BUILDING SCALABLE SOFTWARE DEFINED OPENFLOW NETWORKS

A Dissertation
Presented to
The Academic Faculty

By

Hemin Yang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

August 2019

BUILDING SCALABLE SOFTWARE DEFINED OPENFLOW NETWORKS

Approved by:

Dr. Douglas M. Blough, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Henry L. Owen
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Russell J. Clark
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Yusun Chang
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Richard M. Fujimoto
School of Computational Science and Engineering
Georgia Institute of Technology

Date Approved: April 25, 2019

To my wife Dan who endured the highs and lows alongside me.

To my parents, Weiquan and Cuihua, who raised me up and formed me into the person I have become.

ACKNOWLEDGEMENTS

I must begin by thanking my advisors, Dr. George F. Riley and Dr. Douglas M. Blough. Dr. Riley introduced me into the area of Software Defined Networking and network simulations. He provided me with an exciting and financially comfortable environment for research through his guidance and encouragement. His advice on my research problems always helps to clear up my confusions and thus steered me toward fruitful research endeavors. His knowledge and wisdom shown in class inspired me to be a better engineer and researcher. Although he is no longer with us, his personality, encouragement, and influence will live on forever in me. Dr. Blough always shows his profound insights on my research. He can always find the key problems even through a short talk and shows me key points for improvement. It is the fruitful discussions with him about the details and critical challenges of my research that makes the quality of this thesis reaches the current level. Without him, my Ph.D. program would not have been completed successfully, as well as this dissertation. On a personal level, Dr. Blough inspired me by his rigorous and passionate attitude towards science. All in all, I would give Dr. Riley and Dr. Blough most of the credits for becoming the kind of researcher I am today. Besides my advisors, I would like to thank the rest of my dissertation committee members for their time and valuable comments. As this document is dedicated to them, I also thank my parents for their encouragement, sacrifice, and support throughout the years, and thank my friends and colleagues at Georgia Tech for their friendship and accompany. Especially, I would like to thank my wife, Dan Liu, who firmly supported me to begin this great journey four years ago. She shines brightly when I am in the dark, shows her trust when I am overwhelmed with self-doubt, cheers me up when I am upset, and shares her colorful life and profound visions with me. We together read and discuss topics about history, philosophy, literature, and art, which makes my soul more beautiful and meaningful. She is my strength and I love her more than anything else in this world.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Software Defined Networking (SDN) is widely regarded as the next generation networking technique, which can create programmable, automated, and agile networks whilst reducing costs. The core of SDN is to separate and logically centralize network control from its data plane. To achieve this separation, most SDN implementations use the *de facto* southbound protocol OpenFlow as the communication interface between the control and data planes. However, the scalability problem bottlenecks the deployment of SDN OpenFlow for large networks. The roots of SDN OpenFlow scalability problem are the centralized architecture of the control plane and the unmatched capabilities of OpenFlow switches to deal with the massive events generated by the fine-grained granularity control mechanism.

The objective of this thesis is to address the fundamental problems of scaling SDN OpenFlow networks. On the control plane, this work first investigates the scalability performance of all existing non-centralized control plane architectures versus the centralized one in order to answer the question, "which control plane architecture scales the best?" These comparisons are based on the observation that flow setup and statistic collection are the two main bottlenecks which limit the scalability performance of SDN OpenFlow networks. The simulation results show that the hierarchical control architecture and the peer-to-peer control architecture with local view (a.k.a., distributed control plane) are the most two scalable control architectures. With this conclusion, this work then aims to address the most important challenge, eastbound/westbound interface design, for the distributed control plane, which is more feasible for scaling across geographies than the hierarchical one. Specifically, the designed protocol focuses on traffic engineering, where path aggregation is applied to reduce the exchange message overhead and table sizes with little performance loss. As for the data plane, this research works around the hardware manufacturing related limitations such as CPU and bus bandwidth by improving the utilization of the existing

precise hardware resources and reducing control overheads to mitigate the scalability problem. Specifically, machine learning techniques are exploited to improve proactive flow entry deletion and flow entry eviction. The full stack of applying machine learning to solve these two problems are presented, including dataset collection, feature definition and selection, model selection and training, and how to integrate the trained model with the flow table management policies. The provided theoretical analysis and simulation results in this thesis lay out the foundation for the deployment of large scale SDN OpenFlow networks.

CHAPTER 1
INTRODUCTION

Software Defined Networking (SDN) is revolutionizing network deployment and management. Google, Amazon, Facebook, and other industrial giants have heavily invested and researched in SDN both in their data centers and wide area networks. For example, Google leveraged SDN principles to build its Jupiter network which achieves a capacity increase of 100x [1]. NTT also launched Software Defined Wide Area Network (SD-WAN) platform with coverage spanning over 190 countries in 2017 [2]. The core of SDN is to separate and logically centralize network control from its data plane. With this separation, network operators can implement any choosing control policy with small costs. In this way, the introduction of network innovations can be very fast and the management of large networks can be radically simplified and automated. To achieve this separation, OpenFlow protocol [3] is developed to serve as the southbound protocol of SDN, which specifies the interaction between the control and data planes.

1.1 Motivation and Research Objectives

Although the significance of SDN OpenFlow is commonly identified, its scalability problem bottlenecks the deployment of large SDN OpenFlow networks. To be specific, the roots of SDN OpenFlow scalability problem are:

- **Centralized control architecture is not scalable**. In SDN OpenFlow networks, all physically distributed switches should communicate with the logically centralized controller through OpenFlow protocol, including path setup, statistics collection, and state maintenance. This centralized control mechanism, on one hand, results in large event processing delays. For example, D. Erickson measured the perfor-

mance of different OpenFlow controllers and reported that the latency for processing one `Packet_In` varies between 24 us and 145 us [4]. Kuźniar et al. also found that the state of data plane can fall behind the control plane by up to 400 ms for Pica8 P-3290 switches [5]. On the other hand, the controller has to process a massive amount of events, which incurs unexpected and large queueing delays, which will be investigated in Chapter 6.

- **Fine-grained control mechanism makes flow numbers explosively grow**. SDN allows fine-grained control of traffic flows in the network. In traditional networking, a flow is typically defined as a 5-tuple: source IP address, destination IP address, source port, destination port, and protocol. In the context of OpenFlow, the granularity of a flow can be much finer. OpenFlow 1.5.0 [3] defines 45 fields which can be used to identify a flow such as VLAN ID, TCP flags, and packet type. This means a classical 5-tuple flow can be split into tens of flows in OpenFlow switches. Given the number of flows following the classical 5-tuple flow definition by itself is huge, for example, the arrival rate of flows can reach 10,000 flows per second per server rack in data centers [6], this fine-grained control mechanism makes flow numbers even more terrific and the consequently massive events make both switches and controllers very stressful in terms of computational and communicational overheads.

- **OpenFlow switches are not powerful**. OpenFlow switches are weak in terms of computation ability, bus bandwidth between management CPU and ASIC, and memory size due to cost control. For example, the measured loopback bandwidth between the ASIC and the management CPU is just 80 Mbit/sec in the HP 5406zl [7]. In this case, the latency involved in generating events can be large. According to K. He et al. [8], the latency for an Intel FM6000 switch to generate a `Packet_In` message can reach 8 ms in the case of flow arrival rate = 200 flows/sec. Similar measurements on Broadcom 956846K show that latency for flow entry insertion (modification) is 3 ms

(30 ms), which is much higher than what native TCAM hardware can support (100M updates/s). Another challenge for OpenFlow switches is the limited size of flow tables. The kernel of OpenFlow is a packet processing pipeline consisting of several flow tables. A flow table contains a set of flow entries, which are used to match and process incoming packets (e.g., forward the packet to a port, modify the packet). In most commodity OpenFlow switches, flow table is placed in Ternary Content Addressable Memory (TCAM) which can achieve single clock cycle lookup. Due to power, cost, and silicon area constraints, the size of TCAM is very limited. As reported by G. Lu et al., the Broadcom chipset which is widely used in commercial switches can only accommodate 2000 flow entries [9].

This thesis focuses on designing the non-centralized control architecture, including the relevant important applications, in order to break the limitation incurred by the centralized control plane. On the data plane, this thesis works around the hardware manufacturing related limitations. Instead, machine learning techniques are exploited to optimize the existing flow table management policies. It mitigates the scalability problem from two aspects. One is to improve the utilization of the precious flow tables such that more flows can be handled. The other one is to reduce the control overhead such that fewer events are needed to be processed for both the controller and switch. The objective of this work is to address the challenges therein and thus take the deployment of large scale SDN OpenFlow network further.

1.2 Contributions

The primary contributions of this work are:

- The first contribution (Chapter 3) is to answer the question, "Which control plane architecture is the most scalable?" All SDN control architectures are classified into five categories, each of which behaves significantly different from the perspective of how

controllers cooperate with each other. Simulation is proofed to be the best approach to compare the scalability performance of the five control plane architectures. Furthermore, simulators are developed for these five control architectures based on the abstractions on flow setup and statistic collection processes, which are the two main processes limiting the scalability performance of SDN OpenFlow networks. Various simulation experiments are conducted in different networking scenarios. Through analyzing the simulation results, the hierarchical and the peer-to-peer with local view (a.k.a. distributed) control architectures are concluded as the most two scalable architectures.

- The second contribution (Chapter 4) is that a novel eastbound/westbound protocol, SDNi-TE, for the distributed control plane is proposed to enable traffic engineering across SDN domains. Specifically, this protocol specifies what information is exchanged between neighboring domains and how the information is organized and handled such that a consistent global view of the whole network can be constructed for each SDN domain. To minimize message overhead and table size, path aggregation is applied in SDNi-TE where only one aggregated path for each (source, destination) pair is advertised. Experiments are conducted on various top-down hierarchical topologies with different traffic engineering algorithms. Experiment results show that networks operated with SDNi-TE can achieve nearly the same performance as the one with God's knowledge (the full information of the whole network).

- The third contribution (Chapter 5) is that Smart ProactivE Entry Deletion for Openflow (SPEEDO) is proposed to enable the controller to proactively delete the predicted Least Recently Used (LRU) flow entry when a flow table is close to be overflowed. LRU proactive deletion by itself is not practical for OpenFlow controllers because controllers cannot track the order of flow entry accesses in real time. However, SPEEDO can learn from the historical data of flow entry installations and deletions,

and thus predict the time when a flow entry will be last referred to. Based on the predictions, the flow entry with the smallest last refer time (a.k.a. LRU flow entry) is proactively deleted in the presence of flow table overload. Case studies are conducted to examine the overhead and performance of this novel policy, whose results show that SPEEDO can achieve 85% $\sim$ 96% performance of the infeasible LRU policy. Furthermore, compared with the random deletion and First-In-First-Out (FIFO) deletion policies, SPEEDO can achieve up to 23% fewer capacity misses, as well as a slightly decreased (2% $\sim$ 8%) overhead.

- The fourth contribution (Chapter 6) is that we propose Smart Table EntRy Eviction for Openflow Switches (STEREOS) to intelligently evict inactive flow entries when the flow table is overflowed. Specifically, LRU policy is integrated into STEREOS to overcome the shortcoming of machine learning errors. Detailed case studies are presented and many practical problems for implementing STEREOS such as model selection, model size trade-off, overhead, and feature quantization are solved, which can be a good start point for implementation in real OpenFlow switches. In addition, the first system-level simulation tool for evaluating the effect of different flow table management policies on network performance is developed based on *ns-3*. Simulations based on real network packet traces show that STEREOS can increase the usage of the flow table by over 50% and reduce the number of wrong flow entry eviction by up to 80%, compared with the LRU eviction policy. Moreover, system-level simulation results demonstrate that STEREOS can significantly reduce the control overhead, and thus improve network throughput by 19% and reduce packet loss rate by 70%.

1.3 Dissertation Organization

The remaining components of this dissertation are organized into the following chapters. Chapter 2 describes the background and related work on topics on which this work is based,

including the reasons why SDN is promising to initiate a networking revolution, OpenFlow protocol development, SDN OpenFlow scalability problem, eastbound/westbound protocol design for distributed SDN, traffic engineering in SDN, and flow table management policies. Chapter 3 elaborates how different SDN architectures are abstracted in terms of flow setup and statistic collection to build simulators such that comprehensive comparisons are conducted to answer the question "which control architecture is the most scalable?" Chapter 4 explains the novel eastbound/westbound interface (SDNi-TE) designed specifically for traffic engineering in the distributed SDN. Chapter 5 and 6 describes how to use machine learning techniques to optimize flow table management. A regression model is trained to predict which flow entry has the smallest last refer time in Chapter 5 such that controllers can cognize which flow entry should be proactively deleted when flow table overflow is going to happen. In Chapter 6, a binary classification model is trained and integrated with the flow entry eviction policy to help OpenFlow switches to identify and evict inactive flow entries in the case of flow table overflow. This work is finally concluded with an eye toward future research in Chapter 7.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 SDN and OpenFlow

In traditional networking, network devices consist of two components: data component and control component, as shown in Figure 2.1. The data component is responsible for doing line-rate packet switching, while the control component controls the behavior of the data component according to the applications installed. For example, routing application/protocol controls which port a packet should be outputted. With the traditional networking paradigm, networking innovations are very hard to be verified and implemented. The main reason is that network devices are closed and vertically-integrated bundling software and hardware. In general cases, network devices are equipped with specific software developed by their manufacturers. The software cannot be modified by common researchers to verify new ideas. In addition, to implement an innovation (e.g., access control, load balancing), the control components have to be manually upgraded one by one in the network since the traditional network is operated in a distributed way. Besides, the legacy networking is also ill-suited to handle today's heterogeneous and dynamic network traffic

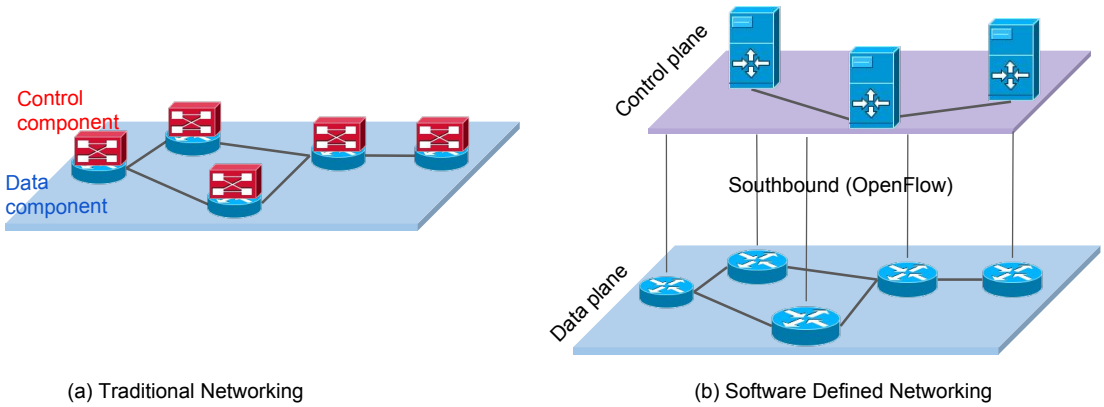


Figure 2.1: Traditional networking v.s. Software Defined Networking

due to the following contradictions [10], [11]:

- **The need of the enforcement of complex and high-level policies** versus **Distributed low-level network configuration ability**: In today's network, explosively emerged applications, including interactive and dynamic multimedia streams, Internet of Things, e-health, e-commerce, cloud computing, and so on, require various levels of QoS/QoE. To deliver the required QoS/QoE, network operators need to deploy complex and high-level policies to respond to various network events (e.g., traffic shifts, intrusions). However, in the traditional distributed networking, this can be only implemented through a highly constrained set of low-level device configuration commands. What makes it worse, a policy generally involves many network devices, which have to be configured one by one.

- **Explosively growing and heterogeneous network** versus **Inefficient network control and management**: Networks are growing explosively both in size and complexity. According to Cisco, over 500 billion devices are expected to be connected to the Internet by 2025 [12]. In addition, switches, routers and a wide range of specialized Middleboxes (e.g., network address translation, firewall, application layer gateway) coexist in the network. All these devices are controlled by their own configuration tools and operated according to specific protocols. From this perspective, the traditional distributed networking paradigm is definitely not efficient to control and manage these numerous and heterogeneous networking devices.

- **Super dynamic network status** versus **Static and manual configuration adjustment ability**: Networks are not only growing explosively in size and complexity, but also in dynamicity. Tremendous applications and the mobile behaviors of users make traffic patterns and network conditions change in a rapid and significant way. However, network configuration methods in traditional networking cannot timely react to continuously changing network status. Instead, it can only handle a snapshot

of the network state. Although external tools are developed to automate the reconfiguration of network devices responding to network events, frequent misconfigurations are inevitable.

In order to resolve the above challenges, Software-Defined Networking (SDN) initiative led by the Open Networking Foundation proposes a novel open architecture where the control plane and the data plane are physically separated with each other, as shown in Fig. 2.1. With SDN, the control plane and data plane can evolve independently since these two planes are not integrated. To be specific, the benefits of SDN are threefold:

- Network innovations can be introduced more easily. Instead of using a fixed set of configuration commands in a command line interface (CLI) environment, network innovations can be implemented by high-level programming language (e.g., P4 [13]), which is much more efficient for networking application development and maintenance.

- Network applications can be more intelligent and elaborate. The control plane in SDN actually plays the role of a network operating system, which provides abstractions (e.g., global network view) to applications running on it. In this way, more intelligent and elaborate applications can be developed to handle various network scenarios. For example, traffic engineering can be optimized since the controller has the full knowledge (e.g., topology, traffic, link utilization) of the network in its control [14].

- Centralized and automatic control. Since all control logic are centralized in the controller, operators do not need to configure all network devices individually. Instead, programming the applications running on top of the controller can directly manipulate network behaviors.

OpenFlow [3] is the *de facto* standard southbound protocol for SDN, which is designed for enabling the communication between the controller and switches. As shown in Fig.

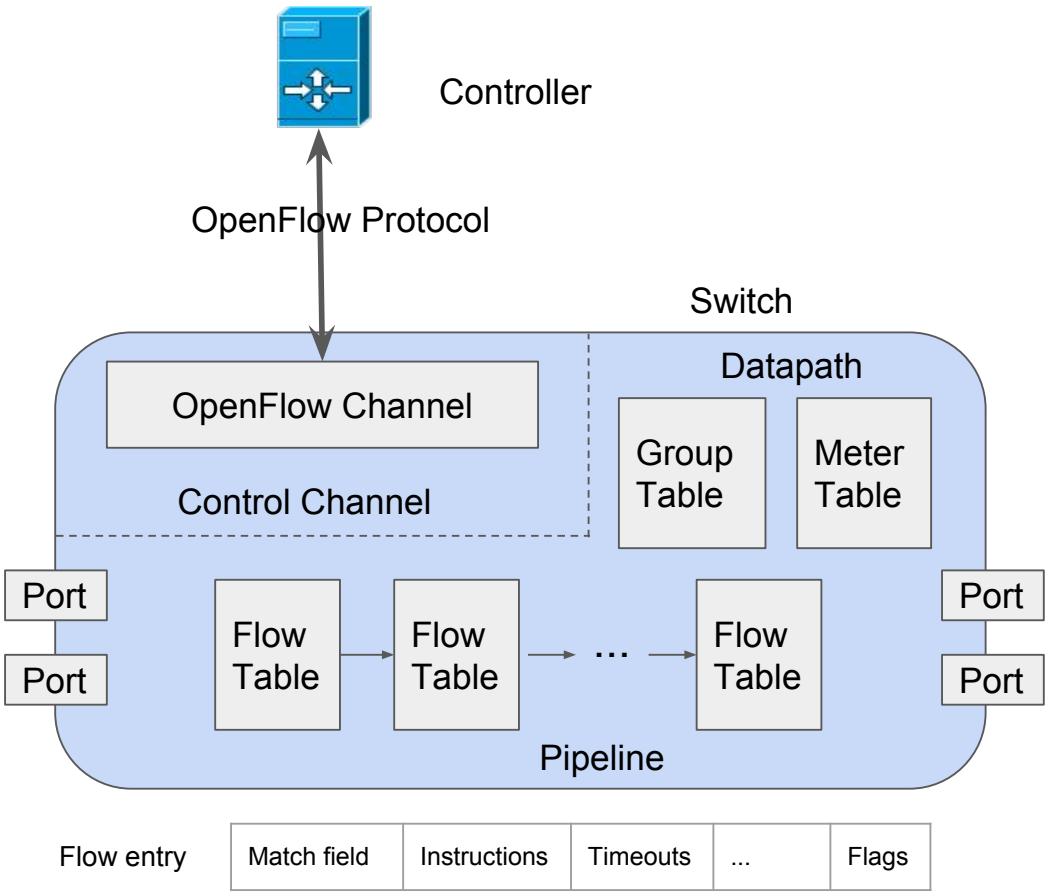


Figure 2.2: Main components of an OpenFlow switch. This figure is regenerated from [3].

2.2, OpenFlow mainly specifies components (e.g., ports, flow table, group table) and basic functions (e.g., flow match, pipeline processing) of OpenFlow switches. In addition, OpenFlow defines the interactions between OpenFlow controllers and switches. The formats of messages are specified as well. With OpenFlow, the control logic is carried by various flow entries in the flow tables. Each flow entry instructs the switch how to handle the matched packets such as forwarding a packet to a specified OpenFlow port, processing a packet through the specified group, and dropping a packet. These flow entries are installed/modified/deleted by the controller according to the applications running and network states collected from switches. More relevant details of OpenFlow will be discussed in the rest of the dissertation.

2.2 Scalability of SDN OpenFlow Networks

The scalability of a computer system is a multi-dimensional and controversial topic. It can be defined as the system's capability of increasing its speed in proportion to the increase of resources (e.g., processor, memory). It can also be referred to the ability to keep the efficiency fixed in case of increased problem size with more resources. In the context of SDN OpenFlow, there are few efforts quantifying the scalability of SDN networks. J. Hu et al. [15] adopted productivity based scalability metric, where the productivity is defined as $\varphi T/C$, φ is the throughput of the control plane in processing network requests, T is the average response time of each request, and C is the cost to deploy the control plane. M. Karakus et al. [16] used the ratio of workload (the number of flows entering the network through the data plane) over overhead (the number of messages processed in the control plane) to evaluate the scalability of SDN control plane. However, most researches measure SDN OpenFlow scalability in terms of typical Quality of Service (QoS) parameters such as throughput, path installation time, link utilization, and latency.

To scale SDN OpenFlow networks, we need to make both the control and data planes scalable. In the control plane, we can make the controller more powerful by exploiting parallelism to increase the speed of processing a massive amount of events. Beacon [4] used multiple network I/O threads where each switch connection is handled by a unique I/O thread such that the controller can simultaneously process events from all connections in its control. However, all the events are processed sequentially by a series of event handlers. To deal with this problem, Song et al. [17] proposed *ParaFlow* which explores event handler parallelism (the event processing can be parallelized to some degree if the involved event handlers are not interdependent) and task parallelism (multiple spawned tasks are executed concurrently to accelerate the event handler with high computational complexity) to support fine-grained parallelism in event processing. Besides, GPU is leveraged to mitigate the scalability problem of SDN controllers. For example, E. G. Renart et al. [18] implemented

a mac learning switch GPU controller where `Packet_In` processing, `Packet_Out` and `Flow_Mod` generation are offloaded to the GPU. Besides designing a powerful controller, redesigning the architecture of the control plane is another approach to improve the scalability of the control plane. In general, there are four non-centralized control architectures in addition to the centralized one: peer-to-peer (p2p) with local view (e.g., [19]), p2p with global view (e.g., [20]), hierarchical control plane (e.g., [21]), and hybrid control plane (e.g., [22]). Different from the centralized control plane, these four control planes contain multiple controllers and each controller only controls a subset of SDN OpenFlow switches. In this way, the control plane can be more scalable because multiple controllers can cooperate to process the massive amount of the events generated in the network. For example, K. Qiu et al. [23] proposed *ParaCon* which can distribute the load of path computation to multiple controllers and minimize the overhead between these controllers. Last but not least, the applications running on the control plane can be optimized to decrease the interaction between controllers and switches. For example, source routing encapsulates the flow's path information into the headers of its packets at the source switch [24]. In this way, there will be no interaction with the controller for the intermediate switches along the routing path during path setup. Another example is from [25], where a routing scheme is proposed to select transit routers through the network at the edge routers such that the processed events resulted from routing decisions of the controller can be reduced.

In the data plane, one of the key operations for OpenFlow switches is to conduct a series of costly packet classifications and thus match packets with flow entries. To accelerate this process, K. Qiu et al. [26] proposed *GFlow* where flow entries are organized into a directed acyclic graph. Based on this graph, flow matching can be solved in a parameterizable and effective way by exploiting the power of GPU. Another bottleneck for scaling the data plane is flow table management. On one hand, the size of flow table is limited due to power, cost, and silicon area. On the other hand, flow table update rate is also limited. R. Bifulco and A. Matsiuk noticed that it is very fast to update software flow tables and deleting entries

from TCAM-based flow tables is always much faster than adding them. Based on these two observations, they presented *ShadowSwitch* which introduces a high-performance software switching layer in the switch's fast path [27]. When a flow entry needs to be installed, it is first placed in the software switching layer and will eventually be moved to the TCAM-based flow tables. In this way, flow entry installation time can be lowered.

Besides scaling the control and data planes, it's beneficial for SDN OpenFlow scalability to redefine the boundary between these two planes. Indeed, some level control logic can be offloaded from controllers to switches. In other words, decisions only based on local states should be made inside SDN switches but not controllers. A. R. Curtis et al. proposed to enable OpenFlow switches to locally clone the wildcard rules and take a small set of possible local routing actions [7]. G. Bianchi et al. [28] proposed eXtended Finite State Machines (XFSM) based abstraction to enable stateful processing of flows in switches. With this abstraction, the control logic such as port knocking and mac learning which can not benefit from the controller's network-wide knowledge can be offloaded to switches.

2.3 Eastbound/Westbound Protocol for Distributed SDN

The p2p with local view control architecture (a.k.a., distributed SDN control plane) is appropriate to scale across geographies on which multiple SDN domains operated by different administers reside. The key challenge for implementing distributed SDN control plane is to design an efficient and effective eastbound/westbound protocol such that neighboring SDN domains can communicate with each other to provide end-to-end network services, just like the role which BGP is playing in the legacy networks. The first try of designing such a protocol was SDNi proposed by Huawei [29]. The SDNi protocol is designed for two goals: coordinate flow setup originated by applications and communicate reachability information to enable inter-domain routing. It defines three types of messages: reachability update, flow setup/tear-down/update request, and capability update. However, this proposal is very conceptional with only several concepts but no detail about what informa-

tion should be included in their defined messages, neither do they provide a prototype. The first prototype came from the DISCO project [19]. It uses an Advanced Message Queueing Protocol (AMQP) based `Messenger` to provide a distributed publish/subscribe communication channel between adjacent SDN domains. On top of the channel, different `Agents` are implemented to provide various functionalities such as path computation, reachability advertisement, and QoS resource reservation. However, the details of the east-west protocol used in DISCO are not disclosed such as what information is exchanged, when the information is sent, and how the information is handled. Without these details, we cannot reproduce and optimize the protocol. The protocol in [30] gives the answer to the question, what information should be shared among different SDN domains? According to the authors' proposal, reachability, topology, network service capabilities, forwarding capabilities, and network status should be shared. Furthermore, the information must be transformed through network virtualization before sharing such that the privacy and security concerns can be well taken. However, it only supports multi-path routing based on different source IP addresses, while we are more interested in delivering the traffic originated from the same address with multiple paths. In addition, F. Benamrane et al. [31] proposed and implemented a Communication Interface for Distributed Control Plane (CIDC) based on event-driven paradigm which consists of `Consumer`, `Producer`, `DataUpdater`, and `DataCollector`. `Consumer` and `Producer` manage the connections between neighboring controllers. `DataCollector` is responsible for collecting various local network status, while `DataUpdater` handles the received external data from the `Consumer` such as directing it to some application and saving it on the console. However, it is not clear how global routing is achieved in the paper and what network status are collected for the routing.

2.4 Traffic Engineering in SDN

Besides designing an eastbound/westbound protocol, developing key applications such as traffic engineering, routing, and admission control is also critical to make distributed SDN work. This work focuses on traffic engineering because it is not only of premier importance to minimize congestion, delay and packet loss, but also can benefit a lot from SDN [14]. To be specific, the fine-grained control mechanism in SDN makes it possible to split a flow into multiple sub-flows, thus improve the network resource utilization. In addition, the centralized control plane can be implemented with more advanced path computation algorithms fed with up-to-date network state information.

Traffic engineering (TE) is intensively researched in the area of networking and communication. Within the context of SDN, most researches focus on the TE in one SDN domain. In this case, the controller has the global network and application information and thus can implement much more efficient and intelligent TE algorithms. Google's B4 WAN connecting its data centers across the planet uses a centralized TE architecture [32]. It defines a bandwidth function for every application which specifies how much bandwidth should be allocated to one application given the application's relative priority. Based on these bandwidth functions, the TE algorithm allocates edge bandwidth among competing flow groups (flows in one flow group have same source address, destination address, and quality of service) and adjusts the allocation splits to the granularity supported by the underlying hardware. Microsoft proposed to allocate bandwidth in strict precedence according to applications' priority classes, while applications with higher classes prefer shorter paths [33]. Within a class, the bandwidth is split in a max-min fairness manner. Huawei presented an adaptive dynamic multipath computation framework for centrally controlled SDN networks to provide the infrastructure and algorithms for resource control and management [34]. In Huawei's solution, historical data of topology, tenants, and application profiles are stored in a data-warehouse. These data can be mined to exhibit flow patterns and generate

features. With the found patterns and features, a combination of enhanced edge-disjoint path algorithms and other heuristic optimization algorithms can optimally find more than 2 paths for every flow under various network state and traffic conditions.

Besides centralized SDN networks, another research scenario for TE is the hybrid SDN, which consists of traditional networking elements and SDN elements. S. Agarwal et al. first considered this scenario [35]. They made a slight modification on the routing table in SDN switches to measure the amount of traffic between the switch and all other nodes in the network. Along with the measurement, OSPF link weights and the traffic at the links are taken into account during route computation. Specifically, they formulated the TE problems in hybrid SDN as an optimization problem involving delay and packet loss which tries to minimize the maximum link utilization. However, their formulation assumes all link weights are fixed and same. In [36], the authors proposed a heuristic algorithm for searching the optimized weight setting. Their algorithm starts from an initial weight setting which is generated from the Floyd algorithm, and searches a better setting in terms of maximum link utilization in the neighbor of the current setting until convergence. These two solutions assume that SDN switches build their routing tables from OSPF, but M. Caria et al. have a different perspective [37]. They proposed to partition the initial OSPF domain with SDN switches into sub-domains. In this way, the routing inside a sub-domain can be based on OSPF solely and remains unchanged. Meanwhile, the inter-sub-domain paths updates can be overridden at SDN switches by the controller. Not only traffic splitting but also next-hops construction (which hop a flow goes through) can impact the effectiveness of TE. With this observation, the authors in [38] proposed a heuristic algorithm to adjust the next-hops of SDN switches in forwarding graphs in order to maximize the minimum satisfied percentage of all the flows. J. He et al. [39] did a similar job, except for that they proposed a novel routing protocol for passing SDN traffic through traditional networks and modeled the TE problems in two different network scenarios. Y. Guo et al. [40] considered TE with multiple traffic metrics in hybrid SDN. To be specific, the authors

optimized OSPF weights offline for legacy switches over the expected traffic metric which is a linear combination of multiple representative traffic metrics found by data mining.

As for the case of distributed SDN, the research of TE is very primitive. The only related publication which can be found online is a patent from Huawei [41]. This patent presents a method for resource provisioning in distributed SDN for TE, which includes receiving border element information from multiple SDN domain controllers, computing allocation constraints based on the collected border element information, and sending the allocation constraints to SDN controllers.

2.5 Flow Table Management for OpenFlow Switches

OpenFlow currently provides three mechanisms for flow table management [3]:

1. Flow expiry. The controller can set `idle_timeout` and `hard_timeout` for each flow entry. If no packet refers to a flow entry within its `idle_timeout`, the switch will delete the flow entry. In addition, a flow entry will be removed from the flow table after the given `hard_timeout` regardless of how many packets it has matched.
2. Proactive deletion. The controller can delete flow entries by explicitly sending specific OpenFlow messages (e.g., `OFPFC_DELETE`) to switches.
3. Eviction. In the case that the flow table is full, the switch can kick out existing flow entries to install newly inserted flow entries instead of simply rejecting them.

Currently, most researches about flow table management focus on how to dynamically and adaptively set `idle_timeout`. For example, A. Vishnoi et al. proposed *SmartTime* which employs an adaptive heuristic to compute `idle_timeout` for flow entries [42]. In *SmartTime*, the `idle_timeout` of a flow will grow exponentially with respect to the flow repeat count until reaches a predefined maximum timeout. In addition, `idle_timeout` will be reduced to the minimum timeout when a flow continues to have a bad average hold factor (the sum of active time and idle time divided by the active time). H. Liang et al. [43] investigated how to set effective `idle_timeout` value for instant messaging

applications. Based on an ON/OFF traffic model for instant messaging applications, the authors tried to set `idle_timeout` to minimize the weighted sum of the invalid lifetime of flow entry and the rate of `Packet_In` event generation. A. Zarek investigated the complex relationship between timeouts, miss rate, and table occupancy and thus argues that the OpenFlow controller should autonomously assign tuned timeouts to individual flows such that network constraints (e.g., miss rate, table size, and power consumption) can be satisfied. Furthermore, he proposed to combine fixed unified timeout with explicit controller eviction messages [44]. H. Zhu et al. assigned suitable timeout depends on flows' characteristics and conducted a feedback control on the maximum timeout [45]. To be specific, `idle_timeout` is increased by the time interval between the re-triggering and the last expire of a flow entry. In addition, a feedback control mechanism which adjusts max timeout value is introduced to keep the flow table load at a suitable level. C. H. He et al. [46] proposed to detect the `FIN` and `RST` packet of a TCP flow which indicates the end of a flow. Once these packets are detected, the switch can then delete the corresponding flow entry. In this way, no timeout is needed to be set. This method can only handle TCP flows and does not work if a flow is defined in a smaller granularity than TCP. A more general approach is proposed in [47], where Q-learning is used to learn traffic dynamics and data plane performance such that different timeout values can be assigned to different rules.

As for the problem of proactive deletion, it remains largely under-explored so far. The only related literature which can be found online is from A. Vishnoi et al [42], where a random deletion policy (delete a random flow entry) and FIFO policy (delete the first installed flow entry) are proposed. Another possible strategy is to delete the least recently used (LRU) flow entry. However, to apply LRU on the controller side, the controller has to track the order of flow entry accesses in real time. This is infeasible within the framework of OpenFlow because the controller only has non-real time accesses to coarse-grained counters. Even if the controller can achieve real-time tracking, the overhead of signaling will be intolerable.

In previous versions of OpenFlow specification, new flow entries will not be inserted in the flow table and an error will be returned to the controller if a flow table is full. However, this approach is problematic because the service may be disrupted. From OpenFlow 1.4.0, eviction mechanism is introduced to enable smoother degradation of behavior in the case of flow table overflow. Once the flow table is full, the eviction enabled OpenFlow switch can kick out existing flow entries of lower importance to install new flow entries instead of simply rejecting them. For flow entry eviction, the key issue is to decide which flow entry should be evicted. Intuitively, we can come up with three naive strategies: LRU, FIFO, and random. These three policies evict the LRU, first installed, and random flow entry respectively. A. Zarek compared the performance of these three strategies [44] based on several real network traces and concluded that LRU outperforms the other two strategies. Besides these three naive strategies, R. Challa et al. [48] employed multiple bloom filters to encode the importance value of flows which captures both the locality and recentness of reference. With these values, the switch can evict the "least important" flow entry in the case of flow table overflow. T. Pan et al. [49] considered the heavy-tailed distribution of network traffic, and proposed to use the correlation between flow size and flow entry evict times to identify elephant flows. Based on the identification, the authors proposed Adaptive Least Frequently Eviction to prevent elephant flows from being evicted by massive mice flows through assigning elephant flows with higher priorities. B. Lee et al. [50] also took network traffic characteristics into account but focused on the fair treatment of elephant flows because these flows are more likely to be evicted when compared with mice flows. To ensure fair treatment of elephant flows, they proposed a fair eviction strategy based on LRU, where a new mice flow can only evict a mice flow and an elephant flow can only evict an elephant flow. K. Kannan et al. [51] built a Markov based learning predictor which captures the probability of transitioning between different intervals and evicts the flow entry in the state where the transition being the least. Their approach is based on the assumption that the arrivals of flows follow Poisson distributions.

CHAPTER 3

SCALABILITY COMPARISON OF SDN OPENFLOW CONTROL PLANE ARCHITECTURES

As discussed in Section 2.2, there are five SDN control plane architectures in total: centralized, p2p with local view, p2p with global view, hierarchical, and hybrid. A straightforward question arises of which control plane architecture has the best scalability performance. This question is significant because it is one of the primary questions to be answered when network operators plan to deploy a large scale SDN OpenFlow networks. The answer to this question determines the architecture and operation of the whole network. However, it is difficult to answer this question because most implementations of these control plane architectures are not publicly available (e.g., Orion [22], Disco [19]) due to security and intellectual property concerns. As the first piece of this dissertation, the work in [52] tries to answer this question by identifying the bottlenecks for scaling SDN OpenFlow networks and abstracting the controller and switches in the five control architectures to some level based on these bottlenecks. The identified bottlenecks are flow setup and statistic collection. Flow setup refers to the process where the controller setups a path for a coming flow which cannot be matched with any existing flow entry. Statistic collection is the process where the controller collects the information (e.g., configuration, capabilities, statistics) from switches via the pull-based Read-State mechanism in OpenFlow. Based on the abstractions on these two bottleneck processes, simulators can be built and thus simulation experiments are conducted to compare the scalability performance of different control architectures.

