow-1.0 protocol (Floodlight Project, 2016). We modify the processing logic of packet-in messages, according to Algorithm 1. Each controller allocates additional memory space for rule storage. The communication among multiple controllers (including rule retrieval, rule installation, consistency check, etc.) is implemented with a remote procedure call (RPC). Since Floodlight is written in Java, we use the Java Remote Method Invocation (RMI) for RPC. There are around 2000 lines of code (LOC) in total.

4.2 Setup and methodology

We use Mininet (Lantz *et al.*, 2010) to emulate a network of Open vSwitches (Pfaff *et al.*, 2009), on a Linux server with two Intel Xeon CPUs (each with six cores) and 32 GB DDR3 memory. We run controllers in separate virtual machines (VMs) hosted on another two Linux servers, and each controller is allocated two CPU cores. Before the experiment, let each pair of hosts ping each other so that the controller can discover all the hosts.

We will compare DRS with two controllers: Floodlight (the native Floodlight controller, which supports only a single controller (Floodlight Project, 2016)) and ONOS (a distributed controller whose implementation is based on Floodlight (Berde *et al.*, 2014)).

We will consider the following five performance metrics:

1. Consistency check efficiency: the time for controllers to detect and repair faulty rules. It measures the efficiency of the consistency check method based on MHT.

2. Flow setup latency: the time between a controller receiving a flow request, and when the corresponding flow rules are sent out. This measures the controllers' response time to flow requests, which reflects the flow setup latency.

3. Processing throughput: the number of requests per second that can be processed by the controllers. This measures the processing throughput of the controllers.

4. Load balance: the maximum processing load of a controller. This measures how well balanced the processing load is among multiple controllers.

5. Rule update overhead: the time for controllers to update rules when link states change. This measures the overhead incurred by our rule update method.

4.3 Consistency check efficiency

This experiment evaluates the cost of MHT-based consistency check in DRS. We construct two FatTree topologies ($k = 4$ and $k = 8$), which are controlled by four controllers. The total forwarding rules in these two topologies are 2184 and 417 028, respectively. To generate faulty rules, we randomly modify a set of rules in one of these four controllers. Fig. 5 reports the time for controllers to detect and repair the faulty rules, where the numbers of faulty rules are 1, 10, 50, and 100, respectively. The error bars indicate the 30- and 70-percentile values. We can see that the cost of consistency check is rather small when there are fewer than 10 faulty rules, for both topologies. In addition, with the increase of the number of faulty rules, the cost grows sub-linearly.

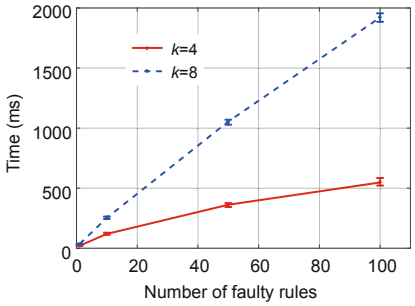

Fig. 5 Time cost of the consistency check under different numbers of faulty rules ($k = 4$ and $k = 8$). The error bars for each data point correspond to the 30- and 70-percentile values

4.4 Flow setup latency

We first compare the response time of DRS with that of Floodlight. We run one controller for both DRS and Floodlight to ensure a fair comparison. We use a FatTree ($k = 6$) topology, and let one host A periodically ping another host B. Since A and B reside on different pods in this topology, the shortest path between them has six hops. The access switch connected with host A will send flow setup requests to the controller (in the form of packet-in messages). We run a simple shortest-path routing application on the controller to process flow requests and install rules. We set the ping interval to 5 s, and the hard rule timeout to 1 s. In this way, each ping will trigger the access switch to send a flow request to the controller.

