AdaFlow: Adaptive Control to Improve Availability of OpenFlow Forwarding for Burst Quantity of Flows

Boyang Zhou[1], Wen Gao[1], Chunming Wu[1](✉), Bin Wang[1], Ming Jiang[2], and Yansong Wang[3]

[1] College of Computer Science, Zhejiang University, Hangzhou 310027, China
{zby,gavingao,wuchunming,bin_wang}zju.edu.cn
[2] Hangzhou Dianzi University, Hangzhou 310018, China
jmzju@163.com
[3] ZTE Corporation, Nanjing 210012, China
wang.yansong@zte.com.cn

Abstract. The Software-Defined Networking (SDN) separates the control plane from the data plane to increase the flexibility. In the data plane, the unavailability of data forwarding is a common problem preventing a switch from configuring a new arrival flow into its flow table. When the burst flows arrived at the switch, the flow table can be consumed, causing the unavailability occurred. However, the problem is more complicated than in Internet due to the limited channel bandwidth for detecting the table usage. Hence, we propose a transparent core layer in the controller. The mechanism of the layer improves the availability in such way, configuring switches adapting to arrival patterns of flows to prevent the resource of switch exceeding its limit. This paper introduces the design and mechanisms of the layer as well as their algorithms. We further use a real flow trace from a Internet core router to evaluate the performance of layer. By emulating on on miniNet-HiFi, the results demonstrate that the layer can smooth the burst flows without making the flow table exceeding its size, without the layer, the switch lost 8% ingress flows. Meanwhile, the control throughput is lowered by 25.8% than before.

Keywords: Software-Defined Networks · Network management

1 Introduction

The Software-Defined Networking (SDN) separates the data plane from the control plane to improve the control flexibility [1,2]. The control plane is consisted of multiple controllers. In the data plane, each controller configures its switches via the control channel that supports the OpenFlow (OF) protocol [1]. Each switch has a flow table to define the forwarding rules. By matching against the rules, the ingress flows are forwarded by a switch to output ports of the switch.

Generally, a flow is mismatched due to either new flow arrivals or expiration of the rule. At that time, the switch first queries its controller to retrieve the new rules for the flow and then programs the rules into the flow table. Such process

© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2014
V.C.M. Leung et al. (Eds.): TridentCom 2014, LNICST 137, pp. 406–415, 2014.
DOI: 10.1007/978-3-319-13326-3_39

generates the control traffic in the channel between controller and switch. However, such process has a problem: new ingress flows can be lost by the switch if the flow table is full or the control channel is congested, thus negatively impacting on the availability of switch forwarding for the ingress flows.

When applying SDN to the wide-area networks (WANs), the problem is more challenging than the current applications of SDNs, since the flows are large-scale, burst, transient and intermittent. These features impact the flow table and the channel on their performance along with variation of new flow arrivals. Hence, by exploiting these resources, the performance of data forwarding can be further exploited, yielding a new adaptive control policy to optimize switch's flows.

Realizing such policy is complicated by the separated architecture in SDNs. It is because the policy should be made based on the statistics of switch resources, however, the channel only has the limited bandwidth. The current work on SDN architecture have not improved the availability.

In this paper, we propose a novel transparent core layer (named as the AdaFlow layer) to adaptively control the active count of flow table entries according to the flow arrival patterns, so as to improve the forwarding availability. The mechanism of the layer is consisted of three workflows: (i) The optimization workflow predicts the expiration of an new flow entry according to a estimated flow throughput, smoothing the active count for the burst flows; (ii) The resource workflow predicts the active count by history; and (iii) The estimation workflow estimate the throughput of a flow according to the flow arrival pattern. The workflows (ii) and (iii) provides the estimation inputs to the workflow (i).

We evaluate the performance of the layer by using miniNet-HiFi. The results demonstrate that the layer can smooth a burst quantity of ingress flows without making the switch exceeding the size of flow table. In comparison, without the layer, the switch lost 8% ingress flows. In addition, the control throughput is lowered by 25.8% than before.

