

Dissertation

zur Erlangung des akademischen Grades
des Doktors der Naturwissenschaften

(Dr. rer. nat)



Performance Evaluation of the Control Plane in OpenFlow Networks

eingereicht

am Institut für Informatik

des Fachbereichs Mathematik und Informatik

der Freien Universität Berlin

von

Zhihao Shang

Berlin, 2019

Gutachter:

Prof. Dr. Katinka Wolter

Department of Computer Science

Freie Universität Berlin, Germany

Prof. Dr. Mesut Güneş

Institut für Intelligente Kooperierende Systeme

Otto von Guericke University Magdeburg, Germany

Date of defense: December 29, 2019

Selbständigkeitserklärung

Ich versichere, dass ich die Doktorarbeit selbständig verfasst, und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit hat keiner anderen Prüfungsbehörde vorgelegen.

Zhihao Shang

Berlin, den 10. Februar 2020

Notations

$f(x)$	pdf function
$F(x)$	cdf function
λ	arrival rate
$M(X^k)$	k -th moment of random variable X
$E[X]$	mean of random variable X
$Var(X)$	variance of random variable X
α	initial probability of a PH distribution
\mathbf{T}	generator matrix of a PH distribution
$\mathbf{1}$	a column vector of ones
π	stationary vector of a CTMC
ϕ_k	lag- k correlation
μ	service rate
ρ	server utilization
\mathbf{I}	unitary matrix
B_i	first phase of the i th Erlang distribution
E_i	last phase of the i th Erlang distribution
λ_p	arrival rate of packet-in messages
λ_s	arrival rate of synchronization messages
\mathbf{A}	controller assignment matrix
M	number of switches migrations
λ_b	mean batch size
$L(S_i)$	the load of server i
$L(C_i)$	CPU utilization of server i
$L(M_i)$	memory utilization of server i
r_i	rank of server i

Abstract

Online services and applications have grown rapidly in the last decade. The network is necessary for many services and applications. Many technologies are invented to meet the requirements of online services, such as micro-services and serverless computing. However, the traditional network architecture suffers from several shortages. It is difficult for the traditional network to adapt to new demands without massive reconfiguration. In traditional IP networks, it is complex to manage and configure the network devices since skilled technicians are required. Changing the policy of a network is also time consuming because network operators need to re-configure multiple network devices and update access control lists using low level commands. The management and configuration becomes more complex and challenging, when the traffic in a network changes frequently.

SDN (Software-defined networking) is an innovative approach to manage networks more flexible. It separates the control plane from forwarding devices and uses a centralized controller to manipulate all the forwarding devices. The separation offers many benefits in terms of network flexibility and management. The controller can provide a global view of a network. Using the controller, network operators can manage and configure all the network devices at a high level interface. With SDN, a network can adapt to new demands by updating the applications in the controller.

However, all these benefits come with a performance penalty. Since the controller manipulates all the forwarding devices, the performance of the controller impacts the performance of the whole network. In this thesis, we investigate the performance of SDN controllers. We also implement a benchmark tool for OpenFlow controllers. It measures the response time of an OpenFlow controller and fit a phase-type distribution to the response time. Based on the distribution of the response time, we build a queueing model for multiple controllers in an OpenFlow network and determine the optimal number of controllers that can minimize the response time of the controllers. We design an algorithm that can optimize the mapping relationship among the switches and controllers. The load of controllers can be balanced with the optimized mapping relationship.

Zusammenfassung

Online-Dienste und -Anwendungen sind in den letzten zehn Jahren rasant gewachsen. Das Netzwerk ist für viele Dienste und Anwendungen erforderlich. Viele Technologien wurden erfunden, um die Anforderungen von Online-Diensten wie Microservices, serverlosen Computing zu erfüllen. Die herkömmliche Netzwerkarchitektur leidet jedoch unter mehreren Engpässen. Es ist für das traditionelle Netzwerk schwierig, sich ohne massive Konfigurationen an neue Anforderungen anzupassen. In herkömmlichen IP-Netzwerken ist die Verwaltung und Konfiguration der Netzwerkgeräte komplex, da erfahrene Techniker benötigt werden. Das Ändern der Switching-Regeln ist ebenfalls zeitaufwändig, da Netzwerkbetreiber mehrere Netzwerkgeräte neu konfigurieren und Zugriffssteu- rungslisten in Befehlen auf niedriger Ebene aktualisieren müssen. Die Verwaltung und Konfigura- tion kann komplexer und anspruchsvoller werden, wenn sich der Datenverkehr in einem Netzwerk häufig ändert.

SDN (Software-defined Networking) ist ein innovativer Ansatz zur Verwaltung von Netzwer- ken. Es trennt die Steuerebene von der Netzwerkschicht und verwendet eine zentrale Steuerung, um alle Switches zu manipulieren. Die Trennung bietet viele Vorteile in Bezug auf Netzwerkflexibili- tät und -verwaltung. Allerdings kommen all diese Vorteile mit einer Leistungseinbuße einhergekom- men. Da der Controller alle Weiterleitungsgeräte manipuliert, wirkt sich die Leistung des Control- lers auf die Leistung des gesamten Netzwerks aus. In dieser Arbeit untersuchen wir die Leistung von SDN-Controllern. Wir implementieren auch ein Benchmark-Tool für OpenFlow-Controller, um an die Reaktionszeit eines OpenFlow-Controllers eine Phasenverteilung anzupassen. Basierend auf der Verteilung der Reaktionszeit erstellen wir ein Warteschlangenmodell für mehrere Controller im OpenFlow-Netzwerk, und ermitteln die optimale Anzahl von Controllern, die die Reaktionszeit der Controller minimieren können. Wir entwerfen einen Algorithmus, der die Zuordnungsbeziehung zwischen den Switches und Controllern optimieren kann. Die Last der Controller kann mit der opti- mierten Zuordnungsbeziehung balanciert werden.

Acknowledgements

I would like to thank my supervisor Prof. Dr. Katinka Wolter. In my four years at Berlin, she helped me a lot on my work. She provided lots of valuable advice and guidance in my research. She taught me writing papers and presentation. She carefully revised my papers sentence by sentence. I deeply admire her positive work attitude and professional skills. She is more than a perfect supervisor during my life at Berlin. Before I came to Berlin, I never lived abroad. I thought it might be difficult to get along with foreigners because of the different culture, but she made everything easy. She helped me to start a new life at Berlin and led me on my research.

I am very grateful for the love and support of my parents. I was a very disobedient child. I dropped out of school in the 8th grade, but they did not give up on my education. They sent me back to school so that I could continue my study. Thanks for the patience and tolerance.

I would like to thank my wife Dr. Yanhua Chen. We met each other at college, and our love has lasted for more than a decade. We got our B.S. degrees together, got our M.S degrees together, and came to Berlin together. She always accompanies and supports me.

Thanks all my colleagues. They are very nice and friendly. I enjoyed working with them very much. I wish to thank Guang Peng, Han Wu, Dr. Tianhui Meng and Dr. Yi Sun. They shared their ideas and discussed the problems I faced. They helped me when I am stuck. Thanks for their help and company.

My financial support is from China Scholarship Council. I am very thankful to China Scholarship Council providing me such an opportunity to study at Free University of Berlin.

Contents

Notations	i
Abstract	iii
Zusammenfassung	v
Acknowledgement	vii
I Introduction	1
1 Basic Concepts and Problems	3
1.1 Overview of SDN	3
1.2 Performance of the Control Plane	5
1.3 Performance Improvement with Multiple Controllers	7
1.4 Performance Improvement with Buffer Management	7
1.5 Contributions	9
1.6 Thesis Outline	10
2 Background	13
2.1 Overview of OpenFlow Networks	13
2.1.1 OpenFlow Switches	13
2.1.2 OpenFlow Controllers	16
2.2 Mathematical Background	20
2.2.1 Exponential Distribution	20

2.2.2	PH Distributions	21
2.2.3	Markovian Arrival Process	24
2.2.4	M/PH/1 Queue	26
3	Related Work	31
3.1	Performance Modeling	31
3.2	Benchmarking Tools	33
3.3	Performance Improvement	34
3.4	Performance of Multiple Controllers	37
II	Tools for Performance Modeling	41
4	HyperStar2: Easy Distribution Fitting of Correlated Data	43
4.1	Fitting Algorithm	44
4.1.1	Constructing the D_0 matrix	45
4.1.2	Constructing the D_1 matrix	46
4.1.3	Example	48
4.2	Implementation	49
4.3	Evaluation of HyperStar2	51
4.4	Summary	55
5	An OpenFlow Controller Performance Evaluation Tool	57
5.1	The OFCP Tool	59
5.1.1	Design Goal	59
5.1.2	Architecture	60
5.1.3	Implementation	61
5.2	Performance Evaluation Result	61
5.3	Summary	64
III	Performance of the Control Plane	67
6	The Performance of Multiple controllers	69
6.1	The Number of Controllers	70
6.1.1	System Description	70

6.1.2	Problem Formulation	71
6.1.3	Derivation of the Analytical Model	74
6.1.4	Evaluation	75
6.2	Balancing the load of Controllers	82
6.2.1	Problem Formulation	82
6.2.2	Controller Assignment Algorithm	86
6.2.3	Evaluation	89
6.3	Summary	94
7	Buffer management	97
7.1	The Limitations of Existing Buffer Management	97
7.2	The Proposed MPT Model	98
7.3	Queueing Model of the OpenFlow Controller	101
7.3.1	Controller Performance of General Buffer	101
7.3.2	Controller Performance of the MPT	104
7.4	Evaluation	106
7.4.1	Queueing Analysis	106
7.4.2	Simulation	109
7.5	Summary	112
8	Server Load Balance in OpenFlow Networks	113
8.1	Load balance for online service	113
8.2	Server Load Balancing Strategy	114
8.2.1	Load Balancing Strategy	115
8.2.2	OpenFlow Based Load Balancer Implementation	116
8.3	Experiment Results	118
8.3.1	Experiment Environment	118
8.3.2	Experiment Results	119
8.4	Summary	122
9	Conclusions and Outlook	123
9.1	Conclusions	123
9.2	Outlook	125
	Bibliography	127

List of Figures	141
List of Tables	145
About the Author	147

Part I

Introduction

Chapter 1

Basic Concepts and Problems

In this chapter, we introduce the components in SDN and compare SDN with traditional IP network. Also we describe the performance problem in SDN and the ways to improve the performance of SDN. We summarize the main contribution of this thesis.

1.1 Overview of SDN

With the development of computer networks, more and more applications and services are running on the internet, such as video conferencing, online storage and instant messaging. The management and configuration of different applications and services have become highly complex and challenging, especially when the traffic in a network changes frequently. Many researchers invested a lot of effort to come up with a general management paradigm that can simplify and improve network management with high flexibility [26, 32, 102]. However, the traditional IP network is an obstacle to the flexible network management. The network devices are controlled in a distributed way in traditional IP networks. Network protocols run in the routers and switches. Each device makes forwarding decisions independently. The distributed control makes the management of traditional IP networks difficult and complex [14]. The network operators have to configure each router and switch individually to apply a high level network policy. Usually, the commands for routers and switches are in low level and vendor specific. Besides the complex configuration, there are no automatic response mechanisms in the traditional IP networks. Network measurement and analysis show that traffic in a network may change very frequently [15, 63, 152]. It is difficult to adapt to the load changes for a network without automatic reconfiguration. Also dynamic faults are a challenge for traditional IP networks.

In traditional IP networks, the bundled control plane and data plane are integrated inside network devices. The control plane in a network device is usually the firmware, and it is not easy to change the functions implemented in the firmware. The immutable functions decrease flexibility and hinder innovation and evolution of networks. A new protocol may take very long time to be fully deployed. For example, IPv6 started more than a decade ago and it is still not completed. Adding new functions to a network without changing any hardware is not feasible in practice [44, 104].

As a new computer network architecture, SDN is considered a promising way towards the future internet [109]. SDN decouples the control plane from forwarding devices and allows one separate controller to manipulate all the switches in a network. The controller in SDN is usually a piece of software that makes the network very flexible and innovative. SDN can also hide the details of the physical infrastructure and enable efficient network management [71, 144, 145]. The architecture of SDN is shown in Figure 1.1.

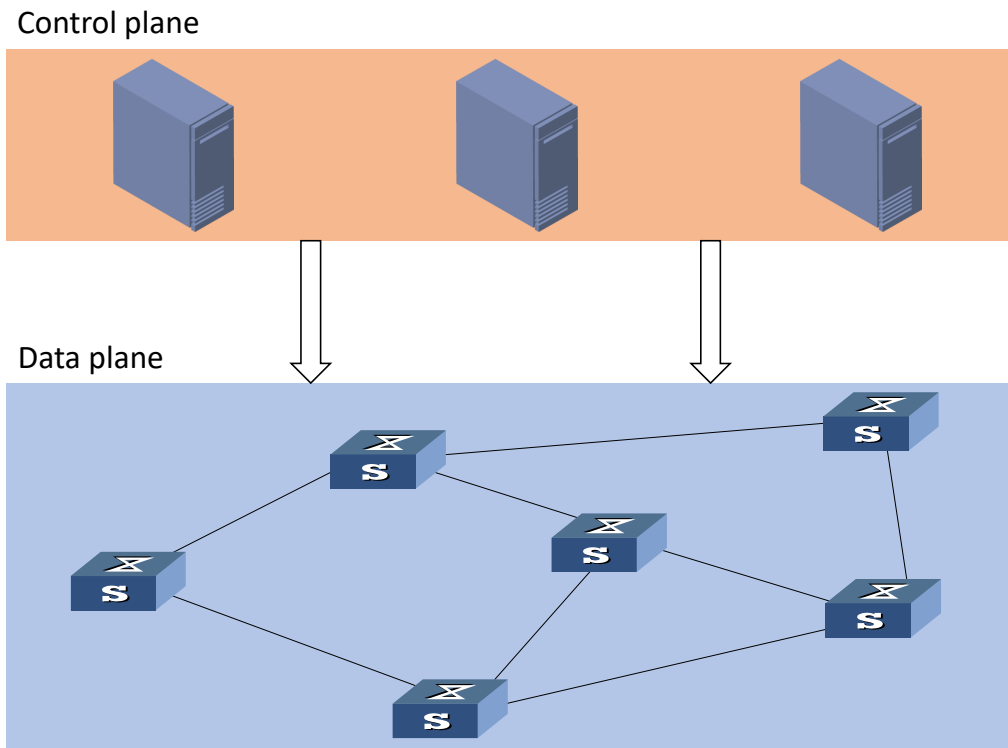


Figure 1.1: The architecture of SDN

The control plane in SDN is usually a programmable controller. The data plane in SDN is a set of switches that are not implemented with intelligence. The SDN switches cannot make routing

decisions. They can only forward packets following the instructions from the controller. A SDN switch sends requests to its controller to get instructions when it cannot find forwarding rules. The controller manages all the switches in a network. It maintains a global view of a network. Based on the global view, the controller manages the network resources more reasonably and may make better routing decisions than the traditional IP networks that are managed in a distributed way.

As its core advantage, SDN offers a high flexibility in the control plane. Network operators can change the routing of some traffic flows without influencing other flows. Researchers can test their new protocols in the existing hardwares [85]. The programmable controller also simplifies the management of networks. Network operators do not have to configure every device manually. The applications in the controller can gracefully make changes of the network topology or traffic route. SDN makes automatic network management feasible [93, 136].

1.2 Performance of the Control Plane

The flexibility of SDN comes at a performance penalty. First, a software controller is usually slower than a logical hardware unit embedded in traditional switches. As we can see from Figure 1.2, in a traditional switch, the control plane is part of the switch, it is a piece of hardware, which has high performance and forwards packets at linear speed. Second, the communication between the controller and switches introduces a new delay in the network. There are no logical units in SDN switches, so the SDN switches have to get instructions from the controller. The communication between the controller and switches degrades the performance of a network. The difference between traditional switches and SDN switches is shown in Figure 1.2.

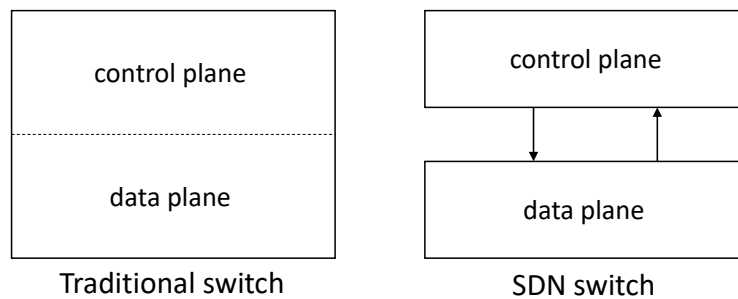


Figure 1.2: The difference between traditional switches and SDN switches

OpenFlow is a popular implementation of SDN. It was first proposed by Nick McKeown to enable research experiments [85]. In OpenFlow networks, switches can only forward packets following the

instructions given by a centralized controller. There are one or more flow tables in an OpenFlow switch, and the flow entries are stored in the flow tables. OpenFlow switches forward packets following the instructions in the flow entries that match the packets. When a packet arrives at an OpenFlow switch and the switch cannot find any flow entries matching the packet, the switch will send the header of the packet to its controller through a packet-in message to obtain new flow entries. The controller will install flow entries into the switch through flow-modify messages. The switch can find forwarding rules for the packets with the same headers after flow entries are installed. Therefore, only the first packet in a flow needs to be sent to the controller.

OpenFlow continues to receive a lot of attentions from researchers, but most work focuses on its availability, scalability and functionality [3,16,34,154]. The performance of OpenFlow has not been investigated much to date. This may become an obstacle for wide deployment. It is a prerequisite to understand the performance and limitations of OpenFlow for its usage in production environment. Most researchers improve the performance of the control plane of OpenFlow in the following two aspects:

Multiple controllers

A single controller is sufficient in most medium size networks [52], but multiple controllers can improve the performance, scalability and availability of the networks [35]. Therefore, multiple controllers are widely used in practice. With multiple controllers, load balancing can reduce the communication latency. New controllers can be deployed or removed dynamically to adapt to variable traffic in a network. Moreover, multiple controllers can also avoid a single point of failure as well as improve the security of the control plane [97,115].

Reduce the communication

The communication between the controller and switches is a new delay in OpenFlow networks. Reducing the packet-in messages from switches to the controller can improve the performance of OpenFlow networks significantly [61,64]. The packet-in messages consume computing resources in switches and controllers, and bandwidth between control plane and data plane. Too high consumption of computing resources and bandwidth may cause a delay in the process of some requests from switches, some requests may even be dropped. The performance of a network may be degraded in such scenario.

1.3 Performance Improvement with Multiple Controllers

The forwarding devices in the network have to send messages to the controller when there are no flow entries for the arriving packets. The communication between the controller and the switches increases the transmission latency. With emerging big data, the overhead will be worse than ever, since the massive traffic in the data plane triggers a lot of requests. It is a significant challenge for the capability of the SDN controllers. To reduce the transmission latency, multiple controllers are deployed into one network, the controllers manage network flows cooperatively [50, 134]. Nevertheless, the multiple controllers introduce a new problem: how many controllers should we use to minimize the flow setup time? Enough controllers must be deployed into a network, so that the controllers are able to handle network traffic. The multiple controllers require communication with each other to maintain a view of the network [76]. The communication overhead can be significant if there are a large number of controllers in a network. Therefore, more controllers may not yield better performance. We must determine the optimal number of controllers to achieve the best performance.

Another problem in the multiple controllers scenario is how to balance the load among all the controllers. The mapping relationship between switches and controllers is statically assigned in some systems [67]. The static assignment may lead to high flow setup time because the traffic in a network fluctuates frequently. A network usually experiences heavy traffic in daytime and light traffic at night [111]. The flow arrival rate may change in short time scales even when the total traffic does not change [63]. The dynamic traffic in a network may cause unbalanced load among controllers. One controller may receive more requests than its capacity while another controller are at low utilization. An overloaded controller leads to high flow setup time for the switches under its management, and the high flow setup time may cause congestion and degrade the performance of the whole network [79]. When multiple controllers are deployed, the assignment between switches and controllers should be considered to achieve good performance.

1.4 Performance Improvement with Buffer Management

The OpenFlow specification [41] defines a buffer in the OpenFlow switch to reduce the traffic in OpenFlow channels. If an OpenFlow switch cannot find any forwarding rules for a packet, it sends a packet-in message to its controller to request flow entries. The packet-in message contains the packet that triggers the packet-in message. OpenFlow switches will send a part of a packet with a buffer identification instead of the whole packet to a controller when a packet-in message is

sent. Reducing the size of a packet-in message is the default way to reduce the traffic in OpenFlow channels. However, an OpenFlow switch buffers packets in packet-granularity [84]. Under this model, the number of packet-in messages cannot be reduced. An OpenFlow controller may receive many packet-in messages belonging to the same flow in a very short time if a host sends many packets and all the packets are in one flow. This may happen very often when a user sends a heavy UDP stream. For connection-oriented TCP traffic, the flow entries are installed during the three-way handshake. TCP traffic does not generate bursts of packet-in messages often. However if there is large dynamic traffic in a network, such as a data center network, the flow entries in the switches may be updated very frequently. Flow entries for a TCP connection may be deleted before the connection completes. This will result in a lot of TCP packets which can not match any flow entries, and lead to many identical packet-in messages [15]. If bursts of packets belonging to one flow arrive at an OpenFlow switch, the OpenFlow controller will continuously receive identical packet-in messages until a flow entry is installed in the switch. This action will consume computing resources of the controller and increase the delay of the network.

OpenFlow switches send all the mismatched packets to controllers. Under this model, OpenFlow switches may send many unnecessary packet-in messages during the flow setup process [101]. The unnecessary packet-in messages will happen if the first packet in a flow is sent to the controller, and many packets in the same flow arrive before the switch receives a response from the controller. These unnecessary messages will increase the workload in both switches and controllers. As a result, the network performance will be degraded, and some OpenFlow messages may be delayed or even dropped. Many switches have a mechanism to limit the overall rate of packet-in messages, which only allows switches to send requests to the controller below a certain rate. This mechanism cause the problem that packet-in messages of non-heavy flows are also dropped at high rate when packets of both heavy flows and non-heavy flows are sent to the controller. An example of non-heavy flows is DNS requests. There are only a few packets in a DNS request. This problem would cause a significant delay in the setup of flow entries for non-heavy flows.

To further reduce the transmission workload in OpenFlow channel, we design a flow-granularity mismatched packet buffer model to only send the first packet of a flow to the controller. It is a simple and efficient solution. We can store the mismatched packets at flow-granularity. Only the first packet of a flow is sent to the controller. The controller can get enough information for installing flow entries when the first packet-in message is received. The OpenFlow switches can stop sending packet-in messages belonging to the same flow to the controller, and wait for forwarding instructions. The OpenFlow switch only sends another packet-in message if a timeout expires. All the other mismatched packets in the same flow can be cached in the buffer of the OpenFlow switch and

will be forwarded after the flow entries are installed.

