Correctness Arguments for an SDN MAC Learning Algorithm

Liang Hao, Xin Sun, Lan Lin
Department of Computer Science
Ball State University
Muncie, IN 47306, USA
{lhao, xsun6, llin4}@bsu.edu

Abstract

The emerging software-defined networking (SDN) paradigm is a radical departure from the traditional networking architecture as it decouples the control plane and the data plane and also centralizes the control plane. While SDN has proven to provide an unprecedented opportunity for creating advanced control functions that are capable of optimizing global properties of a network, recent works including our own have shown that the implementation of such control functions is a software engineering challenge on its own due to the unique architecture of SDN. In our recent work [7], we have taken the first step towards tackling this challenge by presenting a preliminary case study of implementing the classic Media Access Control (MAC) learning algorithm for SDN. A major discovery we made is that, for a proper implementation of the algorithm, the algorithm itself must be modified to account for the architectural differences of SDN. This paper builds upon and advances our prior work, by presenting a detailed description and analysis of the SDN version of the MAC learning algorithm, to justify the modifications as necessary for a correct and efficient implementation. As such this paper sheds light on the interplay between the algorithm design and software implementation of control functions for SDN, which we believe is an important contribution to both the software engineering community and the networking community.

1 Introduction

Software-defined networking (SDN) is a new paradigm in the networking landscape that transforms the way networks are constructed and managed. A defining architectural feature of SDN is the centralized control plane, which is a radical departure from the traditional networking architecture where the control plane is fully decentralized. The control plane serves as the brain of a network: it controls how packets are forwarded by producing the necessary forwarding tables. The data plane, in contrast, handles the packet streaming by looking up those tables.

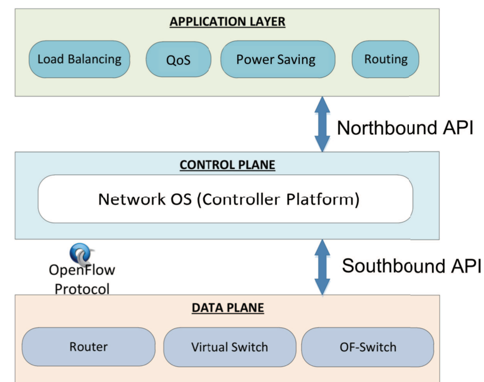

Figure 1. The decoupling of the control and the data planes in the SDN architecture

In the traditional networking architecture, both the control and the data planes are implemented on every networking device (e.g., routers and switches), and hence they are fully decentralized and tightly coupled. In contrast, the SDN control plane is implemented by a logically centralized software platform, called the *network OS* or the *controller*, with all the application programs (called *SDN apps*) running on top of it. The controller, together with all the SDN apps, runs on a PC server that connects to the networking devices (called *SDN switches*). The SDN data plane is implemented by the packet forwarding hardware on all the SDN switches. A communication protocol, called *OpenFlow* [16], is used for the controller to communicate with the SDN switches and vice versa. The SDN architecture is illustrated in Figure 1.

The unique architecture of SDN provides an unprecedented opportunity for creating advanced control functions, but at the same time poses a significant challenge to the implementation of such functions as SDN apps. On the one hand, the centralized control plane oversees the entire network and offers a global view for creating control functions

DOI reference number: 10.18293/SEKE2021-184

capable of global optimization. On the other hand, the decoupling results in the control plane's limited visibility into the data plane. In particular, only a small subset of packets being streamed in the data plane are copied to the controller, so as not to overload the controller or the network. This poses challenges to the implementation of SDN apps.

SDN research has largely focused on the algorithm design of control functions. However, relatively little attention has been given to the software engineering aspect, which we believe is an important research problem worth addressing on its own. As a first step, in our recent paper [7] we presented a case study of implementing the classic Media Access Control (MAC) learning algorithm (an essential link-layer forwarding algorithm implemented by every traditional Ethernet switch) in the SDN architecture. A major discovery we made in [7] is that an efficient implementation requires modifications to the algorithm itself due to the architectural differences of SDN. We presented in [7] such an implementation following rigorous software specification and design methodologies. However, a detailed description of the modified algorithm itself or an analysis of the algorithm to justify the modifications was not presented in [7], due to the fact that the focus of that paper was on applying rigorous software specification and modular design to derive an implementation through stepwise refinement.