3.1 Scalability Evaluation for SDN OpenFlow Control Planes

This section argues that simulation is the best approach to compare the scalability performance of different control plane architectures. In general, all studies of scalability evaluation for SDN can be classified into four classes in terms of the methodology used: empirical approach, emulation approach, simulation approach, and mathematical modeling approach.

3.1.1 Empirical Approach

For the empirical approach, a real SDN network/testbed with physical switches, hosts, controllers, and links should be deployed. For example, D. Erickson et al. [53] built the Data Center Network Research Cluster, which contains 80 Google production servers. M. Casado et al. [54] created a functional Ethane network which consists of 19 switches, 300 registered hosts and several hundred users. The empirical approach is accurate because it investigates a real hardware network and employs no simplified abstractions. However, it needs a large amount of resources (both hardware and software) to collect enough realistic scalability statistics. This is the reason why the maximum size of SDN research testbeds reported is less than 500 servers, which is much smaller than the required size (e.g., data center network should be scaled up to hundreds of thousands of servers) to evaluate the scalability of practical SDN networks.

3.1.2 Emulation Approach

The emulation approach is the most popular way to evaluate the performance of SDN networks. The *de facto* standard for SDN emulation, Mininet, is widely adopted by SDN researchers to demonstrate the advantages of their proposals. In Mininet, virtual Ethernet pairs and processes in Linux network namespace are employed to create a lightweight virtualization of end-hosts, switches, routers, links, and controllers. In this way, a realistic network running real kernel, switch, and application code can be created on a single ma-

chine. Mininet provides Python-based APIs, which can be used to create custom topologies of switches, hosts, links, and built-in controllers. Furthermore, Mininet can be connected to external controllers, which can be running anywhere on the control network. Accordingly, the prototypes of various SDN controllers can be developed and then connected to Mininet for performance evaluation. For example, Disco controllers are deployed on Virtual Machines (VMs) and connected to a Mininet emulated network (3 SDN WANs with 4 switches each and connected to each other) [19]. Orion [22] controllers are tested on an emulated network with maximum 720 nodes in Mininet.

The emulation approach allows creating a network testbed that resembles a hardware network. Without the overhead of the entire computer systems (OS, memory, etc.), it can achieve similar results with significantly fewer resources. However, this approach is faced with three challenges in terms of scalability evaluation:

1. It needs the source codes of external controllers, but most of them are not open source (e.g., Kandoo, Orion, etc.);

2. Mininet has to be adapted to be compatible with some external controllers. For example, Elasticon [55] is tested on an enhanced Mininet testbed which does not involve any packet transmission in the data plane. ONOS [56] provides *onos.py* to help users to emulate ONOS network with Mininet in a single VM. These adaptions are based on the full knowledge of both Mininet and external controllers (e.g., implementation details, software architecture, etc.). It is very difficult, if not impossible, to achieve this knowledge for all external controllers of the five control planes;

3. Mininet has some limitations related to performance and resource usage. All the networking elements (i.e., links, hosts, switches, controllers) in the emulated network share CPU and memory resources in a single machine. Moreover, all Mininet hosts share the host file system and PID space. Therefore, as J. Ivey et al. [57] found, the performance significantly degrades when the emulated network scales to 1,000

nodes and more because the memory of a single system is not enough to support the virtualization of thousands of nodes[1]. As far as known, the maximal number of nodes (controllers, switches, and hosts) in the network emulated by Mininet is 861 [57]. Furthermore, Mininet does not support virtual time and all timing measurements are based on real time. Therefore, faster-than-real-time results are hard to be emulated. In addition, high-speed links (e.g., 10 Gbps) cannot be supported in Mininet due to the fact that all packets are forwarded by a collection of software switches (e.g., Open vSwitch). These software switches have lower performance than dedicated switching hardware. Finally, we cannot specify specific bandwidth limits or quality of service on the default connections type in Mininet. In this case, we may use TCLinks which employ the Linux traffic control program. However, the main Mininet process is still under the control of the Linux scheduler of the system on which it is run. In this way, the emulated nature of Mininet will not guarantee identical results as simulation would.

3.1.3 Simulation Approach

Simulation is another option for evaluating the scalability performance of SDN control planes. Different from emulation, simulation time is not intended to coincide with wall-clock execution time. Another notable difference between simulation and emulation is that simplified abstractions are widely used in simulations. For example, the flow simulator *fs* operates on the higher level notion of a *flowlet* as its network abstraction instead of packets [58]. Another example is that the switch and controller modules in the network simulator *ns-3* do not capture the operation of writing to the consistent distributed file system. These abstractions will definitely have some impacts on simulation accuracy. However, a much larger network can be evaluated by simulations with the abstractions. In addition, these

[1] We can limit the CPU bandwidth of Mininet hosts to scale the emulated network. However, this limitation will impede the realism of the virtualized hosts. Furthermore, such limitations cannot be imposed on the virtualized switches and internal controllers.

abstractions allow verifying the design principles of SDN controllers by simulation. For example, A. R. Crutis et al. [59] built a flow-level data center network simulator to evaluate the performance of DevoFlow on a large-scale network. Their model only captures the overheads generated by each flow and the coarse-grained behavior of flows. H. Owens et al. [25] implemented Video over SDN and Explicit Routing in SDN based *ns-3*.

3.1.4 Mathematical Modeling Approach

The last approach for evaluating the scalability performance of SDN control planes is based on mathematical modeling. J. Hu et al. [15] modeled the flow setup as an M/M/1 queueing and derived the closed-form solution for the response time of flow initiation. However, to get this solution, many unrealistic assumptions are introduced such as independent identical distribution and unlimited controller bandwidth. Moreover, only flow setup is modeled but not other processes such as stochastic collection and state management, which further makes this mathematical approach less convincing.

Based on the above analysis, it is infeasible to evaluate the scalability performance of the five existing SDN control plane architectures to use either empirical approach or emulation approach. We cannot gather enough hardware and software resources to test different SDN controllers on a very large scale "real" network. Furthermore, we do not have the source codes of the necessary SDN controllers (e.g., Kandoo, Orion). Even if we get the source codes, it would also be very difficult to custom Mininet to be compatible with these controllers. In addition, it is doubtable for Mininet to emulate tens of thousands of nodes in a PC/server. Therefore, we can only apply the simulation method or mathematical modeling approach to evaluate the scalability of the five SDN control plane architectures. Compared with the simulation approach, mathematical modeling has to introduce unrealistic assumptions and can only model one specific process (e.g., flow setup). Thus, the simulation approach is applied to compare the scalability performance of the existing SDN control plane architectures in this work. The accuracy of this approach does be lower than

empirical approach and emulation approach due to the level of abstraction, but it is the best of the worst solutions and it can give us a good insight of the scalability performance of different control plane architectures.

3.2 SDN OpenFlow Control Plane Architectures

Obviously, it is infeasible and unnecessary to implement every detail of the controller and switch for simulations. Otherwise, it would be faced with similar problems as the emulation approach. The methodology used in this work is to identify the bottlenecks for scaling SDN networks and abstract the controller and switch to some level based on these bottlenecks. Based on these abstractions, simulators can be built to run simulations.

The control plane in SDN is responsible for providing the SDN applications with an abstract view of the network and configuring the network elements according to the logic of the SDN applications. As discussed in Section 2.2, the scalability of the control plane is restricted by the capability of the single controller and the signaling overhead between controllers and switches. Specifically, the bottleneck for scaling the size of an SDN OpenFlow network lies in flow setup and statistic collection [59], [60]. Flow setup refers to the process where the SDN agent on the switch

1. executes a lookup in the flow tables when a new flow comes in;
2. if no match of this flow is found, a new flow request is sent to the controller;
3. the controller responds the request with a new forwarding rule and sends it to the switch;
4. the switch updates its flow tables according to the received rule.[2]

Statistic collection is the process where the controller collects the information from switches via the pull-based Read-State mechanism. To be specific, the OpenFlow controller can send

[2]Actually, this is reactive flow setup. Flow setup can also be proactive, where the controller populates the flow table ahead of time for all traffic matches that could come into the switch. In this work, only reactive flow setup is considered.

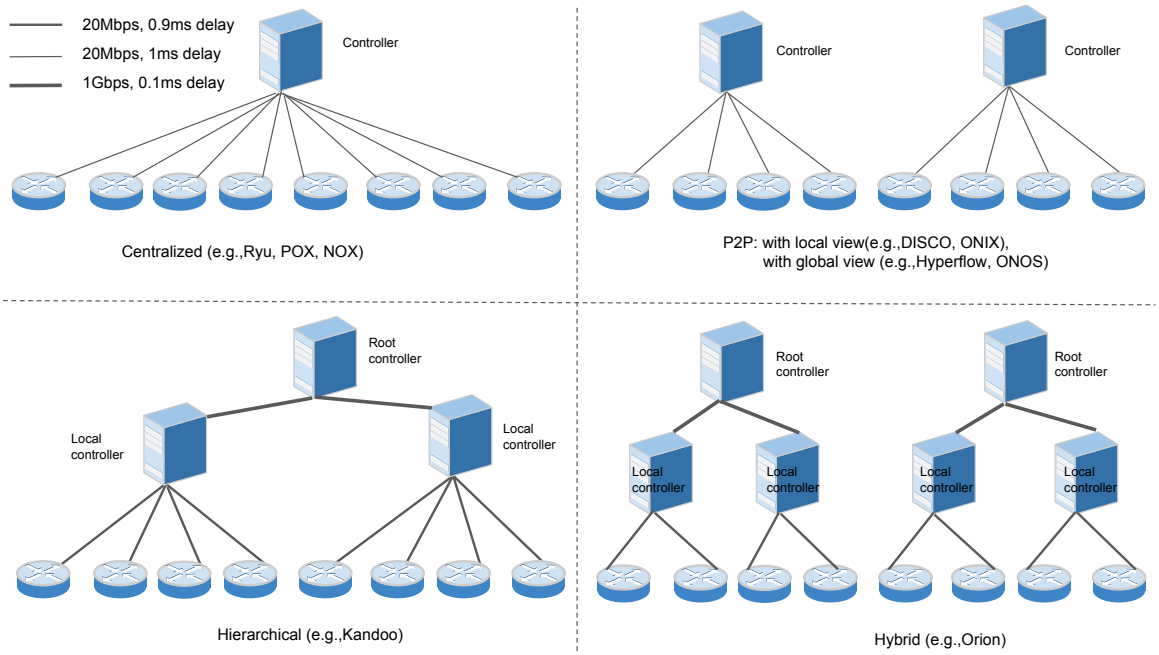


Figure 3.1: Five existing SDN OpenFlow control plane architectures.

requests to switches periodically or upon the requests from control applications. These requests ask switches to report its status, including changes to flow tables made by other controllers, capabilities and configurations of existing tables, table statistics (e.g., number of packets hitting table), meter statistics (e.g., maximum number of meters), port statistics (e.g., number of received packets), flow statistics (e.g., priority of a flow entry), queue statistics (e.g., number of packets dropped), group statistics (e.g., time a group has been alive), and so on. In the presence of receiving a state request, the OpenFlow switch should send reply messages to the controller according to the formats defined in OpenFlow. In the following, the five existing control plane architectures, as shown in Figure 3.1, will be described in terms of these two scaling limitations: flow setup and statistics collection.

3.2.1 Centralized Control Plane

In a centralized control plane, only one logically centralized controller manages the whole network. All flow initialization requests are sent to this controller, and it sends flow installation messages to one or many related switches according to its control logic. As for

statistics collection, the controller will periodically send state requests to all switches in its control to gather resource states. Receiving the state request message, the switch will send state respond messages back to the controller. Examples of the centralized control plane are Ryu, POX, NOX, etc.

3.2.2 P2P Control Plane with Local View

In the P2P control plane, there are multiple controllers. Each controller controls a sub-network of the whole network and these controllers have a peer relationship. The controller has either the local view of the sub-network in its control or the global view of the whole network. In the P2P control plane with local view, the neighboring sub-networks are abstracted as logical nodes. In this way, the controller can propagate the flow initialization request to its neighbors and thus set up global flows. Specifically, receiving a flow request, the controller first identifies whether the destination host of this flow is in its control. If it is, the controller responds to one or many switches in its control with flow installation messages. Otherwise, a new flow initiation request is generated and sent to one of the neighboring controllers. The neighboring controller processes the request as above until the destination host is reached. As for statistics collection, each controller only needs to collect the statistics of the sub-network it controls. (i.e., the controller sends state request messages to the switches in its control and the switches respond with state response messages.)

3.2.3 P2P Control Plane with Global View

Different from the P2P control plane with local view, each controller has a global view of the network and does not abstract the neighboring domains as logical nodes in the P2P control plane with global view. This global view is acquired through the communications between adjacent SDN controllers. With the global view, the controller by itself can process all the flow initialization requests generated by the switches in its control. Specifically, the

controller will send flow setup messages to the required switches regardless of whether or not they are in its control. The statistics collection in P2P with global view is exactly the same as in P2P control plane with local view.

3.2.4 Hierarchical Control Plane

In the hierarchical control plane (e.g., Kandoo [21]), the controllers are organized in a tree structure. Generally speaking, there are two layers in the control plane. The top layer consists of only one root controller while the bottom layer has multiple local controllers. The local controller directly manages the sub-network in its control and only has the view of the sub-network. Furthermore, there is no interconnection among the local controllers. The root controller abstracts each local controller and its sub-network as one logic node. In this way, the root controller can manage the local controllers and has the full view of these logical nodes. This hierarchical structure can keep most of the frequent events handled by the local controllers and thus shield the root controller. Within this structure, the flow initialization request will be processed by the local controller and/or the root controller. If the destination host and the source host of a flow are managed by the same local controller, the flow initialization request will be only processed by the local controller. Otherwise, the local controller will relay the request to the root controller, which will then delegate the flow installation response to the respective local controllers. These local controllers will send flow setup messages to their switches based on the response from the root controller. In this way, a new global flow is established. Statistics collection in the hierarchical control plane will be much more complex than the two planes have been discussed. The local controller not only needs to collect the statistics from the switches but also may want to collect from the root controller. For example, the local controller can request topology information from the root controller. Moreover, the root controller needs to gather the statistics of the logical nodes (note that a logical node is abstracted from a local controller and the sub-network in its control).

3.2.5 Hybrid Control Plane

The hybrid control plane is a combination of the hierarchical control plane and the P2P control plane with local view. In the hybrid control plane, the whole network is divided into several SDN domains without overlapping. Each domain is managed by a hierarchical control plane, consisting of one root controller and multiple local controllers. These root controllers are organized in the P2P way, where the root controller only has the local view of its domain but can synchronize the global abstracted network view through some distributed protocol (e.g., SDNi). Within this control plane, the flow installation is straightforward. The flows can be divided into three types: local flow, domain flow, and global flow. If the source host and destination host of a flow are managed by the same local (root) controller, the flow is a local (domain) flow. Otherwise, it is a global flow. The local flow initialization request is only processed by the local controller. The domain flow is processed by the root controller first and then will be split into multiple local requests which will be processed by the respective local controllers. The global flow installation involves multiple SDN domains and is processed as the following:

1. the root controller processes the flow request in the same way as the one of domain flow;

2. if the flow request is a global request, a new flow request (either global flow or domain flow) is generated and sent to the neighboring root controller. Otherwise, stop.

3. the neighboring root controller processes the received request as 1) and 2).

The statistics collection in each domain is independent and same as in the hierarchical control plane.

3.3 Simulation Implementation and Configuration

3.3.1 Control Plane Abstraction and Implementation

Note that we are primarily concerned with the control plane traffic load for evaluating the scalability performance of SDN control planes, thus it is unnecessary to simulate packet transmission on the data plane. Therefore, like in [55], an OpenFlow switch is abstracted as an entity which sends a `Packet_In` message to the controller when a new flow arrives without actually transmitting packets on the data plane. Furthermore, the switch will check whether the flow entries in the flow table are expired, which incurs `Flow_Removed` message transmission in the case of expiration. In addition, the switch needs to handle the received packets (e.g., `Flow_Mod`) from the controller. On the controller side, the controller needs to periodically pull the statistics of the switches in its control. It also needs to handle the `Packet_In` message, including packet decoding, policy decision, and `Flow_Mod` transmission. Besides the switch and the controller, we abstract the control channel as a TCP socket. To build the control channel, the switch needs to launch the connection and the controller has to start a socket to listen for incoming connection attempts.

Based on these abstractions (shown in Figure 3.2), OpenFlow switch and controller applications for the existing five SDN control plane architectures are individually implemented in *ns-3*. These applications allows the nodes in the simulated topology to set up flows and/or gather statistics. To be specific, the implementation is comprised of the `SdnSwitch` class for all five control planes, `SdnController` class for the centralized control plane, `SdnControllerLocal` for the P2P with local view, `SdnControllerGlobal` for the P2P with global view, `SdnLocalController` and `SdnHiRootController` for the hierarchical, `SdnLocalController` and `SdnHyRootController` for the hybrid. The `SdnSwitch` object can establish the socket connection with the controller object, send packets (i.e., `Packet_In`, `Flow_Removed`, `Multipart_Res` messages) to the controller, receive and handle packets (`Flow_Mod`, `Multipart_Req` messages), and

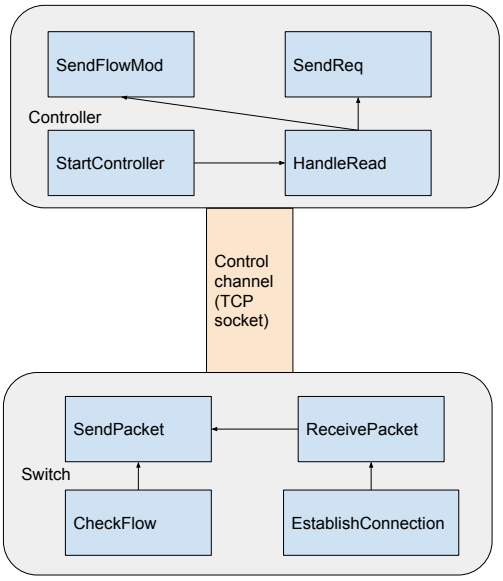


Figure 3.2: The abstractions of the controller, switch and control channel used in the simulators based on *ns-3*.

periodically check whether the flow entries in its flow tables are expired. The `SdnController` object can respond to the flow initialization request with a `Flow_Mod` message for path setup, periodically send `Multipart_Req` message to the switch to gather the statistics, and receive and handle the packets from the switch. The `SdnControllerGlobal` is the same as `SdnController` except that it can access other controllers to get the global view. Similarly, the `SdnHiRootController` is the same as the `SdnController` except that it needs to respond to the `Multipart_Req` messages from the local controllers. The `SdnControllerLocal` extends the `SdnController` to be able to split a global flow initialization request into multiple ones and send them to other controllers. The `SdnHyRootController` also extends `SdnHiRootController` in the same way. The `SdnLocalController` is slightly more complicated than `SdnController` because it not only needs to pull the statistics from the switches and respond to the flow initialization request from the switches but also pull the statistics from the root controller and process the `Multipart_Req` and `Flow_Mod` from the root controller.

Moreover, we need to model the time spent by the CPU in the controller in processing

the flow initialization request. The assumption in [15] that the time for responding to an initial flow setup packet is determined by $Cg(V, E)$ is used to model CPU processing time, where C is a constant representing the CPU speed, $g(V, E)$ is the time complexity to run the routing and/or traffic engineering algorithms in the graph (V, E). In this work, $g(V, E) = V^2$ which is the complexity of the Dijkstra's algorithm. In addition, we also need to model the time spent by the switch in pulling the statistics from its flow tables. According to A. R. Curtis etc. [59], the latency of statistics gathering is almost linear to the size of the flow table. Therefore, $L = KN$ is employed, where L is the latency, N is the number of flows in the flow table, and K is a constant reflecting the speed of the switch to gather the statistics from its flow tables.

3.3.2 Traffic Model and Topology

Since only flow setup and statistics collection are necessary to be simulated, the real concerns for the simulations are when a flow arrives and a statistics request happens, but not the packets in the flow (these packets go through the data plane). Moreover, we do not care which path a flow goes through is, but only how many switches are involved in the path. In this way, the size of the simulated network is the concern in simulations but not the topology. This makes sense because we aim to study the scalability of the control planes regardless of network topology. Therefore, a simplified topology, where each switch is connected to N_h hosts and these hosts are not connected with each other, is simulated. To model the number of switches a flow goes through, the parameter D_f, which is the number of switches a flow passes through on average, is introduced. This parameter is defined in different ways for different control plane architectures. $D_f = D_c D_s$ for the P2P control planes and the hierarchical control plane and $D_f = D_d D_c D_s$ for the hybrid control plane, where D_d is the number of SDN domains crossed by a flow, D_c is the number of controllers in a domain involved in setting up a flow on average, D_s is the average number of switches a flow goes through in a sub-network. As for the flow arrival, we assume that the time

interval of flows at a host is subject to exponential distribution[3]. Therefore, the arrival of flows at the switch is subject to the Poisson distribution with an average inter-arrival time λ.

3.3.3 OpenFlow

This simulation-based study is based on OpenFlow 1.3, which is a generally widespread protocol in SDN. (Since flow setup and statistics collection are same for both OpenFlow 1.3 and the latest OpenFlow 1.5 except that the latest OpenFlow defines different messages, this research can be easily extended to OpenFlow 1.5.) In OpenFlow 1.3, the flow initialization request is packed in `Packet_In` message and the flow installation response is carried by `Flow_Mod`, which instructs the switch to install a specified flow table entry. For statistics collection, the controller can send `Multipart_Req`, `Feature_Req`, `Queue_Get_Config_Req`, etc. to the switches to query the relevant information. Accordingly, the switches will respond to the controller with `Multipart_Res`, `Feature_Res`, `Queue_Get_Config_Res`. In this study, only the process that the controller periodically sends `Multipart_Req` to the switches and they respond with `Multipart_Res` is considered. This is because `Multipart_Req` can be used to query the states of the flow, table, port, queue, group, meter, etc. and it is most frequently used. (Actually, `Queue_Get_Config_Req` and `Feature_Req` no longer exists in OpenFlow 1.4 and 1.5, and their functionalities are integrated into `Multipart_Req`.) Furthermore, the communication between the root controller and the local controller in the hierarchical control plane is assumed to be OpenFlow based. This is reasonable because the local controller and its sub-network are abstracted as a logical node for the root controller. This assumption also applies to the hybrid control plane. In addition to flow setup and statistics collection, the flow remove process is also simulated in this study, which is critical to simulate the delay of statistics collection. A flow entry is removed when idle timeout or hard timeout occurs.

[3] Actually, you can change the statistics of the distributions as you want in simulations. The exponential distribution used here is just for illustration and comparison.

An idle timeout happens when no packets are matched in a period time, and a hard timeout happens when a certain period of time elapses, regardless of the number of matching packets.

3.3.4 Simulation Parameters

Most of the parameters used in the simulations are derived from the empirical studies reported in [59], [6], [61]. According to [61], the average inter-arrival time for flows at a datacenter server is set as 30 ms. Furthermore, the number of hosts connected with one switch is set to be 20, and the number of switches in the network to be 100 or 300. In this way, the maximum number of nodes in our simulated network reaches 6000, which is compatible with the size of commercial data centers reported by T. Benson et al. [6]. In addition, the bandwidth of the control channel (BW_c) is configured to be 20 Mbps[4], and statistics pulling speed to be $178us/flow$ based on the measurements on the HP ProCurve 5406zl [59]. As for the statistics-gathering interval, the configuration from [59] is borrowed, which is 5 seconds. In addition, the flow entry timeout is set to be 10 seconds. To summarize, the relevant parameters used in the simulations are listed in Table 3.1. Note that the bandwidth of the channel between the root controller and the local controller is configured as 1 Gbps, because the factors (e.g., the slow control datapath between the ASIC and the management CPU in the switch, the wimpy CPU in the switch) limiting the data rate between the switch and the controller do not exist for the channel between the root controller and the local controller.

3.4 Simulation Results and Discussions

As discussed in Section 2.2, QoS metrics are typically used to evaluate the scalability of the SDN control plane. In this study, three metrics: flow setup delay, statistics collection

[4]Generally speaking, the switch and controller are connected through a fast physical medium. However, the switch's management CPU is weak and cannot rapidly encapsulate packets for transmission to the controller. Furthermore, the control datapath between the ASIC and the CPU is slow, which further makes the control bandwidth low.

Table 3.1: Simulation parameters

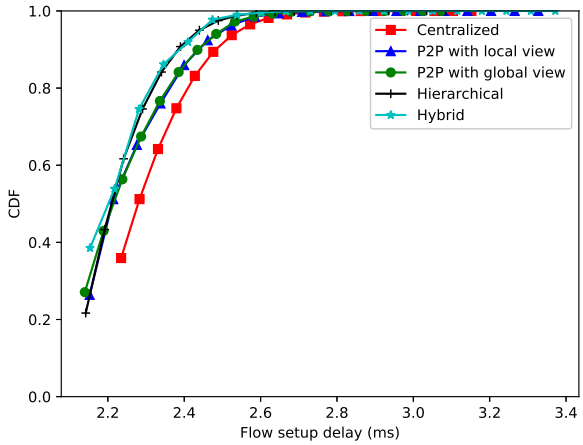
Parameter	Value
K (speed of gathering statistics of the switch)	178 us
N_h (number of hosts connected to a switch)	20
λ (average inter-arrival time of flows at the host/server)	30 ms
C (CPU speed for processing the flow setup request)	10 ns
D_s (average number of domains a flow goes through)	2
BW_c (bandwidth of control channel between the switch and its controller)	20Mbps
The frequency for statistics gathering	every 5 s
The frequency for checking flow entry expiration	every 1 s
The propagation delay between the controller and the switch for the centralized and P2P control planes	1 ms
The propagation delay between the local controller and the switch	100 us
The propagation delay between the root controller and the local controller	900 us
The bandwidth of the channel between the root controller and the local controller	1 Gbps
Flow table entry timeout	10 s
Simulation time	60 s

Table 3.2: Descriptive statistics for the control planes with $D_f = 8$, $BW_c = 20Mbps$, and 100 switches

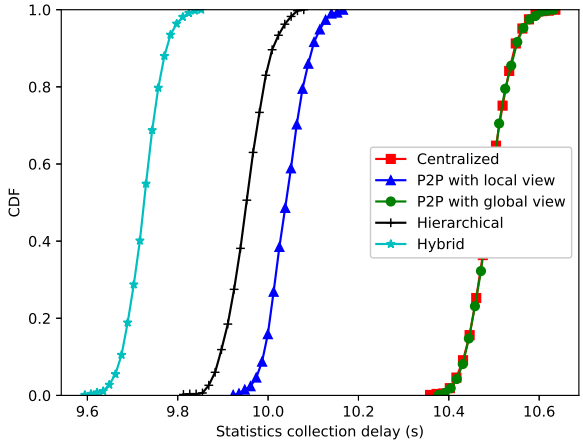
	Flow setup delay (ms)				Statistics collection delay (s)				Flow number (1e3)			
	Min	Max	Mean	STD	Min	Max	Mean	STD	Min	Max	Mean	STD
Centralized	2.19	3.15	2.33	0.12	10.34	10.64	10.49	0.043	61.17	65.62	64.75	0.51
P2P with local view	2.09	3.33	2.25	0.13	9.91	10.17	10.04	0.043	60.99	62.85	62.02	0.26
P2P with global view	2.09	3.07	2.24	0.13	10.36	10.63	10.49	0.043	61.41	65.67	64.76	0.48
Hierarchical	2.09	3.08	2.23	0.11	9.80	10.08	9.95	0.045	60.52	62.34	61.50	0.26
Hybrid	2.09	3.37	2.22	0.11	9.58	9.85	9.72	0.040	59.20	60.91	60.08	0.24

delay, and the number of flows in the flow tables are used to evaluate the performance of the control plane.

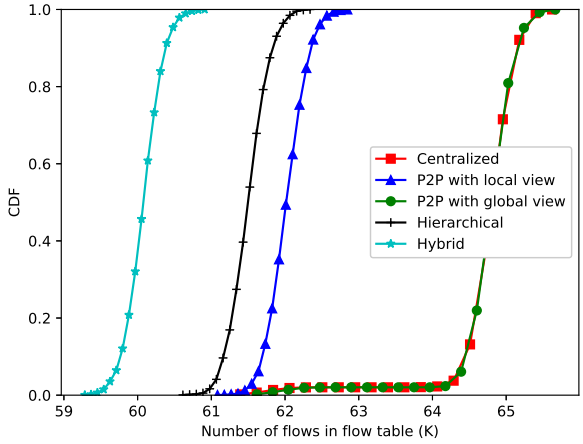
Figure 3.3 shows the performance of the five control planes with 100 switches (2000 hosts in total), $D_f = 8$, and $BW_c = 20Mbps$. And the minimum, maximum, mean, and standard deviation of the three metrics are shown in Table 3.2. We can see that all five control plane architectures have similar flow setup delay distributions. Moreover, among the five architectures, the centralized one achieves worst performance. Statistics collection delay and the number of flows share a similar distribution. This makes sense because the



(a)

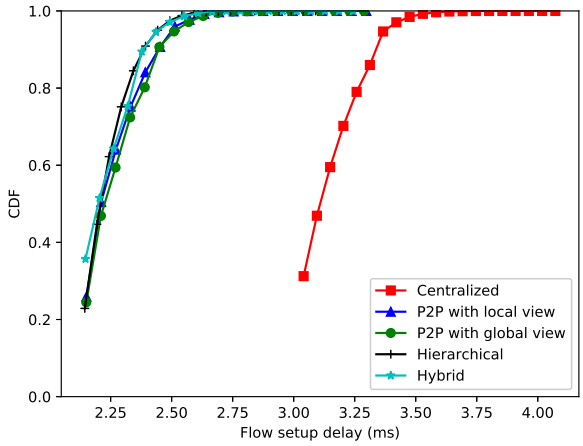


(b)

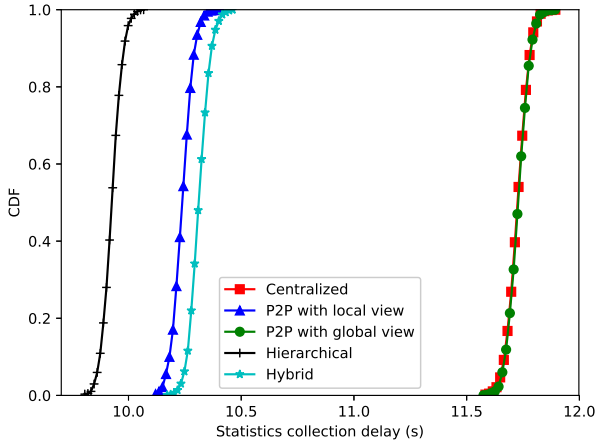


(c)

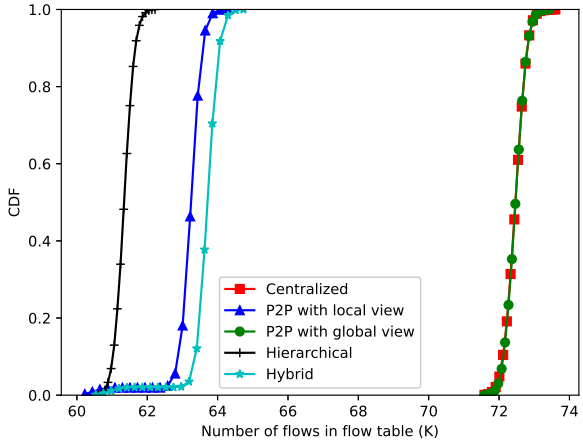
Figure 3.3: The performance of control planes with $D_f = 8$, $BW_c = 20Mbps$, and 100 switches.



(a)



(b)



(c)

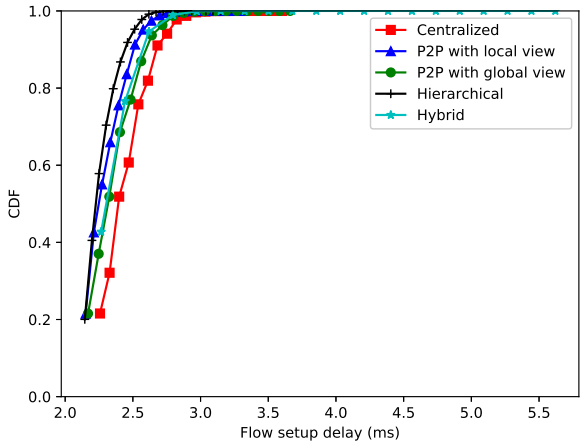
Figure 3.4: The performance of control planes with $D_f = 8$, $BW_c = 20Mbps$, and 300 switches.

Table 3.3: Descriptive statistics for the control planes with $D_f = 8$, $BW_c = 20Mbps$, and 300 switches

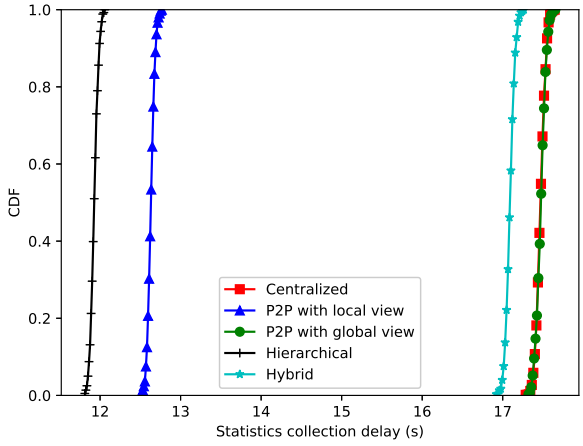
	Flow setup delay (ms)				Statistics collection delay (s)				Flow number (1e3)			
	Min	Max	Mean	STD	Min	Max	Mean	STD	Min	Max	Mean	STD
Centralized	2.99	4.07	3.14	0.13	11.58	11.89	11.73	0.046	71.48	73.60	72.46	0.27
P2P with local view	2.09	3.30	2.25	0.13	10.10	10.41	10.24	0.044	60.01	64.29	63.20	0.46
P2P with global view	2.09	3.29	2.26	0.14	11.56	11.89	11.73	0.045	71.49	73.44	72.47	0.26
Hierarchical	2.09	3.09	2.23	0.11	9.79	10.07	9.93	0.042	60.44	62.22	61.34	0.25
Hybrid	2.09	3.24	2.23	0.12	10.16	10.46	10.31	0.044	60.31	64.74	63.67	0.47

statistics pooling delay is proportional to the number of flow entries in the flow tables. The best architecture in terms of statistics collection delay and flow number is the hybrid control plane, followed by the hierarchical one and then the P2P with local view one, the centralized and P2P with global view control planes are the worst. Combining flow setup and statistic gathering, it is reasonable to say that the hybrid control plane achieves the best performance. When the simulated network scales from 100 switches to 300 switches (6000 hosts in total), as shown in Figure 3.4 and Table. 3.3, we can see that the flow setup delay in the centralized control plane is increased by 35%. Meanwhile, the other four control planes almost maintain the flow setup delay. In terms of statistics collection delay and flow number, the centralized and P2P with global view control planes again share similar distributions. These two control planes get 12% more flow entries and 12% longer statistics collection delay, compared with the case of 100 switches. Furthermore, the performance of the P2P with local view and hybrid control planes are also slightly degraded when the network scales from 2000 nodes to 6000 nodes. However, the hierarchical control plane maintains its performance regardless of the network size, and it outperforms the other four control planes. To summarize, the P2P with global view and centralized control planes have the worst scalability performance. In contrast, the hierarchical control plane has the best performance, which can keep the flow setup, statistics collection delay, and flow number almost unchanged when the network size scales from 2000 nodes to 6000 nodes.

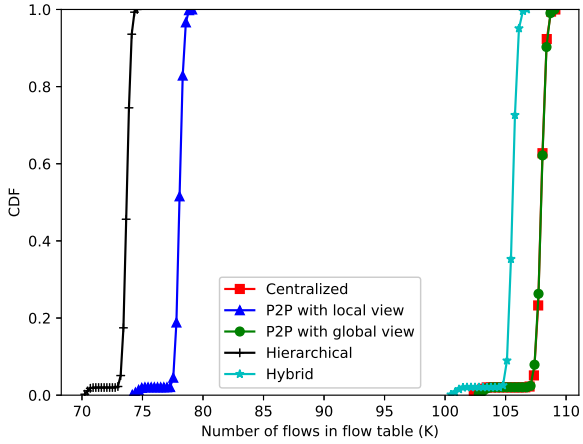
We then increase D_f from 8 to 14 (see Figure 3.5 and Table. 3.4), and try to evaluate



(a)

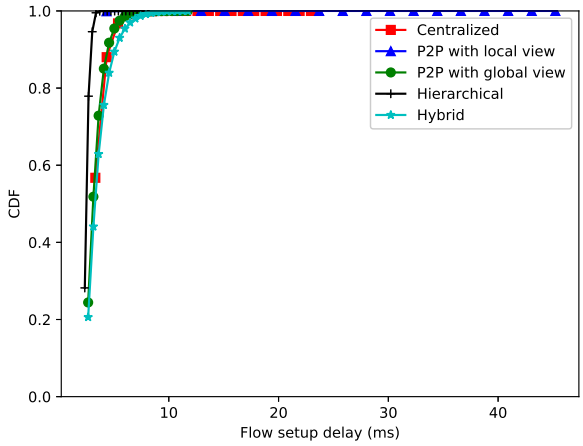


(b)

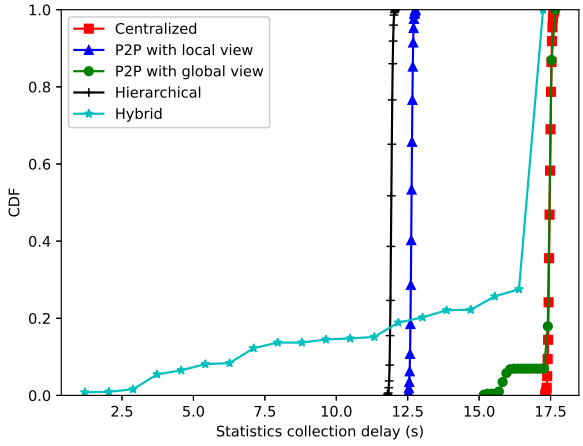


(c)

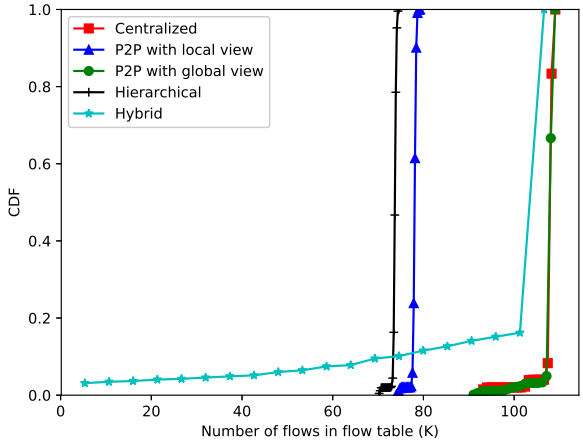
Figure 3.5: The performance of control planes with $D_f = 14$, $BW_c = 20Mbps$, and 100 switches.