1.5 Contributions

The main contributions of this thesis are to analyze and improve the performance of OpenFlow networks with multiple controllers. We develop two tools to analyze the performance of OpenFlow networks. Also we propose some schemes to improve the performance of OpenFlow networks. The efficiency of the schemes is confirmed by either experiments or simulations.

We give a summary of these contributions in the following.

The performance of controllers is a major attribute that must be considered when designing an OpenFlow network. A model-based analysis method can provide valuable insights into the network performance. Researchers have contributed models for the performance of OpenFlow networks. We make a simple survey about the performance models to show their similarity and difference. Besides the modeling, there are also researchers developed benchmark tools for OpenFlow controllers. All the benchmark tools use virtual OpenFlow switches to send generated packets to a controller and measure the response time of the controller, but they focus on different aspects. We give a comparison of the tools and show their key features.

Multiple controllers are necessary in a large OpenFlow network. The mapping relationship among the controllers and switches may impact the performance of the network. In addition, mapping relationship needs to be adjusted when the traffic in the network changes. We also make a survey about the assignment of multiple controllers.

Performance modeling can analyze the behavior of system. However, many benchmark tools only provide users the minimum, maximum throughput as well as the mean and variance of response time. Users cannot get the distribution of the response time of a system. The mean and variance of the response time are the most commonly used metrics in application performance management. However, in reality, the response time often has a long tail, the mean and variance cannot capture the tail of the distribution of the response time. The distribution of response time is necessary to many performance models. We develop a user-friendly tool to fit data trace into a PH (phase-type) distribution or MAP (Markovian Arrival Process) [116]. The tool helps people to build queueing models to evaluate the performance of controllers with the fitted result.

We also develop a benchmark tool for OpenFlow controllers [119]. Unlike other benchmark tools that focus on throughput or mean response time, our tool helps users to build models for OpenFlow networks and evaluate the performance of the controllers with the models. Our tool aims to provide a simple way to analyze the performance of OpenFlow controllers. It measures the response time of

an OpenFlow controller and provides the distribution of response time.

To reduce the flow setup time, multiple controllers are deployed into one network. Nevertheless, the multiple controllers introduce new problems. We have to determine the number of controllers that can minimize the flow setup time. The communication overhead may degrade the performance of a network if there are many controllers in a network. We should also consider the management and communication overhead because too many controllers in one network may increase the flow setup time. We propose a queueing model to evaluate the response time of a controller [117]. Based on a queueing model, we determine the number of controllers that can minimize the flow setup time in an OpenFlow network [121]. We build a multiple controller prototype [29] and measure the response time of controllers. We fit a PH distribution to the response time and determine the optimal number of controllers based on the distribution.

The mapping relationship among the controllers and switches may also impact the performance of an OpenFlow network. With dynamic traffic in a network, a controller may receive more requests than its capacity. Meanwhile, the other controllers are at low utilization. An overloaded controller leads to high flow setup time and degrades the performance of the whole network. We propose a heuristic to solve this problem [118]. We design a greedy algorithm to generate a feasible assignment as the input of the heuristic.

When bursts of packet-in messages happen in an OpenFlow network, the controller will receive lots of requests in a very short time. This degrades the performance of the network. We propose a buffer management method for OpenFlow switches to avoid bursts of packet-in messages [120]. The method can improve the network performance by reducing the workload of a controller.

OpenFlow controllers make networks very flexible. Many controller applications can offer different features to a network. We implement a load balancer to improve the service running on top of OpenFlow networks. The load balancer is a controller application. It modifies the packets between clients and server to distribute requests to different servers. We also design a load balancing strategy. Experimental results show that the proposed strategy can balance the load of servers and decrease the response time [28].

1.6 Thesis Outline

In the first part of the thesis, we introduce the background of this thesis and some related work. Part I has the following structure:

In Chapter 1, we describe the performance issues in OpenFlow networks and introduce the technical background of the thesis. The research field of this thesis is the performance of the OpenFlow

control plane. We introduce the basic concept of OpenFlow networks and the performance issues of OpenFlow controllers. This thesis focuses on performance improvements of OpenFlow controllers. The multiple controllers and buffering mechanisms in OpenFlow switches are presented and the contributions of this thesis are summarized.

In Chapter 2, the background of this thesis is introduced. We introduce the technical details of OpenFlow networks and the mathematical knowledge used in this thesis.

In Chapter 3, a survey on the performance of controller is presented. We introduce the related work in the different areas. First, we compare the existing performance models of OpenFlow networks and benchmark tools for OpenFlow controllers. Then we present some methods of performance improvement and the assignment of multiple controllers.

We develop performance tools for OpenFlow controllers and introduce them in the second part. Part II is structured as follows:

In Chapter 4, we present HyperStar2, a tool for fitting PH distributions or MAPs (Markovian Arrival Processes) to empirical data. The tool targets engineers and scientists who find themselves in need of distribution fitting for non-standard distributions but have little interest in the underlying algorithms and parameter settings.

In Chapter 5, we present a user-friendly OpenFlow controller performance evaluation tool that aims at helping network researchers to build performance models of OpenFlow controllers, and network managers to understand the performance behavior of OpenFlow controllers. The tool uses a virtual OpenFlow switch sending OpenFlow messages to a controller and measures the response time. It fits the response time to a hyper-Erlang distribution.

The third part of this thesis introduces the schemes that improve the performance of OpenFlow controllers. Part III is structured as follows:

In Chapter 6, we build a queueing model to evaluate the flow setup time of multiple controllers, and use an optimization algorithm to determine the optimal number of controllers that can minimize the flow setup time. To reduce the flow setup time and improve the performance of OpenFlow networks, we propose a controller assignment scheme for multiple controllers. It adjusts the assignment among controllers and switches dynamically based on the load of switches and the capacity of controllers. We model each controller as an $M/PH/1$ queue and use a heuristic to optimize the controller assignment based on the queueing model.

In Chapter 7, we focus on bursts of packet-in messages in OpenFlow networks. We build a queueing model for bursts of packet-in messages and present a method that decreases the number of packet-in messages using buffers in OpenFlow switches.

In Chapter 8, we implement a server load balancing in virtual environment. The controller collects

CHAPTER 1. BASIC CONCEPTS AND PROBLEMS

the status of servers and distributes requests based on the load of servers.

In Chapter 9, the main part of this thesis is concluded and the outline of the future research direction is pointed.

Chapter 2

Background

We introduce the technical background of this thesis in this chapter. An OpenFlow network consist OpenFlow switches and controller. They communicate through OpenFlow channel. We introduce the OpenFlow switches and OpenFlow controllers in Section 2.1. We fit a PH distribution to the response time of an OpenFlow controller and use the PH distribution in a queueing models to evaluate the performance of OpenFlow networks. The mathematical knowledge on PH distributions and queueing models are introduced in Section 2.2.

2.1 Overview of OpenFlow Networks

An OpenFlow network contains OpenFlow switches and controllers. We introduce OpenFlow switches and controllers separately.

2.1.1 OpenFlow Switches

An OpenFlow switch contains one or more flow tables and one or more OpenFlow channels, as shown in Figure 2.1. The flow tables perform packet lookups and forwarding, and the OpenFlow channels are used to connect to the controllers.

A flow table contains a set of flow entries. Flow entries are the forwarding rules. The controller manipulates the behaviors of switches by adding, updating and deleting flow entries in the flow tables. The components of a flow entry are shown in Table 2.1.

When a packet arrives at an OpenFlow switch, its header is extracted and used as lookup key. If the switch finds a flow entry that matches the packet, it will apply the instructions in the flow entry to the packet, the packet may be dropped, modified or forwarded. Otherwise, the packet will be sent

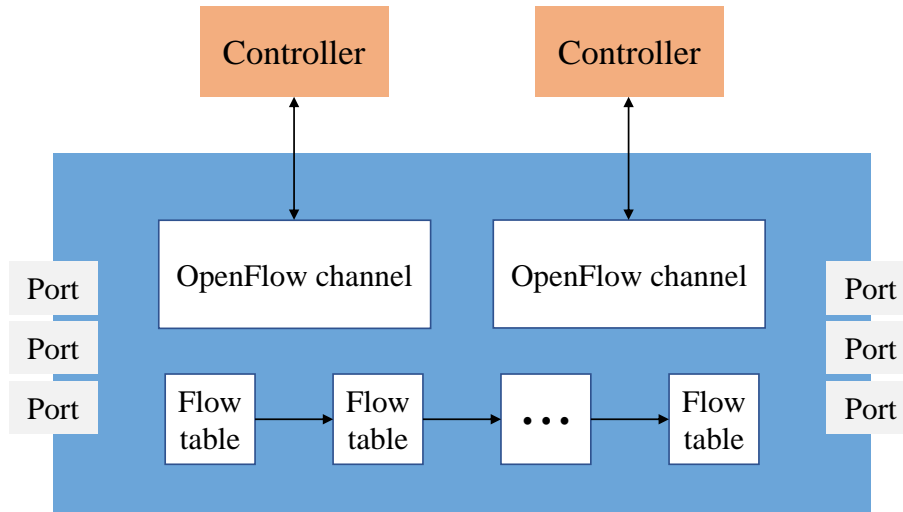


Figure 2.1: The main components of an OpenFlow switch

Table 2.1: The components of a flow entry

Component	Description
Match fields	To match against packets. Consisting of ingress port and packet headers.
Priority	Matching precedence of the flow entry.
Counters	Record how many packets are matched.
Instructions	Actions to apply to matched packets.
Timeouts	Maximum time before the flow expires.
Cookie	Used by the controller to manage flow entries. Not used in packets processing.
Flags	Describe the way flow entries are managed.

to the controller. The packets processing of an OpenFlow switch is shown in Figure 2.2.

A flow entry matches a packet if all the match fields of the flow entry match the header fields from the packet. The match fields include various protocol header fields, such as ethernet destination address, IPv4 destination address and TCP source port. In addition to the protocol headers, the ingress port can be also used for matching. An omitted field matches all possible values in the header field. Only the flow entry with the highest priority is selected if there are multiple flow entries match a packet. The counters in the selected flow entry are updated and the instructions in the selected flow entry are executed. No flow entries are selected if there are multiple flow entries with the same highest priority match a packet. This can only happen if a controller adds overlapping flow entries and does not make the switch check overlaps.

When a packet arrives at an OpenFlow switch and the switch cannot find a flow entry that matches

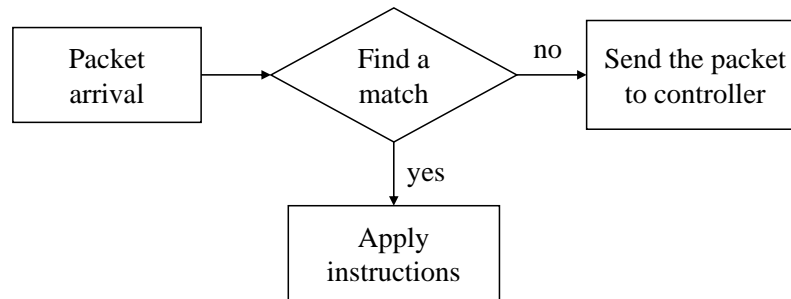


Figure 2.2: Packet processing in an OpenFlow switch

the packet, the packet will be dropped by default. However, a switch configuration can change this default behavior. OpenFlow switches are configured to send unmatched packets to the controller. Another way to handle the unmatched packets is adding a table-miss flow entry into an OpenFlow switch. A table-miss flow entry omits all the match fields and has the lowest priority. These two features make the table-miss flow entry match all the packets that are unmatched by other flow entries. A table-miss flow entry is very similar to other flow entries. It can be added or removed by the controller at any time. The instructions in the table-miss entry will be applied to the packets that are unmatched by other flow entries.

When a switch sends a packet to the controller, it sends a packet-in message that contains the packet to be sent. The main components of a packet-in message are shown in Table 2.2.

Table 2.2: The main components of a packet-in message

Component	Description
Buffer ID	ID assigned by switch.
Length	Total length of the packet.
Reason	The reason a packet is sent to controller.
Data	The packet to be sent.

The buffer ID is a value used by a switch to identify a buffered packet. An OpenFlow switch can implement buffering to reduce the traffic in the OpenFlow channel. If a packet is sent to the controller, the OpenFlow switch does not have to send the whole packet. The switch can store the packet in its buffer and send the header and a buffer ID to the controller. After a connection is established the controller and switches negotiate how many bytes can be carried in a packet-in message. When a packet-in message is sent to the controller, the packet-in message carries only some fraction of the packet header (128 bytes by default) and a buffer ID If the switch has enough

memory to buffer all packets that are sent to the controller,. Switches that do not have sufficient memory or do not support internal buffering must send the full packet to the controller as part of the packet-in message. As for large packets, the packet-in message can easily exceed the maximum transmission unit. The switch has to split the packet-in message into two packets, which is an extra processing step for both the switch and the OpenFlow channel.

The length is the total length of the packet that is sent to the controller. The actual length of the data field is less than the length if the packet is buffered. The "reason" field indicates why the packet is sent to the controller. The reason can be no matching flows (table-miss flow entry), output to controller in instructions or packet has invalid TTL. The data field is the packet to be sent to the controller. If the packet is buffered, the data field only contains the header of the packet. When a controller receives a packet-in message, it may install a flow entry into the switch through a flow-modify message so that the switch can find a match for the packets with the same header.

2.1.2 OpenFlow Controllers

The control plane manages all the underlying forwarding devices through the OpenFlow protocol and maintains a global view of the network. Various applications can run in the control plane to manage the network. If the demands change, one can develop and deploy relevant applications in the control plane. Many applications are implemented by researchers, such as firewalls [103, 156], traffic engineering [1, 2] and server load balancing [103, 142, 158]. The controller can also run traditional network management services and protocols, such as network configuration protocol (NETCONF) [36], Open Shortest Path First (OSPF) [88], simple network management protocol (SNMP) [25]. Due to the advantages of programmable control plane, OpenFlow has been deployed in many networks [39, 56, 94, 110, 132, 133]. Because of the global view of the network, controllers can make better forwarding decisions than traditional network devices. The utilization of links among Google's data centers is 30-40%. Google use OpenFlow to increase the utilization to near 100% [56].

There are over 300 different controllers such as Flowvisor [122], Oflops [108], Beacon [37], NOX [47], POX [65]. They are implemented in different programming languages. In this chapter we only compare the features of four popular controllers, Ryu [95], Trema [73], FloodLight [89] and OpenDaylight [86]. The other controllers are either deprecated or poorly documented.

The four controllers are open source software. Ryu is developed fully in Python language and supported by NTT. There are many pre-defined components and applications in Ryu, such as BGP (border gateway protocol), sFlow [98]. Trema is supported by NEC labs. It is developed in C language and Ruby language. C language is used to achieve the performance, and Ruby language is

used to provides easy API so that it provides effective productivity and high performance at the same time. FloodLight is developed in Java language. It runs on any platform that supports Java. There are a set of modules in FloodLight that offers different features, such as firewall, load balancer. All the modules are well documented. OpenDaylight is an open source project under linux foundation. It is implemented in Java language and use OSGi (Open Service Gateway Initiative) to manage the modules. The modules can be installed or uninstalled in the OSGi console when the controller is running. The comparison of the controllers is shown in Figure 2.3.

Table 2.3: compare controllers

	Ryu	Trema	Floodlight	Opendaylight
GUI	Yes	No	Yes	Yes
REST API	Yes	No	Yes	Yes
Programming language	Python	C/Trema	Java	Java
Supported platform	Linux	Linux	Linux, Mac OS, Windows	Linux, Mac OS, Windows
Modularity	Medium	Medium	High	High
Developer	NTT	NEC	Bigswitch	Linux foundation

An OpenFlow controller can add, update or delete flow entries into an OpenFlow switch using a flow-modify message. The main components of a flow-modify message are shown in Table 2.4

Table 2.4: The main components of a flow-modify message

Component	Description
Cookie	Used by the controller to manage flow entries.
Command	Type of flow-modify message.
Priority	Matching precedence of the flow entry.
Match fields	To match against packets.
Instructions	Instructions to be executed.
Timeout	Maximum time before the flow expires.

The command field can be *add*, *modify* or *delete*. For add requests, the switch must check for overlapping flow entries in the flow table. If there is an overlap between an existing flow entry and the flow-modify message, the switch refuses to add flow entries in the flow table. For modify requests, if there are flow entries in the flow table have the same match fields with the flow-modify message, the instructions field of the entries are updated with the value from the flow-modify message. For delete requests, any flow entries in the flow table have the same match fields with the flow-modify message are deleted.

An OpenFlow controller can also use packet-out message to send a packet to the network. A

packet-out message contains a packet and a port number. When an OpenFlow switch receives a packet-out message from a controller, it sends the packet in the message to the port indicated in the message.

OpenFlow uses a centralized and programmable controller to manage switches. Network managers can develop applications to improve the performance of the network and manage network resources in an efficient way. Because the controller is responsible for the whole network, it undertakes plenty of computing jobs and may become a bottleneck of the network. Distributed architecture is an effective way to avoid performance issues [46, 53, 55]. In the distributed architecture, there are multiple controllers in the control plane. A large scale network may be split into several domains, and each domain needs one controller to manage it [113]. The controllers exchange local information with each other to maintain a global view. In general, there are two architectures of distributed control plane, horizontal architecture and hierarchical architecture.

The horizontal architecture

OpenFlow starts with a single controller, such as NOX [47]. The single controller manages the entire network. It may become a single point of failure and the performance of the system is limited. To address these problems, some distributed controllers are proposed. Some researchers employ horizontal architecture to build distributed control plane [67, 122, 134]. The model of horizontal architecture is shown in Figure 2.3.

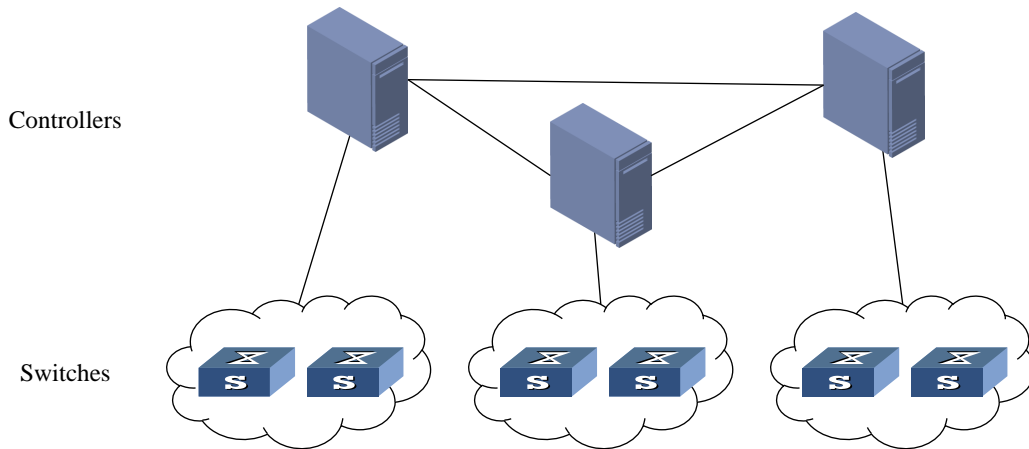


Figure 2.3: The model of horizontal architecture

In the horizontal architecture, the controllers are deployed into different domains. The controllers communicate with each other via east-west interfaces [12, 74]. Although each controller connects

to only one domain, all the controllers are equal and can manage the entire network. They share and maintain a global view together. The controllers make forwarding decision based on the global view. All the controllers must synchronize their knowledge with others. When the topology of one domain changes, all controllers will update the topology information synchronously. This communication overhead among the controllers is the main disadvantage of the horizontal architecture. It is important to reduce the traffic of state synchronization, while keeping consistent information among the controllers.

The hierarchical architecture

A main benefit of OpenFlow is the centralized controller. Based on the global view, the centralized controller can control traffic more efficiently than a traditional network. However, maintaining the global view may lead to an overloaded network if the state of the network changes frequently. Inconsistent network information may degrade the performance of the network [76]. To overcome the shortcomings of the horizontal architecture, a hierarchical architecture has been designed [50, 68, 75]. The hierarchical architecture model is shown in Figure 2.4.

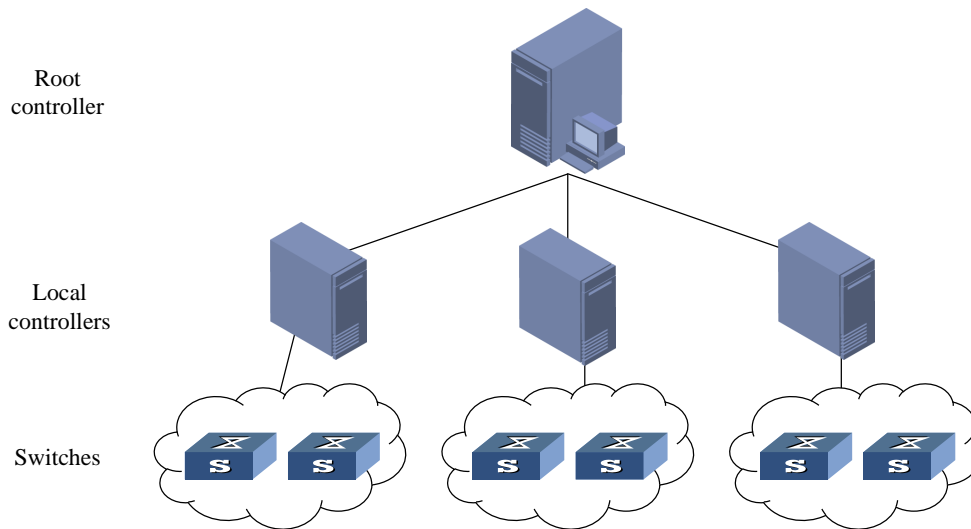


Figure 2.4: The model of hierarchical architecture

There are two types of controllers in the hierarchical architecture: root controllers and local controllers. Local controllers can only manage their own domains. Meanwhile, root controllers manage the whole network, and only root controllers have the global view of the entire network. Local controllers must request to root controllers before handling inter-domain events. When a local controller

receives a packet-in message, it calculates the best path. If all the switches along the path are in its domain, the local controller will respond immediately. Otherwise, the local controller must request to a root controller and execute the instructions from the root controller. In this way, the hierarchical architecture avoids frequent communication among the controllers.

2.2 Mathematical Background

In this section we introduce the mathematical background of this thesis including exponential distributions, PH distributions, MAP and the $M/PH/1$ queue .