Building upon and advancing our prior work, the primary contributions of this paper include a detailed description of the SDN version of the MAC learning algorithm, and a detailed analysis that shows why the modifications are necessary for the SDN architecture. We hope this paper will shed light on the interplay between the algorithm design and the software implementation of SDN control functions.

The rest of the paper is organized as follows. Section 2 presents a description of the classic MAC learning algorithm. Section 3 explains why a direct and straightforward migration of the classic algorithm to SDN is impractical, presents the SDN version of the algorithm, and argues why the modifications are necessary for achieving a correct and efficient implementation. Section 4 discusses related work and Section 5 offers concluding remarks.

2 The Classic MAC Learning Algorithm in Traditional Networks

MAC learning is the classic link-layer algorithm that enables an Ethernet switch to perform two essential functions: (i) constructing and updating the switch table (a control plane function), and (ii) using the table to forward packets toward their destinations (a data plane function) [8]. In the traditional networking architecture, this algorithm runs on every Ethernet switch.

The switch table produced by the algorithm is stored in the switch memory. An entry in this table contains the MAC address of some device in the network, the switch port believed to lead toward that device, and a timer. The MAC address is used as the key for indexing the table (MAC addresses are all unique), and the timer is used to remove the entry in the future in case it becomes stale.

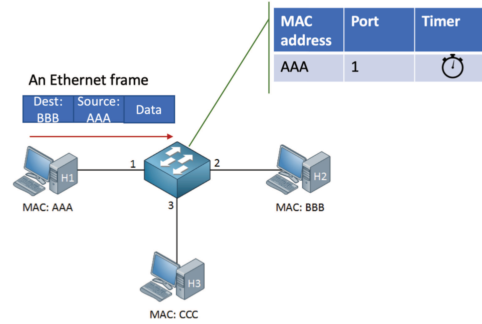

Figure 2. The classic MAC learning algorithm working in a traditional network

On each Ethernet switch, whenever a packet is received on a port p, the algorithm performs the following two steps (illustrated in Figure 2):

Step 1: the algorithm extracts the *source* MAC address s from the packet header. If s is not in the switch table, it creates a new table entry (s, p, t) where p is the incoming port (which must lead toward s) and t is a timer set to expire after some period of time. If such an entry already exists, the timer will be reset. If there exists an entry with the same MAC address s but with a different port, the port will be updated to p and the timer be reset.

Step 2: the algorithm uses the *destination* MAC address d in the packet header to look up the switch table. If an entry with d exists, the packet will be sent out the associated port. If such an entry does not exist, the algorithm will *flood* the packet out all active ports except the incoming port p.

In addition, the algorithm will delete a table entry when the associated timer expires. This is to get rid of any potentially stale entry due to topology changes, e.g., hosts being removed or relocated.

We note that the proper functioning of Step 1, specifically the resetting of timers, relies on the fact that the algorithm sees *every* incoming packet. As we will explain next, this is not the case in the SDN architecture.

3 The SDN Version of the MAC Learning Algorithm and Correctness Arguments

We first describe the key difference of the SDN architecture and explain why a direct and straightforward migration of the MAC learning algorithm to SDN will not work. We

then describe several modifications to the algorithm and argue why those modifications are necessary for achieving a correct and efficient implementation. Lastly we use an example to illustrate the operation of the new algorithm.

3.1 A Direct Migration Is Inefficient

The control plane and the data plane are completely decoupled in the SDN architecture. The control plane is implemented by the software controller running on a centralized server; and the data plane is implemented by all the decentralized SDN switches. Implementation of the MAC learning algorithm will involve the controller *and* all the SDN switches, as Step 1 of the algorithm is a control plane function and Step 2 a data plane function. The two parts of the algorithm will need to interact via the *OpenFlow* communication protocol [16]. A direct migration of the classic MAC learning algorithm to SDN would work as follows.

On the controller: Step 1 of the algorithm would be implemented as an SDN app running on the controller. The app constructs and updates the switch table for every switch. For each switch s, the app collects *every* incoming packet together with the incoming port information. For each packet the app either creates a new switch table entry or updates an existing entry. It then sends the new or updated table entry to s in an OpenFlow protocol message. (The OpenFlow protocol messages are called "*FlowMods*.")

On every SDN switch: Whenever a switch receives a new packet, it sends the packet together with the incoming port information to the controller. The switch then receives a new or updated table entry from the controller and caches it locally. The switch uses the destination MAC address in the packet header to look up its locally cached table, and either forwards the packet to the associated port if an entry is found, or floods the packet on a table miss.