Fig. 6a reports the ping time for both DRS and Floodlight. We can see that DRS has a much shorter

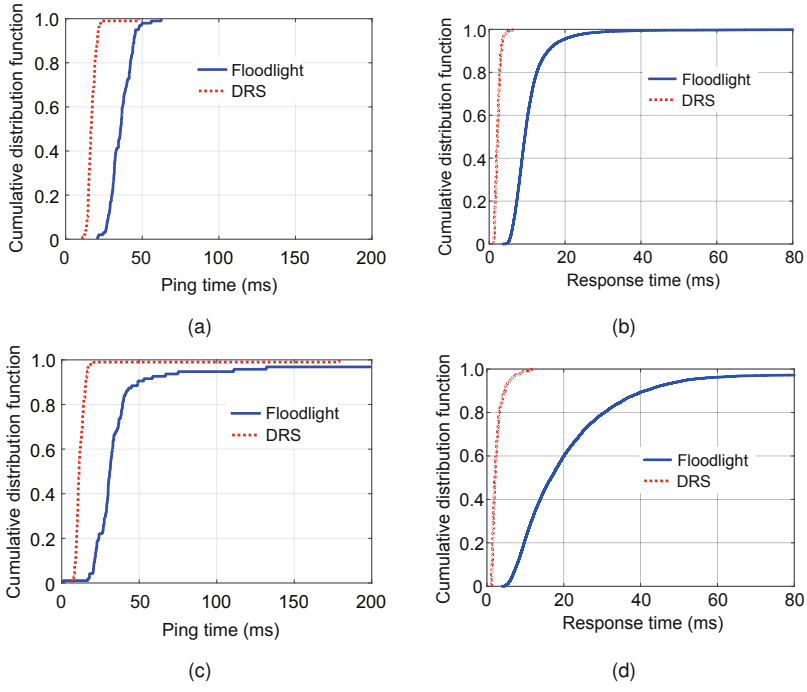

Fig. 6 Comparison of DRS and Floodlight on ping time and response time: (a) ping time without background flow requests; (b) response time without background flow requests; (c) ping time with background flow requests of 1000 per second; (d) response time with background flow requests of 1000 per second

ping time than Floodlight. Note that the ping time is composed of 2× response time to flow requests, and a round-trip propagation time. Fig. 6b reports the controller's response time to flow requests only, measured by the elapsed time from the time at which a request is received by the controller to the time at which the flow rules are sent out by the controller. We can see that the gap between DRS and Floodlight is even more remarkable.

In the above experiments, there are no background flow requests. Figs. 6c and 6d show the results when there are 1000 background requests per second. We observe that the ping and response times for DRS are barely affected, while those for Floodlight increase.

The above results show that the rule caching strategy used in DRS is effective in reducing the controller's response time. The slow response of the Floodlight controller is mainly due to the complicated interaction of the routing application with the Floodlight REST API.

Following the above experiments that measure the response time of a single controller, we continue to evaluate the response time when there are multiple DRS controllers. Different from the above experiments, we implement the routing application as a module in the controller. This can make the interaction of application with the controllers more efficient, compared with that using the REST API. To simplify the assignment of controllers to switches, we use a linear topology consisting of 100 switches. We let the first switch ping the last switch periodically, and measure the ping time.

Fig. 7 reports the ping time for DRS and ONOS, when there are $n = 2, 3, 4$ controllers. We can see that for both DRS and ONOS, the ping time drops when there are more controllers. The ping time for DRS is shorter than that for ONOS when they use the same number of controllers. The results demonstrate the advantage of DRS over ONOS in fast response to flow requests.

4.5 Processing throughput

This experiment measures the processing throughput of DRS, compared with Floodlight. We use a FatTree ($k = 4$) topology, and let multiple end hosts send UDP packets to their neighboring hosts with changing destination port numbers. The controllers install rules that exactly match on the destination port, so that every UDP packet will trigger the receiving switch to send a packet-in message to one of the controllers. To reduce the data-plane traffic, we clear the action lists of each rule installed by the controllers, so that the UDP packet will be discarded without further transmission. We increase the sending rates to saturate the controllers' processing capacity, and report the results in Fig. 8.