2.2.1 Exponential Distribution

At first, we will introduce exponential distributions. Exponential distributions are the most basic distributions in queueing models. Exponential distributions are usually used to describe the time interval between two arrivals or the service time of a server. An exponential distribution is a continuous probability distribution used to model the time we need to wait before a given event occurs. A random variable X is distributed Exponentially with rate λ if the pdf (probability density function) of X is:

$$f(x) = \begin{cases} \lambda e^{(-\lambda x)} & x \geq 0 \\ 0 & x < 0 \end{cases} \quad (2.1)$$

The cdf (cumulative distribution function) of an exponential distribution is

$$F(x) = \begin{cases} 1 - e^{(-\lambda x)} & x \geq 0 \\ 0 & x < 0 \end{cases} \quad (2.2)$$

The mean of an exponential distribution is

$$E[X] = \frac{1}{\lambda}. \quad (2.3)$$

The variance is

$$Var[X] = \frac{1}{\lambda^2}. \quad (2.4)$$

The k -th moment is

$$M(X^k) = \lambda^{-n} n!. \tag{2.5}$$

2.2.2 PH Distributions

Queueing models are often used for performance evaluation of computer systems, and PH distributions are very popular to model the service time of systems [18]. PH distributions have been very successfully used in distribution fitting as they are able to fit different outcomes of one experiment into one distribution. These distributions are able to fit a large class of probability distributions on the positive real axis. Because PH distributions can model various real-world phenomena, they are very useful for performance evaluation.

First, we define one representation of PH distribution with a continuous-time Markov chain (CTMC). Given a CTMC with one absorbing state, as shown in Figure 2.5, we start at state k with probability α_k and state will change as time goes. The time it stays on a state follows an exponential distribution. Thus, it will take some time to reach the absorbing state (state 4 in Figure 2.5) from an initial state and the time is a sum of samples from exponential distributions. For a given CTMC, a PH distribution is the distribution of the time that can be observed along the paths to the absorbing state. This can be summarized as follows:

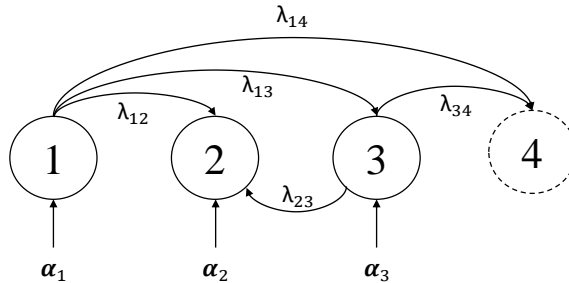


Figure 2.5: A CTMC with one absorbing state

A PH distribution is defined as the distribution of the time to absorption in a CTMC with one absorbing state.

Besides the intuitive definition, a PH distribution is commonly represented by a tuple (α, T) , where α is a vector and T is a square matrix. The tuple describe the transient of a CTMC. α represents the probability of initial state in the transient. T is a sub-matrix of generator of the CTMC, it represents the transition rates among the states. We can refer (α, T) as a Markovian representation of a PH distribution if

$$\boldsymbol{\alpha} = (\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, \dots, \boldsymbol{\alpha}_m) \in \mathbb{R}^m, \quad (2.6)$$

$$\boldsymbol{\alpha} \mathbf{1} = 1, \quad (2.7)$$

$$\boldsymbol{\alpha} \geq 0, \quad (2.8)$$

and

$$\mathbf{T} = \begin{pmatrix} \lambda_{11} & \cdots & \lambda_{1n} \\ \vdots & \ddots & \vdots \\ \lambda_{n1} & \cdots & \lambda_{nn} \end{pmatrix} \in \mathbb{R}^{m \times m} \quad (2.9)$$

is a non-singular matrix with

$$\lambda_{ii} < 0, \quad (2.10)$$

$$\lambda_{ij} > 0 \quad \text{where } i \neq j, \quad (2.11)$$

and

$$\mathbf{T} \mathbf{1} \leq 0, \quad (2.12)$$

$$\sum \mathbf{T} \mathbf{1} < 0, \quad (2.13)$$

where $\mathbf{1}$ is a column vector of ones. In the definition, we assume that there are $m + 1$ states in the CTMC, the absorbing state is the $m + 1$ state, and the size of \mathbf{T} is $m \times m$. With a Markovian representation, the generator matrix of embedded CTMC is

$$\hat{\mathbf{T}} = \begin{pmatrix} \mathbf{T} & -\mathbf{T} \mathbf{1} \\ \mathbf{0} & 0 \end{pmatrix} \in \mathbb{R}^{m \times m}, \quad (2.14)$$

Given a PH distribution $(\boldsymbol{\alpha}, \mathbf{T})$, the pdf is

$$f(x) = \boldsymbol{\alpha} e^{\mathbf{T}x} (-\mathbf{T} \mathbf{1}). \quad (2.15)$$

The cdf is

$$F(x) = 1 - \boldsymbol{\alpha} e^{\mathbf{T}x} \mathbf{1}. \quad (2.16)$$

The k th moment is

$$E[X^k] = k! \boldsymbol{\alpha} (-\mathbf{T})^{-k} \mathbf{1}. \quad (2.17)$$

There are $m(m + 1)$ free parameters in an m -phase PH distribution. Too many free parameters increase the computation in distribution fitting. PH distributions have different representations based on the structure of Markov chain. Some representations can reduce the number of free parameters. We introduce two special cases of PH distributions, Erlang distribution and hyper-Erlang distribution.

An Erlang distribution is the sum of k i.i.d exponential random variables with rate λ . We denote an Erlang distribution $Er(k, \lambda)$. There are only two parameters in an Erlang distribution. The CTMC representation of an Erlang distribution is shown in Figure 2.6.

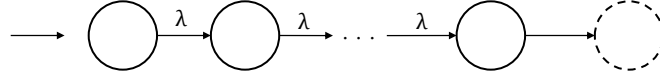


Figure 2.6: CTMC representation of an Erlang distribution

A hyper-Erlang distribution has a branch structure. Let $(\boldsymbol{\alpha}, \mathbf{T})$ be a representation of a PH distribution. The representation has a branch structure if

$$\mathbf{T} = \begin{pmatrix} \mathbf{T}_1 & & & \\ & \mathbf{T}_2 & & \\ & & \dots & \\ & & & \mathbf{T}_n \end{pmatrix}, \quad (2.18)$$

where $\mathbf{T}_1, \mathbf{T}_2, \dots, \mathbf{T}_n$ are the generator matrixes of the branches.

Within a branch structure, the following intuitive interpretation is admitted in terms of CTMC: a representation consists of blocks of states, and the blocks are not connected. Therefore, only one block can be visited in a transition. A hyper-Erlang distribution is a typical representation of branch structure. Each branch of a hyper-Erlang distribution is an Erlang distribution. The CTMC of a hyper-Erlang distribution is shown in Figure 2.7.

A PH distribution $(\boldsymbol{\alpha}, \mathbf{T})$ is a hyper-Erlang distribution if

$$\boldsymbol{\alpha}_i > 0 \quad \text{if} \quad i \in \{1, 1 + s_1, \dots, 1 + \sum_{j=0}^{n-1} b_j\}, \quad (2.19)$$

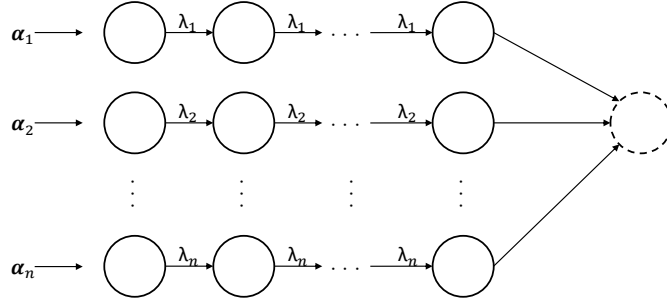


Figure 2.7: CTMC representation of a hyper-Erlang distribution

$$\text{else } \alpha_i = 0, \quad (2.20)$$

and

$$\mathbf{T}_j \in \mathbb{R}^{b_j \times b_j}, \quad (2.21)$$

$$\mathbf{T}_j = \begin{pmatrix} -\lambda_j & \lambda_j & & \\ & \dots & \dots & \\ & & -\lambda_j & \lambda_j \\ & & & -\lambda_j \end{pmatrix}, \quad (2.22)$$

where $j \in [1, n]$ and $\lambda_j > 0$.

2.2.3 Markovian Arrival Process

An essential feature of hyper-Erlang distribution is that an observation belongs to the Erlang branch m always with probability α_m . This creates no correlation in PH distributions. However, in many data sets, such as those sampled from communication systems, there may be strong correlation between inter arrival times. In such cases, to represent correlation, a MAP should be used instead of a PH distribution. A MAP is also very often used to generate correlated packet arrivals. An advantage of MAP is its ability to represent time correlation in arrival streams, as is commonly observed in the internet traffic.

We can decompose the MAP into two parts: the part generating arrivals \mathbf{D}_1 and the part controlling the internals of the generating process \mathbf{D}_0 . In formal words, a MAP of order n is usually defined by two $n \times n$ matrices $(\mathbf{D}_0, \mathbf{D}_1)$. Matrix \mathbf{D}_0 contains the rates of internal transition without

an arrival and matrix D_1 describes the transitions rate with an arrival.

$D = D_0 + D_1$ is an irreducible generator of the embedded n -state CTMC. Let φ be the steady state probability vector of the embedded CTMC, then φ is the solution of the linear system $\varphi D = 0$, $\varphi \mathbf{1} = 1$.

The CTMC embedded in a MAP is shown in Figure 2.8. A transition starts at state k with probability φ_k and the state will change as time goes. The time it stays in a state follows an exponential distribution. If the transition goes through a dashed arch, we call it an arrival. Thus, it will take some time before an arrival occurs from state i and the time is a sum of samples from exponential distributions. For a given CTMC, a MAP is the distribution of the time that can be observed along the paths. The solid arches are presented in the matrix D_0 and the dashed arches are presented in the matrix D_1 .

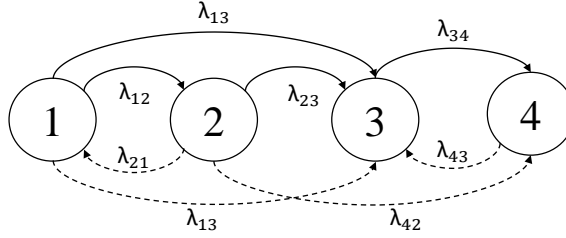


Figure 2.8: CTMC representation of a MAP

Define $P = (-D_0)^{-1}D_1$ as the state transition probability matrix of the embedded process. The stationary vector $\pi P = \pi$, $\pi \mathbf{1}^T = 1$ includes the distribution just after an arrival. The steady state distributions of the original and the embedded processes are related as

$$\varphi = \frac{\pi(-D_0)^{-1}}{\pi(-D_0)^{-1}\mathbf{1}} = \lambda\pi(-D_0)^{-1}. \quad (2.23)$$

In steady state, the inter-arrival time X is PH distributed with initial probability vector π , and generator D_0 . Therefore, the pdf of the inter-arrival time is

$$f(t) = \pi e^{D_0 t} (-D_0 \mathbf{1}). \quad (2.24)$$

The k -th moments of the inter-arrival time process are given by

$$E[X^k] = k! \pi (-D_0)^{-k} \mathbf{1}^T, \quad (2.25)$$

and the lag- k correlation is computed as

$$\phi_k = \frac{\lambda^2 \boldsymbol{\pi}(-\mathbf{D}_0)^{-1} P^k (-\mathbf{D}_0)^{-1} \mathbf{1} - 1}{2\lambda^2 \boldsymbol{\pi}(-\mathbf{D}_0)^{-1} (-\mathbf{D}_0)^{-1} \mathbf{1} - 1}. \quad (2.26)$$

2.2.4 M/PH/1 Queue

An $M/PH/1$ queueing model consists of a single server in which the service time follows a PH distribution, and the customers arrive into the system following a Poisson process. Such a queueing model is shown in Figure 2.9.

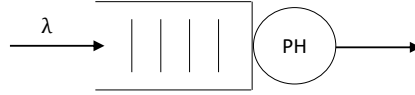


Figure 2.9: $M/PH/1$ queueing model

Since PH distributions can approximate any non-negative distributions and a Poisson process can be observed in many phenomena [42, 49], the $M/PH/1$ queue is widely used to describe system behaviors [40, 45]. The mean service time is $1/\mu = \boldsymbol{\alpha}(-\mathbf{T})^{-1} \mathbf{e}$ in an $M/PH/1$ queue with arrival rate λ and service time $t \sim PH(\boldsymbol{\alpha}, \mathbf{T})$. We denote the server utilization $\rho = \lambda/\mu$.

An $M/PH/1$ queue can be studied as a QBD process with the state space $E = \{0, (i, j), i \geq 1, 1 \leq j \leq v\}$, where v is the phase of the PH distribution. The state 0 corresponds to the empty queue, the state (i, j) corresponds to having i customers in the system and the service process in the phase j . The generator \mathbf{Q} is given by

$$\mathbf{Q} = \begin{pmatrix} -\lambda & \lambda \boldsymbol{\alpha} & 0 & 0 & 0 \\ \tau & \mathbf{T} - \lambda \mathbf{I} & \lambda \mathbf{I} & 0 & 0 \\ 0 & \tau \boldsymbol{\alpha} & \mathbf{T} - \lambda \mathbf{I} & \lambda \mathbf{I} & 0 \\ 0 & 0 & \tau \boldsymbol{\alpha} & \mathbf{T} - \lambda \mathbf{I} & \lambda \mathbf{I} \\ 0 & 0 & \ddots & \ddots & \ddots \end{pmatrix}. \quad (2.27)$$

Denote $\mathbf{x} = x_0, x_1, x_2, \dots$ the steady stationary probability vector. The steady state equations are

$$-\lambda x_0 + x_1 \tau = 0, \quad (2.28)$$

$$\lambda x_0 \boldsymbol{\alpha} + \mathbf{x}_1 (\mathbf{T} - \lambda \mathbf{I}) + x_2 \tau \boldsymbol{\alpha} = 0, \quad (2.29)$$

$$\lambda \mathbf{x}_{i-1} + \mathbf{x}_i (\mathbf{T} - \lambda \mathbf{I}) + \mathbf{x}_{i+1} \tau \boldsymbol{\alpha} = 0, \quad (2.30)$$

where x_0 is the probability that there are no customers in the system. $\mathbf{x}_i = \mathbf{x}_{i,0}, \mathbf{x}_{i,1}, \mathbf{x}_{i,2}, \dots, \mathbf{x}_{i,v}$ and $x_{i,j}$ is the probability that there are i customers in the system and the service process in the phase j . The probability that there are i customers in the system is $\sum \mathbf{x}_i$.

Multiply Equation 2.29 and Equation 2.30 by $\mathbf{1}$ on the right, we get

$$\mathbf{x}_{i+1} \tau = \lambda \mathbf{x}_i e, \quad i \geq 1. \quad (2.31)$$

Multiply Equation 2.31 by $\boldsymbol{\alpha}$ on the right and combine with Equation 2.30.

$$\mathbf{x}_i (\lambda \mathbf{I} - \lambda \mathbf{1} \boldsymbol{\alpha} - \mathbf{T}) = \lambda \mathbf{x}_{i-1}, \quad i \geq 2, \quad (2.32)$$

and similarly,

$$\mathbf{x}_1 (\lambda \mathbf{I} - \lambda \mathbf{1} \boldsymbol{\alpha} - \mathbf{T}) = \lambda x_0 \boldsymbol{\alpha}. \quad (2.33)$$

Therefore,

$$\mathbf{x}_i = x_0 \boldsymbol{\alpha} \mathbf{R}^i, \quad i \geq 1, \quad (2.34)$$

where $\mathbf{R} = \lambda (\lambda \mathbf{I} - \lambda \mathbf{1} \boldsymbol{\alpha} - \mathbf{T})^{-1}$

Since \mathbf{x} is the stationary probability vector,

$$\sum_{i=0}^{\infty} \mathbf{x}_i = \mathbf{1}, \quad (2.35)$$

$$\begin{aligned}
 \sum_{i=0}^{\infty} \mathbf{x}_i &= x_0 + x_0 \boldsymbol{\alpha} \sum_{i=1}^{\infty} \mathbf{R}^i \\
 &= x_0 + x_0 \boldsymbol{\alpha} \mathbf{R} (\mathbf{I} - \mathbf{R})^{-1} \mathbf{e} \\
 &= x_0 - \lambda x_0 \boldsymbol{\alpha} (\lambda \mathbf{1} \boldsymbol{\alpha} + \mathbf{T})^{-1} \mathbf{1} \\
 &= x_0 - \lambda x_0 \boldsymbol{\alpha} \mathbf{T}^{-1} (\mathbf{I} + \lambda \mathbf{1} \boldsymbol{\alpha} \mathbf{T}^{-1})^{-1} \mathbf{1} \\
 &= x_0 - \lambda x_0 \boldsymbol{\alpha} \mathbf{T}^{-1} \sum_{k=0}^{\infty} (-1)^k \lambda^k (\mathbf{e} \boldsymbol{\alpha} \mathbf{T}^{-1})^k \mathbf{1} \\
 &= x_0 - \lambda x_0 \boldsymbol{\alpha} \mathbf{T}^{-1} (\mathbf{I} - \lambda (1 - \rho)^{-1} \mathbf{e} \boldsymbol{\alpha} \mathbf{T}^{-1}) \mathbf{1} \\
 &= x_0 + x_0 \rho + x_0 \rho^2 (1 - \rho)^{-1}
 \end{aligned} \tag{2.36}$$

so that $x_0 = 1 - \rho$.

The average number of customers in the queue can be obtained as

$$\begin{aligned}
 E[N] &= \sum_{i=1}^{\infty} i \mathbf{x}_i \mathbf{e} \\
 &= \sum_{i=1}^{\infty} i \mathbf{x}_1 \mathbf{R}^{i-1} \mathbf{1} \\
 &= \mathbf{x}_1 \sum_{i=1}^{\infty} \frac{d}{d\mathbf{R}} \mathbf{R}^i \mathbf{e} \\
 &= \mathbf{x}_1 \frac{d}{d\mathbf{R}} \left(\sum_{i=1}^{\infty} \mathbf{R}^i \right) \mathbf{1} \\
 &= \mathbf{x}_1 \frac{d}{d\mathbf{R}} ((\mathbf{I} - \mathbf{R})^{-1} - \mathbf{I}) \mathbf{1} \\
 &= \mathbf{x}_1 (\mathbf{I} - \mathbf{R})^{-2} \mathbf{e}
 \end{aligned} \tag{2.37}$$

According to Little's law, we get the average response time $E[S]$.

$$\begin{aligned}
 E[S] &= \frac{E[N]}{\lambda} \\
 &= \frac{\mathbf{x}_1 (\mathbf{I} - \mathbf{R})^{-2} \mathbf{1}}{\lambda}
 \end{aligned} \tag{2.38}$$

The mean waiting time is

$$E[T_Q] = E[S] - \frac{1}{\mu}. \quad (2.39)$$

The mean number of customers in the system is

$$E[N] = \sum_{i=0}^{\infty} i x_i \alpha R^i. \quad (2.40)$$

Chapter 3

Related Work

OpenFlow offers flexibility by the programmable and centralized control plane. However, the OpenFlow architecture introduces new delays and may influence the performance of the whole network. OpenFlow pays a performance penalty due to the traffic between the switches and the controller [59]. The controller should handle all the flows in the network and may become a bottleneck of the OpenFlow network. The performance of the controller is a significant issue in a large network because massive requests from the switches lead to long delays [13,96]. Many researchers investigated the performance of OpenFlow networks. In this chapter, we survey existing work about the performance of OpenFlow networks.

3.1 Performance Modeling

Azodolmolky et al. used network calculus theory to model the performance of OpenFlow switches and controllers [7]. They derived a closed form formula for packet delay and queue length. They estimated the packet processing time of OpenFlow switches in the worst case. They also analyzed how the buffer size impacts the performance of OpenFlow controllers and computed the required buffer space of an OpenFlow controller. Their work can help network designers to have a quick view of the performance of OpenFlow network and necessary buffer size of OpenFlow switches and controllers. Furthermore, Azodolmolky et al. also applied network calculus theory to determine the necessary buffer size of the root controller in a hierarchical controller scenario [8].

Jarschel et al. derived a queueing model for the packet processing delay of OpenFlow networks and the blocking probability of OpenFlow switches [59]. They validated the model by a simulation in OMNeT++ [140]. Their results show that the packet delay in a network mainly depends on

the performance of the OpenFlow controller. They measured the response time of an OpenFlow controller and concluded that the response time of the controller impacts the variation of the packet sojourn time severely. Their work also shows that the probability of new flows arriving at OpenFlow switches impacts the performance of OpenFlow significantly. Given certain parameters, their model can analyze the packet delay in an OpenFlow network. Mahmood et al. proposed a method to evaluate how much time a packet spends on average in an OpenFlow network [82]. They modeled the data plane as a Jackson network and the controller as an $M/M/1$ queue. They derived the pdf and cdf of the time spent by a packet in an OpenFlow network for a given path. Their model can also determine how many packets a network can process given the average delay.

Xiong et al. used a $M^x/M/1$ queue to model an OpenFlow switch to estimate the packet forwarding time [148]. They modeled the OpenFlow controller as a $M/G/1$ queue to estimate its response time. By solving these two models, they obtained the average packet sojourn time and the corresponding pdf in a network. They also measured the response time of an OpenFlow controller using Cbench [123] and used the measured samples to evaluate their controller model. Yao et al. aimed at obtaining the capacity of OpenFlow controllers [151]. They modeled the flow packet-in requests from switches to the controller as a batch arrival process. They obtained an expression of average flow setup time and determined the maximum number of switches a controller can manage. Moreover, they extended the scene from a single controller to multiple controllers.

Mahmood et al. used a modified Jackson network to model OpenFlow networks [83]. They built the model to estimate the average packet sojourn time and the distribution of the time spent in the network by a packet. They evaluated their model in a simulation. They also analyzed the effects of key parameters in an OpenFlow network including flow setup time, arrival rate at a controller, packets sojourn time and network throughput. There was only one switch in their model but they offered a simple case that showed how their model can be used in a scenario of multiple switches. Javed et al. divided the flow setup time into deterministic delay and stochastic delay [60]. They considered the transmission time as the deterministic delay and the response time of the controller as the stochastic delay. They fitted a log-normal distribution to the response time of a controller and modeled the flow setup process as an $M/G/1$ queue. They setup an OpenFlow network in Mininet [130] and generated traffic by iPerf.