The rest of the paper is organized as follows. Section 2 analyzes and states the problem in depth. Section 3 proposes the design and mechanism of AdaFlow layer. Section 4 evaluate the performance of the layer. Section 5 and 6 discuss the related work and conclude the paper.

2 Problem Statement

In this section, we first give the system model of the SDN control on switch and then formulize the problem of availability of switch as a time series problem.

2.1 System Analysis

In SDN, a controller, denoted as c, configures the forwarding table of the i-th switch, denoted as s_i, via the i-th channel, denoted as h_i. All the switches of c are denoted as $S_c = \{s_1, s_2, ..., s_i, ..., s_m\}$, where m is the number of the switches. Fig. 1 gives such an example which shows a common scenario for the OF forwarding. The controller c and its OF switches s_a, s_b and s_i. The bandwidth of h_i

competes with other switches belonging to the controller within a limited physical bandwidth. In the switch s_i, the flow table is consisted of multiple flow table

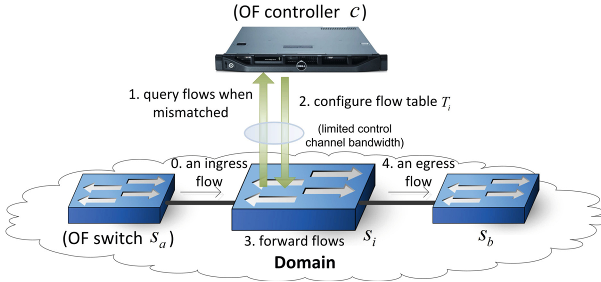


Fig. 1. OpenFlow forwarding example

entries, and each entry is consisted of forwarding rules, actions, hard timeout of flow and idle timeout of flow. The switch executes the actions by matching headers of ingress packets of a flow with the corresponding rules. Formally, we denote the flow table of i-th switch as $T_i = \{f_1, f_2, ..., f_j, ..., f_Q\}$, where f_j is the j-th table entry and Q is the number of the active table entries (thus termed as the active count). The physical size of the table is denoted as L_i. In addition, the idle timeout of the j-th flow entry for the i-th switch is denoted as $k_{i,j}$. It is note that the hard timeout will not be considered in our work as settings of values of the timeouts are different from service to service.

Wherein, the flow table of each switch is initially set to be null. When an ingress flow from s_a is mismatched at s_i, s_i queries new configuration to c via the channel h_i, and then forwards the flow to its output port, e.g., s_b. When the hard timeout of the entry is exceeded or no packet is arrived within the idle timeout, the entry is expired from the table.

For each switch in S_c, the total throughput of the control channel, denoted as λ_{all}, equals to the sum of two types of control traffics for all the switches, namely, configuring the flow tables, and making controllers connections. The latter one is a constant for the number of switches. Formally, the total throughput is specified as Eq. 1, where $C(L)$ is a constant value of L, α_i^t is the hit ratio of the flow table at the time tick t and $\tilde{\theta}_i^t$ is the expiration rate of the flow table of the i-th switch at t. We denote the upper limit of channel bandwidth as Z.

$$\lambda_{all} = \sum_{s_i \in S_c} (1 - E(\tilde{\theta}_i^t \times (1 - \alpha_i^t)) + C(L) \tag{1}$$

Based on the model, we denote the availability ratio of s_i at the time tick t as β_i^t. β_i^t is the probability that the configurations of the flow table entries of s_i, when its ingress flows are arrived in the recent unit time, can be correctly programmed into s_i.

2.2 Problem Statement

The problem is to find the configurations of idle timeouts for all the table flow entries of a switch to minimize the control throughput of the switch restricted by the lower limit of the availability of the switch, given by a continuous time series of ingress flow headers of the switch. We formulize such problem as below.

