Enforcing Timely Network Policies Installation in OpenFlow-based Software Defined Networks

Ting Wang*, Mounir Hamdi†, Jie Chen‡
*Hong Kong University of Science and Technology, †‡Hamad Bin Khalifa University, Qatar
*twangah@connect.ust.hk, †mhamdi@qf.org.qa, ‡jchen@hbku.edu.qa

***Abstract*—As an efficient network innovation enabler, software defined network is designed to address the networking needs that are poorly addressed by existing networks, and makes it easier to create and introduce new abstractions in networking, simplifying network management and facilitating network evolution. The OpenFlow API enables secure communication between controllers and switches, and standardizes the communications. However, there exist critical issues during the procedure of distributing network policies among the switches (especially for in-band communication schemes), which impose various implicit negative impacts on network reliability and efficiency, such as additional computation overhead on both controllers and switches, communication overhead on secure channel, and waste of storage resources on switches. Based on these observations, this paper proposes four practical solutions to deal with these issues. The evaluation results reveal that the proposed solutions reduce the network latency by 50%, improve the goodput by 10-15%, and decrease the hardware cost by 25% at most, which convinces the effectiveness of proposed solutions.**

I. Introduction

As today's emerging Internet applications and services become more and more complex and demanding, the Internet has become increasingly difficult to develop both in terms of its physical infrastructure as well as its protocols and performance. As a result, software defined networks (SDN) has been proposed through the efforts of the Open Networking Foundation (ONF) [1] to solve these Internet ossifications and facilitates new methods for managing and configuring networks, and has been emerged as an active research field in both industry and academia, driving a revolutionary industry transition towards a programmable network [2][3]. In SDN the network intelligence is extracted from the underlying switch hardware and composed to be a separate control plane decoupling from the data plane, which simplifies network provisioning, optimizes performance and achieves granular policy management. The controller maintains the states of all switches, and provides centralized control and global knowledge of the network. Packet forwarding and routing control functions are done outside the switches themselves, which enables advancements in routing control, network virtualization, and visualization. The centralized controller computes network policies for new traffics, and installs forwarding rules on the switches along the routing path. With the benefit of these centralized efforts, the network can achieve direct and dynamic control over traffics and being globally optimized. Ultimately, software defined networks could use software to virtualize networks just as hypervisors enable server virtualization.

As the most typical southbound API of software defined networks, OpenFlow [4] standard enables the communication between control plane and data plane via a secure channel. OpenFlow switches are equipped with numbers of flow tables and group tables to pipeline processing incoming flows. When a new packet comes, firstly it looks up the flow table to execute field matching. If it succeeds in matching any forwarding rules, then it is pushed into openflow pipeline for further processing according to *Instructions* contained in the flow entry. When the *Instruction* set of a flow entry does not contain a *Goto-Table* instruction, pipeline processing terminates and actions in the *Action Set* associated with the packet are executed. If the packet fails in matching any flow entry, then it will execute the *Instructions* associated with the table-miss flow entry. According to the newest version of OpenFlow specification [4], the table-miss does not exist by default in a flow table, the controller may add it or remove it at any time. The packets are dropped by default in case of missing table-miss flow entries. If the table-miss flow entry exists, the switch datapath sends a asynchronous message OFPT_PACKET_IN to controller using controller reserved port. The controller computes its routing decisions based on the current network state and then distributes the forwarding policies to be installed on the switches along the routing path. The problem is how to guarantee these forwarding rules timely to be installed on the following switches along the path before packets arrive, especially when applying in-band communications. To the best of our knowledge, currently there are not any existing efficient mechanisms to guarantee timely installations of network policies on the fly, resulting in various critical issues which are discussed in Section III-B. In response to these issues, in this paper we proposed several practical solutions targeting at enforcing timeliness of network policies in OpenFlow-based software defined networks.