The models of OpenFlow networks are summarized in Table 3.1.

Table 3.1: Models of OpenFlow networks

Reference	Model	Main purpose
Azodolmolky et al. [7]	Network calculus	Determining necessary buffer size of OpenFlow switches and controllers
Azodolmolky et al. [8]	Network calculus	Determining the necessary buffer size of root controller
Jarschel et al. [59]	$M/M/1$	Analyzing the forwarding speed and blocking probability
Mahmood et al. [82]	$M/M/1$	Determining controller capacity
Xiong et al. [148]	$M^x/M/1$ and $M/G/1$	Analyzing throughput and packets sojourn time in a network
Yao et al. [151].	$M/M/1$	Determining the capacity OpenFlow controllers
Mahmood et al. [83]	$M/M/1$	Determining the packet sojourn time
Javed et al. [60]	$M/G/1$	Analyzing flow setup time

3.2 Benchmarking Tools

A controller administrates forwarding devices and provides an application interface to the users. It plays a very important role in the entire network. The performance of controllers influences a network severely. As we introduced in Chapter 2, there are many different controllers developed by different organizations and written in different programming languages. That makes each controller better suited for certain scenarios than others. Some researchers developed benchmark tools to measure the performance of controllers. The benchmark tools can help users to choose the suitable controller for their scenarios.

In [124] Sherwood et al. developed Cbench to benchmark different controller implementations. Cbench creates a set of virtual switches sending requests to OpenFlow controllers, and it can be used to measure the performance of the controllers. However, it can only obtain coarse-grained performance metrics, such as minimum, maximum and average response time of a controller. Tootoonchian et al. [135] used Cbench to measure several performance aspects of different OpenFlow controllers. The authors measured the minimum and maximum controller response time, maximum throughput, and the throughput and latency of the controller with a bounded number of packets. Their experimental results showed that a single controller is not enough to manage a large scale network. In [115], the authors developed hcprobe, a framework that can be used to test the performance

of controllers. They also analyzed the performance of popular open-source OpenFlow controllers. Hcprobe introduced an embedded domain-specific language for users to create custom tests. Jarschel et al. developed OFCBenchmark, a more flexible benchmark tool based on Cbench [57]. Unlike Cbench which creates independent switches OFCBenchmark creates a set of virtual switches that generate and send LLDP packets to each other. The virtual switches act more like a network. OFCBenchmark can get performance statistics for each virtual switch. In [58], Jarschel et al. further developed OFCProbe. To emulate a real network, OFCProbe can create the payload of packet-in messages from a PCAP file. Users can capture packets from a network and use OFCProbe to replay the traffic. In OFCProbe, each virtual switch can be assigned a file that contains the traffic information.

The key features of different benchmark tools are summarized in Table 3.2.

Table 3.2: Models of OpenFlow networks

Reference	Tool	Key feature
Sherwood et al. [124]	Cbench	Generating basic performance metrics
Shalimov et al. [115]	Hcprobe	Custom tests by a specific language
Jarschel et al. [57]	OFCBenchmark	Emulating an OpenFlow network
Jarschel et al. [58]	OFCProbe	Replaying real traffic

3.3 Performance Improvement

The main overhead of an OpenFlow network is the communication on the OpenFlow channel. The OpenFlow specification [41] defines a buffer in the OpenFlow switch to reduce the traffic in an OpenFlow channel. OpenFlow switches will send a part of a packet with a buffer identification instead of the whole packet to a controller to request the forwarding rules. This is the default way to reduce the traffic in OpenFlow channels by reducing the size of packet-in messages. However, the buffer is designed at packet-granularity. It can not reduce the number of messages. The buffer can hardly improve the performance of an OpenFlow network. Many researchers are trying to improve the performance of OpenFlow networks by reducing the number of messages from OpenFlow switches.

Curtis et al. proposed DevoFlow to reduce the communication between switches and the controller [34]. They analyzed the overhead of the OpenFlow architecture and concluded that OpenFlow switches involve the controller too frequently both in flow setup and statistics-gathering. They designed a new OpenFlow switch that can reduce the overheads. In DevoFlow, they put a CLONE

flag in a match field of a flow entry. If the flag is clear, the switch behaves the same as a normal OpenFlow switch. Otherwise, the switch creates a new flow entry. In the new flow entry, all the flagged match fields are replaced by values matching the flow entry and other aspects are inherited from the flow entry. The DevoFlow switches can handle some flows without controller so that the traffic on the OpenFlow channel is reduced. Their result showed that DevoFlow used 10–53 times fewer flow table entries and uses 10–42 times fewer control messages.

In [69], Daisuke et al. proposed a method to reduce CPU utilization in both OpenFlow switches and controllers to reduce the traffic between them. They categorized packet-in messages into three types: State Change, Flow Setup and Forward. The Forward type of messages are less important than the others. Forward messages do not change the state of a network. They proposed a method to identify Forward messages in switches and set a rate limiter for the Forward messages. They implemented their method into Open vSwitch [99]. The evaluation showed that their method can reduce the messages from switches and decrease the utilization of CPU in both switches and controller.

Mao et al. proposed FPB, a flow-granularity buffer management model for OpenFlow switches. FPB has higher performance than the existing packet-granularity buffer management with less communication overhead between switches and controllers. They built *packet-in buffer table* into a switch. An entry in *packet-in buffer table* records the buffer information of a flow, which includes the match fields, start index, current index, packet count and timeout. Before the switch sends a packet to the controller, it looks for an entry in the *packet-in buffer table* that matches the packet. If it finds a matched entry, the packet will be buffered in the *packet-in buffer table*. Otherwise, the packet will be sent to the controller. They also designed a flow action pre-processing mechanism to prevent packets disorder. To evaluate the FPB model, they built prototypes on both software and hardware switches. Experimental results showed that FPB can effectively reduce the communication overhead and decrease the probability of packets disorder.

A packet may be dropped in the flow installation process if the packet arrives earlier than its corresponding flow rules at the switch. To solve this problem, Awan et al. proposed a new mechanism that ensured flow installation before the corresponding packet [6]. They measured the delay between any two switches, and the delay between the switches and the controller. Based on the delays, they determined the order of flow installations. In their approach, some switches are managed in a proactive way. Therefore, the approach can reduce the communication overhead between the switches and the controller. Their experimental results also show that the approach can reduce the communication overhead significantly when there are lots of flows in the network.

Shirali-Shahreza et al. proposed ReWiFlow to reduce the programming complexity of flow entries [125]. ReWiFlow adds a restriction into flow entry. ReWiFlow defines an order among all header

fields that can appear in a flow entry. A header field can be in a flow entries only if the header field before it is in the flow entry. ReWiFlow does not need any changes of the OpenFlow switch specification. It is compatible with normal OpenFlow switches. Their experiments showed that ReWiFlow can decrease the flow setup time and improve the performance of matching process.

The main workload for a controller is the installation of flow entries into OpenFlow switches. Source routing is a good way to reduce flow entry installations because the path information is embedded in the packet header. The controller only needs to install flow entries into the edge switch to assign paths for flows, all the intermediate switches can get the forwarding instructions in the packet headers. Soliman et al. first proposed source routing in OpenFlow networks [127]. Their experiments showed that source routing can reduce 77.6% traffic between the switches and controller in a 34 nodes network, and the more nodes are in a network, the more the traffic will be reduced. Since source routing is a very good way to reduce the communication between the switches and the controller, many researchers applied source routing to improve the scalability and performance of OpenFlow networks [11, 48, 77, 78, 128].

Source routing can reduce the load of controllers significantly. However, the OpenFlow protocol does not support source routing. There are no bits reserved in the packet header for path information. To implement source routing in an OpenFlow network, some researchers have encoded the path into the header of each packet. When a packet arrives at a network, the switch modifies its header and puts the path information into its header. When the packet leaves the network, the switch removes the path information and restores its header. In [62], the authors discussed some possible ways to implement source routing in OpenFlow networks. They observed that path information can be encoded compactly into a single field in a large network. They also proposed an approach to use source routing in data center networks. The experimental results show that the memory utilization of the switches was linearly dependent on the network diameter if source routing was applied. Moreover, the throughput was not negatively impacted when any feasible path can be selected at the edge switches.

Ramos et al. presented SlickFlow, a resilient source routing approach in OpenFlow networks [105]. In SlickFlow, two paths are encoded in a packet, a primary path and an alternative one. If failures happen along the primary path, packets can be rerouted to the alternative path without the controller. They implemented SlickFlow based on Open vSwitch and evaluated it in three topologies. The experiments showed that SlickFlow can reduce the number of requests from the switches and achieve failure recovery without the controller.

3.4 Performance of Multiple Controllers

A single controller is not capable of managing all the switches in a large network. Some researchers designed distributed controllers to improve the reliability and scalability of the control plane [16, 70, 100, 153]. Distributed controllers act as a logically centralized controller. The controllers should exchange network information to keep a consistent view of the whole network. There are problems in the information exchange process, e.g., how to choose the number of controllers and how to determine the assignment relationship between the switches and the controller. Many metrics can be used for tackling these problems. Some researchers focus on a certain metric, while others combine several metrics and try to find the best trade-off [112, 137].

Yao et al. used a capacitated K-center algorithm to avoid overloaded controllers [149]. Their algorithm can reduce the number of required controllers and balance the load among the controllers. Bari et al. proposed a heuristic algorithm to reduce the flow setup time and load of controllers [9]. They used integer linear programming to solve the controller provisioning problem. Their method can determine the optimal number of controllers and the best location of the controllers. They developed a controller management framework that can collect relevant statistics from the controllers and balance the load of controllers. Rath et al. proposed an algorithm based on non-zero-sum game theory to distribute the requests from switches among controllers uniformly [106]. They deployed an optimization engine at each controller. The engine compared its load with its neighbors and added a new controller or deleted an existing controller based on the comparison. Hu et al. proposed BalanceFlow to balance the load of controllers by switch migrations [54]. They designed one controller as super controller. The super controller can adjust the assignment among the switches and controllers. When a controller is overloaded, the super controller runs a load balancing algorithm to reduce the load of the controller. Cello et al. proposed an algorithmic solution to balance the load of controllers [27]. They evaluated the solution based on Matlab simulations with random traffic. The result showed that their method can reduce the variance of the load of controllers by 40%. Wang et al. proposed a greedy algorithm to balance the load among controllers [141]. They collected the load information from the controllers and decided whether to migrate switches. Then they established a tradeoff between the migration cost and the load balancing to decide where to migrate the overloaded switch.

Filali et al. formulated the controller assignment problem as a one-to-many matching game with a minimum utilization each controller has to achieve [38]. To balance the load among controllers, they proposed an algorithm to ensure a stable matching between switches and controllers. Zhang et al. proposed an adaptive controller assignment scheme for multiple controllers that can dynamically

adjust the number of controllers and the mapping relationship among controllers and switches [157]. They designed three algorithms for expanding the controller pool, shrinking the controller pool and controller load balancing, respectively. Huque et al. combined the controller placement problem and dynamic flow management [139]. They aimed at achieving high utilization at low power consumption and maintenance cost. They proposed an algorithm to determine the number of controllers per module for the dynamic load and the best locations of controller modules. Wang et al. formulated the controller assignment problem as a stable matching problem and proposed a hierarchical two phase algorithm that used matching theory and coalitional games to minimize the average response time of the control plane [143].

Song et al. proposed a lightweight load balancing scheme for distributed controllers [129]. In the scheme, switches are not migrated. Instead, idle controllers share workloads with overloaded controllers. Therefore, their method can avoid the migration cost. Kyung introduced a load distribution method that can reduce the blocking probability [72]. When the load of a controller reaches a certain threshold, the controller forwards incoming requests to another controller. Zhou et al. [159] presented DALB, a distributed load balancing algorithm. DALB is a module of an OpenFlow controller. It collects load information of the controller. If the load of a controller exceeds a certain threshold, the controllers will elect a switch and migrate the switch to an idle controller. They built a prototype based on the Floodlight controller and evaluated DALB by simulations.

Yu et al. proposed a load balancing mechanism based on load information [155]. They set a threshold for each controller and used the threshold to determine which controller was most loaded. They preferentially selected a switch with high message arrival rate so that the overloaded controller can release the load as soon as possible. They implemented the framework for the Floodlight controller and run the simulations in Mininet. Cheng et al. formulated the controller load balancing problem as network utility maximization problem [30]. The objective of the network utility problem is to serve as many as possible requests under the available controllers. They designed a distributed algorithm that approximates the optimal solution. They implemented a prototype and evaluated their algorithm in two real topologies. Yao et al. solved the controller placement problem with the node weight, and migrated switches dynamically based on the network traffic to balance the load of controllers [150]. They split a network into multiple domains and determined the best location for the controller in each domain. To adapt dynamic flows, they also designed a switch migration algorithm to balance the load among the controller. When a controller was overloaded, the controller migrated the boundary switch with the arrival arrival rate to its neighbor.

In many works above, the researchers assumed that the service time of controller is constant. However, our measurements show that the service time is random and the distribution of service

3.4. PERFORMANCE OF MULTIPLE CONTROLLERS

has a heavy tail. In this thesis, we measure the service time of controllers, fit a PH distribution to the service time, and build queueing models for the controllers. We optimize the performance of controllers based on the queueing models.

Part II

Tools for Performance Modeling

Chapter 4

HyperStar2: Easy Distribution Fitting of Correlated Data

PH distributions play an important role in the field of performance evaluation because PH distributions can describe many real world phenomena. A PH distribution can be represented by a CTMC 2.2.2 so that models use PH distributions can be easily computed.

However, PH distributions cannot describe correlation in a data set. Many researchers use MAPs to capture the correlation in empirical data. MAPs have a long history in stochastic modeling [91,92] and are powerful modeling tools. MAPs, in theory, allow the representation of almost all relevant stochastic behaviors that are observed in practice. MAPs are popular tools to characterize correlated stochastic processes like inter-arrival times or sequences of correlated service times. MAPs were proposed in [81] and are widely used for probabilistic analysis of communication network traffic. One advantage of MAPs is that they have attractive properties from the viewpoint of stochastic processes. MAPs are general classes of stochastic processes that contain most of the commonly used arrival processes such as the Poisson process, the PH renewal process, and the Markov-modulated Poisson process (MMPP). And any general point process can be approximated by appropriate MAPs [5].

To capture empirical behavior by MAPs, the parameters of a MAP have to be fitted according to some trace resulting from observations. The fitting problem of MAPs is a nonlinear optimization problem that is very complex as there are many free parameters in the matrix representation [131]. Different fitting approaches have been proposed in the literature which all have their pros and cons. The most general approach is to find a MAP that maximizes the likelihood according to the available trace. The EM algorithm [20] can be used for this purpose and many specific variants of the

algorithm for MAP fitting are available [19]. EM algorithms have several disadvantages. They have a slow convergence, may converge towards a local minimum and require a huge effort that grows linearly in the length of the trace. So the EM algorithm is applicable only to small measurement traces. A number of MAP fitting methods were published based on the two-phase approach [22], which suggested splitting the task into two phases: fitting of the inter-arrival times in the first phase by a PH fitting method and fitting the correlation in the second phase. However, it is still an open question what are the statistics that capture the correlation structure of the trace the best. Alternative approaches first derive some quantities from a trace, like higher order moments, joint moments or lag- k auto-correlations and then fit the parameters of a MAP according to these quantities. Recent results on the characterization of MAPs revealed the importance of joint moments. Some more recent work on MAP fitting uses the lag-1 joint moment to fit the auto-correlations [21].

We have developed HyperStar2, a user-friendly tool which allows for intuitive user interaction and provides direct user feed-back. Using HyperStar2 requires no knowledge of the underlying mathematics or theoretical foundations. The tool is used by working directly with the empirical data.

In this chapter, we discuss our cluster-based approach to PH distribution and MAP fitting. We have implemented this approach as HyperStar2, a tool with a graphical user interface. The rest of this chapter is structured as follows: we describe our cluster-based fitting approach in Section 4.1. Section 4.2 gives an overview of our implementation of the HyperStar2 tool. We evaluate the method in Section 4.3 by numerical experiments. Section 4.4 concludes this chapter.

4.1 Fitting Algorithm

As we introduced in Chapter 2, a MAP is usually represented by two matrixes, D_0 and D_1 , where D_0 contains the rates of internal transition without an arrival and D_1 describes the transitions rate with an arrival. There are two steps in our fitting algorithm. In the first step, we use a cluster-based algorithm to fit a hyper-Erlang distribution to the samples. The hyper-Erlang distribution can represent the internal transition without an arrival. The generator matrix of the fitted hyper-Erlang distribution is D_0 . In the second step, we construct D_1 based on the Hyper-Erlang distribution and the clusters of samples. The samples are clustered in the first step. We count the transitions among the clusters and transfer the transitions into the D_1 matrix.

4.1.1 Constructing the D_0 matrix

The cluster-based fitting algorithm consists of splitting the samples into M clusters, fitting each cluster with an Erlang distribution and performing cluster refinement where needed.

More formally, we split all the samples $S = s_1, s_2, \dots, s_n$ into M clusters C_1, C_2, \dots, C_M using the k-means algorithm [80]. Clustering aggregates similar samples in the same cluster and thereby each sample belongs to exactly one cluster. The user can specify the number of clusters and the position of the initial cluster centers by marking important peaks of the density in the GUI. An Erlang distribution Q_m is fitted to each cluster C_m . The Erlang distribution fitting method can be chosen arbitrarily. The hyper-Erlang distribution (α, Q) is obtained as a mixture of the cluster distributions. The initial probabilities α are obtained as the relative cluster sizes

$$\alpha = \left(\frac{|C_1|}{n}, \frac{|C_2|}{n}, \dots, \frac{|C_M|}{n} \right), \quad (4.1)$$

and the generator matrix is constructed from the branch sub-generator matrices.

In the cluster refinement, we assign the samples to new clusters using the strategy described below. These two steps are repeated until either the parameters of each branch distribution no longer change or a maximal number of iterations has been exceeded. We use a probabilistic re-assignment strategy for the cluster refinement. For each sample s_i we compute a vector

$$\varepsilon = \frac{1}{\sum_{j=1}^M f_j(s_i)} (f_1(s_i), f_2(s_i), \dots, f_M(s_i)), \quad (4.2)$$

where f_M is the pdf of the M th Erlang distribution. For each cluster C_j , we use this vector to estimate the probability that the sample s_i is in the respective cluster. Let $\varepsilon_{(j)}$ denote the j th element in ε then we assign the sample s_i to the cluster C_j with probability $\varepsilon_{(j)}$. After the assignment, we obtain M new clusters. Then we fit an Erlang distribution to each cluster again. These two steps are repeated iteratively until either the parameters in each Erlang distribution do not change or a maximal number of iterations has been exceeded, whichever comes first. We store some result candidates in the iterations. If the maximal number of iterations has been reached, we use the candidate with maximum likelihood as result. In the refinement we record which cluster a sample belongs to for the later correlation fitting.

4.1.2 Constructing the D_1 matrix

For a hyper-Erlang distribution, the probability of initial phase after an arrival is always α . For a MAP, the probability of initial state after an arrival depends on the phase immediately before that arrival and D_1 describes the transition rates between phases, which are states of a CTMC.

Our approach is to analyze the relative frequency of transitions between clusters to construct the matrix D_1 based on the transition frequencies. When an arrival occurs, let $\eta_{(i,j)}$ be the probability that the next phase is j given the current phase is i , we can obtain the element of D_1 at row i column j by scaling $\eta_{(i,j)}$ with a proper factor because D_1 and $\eta_{(i,j)}$ describe the same thing using different notation. In our MAP fitting algorithm, we construct D_1 based on the relative frequency of transitions between clusters.

After a hyper-Erlang distribution is fitted, a generator matrix Q for the hyper-Erlang distribution is obtained. For MAP fitting we set $D_0 = Q$. If there are k_i phases in the Erlang distribution of cluster i it follows that D_0 is a $K \times K$ matrix, where

$$K = \sum_{i=1}^M k_i. \quad (4.3)$$

Because the matrix D_0 is associated with transitions without arrivals and we use a generator matrix of a hyper-Erlang distribution in our MAP fitting algorithm, transitions with arrivals only happen between Erlang branches. Every Erlang branch is associated to a sample cluster, we analyze transitions between clusters to get the transitions between Erlang branches that are associated with arrivals. Let C be a $K \times K$ zero matrix and $C_{(r,c)}$ be the element at row r column c , we use matrix C to count the transitions between clusters.

Let B_i be the first phase of the i th Erlang distribution and E_i the last phase of the i th Erlang distribution. We can get

$$B_i = \sum_{a=1}^{i-1} k_a + 1, \quad (4.4)$$

$$E_i = \sum_{a=1}^i k_a - 1. \quad (4.5)$$

If a sample belongs to the i th cluster and the next sample belongs to the j th cluster, where i may be equal to j , we update the counter as below:

$$C_{(E_i, B_j)} = C_{(E_i, B_j)} + 1.. \quad (4.6)$$

We can set D'_1 to the probability matrix for sample transitions

$$D'_1 = \frac{1}{n-1} C.. \quad (4.7)$$

Let $D'_{1(r,c)}$ be the element of D'_1 at position (r, c) . Hence $D'_{1(r,c)}$ is the probability that the sample generating process transits from state r to state c . We assume $D'_{1(r,c)}$ equals the probability that transitions occur between states in the MAP. To get D_1 , we must convert the probability in D'_1 to state transition rates by scaling every element in D'_1 with a suitable factor. Let F be matrix of factors, D_1 is the Hadamard product of D'_1 and F . The Hadamard product is a binary operation that takes two matrices of the same dimensions and produces another matrix of the same dimension as the operands where each element i, j is the product of elements i, j of the original two matrices. The Hadamard product is denoted using the symbol \circ .

$$D_1 = D'_1 \circ F., \quad (4.8)$$

$D_0 + D_1$ is an irreducible generator of a CTMC, so

$$(D_0 + D_1)\mathbf{1} = \mathbf{0}, \quad (4.9)$$

where $\mathbf{0}$ is a row vector of zeros of the appropriate dimension and $\mathbf{1}$ is a vector of ones of the appropriate dimension.