For a switch s_i at t, we denote the time series of ingress flow arrivals as $G_t = (g_0, g_1, g_2, g_2, g_3, ..., g_b, ..., g_t)$, where b is the time tick of arriving of an ingress flow ranging from 0 to t and g_b is a valuable identifies the arrival flow by uniquely hashing the header of the flow. The controller c controls the s_i, and c defines the lower limit of the availability ratio of switch as β_i for all the time ticks. The idle timeouts of T_i are $K = \{k_{i,1}, k_{i,2}, ..., k_{i,Q_t}\}$, where N_t is the size of flow table at t. The problem is specified as Eq. 2, where $\tilde{v}$ is estimated value of v, and H^t is estimated throughputs of part flows in G_t held in the controller.

$$\begin{aligned} \underset{K \in N^{Q_t}}{\text{argmin}} \ (\tilde{S}_c^{\ t+1}) \text{ , subject to:} \\ \tilde{\beta}_i^{t+1} \geq \beta_i \text{ and } \lambda_{all}^{t+1} \leq Z \\ \text{Given by } G_t \text{ and } H = \{\tilde{\lambda}^t(g_b) : g_b \in F_t \text{ and } 0 \leq t \leq t\} \end{aligned} \tag{2}$$

Such minimization problem is challenging as it requires the controller to predict β and λ_{all} with the limited knowledge of H^t. In next, we discuss an heuristic solution.

3 Design and Mechanism of Adaptive Flow Control

In this section, we propose an adaptive control mechanism to address problem of the availability of switch forwarding when there is a burst quantity of ingress flows arrived at the switch. We also introduce its algorithms.

3.1 Overall Design

In general, the proposed adaptive control mechanism ensures the availability of data plane by predicting the flow arrival patterns and by efficiently measuring the resource usage of the flow table. The mechanism is implemented as a transparent AdaFlow layer in Beacon controller [2]. The layer locates between the network services and the OpenFlow protocol stack. The layer optimizes the service performance by changing the idle timeout in the FLOW_MOD message to make it subject to Eq. 2 and then performing the configuration by sending the message to the switch via the OF stack.

Fig. 2 shows the internal design of the layer. It has three concurrent workflows. The optimization workflow decides the idle timeouts of new arrival flows by receiving the two inputs: (i) the flow table usage that is produced by the resource workflow, and (ii) the input of estimated throughput of the ingress flows that is produced by the estimation workflow.

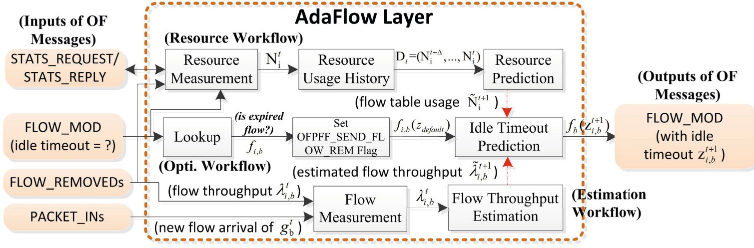


Fig. 2. Internal Design of the AdaFlow Layer

Wherein, the estimation workflow is the simplest. It predicts the throughput of the flow by averaging on all the thoughputs sent by the FLOW_REMOVED messages [3]. The workflow outputs $\lambda_{i,b}^{t+1}$ to the prediction workflow.

We detail the rest of two workflows as follows.

3.2 Optimization Workflow

The optimization workflow decides the optimal value of idle timeout for flow configurations sent from the service. In the workflow, only are the flows with the duration time of larger than γ considered, because that most of survival times of flows live for a very short term even in a core router of a WAN, e.g. 2 seconds (see Subsection 4.1). In next, we give the algorithm of the workflow in the Alg. 1. It has the three steps as following.

First, the lookup step first receives FLOW_MOD message, denoted as f_b. Then, the lookup step lookups f_b to decide it is an expired flow. If not, the idle timeout is set to be a default value, in our prototype, 2 seconds (see lines 5-8 in the Alg. 1). Otherwise, the workflow decides the optimal value for the idle timeout in the next last step.