(a)



(b)



(c)

Figure 3.6: The performance of control planes with $D_f = 14$, $BW_c = 10Mbps$, and 100 switches.

Table 3.4: Descriptive statistics for the control planes with $D_f = 14$, $BW_c = 20Mbps$, and 100 switches

	Flow setup delay (ms)				Statistics collection delay (s)				Flow number (1e3)			
	Min	Max	Mean	STD	Min	Max	Mean	STD	Min	Max	Mean	STD
Centralized	2.19	3.60	2.42	0.18	17.27	17.65	17.47	0.056	102.02	109.11	107.84	0.80
P2P with local view	2.09	3.31	2.28	0.16	12.50	12.77	12.63	0.044	73.90	79.11	78.00	0.57
P2P with global view	2.09	3.66	2.33	0.18	17.30	17.65	17.47	0.059	102.46	109.04	107.85	0.76
Hierarchical	2.09	3.14	2.26	0.12	11.80	12.06	11.93	0.047	70.01	74.57	73.63	0.53
Hybrid	2.09	5.62	2.32	0.17	16.90	17.26	17.09	0.055	100.16	106.77	105.49	0.73

Table 3.5: Descriptive statistics for the control planes with $D_f = 14$, $BW_c = 10Mbps$, and 100 switches

	Flow setup delay (ms)				Statistics collection delay (s)				Flow number (1e3)			
	Min	Max	Mean	STD	Min	Max	Mean	STD	Min	Max	Mean	STD
Centralized	2.27	22.95	3.37	0.86	17.30	17.63	17.47	0.055	102.15	109.09	107.84	0.80
P2P with local view	2.18	45.22	2.62	0.36	12.50	12.77	12.63	0.043	73.74	79.24	78.00	0.57
P2P with global view	2.18	2121	3.33	6.94	15.03	17.67	17.36	0.43	95.60	109.04	107.77	1.19
Hierarchical	2.01	8.81	2.52	0.27	11.77	12.08	11.93	0.044	70.00	74.60	73.64	0.53
Hybrid	2.18	23983	3.78	33.42	1.50	17.24	14.99	4.15	0	106.61	97.14	23.35

the effect of D_f on the scalability performance. We first examine the flow setup delay. As we can see, all five control planes have similar flow setup delay distributions. If we inspect carefully, we can find that hierarchical control plane achieves the best performance with regard to flow setup delay. This observation is also true for statistics collection delay and flow number. Compared with the case of $D_f = 8$, the performance of the centralized, the P2P with global view and the hybrid control planes' performance deteriorate severely, with $\geq 60\%$ more flow entries and longer statistics collection delay. The P2P with local view control plane outperforms these three ones, but loses to the hierarchical one. Therefore, the hierarchical control plane achieves the best scalability performance with regard to network diameter D_f.

Finally, we vary BW_c to investigate the influence of the bandwidth of the control channel on the scalability performance. Figure 3.6 and Table. 3.5 show the performance of the five control planes with 100 switches, $D_f = 8$, and $BW_c = 10Mbps$. Comparing with

Figure 3.3, we can see that all five control planes suffer from performance degradation. The maximum flow setup delay in the hybrid control plane is increased to be around 24 seconds. And some of the switches have no flow entry in their flow tables. This is because the large amount of `Packet_In`, `Multipart_Req`, `Multipart_Res`, and `Flow_Mod` messages congest some control channels, which makes the flow setup delay extremely intolerable. Furthermore, the congestion also prevents new flow entries being installed. On the other hand, more and more flow entries expire due to the timeout. Therefore, the measured minimum flow number reaches 0. This congestion is further demonstrated in the other three control planes: the centralized, the P2P with local view, and the P2P with global view. The maximum flow setup delays in these three control planes are increased by ≥ 7 times, and the average statistics collection delay and flow number by 40%. However, this congestion is not very serious in the hierarchical control plane and this control plane achieves the best performance.

Combining with all the simulation results above, we come to the following conclusions:

1. The hierarchical control plane has the best scalability performance and the P2P with local view achieve second-best performance, with regard to the size, diameter, and control channel bandwidth of the network.

2. The centralized and P2P with global view control planes have similar and the worst scalability performance.

3. The bandwidth of the control channel plays a significant role in the scalability performance. Given a small control channel bandwidth, all these five control planes get intolerable scalability performance. Therefore, increasing the control channel bandwidth (e.g., implementing strong management CPU in the switch, supporting high-frequency communication between the ASIC and the management CPU) is necessary for deploying a scalable SDN control plane.

4. The statistics collection delays in all the simulated networking scenarios are around

10 seconds, which is intolerable in practice. This long delay is due to two factors. One is the large number of flow entries in the flow table due to the scale of the network. Another one is slow statistics pulling speed, i.e., $K = 178us/flow$ which is set according to the empirical study of A. R. Curtis et al. [59]. Therefore, we can reduce the number of flows in the flow table and decrease K to make the statistics collection delay tolerable. To reduce the number of flow entries in the flow table, we can develop more intelligent flow table management algorithms which can remove the unnecessary flow entries timely. To decrease K, we can build faster hardware (e.g., use more powerful CPU in the switch) and develop new statistics pulling algorithms.

CHAPTER 4

SDNI-TE: AN EASTBOUND/WESTBOUND INTERFACE FOR DISTRIBUTED SDN CONTROL PLANE

The hierarchical control plane architecture is shown to have the best scalability performance in Chapter 3. Therefore, to deploy a large scale SDN OpenFlow network in one location, the hierarchical control architecture should be adopted. However, this architecture is not feasible if it is to be scaled up in an enterprise spread across different locations and geographies. In contrast, the second most scalable control architecture, the p2p with local view architecture (a.k.a., distributed control plane), can keep each SDN controller independent, with distinct policies and path setup to apply to network elements in its control. Furthermore, the distributed control plane allows gradual deployment and continuous evolution and enables flexible provisioning of the network. Therefore, the distributed control plane is more feasible for scaling across geographies. To deploy the distributed control plane, the key challenge is to design an eastbound/westbound interface such that peer SDN controllers can exchange control messages with each other. Specifically, what kind of information should be communicated? How the information should be organized? When these control messages should be exchanged? How these messages are generated and handled? The second piece of this work in [62] try to solve these problems for the distributed control plane by focusing on traffic engineering, one of the most important networking management applications.

4.1 Network Model and Problem Formulation

To define the eastbound/westbound protocol design problem in the distributed control plane, the network model used in this work is first presented. As shown in Figure 4.1, the network consists of multiple SDN domains, and each domain is controlled by one controller. In

every SDN domain, there are some border switches, which are directly connected to other SDN domains. For example, (0,0) is one of the border switches in SDN domain #0 (a switch/node $n_{i,s}$ is uniquely identified by a pair <SDN domain number i, switch number s >). The controller has the full information about the network it controls, including the switches, the links, and the applications. A link l is characterized by a pair (w_l, B_l), where w_l represents the weight of the link, and B_l is the capacity of the link. The weight of a link can be the delay, the number of hops, the number of concurrent connections, or any value assigned by network operators in order to change the routing behaviors. For example, in the Routing Information Protocol (RIP), the weight of every link is constant 1, which is the hop count. In the Open Shortest Path First (OSPF), network operators manipulate the weights of links to control traffic routing, which is the case for Border Gateway Protocol (BGP) as well. Based on the definition of link weight and bandwidth, we can get the weight of a path $\mathcal{P}$:

$$w_{\mathcal{P}} = \sum_{l \in \mathcal{P}} w_l, \tag{4.1}$$

and the bandwidth of a path $\mathcal{P}$:

$$B_{\mathcal{P}} = min\{B_l : l \in \mathcal{P}\}. \tag{4.2}$$

Furthermore, the controller also knows its neighboring SDN domains, the switches directly connected in the neighboring SDN domains, and the external links. For example, besides the domain it controls, the controller in SDN domain #0 also knows the switches (1,0), (1,1), (2,0), and (2,1), as well as the links between these switches and its own border switches.

The controller knows not only link bandwidth and weight, but also can track network dynamics such as link load and application state. Do SDN domains need to exchange this dynamic information? This information is definitely beneficial for many applications including traffic engineering. For example, link load can be used to avoid link congestion.

Application information can be helpful to deliver services with the required QoS. However, the load of the link can fluctuate from empty to fully loaded. And the applications can start and end at any time. To exchange this information between neighboring SDN domains, certain measurements are required. On one hand, it would increase the control overhead between the controller and switches and may thus lead to unpredictable instability [63]. On the other hand, the timeliness of this exchanged information would be difficult to predict. It is highly possible that the controller makes some decisions based on some invalid information received from the neighboring SDN domains, which will definitely deteriorate the network performance. Taking into these factors account, it is more practical to only exchange static network information (i.e., link weight and link bandwidth) but not dynamic network and application information[1] in this work.

Based on the above discussions, the problem of designing the eastbound/westbound interface for distributed SDN control plane can be defined as the following:

Given:

- N_c: the number of independent SDN domains
- The controller in the SDN domain i has the following information:
  - $G^i = (\mathcal{N}^i, \mathcal{L}^i, \mathcal{B}^i, \mathcal{W}^i)$: the topology of the SDN switches in the domain, where $\mathcal{N}^i$ is the set of all SDN switches, $\mathcal{L}^i$ is the set of all links between switches, and $\mathcal{B}^i$ and $\mathcal{W}^i$ are the bandwidths and weights of the links.
  - $\mathcal{A}^i$: the IDs of its neighboring SDN domains
  - $\mathcal{C}^{i,j} = \{n_{i,s} \in \mathcal{N}^i\}$: the border switches directly connected to its neighboring domain $j (j \in \mathcal{A}^i)$
  - $\mathcal{O}^{i,j} = \{n_{j,s} \in \mathcal{N}^j\}$: the border switches of its neighboring domains $j (j \in \mathcal{A}^i)$ directly connected with its own border switches

[1] Of course, we can exchange some of this dynamic information if they are valid for a relatively long time. For example, some multimedia applications (e.g., video game, online movie) are expected to last for tens of minutes.

- $\mathcal{E}^{i,j} = \{(w_{l_{p,q}}, B_{l_{p,q}}) : p \in \mathcal{C}^{i,j}, q \in \mathcal{O}^{i,j}\}$: the external links which are used to directly connected its neighboring domain $j (j \in \mathcal{A}^i)$
- The learned reachability information from its neighboring domains

$$\mathcal{R}^i = \{r^k_{j,p,z,q} = (n_{j,p}, n_{z,q}, w^k_{n_{j,p},n_{z,q}}, B^k_{n_{j,p},n_{z,q}}) : j \in \mathcal{A}^i, n_{j,p} \in \mathcal{O}^{i,j}, z \neq i, k \leq K\}, \tag{4.3}$$

and the information $r^k_{j,p,z,q}$ means the kth ($k = 1, 2, ..., K$) path from $n_{j,p}$ to $n_{z,q}$ has bandwidth $B^k_{n_{j,p},n_{z,q}}$ and the weight of the path is $w^k_{n_{j,p},n_{z,q}}$. Moreover, the controller learns at most K paths from $n_{j,p}$ to $n_{z,q}$.

Variables:

- $\mathcal{F}^{i,j} = f^{i,j}(\cdot)$: the information SDN domain i advertises to the domain j. $f^{i,j}$ is a function of network policies and information gathered by the controller i, including $G^i, \mathcal{R}^i, \mathcal{C}^{i,z}, \mathcal{O}^{i,z}, \mathcal{E}^{i,z} (z \in \mathcal{A}^i)$

Constraints:

For the domain i, $\forall n_{j,p} \in \mathcal{O}^{i,j}, \forall n_{z,q} (z \neq i)$

- Reachability: if there exists a path from $n_{j,p}$ to $n_{z,q}$, the controller i must learn that $n_{z,q}$ can be reached from $n_{j,p}$ through the advertised information.
- Validity: if the maximal flow from $n_{j,p}$ to $n_{z,q}$ is $F_{n_{j,p},n_{z,q}}$ and the learned bandwidth of the learned path from $n_{j,p}$ to $n_{z,q}$ is $B_{n_{j,p},n_{z,q}}$, then $B_{n_{j,p},n_{z,q}} \leq F_{n_{j,p},n_{z,q}}$

Objects:

- Minimize the overall signaling overhead

$$min \sum_{i=1}^{N_c} \sum_{j \in \mathcal{A}^i} |\mathcal{F}^{i,j}| \tag{4.4}$$

- Maximize learned path bandwidths

$$max \sum_{i}^{N_c} \sum_{j \in \mathcal{A}^i} \sum_{n \in \mathcal{O}^{i,j}} \sum_{n_{z,p}: z \neq i} B_{n,n_{z,p}} \tag{4.5}$$

4.2 BGP-Addpath

A naive solution to the above problem is to advertise all information a controller has to its neighbors. That is

$$\mathcal{F}^{i,j} = (G^i, \mathcal{R}^i, \mathcal{C}^{i,z}, \mathcal{O}^{i,z}, \mathcal{E}^{i,z}, \forall z \neq j). \tag{4.6}$$

Note it is not necessary to advertise $(\mathcal{C}^{i,j}, \mathcal{O}^{i,j}, \mathcal{E}^{i,j})$ to the domain j because this information is inherently shared between the domain i and domain j according to our assumption. This simple solution fully expose the internal topology of domain i to other domains. It will cause great concerns about privacy and security, especially when the connected neighboring domain cannot be trusted. For example, malicious domains can execute advanced denial-of-service attacks such as link-flooding attacks [64].

To overcome the above problem, BGP only advertises reachability information[2], i.e.,

$$\mathcal{F}^{i,j} = \{r^1_{i,p,z,q} : n_{i,p} \in \mathcal{C}^{i,j}\}, \tag{4.7}$$

where only one path to a specific destination can be advertised. In other words, $K = 1$. Note $r^1_{i,p,z,q}$ is not directly retrieved from the set $\mathcal{R}^i$. Instead, it is computed by the network policy in the domain i based on its own topology and the learned information, which is

$$r^1_{i,p,z,q} = \mathcal{P}(G^i, \mathcal{R}^i, \{\mathcal{C}^{i,x}, \mathcal{O}^{i,x}, \mathcal{E}^{i,x}, \forall x \neq j\}, p, z, q). \tag{4.8}$$

BGP achieves great success since it was introduced. However, in the context of SDN, the

[2]BGP advertises different attributes of a path such as `LOCAL_PREF`, `MULTI_EXIT_DISC`, and `Community`. These attributes help to select the "best" path defined by the configured policy. Here, we abstract all these attributes as one single weight value and the best path achieves the smallest weight.

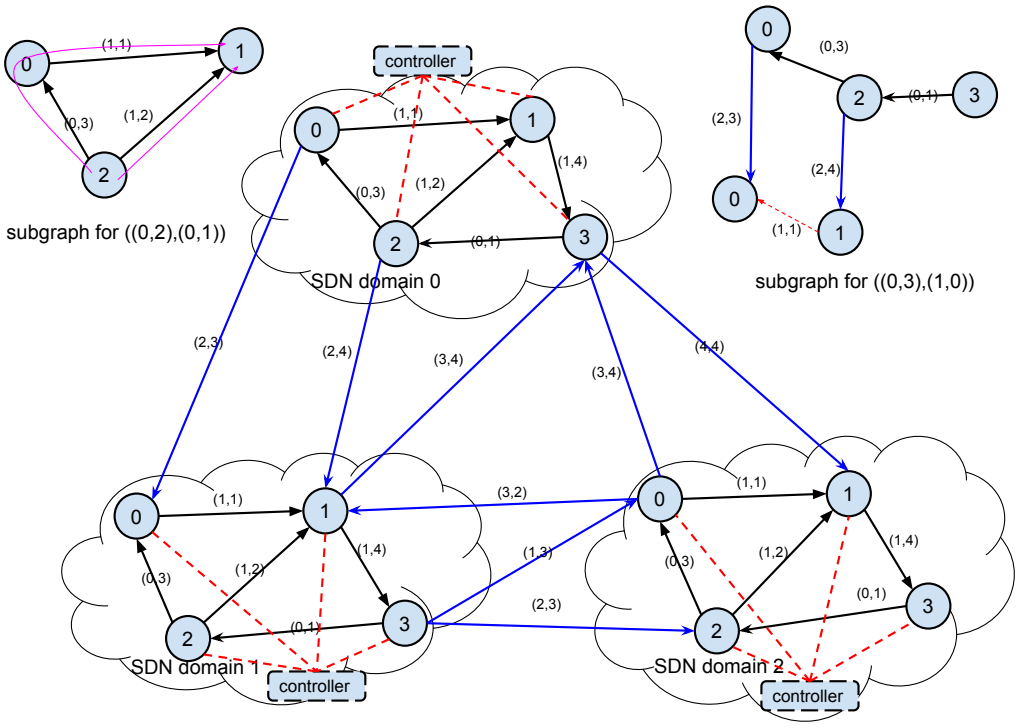


Figure 4.1: An example topology of SDNi-TE with three SDN domains.

controller knows much more information than individual router in traditional distributed networks. For example, the controller knows all paths and their attributes to every destination. This is particularly beneficial for traffic engineering because the controller needs to learn multiple paths to some destination such that the traffic can be split among these paths in order to increase link utilization and minimize link congestion. Therefore, a straightforward improvement to BGP is to advertise the k shortest paths to neighboring domains instead of just one, which is referred as BGP-Addpath [65]:

$$\mathcal{F}^{i,j} = \{r^{k}_{i,p,z,q} : n_{i,p} \in \mathcal{C}^{i,j}, k \leq K\}. \tag{4.9}$$

4.3 SDNi-TE Protocol

Although it is straightforward to extend BGP to BGP-Addpath such that multiple paths for the same address prefix can be advertised, it suffers from the following three problems:

1. The message overhead is almost increased by K times. Generally speaking, there is no direct physical link between controllers between neighboring domains. Otherwise, the cost will be very expensive. Controllers in the distributed SDN OpenFlow networks communicate with each other through the external links between neighboring domains, which means the controller needs to send the message to OpenFlow switches, which then relay to neighboring switches, and finally to the neighboring controller. As discussed in Section 1.1, OpenFlow switches are not powerful in terms of computation ability, memory size, and bus bandwidth. The greatly increased message overhead makes matter worse, and can easily saturate the control channel.

2. Table size is also increased by nearly K times. As reported by G. Huston [66], full internet BGP forwarding table in AS65000 now contains over 767,000 forwarding information base (FIB) records, which are placed in fast but very expensive memory such as SRAM, TCAM, and RLDRAM. In addition, the size of the routing table also reaches over 1.5 million, which requires few gigabytes RAM memory. If these numbers are increased by K times, it will be extremely expensive to hold and process these massive entries.

3. The advantages of SDN are not exploited at all. BGP-Addpath is an extension of BGP, which is developed in the context of traditional networking. With SDN, the only possible optimization is to move the routing decision from switches to the controller. SDN itself cannot add more value to BGP-Addpath. For example, although the SDN controller knows the full topology of the network in its control, the operation of BGP-Addpath is not affected.

For example, in Figure 4.1, with BGP-Addpath, the controller in SDN domain #0 will propagate $(0,3) \rightarrow (0,2) \rightarrow (0,1)$ and $(0,3) \rightarrow (0,2) \rightarrow (0,0) \rightarrow (0,1)$ to the controller in SDN domain #1 through the border switch (0,3). However, for the traffic flow destined for (0,1), the controller in the SDN domain #1 only concerns how much traffic it can send to the switch (0,3) and what the corresponding cost is. And it does not care whether these traffic will be delivered to (0,1) through 2 or 3 paths in SDN domain #0. Therefore, we can aggregate the paths to one specific destination and advertise the aggregated path to the neighboring SDN domains, in order to reduce the overhead and table size.

4.3.1 Path Aggregation

Aggregation was introduced to BGP since BGP-4 [67], which aggregates routes to individual prefixes as one route to a less-specific prefix. For example, instead of advertising 16 individual prefixes with a length of 24 (e.g., 10.1.1.0/24, 10.1.2.0/24), routing aggregation allows to only advertise one aggregated prefix 10.1.0.0/20. In this way, the number of prefixes announced can be greatly reduced. However, routing aggregation does not allow routes with different `MULTI_EXIT_DISC` attributes to be aggregated. In other words, only routes with equal weights can be aggregated. To summary, path aggregation differs from routing aggregation in the following two ways:

1. Path aggregation aggregates multiple paths with the same destination address with the help of SDN, while routing aggregation aggregates multiple routes with different destination address as a route to a less-specific address.

2. Path aggregation aggregates paths with different weights/costs, while routing aggregation only handle routes with the same weight.

Then how to do path aggregation? For SDN domain i, it only cares how much traffic to a specific destination should be injected to switch $n_{j,s} \in \mathcal{O}^{i,j} (j \in \mathcal{A}^i)$ and its cost. Therefore, we can apply the max-flow algorithm to compute the aggregated bandwidth and weight. We

should add that it is impossible to do path aggregation in the traditional networking because routers only learn reachability information but not the topology.

4.3.2 Routing Loop

Routing loop is a common problem in the distributed routing network. In BGP, *AS path* is adopted to address this problem. *AS path* is the list of ASes that a route goes through to reach its destination. And route loop can be detected and avoided by checking whether its own AS number is in the *AS path* received from the neighboring ASes. In the distributed SDN, we also need *domain path* to detect and avoid routing loop. However, the situation in the distributed SDN is different because aggregated paths are advertised to the neighbors. For example, the controller in SDN domain #1 learns that (1,1) can reach (2,1) through $(1,1) \rightarrow (0,3) \rightarrow (2,1)$ and $(1,1) \rightarrow (1,3) \rightarrow (2,2) \rightarrow (2,1)$. The first path goes through SDN domain #1, #0, and #2. And the later path goes through SDN domain #1 and #2. When the two paths are aggregated as one path which would be advertised to SDN domain #0, what should the *domain path* be? Simply, we have two options, i.e., intersection and union. In the case of union operation, the *domain path* of the aggregated path is $\{2,1,0\}$. Thus the controller in SDN domain #0 will discard this path since its own domain number is in the *domain path*. In this case, the controller would not know that it can reach (2,1) through (1,1). For the other case (intersection operation), the *domain path* is $\{2,1\}$. The controller in SDN domain #0 would know the path from (1,1) to (2,1), but it is possible that the traffic would be routed back to SDN domain #0. Of course, we can also simply discard the path which can result in routing loop during path aggregation, and then apply union operation to generate the *domain path*. However, this can still lead to routing loop. For example, suppose there are three routes to the sink s from the switch m in SDN domain #0, the *domain path* of these three routes are $\{1,0,2\}$, $\{1,0,3,4\}$, and $\{0,2,3\}$ respectively. We aggregate these paths as one path which will be advertised to SDN domain #4. Therefore, the route with *domain path* $\{1,0,3,4\}$ is discard. The rest two routes are aggregated as

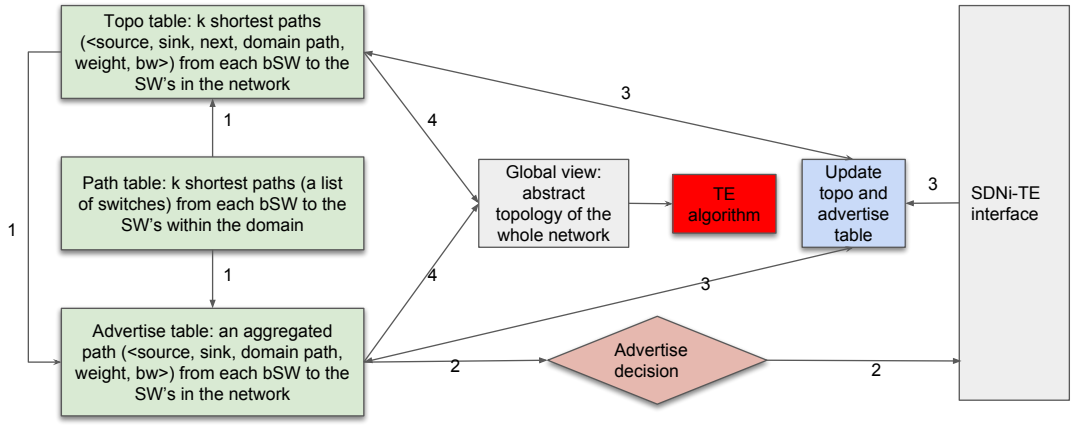


Figure 4.2: SDNi-TE protocol.

one path, whose *domain path* is {0,2,4}. Suppose SDN domain #4 further advertise this aggregated path to SDN domain#3, then there will be a route from SDN domain #3 to #4 to #0 to #2, and back to SDN domain #3. Fortunately, the concept of routing loop in the distributed SDN is slightly different from BGP. In BGP, we detect route loop based on the assumption that AS loop implies route loop. However, in the distributed SDN, route loop can be detected and avoided in a more precious way because the controller has the full information about the network it controls, and domain loop does not necessarily imply route loop. For example, the route $(0,0) \rightarrow (1,0) \rightarrow (1,1) \rightarrow (0,3) \rightarrow (2,1)$ is a domain loop but not route loop. Even if route loop occurs, the controller can check the flow table of the switch to detect it and notifies its neighboring SDN domains. Therefore, in path aggregation, we can simply apply intersection operation to generate the *domain path*.

4.3.3 Information Base

Based on the above discusses, SDNi-TE protocol, as shown in Figure 4.2, is elaborated in this section. In SDNi-TE protocol, all TE-related information is contained in three tables: path table, topology table, and advertisement table. Path table stores all the k-shortest paths

(represented as a list of switches) from each border switch in the SDN domain to all the other switches in the same domain. Obviously, as long as the topology of the SDN domain is not changed, the path table is static and thus can be computed offline. To compute this table, multiple algorithms can be applied. In this work, Yen's algorithm [68] is adopted. When the network topology is updated (e.g., some switches are down, certain link is disconnected, a new switch is added), we can simply apply Yen's algorithm to recompute the paths and update the path table. If the topology is very large, then an incremental k shortest path algorithm can be applied [69], which is out of the scope of this work.

Topology table contains two kinds of paths: the k shortest local paths from the border switches to the switches in the local domain and the k shortest external paths from the border switches to the switches in other domains. Local paths are originated in the local domain and the external paths are directly learned from the peering domains. Different from the path table, the path in the topology table is represented as a tuple *(source, sink, next, domain path, weight, bandwidth)*, where *source* is the source of this path, *sink* is the destination of this path, *next* is the next switch from the source in this path, *domain path* is the set of the SDN domains which the path goes through, *weight* is the sum of the weights of the links in this path, and *bandwidth* is the minimum bandwidth of the links in the path. If the path contains no link (i.e., *source=sink*), *weight=0* and *bandwidth*=∞. If *source* = *next*, the path is a local path. Otherwise, it is a cross-domain path. Based on the topology table, we can build the advertisement table, which is exchanged between neighboring SDN domains through SDNi-TE messages[3]. In the advertisement table, there is one and the only one aggregated path to every reachable switch in the whole network for each border switch in the SDN domain. The representation of the path is as same as the one in the topology table, except that it does not include *next*. With these tables, the controller in the SDN domain can run any traffic engineering algorithm to allocate the bandwidth to competing applications using multiple paths [32], [70].

[3]The format of SDNi-TE messages is one of the future works. Right now, the readers can simply regard the message as one entry in the advertisement table.

4.3.4 Protocol Operation

SDNi-TE protocol has four stages: (1) *Initialization* initializes the three tables in the controller, (2) *Advertisement* decides whether one specific entry in the advertisement table should be advertised to the neighbors, (3) *Update* deals with how to update the three tables when an SDNi-TE message is received, (4) *Global View Construction* builds a global view of the whole network based on the learned routes from the peering domains. In the following, this protocol is described through a concrete example. Figure 4.1 shows an example topology with three SDN domains, where each domain has four switches. For simplicity, we assume the links between switches are directed. And each of these links is labeled with a *(weight, bandwidth)* pair.

Initialization

In *Initialization*, every controller initializes its own tables. For the path table, the controller applies Yen's algorithm to compute the k-shortest paths from each border switch to every local switch and insert the paths into the path table, where the path is represented as a list of switches. For example, let's set k as 2. Then the initialized path table in SDN domain 0 contains 15 entries such as *(0,0)→(0,0), (0,2)→(0,1)*, and *(0,2)→(0,0)→(0,1)*. Based on these paths, the controller in SDN domain i can further initialize the topology table. For every path in the path table, insert *(source, sink, source,* $\{i\}$*, weight, bandwidth)* into the topology table. Therefore, the initialized topology table in SDN domain 0 also contain 15 entries such as $\{$*(0,0),(0,0),(0,0),*$\{0\}$*,0,*$\infty\}$, $\{$*(0,2),(0,1),(0,2),*$\{0\}$*,1,2*$\}$, and $\{$*(0,2),(0,1),(0,2),*$\{0\}$*,1,1*$\}$. After initializing the topology table, the controller can then do the initialization for the advertisement table. For each *(source, sink)* pair in the topology table, extract the corresponding paths (at most k) in the path table to construct a subgraph, and compute the maximum flow f^* of this subgraph from *source* to *sink*. For the obtained maximum flow, insert *(source, sink,* $\{i\}$*, w(*f^**), s(*f^**))* into the advertisement table, where $s(f^*)$ is the size of f^*, $w(f^*) = \frac{\sum_e w_e f_e}{s(f^*)}$, w_e is the weight of the link e, and f_e is the

flow on the link e. If the subgraph only contains one node (i.e., *source=sink*), $w(f^*) = 0$ and $s(f^*) = \infty$. In our example, for ((0,2), (0,1)) pair, extracting the paths *(0,2)→(0,1)* and *(0,2)→(0,0)→(0,1)*, we can get a subgraph as shown in Figure 4.1. In this subgraph, the size of the maximum flow from switch (0,2) to switch (0,1) is 3, and the weight is also 3. Therefore, we insert *((0,2),(0,1),{0},3,3)* into the advertisement table. For ((0,0), (0,0)) pair, the inserted entry is *((0,0),(0,0),{0},0,∞)*. In total, there are 12 entries in the advertisement table after the initialization.

Advertisement

After *Initialization*, the neighboring controller can exchange their advertisement tables. There are lots of factors which may be involved when a controller decides whether to advertise one specific piece of information to the neighboring SDN domains. For example, business relationships between peering domains may keep a controller from advertising full routes to a specific domain [71]. In this work, only the routing loop problem is considered, and more complicated policies can be implemented in the future. To avoid routing loop, the controller will check whether the neighboring SDN domain is in the *domain path* of the entry in the advertisement table. If no, it can advertise this entry to the neighbor. In our example, SDN domain #0 can propagate the whole initialized advertisement table to domain #1 and domain #2 because none of these two SDN domains are in the *domain path*.

Update

The final component is to update the tables when an SDNi-TE message is received. Suppose the controller in SDN domain j receives the message $(n_{i,p}, n_{z,s}, \mathcal{DP}_{n_{i,p},n_{z,s}}, w_{n_{i,p},n_{z,s}}, B_{n_{i,p},n_{z,s}})$ from SDN domain i, which indicates there is a path from $n_{i,p}$ to $n_{z,s}$ through the domains in the domain path $\mathcal{DP}_{n_{i,p},n_{z,s}}$ with weight $w_{n_{i,p},n_{z,s}}$ and its bandwidth is $B_{n_{i,p},n_{z,s}}$. Let's say the border switch $n_{j,q}$ in SDN domain j is directly connected to switch $n_{i,p}$ in domain i, the controller in SDN domain j first updates its topology table: if there exists an entry

($n_{j,q}$, $n_{z,s}$, $n_{i,p}$, $\mathcal{DP}^{n_{i,p}}_{n_{j,q},n_{z,s}}$, $w^{n_{i,p}}_{n_{j,q},n_{z,s}}$, $B^{n_{i,p}}_{n_{j,q},n_{z,s}}$) in the topology table, delete this entry and insert ($n_{j,q}$, $n_{z,s}$, $n_{i,p}$, $\mathcal{DP}_{n_{i,p},n_{z,s}} \cup \{\text{j}\}$, $w_0 + w_{n_{i,p},n_{z,s}}$, min$\{B_{n_{i,p},n_{z,s}}, B_0\}$) into the topology table. Otherwise, just insert the new entry. This update can ensure that any two learned paths in the topology table cannot have the same *source*, *sink*, and *next*.

After updating the topology table, the controller in SDN domain j should then update its advertisement table as following:

1. If no entry in the advertisement table is with the destination $n_{z,s}$, there must be one and only one entry *($n_{j,h}$, $n_{j,q}$, $\{j\}$, w', B')* for each border switch $n_{j,h}$ in the advertisement table. Insert the entry *($n_{j,h}$, $n_{z,s}$, $\mathcal{DP}_{n_{i,p},n_{z,s}} \cup \{j\}$, $w' + w_0 + w_{n_{i,p},n_{z,s}}$, min$\{B', B_0, B_{n_{i,p},n_{z,s}}\}$)* into the advertisement table.

2. If there are entries with destination $n_{z,s}$ in the advertisement table, then for every border switch $n_{j,h}$ in SDN domain j,

   2.1) Find the K shortest path from $n_{j,h}$ to $n_{z,s}$

   i. In the topology table, for every learned path from the border switch $n_{j,a}$ to $n_{z,s}$, $E^{n_{x,b}}_{n_{j,a},n_{z,s}}$ = *($n_{j,a}$, $n_{z,s}$, $n_{x,b}$, $\mathcal{DP}^{n_{x,b}}_{n_{j,a},n_{z,s}}$, $w^{n_{x,b}}_{n_{j,a},n_{z,s}}$, $B^{n_{x,b}}_{n_{j,a},n_{z,s}}$)*, there are at most K entries with source $n_{j,h}$ and destination $n_{j,a}$ in the topology table, say $E^{n_{j,h},k}_{n_{j,h},n_{j,a}}$ = *($n_{j,h}$, $n_{j,a}$, $n_{j,h}$, $\{j\}$, $w^{n_{j,h},k}_{n_{j,h},n_{j,a}}$, $B^{n_{j,h},k}_{n_{j,h},n_{j,a}}$) ($k \leq K$)*. Combine any $E^{n_{x,b}}_{n_{j,a},n_{z,s}}$ and $E^{n_{j,h},k}_{n_{j,h},n_{j,a}}$, we can get a path from $n_{j,h}$ to $n_{z,s}$, which also goes through switch $n_{j,a}$ and $n_{x,b}$. The weight of this path is $w^{n_{x,b}}_{n_{j,a},n_{z,s}} + w^{n_{j,h},k}_{n_{j,h},n_{j,a}}$, the bandwidth is min$\{B^{n_{x,b}}_{n_{j,a},n_{z,s}}, B^{n_{j,h},k}_{n_{j,h},n_{j,a}}\}$, and the domain path is $\mathcal{DP}^{n_{x,b}}_{n_{j,a},n_{z,s}} \cup \{j\}$.

   ii. Among all the generated paths from $n_{j,h}$ to $n_{z,s}$, select the K smallest weight paths: $\mathcal{P}^1_{n_{j,h},n_{z,s}}$,...,$\mathcal{P}^K_{n_{j,h},n_{z,s}}$.

   2.2) Extract the selected K paths (including the paths from $n_{j,h}$ to $n_{j,a}$ extracted from the path table, the edge $(n_{j,a}, n_{x,b})$, and the edge $(n_{x,b}, n_{z,s})$) to form a subgraph. In this subgraph, compute the maximum from f^* from $n_{j,h}$ to $n_{z,s}$.

2.3) If there is an entry with source $n_{j,h}$ and destination $n_{z,s}$, delete this entry

2.4) For the computed maximum flow, insert $(n_{j,h}, n_{z,s}, \mathcal{DP}_{n_{j,h},n_{z,s}}, w(f^*), s(f^*))$ into the advertisement table, where $\mathcal{DP}_{n_{j,h},n_{z,s}} = \bigcap_{k=1,\ldots,K} \mathcal{DP}^K_{n_{j,h},n_{z,s}}$, $w(f^*)$ is the weight of the maximum flow, and $s(f^*)$ is the size of f^*.