From Fig. 8, we can see that DRS with a single controller has roughly the same processing throughput as Floodlight. In addition, the processing throughput of DRS increases when there are multiple controllers. This demonstrates the scalability of DRS when supporting large networks.

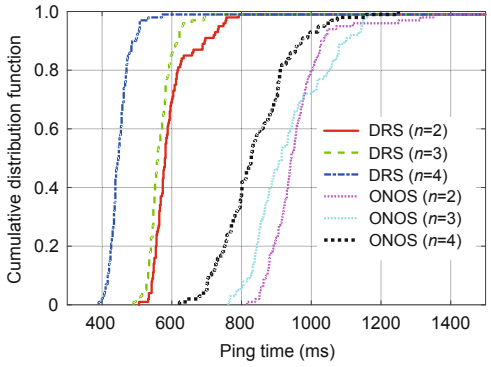

Fig. 7 Comparison of DRS and ONOS on ping time. The ping is made from the first to the last switch in a linear topology of 100 switches

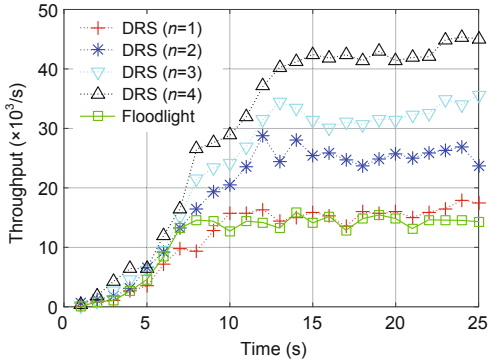

Fig. 8 Comparison of DRS and Floodlight on processing throughput

4.6 Load balance

First, we use simulation to evaluate the approximation rate of Algorithm 2 that we have proposed

for controller assignment. We use a FatTree ($k = 4$) topology with four controllers, and generate random request rates for each pair of switches. We run Algorithm 2 to obtain an approximate assignment strategy, where the maximum load of a controller is L_1. Then we exhaust all assignment strategies to find the optimal value (L_2), and calculate the approximation ratio as L_2/L_1. Fig. 9 reports the CDF of the approximation ratio, where the distribution is obtained from 100 experimental runs. We can see that the ratio is strictly above 0.9 for this topology, meaning that the assignment strategy obtained using Algorithm 2 can well approximate the optimal one.

Then we continue to validate the effectiveness of Algorithm 2 in emulated networks. We construct a FatTree ($k = 4$) topology with four controllers, and let each switch initiate flows to all other switches at the same rate. We apply Algorithm 2 to find an assignment strategy, and use it to configure the connections among switches and controllers. The processing load of a controller is measured by the number of FlowMod messages it sends. For comparison, we also apply a random strategy to assign controllers.

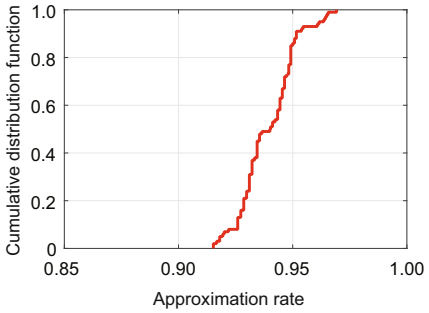

Fig. 9 Approximation rate of Algorithm 2

Figs. 10a and 10b report the numbers of FlowMod messages sent by each controller, under random assignment and Algorithm 2, respectively. We can see that with random assignment, the processing loads of controllers are rather uneven. The busiest controller C_4 has a load that is nearly five times that of controller C_3. On the other hand, the approximate assignment can achieve a well balanced load distribution. Actually, we found that this is also the optimal solution under this specific setting.

4.7 Rule update overhead

This experiment evaluates the cost of the incremental rule update method introduced in Section 3.2. We still consider the routing application, and we are interested in how many rules need to be recomputed if a link state change occurs. Note that the update cost varies among links. For example, link state changes at the bottom layer of a tree network will have a small effect on rules, since the number of flows using these links is relatively small. On the other hand, link ups and downs at the core layer can affect many rules. Thus, apart from the average cost, we measure the maximum and minimum update costs, respectively.