Second, when received the FLOW_MOD message, the set flag step tag OFPFF_SEND_FLOW_REM to the message to measure its throughput (see the line 10 in the Alg. 1).

Last, the idle timeout prediction step first decides an optimal idle timeout for the flow, denoted as $f_b(Z^{i,b})$, and then send $f_b(Z^{i,b})$ to the OF stack. When the flow table reaches to full with the probability of the availability ratio of switch β_i, the step sets the idle timeout to 1 to ensure the availability (see the line 12 in the Alg. 1). Otherwise, the step uses Eq. 3 to compute the timeout with the availability ratio. The mathematical deduction of Eq. 3 is given by Th. 1.

Such workflow provides an adaptive approach to provide a heuristic solution to the problem that is presented in Eq. 2.

Theorem 1 (Idle Timeout). *Given a flow entry f_b, the estimated throughput of the flow $\lambda_{i,b}^{t+1}$, and a timeout probability $1-\beta$, the idle timeout should be set to Eq. 3, considering the ingress flow is in a Poisson distribution.*

$$k_{i,b}^{t+1} = 1 + \frac{1}{1-\beta_i} + \frac{1}{\lambda_{i,b}^{t+1}} \tag{3}$$

Algorithm 1. Optimization Workflow

Require:
FLOW_MOD message: f_b
Flow throughput estimation: $\lambda_{i,b}^{t+1}$
Active count estimation: $\tilde{N_i}^{t+1}$
Availability ratio of switch: β_i
Ensure: f_b with idle timeout: $f_b(k_{i,b}^{t+1})$
1: $FLOWS = \{\}$
2: **repeat**
3: wait for receiving a FLOW_MOD message f_b
4: **if** $f_b \in FLOWS$ **then**
5: $FLOWS = FLOWS \cap \{f_b\}$
6: $k_{i,b}^{t+1} = \gamma$
7: send $f_b(k_{i,b}^{t+1})$ to s_i via the OF stack
8: continue
9: **end if**
10: tag OFPFF_SEND_FLOW_REM to f_b
11: **if** $\tilde{N_i}^{t+1} > L_i$ **then**
12: $k_{i,b}^{t+1} = 1$
13: **else**
14: $k_{i,b}^{t+1} = 1 + 1/(1-\beta_i) + 1/\lambda_{i,b}^{t+1}$
15: **end if**
16: send $f_b(k_{i,b}^{t+1})$ to s_i via the OF stack
17: **until** true

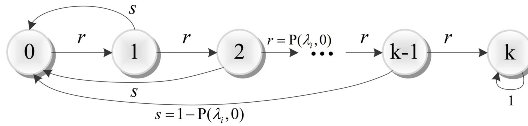


Fig. 3. Transition for timer state of idle timeout

Proof. The state of the timeout value can be modelled by using the Markov chain as follows. Fig. 3 depicts behaviors of the state transition of the timer, where k is the idle timeout, $r = P(\lambda_i, 0)$ is an event when no packet of the flow arrives in a unit time tick, and $s = 1 - P(\lambda_i, 0)$ is the opposite event of r. Each circle represents a state and each arrow is a transition between the states. The state $0 \leq w \leq k-1$ reaches to $w+1 \leq k$ with r, and the state k only can reach to itself. We denoted such transition as a matrix M, where $M(w, v)$ is the probability of transiting from the state w to v. In addition, we denoted a vector, $x = [x_0, x_1, .., x_k]$, as the probabilities at each state, where x_q is the probability of timer at q-th state.

$$\tilde{x} = \frac{r^k}{r^k \times k - r^k + 1} \times [1, \frac{1}{r}, \frac{1}{r^2} - \frac{1}{r} + 1, \frac{1}{r^3} - \frac{1}{r^2} + 1, ..., \frac{1}{r^k} - \frac{1}{r^{k-1}} + 1] \quad (4)$$

$$1 - \beta = P_\theta(f_i, t) = \frac{r^k}{r^k \times k - r^k + 1} \times (\frac{1}{r^k} - \frac{1}{r^{k-1}} + 1) \tag{5}$$