In addition, the SDN app is responsible for creating and resetting all the timers. When a timer expires, the SDN app deletes the associated table entry and informs the switch to delete the same entry from its locally-cached table.

This direct and straightforward migration of the MAC learning algorithm to SDN unfortunately will not work for two reasons. First, requiring every switch to send *every* packet to the controller creates substantial traffic overhead; for every packet the controller will send a FlowMod message back to the switch, further increasing the overhead. Second, the controller will receive and process *every* packet; the amount of link bandwidth and CPU power required to implement such a controller is simply beyond practicality.

3.2 Necessary Algorithm Modifications

To address the above-mentioned efficiency issues, our prior work [7] made several important modifications to the MAC learning algorithm when implementing it for SDN.

First, the modified algorithm requires that, whenever an SDN switch receives a packet, it should *first* look up its locally-cached table[1] using the source and destination MAC addresses in the packet header and information regarding the incoming port. Note that each entry of the locally-cached table maps a three-tuple (source address, incoming port, destination address) to a destination port. If a table entry is found to match the three-tuple (source address, incoming port, destination address), the switch should forward the packet out the associated port *without sending the packet to the controller*. Only upon a table miss should the switch send the packet and information regarding the incoming port to the controller.

Second, the MAC learning app running on top of the controller will construct and update the forwarding tables for all the switches. Whenever the app receives a packet and incoming port information from a switch s, it either creates a new entry in the table of s if the source MAC address of the packet does not exist in the table, or updates the existing entry based on the new port information. It then uses the destination MAC address of the packet to look up the table of s. If an entry exists, it will send the entry to s in a FlowMod message; otherwise, it will instruct s to flood the packet, also in a FlowMod message.

While these changes dramatically reduce the number of packets sent to the controller, they create two new problems for the implementation, which we discuss below.

> *Implementation of the timers associated with table entries*: the controller no longer sees all the packets being forwarded on all the switches, and thus it is unable to properly reset the timers in the switch tables. This means that the timers cannot be maintained by the controller.
>
> *Synchronization of table entries between the controller and the switches*: even after the controller deletes an entry (h, p) in the forwarding table of switch s as the entry may become stale (e.g., the host with MAC address h may have relocated and no longer be reachable from switch s through its port p), s may still have a locally-cached table entry containing (h, p) as the *destination* address and *outgoing* port; this will cause s to continue to forward any packet destined to host h out of port p, resulting in permanent loss of those packets.

To solve both problems, several additional modifications to the MAC learning algorithm are necessary. First, whenever the controller sends a FlowMod message to a switch that contains a table entry, it also creates and sends a *reversed* FlowMod message, in which it flips source address with destination address, and flips incoming port with outgoing port. For instance, if a FlowMod message (to be sent

[1] The locally-cached switch table is also called the FlowMod table.

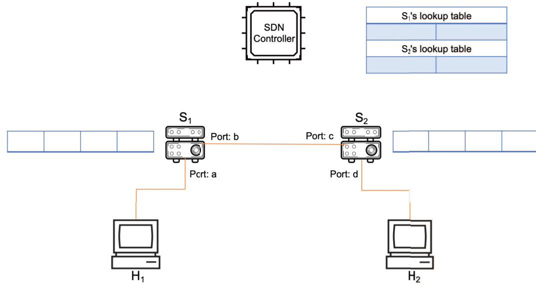

Figure 3. An illustrating case: Initial State

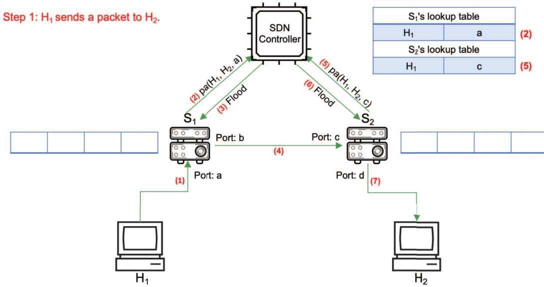

Figure 4. An illustrating case: Step 1

to switch s) with source address sh, incoming port sp, destination address h, and outgoing port p is created, the controller also creates a reversed FlowMod message (to be also sent to switch s) with source address h, incoming port p, destination address sh, and outgoing port sp. By the time they are created, it must be the case that (h, p) and (sh, sp) are both valid table entries, i.e., neither (h, p) nor (sh, sp) is a stale host-port pair.