Let $D_{1(r,c)}$ be the (r, c) element of D_1 and let $F_{(r,c)}$ be the (r, c) element of F . Combining (4.8) and (4.9), then we can get

$$D_{1(r,c)} = \begin{cases} -\frac{D_{0(r)}}{D'_{1(r)}} D'_{1(r,c)} & \text{if } D'_{1(r)} \neq 0 \\ 0 & \text{if } D'_{1(r)} = 0 \end{cases}, \quad (4.10)$$

where $D_{0(r)}$ is the sum of the r -th row in the matrix D_0 , $D'_{1(r)}$ is the sum of the r -th row in the

matrix D'_1 .

$$D_{0(r)} = \sum_{c=1}^K D_{0(r,c)} \quad (4.11)$$

$$D'_{1(r)} = \sum_{c=1}^K D'_{1(r,c)}. \quad (4.12)$$

4.1.3 Example

In this subsection we show in a simple example how the fitting algorithm works. Assume we have a data set that contains 5000 samples. In our example the cluster-based fitting algorithm splits the data set into two clusters and fits the first cluster with generator matrix Q_1 , second cluster with generator matrix Q_2 , e.g.

$$Q_1 = \begin{pmatrix} -2 & 2 \\ 0 & -2 \end{pmatrix} \quad Q_2 = \begin{pmatrix} -5 & 5 \\ 0 & -5 \end{pmatrix}$$

The D_0 matrix is then the generator matrix of hyper-Erlang distribution

$$D_0 = Q = \begin{pmatrix} -2 & 2 & 0 & 0 \\ 0 & -2 & 0 & 0 \\ 0 & 0 & -5 & 5 \\ 0 & 0 & 0 & -5 \end{pmatrix}$$

According to (4.4) and (4.5), we obtain

$$B_1 = 1, E_1 = 2$$

$$B_2 = 3, E_2 = 4.$$

If one sample is in the first cluster and the next is in the second cluster, we count one transition from the first cluster to the second cluster. Assume in our set of 5000 samples we have 1754 pairs with both samples from the first cluster, 750 transitions from the first to the second cluster, 1750 transitions from the second to the first cluster and 745 data pairs both from the second cluster. Then

the counting matrix C is shown below.

$$C = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1754 & 0 & 750 & 0 \\ 0 & 0 & 0 & 0 \\ 1750 & 0 & 745 & 0 \end{pmatrix}.$$

We obtain for the relative frequencies D'_1

$$D'_1 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0.35 & 0 & 0.15 & 0 \\ 0 & 0 & 0 & 0 \\ 0.35 & 0 & 0.149 & 0 \end{pmatrix}$$

According to 4.10, we obtain D_1 as

$$D_1 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1.4 & 0 & 0.6 & 0 \\ 0 & 0 & 0 & 0 \\ 3.5 & 0 & 1.49 & 0 \end{pmatrix}$$

4.2 Implementation

We implemented the fitting method in the tool HyperStar2. In earlier work [107] we found that the human user can very often detect clusters much better than a fully automatic algorithm by marking the peaks in the distribution as relevant values in the observations.

The GUI support allows users to fit a hyper-Erlang distribution or a MAP to a data set quickly and accurately. The GUI mode is shown in Figure 4.1. The left panel displays the histogram and empirical CDF for the data set in different tabs. The fitted pdf and cdf are also shown in the tabs after fitting a hyper-Erlang distribution. If the fitting result is a MAP, the tools will create a third tab to show the auto-correlation.

For most cases, users do not have to set the parameters, HyperStar2 can provide good fitting results with the default values after simply clicking the *Fit* button on the right panel. If the user is not satisfied with the fitting result, because either the distribution does not sufficiently well fit the

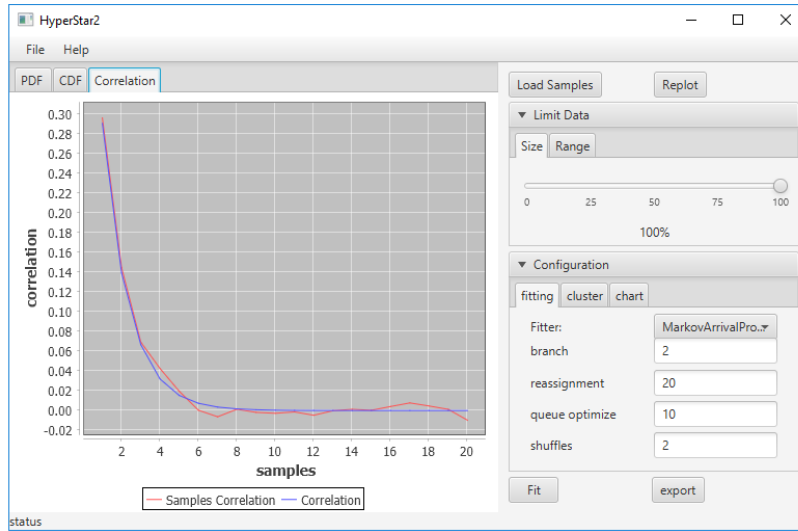


Figure 4.1: User interface of HyperStar2 with fitted result

empirical data, or the theoretical and the empirical autocorrelation do not match as expected, the peaks on the histogram can be marked differently and some fitting parameters can be set on the right panel. The density peaks are used as the initial centers of K-means algorithm.

At the *Limit data* panel, the user can choose how many samples should be used for the fit or what range of samples should be included in the fitting process. For very large data sets the fitting algorithm might become too slow. In that case samples can be removed by their order, e.g. only the first 75% samples are fitted if the user sets the size to 75. Another option is to reduce the range, which means that only samples that lie within the set range are fitted. This can be also used to remove obvious outliers. Parameters in the *fitting* tab are the most important ones, they are used by the fitting algorithm and impact the fitting result. Table 4.1 lists the fitting parameters to control the behavior of the fitting algorithm. The tool can export results after the fitting is done.

Table 4.1: Fitting parameters

Parameter	Description(default value)
Fitter	Distribution to be fitted (MAP)
Branch	Number of branches to be fitted (6)
Reassignment	Maximum number of iterations (20)
Shuffles	Number of reassignments in an iteration (2)
Queue optimize	Number of result candidates (10)

4.3 Evaluation of HyperStar2

In this section we demonstrate the fitting properties of HyperStar2 by analyzing some examples. We fit distributions and evaluate the goodness of fit by comparing the empirical and the theoretical probability density function as well as the lag-k correlation. We also compare our results with results from ProFiDo [10], which we found to be a very versatile fitting tool.

We show two examples that demonstrate the advantages and disadvantages of our tool. We first generated samples from a given MAP and then study how well HyperStar2 is able to approximate this distribution. We use samples that contain obvious peaks in the first example. There are not many samples in the overlap area between clusters. Therefore, it is clear which cluster the samples are assigned to, so the estimation of the relative frequency of transitions between cluster in the sample sequence is highly accurate and the algorithm provides good correlation fitting. In the second example, the samples do not contain obvious peaks. Hence, there are many samples in the overlap area between clusters. These samples may be assigned to different clusters. The assignment may not impact the density fitting very much, but we assume that it impairs the autocorrelation fitting, since the estimate of relative frequencies in changes between clusters in the sample sequence has much more uncertainty.

The *branch* parameter indicates into how many clusters the data is split. This is a very import parameter in our algorithm, as it impacts the fitting result severely. So we show fitting results with different number of *branches* in the two examples. The two examples show the advantages and disadvantages of HyperStar2.

The first example uses a 6-state MAP with the following matrices. Using this generator there are two states that contribute strongly to the dynamics of the model, while the remaining four states are less influential. This can also be seen in Figure 4.2, where the empirical density has two peaks.

$$D_0 = \begin{pmatrix} -10 & 10 & 0 & 0 & 0 & 0 \\ 0 & -10 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0.3 & 0 & 0 & 0.7 & 0 & 0 \end{pmatrix}$$

The first data set consists of 25000 samples generated from this MAP. We fit the samples using HyperStar2 and ProFiDo. Setting the *branch* parameter to 2, which means the algorithm splits the data into 2 clusters and there are 2 Erlang branches in the hyper-Erlang distribution. HyperStar2 gives the following result.

$$D_0 = \begin{pmatrix} -9.892 & 9.892 & 0 & 0 & 0 & 0 \\ 0 & -9.892 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 6.92 & 0 & 2.97 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0.283 & 0 & 0 & 0.717 & 0 & 0 \end{pmatrix}$$

Figure 4.2 shows the histogram of samples and the densities of the fitted distributions. The solid line represents the pdf of the result from ProFiDo. The other three lines represent the pdf of results from HyperStar2 using different parameter settings. The red line shows the 2-branch result from HyperStar2, the yellow and blue lines show the results for 3 branches and 4 branches respectively. From Figure 4.2 we observe that HyperStar2 fits this data set very well with 2, 3 or 4 branches. ProFiDo provides a good fit of the density but is not able to capture the second peak and the gap

between the peaks as precisely.

Figure 4.3 shows the autocorrelation function of the resulting MAP. As expected, the autocorrelation function is fitted well by both tools. The figure indicates that the result with 2 Erlang branches is the best of the results we have computed using HyperStar2. It is as good as the result from ProFiDo.

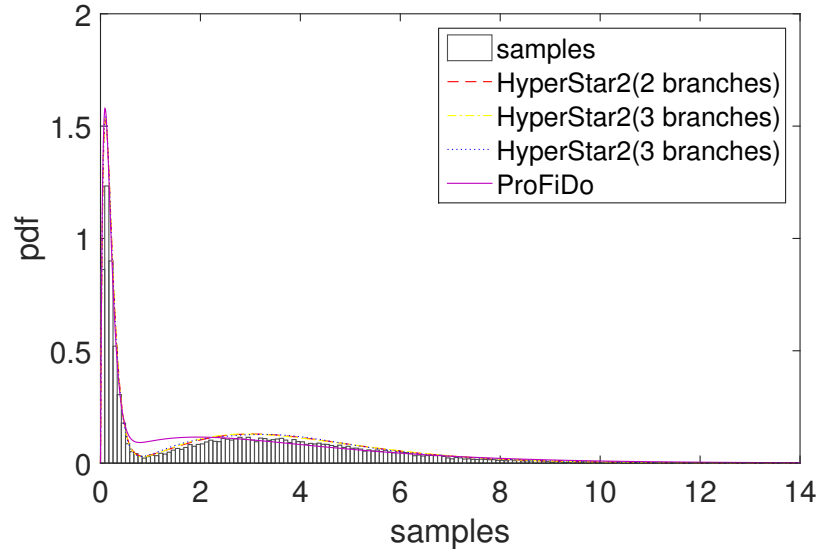


Figure 4.2: The pdf of the first example

But for different numbers of Erlang branches the results are not as good. The 3-branch and 4-branch results for the density are not as good as 2-branch result, but they are also very close to the samples' autocorrelation.

In the second example, we approximate the following MAP:

$$D_0 = \begin{pmatrix} -0.2 & 0.2 & 0 & 0 & 0 & 0 \\ 0 & -0.2 & 0.2 & 0 & 0 & 0 \\ 0 & 0 & -0.2 & 0 & 0 & 0 \\ 0 & 0 & 0 & -0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & -0.5 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & -0.5 \end{pmatrix}$$

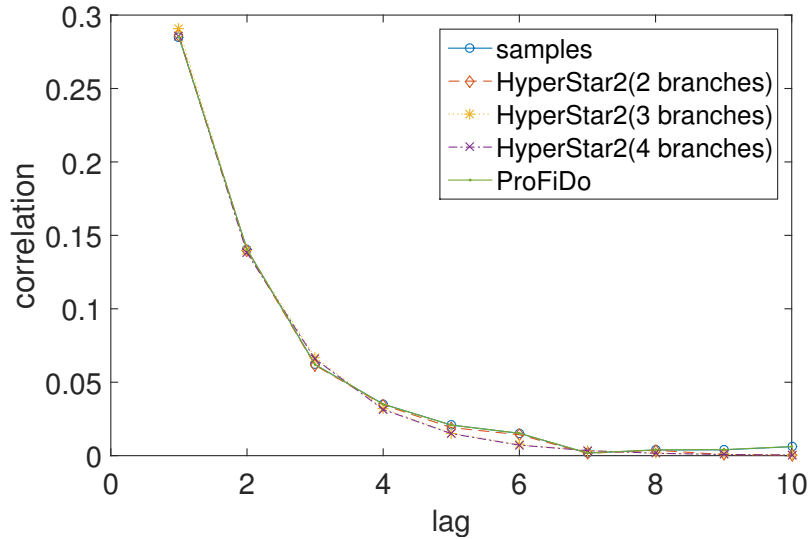


Figure 4.3: Correlation of the first example

$$D_1 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0.16 & 0 & 0 & 0.04 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0.02 & 0 & 0 & 0.03 & 0 & 0 \end{pmatrix}$$

We again generated 25000 samples from this MAP and fitted them with HyperStar2 and ProFiDo.

We used 2, 4 and 6 *branches*. It is not easy to see the peaks in this, so we did not mark any peak on the histogram. Figure 4.4 and Figure 4.5 show the fitting results.

Figure 4.4 shows that both ProFiDo and HyperStar2 can fit the distribution very well. Although there are 2 hyper-Erlang branches in the original MAP, HyperStar2 can still provide a good fit with 4, or 6 branches.

From Figure 4.5, we can see that ProFiDo fits the correlation better than HyperStar2. But the distribution fit of HyperStar2 is still better than that of ProFiDo. We assume that HyperStar2 is less suited to capture relatively small autocorrelation, which means that clusters have a strong overlap. High autocorrelation typically shows as strongly distinct clusters and those are captured much better by HyperStar2 than less distinct clusters.

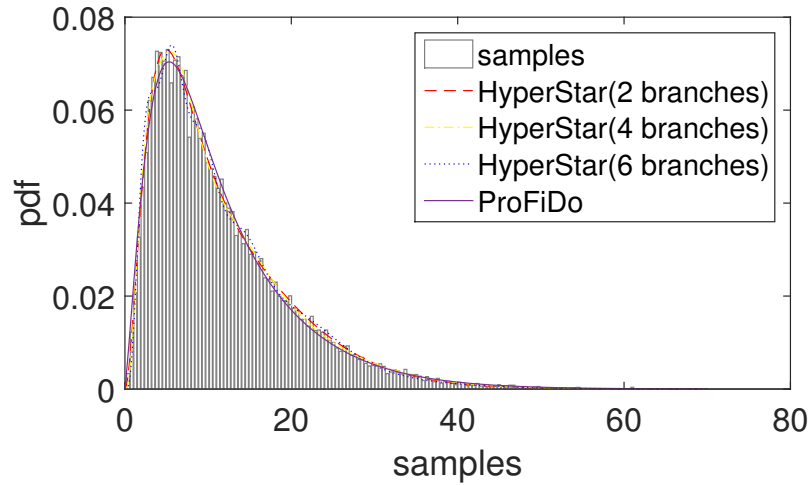


Figure 4.4: The pdf of the second example

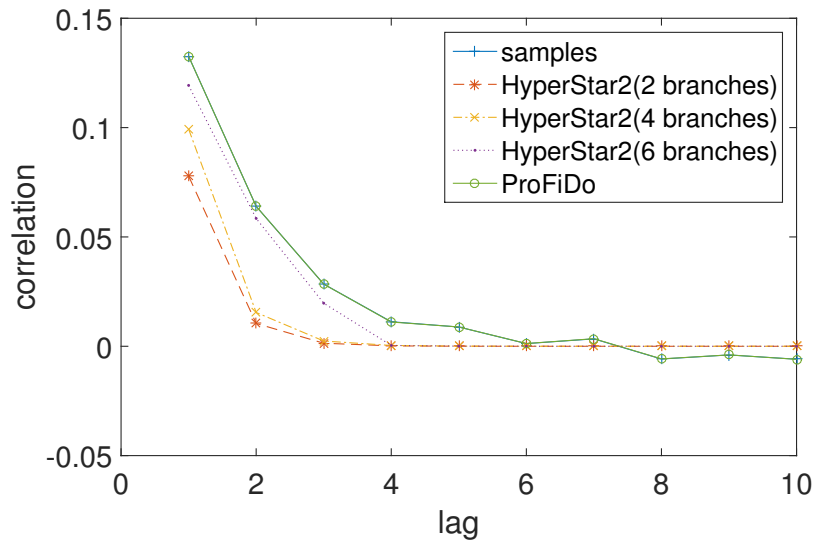


Figure 4.5: Correlation of the second example

4.4 Summary

In this chapter we have presented HyperStar2, a fitting tool for correlated data and the algorithms behind it. We have shown that HyperStar2 can capture the shape of a distribution very well. HyperStar2 can also fit an MAP to a data trace well, but autocorrelation is not represented as well as done by ProFiDo if there are not obvious peaks in the histogram of samples.

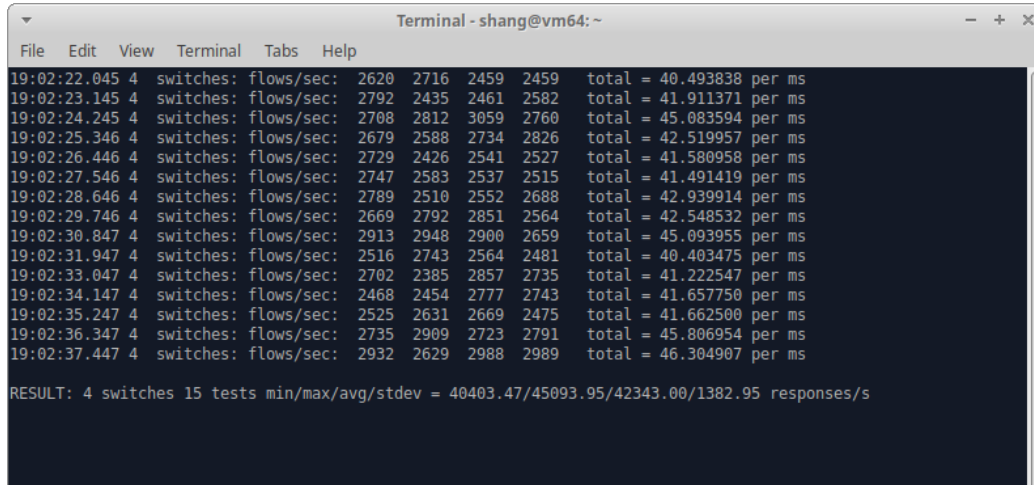
Chapter 5

An OpenFlow Controller Performance Evaluation Tool

Since the OpenFlow specification does not dictate how a controller should be implemented, there are different controllers developed by different organizations in different programming languages. That makes each controller better suited for certain scenarios than others. The current OpenFlow ecosystem is fragmented due to the variety of controller platforms. We must understand the difference between the controllers to choose an implementation or to analyze the behavior of an OpenFlow network.

There are already benchmark tools to analyze the performance of an OpenFlow controller, such as Cbench [124]. Cbench can provide users the minimum, maximum throughput as well as the mean and variance of throughput. An example benchmark result from Cbench is shown in Figure 5.1.

The metrics Cbench provides may be enough for some users, but researchers who want to build a performance model for OpenFlow controller or people who want to understand the reason of the behaviors may need more detail. The mean and variance of the response time are the most commonly used metrics in application performance management. However, in reality, the response time often has a long tail, the mean and variance cannot provide enough insight into the performance. So it is better to provide a distribution of response time. In this chapter, we introduce a user-friendly tool to obtain the performance of OpenFlow controllers. Unlike other benchmark tools that focus on throughput, our tool helps users to build models for OpenFlow networks and evaluate the performance of the controllers with the models. So we can understand the behavior of OpenFlow controllers and get a detailed analysis of the performance. Our tool aims at providing a simple way to analyze the performance of OpenFlow controllers. It can estimate the performance of their



```
Terminal - shang@vm64:~
File Edit View Terminal Tabs Help
19:02:22.045 4 switches: flows/sec: 2620 2716 2459 2459 total = 40.493838 per ms
19:02:23.145 4 switches: flows/sec: 2792 2435 2461 2582 total = 41.911371 per ms
19:02:24.245 4 switches: flows/sec: 2708 2812 3059 2760 total = 45.083594 per ms
19:02:25.346 4 switches: flows/sec: 2679 2588 2734 2826 total = 42.519957 per ms
19:02:26.446 4 switches: flows/sec: 2729 2426 2541 2527 total = 41.580958 per ms
19:02:27.546 4 switches: flows/sec: 2747 2583 2537 2515 total = 41.491419 per ms
19:02:28.646 4 switches: flows/sec: 2789 2510 2552 2688 total = 42.939914 per ms
19:02:29.746 4 switches: flows/sec: 2669 2792 2851 2564 total = 42.548532 per ms
19:02:30.847 4 switches: flows/sec: 2913 2948 2900 2659 total = 45.093955 per ms
19:02:31.947 4 switches: flows/sec: 2516 2743 2564 2481 total = 40.403475 per ms
19:02:33.047 4 switches: flows/sec: 2702 2385 2857 2735 total = 41.222547 per ms
19:02:34.147 4 switches: flows/sec: 2468 2454 2777 2743 total = 41.657750 per ms
19:02:35.247 4 switches: flows/sec: 2525 2631 2669 2475 total = 41.662500 per ms
19:02:36.347 4 switches: flows/sec: 2735 2909 2723 2791 total = 45.806954 per ms
19:02:37.447 4 switches: flows/sec: 2932 2629 2988 2989 total = 46.304907 per ms

RESULT: 4 switches 15 tests min/max/avg/stdev = 40403.47/45093.95/42343.00/1382.95 responses/s
```

Figure 5.1: A benchmark result from Cbench

network design with the model. To achieve this, we develop a tool named OFCP to help researchers to build performance models. OFCP contains a virtual switch, which can send messages to and receive messages from an OpenFlow controller. Packet-in messages are the most frequent in an OpenFlow channel. So this tool focuses on the performance of the controller processing the packet-in messages. The tool sends packet-in message to an OpenFlow controller, receives a flow-modify message, records the round trip time, and fits the times into a hyper-Erlang distribution.

In this chapter, we discuss our OpenFlow controller performance evaluation tool. It is a tool with a graphical user interface to help users to build performance model of OpenFlow controllers. We provide a discussion of the implementation and the use of OFCP in common tasks. OFCP implements a virtual switch to measure the response time of OpenFlow controllers and a distribution fitting algorithm to fit the response time to a hyper-Erlang distribution. Our focus will be on the illustration of OFCP in typical scenarios. With the tool, users can gather the response time of OpenFlow controllers and fit the response time with a hyper-Erlang distribution. Furthermore, they can export the result into other modeling tools to build and evaluate their model.

The rest of this chapter is structured as follows. In Section 5.1 we present the implementation of OFCP. In Section 5.2 we discuss how to use the tool and present a performance evaluation result. Finally, we conclude this chapter in Section 5.3

5.1 The OFCP Tool

In this section we present the implementation of our performance evaluation tool.