For example, after initialization, SDN domain #0 receives the first the entry *((1,0),(1,0),{1},0,∞)* which is from SDN domain #1. At this moment, there is no entry with destination (1,0) and source (0,0) in the topology table in SDN domain #0. Therefore, the entry *((0,0),(1,0),(1,0),{1,0},2,3)* is inserted into the topology table. Since no entry with sink (1,0) is in the advertisement table and there are three entries with destination (0,0) in the advertisement table, i.e., *((0,0),(0,0),{0},0,∞)*, *((0,2),(0,0),{0},0,3)*, and *((0,3),(0,0),{0},0,1)*, the controller in SDN domain #0 will insert *((0,0),(1,0),{1,0},2,3)*, *((0,2),(1,0),{1,0},2,3)*, and *((0,3),(1,0),{1,0},2,1)* into the advertisement table. Suppose SDN domain #0 then receives *((1,1),(1,0),{1},1,1)* from SDN domain#1 through the switch (0,2). At this point, no entry with source (0,2) and destination (1,0) exits in the topology table. Thus, *((0,2),(1,0),(1,1),{1,0},3,1)* is inserted into the topology table. Since there are entries with destination (1,0) in the advertisement table, the protocol searches all the corresponding entries in the topology table, namely *((0,2),(1,0),(1,1),{1,0},3,1)* and *((0,0),(1,0),(1,0),{1,0},2,3)*. Then for the border switch (0,3), there is one path from (0,3) to (0,0) and (0,2) respectively. Extracting these paths, we have a subgraph as shown in Figure 4.1. In this graph, the size of the maximum flow from (0,3) to (1,0) is 1, and the cost is 2. Therefore, the entry *((0,3),(1,0),{1,0},2,1)* is inserted into the advertisement table. After several rounds of exchanging messages, the tables in the controller will be stable.

Global View Construction

With the stable tables, the controller needs to build the global view about the whole network. Obviously, the controller has the whole information about the domain in its control, e.g., switches, link capacity, link weight. In addition, as discussed in Section 4.1, the con-

troller knows the external switches directly connected and the external links. Suppose this information is abstracted as $G = (V, E, C, W)$, where V is the set of the switches (including the switches which is directly connected in the neighboring SDN domains), E is the set of the links (including the links between neighboring SDN domains), $C(e)$ and $W(e)$ is the bandwidth and weight of the link e respectively. Combined G and the topology table, the controller can construct a graph $G^a = (V^a, E^a, C^a, W^a)$ to abstract the topology of the whole network as following:

1. G is a subgraph of G^a ($\forall v \in V, v \in V^a$;$\forall e \in E, e \in E^a, C^a(e) = C(a), W^a(e) = W(a)$). In addition, we set $V_{sink} = \varnothing$

2. Search the topology table to find the entries where *next* is not in the SDN domain (i.e., non-local paths). For each such entry, say *(s, d, n, {domain path}, w, b)*

   2.1) if $d \notin V_{sink}$, $V_{sink} \leftarrow V_{sink} \cup \{d\}$, $V^a \leftarrow V^a \cup \{d\}$.

   2.2) $E^a \leftarrow E^a \cup \{(n, d)\}$, where $W^a((n, d)) = w - W^a((s, n))$, $C^a((n, d)) = min\{b, C^a((s, n))\}$

In our example, the abstract topology of the whole network for SDN domain #0 is shown in Figure 4.3. In this figure, we use *domain number/switch id* to label the node for convenience. For example, 0/1 represents the switch (0,1). Once the controller generates the abstract topology, it can apply any choosing algorithm for traffic engineering based on this global view.

4.3.5 Comparison between SDNi-TE and BGP-Addpath

The main difference between SDNi-TE and BGP-Addpath is that the advertised k-shortest paths are aggregated into one path. In this part, we will compare the two protocols from the perspective of table size, message overhead, and computation overhead.

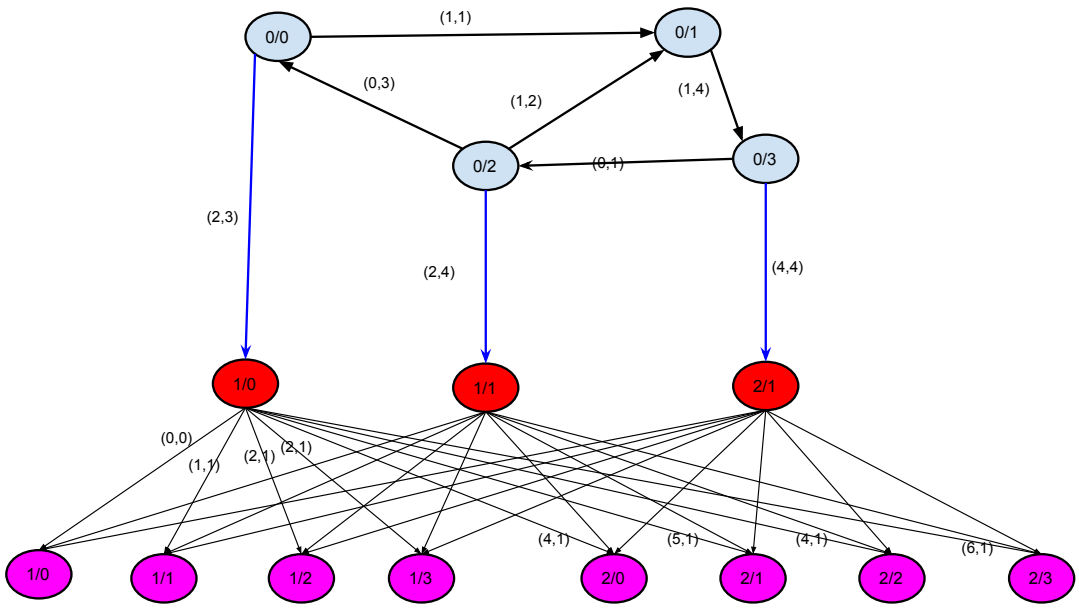


Figure 4.3: The abstract topology generated in SDN domain #0. Only parts of the edges' weights and bandwidths are shown for simplicity.

Table Size

Suppose the number of border switches in an SDN domain is N_b, the number of switches in the whole network is M, and the number of domains in the network is N_c. In addition, let α be the average number of border switches in other SDN domains for one border switch is connected to. Furthermore, we assume that the whole network is connected, i.e., there is a path between every pair of switches.

- **Topology table**: For SDNi-TE, the topology table first contains the K shortest paths between its border switches and all the switches in the same domain, which is $KN_bM/N_c - (K-1)N_b$. In addition, for every directly connected border switch in the neighboring domain, an aggregated path to each reachable address originated from it will be advertised, which in total account for $\alpha N_b(M - M/N_c)$. In the case of BGP-Addpath, the difference is that K paths from each border switch to every reachable address will be advertised to neighboring domains. Therefore, the topology table size will be $KN_bM/N_c - (K-1)N_b + \alpha KN_b(M - M/N_c)$.

- **Advertisement table**: For SDNi-TE, the advertisement table size is $N_b M$. As for BGP-Addpath, since K paths are allowed to be advertised to neighbors, the size of the advertisement table will be $K N_b M$.

In summary, the ratio of the table size in SDNi-TE to BGP-Addpath is

$$\frac{N_b M + K N_b M/N_c - (K-1)N_b + \alpha N_b(M - M/N_c)}{K N_b M + K N_b M/N_c - (K-1)N_b + \alpha K N_b(M - M/N_c)} \approx \frac{\alpha + 1 + (K-\alpha)/N_c}{(\alpha+1)K + (K - \alpha K)/N_c} \tag{4.10}$$

Message Overhead

The message overhead is directly related to the size of the advertisement table since all entries in the table will be advertised to neighboring domain except for those which may result in routing loop. Therefore, the message overhead of BGP-Addpath is approximately K times to that of SDNi-TE.

Computation Overhead

At the `Initialization` stage, both SDNi-TE and BGP-Addpath need to apply the k-shortest path algorithm to compute the k-shortest paths between any border switch and all the other switches in the same domain. In this work, Yen's algorithm is applied, thus the computation overhead is $O(K N_b |V|^2(|E| + |V|\log(|V|)))$, where $|V| = M/N_c$ is the number of switches in the domain and $|E|$ is the number of edges. Besides the topology table, the advertisement table should also be initialized. For SDNi-TE, it needs to apply max-flow algorithm to compute the maximal flow between any border switch and all the other local switches, which requires $O(N_b|V|^2|E|^2)$ computation overhead (using EdmondsKarp algorithm to compute max-flow). Since the time complexity of Yen's algorithm dominates that of EdmondsKarp algorithm, the total computation overhead for *Initialization* is $O(K N_b |V|^2(|E| + |V|\log(|V|)))$. As for BGP-Addpath, the advertisement table is the same as the topology table at the `Initialization` stage, thus no extra computa-

tion overhead is required. Therefore, both SDNi-TE and BGP-Addpath requires almost the same computation overhead, which is $O(KN_b|V|^2(|E| + |V|\log(|V|)))$.

At the `Advertisement` stage, both SDNi-TE and BGP-Addpath directly advertise all satisfied entries in the advertisement table to neighboring domains. Therefore, the computation overhead is linear to the size of the advertisement table, which is $O(N_bM)$ for SDNi-TE and $O(KN_bM)$ for BGP-Addpath.

At the `Update` stage, the computation overhead depends on the data structures used for topology table and advertisement table. The topology table in SDNi-TE uses a map data structure, where the key of the map is the *source* and the value is the vector of all entries have the same *source*. With this data structure, the time complexity for determining whether the insertion is necessary is $O(T)$, and $O(T)$ for insertion and deletion, where T is the number of entries with the same *source*. According to our discussions on table size, $T = KM/N_c - (K-1) + \alpha K(M - M/N_c)$. Therefore, the total computation overhead for updating the topology table in presence of receiving an advertisement from a peering domain is $O(T)$. As for the advertisement table, we also use the data structure of map, where the key is the *source* and the value is the vector of all entries with the same *source*. As discussed in Section 4.3.4, we need to compute the maximum flow from all border switches to the sink $n_{z,s}$ when receiving a route to $n_{z,s}$. Before we do the max-flow computation, we first need to find the K shortest paths for any border switch to $n_{z,s}$, which requires $O(N_bT^2)$. With the selected paths, $O(N_b|V'||E'|^2)$ is required for max-flow computation, where $|V'|$ and $|E'|$ are the number of nodes and edges in the extracted subgraph, and $|V'| \leq |V|, |E'| \leq |E|$. Therefore, the computation overhead for updating the advertisement table is $O(N_bT^2) + O(N_b|V'||E'|^2)$ when a route is learned from the peering domain. In the case of BGP-Addpath, topology table update is the same as SDNi-TE except that the number of entries with the same *source* is K times more than that in SDNi-TE. Therefore, the computation overhead is $O(KT)$. As for the advertisement table, BGP-Addpath only needs to find the K shortest paths but does not need to compute the

Table 4.1: Comparison between SDNi-TE and BGP-Addpath

		SDNi-TE	BGP-Addpath
Topology table		N_bT	$\approx KN_bT$
Advertisement table		N_bM	KN_bM
Message overhead		BGP-Addpath requires K times more overhead	
Computation overhead	Initialization	Same computation overhead	
	Advertise	$O(N_bM)$	$O(KN_bM)$
	Update topology table	$O(T)$	$O(KT)$
	Update advertise table	$O(N_bT^2) + O(N_b\|V'\|\|E'\|^2)$	$O(N_bK^2T^2)$
	Build global view	$O(N_bT)$	$O(KN_bT)$

Note: $T = KM/N_c - (K-1) + \alpha K(M - M/N_c)$.

maximum flow. Therefore, the computation overhead for finding the K shortest paths is $O(N_bK^2T^2)$.

At the `Global View Construction` stage, BGP-Addpath and SDNi-TE follow the same procedure, which needs to go through the whole topology table. Therefore, the computation overheads to construct the global view for SDNi-TE and BGP-Addpath are $O(N_bT)$ and $O(KN_bT)$, respectively.

The differences between SDNi-TE and BGP-Addpath are summarized in Table 4.1. From the table, we can see BGP-Addpath not only has larger tables and higher message overhead but also requires higher computation overheads except for advertisement table update. In general case, T is much larger than both $|V'|$ and $|E'|$ because $M \gg |V'|$, therefore SDNi-TE generally requires less computation overhead than BGP-Addpath.

4.4 Performance Evaluation

4.4.1 Simulator Implementation

To evaluate the performance of SDNi-TE, a flow level simulator is developed to implement the protocol in C++ [72], whose framework is shown in Listing 1. In this simulator, SDN domains exchange information iteratively until the topology tables of all SDN domains are stable (line 1 - line 13). In every iteration, each SDN domain sends all satisfied entries

Table 4.2: Main Parameters for BRITE Network Topology Generator

Parameter	Value
Model	Top-down hierarchical topology
Domain generation method	Barabasi
Node generation method	Waxman
Edge connection method	Random
Inter-domain bandwidth distribution	Constant (i.e.,1000)
Intra-domain bandwidth distribution	Uniform (min=1, max=10)
Growth type	Incremental
Waxman-specific exponent	alpha=0.15, beta=0.2
number of SDN domains	10
number of nodes in one domain	30

in its advertisement table to all of its neighboring domains, and consequently updates its topology table and advertisement table when receiving information from the neighboring domains. Once all domains' tables are stable, each SDN domain then generates traffic, allocates link bandwidth, and delivers the traffic through its own domain (line 14 - line 20).

Topology Generation

In our simulations, we use BRITE [73] to generate the top-down hierarchical topologies[4]. BRITE is a universal tool designed to generate topology according to a wide variety of models. The top-down hierarchical topologies used in our simulations are generated according to the following steps: (1) generate domain-level topology according to one of the available domain-level models (e.g., Waxman, Barabasi); (2) for each node in the domain-level topology, generate a switch-level topology using another model; (3) using an edge connection mechanism to interconnect switch-level topologies as dictated by the connectivity of the domain-level topology. The main parameters for BRITE used in our experiments are shown in Table 4.2.

[4]Use the configuration file `TD_ASBarabasi_RTWaxman.conf` provided by BRITE.

Traffic Generation

Each node in the simulated topology generates a traffic flow with a probability 0.5. If a node generates a traffic flow, the destination of this flow can either be in the same SDN domain (with probability 0.1) or another SDN domain. If the destination is in the same SDN domain as the source, we uniformly pick one node except for the source itself in the SDN domain as the destination. If not, we uniformly pick one SDN domain from the rest SDN domains as the destination domain and any node in the picked SDN domain as the destination node. Furthermore, the size of this traffic flow is the product of a uniformly picked value from a set $\{1,2,3,4,5,6,7,8,9,10\}$ and the load factor σ.

Bandwidth Allocation

SDNi-TE only provides a consistent global view for each involved SDN domain, but does not specify how to do traffic engineering. In this simulator, we implement two traffic engineering algorithms to allocate bandwidth for flows. One is Google's TE Optimization algorithm [32]. This algorithm assigns a bandwidth function for each flow, and allocates link bandwidth among flows according to their bandwidth functions such that all competing flows on one edge either equally share the bandwidth or fully satisfy their demand. This algorithm achieves a trade-off between fairness and throughput. The other one aims to maximize the throughput of the whole network. It iterates until no path exists for any traffic flow, where it finds the shortest path for every traffic flow, and saturate the shortest one among all the shortest paths in each iteration. This algorithm aims to maximize the throughput of the whole network, but losses fairness among flows. Each SDN domain can use either algorithm.

Traffic Delivery

The simulator is a flow-level simulator, which does not deliver individual packets. For every time slice ($\mathcal{T}$ in total), each SDN domain delivers the flows which are injected from its

neighboring domains in the previous slice or generated in the current slice. To be specific, for SDN domain i at time slice t_l, when its TE algorithm allocates a path $\mathcal{P}$ for a traffic flow f (generated locally or injected from any domain $j \in \mathcal{A}^i$ at t_{l-1}), $B_{\mathcal{P}}$ units traffic of the flow f will be delivered during time slice t_l. If the destination of flow f is within the domain i, $B_{\mathcal{P}}$ units traffic is delivered to the destination at t_l. Otherwise, $B_{\mathcal{P}}$ units traffic of the flow f will be injected to the neighboring domain j containing the switch in $\mathcal{P}$ and thus will be delivered further to the destination by domain j in the time slice t_{l+1}.

Benchmarks

To investigate the performance of SDNi-TE, it is necessary to compare it with BGP-Addpath. Therefore, BGP-Addpath is also implemented in the simulator. The implementation of BGP-Addpath is the same as SDNi-TE except that the behavior of updating tables are different. Instead of aggregating the K shortest paths as an aggregated path, BGP-Addpath selects the K shortest paths from the topology table and directly advertises them to the adjacent domains. In addition, the perfect but impractical scenario, where each controller has the full knowledge of the topology of the whole network, is also implemented and referred as "Benchmark" in the rest of discussions. In this scenario, no eastbound/westbound interface is needed and every controller runs its traffic engineering algorithm based on the full network view directly. Finally, the Dijkstra algorithm is also implemented to compare the performance of TE with no-TE. In summary, we consider four scenarios: SDNi-TE, BGP-Addpath, Benchmark, and Dijkstra. For each of the first three scenarios, three different TE cases are investigated: all SDN domains use Google's algorithm (labeled as *Google*), all SDN domains use the other algorithm to maximize throughput (labeled as *MaxThr*), and each SDN domain uniformly pick one of the two algorithms as its traffic engineering algorithm (labeled as *Mix*).

Listing 1 SDNi-TE Simulator Framework

Require: simulating time T

```
1: notStable ← true
2: while notStable do
3:    notStable ← false
4:    for each domain i do
5:       for each domain j ∈ A^i do
6:          domain j send the satisfied entries in its advertisement table to domain i
7:          domain i updates its topology table and advertisement table based on the received entries from domain j
8:          if the topology table of domain i is updated then
9:             notStable ← true
10:         end if
11:      end for
12:   end for
13: end while
14: for time slice t < T do
15:   generate traffics
16:   for each domain i do
17:      allocate bandwidth and consequently deliver traffics through the domain according to the traffic engineering algorithm
18:      push across-domain traffic to the neighboring domains
19:   end for
20: end for
```

4.4.2 Simulation Results

To evaluate the performance, we choose throughput and cost as the performance metrics. The throughput is the sum of the received traffic on each node, and the cost is defined as the average cost/weight to transmit one unit traffic from the source to the sink. The performance of SDNi-TE with various loads is first investigated, which is shown in Figure 4.4. From this figure, we have the following observations:

- As the load factor σ increases, both the throughput and costs increase up to a certain level, then stop increasing or does not increase further significantly. This is because traffic delivery is subjected to both the traffic load and link bandwidth. When σ is small, the dominated factor is σ and more paths are used to deliver traffic when σ increases. Therefore, throughput increases as σ grows. Meanwhile, more sub-optimal (in terms of cost) paths are used as σ grows such that the cost also increases. However, when σ is increased up to some threshold such that the load is larger than link bandwidth, throughput cannot be improved any more. This is also true for the cost.

- SDNi-TE achieves almost the same throughput and cost than BGP-Addpath. For example, the cost of SDNi-TE is only about 1% lower than that of BGP-Addpath.

- SDNi-TE achieves smaller throughput but higher cost, compared with Benchmark. However, the performance loss is very small. In the case of *MaxThr*, SDNi-TE losses 4.5% throughput and increases $< 5\%$ cost. For *Mix* case, SDNi-TE achieves even lower cost. Especially, in the most practical case *Google*, we can see that SDNi-TE achieves almost the same throughput (96%) and cost (98%) as Benchmark.

- Compared with Dijkstra, SDNi-TE can achieve much higher throughput but also results in higher cost. In the *Google* case, SDNi-TE achieves 238% throughput gain but also 74% cost loss. This is because SDNi-TE provides multiple paths to deliver

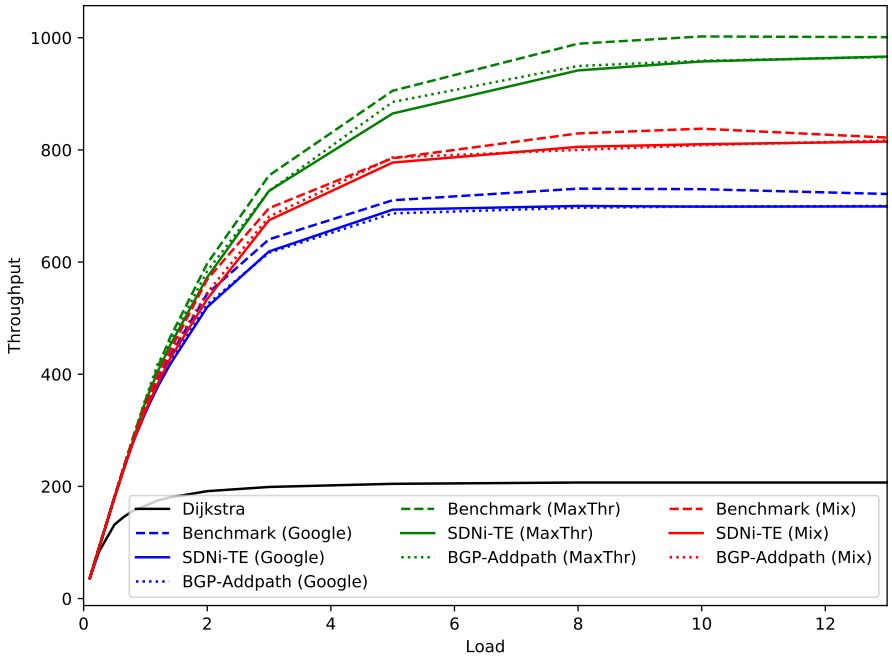
one flow while Dijkstra only provisions the shortest path for routing traffic.

Combined the discussions in Section 4.3.5, we can conclude that 1) SDNi-TE can fully exploit the information owned by the controllers to enable better TE because it can achieve nearly the same performance as the one which has the God's knowledge even in the case where different SDN domains apply different TE policies. 2) SDNi-TE achieves the same performance as BGP-Addpath, with much smaller tables, message overheads, and computation overheads. 3) With the full information of the network state, SDNi-TE can achieve much higher throughput than the traditional Dijkstra method, at the cost of improving the cost of packet transmission.

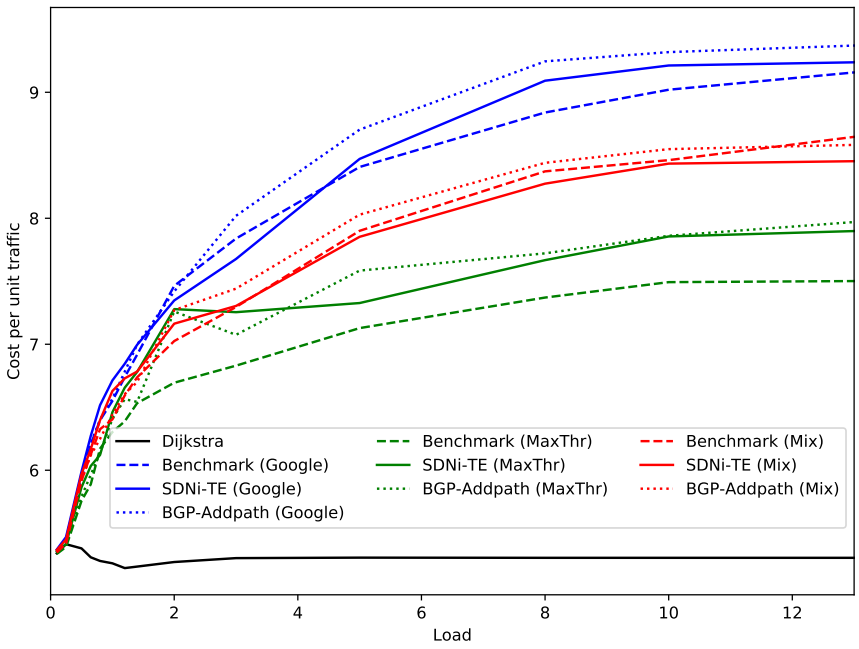
We then study the relationship between the performance of SDNi-TE and the number of paths (k). Figure 4.5 shows the simulation results. As we can see, when k increases from 1 to 4, the throughput gain is around 15% (Google), 21% (MaxThr) and 17% (Mix). This is because, with larger k, an aggregated path with a larger bandwidth can be propagated to the neighboring SDN domains and the capacity of the network can be better utilized. Of course, the corresponding cost of transmitting one unit traffic is increased. The cost is increased by 18% (Google), 13% (MaxThr), and 15% (Mix). This is because the aggregated path consists of suboptimal paths in the case of $k = 4$ and the traffic flow going through the suboptimal paths would incur higher cost. However, when k continues to be increased, we find that the throughput cannot be increased significantly, even slightly decreased when $k = 5$. This is because, in one SDN domain, the injected flow from other SDN domain has to share the network bandwidth with other flows. Therefore, even if the size of the injected flow is increased due to the larger aggregated bandwidth, the amount of the traffic of the flow can be delivered through the SDN domain may not be increased because of the network bandwidth and the competitions from other flows. Moreover, large k will definitely increase the size of the path and topology tables. Actually, we have done experiments with various k in different network scenarios, and the results show that $k = 3$ or $k = 4$ is a good value for balancing the network performance and the size of the tables.

4.5 Discussion

Besides TE, other applications can also benefit from the global view constructed based on SDNi-TE. The first potential beneficial application is admission control. Since bandwidth information is advertised through SDNi-TE, the controller can understand how much traffic can be sent out through one specific crossing domain link. In this way, the controller can check whether there is enough amount of bandwidths in the network to accept new traffic transmission requests. In addition, load balancing can also increase resource utilization efficiency through adjusting weights of links since the controller knows the full topology.

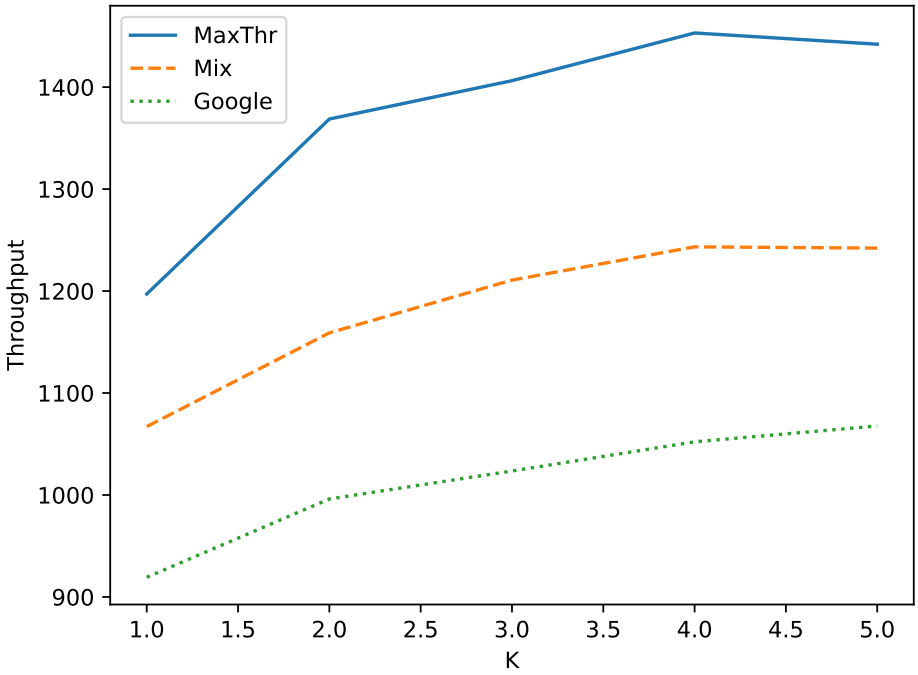


(a)

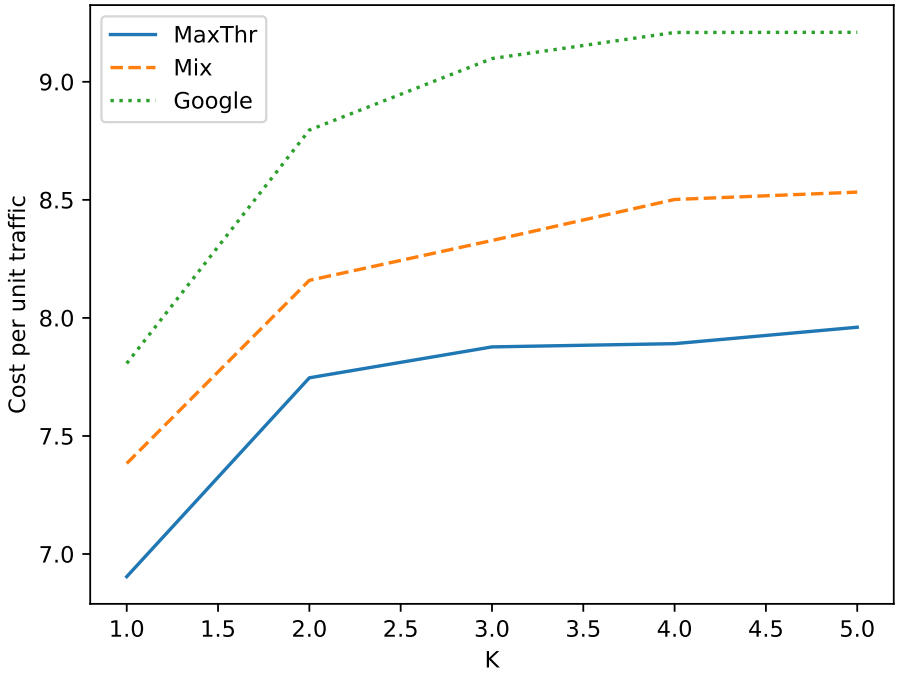


(b)

Figure 4.4: The performance of SDNi-TE versus various load factor σ ($N_{conn} = 6$, $k = 3$).



(a)



(b)

Figure 4.5: The performance of SDNi-TE v.s. number of paths ($N_{conn} = 9$, $\sigma = 10$).

CHAPTER 5

SMART PROACTIVE ENTRY DELETION FOR OPENFLOW

The last two chapters focus on scaling the control plane by designing an eastbound/westbound interface for the distributed SDN. This chapter turns to the data plane. As discussed in Chapter 1, this dissertation works around the physical limitations on hardware capabilities in the data plane but focuses on optimizing resource management, specifically flow table management such that the inadequate flow table can be better utilized and the control overhead can be reduced. In this chapter, proactive flow entry deletion which aims to prevent flow table overflow is optimized. Proactive flow entry deletion refers to the process where the controller commands OpenFlow switches to delete flow entries by explicitly sending specific OpenFlow messages when the flow table is close to be overflowed. The key challenge for proactive deletion is to decide which flow entry should be deleted. If an active flow entry is removed from the flow table, the switch has to query the controller again to reinstall the flow entry when the packets of the flow arrive in the future. This re-installment not only incurs unexpected delays [5] but also increases controllers' workloads. Furthermore, evicting an active TCP flow entry can seriously degrade the performance of TCP connections because it may result in packet loss and congestion window shrinkage for all TCP flows which share a same switch buffer [74]. The third piece of this dissertation in [75] proposed to use machine learning techniques to learn knowledge from historical flow entry removals and thus predict the time when a flow entry will be last referred to. Based on the predictions, the flow entry which is the least recently used will be proactively deleted when the flow table is close to be overflow.

5.1 Challenges of Proactive Flow Entry Deletion

To apply proactive flow entry deletion, two problems should be addressed. First, when should the controller proactively delete flow entries? In general, we should not remove flow entries if the flow table is not about to be overflowed. Otherwise, it is possible that an active flow entry will be removed, which can hurt the performance of OpenFlow networks. A naive solution is to set a threshold (β) for the flow table and the controller starts to proactively delete flow entries when flow table utilization crosses this threshold. Second, which flow entries should be deleted? Of course, we want to remove inactive flow entries, to which no packet will refer in the future. However, it is difficult if not impossible for the controller to exactly know which flow entries in the flow table are inactive. Therefore, some intuitive and easy strategies are applied [42]. These strategies including deleting a random flow entry (i.e., random policy) and deleting the first installed flow entry (i.e., FIFO policy). Another possible strategy is to delete the LRU flow entry. A. Zarek shows that LRU is not a feasible approach for practical implementation because of architectural reasons [44]. To apply LRU on the controller side, the controller has to track the order of flow entry accesses in real time. This is infeasible within the framework of OpenFlow because the controller only has non-real time accesses to coarse-grained counters. Even if the controller can achieve real-time tracking, the overhead of signaling will be intolerable.

5.2 Why Machine Learning Can Help

Although we cannot know the exact order of flow entry access, can we infer the order of accesses based on historical flow removes by exploiting machine learning techniques? To explore this possibility, we first examine what data we can collect from the previous flow removes. Once a flow entry is removed from the flow table due to flow expiry, proactive deletion, or eviction, the controller can configure switches to send a `Flow_Removed` message to it. This message contains the remove reason of the entry, the entry duration (the

time interval between flow entry installation and flow removal), number of packets matched by the entry, idle timeout, hard timeout, number of bytes in packets matched by the entry, and other experimenter-specific stats. For flow f, suppose we have already collected the stats of n flow removals. And there is a flow entry e_{n+1} corresponding to f in the flow table of switch s. Can we use the stats of last n flow removals to predict the time $tLastVisit_f$ when the last packet will be matched by flow entry e_{n+1}? If we can do this prediction with reasonable accuracy, we can then delete the flow entry with the smallest $tLastVisit_f$, which is actually an LRU-like policy.

5.3 Overhead of Proactive Flow Entry Deletion

The overhead[1] of proactive flow entry deletion mainly contains two parts. One is the overhead of deciding when proactive deletion should be started. In this work, proactive deletion is assumed to be started when flow table usage crosses a threshold (β) chosen by the controller. This mechanism is actually specified in the OpenFlow protocol, where vacancy events are generated to warn the controller to react in advance to avoid flow table overflow. Vacancy mechanism depends on two parameters, `vacancy_down` and `vacancy_up`, chosen by the controller, and the process of generating vacancy events is shown in Listing 3. Once receiving vacancy events, the controller can commence the process of proactive flow entry deletion. However, if the switch is configured to send a `Flow_Removed` message for every flow entry removal, the controller can track flow table usage in real time because it knows the time of installation and removal for every flow entry. In this case, it is unnecessary to generate and send vacancy events for the switch. The other part of the overhead comes from deciding which flow entry should be proactively deleted. Machine learning based policy depends on `Flow_Removed` messages, but random and FIFO policy can make decisions without overheads. However, without `Flow_Removed` messages, the controller does not know which flow entries stay in the flow table. Therefore, it is

[1] In this work, the overhead of proactive flow entry deletion refers to the number of OpenFlow messages exchanged between the controller and the switch.

possible that the controller will instruct a switch to delete a flow entry which is actually not residing in its flow table. On one hand, this wrong instruction (sending `DELETE` or `DELETE_STRICT` message) will intensify the switch and the controller's workload. On the other hand, it is highly possible that the flow table may overflow since no flow entry is removed from the table. Flow table overflow can have serious impacts on OpenFlow networks [74]. Firstly, flow overflow will lead to a chain reaction of flow entry eviction and re-installment (i.e., evict an active flow entry, reinstall this flow entry in the near future, which in turn evicts another active flow entry), which makes flow setup request rate grow almost proportionally to flow data rates. These massive flow setup requests can overwhelm the controller and switches' CPUs. Secondly, flow table overflow could frequently reduce TCP congestion window size, thus resulting in low TCP throughput and long packet delay. Finally, flow table overflow can also lead to inference attack and privacy leakage under certain circumstances [76]. On the contrary, if sending `Flow_Removed` messages is enabled, there will be no such wrong instructions because the controller knows which flow entries are in the switch's flow tables.

Listing 2 Vacancy event generation

Require: `vacancy_down`, `vacancy_up`
  isSendDown=True, isSendUp=True, isUpState=False
  if remainSpace $\geq$ `vacancy_down` and isSendDown **then**
    isSendDown=False, isSendUp=True, isUpState=False
    generate a vacancy down event
  else if remainSpace $\geq$ `vacancy_up` and isSendUp **then**
    isSendUp=False, isSendDown=True, isUpState=True
    generate a vacancy up event
  end if

5.4 Smart Proactive Entry Deletion for OpenFlow

As discussed in Section 5.2, we can collect stats of previous flow entry removals in order to predict the time when a flow entry is lastly referred to. Based on the predictions, we can then derive an LRU-like proactive flow deletion policy. Motivated by such observations,

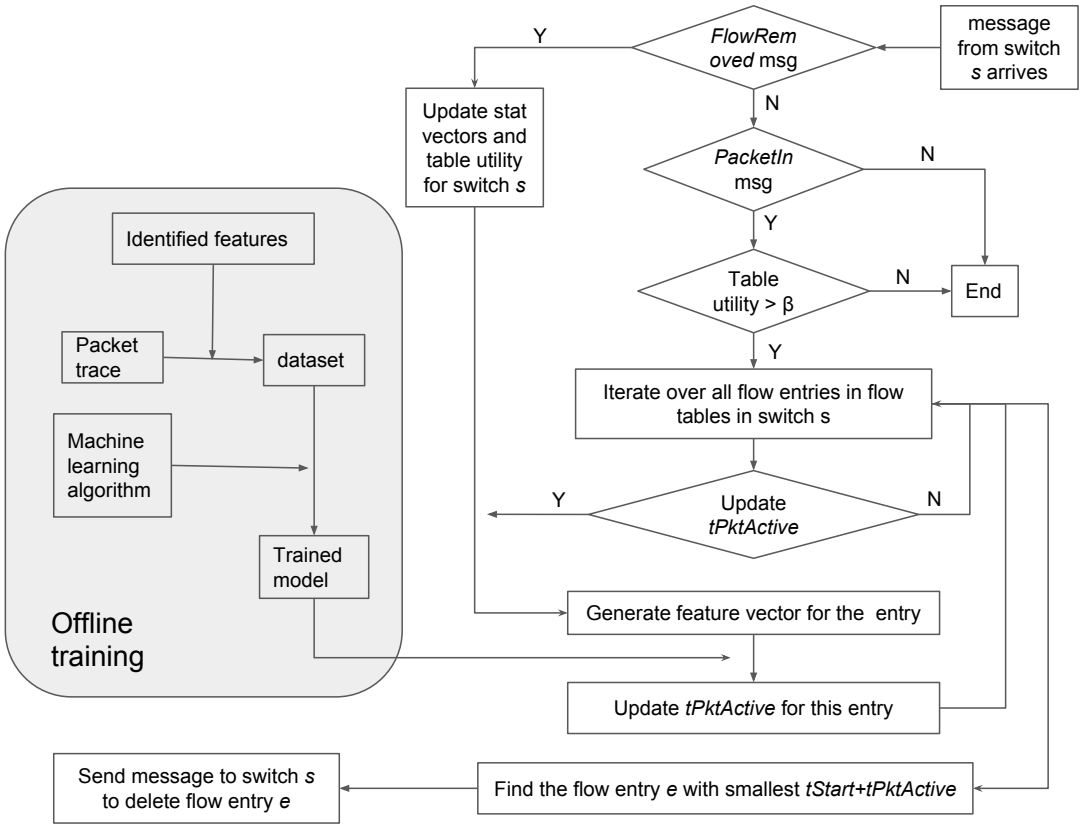


Figure 5.1: The framework of SPEEDO.