Second, switches maintain the timers for all the locally-cached table entries, as switches see all the packets and can thus properly reset the timers. The controller on the other hand does not maintain any timer.

Third, whenever a locally-cached FlowMod table entry, with (sh, sp) as (source address, incoming port) and (h, p) as (destination address, outgoing port), gets deleted due to timer expiring, the switch sends a FlowRemoved message notifying the controller. The controller then deletes (sh, sp) from the lookup table, and sends a FlowMod message instructing the switch to delete the reversed entry with (h, p) as (source address, incoming port) and (sh, sp) as (destination address, outgoing port) from its locally-cached table. Therefore, in case any host-port pair (h, p) goes stale, i.e., host h is no longer reachable from port p, for any locally-cached table entry with (h, p) as (destination address, outgoing port), its reversed table entry, in which (h, p) is the (source address, incoming port) pair will eventually expire and get removed, leading to the former table entry with a stale (destination address, outgoing port) pair being removed as well.

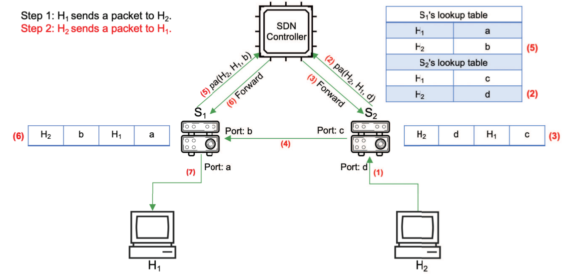

Figure 5. An illustrating case: Step 2, without reversed FlowMod messages

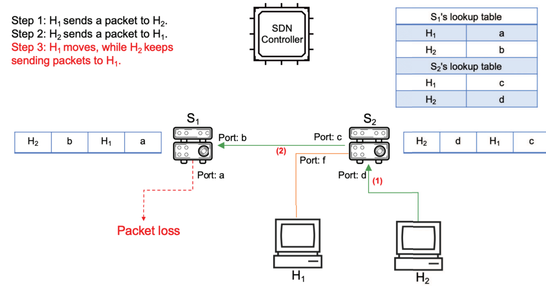

Figure 6. An illustrating case: Step 3, without reversed FlowMod messages

3.3 Illustrating the Algorithm

We illustrate this subtlety of the SDN MAC learning algorithm using an example. Assume there are two hosts in the network with two SDN switches (Figure 3). Host H_1 is connected to switch S_1 through port a. Host H_2 is connected to switch S_2 through port d. S_1 connects to S_2 through port b on S_1 and port c on S_2. The SDN controller maintains lookup tables for S_1 and S_2, which are initially both empty. The two switches each maintain their own FlowMod tables, which are also initially empty.

Assume H_1 first sends a packet to H_2 (Figure 4 sub-step (1)). S_1 consults the SDN controller for how to handle this packet (sub-step (2)). The controller adds (H_1, a) to S_1's lookup table (sub-step (2)). The controller instructs S_1 to flood the packet (sub-step (3)). The flooded packet arrives at S_2 though port c (sub-step (4)). S_2 consults the controller resulting in (H_1, c) being added to S_2's lookup table (sub-step (5)). The controller instructs S_2 to flood the packet (sub-step (6)). The flooded packet gets to H_2 (sub-step (7)).

Now assume H_2 sends a packet back to H_1 (Figure 5 sub-step (1)) and assume the migrated algorithm didn't enforce reversed FlowMod messages to handle the subtlety introduced by stale (host, port) entries. (H_2, d) is added to S_2's lookup table as S_2 consults the controller for how to handle this packet (sub-step (2)). Since the destination