5.1.1 Design Goal

The OpenFlow protocol introduce a new forwarding delay to the networks because of the communication between the switches and the controller. The controller may become a bottleneck in a large network. Many researchers have noticed this problem and built queueing model for OpenFlow networks to measure the impact of the communication. Many studies assume that the message processing time of controllers following exponential distribution [59,148]. Based on our measurements, the exponential distribution cannot fit the message processing very well. At the same time, there are no tools that offer the response time for individual messages, so we develop this tool. This tool can not only evaluate the performance of OpenFlow controllers but also provide statistical details about the response time of a controller. Researchers can use the tool to analyze the response time of an OpenFlow controller and obtain the distribution of the response time. They can use the distribution in their analytical model. One of our design objectives is to build a tool that is interactive and easy to use. The architecture of OFCP is guided by the following design goals.

- *Detailed analysis:* The main purpose of the tool is to help researchers to build their models of OpenFlow controllers. To achieve this, the mean response time and the variance is not enough. Our performance evaluation tool should provide the response time and the fitted distribution of the response time. If the users are not satisfied with the fitted result, they can also use other fitting tools. In addition, it also provides other performance metrics such as the number of outstanding packets.
- *Interaction with modeling tools:* This tool is used for building performance models of OpenFlow controllers, but we only focus on response time analysis. Users need modeling tools to build and evaluate their models. It would be helpful if this tool can interact with other modeling tools, e.g. JMT [17]. Users obtain controller response time, analyze the response time in this tool and export the result to other modeling tools to build and evaluate their models.
- *Flexibility:* By default, the tool sends a message to the controller, waits for the response, and sends the next message when a response message is received. This means that the tool can only analyze the response time of a controller. Researchers may want to control the arrival process of the messages in the performance evaluation. This is a common operation in a queueing model. In addition, the researchers may have a different demand for different topics. The tool should be adaptable to new scenarios. We want to develop a flexible tool to make it easy to

adapt to different arrival processes.

- *User-friendly*: There are other open source OpenFlow benchmark tools, such as Cbench [124], OFCBenchmark [57]. They are both command line tools and only work on a Linux platform. One of our goals is to develop a user-friendly performance evaluation tool with a graphical user interface. Users can get the performance metrics with some simple clicks.

5.1.2 Architecture

There are four main components in OFCP: virtual switch, time measurement, arrival process configuration and distribution fitting. They are illustrated in Figure 5.2.

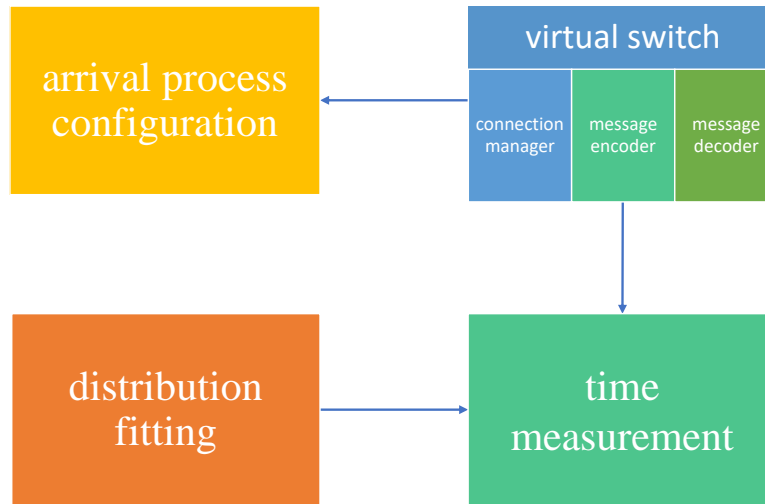


Figure 5.2: The architecture of OFCP

The key components of the OFCP are the virtual switch and the distribution fitting. Figure 5.2 shows the structure of the virtual switch. It holds a connection to an OpenFlow controller, through which it sends OpenFlow messages to the controller and receives messages from the controller. The message encoder transfers OpenFlow messages into bytes and the message decoder does the reverse. The time measurement component stores the time when an OpenFlow message is sent or received. The time measurement is triggered by the message encoder and decoder. It records a time stamp when the message encoder or decoder transfers each message. After the time measurement, the distribution fitting component fits a hyper-Erlang distribution to the response time of the OpenFlow controller.

By default, the virtual switch sends the next OpenFlow messages after it receives the response for

the previous one. Consequently the controller only processes one request at any time. The user may want to change the arrival process to meet their demands. This is done using the the arrival process configuration component. The user can define the arrival process of the OpenFlow messages in this component.

5.1.3 Implementation

OFCP is written in Java using the OpenFlowJ library [90], which exposes the OpenFlow protocol through a Java API. Experiments can be configured directly via the graphical user interface. Configuration options include measurement time, arrival process.

When a virtual switch is created, it reads the configuration, performs the OpenFlow handshake process and answers other controller requests. After the handshake is finished, an inside packet-generator creates packet-in messages and sends them to the controller. The time between two messages can be configured in the arrival process configuration component. Each packet-in message contains a packet header that the controller has not yet encountered. A packet-in message is identified by its buffer id. The controller responds to each packet-in message with a packet-out message or flow-modify message using the same buffer id to identify the corresponding packet-in message.

The time measurement component is informed that a request or response arrives by the encoder and decoder. Before a packet-in message can be sent, it should be encoded into bytes. The encoder informs time measurement component the buffer id, The time measurement component records the buffer id and the time stamp. After a packet-out or flow-modify message is received, the message decoder transfers the bytes into an OpenFlow message, it parses the buffer id and informs the time measurement component. The time measurement component gets another time stamp and calculates the response time for the request.

After the measurement is finished, the distribution fitting component gets the measured samples from the time measurement component and fits the response time to a hyper-Erlang distribution using the fitting algorithm described in Chapter 4.

The OFCP has a graphical user interface, the users can see the response time in real time. They can also see the cdf and the pdf of the distribution of the response time after the distribution fitting.

5.2 Performance Evaluation Result

In this section we demonstrate the usage of our tool. We discuss its functionality and show an example of the performance evaluation of Ryu controller.

The tool helps researchers to build a performance model of OpenFlow controllers and the network

managers to understand the behaviors of OpenFlow controllers. It measures the response time of the controller and fits a hyper-Erlang distribution the response time. The users can use the distribution to build and validate their model. So the first step is to measure the performance of OpenFlow controllers. There are two modes, latency and throughput. In latency mode, the tool measures the response of the controller, it sends packet-in messages following the given arrival process and measures the response time. In throughput mode, the tool sends as many as possible packet-in messages, and measures how many response messages it receives. the default mode is to measure the response time because the response time is more relevant for building models. The users can also configure the duration of the measurement. There are two ways to configure it, measurement time and sample number. By configuring the measurement time, the measurement will last for the given time. If a sample number is given, the tool will gather samples until it reaches the given number. The arrival process is essential in a queueing model. The users can configure the arrival rate, by which the tool will send packet-in messages following the given arrival rate. For now, the tool only supports Poisson process. By default, after the response for the previous one is received. It makes the controller only process one message at any time. All the configurations can be set on the right panel of the tool. The detailed configuration are shown in Table 5.1.

Table 5.1: The detailed configuration

Parameter	Default	Comment
IP address	127.0.0.1	IP address of the controller
Port	6633	TCP port of the controller
Mode	Latency	The measurement mode (latency or throughput)
Duration	30	How long the measurement lasts (in seconds)
Sample number	10000	How many samples to gather
Arrival mode	Once a time	The mode of packet-in arrival at the controller
Arrival rate	500	The arrival rate of Poisson process

We set up a Ryu controller and use the tool to measure the response time of Ryu. The Ryu controller runs on ubuntu 18.04 with 4G memory and 2.3GHz CPU. There is a Ryu application for benchmark in the Ryu repository. We modify the benchmark application to make it send a flow-modify message with a buffer id. Then we run it and measure the response time. The measurement process is shown in Figure 5.3. OFCP sends a packet-in message to the Ryu controller and wait for a flow-modify message from the Ryu controller and records the response time. The next packet-in message is sent when OFCP receives a flow-modify message. After this process runs for 10 seconds, OFCP stops sending packet-in message and fit a hyper-Erlang distribution to the response time. The result is shown in Figure 5.4.

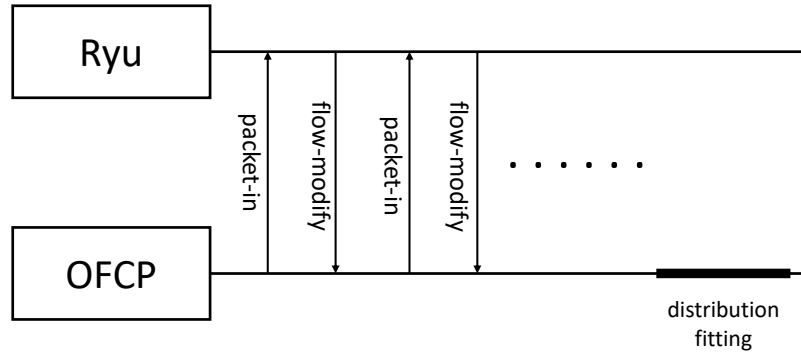


Figure 5.3: The measurement process

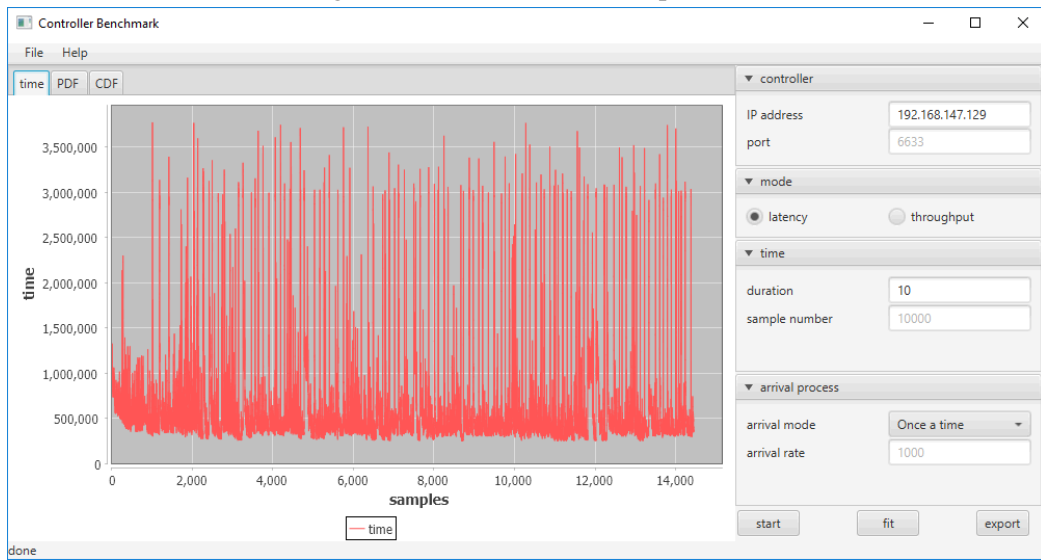


Figure 5.4: The response time of Ryu controller

We can see the configuration on the right of Figure 5.4. We use the latency mode to measure the response time of the controller, and the measurement runs for 10 seconds. The response time of the Ryu controller is shown on the left of Figure 5.4. The x-axis shows the number of the gathered samples, and the y-axis shows the response time in nanosecond. The line chart is updated in real time when the measurement is running.

After the measurement completes, we fit the distribution to the measured response time. The pdf of the fitted distribution is shown in Figure 5.5. It fits a hyper-Erlang distribution to the samples. We can see in the figure that there is a long tail in the distribution, which happens often in real world systems. As illustrated in the figure, the fitted distribution captures the peaks and valleys of the

histogram. It even captures the little peak in the long tail. The fitted distribution is a hyper-Erlang distribution with 6 Erlang branches. The parameters of the Erlang branches are shown in Table 5.2

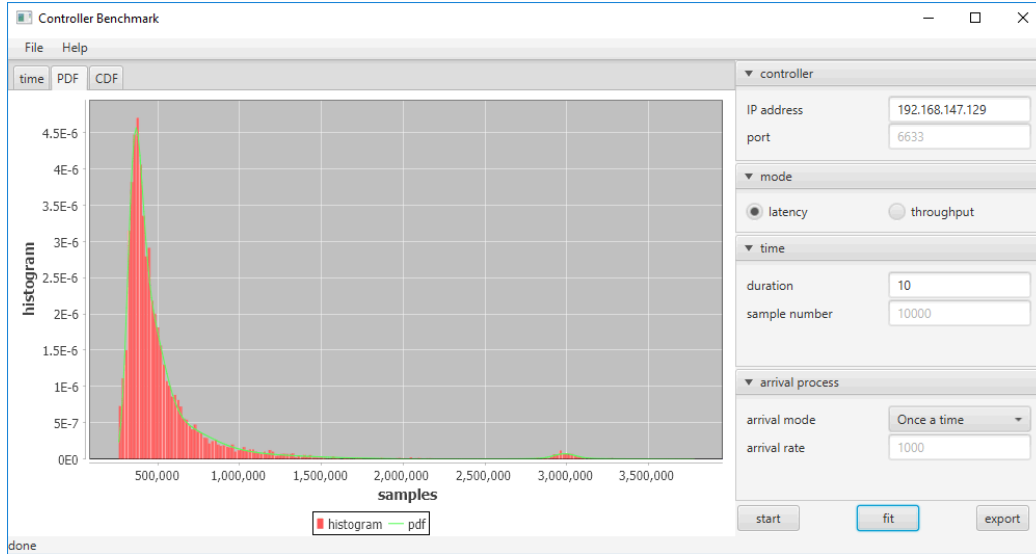


Figure 5.5: The pdf of fitted distribution

Table 5.2: The parameters of the Erlang branches

Probability	Phase	Rate
0.349	134	$3.16E - 4$
0.01	590	$2.01E - 4$
0.203	48	$7.99E - 5$
0.051	11	$7.54E - 6$
0.105	25	$2.97E - 5$
0.279	87	$1.73E - 4$

5.3 Summary

An OpenFlow controller is an essential component in an OpenFlow network. It is key to understanding the performance for researchers and managers of productive networks. In this chapter, we have introduced our tool to evaluate the performance of OpenFlow controllers. It provides users not only with the performance metrics but also fits a distribution to the response time of the OpenFlow controller. Through the distribution of response time, users can understand the underlying reason for

the controllers' behaviors. We also present a cluster-based fitting algorithm that fits a hyper-Erlang distribution the samples. The fitting result shows that the algorithm can fit the response time of OpenFlow well. Since hyper-Erlang distributions have Markovian representation, the fitted result can be easily used in analytical and simulation approaches to performance evaluation.

Part III

Performance of the Control Plane

Chapter 6

The Performance of Multiple controllers

The growth of network traffic has been immense over the last years. The latest Cisco Visual Networking Index [33] reported that the annual run rate for global IP traffic was 1.2ZB in 2016 and the global IP traffic will increase nearly threefold from 2016 to 2021, the annual global IP traffic will reach 3.3ZB per year by 2021. As a consequence handling the increasing traffic effectively and providing high quality service becomes an important challenge. OpenFlow is considered a promising way and has attracted a lot of attention from both researchers and industry. OpenFlow offers the high flexibility by separating the control plane from the data plane and using a centralized controller.

The communication between the controller and the infrastructure increases the transmission latency. With the emerging big data, the overhead will be worse than ever, since the massive traffic in the data plane triggers a lot of requests on the OpenFlow channel. It is a significant challenge for the capability of the OpenFlow controller. Multiple controllers have to be deployed in one network if a single controller is not capable of handling the massive traffic. Many researchers have noticed the shortages of a single controller. They studied the performance of multiple OpenFlow controllers by analytical modeling and experiments [3, 138, 143, 147].

In this chapter, we use a queueing model to evaluate the performance of multiple controllers. We measure the response time of an OpenFlow controller using OFCP and determine the optimal number of controller with different rates of synchronization messages and packet-in messages. We also proposed a heuristic to study on the controller assignment problem. We model each controller as an $M/PH/1$ queue to capture its response time and optimize the assignment between the switches and controller to minimize the response time. We evaluate our solution in Mininet.

6.1 The Number of Controllers

To reduce the flow setup time, multiple controllers are deployed into one network and the controllers manage network flows cooperatively [50, 134]. Nevertheless, the multiple controllers introduce a new overhead, the communication among the controllers. On one hand, more controllers provide more computing resources. On the other hand, more controllers incur more communication overhead. Therefore, we have to determine the optimal number of controllers. In this section, we propose a queueing model to evaluate the service time and communication overhead of multiple controllers and determine the number of controllers that can minimize the flow setup time in OpenFlow networks. Our solution takes into account the random arrival process and service time. We formulate the performance of a controller based on queueing theory and determine the optimal number of controllers based on the response time of the queue.

6.1.1 System Description

We give an overview of the system before introducing the performance model. In this section, we consider a large OpenFlow network. There are two components in the network: the OpenFlow switches and the controllers. In OpenFlow networks, all forwarding decisions are made by the controller. The switches are forwarding devices, which can only forward packets following the instructions from the controller. The controller is usually implemented as a piece of software. The controller can add, delete and modify flow entries into OpenFlow switches proactively and reactively. An OpenFlow switch contains one or several flow tables, which stores a set of flow entries. A flow entry consists of match fields, a set of instructions and counters. When a packet arrives at an OpenFlow switch, the switch tries to find a flow entry that matches the header of the packet. If a flow entry is found, the instructions in the flow entry will be executed. The packet will be forwarded, modified or dropped. Otherwise, the switch will send a packet-in message to the controller. The controller may install a flow entry into the switch, so that the packets in the same flow need not be forwarded to the controller.

Since the network is assumed to be very large, a single controller is often not capable of all the requests to all the switches. The switches are split into domains, each switch can be only in one domain. For each domain, there is a controller to manage the switches. Therefore one switch can be only controlled by one controller.

Since there are no controllers that manage all the switches in the network, a controller can not collect the network information of the whole network. The controllers have to exchange their information about the network. This communication causes additional overhead. The more controllers

here are in a network, the higher is the overhead. Therefore, a controller receives less of the volume of the requests from switches with increasing number of controllers. At the same time it spends more time on communication with other controllers.

When a packet arrives at a switch, the switch checks its flow table to find a matching flow entry. If a match is found, the packet will be forwarded according to the forwarding rule. If no matches can be found, the switch will send a packet-in message to the controller which contains the header of the packet. The controller computes a path and installs the necessary flow entries into the switches along the path. There are two possible cases in the flows installation: (1) the path is inside one domain, (2) the path across more than one domain. In the first case, all the switches along the path are in one domain and controlled by one switch. Then the switch installs flow entries into all the involved switches. However, in the second case, the switches along the path are not controlled by one controller and the controller can only install flow entries into the switches in its domain. The controller must invoke other controllers to install the flow entries into the switches out side its domain. In this case, the controller will spend more time to install flows. Therefore, there are two kinds of jobs for a controller, and the controller will serve them at different rates.

The first case is shown in Figure 6.1.a. A packet arrives at *switch 1*, and there are no forwarding rules for it. *Switch 1* sends a packet-in message to *controller 1*. The controller computes a path *Switch 1* \rightarrow *Switch 2*. As all switches along the path are in *domain 1*, the controller installs flow entries into all the involved switches.

The second case is shown in Figure 6.1.b. When a packet arrives at *switch 1* and its destination is at *switch 4*, *switch 1* can not find a match for the packet and sends a packet-in message to *controller 1*. *Controller 1* computes a path for the packet, the path is *switch 1* \rightarrow *switch 2* \rightarrow *switch 3* \rightarrow *switch 4*. *Switch 1* and *Switch 2* are in the same domain and controlled by *controller 1*, *controller 1* installs flow entries into *Switch 1* and *Switch 2*. The flow setup is not completed, because the path also contains *Switch 3* and *Switch 4*, which are controlled by *controller 2*. Therefore, *controller 1* cannot install flow entries into them and it has to invoke *controller 2* and *controller 2* installs flow entries into *Switch 3* and *Switch 4*, so that *Switch 3* will not send packet-in to *controller 2* when the packet arrives at *Switch 3*. A controller must should process the packet-in messages from the switches in its domain. Meanwhile, it also should process the invocations from other controllers.

6.1.2 Problem Formulation

In this section, we consider a queueing model for the controllers to capture the flow setup time. The controllers are modeled as servers in the queueing model. We approximated the service time of the controllers using PH distributions.

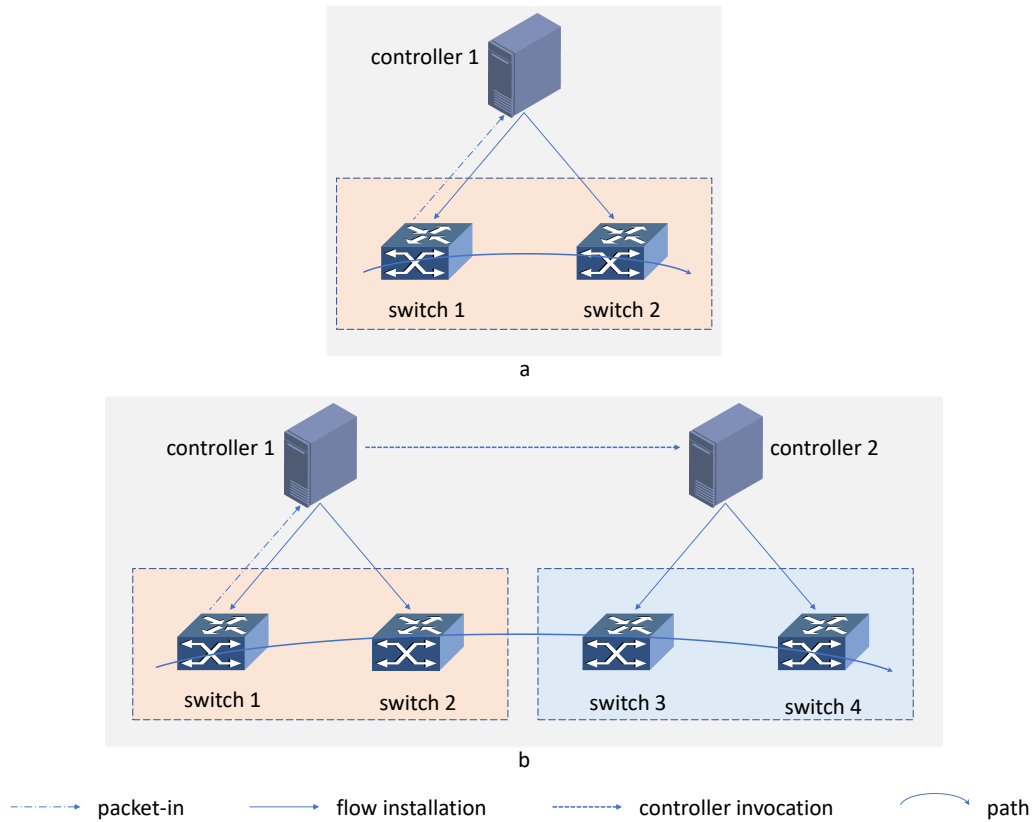


Figure 6.1: Flow setup process under multiple controllers

As we introduced in Section 6.1.1, there are two kinds of jobs for a controller. A controller has two different service rates for them. If the controller can complete the flow installation without involving other controllers, the service time is termed as single controller response time. If the controller invokes other controllers to complete the flow installation, the service time is termed as multiple controllers response time. The single controller response time and multiple controllers response time are denoted as t_1 and t_2 . We approximate both t_1 and t_2 by PH distributions. In the invocation process, a controller receives invocations and installs flow entries into switches. It does not need to invoke other controllers, so it will spend time t_1 to complete the flow installation.