The stationary states of M is when $xM = x$, denoted as $\tilde{x}$. By solving the linear equations, $\tilde{x}$ is computed as Eq. 4. Thus, the probability of state at the timeout is denoted as $1-\beta = P_\theta(f_i, t)$ as Eq. 5. Last, we solve k in Eq. 5 by differentiating on k to get Eq. 3. Proof is done.

3.3 Resource Workflow

The resource workflow predicts the active count of flow table entries, named as the active count, by periodically querying the switch. It has the three steps as following (see the three white rectangles in the top of Fig. 2). First, it measures the active count of flow table entries of the i-th switch at the time tick t, denoted as N_i^t. Then, it saves N_i^t into the memory of controller to form the time series history, denoted as $D = (N_i^{t-\delta}, N_i^{t-\delta+1}, ..., N_i^t)$. The layer only maintains the δ size of the history. Last, it predicts $\tilde{N}_i^{t+1}$ based on the history.

In detail, the algorithm of the prediction is given in the Alg. 2 as below. The algorithm gives the upper limit of active count in the next tick given by the availability ratio of switch β_i. In the lines 4-12, the controller c measures the performance statistics of all the OF switch in S_c for the recent δ seconds in a periodical mode. The statistics cover on all the statistics fields defined the OF specification 1.0 [3], e.g., the active count of flow table entries. In addition, we add two extra statistics, namely, the count of flows removed and the rate of flow modifications (see lines 6-7 in Alg. 2). Based on those statistics, the workflow predicts the active count by Eq. 6 (see lines 13-16 in Alg. 2). In line 13, we use the fast Poisson algorithm to compute the confidence value range for the β_i.

The algorithm output of $\tilde{N}_i^{t+1}$ is utilized by the prediction workflow to compute the optimal idle timeout of the flow (see Subsection 3.1).

$$\tilde{N}_i^{t+1} = \tilde{\lambda}_i^{t+1} + N_i^{t+1} - \theta_i^{t+1} \tag{6}$$

The correctness of the algorithm holds since Eq. 6 exploits the Markov property of state of the active count of flow table entries. Because the state of N_i^{t+1} only depends on the state of N_i^t. Thus, the state can be predicted by its state in the current time tick and patterns of ingress flows, as Eq. 7 shows, where E is the variable expectation, $\tilde{\theta}_i^t$ is the expiration rate of the flow table of the i-th switch at t. Eq. 7 indicates that variation states of N_i^t depends on ingress flows. Hence, Alg. 6 is correct.

$$E(N_i^{t+1}) - E(N_i^t) \approx E(\tilde{\lambda}_i^t) \times (1 - \alpha_i^t) - E(\tilde{\theta}_i^t) \tag{7}$$

4 Evaluation

4.1 Performance Preliminaries

The AdaFlow layer is implemented on the Beacon controller 1.0.3 [2] with support of OF 1.0.3 protocol [3]. The layer registers the listeners for all the OF

Algorithm 2. Resource Workflow

Require:
OF messages: STATS_REQUEST, FLOW_MOD and FLOW_REMOVED
The availability of switch: β_i
Ensure: Flow table usage: $\tilde{N}_i^{t+1}$