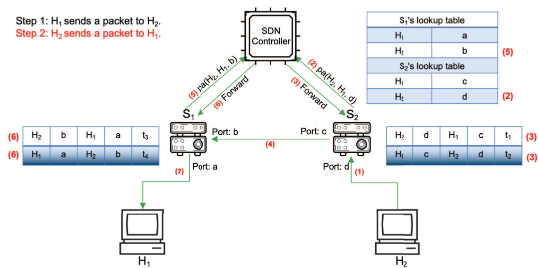

Figure 7. An illustrating case: Step 2, with reversed FlowMod messages

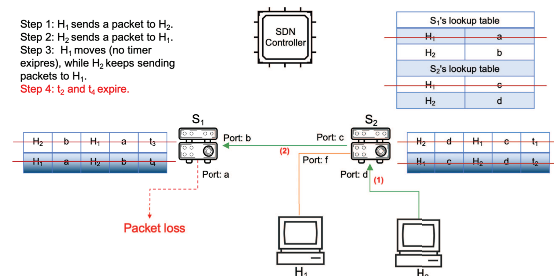

Figure 9. An illustrating case: Step 4, with reversed FlowMod messages

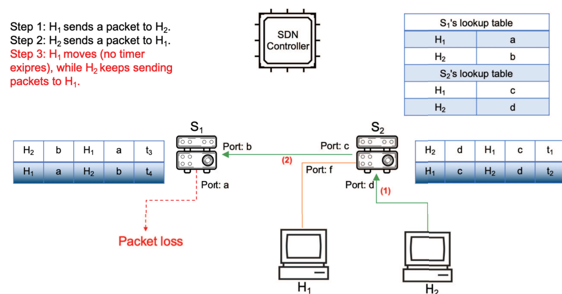

Figure 8. An illustrating case: Step 3, with reversed FlowMod messages

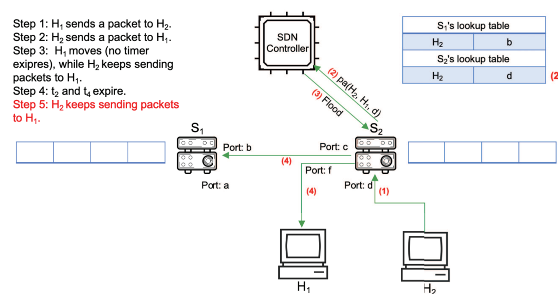

Figure 10. An illustrating case: Step 5, with reversed FlowMod messages

address H_1 is already in S_2's lookup table, the controller instructs S_2 to forward this packet to port c (sub-step (3)). S_2 caches this instruction in its FlowMod table (sub-step (3)). The forwarded packet gets to S_1 through port b (sub-step (4)). Now it's S_1's turn to consult the controller, which leads to another entry being added to S_1's lookup table (sub-step (5)). The controller instructs forwarding, and the instruction is cached in S_1's FlowMod table (sub-step (6)). Finally, the forwarded packet gets to H_1 (sub-step (7)).

Next we assume H_1 is disconnected from S_1 and connected to S_2 through port f, while H_2 continues sending more packets to H_1 (Figure 6). This topology change is not observed by either switch, as packets are being received through the same (source address, incoming port) pair, and forwarded to the same destination using the cached FlowMod table entries. Notice that all such packets from H_2 to H_1 will be lost with neither the switches nor the SDN controller being aware of the underlying topology change of the network and the resulting permanent packet loss.

With the correct algorithm that enforces reversed FlowMod messages, in Step 2 any forwarding instruction from the controller to the switch enforces a FlowMod table entry, together with its reversed FlowMod table entry, to be simultaneously cached on the switch (Figure 7 sub-steps (3) and (6) and shaded FlowMod table entries). In Step 3 after H_1 moves to the new location (Figure 8), although there will be a little packet loss at the beginning, soon this will be fixed. This is because in Step 4 (Figure 9) the FlowMod table entries on both switches for the reversed FlowMod messages (sent earlier by the controller) will expire (after H_1 changes the switch it is connected to). This results in two lookup table entries being deleted from the controller, and instructions from the controller to both switches to delete the remaining two FlowMod table entries. Finally in Step 5 (Figure 10) subsequent packets from H_2 to H_1 are consulted with the controller and flooded, reaching H_1 through port f. The first packet from H_1 through port f will be seen by the controller, which will add an updated lookup table entry for both switches with H_1 as the source address. Future packets from H_2 to H_1 will be forwarded correctly by the controller using the updated lookup table entries.

4 Related Work

Since the inception of SDN around a decade ago [9], the research in SDN has primarily focused on (i) creating novel control functions, such as load balancing [6], power saving [1], anomaly detection [14], etc.; these works mostly focused on designing the algorithms and protocols, and not on the implementation; (ii) the orchestration of multiple control functions running on the same software-defined network [2, 3, 10, 15, 17], and (iii) improving the performance,

reliability, and security of both SDN switches [5, 11, 12] and controllers [4, 13]. However, relatively little attention has been given to the software engineering aspects of SDN apps. Our recent work [7] took the first step in tackling the software engineering challenges of implementing control functions as SDN apps, through implementing the classic MAC learning algorithm for the SDN architecture. In contrast, this paper presents a detailed analysis of the SDN version of the algorithm that looks deeper into the subtleties in algorithm design required for a correct and practical implementation. To the best of our knowledge, this is a first work that explicitly studies the interplay between algorithm design and software implementation of SDN apps.