Smart ProactivE Entry Deletion for Openflow (SPEEDO) is proposed, which is shown in Figure 5.1. As the figure shows, SPEEDO mainly consists of two parts: offline model training and online proactive flow deletion based on the trained model.

5.4.1 Offline Model Training

To train a machine learning model, we first need to collect appropriate datasets. According to the discussions in Section 5.2, we need to collect data where each data sample contains features of previous flow removals and the label $tLastVisit_f$ (i.e., the time when a flow entry is last referred to). $tLastVisit_f$ can be computed as $tStart_f + tPktActive_f$, where $tStart_f$ is the time when the current flow entry of flow f is installed in the flow table, and $tPktActive_f$ is the elapsed time the flow entry has been referred to. The controller can estimate $tStart_f$ by simply add the time when the controller instructs the switch to

Table 5.1: $tPktActive$ prediction features

Feature	Description
$numRemoval$	the number of flow entry removals
$mean\_removeReason$	mean of $removeReason$
$std\_removeReason$	standard deviation of $removeReason$
$mean\_tDuration$	mean of $tDuration$
$std\_tDuration$	standard deviation of $tDuration$
$mean\_numPkt$	mean of $numPkt$
std_numPkt	standard deviation of $numPkt$
$mean\_tInterval$	mean of $tInterval$
$std\_tInterval$	standard deviation of $tInterval$
$cur\_tInterval$	the interval between last flow entry removal and current flow entry installation

install the flow entry with the propagation delay between the controller and the switch. Therefore, we use $tPktActive$ as the label. As for features, we use the mean and standard deviation of the stats carried by `Flow_Removed` messages, i.e., number of packets matched by a flow entry ($numPkt$), entry duration ($tDuration$), and remove reason ($removeReason$[2]). Besides, we use the mean and standard deviation of the intervals between two successive entry removals ($tInterval$). We also use the number of flow entry removals ($numRemoval$) and the interval between last flow entry removal and current flow entry installation ($cur\_tInterval$) as another two features. Table 6.1 summarizes all the features used in our machine learning model training.

With the identified features, we can then generate the dataset for training from real network packet traces. The dataset generation process consists of two steps. The first step is to use OpenFlow simulator (see Section 5.5 for more details) to replay a packet trace. This simulator can simulate the behaviors of the controller and the switch when the packets in a packet trace are presented, hence collect the stats of all flow removals for every flow. These stats are stored in several statistic vectors: v_f_tStart (the vector of flow entry installation time for flow f), $v_f\_removeReason$ (the vector of flow entry

[2] $removeReason = 0$ represents the flow is removed by eviction, 1 by proactive deletion, and 2 by flow expiry.

removal reasons for flow f), $v_f\_tDuration$ (the vector of flow entry durations for flow f), v_f_numPkt (the vector of the number of packets matched by flow entries for flow f), $v_f\_tInterval$ (the vector of the interval between two successive flow removals for flow f), and $v_f\_tPktActive$ (the vector of the elapsed time flow entries have been referred to for flow f). With these vectors, we can generate $N_f - 1$ (N_f is the number of flow removals for flow f) data samples for flow f as following: for $1 \leq i < N_f$, generate a data sample $\big\{i$, mean($v_f\_removeReason[:i]$[3]), std($v_f\_removeReason[:i]$), mean($v_f\_tDuration[:i]$), std($v_f\_tDuration[:i]$), mean($v_f\_numPkt[:i]$), std($v_f\_numPkt[:i]$), mean($v_f\_tInterval[:i]$), std($v_f\_tInterval[:i]$), $v_f\_tInterval[i]$, $v_f\_tPktActive[i+1]\big\}$, where $v_f\_tPktActive[i]$ is the label and the others are features.

With the collected dataset, we need to select an appropriate machine learning algorithm and tune its hyperparameters[4]. As we can see, the learning problem is a regression problem. Many algorithms can be used for regression, such as support vector regression, neural network, random forest, gradient boosting regression, and decision tree. To select the best algorithm and tune its hyperparameters, we can apply k-fold cross validation to evaluate the performance of different algorithms with various hyperparameter configurations, where mean squared error is adopted as the performance metric.

5.4.2 Online Proactive Flow Entry Deletion

Once offline training is finished, we can apply the trained model for online proactive flow entry deletion. Similar to dataset generation, the controller uses statistic vectors to record the stats of all flow removals in each switch under its control. When a `Flow_Removed` message from switch s arrives, the controller will update the statistic vectors for switch s. Besides, the controller can update the utility of the flow table (U_f) for switch s since a flow entry is removed from the flow table. When a `Packet_In` message arrives, the controller

[3] vector[:i] represents the first i elements in the vector.

[4] For a machine learning algorithm, hyperparameters are parameters whose values are set manually before the learning process begins. For example, we need to determine the number of trees before we train a random forest model.

will first check whether $U_f > \beta$. If yes, the controller will commence the process of proactive entry deletion. It will first update $tPktActive_f$ for all flows (if the statistic vectors of flow f stay same since the last update of $tPktActive_f$, it is unnecessary to resort to the trained model to update $tPktActive_f$). After all $tPktActive_f$ are updated, the flow with smallest $tStart_f + tPktActive_f$ will be searched and its flow entry will be proactively deleted from the flow table by the controller.

5.5 Case Study

In this section, case studies on two real network packet traces will be presented to show the performance gain of our SPEEDO. In the case studies, a flow is defined by five tuples, i.e., source IP address, source port number, destination IP address, destination port number, and the protocol.

5.5.1 Data Sources

In this case study, two real network packet traces (i.e., UNIV1 and UNIV2) are used for performance evaluation. These two traces[5] are collected from university data centers by the authors of [6]. The characteristics of the traces are summarized in Table 5.2.

As discussed in Section 5.4.1, we need to build an OpenFlow simulator which can replay packet traces according to the OpenFlow specification. The implemented simulator contains two objects: a switch and a controller. On the switch side, when a packet p in the trace arrives, the switch will check whether any flow entry in the flow table can match with p. If none of the entries can match, a `Packet_In` message will be generated and sent to the controller. When a flow entry is removed (because of expiry, deletion, or eviction), the switch will send a `Flow_Removed` message which carries the stats to the controller. On the controller side, it will update the corresponding statistic vectors when

[5]They can be downloaded from `http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html`. UNIV2 trace contains 9 sub-traces and collects approximately 1 billion packets in total. only the first two sub-traces are used in this work for convenience because it takes a very long time to process 1 billion packets in the available PC.

Table 5.2: Summary of packet traces and generated datasets.

	UNIV1	UNIV2
Duration (s)	3914	2357
Number of packets	19,855,388	23,835,028
Number of flows	577,675	61,930
Flow table size	1K, 2K, 4K	1K, 2K, 4K
Number of samples	129,479 (1K)	100,552 (1K)
	92,625 (2K)	76,225 (2K)
	64,779 (4K)	56,873 (4K)
Training duration (s)	1000	1000
Number of packets within training duration	3,358,010	10,340,784

a `Flow_Removed` message arrives. Moreover, when a `Packet_In` message arrives, the controller will check whether the flow table usage $> \beta$. If yes, the controller will select a random flow entry and instructs the switch to delete the flow entry. Note here the controller proactively deletes a random flow entry in the presence of $U_f > \beta$ such that the generated dataset can have a similar distribution as the data fed into the trained model in the online proactive flow entry deletion. This is the foundation on which the trained model can be generalized to handle test data.

For both traces, only the packets in the first 1000 seconds are replayed to generate the datasets. Furthermore, the size of flow table is set to be 1K, 2K, or 4K, which are compatible with the configurations in many studies [48], [42], [51]. Moreover, $\beta = 0.95$ is configured, which is same as [42]. In total, for each packet trace, three datasets corresponding to three flow table sizes are generated, and these datasets are summarized in Table 5.2.

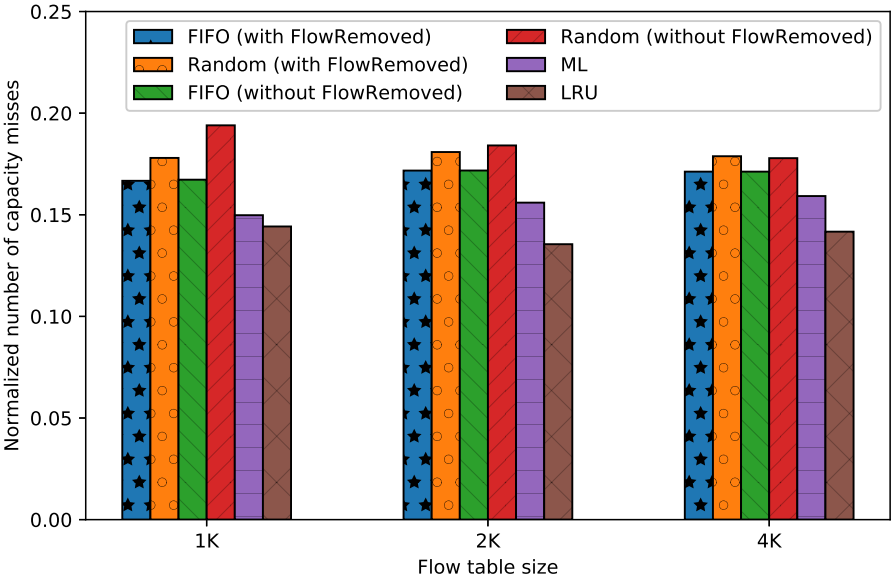
5.5.2 Offline Training Results

The offline training is based on `scikit-learn` [77], an open source machine learning library in Python. This library provides many regression algorithms, as well as the APIs for model selection based on cross validation. Support vector regression (SVR), neural net-

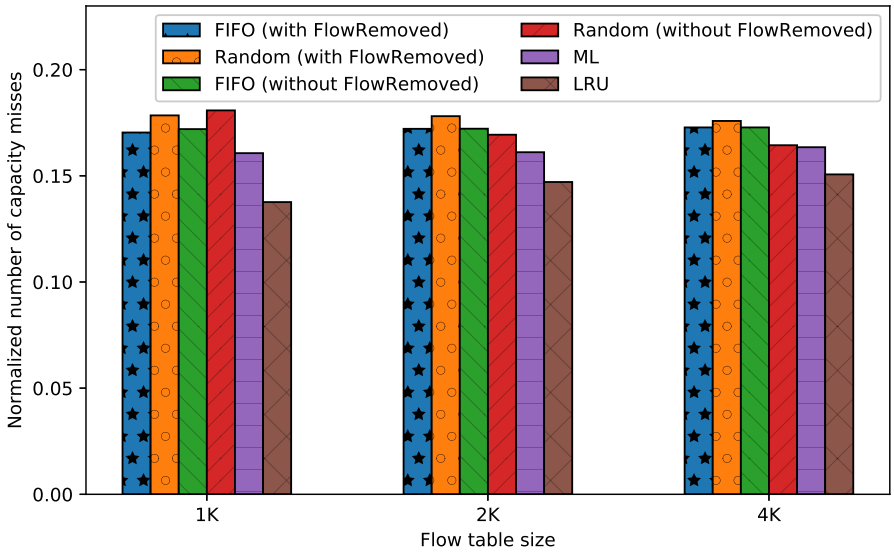
work, random forest, and gradient boosting regression (GBR) algorithms are tried for each dataset. Each dataset is split into a training set and a test set, and the splitting ratio is 80/20. We then apply 5-fold cross validation on the training set to tune the hyperparameters for each algorithm, and the hyperparameter search space for each algorithm is shown in Table 5.3. These hyperparameters are defined in the APIs for the corresponding algorithms in `sklearn` library (e.g., `sklearn.ensemble.RandomForestRegressor`, refer to [78] for more details about the definitions of hyperparameters), all the other hyperparameters defined in the library but not included in Table 5.3 use the default values provided by the library. The algorithm (with the best tuned hyperparameters) which can achieve minimal mean squared error is selected as the offline training model. Interestingly, random forest regression is proved to be the best algorithm for all six datasets, where the best tuned hyperparameters are slightly different for each dataset. With the selected random forest algorithm and its best tuned hyperparameters, the algorithm is trained on each dataset and the trained model is saved with the help of `joblib`, which can be loaded for online flow table management.

5.5.3 Online Simulation Results

Simulations on UNIV1 and UNIV2 packet traces are carried out with different flow table sizes (1K, 2K, and 4K). The simulator is similar to the one used in dataset generation except that the proactive flow entry deletion policy can be configured. Six policies are implemented: LRU, SPEEDO(ML), random policy with/without `Flow_Removed`, and FIFO policy with/without `Flow_Removed`. Although LRU is not feasible in practice as discussed in Section 5.1, it's good to see how big the performance gap between LRU and SPEEDO is. For SPEEDO, the trained models in the last section will be used to decide which flow entry should be proactively deleted, and the switch will send a `Flow_Removed` message to the controller whenever a flow entry is removed. For random policy, the controller will proactively delete a random flow entry when proactive deletion is required. For

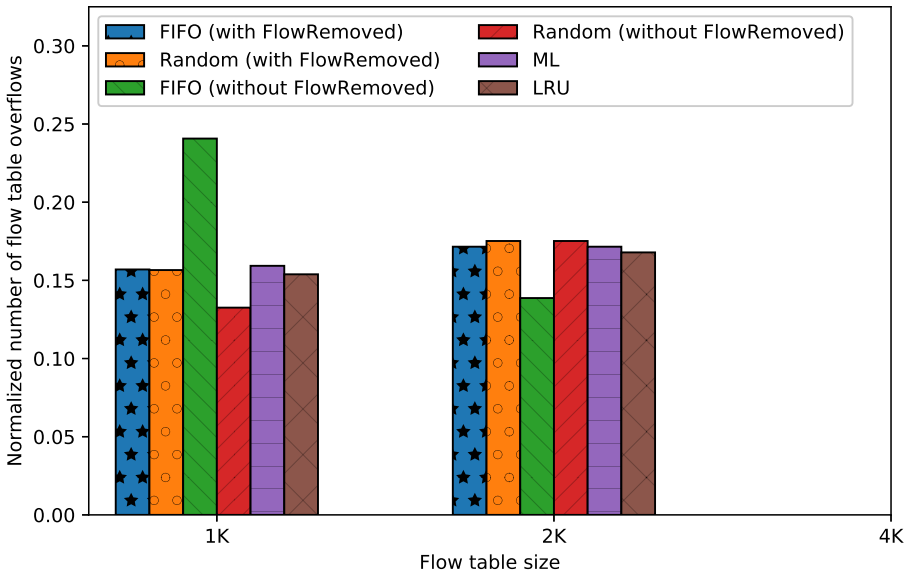


(a) UNIV1

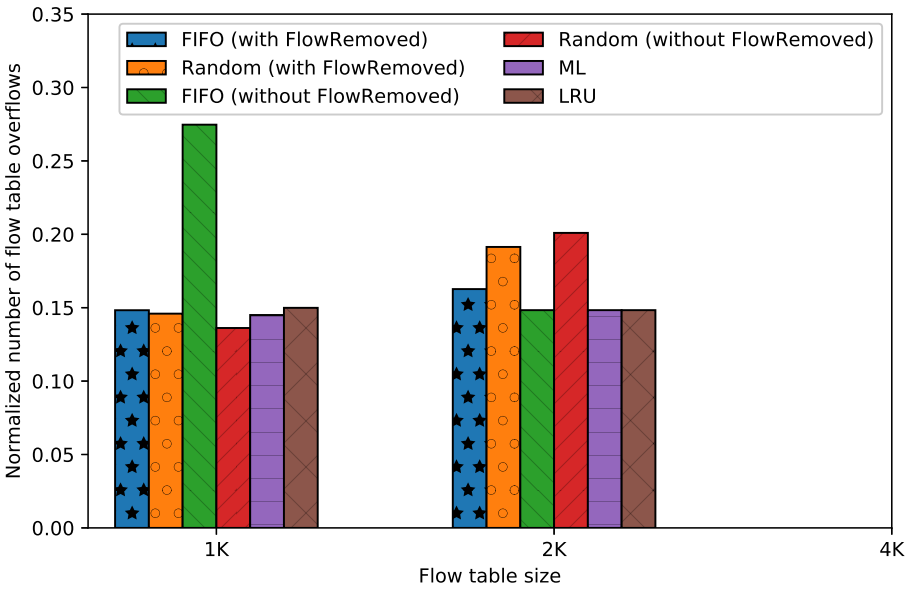


(b) UNIV2

Figure 5.2: The performance of different proactive deletion policies in terms of the number of capacity misses.

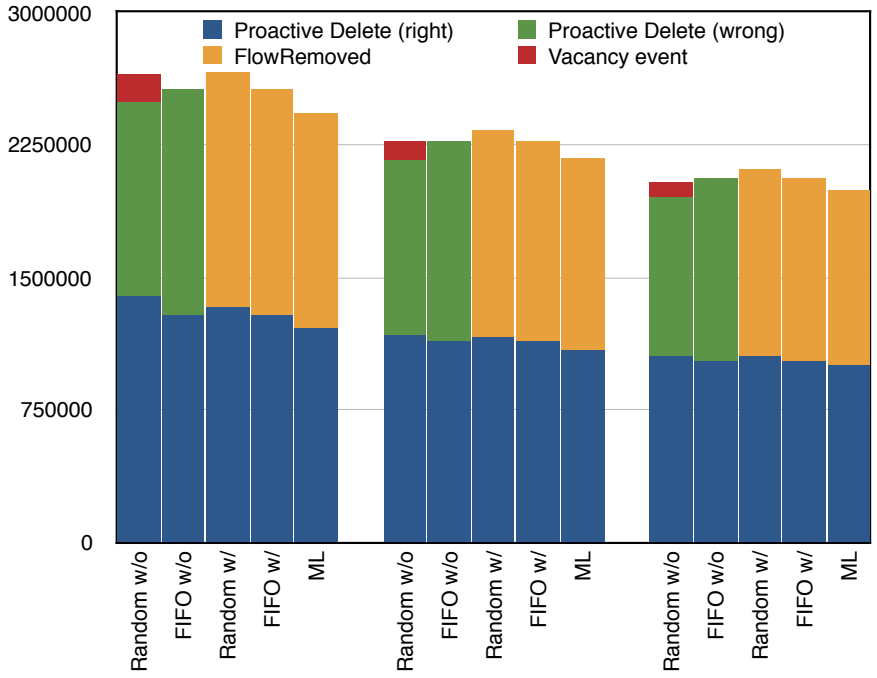


(a) UNIV1

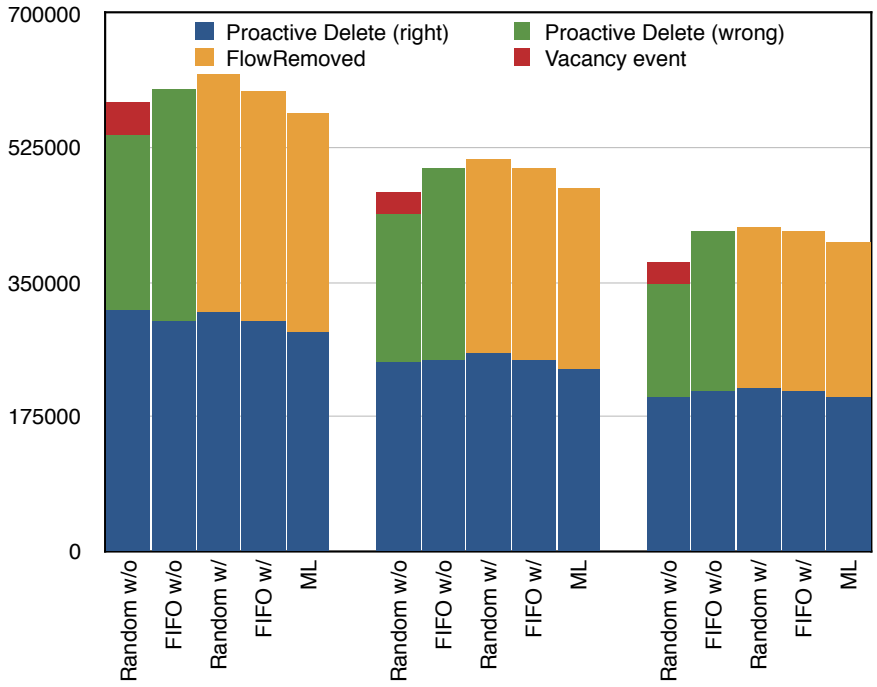


(b) UNIV2

Figure 5.3: The performance of different proactive deletion policies in terms of number of flow table overflows. With 4K flow table, for both UNIV1 and UNIV2, no flow table overflow happened.



(a) UNIV1



(b) UNIV2

Figure 5.4: The overheads of different proactive deletion policies. The first five bars are the overheads for 1K flow table, the second five for 2K, and the last five for 4K. A right proactive deletion is the one which can successfully delete a flow entry in the flow table, while a wrong deletion fails to delete an entry because the entry is not residing in the flow table.

Table 5.3: Hyperparameter search space and results for each studied algorithm.

Algorithm	Hyperparameters	Value
SVR	C	0.01, 0.1, 1, 10
	gamma	0.01, 0.1, 1
Random Forest	max_features	auto$^{*\text{-}*}$, sqrt, log2
	max_depth	10, 20$^{1\text{-}1}$, 30$^{1\text{-}2,\ 1\text{-}4}$, 40$^{2\text{-}*}$
	n_estimators	20$^{1\text{-}1}$, 30$^{1\text{-}2,\ 1\text{-}4}$, 40$^{2\text{-}1}$, 100$^{2\text{-}2,\ 2\text{-}4}$
Neural Networks	hidden_layer_sizes	{3600}, {150,150}, {100,100,100},{85,85,85,85}
	alpha	0.01, 0.1, 1, 10
	learning_rate	constant, invscaling, adaptive
	learning_rate_init	0.01, 0.1, 1, 10
GBR	n_estimators	20, 30, 40, 100
	max_depth	10, 20, 30, 40
	loss	ls, lad, huber, quantile
	learning_rate	0.01, 0.1, 1, 10

Note: random forest regression is the best algorithm for all datasets, and the best tuned hyperparameters are marked with {dataset-size}. For example, the hyperparameters marked with 1-1 belongs to the algorithm on UNIV1 dataset with 1K flow table. In addition, the * wildcard is also applied.

FIFO policy, the first installed flow entry will be proactively deleted by the controller when proactive deletion is started. Furthermore, random and FIFO policies can be configured with/without `Flow_Removed`. With `Flow_Removed`, vacancy event will not be generated. In this scenario, the controller will proactively delete one flow entry once received a `Packet_In` message if the flow table usage $U_f > \beta$ ($\beta = 0.95$ in our simulations). Without `Flow_Removed`, vacancy mechanism will be turned on and the switch will generate vacancy events according to Listing 1 (`vacancy_up` is set to be 5% of the flow table size, and `vacancy_down` is set to be 20% of the table size). In this case, after the controller receives a vacancy up event, it will proactively delete one flow entry once receives a `Packet_In` message until the controller receives a vacancy down event.

First, the performance of the six proactive deletion policies are evaluated in terms of the number of capacity misses. In OpenFlow, a flow table miss occurs if an incoming packet cannot match any flow entry in the flow table. In general, there are two kinds of flow table misses. One is the compulsory miss, which occurs when the first packet of a

flow arrives at the flow table. The other one is the capacity miss, which occurs when flow entries are discarded from the flow table because of the limitation on the size of the flow table. Compulsory miss is inevitable and we here only consider capacity miss. It is significant to reduce capacity misses because the switch has to query the controller to reinstall the flow entry once a capacity miss occurs. On one hand, this re-installment incurs a long flow setup delay. On the other hand, both the switch and controller have to process more events and thus affect their performance (e.g., flow table update rate). Moreover, removing an active TCP flow entry can seriously degrade the performance of TCP connections [74]. Figure 5.2 shows the number of capacity misses achieved by the six proactive flow entry deletion policies for UNIV1 and UNIV2. Here the number of capacity misses for each policy is normalized with respect to the total number of capacity misses across all policies for that trace. As we can see, for both UNIV1 and UNIV2, SPEEDO can achieve fewer capacity misses than the other four practical policies for all considered flow table sizes. And the performance gain decreases as the flow table size increases. For example, compared with random policy without `Flow_Removed`, SPEEDO can achieve 23%, 15%, and 11% fewer capacity misses in the case of 1K, 2K, and 4K flow table on UNIV1 trace, respectively. This is because the size of the collected dataset decreases when the flow table size increases, hence making the prediction accuracy of the trained model drop off. In addition, the performance gain of SPEEDO on UNIV2 is smaller than that on UNIV1. For example, compared with FIFO policy with `Flow_Removed`, only 5% fewer capacity misses can be reduced by SPEEDO in the case of 4K flow table on UNIV2. However, the gain is 7% for UNIV1. This is because the Mean Squared Error (MSE) of the trained models on UNIV1 is smaller than that on UNIV2. For example, the MSE achieved by the trained RF model on UNIV1 is 0.576 in the case of 1K flow table, but the MSE is 4.861 on UNIV2. As for the performance gap between LRU and SPEEDO, we can see that LRU achieves smaller capacity misses but the difference is small. For example, for UNIV1, LRU only achieves less than 4% performance gain with 1K flow table. And for UNIV2, the performance gain

of LRU over SPEEDO is also less than 10% with 2K and 4K flow table. However, for UNIV1 (2K, 4K) and UNIV2 (1K), the gain of LRU over SPEEDO is more significant, which is around 15%.

Second, the number of flow table overflows is measured because table overflows can seriously hurt the OpenFlow network performance as discussed in Section 5.3. Figure 5.3 shows the normalized number of table overflows for all studied proactive flow entry deletion policies. When the flow table size is 4K, no flow table overflow occurs for all policies on both UNIV1 and UNIV2 packet traces. When the flow table is 1K, SPEEDO results in a slightly higher number of overflows ($< 1\%$) than random and FIFO policy with `Flow_Removed`. However, compared with random policy without `Flow_Removed`, SPEEDO results in 20% more table overflows on UNIV1 packet trace. On the other hand, SPEEDO can achieve 22% fewer number of capacity misses and 9% fewer overheads (see Figure 5.4) comparing with random policy without `Flow_Removed`. This trend, more overflows but fewer capacity misses and overheads, is also true for 1K flow table on UNIV2. It seems that there is a trade-off between capacity misses, overheads and flow table overflows in the case of 1K flow table. In this case, the question of which policy is better depends on the cost of capacity misses and flow table overflows. Suppose the cost of one capacity miss is $C_{capMiss}$ and the cost of one flow table overflow is $C_{overflow}$, machine learning policy is better if $C_{capMiss} * In_{capMiss} > C_{overflow} * De_{overflow}$ where $In_{capMiss}$ is the decreased number of capacity misses and $De_{overflow}$ is the increased number of flow table overflows. SPEEDO outperforms random policy without `Flow_Removed` because $In_{capMiss} > 18000(12000)$ while $De_{overflow} = 69(38)$ on UNIV1 (UNIV2) trace with 1K flow table. As for the performance gap between SPEEDO and LRU, we again observe that they have almost the same flow table overflows. Combined with Figure 5.2, we can conclude that SPEEDO is a good approximation for LRU, which is exactly the goal of SPEEDO.

Finally, the overheads of the five studied proactive deletion policies are also measured.

The overhead is measured as the number of OpenFlow messages (including `Flow_Removed`, vacancy events, and proactive deletion messages) exchanged between the controller and the switch. As we can see from Figure 5.4, SPEEDO has smaller overheads ($2\% \sim 8\%$) than the other four policies on both UNIV1 and UNIV2 traces, except that SPEEDO has 6% more overheads than random policy without `Flow_Removed` on UNIV2 trace with 4K flow table.

CHAPTER 6

SMART TABLE ENTRY EVICTION FOR OPENFLOW SWITCHES

In the last chapter, machine learning is employed to optimize proactive flow entry deletion to mitigate flow table overflow. However, flow table overflow is inevitable. In the presence of flow table overflow, flow entry eviction is necessary to provision non-disrupt services. All the existing flow entry eviction policies apply some heuristics to infer which flow entry is most likely to be inactive. For example, the least recently used flow entry is more likely to be inactive than the most recently used one. However, the inferences based on these heuristics cannot be very accurate, which seriously affect the usage of precious flow table space. To improve accuracy, machine learning is a straightforward approach. Nowadays, machine learning, especially deep learning which can automatically learn data representations and build complex concepts out of simpler concepts using multi-layer neural networks, is widely used in commercial products (e.g., image classification, speech recognition, natural language processing) [79]. To apply machine learning, two conditions must be satisfied: an existed pattern and related raw data. In the case of flow entry eviction, there exists a pattern for sure. For example, FIN packets will be sent when a TCP connection is terminated. In addition, we can collect stats of billions of packets in practice easily. For example, Wireshark can be used to capture live network data. With the collected data, the problem of identifying inactive flow entries is actually a binary classification problem, where each flow entry in the flow table is classified as either active or inactive. Following this idea, the last piece of this thesis in [80] and [81] tries to improve the accuracy of flow entry eviction such that the precious flow table can be better utilized. To be specific, Smart Table EntRy Eviction for Openflow Switches (STEREOS) is proposed to employ machine learning techniques to identify and evict inactive flow entries in the presence of flow table overflow.

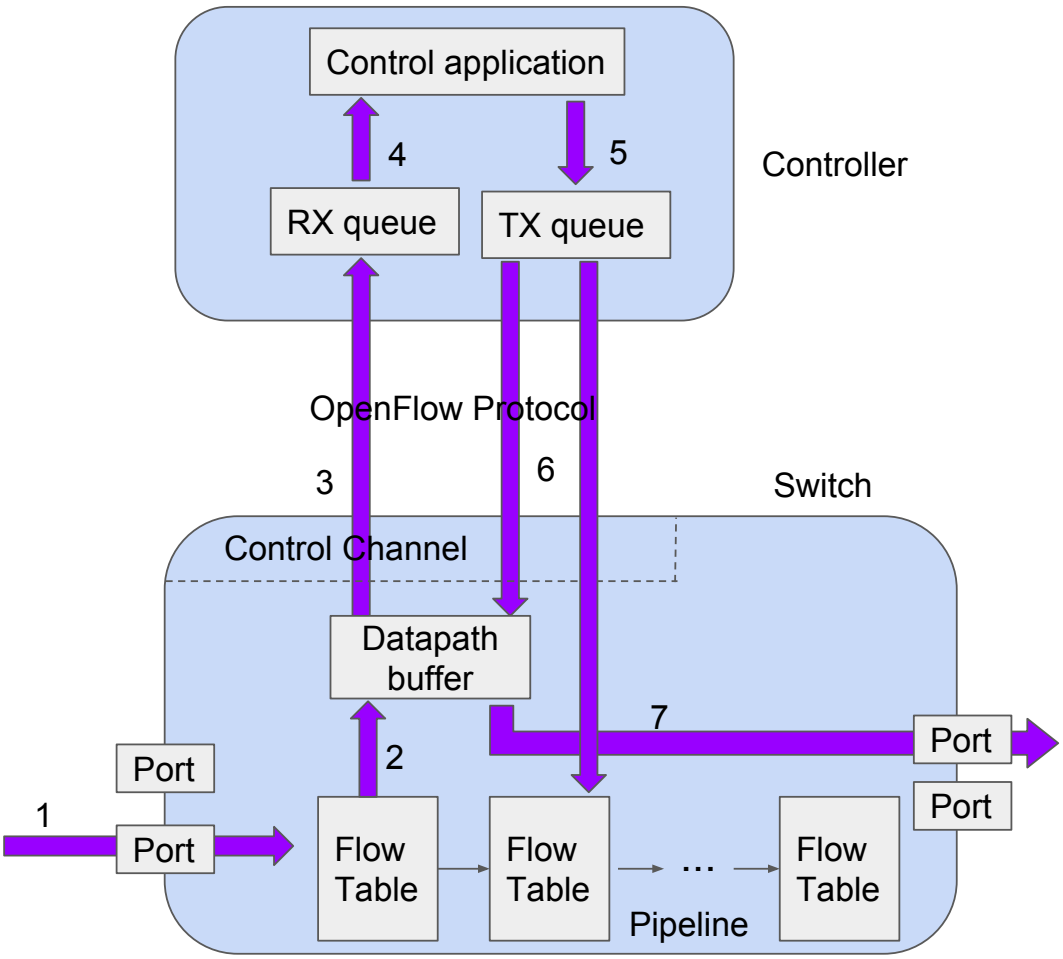


Figure 6.1: Flow setup in OpenFlow

6.1 Flow Entry Eviction

6.1.1 Flow Setup in OpenFlow

Before we dive into flow entry eviction, we first need to look at the flow setup process in OpenFlow. In Section 3.2, we have talked about flow setup, which is the process where the controller setups a path for a coming flow which cannot be matched with any existing flow entry. In this section, we need to add more details to flow setup such that the mechanism about how flow entry eviction can affect network performance can be revealed. As shown in Figure 6.1, flow setup consists of the following steps:

1. When the OpenFlow switch receives a packet from one of its ports, it first checks its

flow tables and tries to find one existing flow entry which can match the incoming packet.

2. If there exists a matched flow entry, the switch will handle this packet according to the matched entry such as outputting the packet to one specific port, dropping the packet, and modifying the packet. If no flow entry can match the packet, it will be sent to the datapath buffer.

3. A `Packet_In` message including some fraction of the buffered packet header is generated and sent to the controller.

4. The received `Packet_In` message is queued at the RX queue in the controller and then handled by the control application in order.

5. The controller generates `Packet_Out` and `Flow_Mod` events based on its logic and the received `Packet_In`. These two newly generated messages are then sent to the TX buffer for transmission in order.

6. Once the switch receives the `Packet_Out` message, it retrieves the corresponding buffered packet from the datapath buffer and output the packet to the port specified in the `Packet_Out` message.

7. Once the switch receives the `Flow_Mod` message, it installs a new flow entry into the flow table such that all successive packets belonging to the new flow can be automatically handled by the switch without involving the controller.

6.1.2 Why Flow Entry Eviction is Important

The flow setup process is the bottleneck which limits the performance of OpenFlow and many efforts have been made to optimize it [82], [83]. The key challenge associated with this process is that the messages exchanged between the controller and switches in this process contribute the majority of the control overhead. For example, A. A. Pranata et al.

[84] proposed an framework for the controller and switch targeted on Packet_In and Packet_Out message in order to reduce control overhead. The heavy overhead of the control channel makes OpenFlow networks less scalable because the CPU of the ASIC based OpenFlow switch is limited in computation capacity and the bus bandwidth between the ASIC and switch CPU is limited as well [59], [85]. This means the delay for generating Packet_In events and installing new flow entries can be unacceptable if the volume of the control overhead is large. The measurements on Intel FM6000 OpenFlow switch show the mean delay for generating a Packet_In event is 8 ms in the case of 200 flows/s [8]. Similar measurements on Broadcom 956846K show that the latency for flow entry insertion (modification) is 3 ms (30ms), which is much higher than what native TCAM hardware can support (100M updates/s). On the controller side, D. Erickson measured the performance of different controllers and the latency for processing one Packet_In event varies between 24 us and 145 us [4]. These measurements on real devices reveal the significance of reducing the control overhead.

Based on the above discussions, we come to the conclusion that flow entry eviction is extremely important for OpenFlow network performance because every wrong flow entry eviction (i.e., evicting an active flow entry) will initiate one flow setup. In general, evicting an active flow entry will generate one extra Packet_In message, one extra Packet_Out, and more than one Flow_Mod. Furthermore, one Flow_Removed message will be generated for every eviction in general. In our system-level OpenFlow networking simulations in Section 6.4, as shown in Figure 6.15, these four OpenFlow messages generated due to wrong flow entry evictions can account for 70% control overhead.

6.1.3 Impacts of Flow Entry Eviction

The negative impacts caused by flow entry eviction are rooted in the datapath buffer in the switch and the RX/TX queue in the controller. The saved packets in the datapath buffer are retrieved until the switch receives the corresponding Packet_Out messages from the

controller. What makes it worse, all successive packets belonging to a flow have to be buffered in the datapath buffer before the flow entry corresponding to the flow is installed in the switch. And these buffered packets in turn generate more `Packet_In` events. Suppose flow arrivals follow a Poisson process with rate λ, and the packets in one flow arrive with a constant speed γ. In addition, we let the latency between saving a packet to the datapath buffer and installing the flow entry corresponding to the new flow be τ. Then we can prove that the expectation of the number of packets buffered in the datapath buffer is

$$E[N_\tau] = 0.5\lambda\gamma\tau^2 - 0.5\gamma\tau^2 + \lambda\tau - 0.5\gamma\tau, \tag{6.1}$$

where the proof is provided in the Appendix. According to the equation 6.1, the number of packets buffered in the datapath buffer grows linearly to τ^2. Besides, Openflow switches have limited memory, therefore the size of datapath buffer cannot be very large. For example, in the `ofsoftswitch13` [86], a buffered packet will be dropped after 1 second. Furthermore, λ ($>$ 1000 flows per second [6]) and γ can be large in the real network. These factors all together show that τ has a significant impact on the number of packets buffered in the datapath buffer.