Table 1 shows the update percentage and time for four different topologies, i.e., linear ($n = 100$) and FatTrees ($k = 4, 6, 8$). Here, the update percentage is defined as the ratio of the number of updated rules to the number of all rules. We can see that for FatTrees, a single link state change will cause only a small portion of rules to be recomputed, due to mainly the multi-path nature of FatTrees. On the other hand, the linear topology stands as an extreme case, where

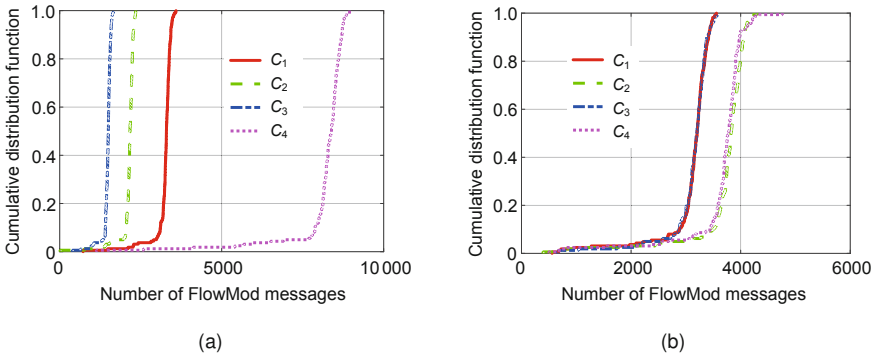

Fig. 10 Processing load of each controller, with random assignment (a) and Algorithm 2 (b)

Table 1 Rule update cost when using our incremental update strategy

Metric	Linear ($n=100$)	FatTree ($k=4$)	FatTree ($k=6$)	FatTree ($k=8$)
Minimum percentage	<0.01%	0.27%	0.02%	<0.01%
Maximum percentage	36.76%	3.07%	0.61%	0.19%
Average percentage	12.57%	2.23%	0.41%	0.12%
Minimum time (ms)	<1	<1	<1	<1
Maximum time (ms)	8763	61	81	243
Average time (ms)	2237	16	27	42

there is only one path between any pair of switches. Thus, we can see that the update cost is much higher than that in FatTrees. The variance of update cost is also very large. For example, the minimum update percentage is less than 0.01%, while the maximum value is 36.76%.

5 Related work

5.1 Controller capacity saturation

Shin *et al.* (2013) introduced the controller capacity saturation attack, in which an adversary tries to exhaust the controller's resource by mounting SYN flooding attacks. They implemented a connection migration method to proxy all Transmission Control Protocol (TCP) connections at switches. This method works for only TCP SYN flooding, without considering other distributed denial-of-service (DDoS) attack methods like UDP flooding. In addition, it requires much modification at SDN switches.

5.2 SDN controllers

There are several open source SDN controllers, like NOX (Gude *et al.*, 2008), its Python variant POX (NOXRepo, 2016), Ryu (Ryu SDN Framework Community, 2014), and OpenDaylight (OpenDaylight Project, 2016). These controllers focus mainly on the single-controller scenario, and are not optimized for distributed control.

5.3 Distributed SDN controllers

Many distributed SDN controller solutions have been proposed to improve the control plane performance. We group these solutions into two classes, depending on whether a distributed storage system is used.