The primary contributions of this research can be summarized as follows:

1) Addressed the critical issues of delayed network policy installations in OpenFlow networks. To the best of our knowledge, we are the first to address such issues.
2) Proposed four efficient solutions to these issues, avoiding direct packet drops and eliminating negative effects.
3) Conducted extensive simulations to evaluate the feasibility and efficiency of these proposed solutions.

The remainder of this paper is organized as follows. The background and related work are briefly reviewed in Section II, and the motivation is described in Section III. Afterwards, the proposed solutions are presented with analysis in Section IV. Section V demonstrates the evaluation results. Finally, this paper concludes in Section VI.

II. Background and Related Work

OpenFlow [5] is an open source protocol pioneered by Stanford University faculty with funding from the Clean Slate Lab hosted at Stanford. In a standard network, both the packets being routed (the data path) and the routing decisions (the

978-1-4673-8999-0/17/$31.00 ©2017 IEEE

control path) are handled by the router or switch. With OpenFlow protocol, these functions are separated [6]. The router or switch handles the data path, and a separate, programmable controller handles the control path. The switch and controller communicate via the OpenFlow standard. This additional layer of abstraction introduces greater flexibility into the network and simplifies management, provisioning, and configuration of network devices.

As shown in Figure 1, the main components of an OpenFlow logical switch includes one or more *flow tables* and a *group table*, which perform packet lookups and forwarding, and one or more *OpenFlow channels* to an external controller [4]. The switch communicates with controller and the controller manages the switch via the OpenFlow protocol. The OpenFlow protocol specification standardizes three types of message types, which are *controller-to-switch* messages, *asynchronous* messages, and *symmetric* messages. The *controller-to-switch* messages are initiated by controller to manage, identify and read the switch states, examples like handshake message (OFPT_FEATURES_REQUEST), switch configuration message(OFPT_SET_CONFIG), flow table configuration message (OFPT_FLOW_MOD), modify state message (e.g. messages modifying flow table, flow entry, group entry, port, meter), multipart messages, packet-out messages (OFPT_PACKET_OUT), barrier message (OFPT_BARRIER_REQUEST), role quest message (OFPT_ROLE_REQUEST), bundle message, and set asynchronous configuration message. The *asynchronous* messages are generated by switch and sent to controllers in case of OpenFlow state changes so that the controller view of the switch is consistent with its actual state, such as packet-in message (OFPT_PACKET_IN), flow removed message (OFPT_FLOW_REMOVED), port status message (OFPT_PORT_STATUS), controller status message (OFPT_ROLE_STATUS), table status message (OFPT_TABLE_STATUS), request forward message (OFPT_REQUESTFORWARD), and controller status message (OFPT_CONTROLLER_STATUS). The *symmetric* messages are sent without solicitation, in either direction, including hello, echo, error, and experimenter messages.

The OpenFlow specification [4] specifies six reasons triggering the packet-in message that this packet needs being sent to the controller. The six reasons are given as OFPR_TABLE_MISS (no matching flow (table-miss flow entry)), OFPR_APPLY_ACTION (output to controller in apply-actions), OFPR_INVALID_TTL (packet has invalid TTL), OFPR_ACTION_SET (output to controller in action set), OFPR_GROUP (output to controller in group bucket), OFPR_PACKET_OUT (output to controller in packet-out). The problem this paper tries to resolve is how to enforce the network policies, computed by controller after receiving OPFT_PACKET_IN message with the reason OFPR_TABLE_MISS (no matching flow), to be timely installed on all switches along the routing path before the packets arrive. Unfortunately, up to now we fail to find any existing related work to solve this problem.