In general, τ consists of the latency of generating a `Packet_In` event (e.g., 8 ms), queueing delay in the TX/RX queues in the controller, the latency for processing one `Packet_In` event for the controller (e.g., 50 us), the latency for installing a flow entry into the flow table (e.g., 3 ms), and the round-trip time (RTT) between the controller and the switch (e.g., 1 ms). Except for the queueing delay in the controller, the other components subject to physical constraints. For example, RTT can be reduced if switches are physically closer to the controller. With a more powerful switch CPU, the latency of generating a packet can be decreased. As for the queueing delay in the controller, it is directly related to control overhead. The RX and TX queues in the controller can be very large (e.g., 4GB), but they need to buffer all packets from/to all OpenFlow switches connected

to the controller. In addition, large queues also result in large queueing time. Suppose the queue size is 4GB and the size of one packet is 512 Bytes, then the last packet in the buffer needs to wait for 419s before it can be processed by the controller application if the mean latency for processing one packet is 50 us [4].

As we discussed in Section 6.1.2, control overhead is heavily affected by flow entry eviction policy. If wrong flow entry evictions are frequent, the control overhead will be large such that the queueing time can be so long that the datapath buffer is overflowed. The outcomes of datapath buffer overflow are:

1. **The buffered packets are lost**. The direct consequence of buffer overflow is that the saved packets are dropped. This will increase packet loss rate.

2. **Applications launch is delayed**. If the dropped packets are SYN packets or ARP packets, retransmissions have to be initiated. Accordingly, the launch of the corresponding applications will be delayed because the connection cannot be built.

3. **TCP connection failure**. If the retransmission of SYN/ARP packet fails for several times (e.g., 6 for SYN) because the packet is continuously deleted from the datapath buffer, TCP and ARP protocol will stop retrying transmitting SYN and ARP request, which means the connection cannot even be built and no packet belonging to the flow can ever be sent.

4. **TCP connection interruption**. If the connection can be built, packet drop will enforce recovery on TCP and the dropped packets will be retransmitted. This means the congestion window size will be shrank and the effective throughput drops.

5. **TCP connection drop**. If a packet is retransmitted for more than certain times (e.g., 3) before it is ACKed, the TCP connection will be closed and the rest packets in the flow cannot be sent.

In summarize, wrongly evicting active flow entries will increase control overhead significantly. The massive control messages make the queueing delay in the controller unacceptable such that datapath buffer is frequently overflowed. This overflow can interrupt or even stop services, which will have a serious negative impact on quality of service providing to customers.

6.2 Smart table entry eviction

As discussed at the beginning of this chapter, each flow entry can be either classified as inactive (positive) or active (negative). To apply this classification, we first need to train a binary classification model. Therefore, STEREOS mainly consists of two parts. One is offline training to generate the classification model, and the other one is online flow table eviction utilizing the trained model.

6.2.1 Offline Training: Data Collection

First and foremost, we need to collect data for training classification model. For every data point, it should contain two parts: features and label. Features are used to characterize the state of one specific flow entry, and the label indicates whether this flow entry is active or inactive. In this study, a flow is defined by five tuples, i.e., source IP address, source port number, destination IP address, destination port number, and the protocol. With this definition, we use the following features to characterize a flow entry: the protocol (we only consider TCP and UDP flows in this study) of the flow (1_{tcp}), the period of time since the last reference of the flow entry (i.e., t_{idle}=current time - the time when the flow entry is last referred), the mean and standard deviation of the inter-arrival time of the last N_{pkt} packets referring to the flow entry (t_{ia} and t_{is}), and the length (l_i) of the last N_{pkt} packets referring to the flow entry. t_{idle} captures the recentness of the flow entry, and larger t_{idle} means the flow entry is more likely to be inactive since no packet will refer to an inactive flow entry. t_{ia} and t_{is} can reflect the locality of references, and the reference tends to be more local with

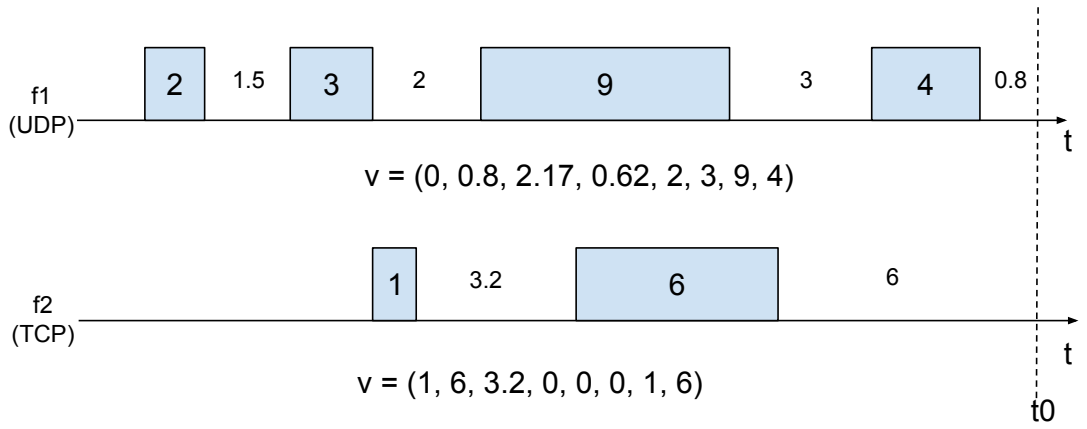


Figure 6.2: Examples of flow entry feature vector with $N_{\text{pkt}} = 4$.

Table 6.1: Feature List

Feature	Description
1_{tcp}	whether the flow is a TCP flow
t_{idle}	the period of time since the last reference of a flow entry
$t_{\text{ia}}(t_{\text{is}})$	mean (std) of the inter-arrival time of the last N_{pkt} packets referring to a flow entry
$l_i (0 \leq i < N_{\text{pkt}})$	the length of the last N_{pkt} packets referring to a flow entry

smaller t_{ia} and t_{is}. l_i of last N_{pkt} packets can reflect the communication state of a flow. For example, FIN packets will be sent when a TCP connection is terminated. A larger N_{pkt} can provide more information but consume more OpenFlow switch memories. Therefore, we should set N_{pkt} as small as possible without a significant classification accuracy loss. These features constitute the feature vector $v = (1_{\text{tcp}}, t_{\text{idle}}, t_{\text{ia}}, t_{\text{is}}, l_1, l_2, \cdots, l_{N_{\text{pkt}}}) \in \mathbb{R}^{N_{\text{pkt}}+4}$ for each flow entry. For example, Figure 6.2 shows two flow entries with $N_{\text{pkt}} = 4$. The upper flow entry is UDP and it is referred by four packets until t_0, with length 2, 3, 9, and 4. The inter-arrival time of these packets are 1.5, 2, and 3. At time t_0, the feature vector of this flow entry is $v = (0, 0.8, 2.17, 0.62, 2, 3, 9, 4)$. As for the lower TCP flow entry, there are only two packets referred to it. In the case when there are less than N_{pkt} packets referring to a flow entry, we set l_i of the "missing" packets as 0. Therefore, the feature vector of the lower flow entry at t_0 is $v = (1, 6, 3.2, 0, 0, 0, 1, 6)$. Table 6.1 summarizes all the features used in our machine learning model training.

With these identified features, we can then generate the dataset for training from real network packet traces. The generation process is described in Listing 3, which simulates the arrival of packets and updates the feature vector of the corresponding flow entry when a packet arrives. The features and label of every flow entry will be outputted as a data sample in the case of flow table overflow, where a flow entry is labeled as inactive/positive when packet p refers to this flow entry and no more packet will refer to it in the rest of the trace. Here we assume that a flow is dead if no packet belonging to it arrives within $t_{\text{threshold}}$. This is reasonable if $t_{\text{threshold}}$ is long enough (e.g., 1 hour). Note that all identified features are time-varying except for 1_{tcp}. Particularly, t_{idle} of a flow entry changes as time elapsed even if no more packet refers to it. In this case, there will be thousands of data samples which are exactly the same except that their t_{idle} are slightly different. For example, for the upper flow entry in Figure 6.2, the feature vector v will be $(0, 0.801, 2.17, 0.62, 2, 3, 9, 4)$ at time $t_0 + 0.01$ and $(0, 0.802, 2.17, 0.62, 2, 3, 9, 4)$ at $t_0 + 0.02$. To prevent this, we employ t_{record} to record the time when the features and label of a flow entry are outputted as a data sample. A flow entry cannot generate data samples within t_{interval} after its t_{record} if no packet refers to it (see line 5 in Listing 3). Another important issue is which policy (e.g., random, LRU) should be used for flow entry eviction in the case of flow table overflow. In machine learning, we would like training data and test data sets come from the same underlying data distribution such that the trained model can achieve low generalization error. If we use a policy other the random policy for flow entry eviction during dataset generation, it prefers to evict some certain flow entries and this preference is very likely to be different from the machine learning policy we want to learn. Thus the difference of the distributions between the generated training dataset and the test dataset will be large. Therefore, we evict a random flow entry when the flow table is overflowed (see line 7) because there is no preference for the evicted flow entries for the random policy. In this way, the generated dataset will have a similar distribution as the feature vectors which are fed into the trained model in the online flow table eviction.

Listing 3 Dataset generation

Require: Packet trace, $t_{\max}$, $N_{\max}$, t_{interval}, N_{pkt}, $t_{\text{threshold}}$

1: **for** Tcp/Udp packet p whose arrival time $t_p \leq t_{\max}$ **do**
2: If the next packet belonging to the flow containing p will arrive after $t_{\text{threshold}}$ or p is the last packet, label the flow as inactive. Otherwise, label it as active.
3: **if** p cannot match any flow entry **then**
4: **if** the size of the flow table is $N_{\max}$ **then**
5: Output the features and label of each flow entry which is recorded $\geq t_{\text{interval}}$ ago or updated;
6: Set each flow entry as non-updated and t_{record} of each flow entry as t_p;
7: Evict a random flow entry;
8: **end if**
9: Insert the flow entry subject to p in the flow table.
10: Set the flow entry as updated.
11: **else**
12: Update the features of the flow entry referred by p
13: **end if**
14: **end for**

6.2.2 Offline Model Training: Model Tuning

With the collected dataset, we need to select an appropriate machine learning algorithm and tune its hyperparameters to achieve the best performance. Many algorithms can be used for classification problems, such as nearest neighbors, support vector machine, decision tree, random forest, and multiple layer perception [87]. To select the best machine learning algorithm for STEREOS and tune its hyperparameters, we need to select an appropriate performance metric. There are many performance metrics for classification, such as classification accuracy, recall, and F1-score [87]. If an active flow entry is evicted from the flow table, the switch has to query the controller again to reinstall the flow entry when the packets of the flow arrive in the future. This re-installment not only incurs unexpected delays [5] but also increases controllers' workloads. Furthermore, evicting an active TCP flow entry can seriously degrade the performance of TCP connections because it may result in packet loss and congestion window shrinkage for all TCP flows which share a same switch buffer [74]. Therefore, we want to minimize false positives (i.e., active flows are misclassified as inactive). On the other hand, if an inactive flow entry is misclassified as

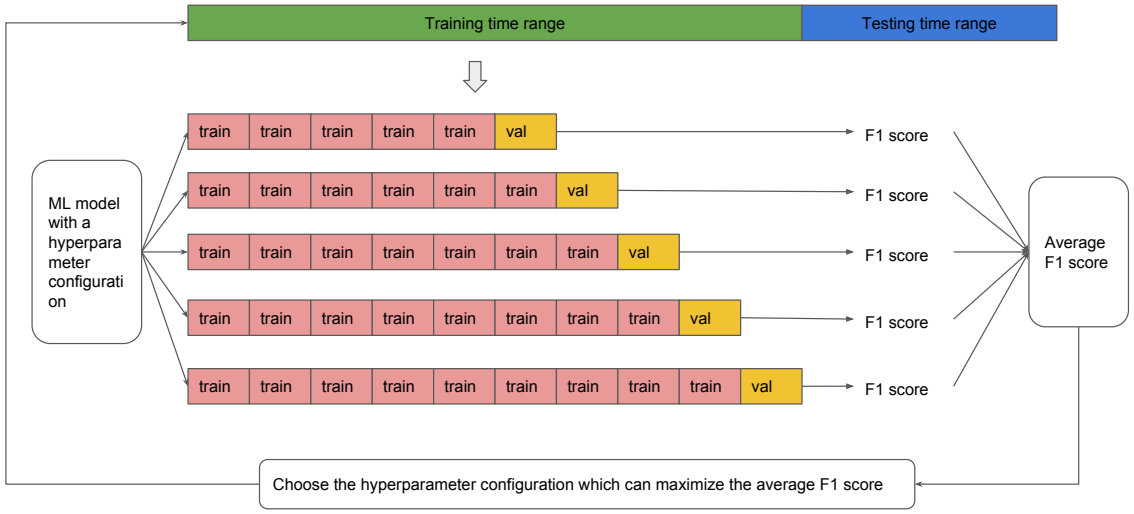


Figure 6.3: K-fold rolling-origin cross validation (K = 5) for tuning the hyperparameters of classification models.

active, then it will never be kicked out from the flow table which definitely wastes precious flow table resources. From this perspective, false negatives should be minimized as well. With this respect, we use F1 score as the performance metric. It is defined by

$$\mathrm{F1} = 2 \times \mathrm{precision} \times \mathrm{recall}/(\mathrm{precision} + \mathrm{recall}), \tag{6.2}$$

where $\mathrm{precision} = \mathrm{TP}/(\mathrm{TP}+\mathrm{FP})$, $\mathrm{recall} = \mathrm{TP}/(\mathrm{TP}+\mathrm{FN})$, TP is the number of correctly classified inactive flow entries, FP is the number of misclassified active flow entries, and FN is the number of misclassified inactive flow entries. According to the definition, a high F1 score indicates low false negatives and low false positives, which are our objectives.

Based on F1 score, we can use k-fold rolling-origin cross validation, as shown in Figure 6.3, to evaluate the performance of different machine learning models with different hyperparameter configurations. K-fold rolling-origin cross validation is a common approach to fine-tune model hyperparameters for time series data [88], where the whole time is first split into a training range and a testing range such that all observations in the training range occurred prior to any observation in the testing range. The training range is further divided

into 2K roughly equal time slices (K=5 in Figure 6.3). For fold $k = 1, 2, \cdots, K$, we fit a machine learning model with its hyperparameters to the first $K + k - 1$ time slices, and compute its F1 score in classifying the $(K+k)\text{th}$ part (a.k.a., validation slice). Then we get the average F1 score of these K fold cross validations. For every machine learning model, we do this for many values of the hyperparameters and choose the values which makes the average F1 score largest. In this way, we can find the best machine learning model and its tuned best hyperparameter values such that the model can achieve the highest F1 score. With the chosen model and hyperparameters, we can fit the model to the whole training range and use the trained model for online flow table eviction.

6.2.3 Online Flow Table Eviction

Once offline training is finished, we can apply the trained binary classification model for online flow table eviction. A straightforward idea of applying the trained model is to evict all flow entries which are classified as inactive by the model. However, this native approach may suffer from serious performance degradation. On one hand, the trained model can make wrong classifications and many misclassified active flow entries thus may be eliminated. On the other hand, one and only one flow entry is needed to be evicted when a new flow entry arrives. Therefore, it is a better way to evict one flow entry which is most likely to be inactive when the flow table is full. Fortunately, most binary classification algorithms cannot only predict whether one flow entry is inactive but also give the confidence of the prediction (i.e., the probability of being inactive for the flow entry). We rely on these probabilities for online flow entry eviction, as shown in Listing 4. Suppose we train a binary classification model h from the collected dataset, where $h(v_e)$ gives the probability of being inactive for flow entry e with feature vector v_e (i.e., P^e_{inactive}). A feature vector v is associated with each flow entry to extract the features. The feature vector contains all identified features except for t_{idle}. In principle, t_{idle} of a flow entry should be updated as time elapses while the other features should be updated only when a new packet refers to

Listing 4 Online Flow Entry Eviction

Require: Trained model h, $P_{\min}$, t_{interval}

while a packet p is arriving at the switch **do**
  if p is matched with a flow entry e_p **then**
    update the feature vector v_{e_p} associate with e_p
  else
    if the flow table is overflow **then**
      $isEvicted \leftarrow false$
      for every flow entry e in the flow table **do**
        if v_e is updated or P^e_{inactive} is updated t_{interval} ago **then**
          $P^e_{\text{inactive}} \leftarrow h(v_e)$
          if $P^e_{\text{inactive}} > 0.9$ **then**
            1) evict the entry e
            2) $isEvicted \leftarrow true$
            break
          end if
        end if
      end for
      if not $isEvicted$ **then**
        $e^* = argmax\{P^e_{\text{inactive}}\}$
        if $P^{e^*}_{\text{inactive}} > P_{\min}$ **then**
          evict the flow entry e^*
        else
          evict the least recently used flow entry
        end if
      end if
    end if
    send flow setup request to the controller to install flow entry for packet p
  end if
end while

the flow entry. To avoid frequent updates of t_{idle}, the feature vector of a flow entry contains the arrival time of the last packet referring to this entry (t_{last}). When the feature vector is fed to the trained model for classification, t_{idle} is calculated by $t_{\text{now}} - t_{\text{last}}$ where t_{now} is the current time. In this respect, when a new packet arrives, only the feature vector associated with the flow entry the packet refers to will be updated.

Another issue is to determine when the trained model should be applied for classification. When flow table overflow happens, we need to apply the trained model to find the flow entry which is most likely to be inactive. Intuitively, we can use the implemented model to compute P_{inactive} (the probability of being inactive for a flow entry) of all flow entries and evict the flow entry with the maximum P_{inactive}. This straightforward approach suffers from two disadvantages. First of all, it will be very computation intensive because flow table overflow happens very frequently. The intensive computation not only incurs a heavy workload on the weak switch management CPU but also introduces unacceptable latency. Secondly, it is meaningless to do classification on the same flow entry again if its feature vector has no or little change. For example, we do not want to classify a flow entry again if its feature vector stays the same except that t_{idle} is 1 millisecond different. To address these problems, the time when P_{inactive} is last updated is recorded in our proposal. If no packet refers to the flow entry, P_{inactive} cannot be updated within t_{interval} second, where t_{interval} is a given constant (see line 8 in Listing 4). In general, large t_{interval} makes the classification less intense but reduces the sensitivity to find an inactive flow entry. Furthermore, when a flow entry is predicted to be inactive with $P_{\text{inactive}} > 0.9$, it will be evicted immediately without updating P_{inactive} of other flow entries (see line 10-13). Otherwise, we have to find the flow entry with the maximum P_{inactive}. In this way, the frequency of doing classification is greatly reduced.

The last issue is how to kick out the misclassified inactive flow entries from the flow table. For inactive flow entries, only t_{idle} will change as time elapses. In this case, it is possible that some inactive flow entries will always be classified as active and thus reside

in the flow table forever. To make matters worse, these inactive flow entries will accumulate as time passes and occupy most of the flow table space, which seriously affects the usage of flow tables. To address this issue, we evict the least recently used flow entry if $max\{P^e_{\text{inactive}}\} \leq P_{\min}$ (see line 22). Otherwise, the flow entry with maximum P_{inactive} will be evicted (see line 20). In this way, the misclassified inactive flow entries, whose t_{idle} tends to be large, can be removed if no flow entry can be classified to be inactive with probability higher than $P_{\min}$.

6.2.4 Overheads of STEREOS

The main concern regarding the overheads of STEREOS is how large the feature vector will be, compared with the flow entry. Each flow entry contains match fields, priority, counters, instructions, timeouts, cookie, and flags. The match fields are described using OpenFlow Extensible Match (OXM) format, and each OXM occupied 5 to 259 bytes. For the classical 5-tuple flow match fields (i.e., protocol, source IPv4 address, destination IPv4 address, source TCP/UDP port, destination TCP/UDP port), 4 OXMs (i.e., OXM_OF_IPV4_SRC, OXM_OF_IPV4_DST, OXM_OF_TCP_SRC/ OXM_OF_UDP_SRC, OXM_OF_TCP_DST/ OXM_OF_UDP_DST) are required and 28 bytes will be consumed. The priority needs 2 bytes, timeout 2 bytes, cookie 16 bytes, and flags 4 bytes. Depending on which counters are included in the flow entry, the match field of counters consumes 4 to 24 bytes. As for the field of instructions, the number of bytes it occupies depends on what actions are included. For example, if one output action is included, then the field of instructions will occupy 24 bytes. In summary, the overhead of one flow entry varies depending on what match fields are used, what counters are included, and what actions are applied. For the classical 5-tuple output flow entry, it takes 80 bytes. As for the overhead for STEREOS, each flow entry will be associated with a feature vector, which includes idle time, the packet length and inter-arrival time of the last N_{pkt} packets referred to the flow entry. Therefore, if we use B_t bytes to quantize the time features, and B_l to quantize packet length, then the overhead of

Table 6.2: Summary of packet traces used for case study

Packet trace	Duration (s)	Number of packets	Number of flows	TCP flow percentage
UNIBS20090930	81,203	4,189,545	43,489	0.973
UNIBS20091001	32,407	3,321,426	39,730	0.974
UNIV1	3,914	17,131,142	439,133	0.674
UNIV2	3,558	35,243,160	38,016	0.087

machine learning eviction policy is $B_t N_{pkt} + B_l N_{pkt}$. We will talk about the value of N_{pkt}, B_t, and B_l in the following section.

6.3 Case Study

In this section, we present case studies based on four real packet traces collected from university datacenters. Besides UNIV1 and UNIV2 used in Section 5.5, another two packet traces, UNIBS20090930 and UNIBS20091001 which are collected on the edge router of the campus network of the University of Brescia on two working days (i.e., 09/30/2009 and 10/01/2009) [89], are also used. They are composed of traffic generated by a set of twenty workstations running the ground truth client daemon. For simplicity, we only consider TCP and UDP flows in this case study. We summarize these four packet traces in Table 6.2.

6.3.1 Dataset collection

To collect data for learning, we build an OpenFlow simulator which can replay packet traces according to the OpenFlow specification [3] and collect data according to Listing 3. Our simulator contains two objects: an OpenFlow switch and a controller. All the packets in a packet trace will be replayed and arrive at the OpenFlow switch. When a packet p in the trace arrives, the switch will check whether any flow entry in the flow table can match with p. If none of the entries can match, a `PacketIn` message will be sent to the controller. Once received a `PacketIn`, the controller will instruct the switch to install a new flow entry with respect to p. Furthermore, the switch will update the feature vectors

Table 6.3: Summary of generated datasets

Packet trace	t_{max}	t_{int}	Train flows	Test flows	Cross flows	Table size	Dataset size	Class ratio
UNIBS 20090930	10000	20	11283	33076	870	1K	563537	1.022:1
						2K	829985	2.023:1
						4K	1037730	3.434:1
UNIBS 20091001	5000	10	7284	33181	735	1K	230977	1.610:1
						2K	381279	2.958:1
						4K	438524	5.110:1
UNIV1	600	1	95503	398395	54765	1K	874663	0.747:1
						2K	1420374	1.462:1
						4K	2472854	2.797:1
UNIV2	550	1	19073	34540	15597	1K	873753	0.070:1
						2K	1336900	0.109:1
						4K	2229184	0.164:1

of flow entries and output data samples according to Listing 3. For all four packet traces, we set the size of flow table ($N_{\max}$) to be 1K, 2K, and 4K, which are compatible with the configurations in many studies [48], [42]. As for N_{pkt}, we set it 10 for all traces and table sizes. All these generated datasets are summarized in Table 6.3. Here we define train flows as the flows which start in the train duration t_{max}, test flows as the flows start beyond t_{max}, cross flows as the flows start within t_{max} and end beyond t_{max}. The reason why we want to distinguish these flows is that we want to figure out whether the learned pattern of train flows can also apply to test flows. This is significant because the benefits of STEREOS will degrade as time elapsed if the learned pattern cannot apply to test flows. In addition, we can see that the class ratio (i.e., the ratio of positive to negative instances) increases as the flow table size becomes larger. This is because a flow entry resides in the flow table for a longer time and thus be more likely to be inactive when eviction happened if the flow table is larger. Furthermore, except for UNIV2, the datasets of the other 3 traces do not suffer from the problem of imbalance. This is because most flows in UNIV2 last for a very long time and thus most of the flow entries in the flow table are active.

6.3.2 Offline training results

We use `scikit-learn` [77], an open source machine learning library in Python, for selecting the best machine learning models. This library provides many classification algorithms, as well as the APIs for model selection based on cross-validation. As shown in Figure 6.3, each piece of the last five periods $t_{max}/10$ is a validation set and we consider the previous time's data is the training set. In this way, we can apply 5-fold rolling-origin-cross-validation to tune the hyperparameters of different machine learning algorithms. In this study, we consider seven classification algorithms: gradient boosting tree (GBT), decision tree (DT), random forest (RF), Ada boosting (Ada), logistic regression (LR), neural networks (NN), and Gaussian naive Bayes (GNB). The advantages and disadvantages of these algorithms are well summarized in [90], [91]. We do not consider K nearest neighbor algorithm because it requires to memorize the whole dataset which is not feasible for the OpenFlow switches with limited memory. We also do not consider support vector machine (SVM) algorithm because it is very computationally expensive for training. For each of the considered algorithm, it has many hyperparameters and we only consider some of the most import ones[1], which are shown in Table 6.4. All the other hyperparameters not specified in Table 6.4 use the default values provided by the library.

All possible combinations of the hyperparameter values specified in Table 6.4 are evaluated by 5-fold rolling-origin-cross-validation in terms of F1 score. In this way, we can pick the best algorithm and its best tuned hyperparameters in terms of the achieved F1 score, which are shown in Table 6.5. As we can see, GBT and RF are the best two algorithms for all four packet traces (their achieved F1 score fall within about 0.5% of each other), which are consistent with the results from [92]. These two methods are both ensemble methods which are unlikely to overfit. Another important observation is that the F1 score for the validation on the non-cross flows of UNIBS packet traces is slightly smaller than that on

[1]Refer to `https://scikit-learn.org/stable/supervised_learning.html#supervised-learning` for the definitions of these hyperparameters.

Table 6.4: Hyperparameter search space for model tuning

Algorithm	Hyperparameter	Search space
RF	n_estimators	[10, 20, 30]
	criterion	[entropy, gini]
	max_depth	[10, 20, 30]
DT	criterion	[entropy, gini]
	max_depth	[10, 20, 30]
Ada	n_estimators	[10, 20, 30]
	learning_rate	[0.8, 0.9, 1.1]
GBT	n_estimators	[10, 20, 30]
	subsample	[0.6, 0.8, 1.0]
	learning_rate	[0.01, 0.1, 0.5]
	max_depth	[10, 20, 30]
LR	penalty	[L1, L2]
	C	[0.01, 0.1, 1.0]
NN	alpha	[0.01, 0.1, 1.0, 10.0]
	learning_rate	[constant, invscaling, adaptive]
	learning_rate_init	[0.01, 0.1, 1.0, 10.0]
	hidden_layer_sizes	[(30), (15, 15), (10, 10, 10)]

the cross flows, while for UNIV packet traces, the F1 score of non-cross flows is higher than cross ones. It demonstrates that the learned pattern of one flow can be generalized to another unseen flow. Otherwise, the score of the non-cross flows will be much lower than the cross ones.

After we select the best model and determine its best-tuned hyperparameters for each dataset, we train the best-tuned machine learning model on the whole training duration t_{max} and save the trained model with the help of `joblib`, which will be loaded for online flow entry eviction.

6.3.3 Online simulation results

We carried out simulations on the four packet traces with different flow table sizes (1K, 2K, and 4K). The simulator is similar to the one used in dataset generation except that the flow entry eviction policy can be configured. We have implemented two policies, machine learning policy and LRU policy, for performance comparison. For machine learning policy,

Table 6.5: Model selection results

Packet trace	Table size	Best model in terms of F1 score	Train	Validation on cross flows	Validation on non-cross flows	Validation
UNIBS 20090930	1K	gbt { n_estimators:30; subsample:0.8; learning_rate:0.1; max_depth:10; }	0.98784	0.98380	0.93758	0.96171
	2K	gbt { n_estimators:30; subsample:0.8; learning_rate:0.1; max_depth:10; }	0.99499	0.99303	0.95014	0.98106
	4K	gbt { n_estimators:30; subsample:0.6; learning_rate:0.1; max_depth:10; }	0.99775	0.99494	0.94634	0.98811
UNIBS 20091001	1K	rf { n_estimators:30; criterion:entropy; max_depth:20; }	0.99662	0.99454	0.93710	0.97660
	2K	gbt { n_estimators:30; subsample:1.0; learning_rate:0.1; max_depth:10; }	0.99497	0.99661	0.94930	0.98861
	4K	rf { n_estimators:30; criterion:gini; max_depth:20; }	0.99928	0.99901	0.94492	0.99526
UNIV1	1K	rf { n_estimators:30; criterion:entropy; max_depth:20; }	0.93629	0.79215	0.93148	0.90258
	2K	rf { n_estimators:30; criterion:entropy; max_depth:20; }	0.96081	0.91783	0.95195	0.93946
	4K	rf { n_estimators:30; criterion:entropy; max_depth:20; }	0.97890	0.96687	0.95983	0.96435
UNIV2	1K	gbt { n_estimators:30; subsample:0.8; learning_rate:0.1; max_depth:10; }	0.94876	0.69402	0.93979	0.89006
	2K	gbt { n_estimators:30; subsample:0.6; learning_rate:0.1; max_depth:20; }	0.96857	0.84351	0.94421	0.90926
	4K	gbt { n_estimators:20; subsample:0.6; learning_rate:0.1; max_depth:20; }	0.98324	0.94429	0.95206	0.94790

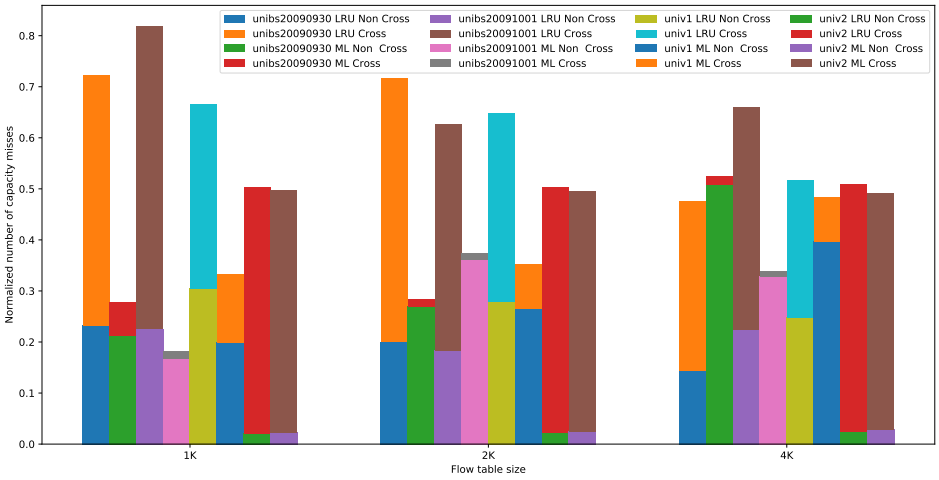


Figure 6.4: The performance of machine learning eviction policy and LRU policy in terms of number of capacity misses

the switch determines which flow entry should be evicted in the case of flow table overflow according to Listing 4, where $P_{\min} = 0.65$ and t_{interval} is set according to Table 6.3. And the models used for prediction are generated in the last subsection. For LRU policy, the switch kicks out the least recently used flow entry when the flow table is full.

First, we investigate the number of capacity misses just like in Section 5.5.3. Figure 6.4 shows the performance of STEREOS in terms of the number of capacity misses, where the number of capacity misses of one packet trace for each policy is normalized with respect to the total number of capacity misses across all policies for that trace. From Figure 6.4, we have the following observations:

1. For UNIV2 trace, STEREOS can only achieve less than 5% performance gain. This is because the number of active flows in UNIV2 is always much larger than the flow table size, as shown in Figure 6.5. It means that most of the flow entries in the flow table are always active, and thus STEREOS cannot outperform LRU policy since our proposal is based on the assumption that some of the flow entries are inactive. Fortunately, this situation is not typical in the real networks because most of the flows in UNIV2 are UDP flows, as shown in Table 6.2. According to the report from [50],

UDP data volume is only 3% $\sim$ 9% of TCP data. However, in UNIV2, most flows are UDP and UDP data dominates the trace (refer to Table 6.2). Since STEREOS only slightly outperforms LRU for UNIV2, we will not investigate it in the rest of our discussions.

2. For the other three packet traces, we can see that the performance gain of STEREOS generally decreases as the flow table size increases. In the case of 1K and 2K flow table, STEREOS can achieve over 45% fewer capacity misses on the UNIV1 packet trace and over 60% on the UNIBS20090930 and UNIBS20091001 traces. Especially, for the UNIBS1001, our proposal can achieve 78% fewer capacity misses in the case of 1K flow table. We should add that reducing capacity misses is extremely important for OpenFlow network performance. On one hand, fewer capacity misses will reduce the number of `PacketIn` events, and thus relieve the load on control channels and controllers. On the other hand, fewer capacity misses means that fewer TCP transmissions are interrupted. With this respect, the over 45% performance gain in terms of capacity miss achieved by our proposal is very significant. We will further demonstrate this argument in Section 6.4. However, with 4K flow table size, for example, STEREOS can only reduce the number of capacity misses on the UNIV1 trace by 7% compared with LRU policy. And for UNIBS20090930, STEREOS even performs worse than LRU. This is because it is more likely for LRU policy to remove an inactive flow entry with a larger flow table. In addition, the class ratio is even more imbalanced in the case of 4K flow table, which has negative impacts on the prediction accuracy.

3. Cross flows contribute most capacity misses in the case of LRU policy although the number of cross flows is much less than non-cross flows, as shown in Table 6.3. For example, the number of cross flows is only 2.6% of all test flows, but cross flows contribute more than 67% capacity misses. This is because the packet inter-arrival

time of cross flows are larger than the non-cross flows, as shown in Figure 6.6. The flow entry with larger packet inter-arrival time is more likely to be evicted with LRU policy. In this way, cross flows contribute most capacity misses.

4. The performance gain of machine learning is mainly from the gain of cross flows. For example, with 2K flow table, the number of capacity misses on cross flows is reduced over 97% for UNIBS packet traces and 76% for UNIV1 trace. However, for the non-cross flows, the capacity misses are surprisingly increased for UNIBS traces and only decreased by 4% for UNIV1 trace. One straightforward guess for this is that our machine learning model does not learn the pattern of the non-cross flows because non-cross flows are not seen by the model during training. However, as we discussed in Section 6.3.2, the offline training results show that the learned pattern of train flows can be generalized to unseen flows. A capacity miss is generated when an active flow is misclassified as inactive and thus be evicted. According to Listing 4, STEREOS only evicts the flow entry with the highest inactive probability. Therefore, if the classification accuracy of active cross flows is higher than that of non-cross flows, an active cross flow entry will be much less likely to be wrongly evicted and thus fewer capacity misses will be generated on the cross flows, which is the exact case as shown in Figure 6.7.