The first class includes Hyperflow (Tootoonchian and Ganjali, 2010), Kandoo (Yeganeh and Ganjali, 2012), and Pratyaastha (Krishnamurthy *et al.*, 2014). Hyperflow (Tootoonchian and Ganjali, 2010) is an early effort in designing distributed SDN controllers. It introduces a publish/subscribe mechanism atop the NOX controller using distributed file systems. Individual controllers communicate with one another, so that each controller has the global view. In this way, controllers can locally process all requests from switches they control. However, the frequent controller-to-controller interaction can incur a large communication overhead. Kandoo (Yeganeh and Ganjali, 2012) and Pratyaastha (Krishnamurthy *et al.*, 2014) mitigate this problem by partitioning network states. Each controller maintains only a subset of states, and can handle local events in the subnet controlled by it. Specifically, Kandoo distinguishes between local applications requiring only local states, and global applications requiring network-wide states. Local applications run on multiple local controllers, while global applications run on the root controller. A problem with Kandoo is that the root controller can cause single-point failures. In addition, both Kandoo and Pratyaastha are application-dependent, and require application developers to carefully partition states, which can become a highly demanding task.

The second class includes Onix (Koponen *et al.*, 2010) and ONOS (Berde *et al.*, 2014). They use distributed storage systems to maintain the global network states, which provide a graph API for applications. Controllers are responsible mainly for managing connections with switches, and propagating events to the distributed storage systems. For these solutions, the distributed storage system acts as a middle layer for applications and controllers. Any request from a switch will go through the controller, the storage systems, and then the applications. After the applications finish processing the request, the

flow rules will be propagated all the way back to the switch. This incurs a relatively large latency when processing flow requests.

5.4 SDN controller migration

ElastiCon (Dixit *et al.*, 2013) studies how to achieve load balance among multiple controllers. The authors leveraged OpenFlow to perform switch migration, so that switches can migrate from one controller to another depending on controllers' current loads. The migration approach is orthogonal to our multi-controller architecture, but can be leveraged to reassign controllers periodically in DRS.

6 Conclusions

This paper deals with the security and performance issues of current SDN controllers. We proposed DRS, a new multi-controller solution which can help mitigate security issues in SDN, and can improve the performance of SDN controllers. Specifically, DRS can mitigate the controller capacity saturation attack by strategically distributing flow rules and assigning controllers. In addition, DRS uses an MHT-based consistency check to detect rule modifications. As for performance, DRS uses rule caching to reduce the response time to flow requests, and thereby can achieve a higher processing throughput. We realized a distributed controller based on DRS and evaluated it with Mininet. Experimental results demonstrated the efficiency of consistency check and rule update, and also showed that DRS outperforms native Floodlight and ONOS in terms of flow setup time and processing throughput.

References

Berde, P., Gerola, M., Hart, J., *et al.*, 2014. ONOS: towards an open, distributed SDN OS. Proc. 3rd Workshop on Hot Topics in Software Defined Networking, p.1-6. http://dx.doi.org/10.1145/2620728.2620744

Dittrich, D., 1999. The DoS Project's 'Trinoo' Distributed Denial of Service Attack Tool. University of Washington, USA. Available from http://staff.washington.edu/dittrich/misc/trinoo.analysis.txt.

Dixit, A., Hao, F., Mukherjee, S., *et al.*, 2013. Towards an elastic distributed SDN controller. *ACM SIGCOMM Comput. Commun. Rev.*, **43**(4):7-12. http://dx.doi.org/10.1145/2534169.2491193

Floodlight Project, 2016. Floodlight Controller. Available from http://www.projectfloodlight.org/floodlight/.

Gude, N., Koponen, T., Pettit, J., *et al.*, 2008. NOX: towards an operating system for networks. *ACM SIGCOMM Comput. Commun. Rev.*, **38**(3):105-110. http://dx.doi.org/10.1145/1384609.1384625

Karger, D., Lehman, E., Leighton, T., *et al.*, 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. Proc. 29th Annual ACM Symp. on Theory of Computing, p.654-663. http://dx.doi.org/10.1145/258533.258660

Katta, N.P., Rexford, J., Walker, D., 2013. Incremental consistent updates. Proc. 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, p.49-54. http://dx.doi.org/10.1145/2491185.2491191

Koponen, T., Casado, M., Gude, N., *et al.*, 2010. Onix: a distributed control platform for large-scale production networks. Proc. 9th USENIX Symp. on Operating Systems Design and Implementation, p.1-6.