III. Motivation

A. Scenario and Problem Statement

As aforementioned, the forwarding policies cannot be guaranteed to be installed in time on switches (especially when applying in-band communication mechanisms between controller and switch) along the routing path before packet arrival, which leads to packets drops or repeatedly sending to controller for policy recomputing. Figure 2 depicts a typical scenario suffering this issue. In this scenario, host A is trying to send traffic to host B. As a new flow, when its first packet arrives at switch 1, after matching table-miss flow entry the switch will send a asynchronous message OFPT_PACKET_IN (or the whole packet) with the packet buffer ID referencing a packet stored in the switch to the controller. The controller then computes its forwarding rules heading to host B, and sends the forwarding rules to all switches along the path. After switch 1 receiving the OFPT_PACKET_OUT message, the packets will be pipeline processed according to the newly installed forwarding rules, and forwarded to the designated output port heading to the next hop switch after pipleline processing. The problem happens when the packet arrives at the next hop switch 2 but the forwarding policy has not been successfully installed yet due to transmission delay between controller and switch or other reasons. In this case, the packet will match table-miss flow entry again (or directly be dropped if there is no table-miss flow entry), and will be again forwarded to the controller to recompute its forwarding policies and re-install policies on switches once again. This situation may occur on each hop along the routing path as long as the network policy installation is delayed, and significantly jeopardizes the network efficiency and robustness.

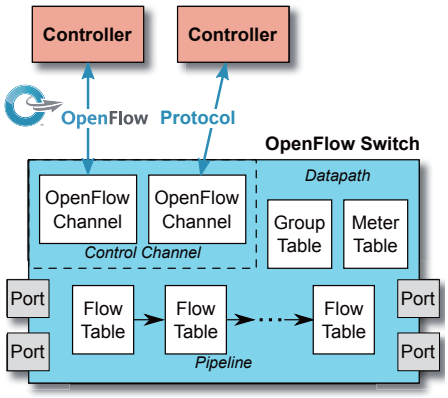

Fig. 1. Main components of an OpenFlow switch [4].

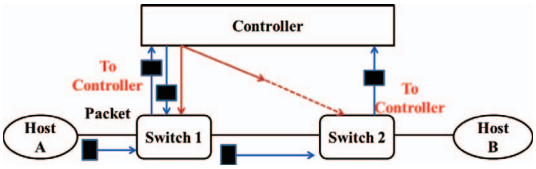

Fig. 2. Scenario of delayed policy installation.

B. Negative Impacts

Specifically, the issue of late installation of network policies incurs severe harms to both controllers and switches accompanying with numbers of negative impacts, some of which are summarized as below.

1) When packets arrive at next hop switches, table-miss flow entry matching re-invokes "send to controller" action to send OFPT_PACKET_IN message again to controller requesting for forwarding rules for current

"new" flow, giving rise to extra computation overhead and buffer wastes.

2) The controller needs to recompute forwarding rules and re-send them to the switches along the path again, which increases computation overhead on controllers and imposes additional communication overhead (OFPT_PACKET_IN messages sent to controller, and recomputed forwarding rules sent to switches) on the secure channel between controller and switch.
3) The repeatedly re-installed flow entries consume amount of precious hardware (e.g. TCAM) resources, which leads to a great waste of constrained storage space.
4) The overlap checking and merging of duplicate flow entries induces much computation overhead on switches.
5) The recomputed forwarding rules may not be consistent with the previous rules in case of network state changes, as the controller dynamically computes and optimizes routing decision according to current network state. This causes that packets of the same flow will traverse different paths, resulting in packet out-of-order issue.
6) The network latency would be increased as well, including additional round trip time between switch and controller, additional computation time on controller, and extra processing time on switch, etc.

Each of the above defects could induce serious harms to the network reliability and efficiency. Motivated by these observations, this paper proposes several efficient approaches aiming to solve these issues by enforcing timely network policies installation in OpenFlow networks.

IV. Solutions Design

Intuitively, the issue of delayed policy installation can be mitigated from two aspects. The first approach is to guarantee the timeliness of policy installation, which should ensure the policies arrive earlier than packets. This requires to expedite the arrival of policies, or to delay the packet's arrival time, however, both of which are not suitable where the former way is impractical and difficult to be implemented while the latter one will bring in unpredictable network delay. The second compromising approach is to mitigate the impact caused by the late installation of policies, for example, to neglect the repeated OFPT_PACKET_IN requests at the controller by checking the recent cached requests, or to filter the unnecessary send-to-controller requests at the switch side. This kind of approach can mitigate the negative impacts to some extent, but cannot totally eliminate them.