Second, we investigate the number of active flow entries in the flow table. Figure 6.8 shows the active flow entries in the flow table with machine learning and LRU policies on the UNIV1 packet trace. As we can see, the number of active flow entries in the flow table with STEREOS is much larger than LRU. On average, STEREOS can increase the usage of the flow table with 1K, 2K, and 4K capacity by 58%, 60%, and 54% respectively, compared with LRU. This significant improvement is achieved because STEREOS can correctly identify and evict inactive flow entries when flow table overflow occurs. In contrast, LRU may frequently remove active flow entries and leave inactive flow entries in the flow table. Combining Figure 6.4 and 6.8, we can reach the conclusion that our machine learn-

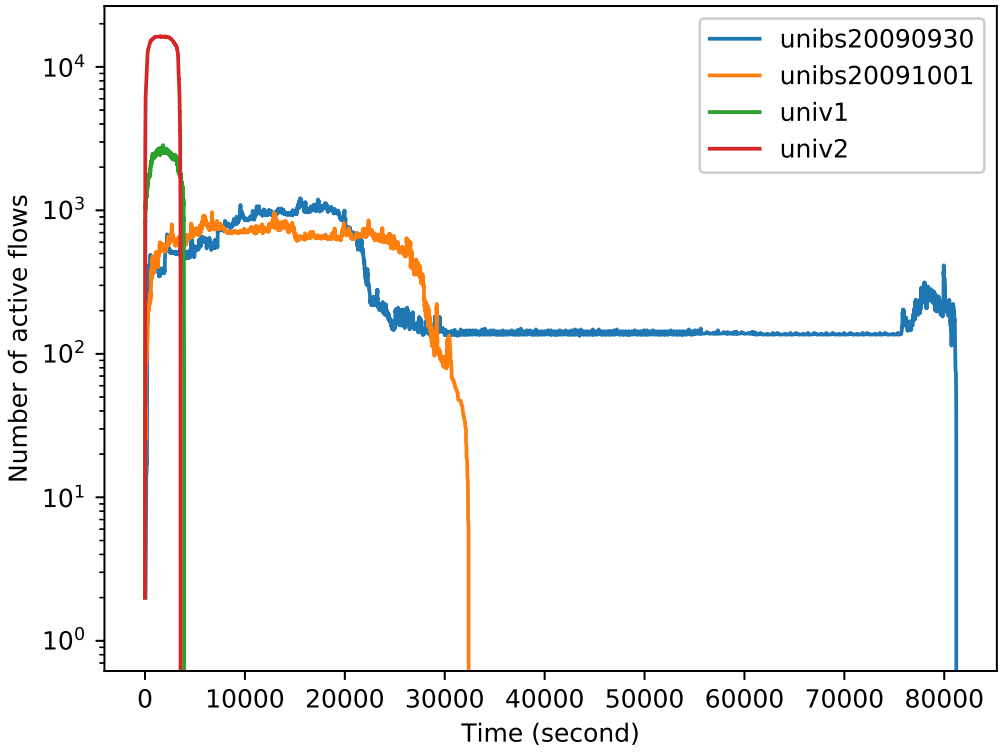
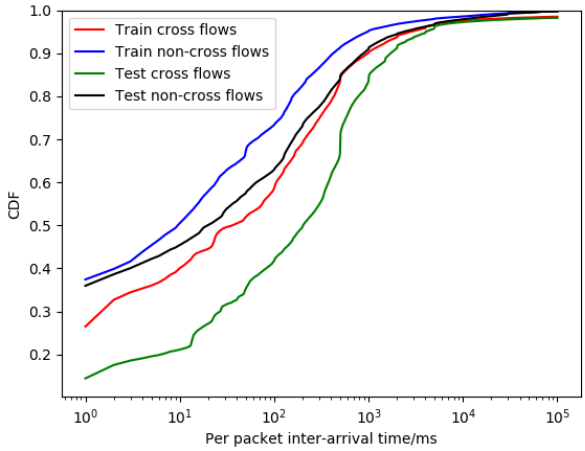
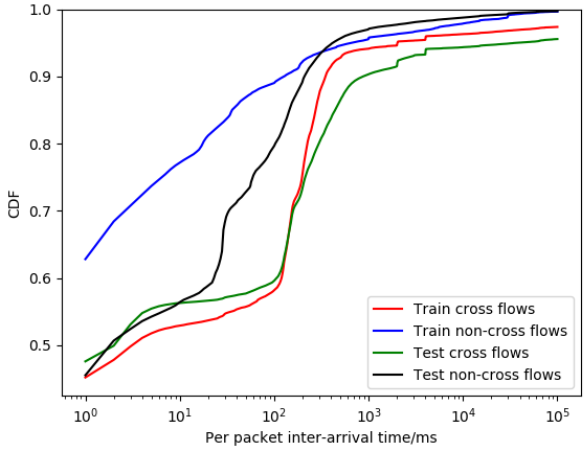


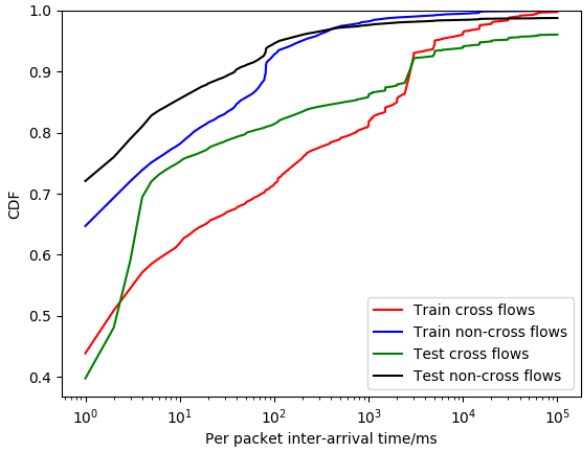
Figure 6.5: The number of active flows



(a)

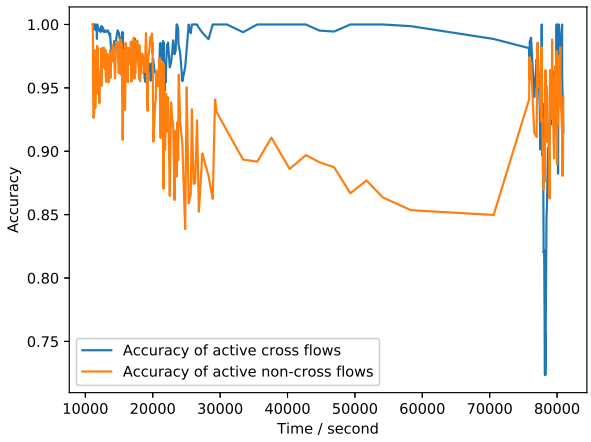


(b)

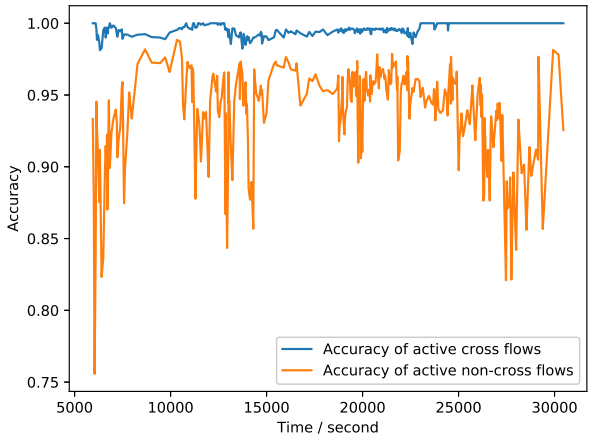


(c)

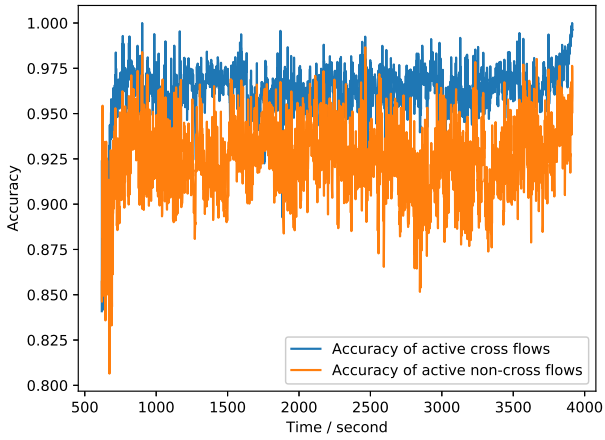
Figure 6.6: The distribution of per packet inter-arrival time. (a) UNIBS20090930; (b) UNIBS20091001; (c) UNIV1.



(a)



(b)



(c)

Figure 6.7: The prediction accuracy of active cross and non-cross flows with 2K flow table. (a) UNIBS20090930; (b) UNIBS20091001; (c) UNIV1.

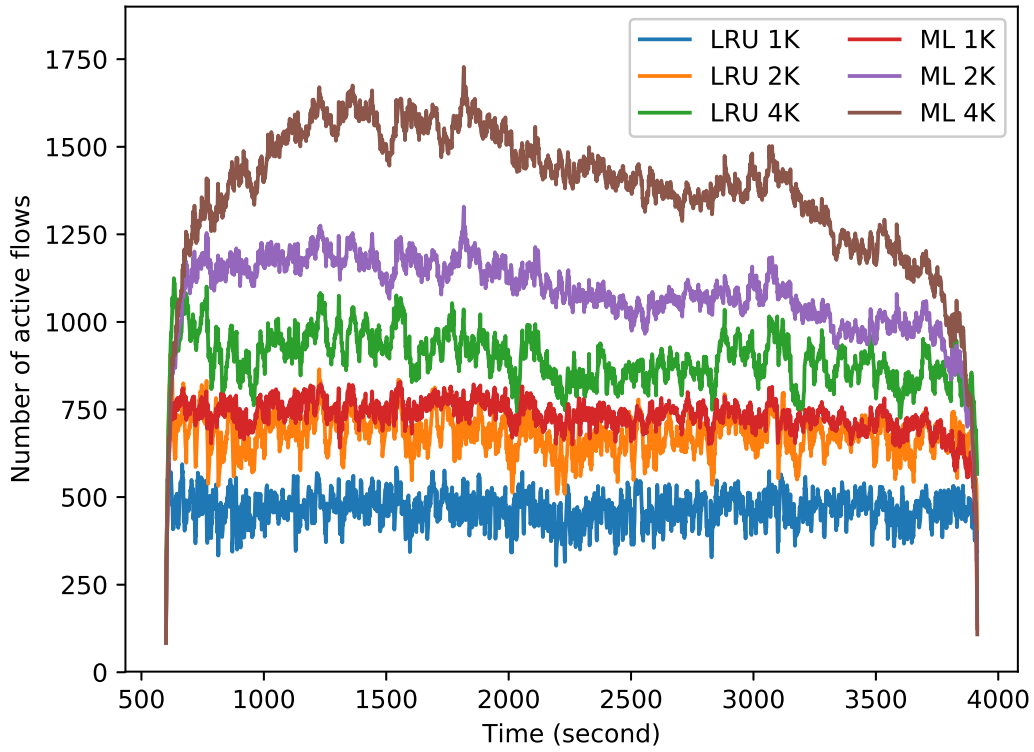


Figure 6.8: Number of active flow entries in the flow table for the UNIV1 packet trace.

ing based flow entry eviction policy can achieve significant performance gain compared with LRU policy.

6.3.4 Model size trade-off

When we do model selection in Section 6.3.2, we only consider the F1 score a model can achieve. This may suffer from two problems. First, F1 score cannot fully describe the performance of a model when it is applied to STEREOS. Although F1 score makes a trade-off between false positives and false negatives, it may not perfectly reflect modes' relative performance to reduce the number of capacity misses, especially in the case where F1 score varies a little. Second, OpenFlow switches are always limited in memory. Therefore, we also need to consider the model size when selecting models. Table 6.6 shows the performance of different models with their size, where we have the following observations:

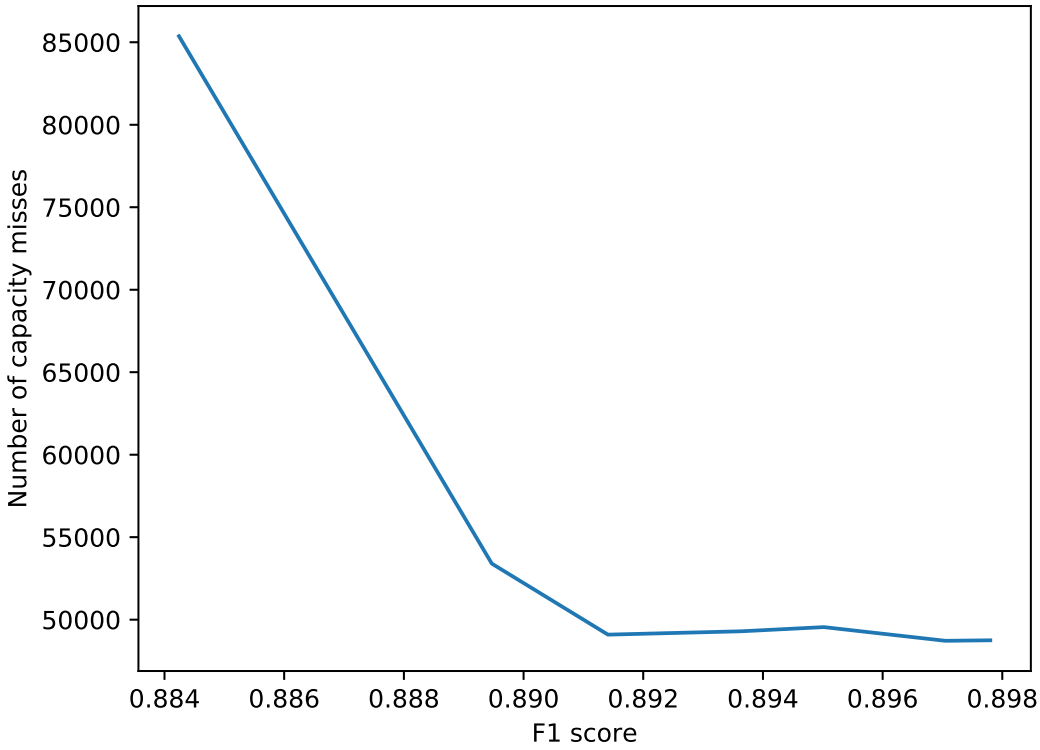


Figure 6.9: The effects of F1 score on STEREOS.

1. The model with the highest F1 score may not achieve the best performance in terms of reducing capacity misses. For example, in the case of UNIBS20090930 with 4K flow table, the best model in terms of F1 score results to 1145 capacity misses, which is actually even larger than LRU policy (i.e., 1037). However, the random forest model (`n_estimators`: 20, `criterion`: entropy, `max_depth`: 20) achieves only 489 capacity misses although its F1 score is slightly smaller than the "best" model. The reason for this phenomena may due to the fact that there is some randomness in terms of the number of capacity misses for models with close F1 score, which is shown in Figure 6.9. This figure is achieved through running simulations on UNIV1 trace with 1K flow table, where the model used for evicting flow entry is GBT (subsample:0.8; learning_rate:0.1; max_depth:10) with `n_estimators`={`5, 10, 15, 20, 25, 30, 35`}. As we can see, as the F1 score within the range of [0.892, 0.898], the number of capacity misses fluctuates little.

2. Smaller model can achieve even fewer capacity misses. For example, for UNIV1 trace with 1K flow table, the "best" model (i.e., RF model) takes 41.07 MB and achieves 49,603 capacity misses. However, the GBT model (`n_estimators`:30; `subsample`:0.8; `learning_rate`:0.1; `max_depth`:10) only consumes 3.31 MB but achieves 48,719 capacity misses.

Based on the above two observations, we re-select the models for different packet traces in the rest of our discussions considering both model size and performance, which are marked with "$\star$" in Table 6.6.

6.3.5 Tuning P_{min}

We also investigate the effect of $P_{\min}$ on our proposal. We conducted simulations on the UNIV1 trace with 1K flow table and different $P_{\min}$. As we can see from Figure 6.10, the number of capacity misses decreases up to a certain $P_{\min}$, then increases as $P_{\min}$ grows. For example, the number of capacity misses is reduced by 24% when $P_{\min}$ is changed from

Table 6.6: Performance and model size trade-off.

Packet trace	Table size	F1 score	Model size (MB)	Model hyperparameters	Capacity misses
UNIBS 20090930	1K	0.96060	1.79	gbt⋆ { n_estimators:20; subsample:0.6; learning_rate:0.1; max_depth:10; }	2227
		0.95104	0.08	dt { criterion:entropy; max_depth:10; }	4312
		0.96171	2.75	gbt* { n_estimators:30; subsample:0.8; learning_rate:0.1; max_depth:10; }	2410
	2K	0.98037	1.70	gbt⋆ { n_estimators:20; subsample:0.8; learning_rate:0.1; max_depth:10; }	1023
		0.97606	0.08	dt { criterion:gini; max_depth:10; }	2623
		0.98106	2.65	gbt* { n_estimators:30; subsample:0.8; learning_rate:0.1; max_depth:10; }	1181
	4K	0.98759	1.84	gbt { n_estimators:20; subsample:0.6; learning_rate:0.1; max_depth:10; }	1118
		0.98531	13.48	rf⋆ {n_estimators: 20, criterion: entropy, max_depth: 20}	489
		0.98811	2.73	gbt* { n_estimators:30; subsample:0.6; learning_rate:0.1; max_depth:10; }	1145
UNIBS 20091001	1K	0.97606	8.01	rf { n_estimators:20; criterion:entropy; max_depth:20; }	2150
		0.97502	1.02	gbt⋆ { n_estimators:20; subsample:0.8; learning_rate:0.1; max_depth:10; }	2074
		0.97645	13.08	rf* { n_estimators:30; criterion:entropy; max_depth:20; }	1912
	2K	0.98833	0.98	gbt⋆ { n_estimators:20; subsample:0.6; learning_rate:0.1; max_depth:10; }	1406
		0.98604	0.05	dt { criterion:gini; max_depth:10; }	2697
		0.98857	1.50	gbt* { n_estimators:30; subsample:1.0; learning_rate:0.1; max_depth:10; }	1705
	4K	0.99515	5.21	rf⋆ { n_estimators:20; criterion:entropy; max_depth:20; }	405
		0.99480	0.93	gbt { n_estimators:20; subsample:0.8; learning_rate:0.1; max_depth:10; }	1413
		0.99526	8.24	rf* { n_estimators:30; criterion:gini; max_depth:20; }	410
UNIV1	1K	0.89705	3.31	gbt⋆ { n_estimators:30; subsample:0.8; learning_rate:0.1; max_depth:10; }	48719
		0.89364	2.22	gbt { n_estimators:20; subsample:0.8; learning_rate:0.1; max_depth:10; }	49296
		0.90258	41.07	rf* { n_estimators:30; criterion:entropy; max_depth:20; }	49603
	2K	0.93673	14.42	rf { n_estimators:10; criterion:entropy; max_depth:20; }	31607
		0.93385	3.34	gbt⋆ { n_estimators:30; subsample:0.6; learning_rate:0.1; max_depth:10; }	30794
		0.93946	46.47	rf* { n_estimators:30; criterion:entropy; max_depth:20; }	28428
	4K	0.96389	30.88	rf { n_estimators:20; criterion:entropy; max_depth:20; }	22497
		0.95862	3.29	gbt⋆ { n_estimators:30; subsample:0.8; learning_rate:0.1; max_depth:10; }	20664
		0.96435	46.14	rf* { n_estimators:30; criterion:entropy; max_depth:20; }	22568

Note: The best model in terms of F1 score is marked with "*", and the re-selected model considering both model size and performance is marked with "⋆".

0.5 to 0.65, and increased by 74% from 0.65 to 0.9. This is because small $P_{\min}$ allows the switch to evict flow entries which are classified as inactive with low confidence. In this case, it is highly possible that a misclassified active flow entries will be evicted. In contrast, large $P_{\min}$ will prevent the switch from evicting inactive flow entries which are not identified by the trained model with very high confidence. Actually, with large $P_{\min}$, the switch will heavily rely on the LRU policy (line 22 in Listing 4) for eviction instead of the machine learning one. Then, how can we set $P_{\min}$ properly? According to Listing 4, a flow entry with $P_{\text{inactive}} > P_{\min}$ will be evicted in the case of flow table overflow (see line 20). Suppose the total number of such evictions is N_e. Then the objective of our proposal is to maximize the number of right evictions $N_{\text{right}} = N_e * (1 - P[y = 0|P_{\text{inactive}} > P_{\min}])$, where $y = 0$ indicates the flow entry is active. Note that minimizing the number of wrong evictions is different from maximizing the number of right evictions. If we want to minimize the number of wrong evictions, we can just set $P_{\min} = 1$ such that N_e will approximate 0. In this case, our proposal will be meaningless because eviction decisions seldom depend on the predictions of the trained random forest model.

Given $P_{\min}$, N_{right} can be approximated from the dataset generated by Listing 3. In the dataset, every data sample has a label (i.e., y). Furthermore, with the trained model, we can calculate P_{inactive} for every data sample. Therefore, N_{right} can be estimated through:

$$N_e \approx N(P_{\text{inactive}} > P_{\min}), \tag{6.3}$$

$$P[y = 0|P_{\text{inactive}} > P_{\min}] \approx \frac{N(P_{\text{inactive}} > P_{\min} \wedge y=0)}{N(P_{\text{inactive}} > P_{\min})}, \tag{6.4}$$

where $N(\cdot)$ is a function returns the number of elements satisfying a predicate. Combining (2) and (3), we can get

$$N_{\text{right}} \approx N(P_{\text{inactive}} > P_{\min} \wedge y = 1). \tag{6.5}$$

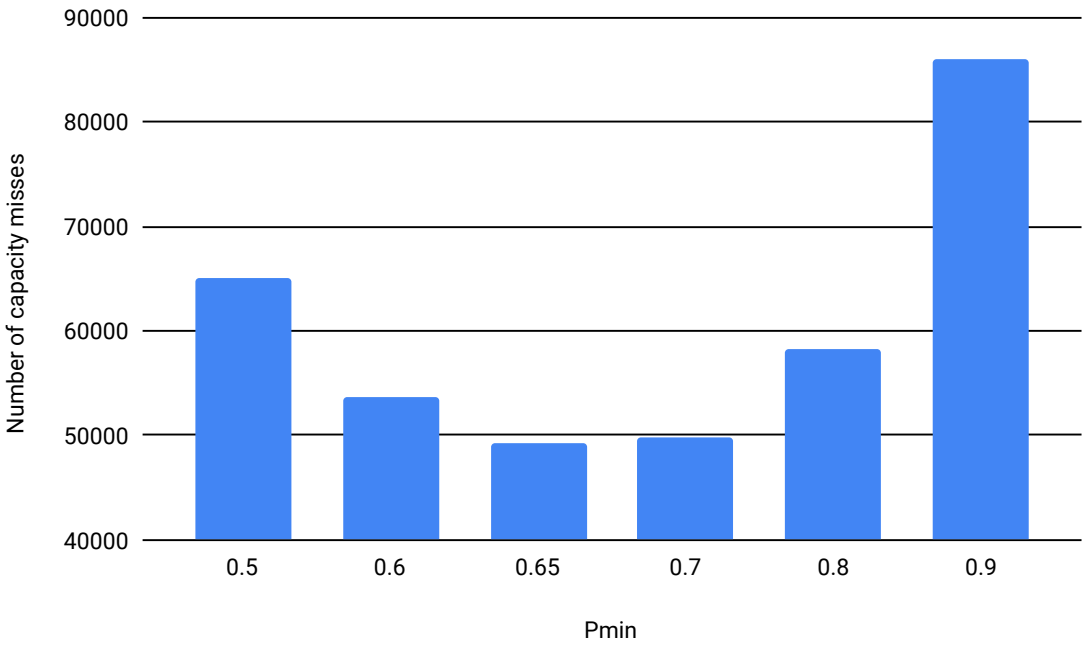


Figure 6.10: The effects of P_{min} on our proposal.

Therefore, we can set P_{min} by

$$P^*_{\text{min}} = \underset{P_{\text{min}}}{\text{argmax}}\, N(P_{\text{inactive}} > P_{\text{min}} \wedge y = 1). \tag{6.6}$$

In the case of the UNIV1 trace with 1K flow table, P_{min} generated by (6.6) is 0.65, which is same as the optimal value in Figure 6.10.

6.3.6 Feature selection

As we discussed in Section 6.2.4, the overhead of our proposal is directly related to N_{pkt}. In our above experiments, we set N_{pkt} to be 10, which may incur heavy overhead. In this subsection, we study how N_{pkt} can affect the performance of STEREOS on UNIV1 trace. We first generate the datasets with $N_{\text{pkt}} = 5, 6, 7, 8, 9$ for UNIV1 traces. Then we carried out simulations with 1K, 2K, and 4K flow tables, where the models used for online flow table eviction are the ones selected in Table 6.6. The simulation results are shown

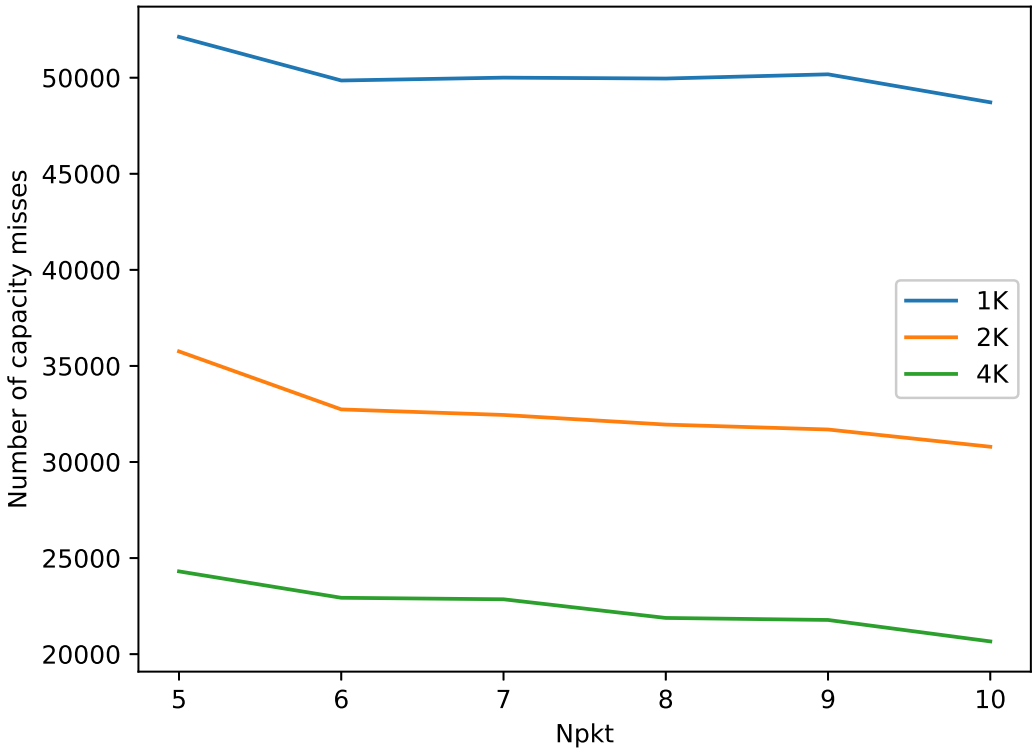


Figure 6.11: The effects of N_{pkt} on our proposal.

in Figure 6.11. As we can see, the number of capacity misses is generally increased as N_{pkt} decreased. For example, the number of capacity misses is increased by 7%, 16%, and 17.6% when N_{pkt} is reduced from 10 to 5 for the 1K, 2K, and 4K table, respectively. This is because larger N_{pkt} can, in general, provide more information for the model and thus help increase the classification accuracy. In this way, STEREOS can be more likely to evict inactive flow entries. On the other hand, larger N_{pkt} means the OpenFlow switch need more memory to store the feature vectors. For example, when N_{pkt} is increased from 5 to 10, the memory cost of storing feature vectors will almost double. Therefore, we need to make a trade-off between memory consumption and classification accuracy in practical implementation.

6.3.7 Feature quantization

Not only N_{pkt} affects the overhead of our proposal, but also B_t and B_l. We use uniform quantization in this study [93], i.e., the quantization level q of value $v > 0$ is given by

$$q = \begin{cases} 2^B - 1, & \text{if } v > v_{max} \\ 0, & \text{not exist} \\ \lceil v/\Delta v \rceil, & \text{otherwise} \end{cases} \tag{6.7}$$

where B is the number of bits used for quantization, v_{max} is the maximum value to be quantized, and $\Delta v = v_{max}/(2^B - 1)$. Note there is a special case, if a value does not exist, it also should be quantized. For example, in Figure 6.2, there is neither the third nor fourth packet referring to the flow entry, the corresponding features for packet length should be quantized as 0.

The size of a packet is limited, for example, Microsoft Windows computers default to a maximum packet size of 1500 bytes for broadband connections. And the maximum transmission unit (MTU) of Ethernet is 1500 bytes. Given these facts, we argue that $B_l = 1$ is enough to quantize packet length features. Accordingly, we set the v_{max} for packet length to be 1500.

As for the time features (t_{idle} and t_{ia}), we tried 1 byte and 2 bytes for quantization, and the results are shown in Table 6.7. As we can see, the performance of our proposal slightly degrades when quantization is applied in most scenarios. For example, for UNIV1 with 1K flow table, the number of capacity misses is increased by 2% and 1.4% with 1 byte and 2 bytes quantization, respectively. In other cases, quantization even helps to slightly improve the performance of STEREOS. For example, for UNIV1 with 2K flow table, capacity misses are reduced by 4% (1 byte quantization) and 6% (2 byte quantization). The reason for this may due to the fact that quantization can drop some unnecessary information for training, and thus increase the model's accuracy. Furthermore, it does not help a lot to use

Table 6.7: Performance of STEREOS with feature quantization

Packet trace	Table size	LRU	No quan	v_{max} (1 B)	Capacity misses (1 B)	v_{max} (2 B)	Capacity misses (2 B)
UNIBS 20090930	1K	6293	2150	250	2553	500	2273
	2K	2973	989	300	957	600	1126
	4K	1037	388	450	664	700	560
UNIBS 20091001	1K	8620	1922	250	2252	600	1957
	2K	2857	1235	300	1364	1000	1371
	4K	797	423	450	619	1500	517
UNIV1	1K	99032	52125	10	53193	20	52839
	2K	52168	35756	10	34189	50	33638
	4K	24212	24308	90	24554	150	20977

2 bytes for quantization, compared with the 1 byte case. In summary, 1 byte for quantizing the time feature is enough to achieve very close performance to the case where no quantization is applied. So far, we have discussed the values for N_{pkt}, B_t, and B_l, and we can say that the overhead of STEREOS is acceptable since it only requires extra 10 bytes for each flow entry, which typically takes more than 80 bytes according to the OpenFlow protocol [3].

6.3.8 Model interpretation

So far, we have done a detailed investigation on the performance of STEREOS, and all experiments show that STEREOS outperforms LRU. A straightforward question arises, why STEREOS does better? This problem is actually about how to interpret a machine learning model, which is still far from being solved [94].

Since this case study involves tens of models, we only try to interpret the final model for UNIV1 trace with 1K flow table, which is GBT (`n_estimators`:30; `subsample`:0.8; `learning_rate`:0.1; `max_depth`:10) with $N_{pkt} = 5$ and uses 1 byte to quantize the time and packet length features. Through this interpretation, although we cannot uncover the full truth behind machine learning's power in flow entry eviction, some hints can be given to make the story more understandable.

Our interpretations are based on SHAP framework [95], which assigns each feature an

importance (i.e., SHAP value) for a particular prediction such that the generated explanation model follows the definition of additive feature attribution methods and subjects to the property of local accuracy, missingness, and consistency. The larger —SHAP value— of a feature is, the larger magnitude change in model output due to the feature is.

We first check the global mean of the absolute value of SHAP values for each feature, which are shown in Figure 6.12. We can see that t_{idle} is actually the most important (1.35), which is the only feature used in LRU policy. However, the following features (l_5, l_4, l_1, t_{is}) are also important, which account for over 1.74. With this regard, it is reasonable that LRU policy loses to machine learning one.

Since SHAP framework provides individualized explanations for every data sample, we also plot the SHAP values of every feature for every sample, as shown in Figure 6.13. We again observe that t_{idle} is the strongest predictor of deciding whether one flow is inactive. And the larger t_{idle} is, the more contribution of this feature in determining the flow is inactive is. This is exactly the heuristic used in LRU policy. In addition, for the feature l_5, we find the pattern that small packet indicates that the flow is more likely to be inactive and large one indicates it is active. This is reasonable because a packet carrying data tends to be large, and signaling packets are small. In addition, we also find that large l_2 and l_3 indicate that the flow may be inactive, while small ones are good signs of the flow being active.

6.4 System-level Simulation Results

So far, we use the number of capacity misses to evaluate the performance of STEREOS. However, we are more interested in how the reduction of capacity misses will affect the network performance in terms of throughput, delay, and packet loss rate. To learn this, we present system level simulation experiments based on Network Simulator 3 (NS3) in this subsection. In these simulations, we use a classical datacenter topology as shown in Figure 6.14. This topology consists of two layers of switches. The bottom layer is referred as the access switches, each of which is connected to 15 hosts or servers of the datacenter.

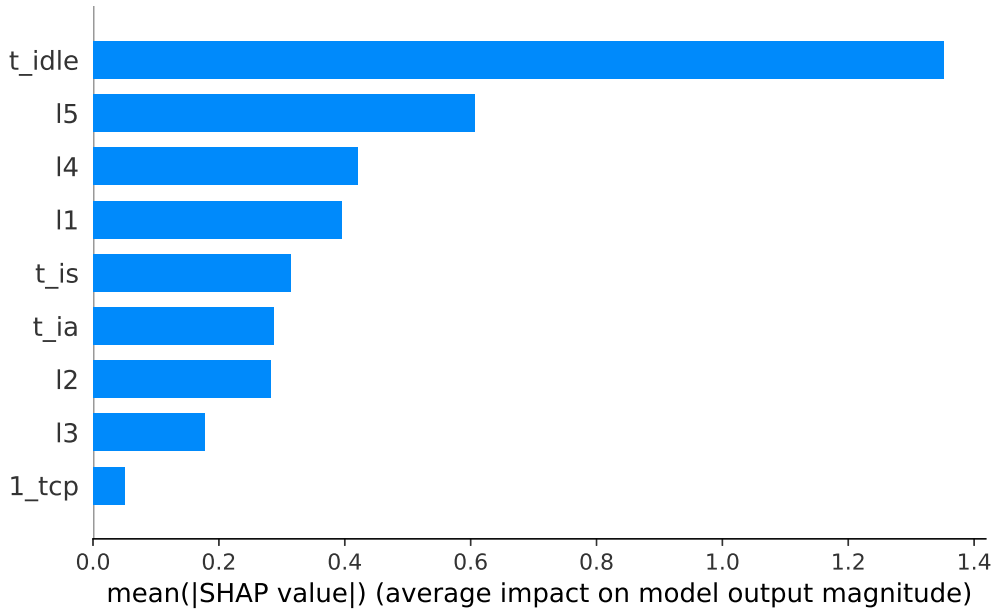


Figure 6.12: The importance of different features of the GBT model (`n_estimators`:30; `subsample`:0.8; `learning_rate`:0.1; `max_depth`:10) for UNIV1 trace with 1K flow table.

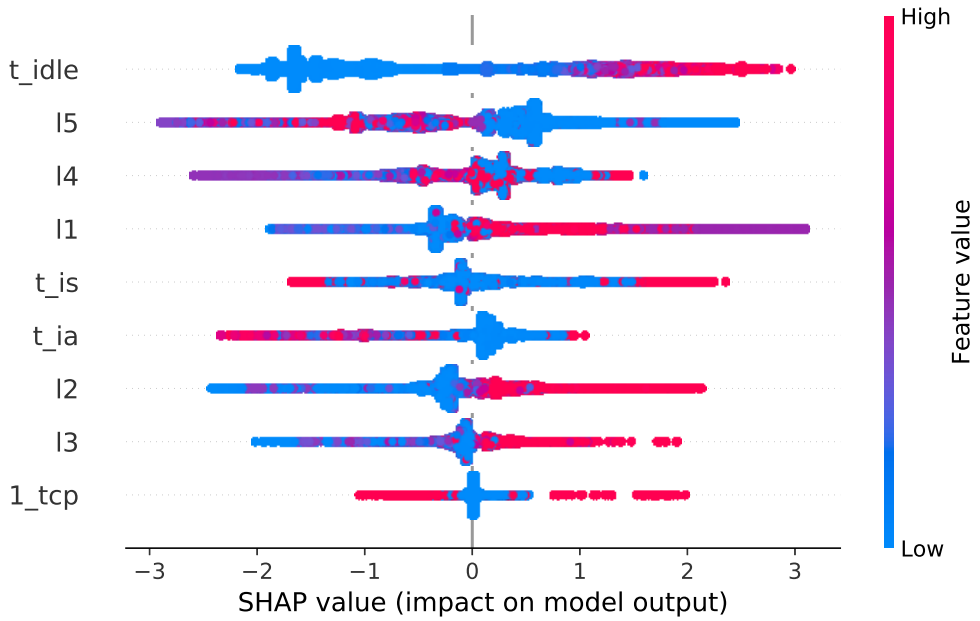


Figure 6.13: The SHAP values of every feature of the GBT model (`n_estimators`:30; `subsample`:0.8; `learning_rate`:0.1; `max_depth`:10) for every data sample (represented by one dot on each row), for UNIV1 trace with 1K flow table.

The access switches are also fully connected to the top layer of aggregate switches. In addition, all aggregate switches connect to a single cloud switch as a simplification. This cloud switch models the gateway for the datacenter to the broader internet. And a single "Internet" host/server is connected to the cloud switch. Every host connected to the access switches initiates an ON-OFF application destined to every other host in the datacenter subjected to the constraint that there is one and only one ON-OFF application between any two hosts. The attributes (i.e., the distribution of the duration of ON state and OFF state, data rate in ON state, and the number of packets to send) of the ON-OFF application[2] are derived from the UNIV1 packet trace. We used the approach in [6] to determine the ON and OFF periods for each flow, i.e., ON period is defined as the longest continual period during which all the packet inter-arrival times are smaller than $arrival_{95}$ and OFF period is a period between any two ON periods, where $arrival_{95}$ is the 95th percentile value in the inter-arrival time distribution. According to this approach, we try to find the distribution (among Weibull, lognormal, exponential, Pareto, Erlang, and gamma distributions) that best fits the arrival process, which are shown in Table 6.8.

To do these simulations, we extend the `ofswitch13` module of NS3 [96], [97]. On the controller side, all switches (including cloud switch, aggregate switches, and access switches) are controlled by an SDN controller, upon which a newly developed routing application runs. This application finds the shortest path between the given source and destination hosts in the simulated datacenter topology, and install the corresponding flow entries to the switches in the path. On the switch side, we modified the `ofsoftswitch13` library included in the `ofswitch13` to implement the LRU eviction policy and machine learning eviction policy for the flow table. Since we only care how the number of capacity misses will affect the network performance, we do not implement real machine learning model for flow entry eviction. Instead, we use a probability to control how accurate an inactive flow entry can be detected and evicted. This is doable because we know exactly when a flow

[2]Refer to `https://www.nsnam.org/docs/release/3.29/doxygen/classns3_1_1_on_off_application.html` for more details of the attributes for ON-OFF application

starts and ends in the simulation environment, which is not true in real networks.

According to our previous analysis, the reason why reducing capacity misses is significant for network performance is that massive `Packet-In` events due to capacity misses will cause large delay and even packet loss because the RX and TX buffers/queues in the controller is limited in size and its CPU may be burned because of the large computation overhead. In the simulation setup, it either requires to set small TX/RX buffers or make very large scale simulations. Our simulation experiments are carried out on a system with 8GB RAM and a 3.6GHz Intel(R) Xeon(R) E5-1620 processor running Ubuntu 16.04. The maximum number of hosts can be simulated is limited to a few thousand due to memory limitation, and one simulation lasts for several days. Even we make the simulated network scale as large as possible, which is still far smaller than the real datacenters, we do not see obvious delay enlargement and buffer overflow. Given this fact, we can only use the other approach, where we set the TX/RX buffer size to be 100 KB.