5 Conclusion

This paper presents a detailed analysis of the MAC learning algorithm and its implementation in the context of SDN. We show that a direct and straightforward migration of the classic MAC learning algorithm to SDN will not work, due to its substantial bandwidth overhead and the impractical workload placed on the controller. We then present a modified version of the algorithm that takes into account the SDN architecture. Through a detailed analysis we show that the modifications are necessary for a correct, practical and efficient implementation without impacting network performance. We believe this work will be of interest to both the software engineering and the networking communities. We plan on applying the new understanding developed in this study to migrating more algorithms and protocols to the SDN environment.

Acknowledgments

This work was generously funded by Air Force Research Laboratory through the National Science Foundation (NSF) Security and Software Engineering Research Center (S^2ERC), and by the NSF under Grants CNS-1660569 and 1835602.

References

[1] B. G. Assefa and Ö. Özkasap. A survey of energy efficiency in SDN: Software-based methods and optimization models. *Journal of Network and Computer Applications*, 137:127–143, 2019.

[2] A. AuYoung, Y. Ma, S. Banerjee, J. Lee, P. Sharma, Y. Turner, C. Liang, and J. C. Mogul. Democratic resolution of resource conflicts between SDN control programs. In *Proc. of ACM CoNext*, 2014.

[3] A. Bairley and G. Xie. Orchestrating network control functions via comprehensive trade-off exploration. In *Proc. of IEEE NFV-SDN*, 2016.

[4] T. Das, V. Sridharan, and M. Gurusamy. A survey on controller placement in SDN. *IEEE Communications Surveys & Tutorials*, 22(1):472–503, 2019.

[5] R. Durner, C. Lorenz, M. Wiedemann, and W. Kellerer. Detecting and mitigating denial of service attacks against the data plane in software defined networks. In *2017 IEEE Conference on Network Softwarization (NetSoft)*, pages 1–6, 2017.

[6] M. Hamdan, E. Hassan, A. Abdelaziz, A. Elhigazi, B. Mohammed, S. Khan, A. V. Vasilakos, and M. Marsono. A comprehensive survey of load balancing techniques in software-defined network. *Journal of Network and Computer Applications*, 2020.

[7] L. Hao, X. Sun, L. Lin, and Z. Peng. Correct software by design for software-defined networking: A preliminary study. In *Proceedings of the 32nd International Conference on Software Engineering & Knowledge Engineering (SEKE)*, pages 127–134, Virtual Conference, 2020.

[8] J. Kurose and K. Ross. *Computer Networking: A Top-Down Approach*. Pearson, 7th edition, 2016.

[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):6974, 2008.

[10] J. Mogul, A. AuYoung, S. Banerjee, L. Popa, J. Lee, J. Muidgonda, P. Sharma, and Y. Turner. Corybantic: Towards the modular composition of SDN control programs. In *Proc. of ACM HotNets*, 2013.

[11] P. M. Mohan, T. Truong-Huu, and M. Gurusamy. Fault tolerance in TCAM-limited software defined networks. *Computer Networks*, 116:47–62, 2017.

[12] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, page 8598, 2017.

[13] Y. E. Oktian, S. Lee, H. Lee, and J. Lam. Distributed SDN controller system: A survey on design choice. *computer networks*, 121:100–111, 2017.

[14] N. Sultana, N. Chilamkurti, W. Peng, and R. Alhadad. Survey on SDN based network intrusion detection system using machine learning approaches. *Peer-to-Peer Networking and Applications*, 12(2):493–501, 2019.

[15] X. Sun and L. Lin. Leveraging rigorous software specification towards systematic detection of SDN control conflicts. In *Proceedings of the 31st International Conference on Software Engineering & Knowledge Engineering(SEKE)*, Lisbon, Portugal, 2019.

[16] The Open Networking Foundation (ONF), 2015. The OpenFlow switch specification. https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[17] D. Volpano, X. Sun, and G. Xie. Towards systematic detection and resolution of network control conflicts. In *Proc. of ACM HotSDN*, 2014.