Based on these observations, in this section, we proposed four practical solutions from different aspects. The evaluation of these solutions is conducted in Section V.

A. Solution 1: Domain Construction and Roles Assignment

In this scheme, the network is divided into a number of domains which are controlled by different controllers. A network domain is a sub-network consisting of a set of OpenFlow switches under the control of one or multiple controllers. For the case of multiple controllers within the same domain, the controller role (*equal, master, slave*) assignment and *master* controller election process are same as specified in [4]. Accordingly, the switches are also assigned with different roles, namely *edge switch* and *inner switch*, taking different actions when matching table-miss flow entry. The definitions of edge switch and inner switch are given as below.

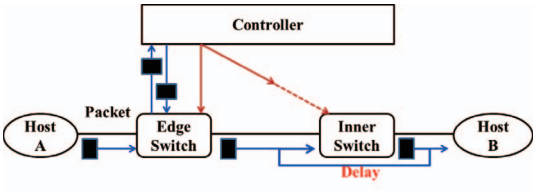

Fig. 3. Example of solution 1.

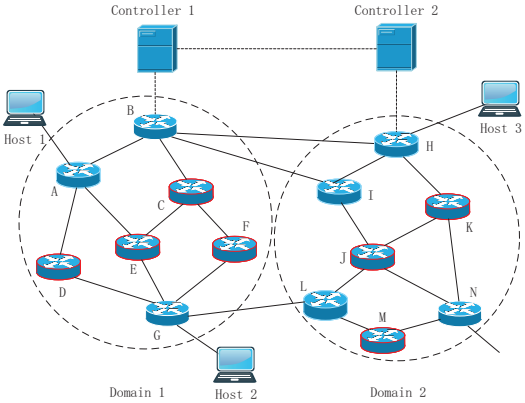

Fig. 4. Example of network domain construction and switch role assignment.

- **Edge Switch**: An edge switch is a switch that provides an entry point into the network domain. If the ingress port (OXM_OF_IN_PORT) of incoming packets connects to an end host or a switch in a different network domain, then this switch must be treated as an edge switch.
- **Inner Switch**: An inner switch is a switch (no connected hosts) whose neighbour switches are all located in the same network domain under the same controller(s). If incoming packets come from one port that connects to an inner switch, then this switch should also be treated as an inner switch when packet processing.

Figure 4 illustrates an example, as shown, the network is partitioned into two domains, which are controlled by Controller 1 and Controller 2, respectively. According to the switch role assignment mechanism, the inner switches include switch C, D, E, F, J, K and M. The switches A, B, G, H, I, L and N could be edge switches or inner switches depending on where its packets come from.

After switch role assignment, the table-miss matching actions of switches with different roles should be specified, accordingly. When looking up flow tables, the port is included in the matching field, especially for matching table-miss flow entry. For an edge switch, if its ingress port connects to an host or a switch in a different network domain, then the instruction of table-miss flow entry for new incoming packets is "send to controller" triggering packet-in message. Otherwise, for an inner switch, or the ingress port of an edge switch connects to inner switches within the same domain, then the table-miss action is set to be "look up flow tables after a certain delay time" by keeping the packets in original "PACKET_IN" buffer for a certain time. When delaying time is up, push the packet into the pipeline again to check if the forwarding rule has been installed. The packet will be dropped if it still does not match any flow entries after delaying. The delay time can be set as the average communication time between controller and this switch (single trip time, i.e. 0.5*RTT). Alternatively,

the delay time can be set to be a very short time and if it fails in a certain times of tries (e.g. when forwarding rules are lost during transmission), then drop the packet. If the estimated single trip time between the switch and controller is denoted as T, and the number of trying times is limited as N, then the interval waiting time between two consequent tries could be T/N. This fine-grained time control mechanism can greatly help reduce the waiting time to some extent. Figure 3 illustrates an example of this approach.