For simplicity, we apply the following reasonable assumptions. The packet-in messages arrive at each switch at the same rate, and the destinations of flows are uniformly distributed. We also assume that the controllers are powerful enough to handle all the requests from the network that the performance of the controllers is homogeneous, and the controllers have unlimited capacity to store the requests in the queue.

Network traffic measurements have indicated that flow arrival in packet switching networks follow Poisson process [146]. We assume that the packet-in messages arriving at controller follow a Poisson process. We denote the total arrival rate of packet-in messages for all controllers λ_p .

If there are n controllers in the network, the packet-in messages arrive on average at each switch at rate λ_p/n , and the probability that a controller can not finish the flow installation alone is $(n-1)/n$. For simplicity, we assume that the invocations from other controllers also follow a Poisson process. We denote the arrival rate of invocations λ_i . If we tag one controller, there are $n-1$ controllers send invocations to the tagged one, and each of them sends invocations at rate of $\lambda_p/(n-1)$. We have

$$\lambda_i = \frac{(n-1)\lambda_p}{n^2}. \quad (6.1)$$

The controllers send and receive synchronization messages in order to maintain a consistent network-wide view in all controllers. We assume critical events that make the controllers synchronize the state follow a Poisson process, and the arrival rate at each controller is λ_s . For a controller, the arrival rate of the message exchange is $(n-1)\lambda_s$.

The controller spends time $t_1 \sim PH(\alpha_1, T_1)$ on a request, if the controller can handle the request alone. Otherwise, the controller spends time $t_2 \sim PH(\alpha_2, T_2)$. The queueing system is illustrated in Figure 6.2. The arrival rate of message exchange is $(n-1)\lambda_s$, the arrival rate of packet-in messages is λ_p/n , and the arrival rate of invocations is λ_i . The controller spends time t_1 with probability γ_1 , spends time t_2 with probability γ_2 .

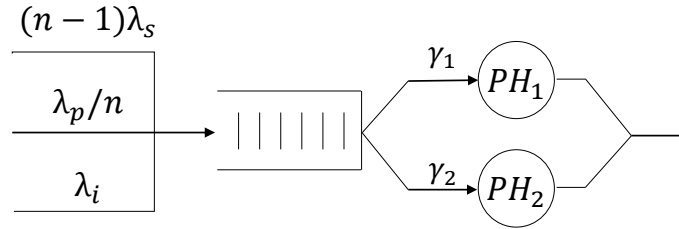


Figure 6.2: The queueing model for flow installation

For the jobs from the switches, a controller spends time t_1 with probability $1/n$, and it spends time t_2 with probability $(n-1)/n$. For the message exchange, a controller always spends time t_1 . The probability that a switch spends time t_1 on a job is γ_1 , and probability that a switch spends time t_2 on a job is $\gamma_2 = 1 - \gamma_1$. We have

$$\gamma_1 = \frac{n^2(n-1)\lambda_s + n\lambda_p}{n^2(n-1)\lambda_s + n\lambda_p + (n-1)\lambda_p}. \quad (6.2)$$

6.1.3 Derivation of the Analytical Model

The combination of Poisson processes is still a Poisson process, the arrival rate can be denoted as

$$\lambda = (n-1)\lambda + \lambda_i + \frac{\lambda_p}{n}. \quad (6.3)$$

The distribution of the service time is the combination of $PH(\alpha_1, \mathbf{T}_1)$ and $PH(\alpha_2, \mathbf{T}_2)$ with probability γ_1 and γ_2 . We denote the distribution of the service time $PH(\alpha, \mathbf{T})$. We have

$$\alpha = (\gamma_1\alpha_1, \gamma_2\alpha_2), \quad (6.4)$$

$$\mathbf{T} = \begin{pmatrix} \mathbf{T}_1 & 0 \\ 0 & \mathbf{T}_2 \end{pmatrix}. \quad (6.5)$$

Therefore, we can use an $M/PH/1$ queue to describe the flow installation process with arrival rate λ and service time $t \sim PH(\alpha, \mathbf{T})$. The mean service time is $1/\mu = \alpha(-\mathbf{T})^{-1}\mathbf{e}$. We denote the server utilization $\rho = \lambda/\mu$. The probability that there are no jobs in the system is $x_0 = 1 - \rho$. We can obtain the average flow setup time $E[S]$ as

$$\begin{aligned} E[S] &= \frac{E[N]}{\lambda} \\ &= \frac{x_0\alpha\mathbf{R}(\mathbf{I} - \mathbf{R})^{-2}\mathbf{1}}{\lambda}. \end{aligned} \quad (6.6)$$

The optimal number of controller can be obtained by optimizing the following expression.

$$n = \arg \min_n E[S]. \quad (6.7)$$

We use PSO [66] implemented in PySwarms [87] to get the optimal number of controllers. The fitness function is the mean response of the controllers the parameter to be optimized is the number of controllers.

6.1.4 Evaluation

In this section, we introduce the measurement of the flow setup time of our prototype, and evaluate the queuing model with different parameters by queuing analysis.

Measurement of Response Time

We also implement a prototype of multiple controllers based on the Ryu framework. The architecture of the prototype is shown in Figure 6.3. The controller applications communicate with each other via a centralized message queue. Each controller listens to a topic. a controller invokes another controller by sending messages to its related topic.

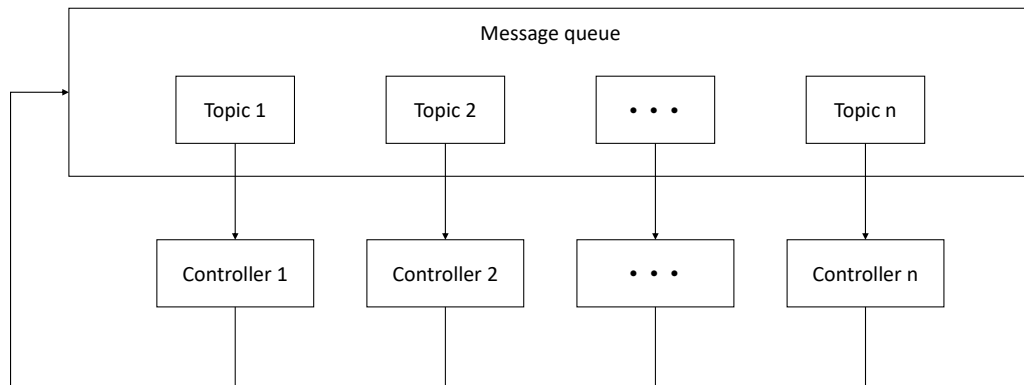


Figure 6.3: The architecture of the prototype

We use the tool introduced in Chapter 5 to measure the response time, and fit the response time to a hyper-Erlang distribution. We run the controller application on ubuntu 18.04 with 4G memory and 2.3GHz CPU. The single controller response time and fitted pdf is shown in Figure 6.4. The parameters of the fitted hyper-Erlang distribution are shown in Table 6.1. The multiple controllers response time and fitted pdf is shown in Figure 6.5. The parameters of the fitted hyper-Erlang distribution are shown in Table 6.2.

Table 6.1: Distribution of single controller response time

Probability	Erlang distribution
0.306983	Er(667, 0.001657)
0.053984	Er(209, 0.000354)
0.096460	Er(381, 0.000741)
0.226080	Er(421, 0.000946)
0.030528	Er(24, 0.000035)
0.285965	Er(1669, 0.003911)

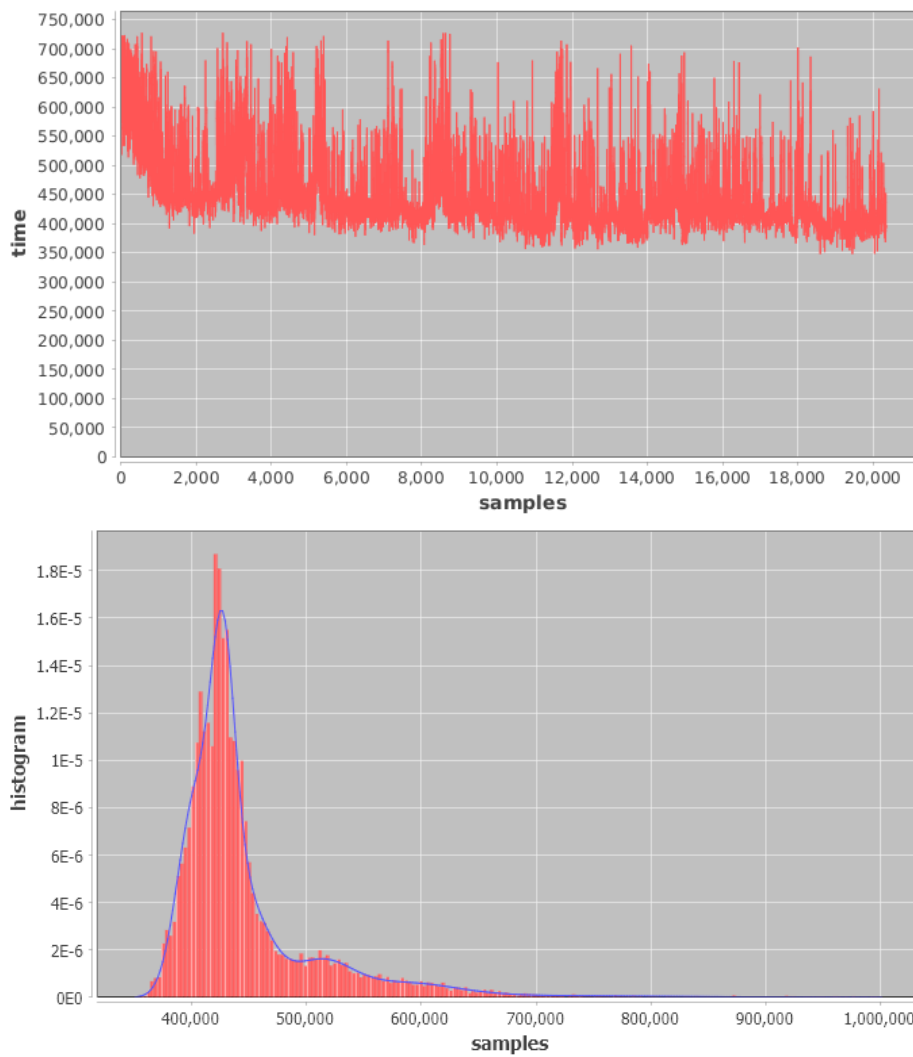


Figure 6.4: The single controller response time

Table 6.2: Distribution of multiple controllers response time

Probability	Erlang distribution
0.277380	Er(1544, 0.002967)
0.212247	Er(744, 0.001343)
0.311721	Er(485, 0.000995)
0.114605	Er(263, 0.000425)
0.055441	Er(135, 0.000188)
0.028607	Er(47, 0.000053)

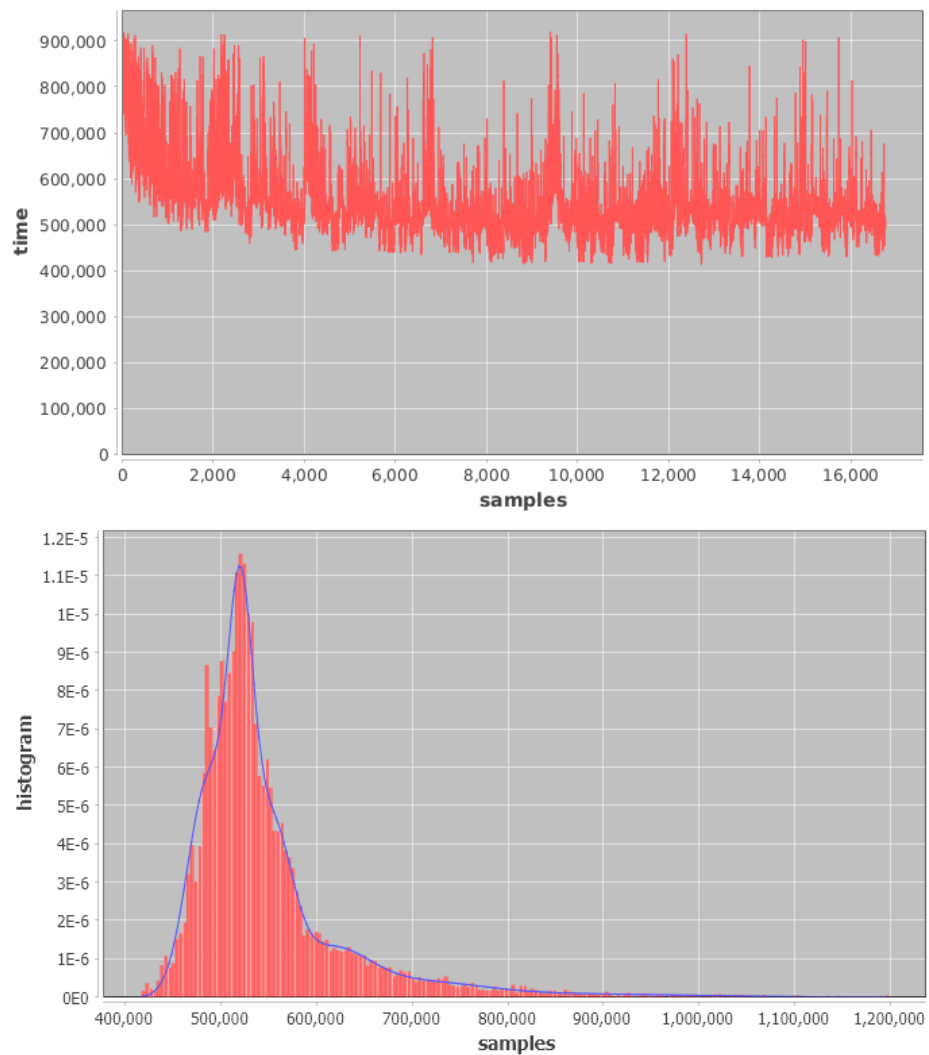


Figure 6.5: The multiple controllers response time

Queueing Analysis

To capture the performance of the multiple controllers, we evaluate the proposed queueing model with different arrival rates of packet-in messages and synchronization messages.

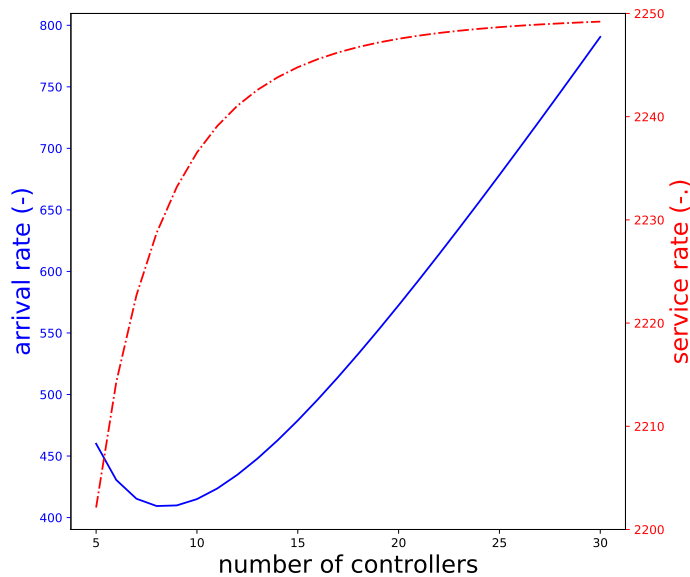


Figure 6.6: Arrival rate and service rate

We assume that all the controllers send synchronization messages at rate of 25 per second, the switches send packet-in messages at rate of 1000 per second, and all switches get the same number of packet-in messages. Figure 6.6 shows the arrival rate and service rate of a controller with different numbers of controllers.

The arrival rate decreases with the number of switches before 9 and increases with the number of switches after 9. Additional controllers will share the load of packet-in messages and come with new synchronization messages. When the number of controllers is small, additional switches can share a large volume of packet-in messages and bring a few additional synchronization messages. When the number of controllers is large, additional switches share a small volume of packet-in messages and bring a large number of synchronization messages. Therefore, the arrival rate decreases with the number of controllers before a certain point and increases with the number of controllers after that point. The service rate always increases with the number of switches, the rate of ascent is slower and slower, because the probability of a controller sending invocations decreases with the number of controllers.

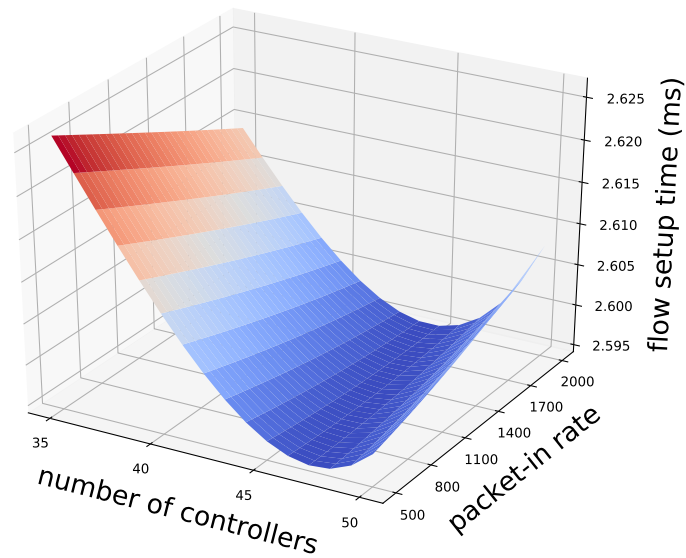


Figure 6.7: Flow setup time with different packet-in rate

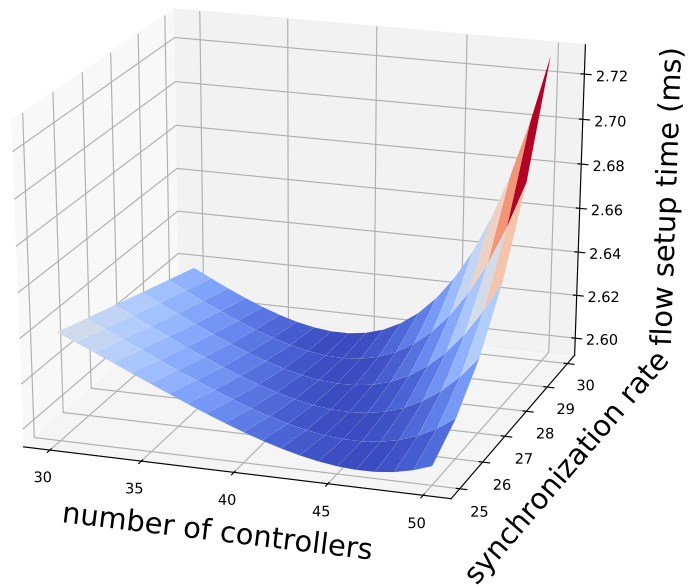


Figure 6.8: Flow setup time with different synchronization rate

Figure 6.7 shows the flow setup time with different number of controllers and arrival rate of packet-in messages. The arrival rate of synchronization messages is fixed at 25 per second, the arrival rate of packet-in is in the range of [500, 2000] per second, and the number of controllers is in

the range of [35, 51].

We can see that the flow setup time decreases with the number of controllers at first, and increases after a certain number. Because the service rate decreases slower and slower with the number of controllers, the arrival rate decreases before the inflection point and increases with the number of controllers at linear speed after the inflection point. If there are enough controllers to process the packet-in messages, the extra controllers will decrease the flow setup time because of the communication overhead.

Figure 6.8 shows the flow setup time with different number of controllers and arrival rate of synchronization messages. The arrival rate of packet-in messages is fixed at 1000 per second. the arrival rate of synchronization messages is in the range of [25, 31] per second, and the number of controllers is in the range of [30, 50]. We can see that the flow setup time is similar to Figure 6.7. The flow setup time increases with the number of controllers at first, and decreases after a certain number. As illustrated in the figure, the flow setup time increases very rapidly with the rate of the synchronization messages when the number of controllers is large. This can be explained with the fact that the synchronization messages cause high overhead when there are a large number of controllers.

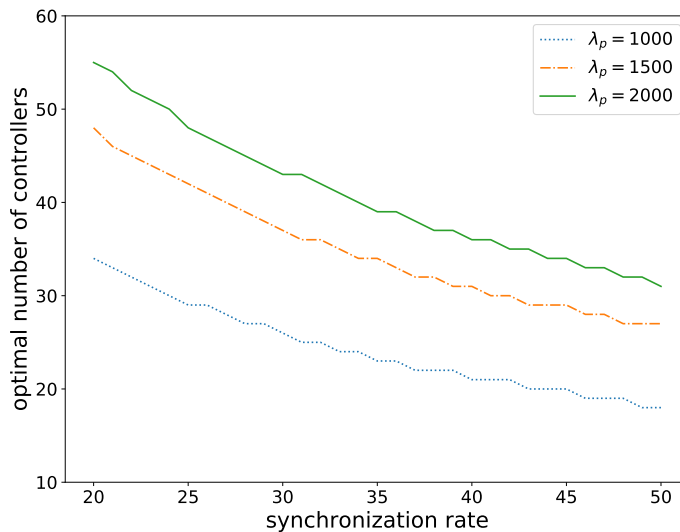


Figure 6.9: The optimal number of controllers

Figure 6.9 shows the optimal number of controllers that achieve the minimum flow setup time with different rates of packet-in messages and synchronization messages. We can see that the optimal number of controllers decreases with the rate of synchronization messages and increases with the

rate of packet-in messages. The controllers must process the requests from the switches, so the more packet-in messages there are, the more controllers we need. However, multiple controllers need inevitable synchronization messages. The more synchronization messages are sent the lower is the optimal number of controllers.

6.2 Balancing the load of Controllers

When multiple controllers are deployed in an OpenFlow network, the assignment between switches and controllers impacts the performance of the network. The flow setup time can be minimized if the load of controllers is balanced. The variance of the utilization of the controllers is usually used as the metrics to determine whether the load of controllers is balanced.