Krishnamurthy, A., Chandrabose, S.P., Gember-Jacobson, A., 2014. Pratyaastha: an efficient elastic distributed SDN control plane. Proc. 3rd Workshop on Hot Topics in Software Defined Networking, p.133-138. http://dx.doi.org/10.1145/2620728.2620748

Lakshman, A., Malik, P., 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.*, **44**(2):35-40. http://dx.doi.org/10.1145/1773912.1773922

Lantz, B., Heller, B., McKeown, N., 2010. A network in a laptop: rapid prototyping for software-defined networks. Proc. 9th ACM SIGCOMM Workshop on Hot Topics in Networks, Article 19. http://dx.doi.org/10.1145/1868447.1868466

Mahajan, R., Wattenhofer, R., 2013. On consistent updates in software defined networks. Proc. 12th ACM Workshop on Hot Topics in Networks, Article 20. http://dx.doi.org/10.1145/2535771.2535791

McGeer, R., 2012. A safe, efficient update protocol for OpenFlow networks. Proc. 1st Workshop on Hot Topics in Software Defined Networks, p.61-66. http://dx.doi.org/10.1145/2342441.2342454

McKeown, N., Anderson, T., Balakrishnan, H., *et al.*, 2008. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Comput. Commun. Rev.*, **38**(2):69-74. http://dx.doi.org/10.1145/1355734.1355746

Merkle, R.C., 1988. A digital signature based on a conventional encryption function. *In*: Pomerance, C. (Ed.), Advances in Cryptology, p.369-378. http://dx.doi.org/10.1007/3-540-48184-2_32

NOXRepo, 2016. The POX Controller. Available from http://www.noxrepo.org/.

OpenDaylight Project, 2016. The OpenDaylight Controller. Available from https://www.opendaylight.org/.

Ousterhout, J., Agrawal, P., Erickson, D., *et al.*, 2010. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Oper. Syst. Rev.*, **43**(4):92-105. http://dx.doi.org/10.1145/1713254.1713276

Paul, S., 2014. Software Defined Application Delivery Networking. PhD Thesis, School of Engineering & Applied Science, Washington University in St. Louis, USA. http://dx.doi.org/10.7936/K7CJ8BJH

Perešíni, P., Kuzniar, M., Vasić, N., *et al.*, 2013. OF.CPP: consistent packet processing for OpenFlow. Proc. 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, p.97-102. http://dx.doi.org/10.1145/2491185.2491205

Pfaff, B., Pettit, J., Amidon, K., *et al.*, 2009. Extending Networking into the Virtualization Layer. Available from http://openvswitch.github.io/papers/hotnets2009.pdf.

Reitblatt, M., Foster, N., Rexford, J., *et al.*, 2012. Abstractions for network update. Proc. ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocals for Computer Communication, p.323-334. http://dx.doi.org/10.1145/2342356.2342427

Ryu SDN Framework Community, 2014. The Ryu Controller. Available from http://osrg.github.io/ryu/.

Shin, S., Yegneswaran, V., Porras, P., *et al.*, 2013. AVANT-GUARD: scalable and vigilant switch flow management in software-defined networks. Proc. ACM SIGSAC Conf. on Computer & Communications Security, p.413-424. http://dx.doi.org/10.1145/2508859.2516684

Stoica, I., Morris, R., Karger, D., *et al.*, 2001. Chord: a scalable peer-to-peer lookup service for Internet applications. *ACM SIGCOMM Comput. Commun. Rev.*, **31**(4):149-160. http://dx.doi.org/10.1145/964723.383071

Tootoonchian, A., Ganjali, Y., 2010. HyperFlow: a distributed control plane for OpenFlow. Proc. Internet Network Management Conf. on Research on Enterprise Networking, p.1-6.

Yeganeh, S.H., Ganjali, Y., 2012. Kandoo: a framework for efficient and scalable offloading of control applications. Proc. 1st Workshop on Hot Topics in Software Defined Networks, p.19-24. http://dx.doi.org/10.1145/2342441.2342446