This solution is designed based on the observation that the new traffics always come from the edge of network, i.e. happen at edge switches. Therefore, the inner switches should avoid invoking packet-in messages. By doing this, this solution strictly prevents the inner switches from "send-to-controller" actions in case of table-miss matching. Besides, it is easy to be implemented and configured without changing any data structures or protocols. Although this solution needs to buffer the packets for a certain time (less than 0.5*RTT), it still saves more time than the traditional approach (sending packet-in message to controller), and the fine-grained time control mechanism can further help save more time.

B. Solution 2: Resort to Barrier Message

As specified in OpenFlow specification [4], when a controller wants to control message processing and ordering, it may use an OFPT_BARRIER message, which is composed of a small sized OpenFlow header (8 bytes) without body. OpenFlow specification specifies three functions of barrier message. *The first function of barrier message is to ensure message ordering, so that message dependencies are not violated. The second function of barrier message is for the controller to receive notifications for completed operations. The third function of barrier is to ensure that the state sent by the controller is committed to the datapath* [4].

Therefore, the barrier message can be effectively leveraged to ensure the network policies installation. The general working procedure is designed as below.

1) New packets arrive at the first switch. The switch sends a Packet-In message to controller when table-miss matching.
2) The controller computes forwarding rules based on current network states and optimization mechanisms.
3) Controller sends OFPT_FLOW_MOD messages to switches to install forwarding rules (adding new flow entries), and uses OFPT_BARRIER_REQUEST message to ensure the completion of policy installation. Accordingly, all involved switches must send a OFPT_BARRIER_REPLY message to controller upon installation completion.
4) Controller waits responses from switches.
5) After collecting OFPT_BARRIER_REPLY messages from all switches, the controller then sends an OFPT_PACKET_OUT message (containing a list of actions to be applied in the order they are specified) to the first switch.
6) The packets are then pipelined processed following traditional procedures.

Figure 5 gives a simple example to illustrate the working procedure, where procedure ① indicates switch sending packet-in message to controller, procedure ②③ denote sending/replying barrier messages, and procedure ④ implies

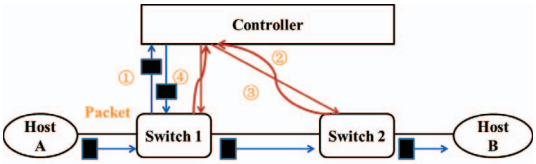

Fig. 5. Example of solution 2.

controller sending packet-out message to the first switch. With the benefit of barrier message, this solution can completely guarantee the policies are installed before packets arrival at next hop switches. Admittedly, it has a major concern that it may result in an extra latency, where the maximum latency will be the response time of barrier message from the latest switch ($Max(0.5 * RTT_i)$). In order to mitigate this issue, a fine-grained time control mechanism can be applied. It is observed that the time difference between packet arrival time and policy arrival & installation completion time is not very large, and the most affected switches are the first few ones on the routing path while the last few switches are less affected. This observation is convinced by extensive simulation results. Consequently, it can be derived that it may be not necessary to collect all responses of barrier messages, instead receiving the responses from only the first few next-hop switches on the path is enough. In the simulations of Section V, the barrier message is only sending to the first next-hop switch.

C. Solution 3: Proactive Strategy

The main reason causing the network policies late installation is owing to that the forwarding rules sending to different switches along the routing path may traverse different paths using different time. Thus, the problem happens when the packets reach the next hop switch but the OFPT_FLOW_MOD message has not arrived yet. From this perspective, the issue can be solved by restricting the forwarding rules to transmit on the same path. It works as below.

1) New packets arrive at a switch. After table-miss matching, switch sends an OFPT_PACKET_IN message to controller requesting forwarding rules.
2) Controller computes the forwarding decisions. Traditionally, at this time controller will send OFPT_FLOW_MOD messages to all involved switches traversing different paths based on network states and path distance. In this solution, all the OFPT_FLOW_MOD messages are restricted to be forwarded along the same path. Firstly, the forwarding rules are sent to the first switch which initiates the packet-in message. Then, the packets of flow are pipelined processed on this switch, and the other forwarding rules for next switches are forwarded along the same computed routing path as the packets.