We first investigate the overhead of control channel with different capacity misses, which is shown in Figure 6.15. The x-axis is the average number of capacity misses caused by each flow entry eviction, which can be used to map the system-level simulation results to our previous case studies such that we can infer how better our machine learning policy can perform in terms of network performance metrics compared with LRU policy. As we can see, both the received and transmitted control messages on the control channels increase as the number of capacity misses increases. For example, when the number of capacity misses per eviction increased from 0.11 to 0.37, the received control messages jump by 113% and the transmitted control messages are boosted by 130%. This is reasonable because more capacity misses means more `Packet_In` and `Flow_Removed` events. Accordingly, the controller needs to send more `Packet_Out` and `Flow_Mod` messages to switches to install flow entries. Furthermore, we can see that the extra messages caused by capacity misses account for larger proportion of the total overhead on the control channels as the number of capacity misses per eviction increases. In the case of 0.11 capacity

misses/eviction, the messages due to wrong flow entry eviction only account for 26% of RX messages and 32% TX messages. However, these numbers become 64% for RX and 74% for TX when it comes to 0.37 capacity miss per eviction.

We then examine network performance, which are shown in Figure 6.16. As we can see, the throughput is decreased and the packet loss rate is increased as the number of capacity misses per eviction increases. This is because the increased overhead of control channels (as shown in Figure 6.15) results to TX and RX buffer overflows and thus many packets are dropped. As for the delay, the results are against our intuition. In general, packet delay should be enlarged as the channel becomes more congested. However, it drops as the number of capacity misses per eviction increases. The fact that congested channel results to large delay is due to the queueing delay in the buffer increases. In our simulation setup, the RX/TX buffers are set to be 100KB, which can store only around 200 packets. And the delay for processing one `Packet_In` is 50 us, thus the largest queueing delay for a packet is just 10 ms. In contrast, if connections are not built or closed because of packet loss, no packet will be sent and these packets will certainly not be included for computing packet delay. This has more weight on packet delay compared with the small queueing delay. Therefore, the packet delay instead decreases as the number of capacity misses increases.

In addition, we map the number of capacity misses per eviction of UNIV1 trace with 1K flow table in Section 6.3 to Figure 6.16. And we can see that the throughput achieved by machine learning eviction is augmented by 19%, from 37 MBps to 43.9 MBps. The packet loss rate achieved by STEREOS is reduced by 70%, but the packet delay is slightly increased by 8%. The system-level simulation results demonstrate that our machine learning eviction policy can greatly improve network performance.

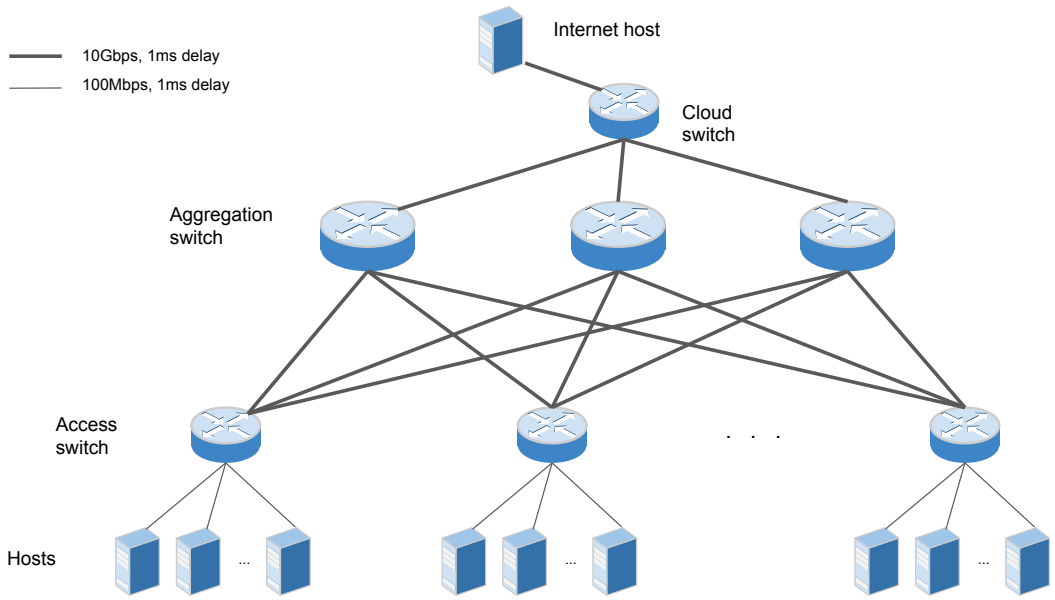
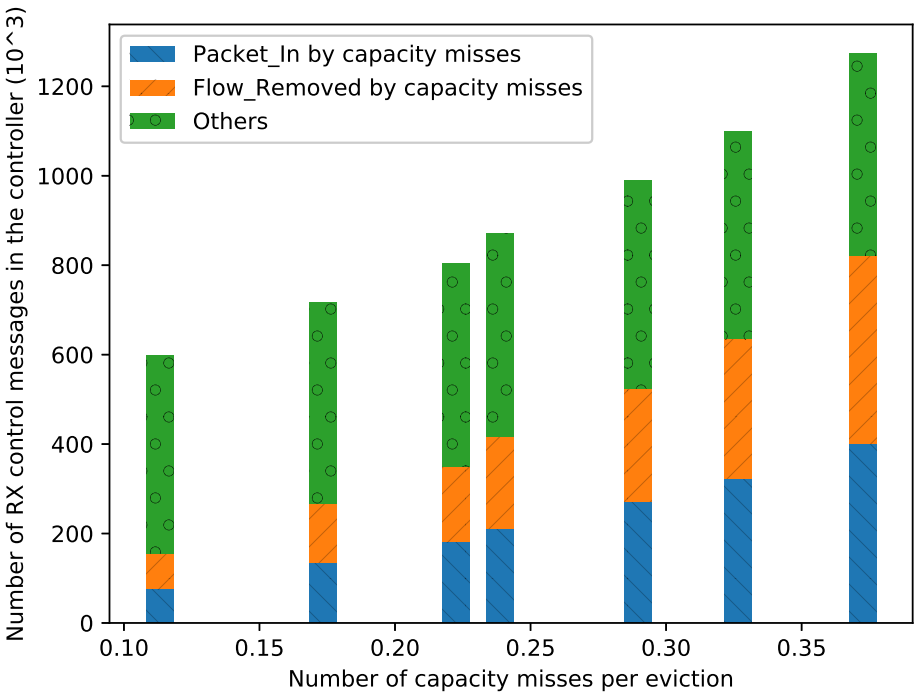


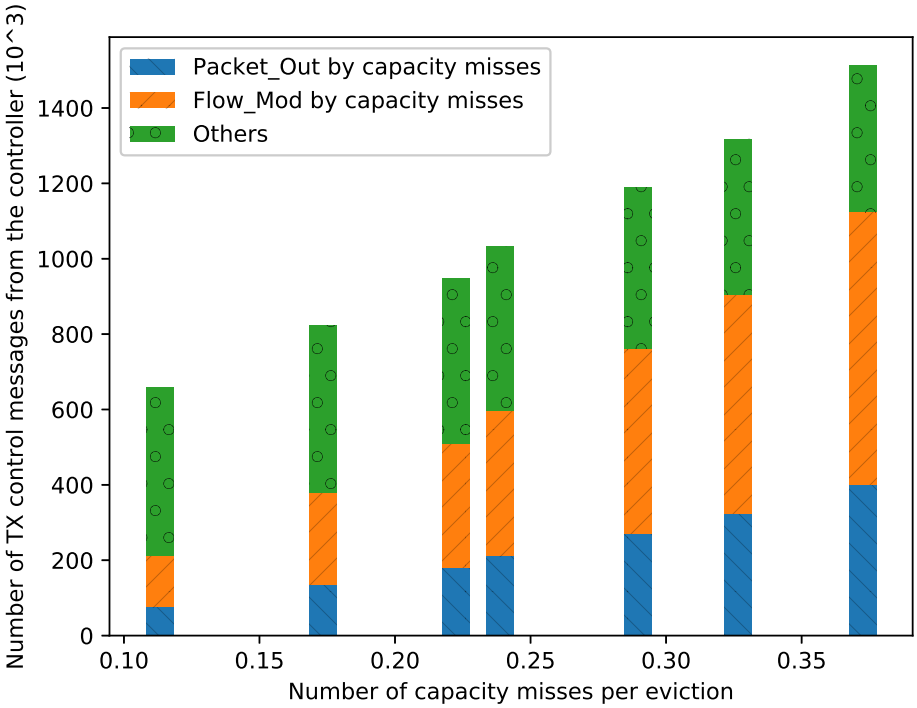
Figure 6.14: Simulated datacenter topology

Table 6.8: Simulation parameters

Parameter	Distribution/Value
Duration of 'On' state	lognorm($\mu = -2.32, \sigma = 2.13$)
Duration of 'Off' state	lognorm($\mu = 0.70, \sigma = 1.99$)+0.99
Flow inter-arrival time	lognorm($\mu = -6.33, \sigma = 2.02$)
Number of packets to send per flow	lognorm($\mu = 1.87, \sigma = 1.08$)+1.86
Size of a packet	512 bytes
Number of aggregate switches	3
Number of access switches	16
Control channel bandwidth	17Mbps
Data rate in ON state	0.842MBps
RX/TX queue size of controller	100KB
Flow table size	1024
Delay for processing one Packet-In	50 us

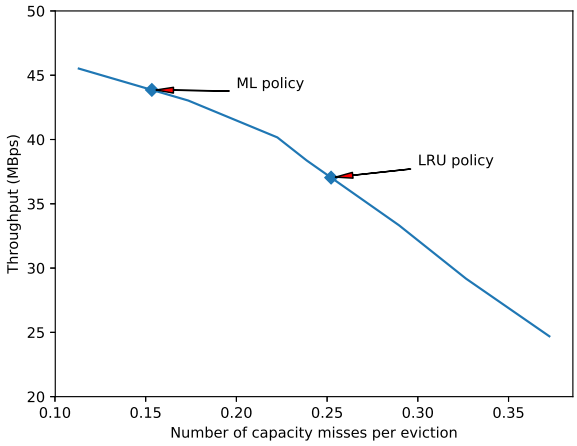


(a)

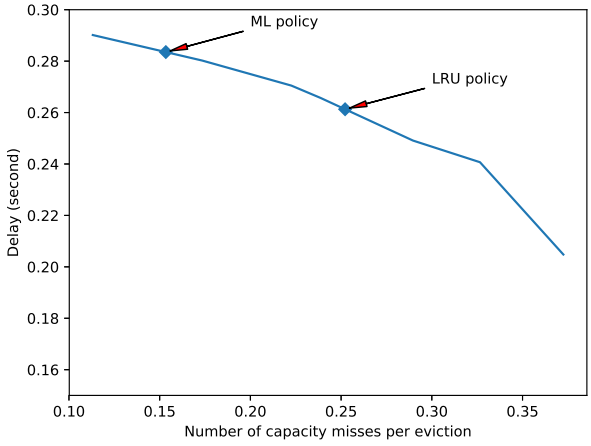


(b)

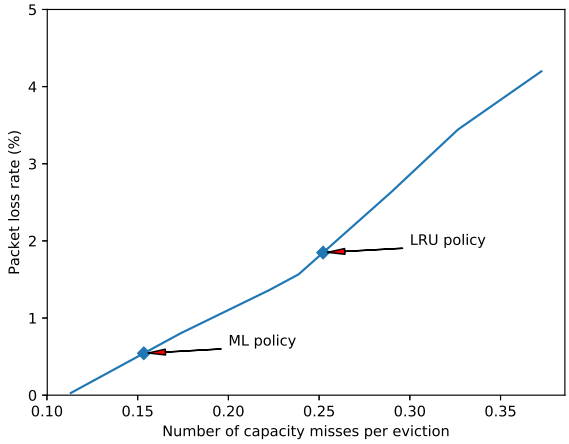
Figure 6.15: Overhead of controller-switch communication. (a) received control messages in the controller; (b) transmitted control messages from the controller.



(a)



(b)



(c)

Figure 6.16: System-level simulation results for STEREOS and LRU policy. (a) throughput; (b) delay; (c) packet loss rate.

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

7.1 Conclusion

This work has covered a variety of challenges which limit the scalability of SDN OpenFlow networks. On the control plane, all existing control plane architectures are examined to find the most scalable architecture for deploying a large scale SDN OpenFlow network. We argue that simulation is the best approach to study the scalability performance of different SDN OpenFlow control architectures. Simulators for all non-centralized control architectures and the centralized one are developed based on the abstractions of the two bottlenecks, flow setup and statistic collection, which limit the scalability performance of SDN OpenFlow networks. The simulation results show that the hierarchical control architecture is the best choice if the network is constrained to some limited geographic area. However, when the network is to scale across geographies, the peer-to-peer with local view (a.k.a. distributed) control architecture is the most appropriate. For the distributed control plane, a novel eastbound/westbound protocol, SDNi-TE, is designed to enable traffic engineering relevant information to be exchanged between neighboring domains such that all involved SDN domains can build consistent global views about the whole network. Based on this global view, any traffic engineering algorithm can run in a centralized way. The simulation results show that SDNi-TE can achieve almost the same throughput and cost as the one which impractically assumes every domain knows the full topology of the whole network. In summary, SDNi-TE is a protocol which makes the deployment of large scale distributed SDN OpenFlow networks possible. On the data plane, flow table management is optimized by applying machine learning techniques. Specifically, SPEEDO is proposed to approximate the LRU proactive deletion policy because the LRU policy by itself is not

feasible for the controller due to the fact the controller cannot tract the order of flow entry access in real time. To approximate the LRU policy, SPEEDO applies regression models to predict the time when a flow entry will be last referred to. Based on the predictions, the controller can proactively delete the flow entry with the smallest last refer time instead of a random one or the first installed one. Case studies demonstrate that SPEEDO can approximate the LRU policy very well, with $85\% \sim 96\%$ accuracy for different scenarios. Furthermore, SPEEDO can decrease up to 23% fewer capacity misses compared with the available random deletion and FIFO deletion policies. SPEEDO is designed to mitigate flow table overflow. However, flow table overflow is inevitable since the number of flows in the context of SDN is very large. In the presence of flow table overflow, we further propose STEREOS which can identify whether a flow entry is active or inactive based on machine learning algorithm and thus timely evict the inactive flow entries when flow table overflow occurs. STEREOS includes collecting datasets from packet traces, training a binary classification model based on the collected data, and applying the trained model for online flow entry eviction. We discussed various issues for implementing STEREOS in OpenFlow switches, including model selection, model size, overhead, and feature quantization. Our case studies show that our proposal can achieve much fewer capacity misses and higher flow table usage, compared with the LRU eviction policy. Last but not least, we developed the first system-level simulation tools based on *ns-3* to evaluate the performance of our proposal in terms of networking metrics. The simulation results demonstrate that STEREOS can remarkably improve network throughput and reduce packet loss rate, with a slight raise for packet delay.

7.2 Limitations of This Work and Recommendations for Future Research

First and foremost, this study is limited in scope. SDN OpenFlow scalability is a multi-dimensional topic, which involves every aspect, from hardware to software, of a network including hardware availability, manufacture cost, resource management, protocol design,

and so on. This work only considers four typical problems related to scalability: find the most scalable control architecture, design eastbound/westbound protocol to enable the communications between multiple controllers, reduce control overhead, and optimize flow table management. There are definitely other important issues to be addressed to build scalable SDN OpenFlow networks. For example, how to set `idle_timeout` for each flow entry to increase flow table utilization. In addition, it is challenging and important to make OpenFlow switches more powerful without significant cost growth.

Second, the proposals in this work offer a starting point, not a set of final solutions. The approach simulating bottleneck processes in SDN OpenFlow networks to study the scalability performance of different control architectures is the right direction. But more details can be added to the current implementations. For example, not only flow setup and statistic collection, but also how peering controllers in different domains communicate with each other can be considered in future simulations. In addition, in current simulation implementation, only flow entry expiration is simulated but not flow entry eviction and proactive flow entry deletion. A more detailed simulator which incorporates these two flow table management policies will be definitely beneficial for the credibility of the conclusions derived from the simulation results. As for SDNi-TE, the advantages of SDN are utilized to build aggregated paths, but more complicated policies instead of simply avoiding routing loop can be investigated for advertisement in the future. Another future work which is worth investigating is to advertise some semi-dynamic information which can stay the same in a relatively long period. For example, flow table utilization can be advertised such that OpenFlow switches with overloaded flow tables can be removed from routing decisions. Furthermore, we do not consider the problem of building and maintaining the connections between neighboring domains in this dissertation. The last suggested future work for SDNi-TE is to define the formats of the messages, just like the `Update` message in BGP. For SPEEDO, we can see the performance gain is not very significant now. Therefore, more efforts should be made to increase the regression accuracy such as trying more

cultivated features by exploiting feature engineering. The features used in this work for SPEEDO are the mean and standard deviation of the collected stats from switches. This is straightforward but may not be very effective, one improvement is to use a machine learning model (e.g., neural network) to take the raw stats and generate synthetic features. Another possible work is to drop the current regression-based approach. In this work, a regression model is trained to predict the last refer time for every flow entry and thus we can know which entry is the LRU one. This is actually an indirect method, because we only want to know the access order of flow entries but do not care when a flow entry will be exactly last referred. Therefore, the learning goal can directly be finding the LRU flow entry. However, in this case, one data sample should contain all stats from each flow entry, which will result in the curse of dimensionality. For STEREOS, only the case where some inactive flow entries exist in the flow table is considered. In the presence of no inactive flow entry, which one should be evicted? In addition, for both SPEEDO and STEREOS, only static cases are considered. What if the underlying traffic changes? Although the current proposals are demonstrated to be effective within the timescale of several hours, online models which can adapt to traffic pattern variation is necessary in the long term.

Finally, more-depth analysis of the proposed solutions are required. For SDNi-TE, how fast it can converge is required to be investigated. Furthermore, more tests should be conducted on SDNi-TE for the cases where a link is failed, a switch is down, and so on. For flow table management, both STEREOS and SPEEDO should be further implemented and tested on "real" switches such as OpenvSwitch by emulations.

7.3 Publications

As part of the research conducted in this thesis, several papers are either published or in progress as follows:

- H. Yang and G. F. Riley, "Traffic engineering in the peer-to-peer sdn," in *Computing, Networking and Communications (ICNC), 2017 International Conference on*, IEEE,

2017, pp. 649–655

- H. Yang, C. Zhang, and G. F. Riley, "Support multiple auxiliary tcp/udp connections in sdn simulations based on ns-3," in *Proceedings of the Workshop on ns-3*, ACM, 2017, pp. 24–30

- H. Yang, J. Ivey, and G. F. Riley, "Scalability comparison of sdn control plane architectures based on simulations," in *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, IEEE, 2017, pp. 1–8

- H. Yang and G. F. Riley, "Machine learning based proactive flow entry deletion for openflow," in *2018 IEEE International Conference on Communications (ICC)*, IEEE, 2018, pp. 1–6

- H. Yang and G. F. Riley, "Machine learning based flow entry eviction for openflow switches," in *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, IEEE, 2018, pp. 1–8

- H. Yang, D. M. Blough, and G. F. Riley, "STEREOS: Smart Table EntRy Eviction for Openflow Switches," in progress to submit to IEEE Journal on Selected Areas in Communications, 2019

Appendices

APPENDIX A

EXPECTATION OF BUFFERED PACKETS IN THE DATAPATH BUFFER

Let $\mathbf{X} = (X_1, X_2, ...)$ be the sequence of flow inter-arrival times in the Poisson process, and the sequence of arrival times is $\mathbf{T} = (T_0, T_1, T_2, ...)$, where $T_0 = 0$, and

$$T_n = \sum_{i=1}^{n} X_i. \tag{A.1}$$

Then within the interval τ, the number of packets saved to the buffer is

$$N_\tau = \sum_{i=0}^{N}((\tau - T_i)\gamma + 1), \tag{A.2}$$

where N is the number of flows arrived within τ and it is a random variable. Thus, we have

$$\begin{aligned} E[N_\tau | N = n] &= E[\sum_{i=0}^{n}((\tau - T_i)\gamma + 1)] \\ &= n(\tau\gamma + 1) - \gamma E[\sum_{i=0}^{n} T_i]. \end{aligned} \tag{A.3}$$

According to the equation A.1, we can get

$$\begin{aligned} E[\sum_{i=0}^{n} T_i] &= E[\sum_{i=1}^{n}\sum_{j=1}^{i} X_j] \\ = \sum_{i=1}^{n}\sum_{j=1}^{i} E[X_j] &= \frac{1}{2\lambda}(n+1)n \end{aligned} \tag{A.4}$$

Therefore,

$$\begin{aligned} E[N_\tau] &= E[E[N_\tau|N=n]] \\ &= E[n(\tau\gamma+1) - \frac{\gamma}{2\lambda}(n+1)n] \end{aligned} \tag{A.5}$$

Since N is the number of arrival flows within τ, according the properties of Poisson process, we have

$$E[N] = \lambda\tau \tag{A.6}$$

$$VAR[N] = \lambda\tau^2 \tag{A.7}$$

Based on the above two equations, we finally get

$$\begin{aligned} E[N_\tau] &= (\tau\gamma+1)\lambda\tau - \frac{\gamma}{2\lambda}(\lambda^2\tau^2 + \lambda\tau^2 + \lambda\tau) \\ &= \frac{1}{2}\lambda\gamma\tau^2 - \frac{1}{2}\gamma\tau^2 + \lambda\tau - \frac{1}{2}\gamma\tau \end{aligned} \tag{A.8}$$

REFERENCES

[1] A. Singh *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 183–197, Aug. 2015.

[2] N. Communications. (2017). NTT Communications Launches World's Largest SD-WAN Footprint Covering Over 190 Countries with Industry's Most Comprehensive End-to-End SD-WAN Service Portfolio, [Online]. Available: `https://www.ntt.com/en/about-us/press-releases/news/article/2017/0620.html`.

[3] *Openflow switch specification (version 1.5.1)*, Mar. 2015.

[4] D. Erickson, "The beacon openflow controller," in *Proceedings of the 2nd ACM SIGCOMM workshop on Hot topics in SDN*, 2013, pp. 13–18.

[5] M. Kuźniar *et al.*, "What you need to know about sdn flow tables," in *International Conference on Passive and Active Network Measurement*, 2015, pp. 347–359.

[6] T. Benson *et al.*, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, 2010, pp. 267–280.

[7] A. R. Curtis *et al.*, "Devoflow: Scaling flow management for high-performance networks," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 254–265, 2011.

[8] K. He *et al.*, "Measuring control plane latency in sdn-enabled switches," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ACM, 2015, p. 25.

[9] G. Lu *et al.*, "Serverswitch: A programmable and high performance platform for data center networks.," in *Nsdi*, vol. 11, 2011, pp. 2–2.

[10] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[11] F. Bannour, S. Souihi, and A. Mellouk, "Distributed sdn control: Survey, taxonomy, and challenges," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 333–354, 2018.

[12] J Camhi, "Former cisco ceo john chambers predicts 500 billion connected devices by 2025," *Business Insider*, 2015.

[13] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[14] A. Mendiola *et al.*, "A survey on the contributions of software-defined networking to traffic engineering," *IEEE Communications Surveys & Tutorials*, 2017.

[15] J. Hu *et al.*, "Scalability of control planes for software defined networks: Modeling and evaluation," in *2014 IEEE 22nd International Symposium of Quality of Service (IWQoS)*, 2014, pp. 147–152.

[16] M. Karakus and A. Durresi, "A scalability metric for control planes in software defined networks (sdns)," in *Advanced Information Networking and Applications (AINA), 2016 IEEE 30th International Conference on*, IEEE, 2016, pp. 282–289.

[17] P. Song *et al.*, "Paraflow: Fine-grained parallel sdn controller for large-scale networks," *Journal of Network and Computer Applications*, vol. 87, pp. 46–59, 2017.

[18] E. G. Renart *et al.*, "Towards a gpu sdn controller," in *Networked Systems (NetSys), 2015 International Conference and Workshops on*, IEEE, 2015, pp. 1–5.

[19] K. Phemius *et al.*, "Disco: Distributed multi-domain sdn controllers," in *Proc. IEEE Network Operations and Management Symposium (NOMS'14)*, May 2014, pp. 1–4.

[20] A. Tootoonchian and G. Yashar, "Hyperflow: A distributed control plane for openflow," in *Proc. USENIX internet network management conference on Research on enterprise networking (INM/WREN'10)*, Apr. 2010, pp. 3–3.

[21] S. H. Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proc. ACM Workshop on Hot Topics in Software Defined Networks (HotSDN'12)*, Helsinki, Finland, 2012, pp. 19–24.

[22] Y. Fu *et al.*, "Orion: A hybrid hierarchical control plane of software-defined networking for large-scale networks," in *Proc. IEEE International Conference on Network Protocols (ICNP'14)*, 2014, pp. 569–576.

[23] K. Qiu *et al.*, "Paracon: A parallel control plane for scaling up path computation in sdn," 2017.

[24] A. Hari *et al.*, "Path switching: Reduced-state flow handling in sdn using path information," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, ACM, 2015, p. 36.

[25] H. Owens and A. Durresi, "Explicit routing in software-defined networking (ersdn): Addressing controller scalability," in *2014 17th International Conference on Network-Based Information Systems*, 2014, pp. 128–134.

[26] K. Qiu *et al.*, "Gflow: Towards gpu-based high-performance table matching in openflow switches," in *Information Networking (ICOIN), 2015 International Conference on*, IEEE, 2015, pp. 283–288.

[27] R. Bifulco and A. Matsiuk, "Towards scalable sdn switches: Enabling faster flow table entries installation," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 343–344, 2015.

[28] G. Bianchi *et al.*, "Openstate: Programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 44–51, 2014.

[29] H. Yin, H. Xie, T. Tsou, D. Lopez, P. Aranda, and R. Sidi, "Sdni: A message exchange protocol for software defined networks (sdns) across multiple domains," *IETF draft, work in progress*, 2012.

[30] P. Lin, J. Bi, S. Wolff, Y. Wang, A. Xu, Z. Chen, H. Hu, and Y. Lin, "A west-east bridge based sdn inter-domain testbed," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 190–197, 2015.

[31] F. Benamrane, R. Benaini, *et al.*, "An east-west interface for distributed sdn control plane: Implementation and evaluation," *Computers & Electrical Engineering*, vol. 57, pp. 162–175, 2017.

[32] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined wan," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013.

[33] C. Y. Hong *et al.*, "Achieving high utilization with software-driven WAN," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 15–26, 2013.

[34] M. Luo *et al.*, "An adaptive multi-path computation framework for centrally controlled networks," *Computer Networks*, vol. 83, pp. 30–44, Jun. 2015.

[35] S. Agarwal *et al.*, "Traffic engineering in software defined networks," in *Proc. IEEE International Conference on Computer Communications (INFOCOM'13)*, Turin, Apr. 2013, pp. 2211–2219.

[36] Y. Guo *et al.*, "Traffic engineering in sdn/ospf hybrid network," in *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*, IEEE, 2014, pp. 563–568.

[37] M. Caria *et al.*, "Divide and conquer: Partitioning ospf networks with sdn," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, IEEE, 2015, pp. 467–474.

[38] W. Wang *et al.*, "Enhancing the effectiveness of traffic engineering in hybrid sdn," in *Communications (ICC), 2017 IEEE International Conference on*, 2017, pp. 1–6.

[39] J. He and W. Song, "Achieving near-optimal traffic engineering in hybrid software defined networks," in *Proc. IEEE IFIP Networking Conference (IFIP Networking'11)*, Toulouse, May 2015, pp. 1–9i.

[40] Y. Guo *et al.*, "Traffic engineering in hybrid sdn networks with multiple traffic matrices," *Computer Networks*, vol. 126, pp. 187–199, 2017.

[41] X. Li *et al.*, *Inter-domain sdn traffic engineering*, US Patent 9,559,980, 2017. [Online]. Available: `https://www.google.com/patents/US9559980`.

[42] A. Vishnoi *et al.*, "Effective switch memory management in openflow networks," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, 2014, pp. 177–188.

[43] H. Liang *et al.*, "Effective idle_timeout value for instant messaging in software defined networks," in *Communication Workshop (ICCW), 2015 IEEE International Conference on*, IEEE, 2015, pp. 352–356.

[44] A. Zarek, "Openflow timeouts demystified," Master's thesis, University of Toronto, 2012.

[45] H. Zhu *et al.*, "Intelligent timeout master: Dynamic timeout for sdn-based data centers," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, IEEE, 2015, pp. 734–737.

[46] C.-H. He, B. Y. Chang, S. Chakraborty, C. Chen, and L. C. Wang, "A zero flow entry expiration timeout p4 switch," in *Proceedings of the Symposium on SDN Research*, ACM, 2018, p. 19.

[47] Q. Li, N. Huang, D. Wang, X. Li, Y. Jiang, and Z. Song, "Hqtimer: A hybrid q-learning based timeout mechanism in software-defined networks," *IEEE Transactions on Network and Service Management*, 2019.

[48] R. Challa *et al.*, "Intelligent eviction strategy for efficient flow table management in openflow switches," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pp. 312–318.

[49] T. Pan *et al.*, "Alfe: A replacement policy to cache elephant flows in the presence of mice flooding," in *2012 IEEE International Conference on Communications (ICC)*, pp. 2961–2965.

[50] B.-S. Lee *et al.*, "An efficient flow cache algorithm with improved fairness in software-defined data center networks," in *2013 IEEE 2nd International Conference on Cloud Networking (CloudNet)*, pp. 18–24.

[51] K. Kannan and S. Banerjee, "Flowmaster: Early eviction of dead flow on sdn switches," in *International Conference on Distributed Computing and Networking*, Springer, 2014, pp. 484–498.

[52] H. Yang, J. Ivey, and G. F. Riley, "Scalability comparison of sdn control plane architectures based on simulations," in *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, IEEE, 2017, pp. 1–8.

[53] D. Erickson, "Using network knowledge to improve workload performance in virtualized data centers," PhD thesis, Citeseer, 2013.

[54] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," in *ACM SIGCOMM Computer Communication Review*, ACM, vol. 37, 2007, pp. 1–12.

[55] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. R. Kompella, "Elasticon; an elastic distributed sdn controller," in *2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, IEEE, 2014, pp. 17–27.

[56] P. Berde *et al.*, "ONOS: Towards an open, distributed SDN OS," in *Proc. ACM workshop on Hot topics in software defined networking (HotSDN'14)*, Aug. 2014, pp. 1–6.

[57] J. Ivey, H. Yang, C. Zhang, and G. Riley, "Comparing a scalable sdn simulation framework built on ns-3 and dce with existing sdn simulators and emulators," in *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ACM, 2016, pp. 153–164.

[58] J. Sommers, R. Bowden, B. Eriksson, P. Barford, M. Roughan, and N. Duffield, "Efficient network-wide flow record generation," in *INFOCOM, 2011 Proceedings IEEE*, IEEE, 2011, pp. 2363–2371.

[59] A. R. Curtis *et al.*, "Devoflow: Scaling flow management for high-performance networks," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 254–265, 2011.

[60] S. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," vol. 52, no. 7, pp. 116–123, 2014.

[61] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, ACM, 2009, pp. 202–208.

[62] H. Yang and G. F. Riley, "Traffic engineering in the peer-to-peer sdn," in *Computing, Networking and Communications (ICNC), 2017 International Conference on*, IEEE, 2017, pp. 649–655.

[63] B. Quoitin, C. Pelsser, L. Swinnen, O. Bonaventure, and S. Uhlig, "Interdomain traffic engineering with BGP," vol. 41, no. 5, pp. 122–128, 2003.

[64] R. Meier, P. Tsankov, V. Lenders, L. Vanbever, and M. Vechev, "Nethide: Secure and practical network topology obfuscation," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 693–709.

[65] D. Walton, A. Retana, E. Chen, and J. Scudder, "Advertisement of multiple paths in bgp," Tech. Rep., 2016.

[66] G. Huston. (2019). AS65000 BGP routing table analysis report, [Online]. Available: `http://bgp.potaroo.net/as2.0/bgp-active.html`.

[67] Y. Rekhter, T. Li, and S. Hares, "A border gateway protocol 4 (bgp-4)," Tech. Rep., 2005.

[68] J. Y. Yen, "Finding the k shortest loopless paths in a network," *Management Science*, vol. 17, no. 11, pp. 712–716, 1971.

[69] M. Gupta, C. C. Aggarwal, and J. Han, "Finding top-k shortest path distance changes in an evolutionary network," in *International Symposium on Spatial and Temporal Databases*, Springer, 2011, pp. 130–148.

[70] E. Danna *et al.*, "A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering," in *Proc. IEEE International Conference on Computer Communications (INFOCOM'12)*, Mar. 2012, pp. 846–854.

[71] M. Caesar and J. Rexford, "Bgp routing policies in isp networks," *IEEE network*, vol. 19, no. 6, pp. 5–11, 2005.

[72] F. Baccelli and D. Hong, "Flow level simulation of large IP networks," in *Proc. IEEE International Conference on Computer Communications (INFOCOM'03)*, Mar. 2003, pp. 1911–1921.

[73] A. Medina *et al.* (2001). BRITE: Universal topology generation from a user's perspective, [Online]. Available: `http://www.cs.bu.edu/brite/user_manual/BritePaper.html`.

[74] Z. Guo *et al.*, "Star: Preventing flow-table overflow in software-defined networks," *Computer Networks*, 2017.

[75] H. Yang and G. F. Riley, "Machine learning based proactive flow entry deletion for openflow," in *2018 IEEE International Conference on Communications (ICC)*, IEEE, 2018, pp. 1–6.

[76] J. Leng, Y. Zhou, J. Zhang, and C. Hu, "An inference attack model for flow table capacity and usage: Exploiting the vulnerability of flow table overflow in software-defined network," *arXiv preprint arXiv:1504.03095*, 2015.

[77] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[78] G. Hackeling, *Mastering Machine Learning with scikit-learn*. Packt Publishing Ltd, 2017.

[79] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[80] H. Yang and G. F. Riley, "Machine learning based flow entry eviction for openflow switches," in *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, IEEE, 2018, pp. 1–8.

[81] H. Yang, D. M. Blough, and G. F. Riley, "STEREOS: Smart Table EntRy Eviction for Openflow Switches," in progress to submit to IEEE Journal on Selected Areas in Communications, 2019.

[82] M. Kuźniar, P. Perešíni, D. Kostić, and M. Canini, "Methodology, measurement and analysis of flow table update characteristics in hardware openflow switches," *Computer Networks*, vol. 136, pp. 22–36, 2018.

[83] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, "Rules placement problem in openflow networks: A survey," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1273–1286, 2016.

[84] A. A. Pranata, T. S. Jun, and D. S. Kim, "Overhead reduction scheme for sdn-based data center networks," *Computer Standards & Interfaces*, vol. 63, pp. 1–15, 2019.

[85] M. Kobayashi, S. Seetharaman, G. Parulkar, G. Appenzeller, J. Little, J. Van Reijendam, P. Weissmann, and N. Mckeown, "Maturing of openflow and software-defined networking through deployments," *Computer Networks*, vol. 61, pp. 151–175, 2014.

[86] E. L. Fernandes, E. Rojas, J. Alvarez-Horcajo, Z. L. Kis, D. Sanvito, N. Bonelli, C. Cascone, and C. E. Rothenberg, "The road to bofuss: The basic openflow user-space software switch," *arXiv preprint arXiv:1901.06699*, 2019.

[87] E. Alpaydin, *Introduction to machine learning*. MIT press, 2014.

[88] C. Bergmeir and J. M. Benítez, "On the use of cross-validation for time series predictor evaluation," *Information Sciences*, vol. 191, pp. 192–213, 2012.

[89] M. Dusi, F. Gringoli, and L. Salgarelli, "Quantifying the accuracy of the ground truth associated with internet traffic traces," *Computer Networks*, vol. 55, no. 5, pp. 1158–1167, 2011.

[90] S. B. Kotsiantis, I Zaharakis, and P Pintelas, "Supervised machine learning: A review of classification techniques," *Emerging artificial intelligence applications in computer engineering*, vol. 160, pp. 3–24, 2007.

[91] R. Caruana *et al.*, "An empirical evaluation of supervised learning in high dimensions," in *Proceedings of the 25th international conference on Machine learning*, ACM, 2008, pp. 96–103.

[92] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms," in *Proceedings of the 23rd international conference on Machine learning*, ACM, 2006, pp. 161–168.

[93] A. Grami, *Introduction to Digital Communications*. Academic Press, 2015.

[94] M. Du, N. Liu, and X. Hu, "Techniques for interpretable machine learning," *arXiv preprint arXiv:1808.00033*, 2018.

[95] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems*, 2017, pp. 4765–4774.

[96] L. J. Chaves, I. C. Garcia, and E. R. M. Madeira, "Ofswitch13: Enhancing ns-3 with openflow 1.3 support," in *Proceedings of the Workshop on ns-3*, ACM, 2016, pp. 33–40.

[97] Computer Networks Laboratory at Unicamp, Brazil. (2018). Openflow 1.3 module documentation (release 3.3.0), [Online]. Available: `http://www.lrc.ic.unicamp.br/ofswitch13/ofswitch13.pdf`.

[98] H. Yang, C. Zhang, and G. F. Riley, "Support multiple auxiliary tcp/udp connections in sdn simulations based on ns-3," in *Proceedings of the Workshop on ns-3*, ACM, 2017, pp. 24–30.