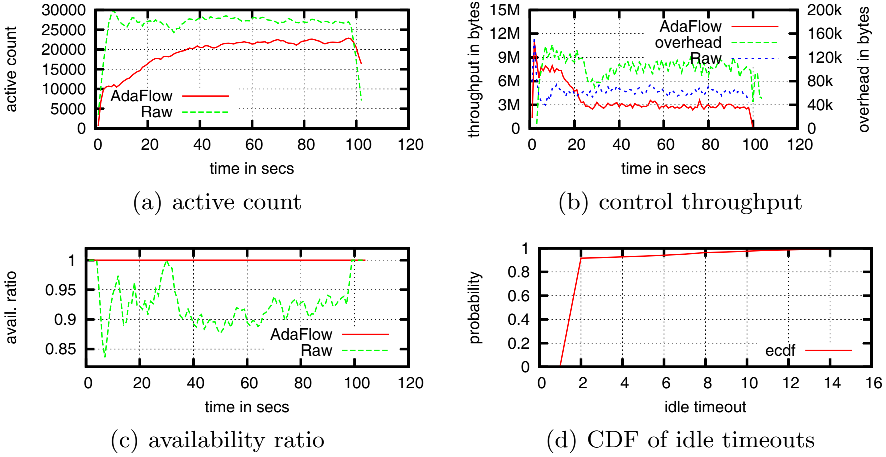
1: $D_{1\leq i\leq |S_c|} = ()$
2: **repeat**
3: wait for a new second t
4: **for** $s_i \in S_c$ **do**
5: $features = send(s_i, \text{STATS_REQUEST})$
6: $features.add(\#$ of modFlows for $s_i)$
7: $features.add($rate of flowRemoved for s_i
8: **if** $|D_i| > \delta$ **then**
9: $D_i.dequeue()$
10: **end if**
11: $D_i.enqueue(features)$
12: **end for**
13: $\tilde{\lambda}_i^{t+1} = PoissonPdf(mean(D_i.modFlows), \beta_i)$
14: $N_i^{t+1} = mean(D_i.activeCount)$
15: $\theta_i^{t+1} = mean(D_i.flowRemoved)$
16: $\tilde{N}_i^{t+1} = \tilde{\lambda}_i^{t+1} + N_i^{t+1} - \theta_i^{t+1}$
17: **until** true

messages required. We test the performance of AdaFlow layer by using the routing service provided by the Beacon itself.

We setup a basic topology as the Fig. 1 shows, to simplify the problem. We emulate the topology by using the miniNet-HiFi [4]. It provides the traffic shaping for links and the cgroup based isolation of resources. We limit the bandwidth of all the links to 1000Mbps. In detail, s_a is emulated as a packet generator by using the TcpReplay tool. The traffic is generated at maximum speed of link. s_b is emulated as a flow receiver that is replaced by the tcpdump tool. And s_i is an OpenvSwitch [5] with supporting of the OF 1.0.3 protocol [3]. We limit the table size of s_i to 25000 in the controller which is slightly larger than the average rate of new flow arrival rate, so that we can emulating the burst events of the flows.

Wherein, the generator replays a real packet trace of a core Internet router amount of 544040 flows[1]. We replay the trace in the speed of 99 seconds, since OS is hard to emulate the trace in its real speed. We count the cumulative distribution function (CDF) of the living time of all the flows in the trace. In the trace, we find the 79.55% of flows only live for less than 2s. Thus, in the Alg. 1, γ is set to 2 seconds. In the Alg. 2, δ is set to 60 seconds. In addition, the availability ratio β_i is set to 0.95 (see Subsection 2.2).

[1] The trace file can be downloaded from the following URL: http://data.caida.org/datasets/passive-2013/equinix-chicago/20130529-130000.UTC/equinix-chicago.dirA.20130529-130100.UTC.anon.pcap



(a) active count (b) control throughput

(c) availability ratio (d) CDF of idle timeouts

Fig. 4. Performance of AdaFlow Layer

4.2 Performance Results

The availability ratio of switch forwarding for s_i is denoted as β_i (see Subsection 2.2). In practical, $\beta_i^t = \#$ of mod flows$/\tilde{\lambda}_i^t$. In the following, we compare performance of the AdaFlow version of the Beacon controller with its raw implementation version by using the same trace previously discussed.