Take figure 4 as an example, assume host 1 is sending traffic to host 2, when packets arrive at switch A and matches table-miss flow entry, it will send packet-in message to controller 1 requesting for forwarding decisions. Suppose the controller computed its best routing path as switch A–>E–>G. Then, the controller needs to send the forwarding rules to be installed on all involved switches, i.e. switch A, E, and G. Traditionally, the forwarding rules sent to switch A, E, G may transmit along different paths according to network states,

such as controller1–>B–>A (for switch A), controller1–>B–>C–>E (for switch E), and controller1–>B–>C–>F–>G (for switch G). Comparatively, in this approach, the forwarding rules are all restricted to be forwarded through the same routing path, e.g. controller1–>B–>A–>E–>G, where the packets will traverse the same path (A–>E–>G) as forwarding rules. The policy messages will be processed with the highest priority at each switch.

Consequently, the issue of network policy late installation can be thoroughly solved by strictly following this strategy. The only problem resides in potential poor load balancing and increasing the traffic burden on the same routing path. However, the load balancing can be taken into consideration when controller computing the optimized forwarding decisions as it has a global knowledge about the network states and can make a better balance among all traffics. Moreover, this approach avoids incurring any computation overhead or communication overhead, and no additional caused latency as well. Furthermore, the implementation of this approach is simple and practical without any modifications or changes to switches and OpenFlow protocols.

D. Solution 4: Request Caching Mechanism

This approach works in the controller side. When packets of a new flow arrive at the entry switch and match table-miss flow entry, then the switch will request controller for forwarding rules. Notably, the controller will cache a certain number of most recent OFPT_PACKET_IN requests. As specified in OpenFlow specification, when a flow entry is inserted in a flow table, its *idle_timeout* and *hard_timeout* fields are set with the values from the messages. If the *idle_timeout* is set, the flow entry must expire after *idle_timeout* seconds with no received traffic. The *hard_timeout* requires that the entry must expire in *hard_timeout* seconds regardless of whether or not packets are hitting the entry. Therefore, one unexpected scenario may happen that the packet-in message may be rejected or discarded by the controller if it is found in the caching records when the packet-in message is invoked by table-miss matching because of flow entry expiration other than late installation. In order to prevent this happening, the maximum standing time ($T_{caching}$) of each cached request in controller should be less than the minimum timeout of each flow entry in all involved switches, i.e. $T_{caching} < Min\{idle\_timeout_i, hard\_timeout_i\}$ for each involved switch i. In the simulations, the $T_{caching}$ is set to be a random number ranging between the average single trip time $0.5{*}RTT_i$ and $Min\{idle\_timeout_i, hard\_timeout_i\}$. If the packets happen to arrive earlier than policy installation at next hop switch, then the packet will fail in field matching and be sent to controller. The controller checks the recent cached requests and discards this request if its according cached record is found, without re-computing and re-distributing its forwarding rules.

This approach provides an efficient way to the mitigate the issue of network policy late installation, and solves the issues of (2), (3), (4), (5), (6) listed in subsection III-B. Besides, this approach is easy to be implemented on controllers, and without any changes to switches. However, the duplicate packet-in messages still exist in this approach.

E. Comparisons

Table I presents the comparison results of four solutions from various aspects with respect to the solved issues, complexity, efficiency, and practicality. As analyzed, solution 1 and 3 succeed in eliminating all the negative impacts listed in subsection III-B and achieve higher efficiency with better practicality. Comparatively, although solution 2 and 4 also greatly mitigate the severe issue of network policy late installation, some of negative impacts are only slightly improved other than totally eliminated.

V. Evaluation

This section presents the performance evaluation of the proposed approaches from different aspects, including latency, goodput, and cost.

A. Simulation Environment

The simulations are implemented on top of POX controller [7] and Mininet2.1 [8] in Python. The network topology is generated using Erdős-Rényi (ER) random graph [9] consisting of 30 switches. The flow distribution (the way choosing source and destination) follows uniformly random mode, and the Weibull distribution is applied to determine the packet inter-arrival time. The unidirectional link bandwidth is 1 Gbps and each link is capable of bidirectional communications. The propagation delay of a link is set to be 5 μs by default. The TTL is set to be 128. The default forwarding time (pipeline processing time at each switch) is 10 μs, excluding the queuing time. The maximum segment size (MSS) is 1460 bytes (MTU=1500 bytes). The parameter N in the fine-grained time control mechanism of solution 1 is set to be 10.

B. Compared Baseline

In order to better demonstrate the existing issues and compare with the proposed solutions, the compared baseline solution works as follows. The controller computes different routing paths for the forwarding rules to be installed on different switches according to the network state at that time, and restricts the forwarding rule for the most-adjacent next-hop switch follows a longer path with higher transmission time than that for the first switch. This intends to explicitly create a scenario where the network policy installation will be delayed so as to simplify the comparisons.

C. Simulation Results

1) Latency: The network latency performance is evaluated in terms of packet's round-trip time (RTT). In the simulations, one host sends ICMP packets to a set of different destinations (ping host). Figure 6 exhibits the round trip time of these ICMP packets using different solutions. It can be seen that all solutions 1-4 significantly reduce the overall network latency, where solution 1 and 3 achieve minimum latency with approximately 50% latency reduction while solution 2 and 4 reduce around 30% network latency compared with the baseline solution.

2) Goodput: The goodput evaluates the application level throughput which only refers to the transferred "useful" data per time unit (application data rate), excluding protocol overhead bits as well as retransmitted data packets. Figure 7 illustrates the simulation results of goodput, which is evaluated at the receiver side, between a certain host pair $< src, dest >$ using different solutions. The simulation result reveals that solution 1 & 3 and solution 2 & 4 improve goodput by approximately 10%-15% and 5%-10% at most, respectively, comparing with the baseline solution. Specially, solution 1 & 3 achieve better performance than solution 2 & 4.

TABLE I
THE COMPARISON BETWEEN DIFFERENT SOLUTIONS

Solutions	Key Characteristics	Solved Issues	Complexity of Implementation	Efficiency	Practicality
Solution 1	Domain construction	(1),(2),(3),(4),(5),(6)	Low	★★★★★	★★★★★
Solution 2	With barrier message	(1),(2),(3),(4),(5)	Low	★★★★	★★★★
Solution 3	Proactive strategy	(1),(2),(3),(4),(5),(6)	Low	★★★★★	★★★★★
Solution 4	Request Caching	(2),(3),(4),(5),(6)	Low	★★★★	★★★★

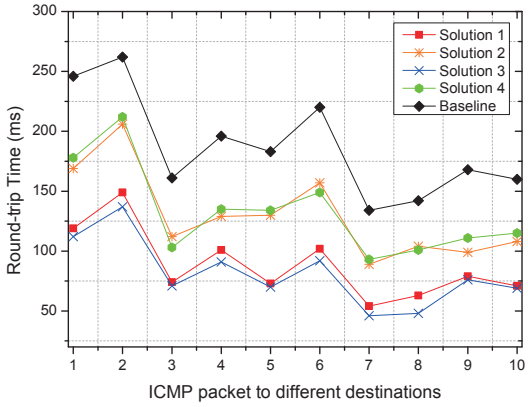

Fig. 6. The RTT of ICMP packets to different destinations.