The evaluation results are given in Fig. 4. Fig. 4(a) shows the active count of flow table for the AdaFlow version is decreased by 26.4% than the raw on the average. The means of the active counts for them are 19073 and 25917 (see Fig. 4(a)). In addition, in Fig. 4(b), the control throughput of the AdaFlow version is decreased by 25.8% than the raw on the average. The means of the throughputs for both of them are 3.504MBps and 4.724MBps. For the AdaFlow, the FLOW_REMOVED messages only consume throughput of 100.582KBps. When a burst amount of flows arrived, the availability of switch forwarding can be much improved by such a sharply decreasing. Fig. 4(c) shows the availability ratio of the AdaFlow version is increased by 8% comparing to the raw. The means of the ratios for them are 1 and 0.9261. Fig. 4(d) shows the CDF of the idle timeouts for the AdaFlow version.

5 Related Work

The recent work on the efficiency of control protocol in SDN focus on strengthening the controller architectures in these three ways: (i) The architecture enables the features of multi-threading, multi-core and I/O batching when the controller processes OF messages [6,7], e.g. OpenDayLight [8], Beacon [2] and Maestro [9].

(ii) The architecture clusters several controllers in the same domain to load balance the arrival of OF messages from the switches, e.g. OpenDayLight uses the shared pool to distribute the messages among the controllers [8].

And (iii) The architecture moves part of controller functions to the switch side to improve the performance of switch, e.g. DevoFlow [10] and NOSIX [11].

Hence, none of these researches considers optimizing the OpenFlow protocol itself by controlling the flows according their arrival patterns. In addition, our approach improves performance of these architecture and does not conflict with these ways. Our work is innovative in dealing with the patterns.

6 Conclusion

We propose a novel adaptive control mechanism for SDN by exploiting the arrival patterns of flows and detecting the active count of flow table. The mechanism can smooth a burst quantity of ingress flows to ensure their availability being processed by the switch, meanwhile, lowering the control throughput. We demonstrate these benefits by using a real flow trace from an Internet core router. Our solution provides an easy way to improve performance of SDN services.

Acknowledgments. This work is supported by the National Basic Research Program of China (973 Program) (2012CB315903), the Key Science and Technology Innovation Team Project of Zhejiang Province (2011R50010-05) and the National Natural Science Foundation of China (61379118 and 61103200). This work is sponsored by the Research Fund of ZTE Corporation.

References

1. McKeown, N., Anderson, T., Balakrishnan, H., et al.: Openflow: enabling innovation in campus networks. In: ACM SIGCOMM (2008)
2. Erickson, D.: The beacon openflow controller. In: ACM SIGCOMM Workshop on HotSDN (2013)
3. Openflow switch specification, version 1.0.0, Open Networking Foundation (2009)
4. Handigol, N., Heller, B., Jeyakumar, V., et al.: Reproducible network experiments using container-based emulation. In: ACM International Conference on Emerging Networking Experiments and Technologies (2012)
5. Pfaff, B., Pettit, J., Amidon, K., et al.: Extending networking into the virtualization layer. In: Hotnets (2009)
6. Tootoonchian, A., Gorbunov, S., Ganjali, Y., et al.: On controller performance in software-defined networks. In: USENIX Hot-ICE (2012)
7. Yeganeh, S.H., Tootoonchian, A., Ganjali, Y.: On scalability of software-defined networking. IEEE Communications Magazine **51**(2), 136–141 (2013)
8. Ortiz Jr., S.: Software-defined networking: On the verge of a breakthrough? IEEE Computer **46**(7), 10–12 (2013)
9. Ng, E.: Maestro: A system for scalable openflow control, Technical Report of Rice University (2011)
10. Curtis, A.R., Mogul, J.C., Tourrilhes, J., et al.: Devoflow: scaling flow management for high-performance networks. In: ACM SIGCOMM (2011)
11. Yu, M., Wundsam, A., Raju, M.: Nosix: A lightweight portability layer for the sdn operating system. ACM Computer Communication Review (2014)