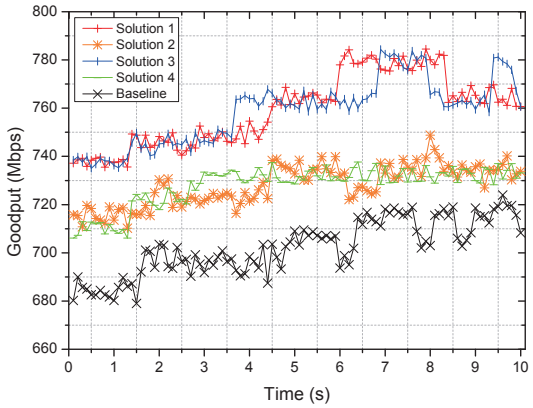

Fig. 7. The performance of goodput.

3) Cost: The cost refers to the computation overhead of controllers and switches, the storage/hardware (e.g. TCAM) cost of switches, and the communication cost (e.g. bandwidth) between the controller and switch. For the sake of simplicity, we use the number of flow entries on each switch (without merging duplicated inserted flow entries) as an evaluation metric to test the hardware cost of each switch. From another perspective, the number of installed flow entries, from a side, also reflects the required communication overhead and the computation overhead to process them. In simulations, firstly all-to-all traffic pattern is applied to transmit flows across the network. After the completion of all flows, the number of flow entries on each switch is polled. Then, the average number of flow entries on each switch is calculated by repeating this procedure five times, and the obtained results are illustrated in Figure 8. The evaluation results disclose that more flow entries are installed on each switch using baseline approach, which implies numerous duplicate flow entries are computed and installed with additional computation overhead on both controllers and switches, as well as the communication overhead to transmit them. Comparatively, solution 1-4, which

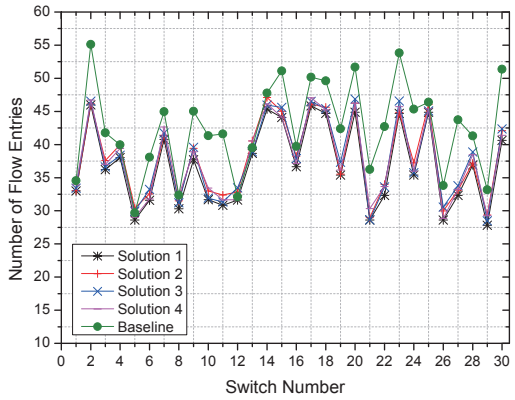

Fig. 8. The average number of flow entries on each switch.

essentially avoid installing duplicate flow entries, succeed in reducing the number of flow entries on each switch and decreasing the cost from various aspects.

VI. CONCLUSION

As a de facto standard communication interface for software defined networks, OpenFlow provides secure and efficient communications between the control plane and data plane. However, the issue of network policy late installation existed in OpenFlow networks brings numerous negative impacts on network reliability and efficiency. In response to these intractable issues, this paper proposes four efficient and practical solutions. The evaluation results further prove the effectiveness and good performance of the proposed approaches. To the best of our knowledge, this is the first work to address and solve such issues.

VII. ACKNOWLEDGEMENT

This publication was made possible, in part, by NPRP grant #NPRP 6-718-2-298 from the Qatar National Research Fund.

REFERENCES

[1] Open networking foundation. https://www.opennetworking.org.
[2] Bruno AA Nunes, Manoel Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *Communications Surveys & Tutorials, IEEE*, 16(3):1617–1634, 2014.
[3] Yosr Jarraya, Taous Madi, and Mourad Debbabi. A survey and a layered taxonomy of software-defined networking. *Communications Surveys & Tutorials, IEEE*, 16(4):1955–1980, 2014.
[4] Openflow swtich specification version 1.5.0 (protocol version 0x06). OPEN NETWORKING FOUNDATION, 2014.
[5] N. McKeown, T. Anderson, H. Balakrishna, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM*, 38(2):69–74, 2008.
[6] Fei Hu, Qi Hao, and Ke Bao. A survey on software-defined network and openflow: from concept to implementatioin. *Communications Surveys & Tutorials, IEEE*, 16(4):2181–2206, 2014.
[7] Pox controller. https://github.com/noxrepo/pox.
[8] Mininet. https://github.com/mininet.
[9] Béla Bollobás. *Random graphs*. Springer, 1998.