There are two significant issues in a network with multiple controllers, the assignment problem and the placement problem. In this section, we only focus on the assignment problem that determines the mapping between controllers and switches to achieve a low flow setup time. Some studies of the controller assignment problem exist [38, 157]. Most of them consider static assignment switches to controllers. However, the traffic load in a network changes frequently. The static assignment of controllers and switches may cause load imbalance among controllers. As a result, the switches that connect to overloaded controllers suffer from a high flow setup time and some controllers even cannot be fully utilized. A static controller assignment is not suitable for variable traffic. Therefore, it is important to study controller load balancing for low flow setup time. We investigate the controller assignment problem aiming at minimizing flow setup time. We measure the service time of controllers and estimate the flow setup time using a queueing model. Then we formulate the controller assignment problem as an optimization problem to minimize the flow setup time.

6.2.1 Problem Formulation

System Overview

We consider a large OpenFlow network. There are two components in the network: the OpenFlow switches and the controllers. A switch may connect to one or more controllers but only has one master controller, as shown in Figure 6.10.

Since there are multiple controllers in the network, the switches should be split into some domains and each domain is assigned to one controller. Each switch can be only in one domain. For each domain, there is a controller to manage the switches. Therefore, one switch can be only controlled by one controller. Given a set of switches, the controller load balancing problem is to divide the switches into a collection of mutually exclusive and collectively exhaustive subsets and assign each subset to a controller so that the average flow setup time is as low as possible.

The initial controller assignment can be solved as controller placement problem. However, the controller placement problem does not consider the dynamic load of networks. Once the assignment

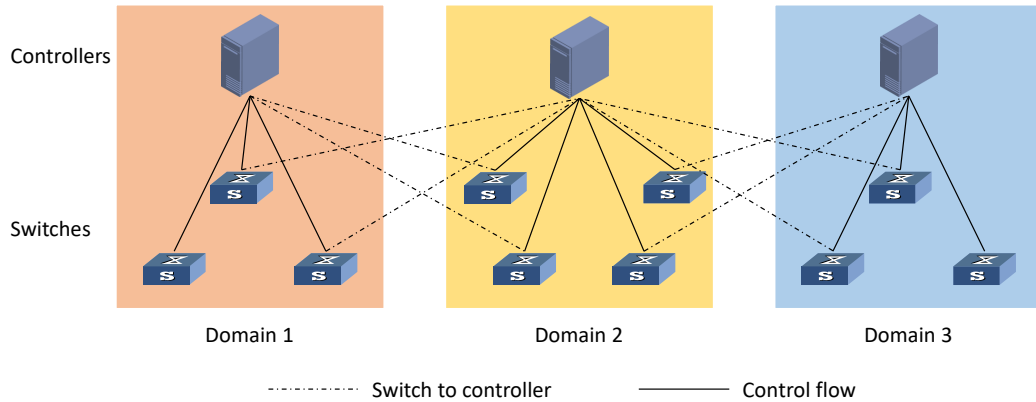


Figure 6.10: OpenFlow with multiple controllers

and positions are chosen, they are constant. Each controller manages the switches in its domain. A real network may have significant variations in traffic characteristics. Therefore, the static assignment may face some problems when the traffic load changes. For example, when a switch receives burst flows, its controller will receive a lot of flow requests in a short time. An overloaded controller may degrade the performance of the whole network. To avoid this, the controller assignment should be changed with the traffic in the network. The controller assignment problem is to find an assignment between the switches and controllers that can minimize the flow setup time and makes the least number of migrations.

Switch Assignment to Controllers

We assume that flow requests arrive at a controller following a Poisson process. We use a PH distribution to describe the response time of a controller, since PH distributions are a very flexible class of distributions for performance modeling. As PH distributions have Markovian representation, they can easily be used in analytical and simulation approaches for performance evaluation. We consider an $M/PH/1$ queueing model for the controllers to capture the performance of a controller. The controllers are modeled as servers in a queueing model. The requests from switches under each controller are the jobs for the controllers. The queueing model for a controller is depicted in Figure 6.11. There are m switches managed by a controller. Each switch sends requests to the controller following a Poisson process. The i -th switch sends requests to the controller at rate λ_i . We denote the arrival rate at a controller as $\lambda = \sum_{i=1}^m \lambda_i$. The service time of the controller follows a PH distribution (α, \mathbf{T}) .

In a distributed architecture of the control plane within an OpenFlow network, the control plane

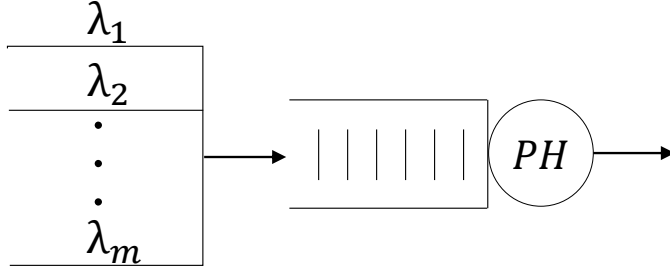


Figure 6.11: Queuing model of a controller

consists of a set of controllers. The set of controllers is denoted $C = \{c_i | i = 1, 2, \dots, n\}$. The capacity of controller c_i is μ_i , which is the maximum number of requests controller c_i can process per second. The average load of controller c_i is λ_{c_i} , which is the number of requests controller c_i receives per second. The data plane consists of a set of switches. The set of switches is denoted by $S = \{s_j | j = 1, 2, \dots, m\}$. We assume the rate of new flows arrival at switch s_j is denoted by λ_{s_j} , and the switch sends requests to its controller at the same rate.

The assignment among controllers and switches is denoted by an $n \times m$ matrix \mathbf{A} . The assignment matrix \mathbf{A} is binary. $A_{ij} = 1$ if controller c_i manages switch s_j . Otherwise, $A_{ij} = 0$. A switch can connect to one or more controllers at the same time, but only one controller is its master controller during each time interval. Therefore, the sum of the elements in a column of \mathbf{A} is 1.

We consider two delays that will be incurred when multiple controllers are deployed in a network: the flow setup time and the switch migration time.

1) Flow setup time. There are multiple controllers in a large network. Each controller manages a subset of switches in the network. The rate of packet-in messages arriving at controller c_i is

$$\lambda_{c_i} = \sum_{j=1}^m A_{ij} s_j. \quad (6.8)$$

The distribution of the flow setup time of controller c_i is $PH(\alpha_i, T_i)$. The utilization of controller c_i is ρ_{c_i} . We can obtain the average flow setup time of controller c_i :

$$E[S_{c_i}] = \frac{\alpha_i \mathbf{R}_i (1 - \rho_i) (\mathbf{I} - \mathbf{R}_i)^{-2} \mathbf{1}}{\lambda_{c_i}}. \quad (6.9)$$

The average response time of all controllers is

$$E[S] = \sum_{j=1}^m \frac{\lambda_j}{\lambda} E[S_j]. \quad (6.10)$$

2) Switch migration time. Dynamic changes in traffic load may disturb the load balancing among controllers. The switches should be migrated with the traffic load to improve the performance of the network. Denote the time cost of migrating a switch from one controller to another as σ . Assume that the previous assignment matrix is A_p , and the current assignment matrix is A_c . We define a matrix B as the XOR between A_p and A_c . $B_{ij} = 1$ if the switch s_j is migrated out of or into controller c_i , Otherwise, $B_{ij} = 0$. The number of switch migrations is

$$M = \frac{\sum_{i=1}^n \sum_{j=1}^m B_{ij}}{2}. \quad (6.11)$$

Therefore, the switch migration time is

$$\Delta = M\sigma. \quad (6.12)$$

The controller assignment problem can be formulated as an optimization problem. The objective is to find an assignment that minimizes the weighted sum of the flow setup time and switch migration time.

$$\text{minimize } w_1 E[S] + (1 - w_1)\Delta, \quad (6.13)$$

where w_1 is constant. The network operator can use w_1 to adjust the relative significance of the two costs. Furthermore, the following constraints must be satisfied.

$$\sum_{i=1}^n A_{ij} = 1, \quad (6.14)$$

$$\lambda_j < \mu_j, \quad (6.15)$$

$$A_{ij} \in \{0, 1\}, \quad (6.16)$$

$$\forall i \in \{1, 2, \dots, n\}, \forall j \in \{1, 2, \dots, m\}, . \quad (6.17)$$

Constraint (6.14) assures that each switch must have only one master controller. Constraint (6.15)

assures that the number of requests each controller processes cannot exceed its capacity. Constraint (6.16) and (6.17) are numerical constraints.

6.2.2 Controller Assignment Algorithm

We describe an algorithm for solving the dynamic controller assignment problem. In order to improve the performance of a network, the assignment among switches and controllers must be adjusted with the state of the network since the traffic in a network may change quickly.

The controller assignment problem is similar to the number partitioning problem, which is known to be NP-complete. Here, the set of numbers consists of the flow arrival rates of the switches. We divide the switches into multiple subsets and assign a controller to each subset. The controller assignment problem should avoid too many migrations when the traffic load of switches changes. To obtain the optimal solution, we design a heuristic for solving the controller assignment problem named LANS (Late Acceptance Neighbor Search). It accepts the current controller assignment matrix $\tilde{\mathbf{A}}$ and provides a new controller assignment matrix \mathbf{A} that can minimize the weighted sum of the flow setup time and switch migration time.

Initial Solution

Due to the changes of traffic, the current controller assignment matrix $\tilde{\mathbf{A}}$ may not satisfy all constraints. We design a greedy algorithm that generates a feasible controller assignment matrix. The output of this algorithm is the input of LANS.

More specifically, the greedy algorithm tries to balance the load of the controllers with the least number of migrations. It migrates switches from the most loaded controller to the least loaded controllers. It sorts the controllers by their utilizations. Let ρ_{min} denote the minimum utilization of all controllers, the target controller that accepts the migrated switches is selected randomly among the controllers whose utilization is less than $1.1\rho_{min}$. After the source controller and target controller are selected, the algorithm tries to balance the load between them in one migration. The pseudocode

of the greedy algorithm is shown in Algorithm 1.

Algorithm 1: Generate initial solution

Input: set of switches, S

set of controllers, C

previous assignment matrix, \tilde{A}

Output: new assignment matrix, A

```

1 while true do
2    $M \leftarrow$  most loaded controller in  $\tilde{A}$ ;
3    $L \leftarrow$  target controller; ;
4    $d \leftarrow \frac{\lambda_M \mu_L - \lambda_L \mu_M}{\mu_L - \mu_M}$ ;
5    $S_s \leftarrow$  null;
6   if  $\mu_M > \mu_L$  then
7      $S_g \leftarrow$  switches in  $M$  with arrival rate greater than  $d$ ;
8     if  $S_g \neq \emptyset$  then
9        $S_s \leftarrow$  switch with the lowest arrival rate in  $S_g$ ;
10    end
11  else
12     $S_l \leftarrow$  switches in  $M$  with arrival rate less than  $d$ ;
13    if  $S_l \neq \emptyset$  then
14       $S_s \leftarrow$  switch with the most arrival rate in  $S_l$ ;
15    end
16  end
17  if  $S_l \neq \emptyset$  then
18    break;
19  end
20  migrate  $S_s$  from  $M$  to  $L$ ;
21  update  $\tilde{A}$ ;
22   $A \leftarrow \tilde{A}$ ;
23 end

```

In line 3, the target controller is the least loaded controller. We calculate the difference of the load between the most loaded controller and the target controller based on their service rate in line 4. If the arrival rate of a switch equals to d and we migrate the switch, the two controllers will have the same utilization. From line 6 to line 16, we are trying to balance the load of two controllers with one migration. We collect all the switches meet the requirement and migrate the switch whose

arrival rate is closest to d . After a migration, the most loaded controller may change. Then, we try reduce the load of the most loaded controller, until any migrations will cause the load of the target controller more than the most loaded controller. After these migrations, any migration will lead the least loaded controller to the most loaded controller. This algorithm is greedy. It tries best to make the two controllers balanced for every iteration.

Algorithm 2: LANS

Input: initial assignment matrix, $\tilde{\mathbf{A}}$

Output: new assignment matrix, \mathbf{A}

```
1 calculate initial cost function  $C(\tilde{\mathbf{A}})$ ;
2 specify  $L_h$ ;
3 for all  $k \in \{1, 2, \dots, L_h\} f_k = C(\tilde{\mathbf{A}})$ ;
4 first iteration  $N \leftarrow 0, N_{idle} \leftarrow 0$ ;
5 while true do
6   | construct a candidate solution  $\mathbf{A}$ ;
7   | calculate a candidate cost function  $C(\mathbf{A})$ ;
8   | if  $C(\mathbf{A}) > C(\tilde{\mathbf{A}})$  then
9   |   |  $N_{idle} \leftarrow N_{idle} + 1$ ;
10  | else
11  |   |  $N_{idle} \leftarrow 0$ ;
12  | end
13  | calculate the virtual beginning  $v \leftarrow N \bmod L_h$ ;
14  | if  $C(\mathbf{A}) < C(\tilde{\mathbf{A}})$  or  $C(\mathbf{A}) < f_v$  then
15  |   | accept the candidate  $C(\tilde{\mathbf{A}}) \leftarrow C(\mathbf{A})$ ;
16  | else
17  |   | reject the candidate;
18  | end
19  | if  $C(\mathbf{A}) < f_v$  then
20  |   | update the fitness array  $f_v \leftarrow C(\tilde{\mathbf{A}})$ ;
21  | end
22  | if  $N > \text{minimum iterations}$  and  $N_{idle} > N * 0.02$  then
23  |   | break;
24  | end
25  | increase the iteration number  $N \leftarrow N + 1$ ;
26 end
```

LANS

LANS starts from the output of the heuristic and further optimize the assignment by searching for neighbors iteratively. The idea of LANS is that the acceptance condition is taken from the search history. LANS maintains a list of fixed length that saves the previous values of the cost function. The cost of a candidate is compared with the last element in the list and if it has a lower cost, the candidate is accepted. The list is updated when an acceptance happens. The pseudocode of LANS is shown in Algorithm 2.

In line 2, L_h is the length of the list that saves the previous values. We define the following moves to generate a candidate.

- Relocation: select a random switch and migrate it to a random controller.
- Exchange: select two switches randomly from two different controllers and swap their assignments.

Using one of these moves, we construct a candidate in line 6. The moves are chosen randomly. We determine whether the new candidate is accepted in line 14. If the new candidate has lower cost than its previous candidate or its virtual beginning, it is accept. Then we can construct the next candidate from it. If the new candidate has lower cost than its virtual beginning, we update the cost of its virtual beginning in line 20.

6.2.3 Evaluation

In this section, we introduce the measurement of the flow setup time of our prototype, and evaluate the performance of different controller assignment schemes.

In our experiments, we setup a network with 32 switches with Mininet. We generate flows between the end hosts. The source and destination of each flow are chosen randomly. The distribution of controller response time is measured in Chapter 5. We run the experiments under two types of traffic. In case 1, the flow arrival rate follows a real data set from CAIDA [23]. In case 2, the flow arrival rate follows the characteristics of traffic described in the work of Gebert [43] The generated flows span 24 hours. The traffic we generate is shown in Figure 6.12 and 6.13.

In order to evaluate the performance of LANS, we compare LANS with the dynamic controller assignment algorithm DCP-SA in the work of Bari [9] in terms of the flow setup time, migrations and CV (Coefficient of Variation) of utilization of the controllers. DCP-SA uses simulated annealing algorithm to optimize the controller assignment. The simulation runs for 24 hours and both algorithms are executed every 5 minutes.

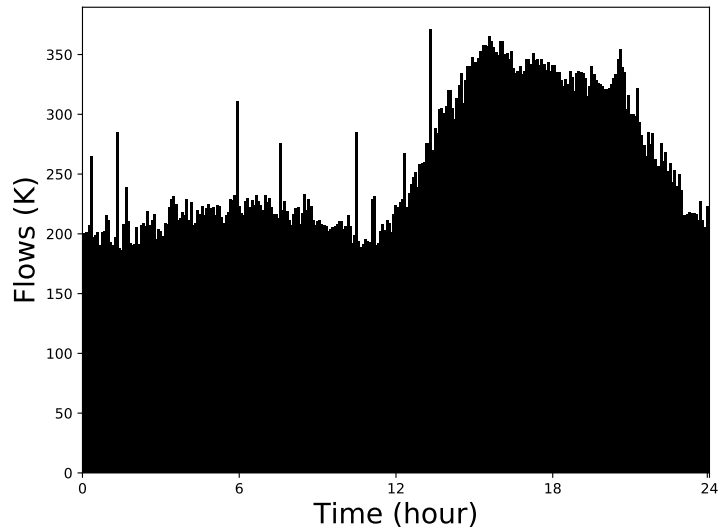


Figure 6.12: Case 1: generated traffic

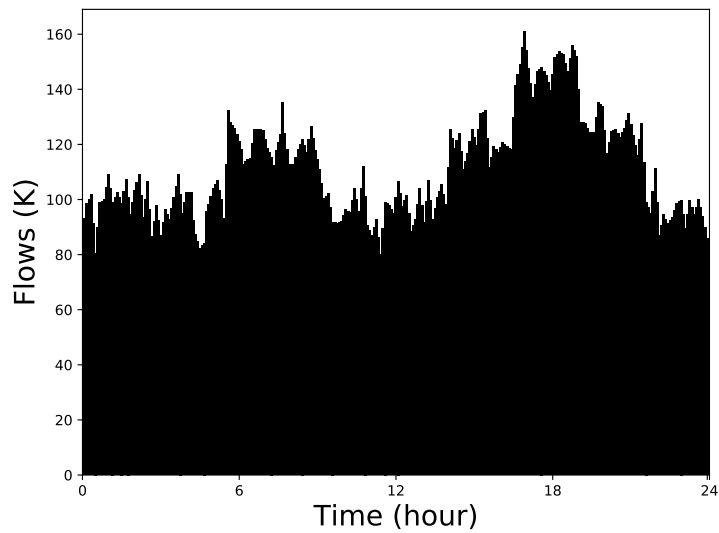


Figure 6.13: Case 2: generated traffic

The flow setup time is the main latency of controllers in OpenFlow networks. It is important to decrease the flow setup time to improve the performance of an OpenFlow network. We analyze the mean flow setup time of each assignment for LANS and DCP-SA. The result is shown in Figure

6.14 and 6.15.

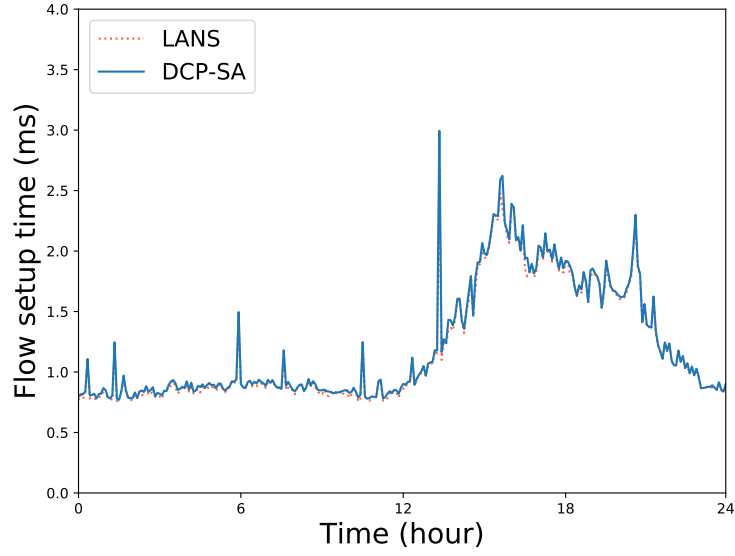


Figure 6.14: mean flow setup time in case 1

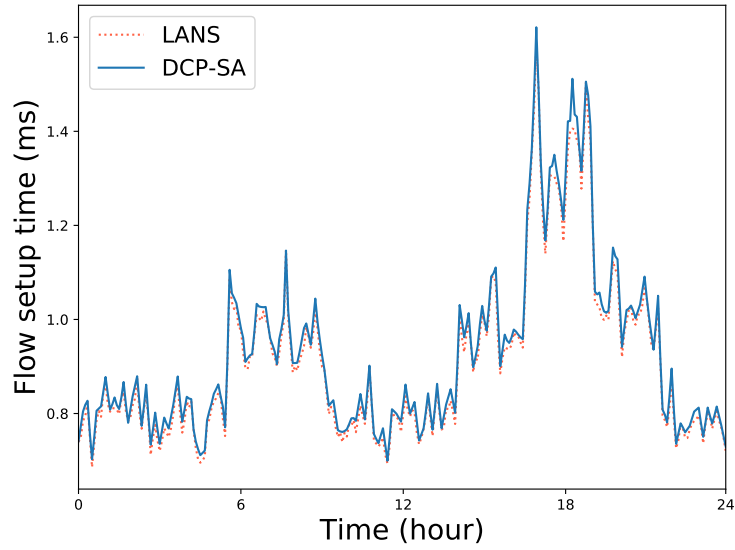


Figure 6.15: mean flow setup time in case 2

The mean flow setup time under both algorithms increases with total traffic load because the

capacity of controllers is limited. We can see that the mean flow setup time under LANS is lower than using DCP-SA for most assignments. The mean flow set up time under LANS is $1.1ms$ and $0.9ms$ in the two cases. It is 8% and 17% lower than using DCP-SA, respectively.

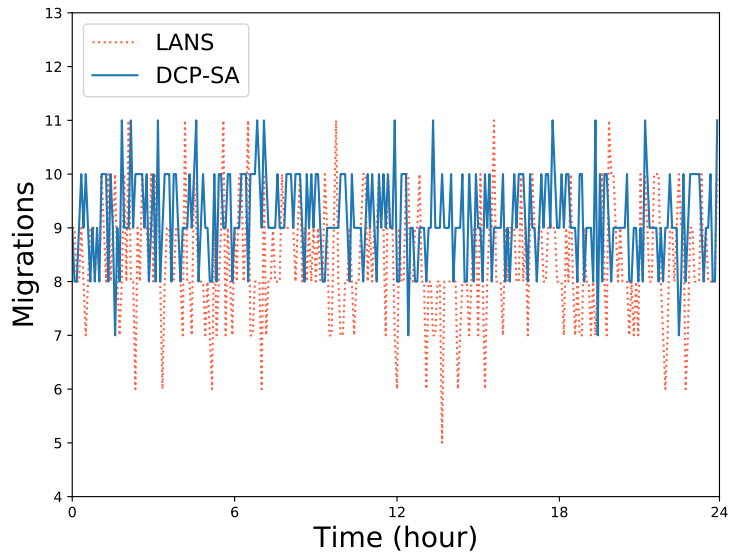


Figure 6.16: Number of migrations in case 1

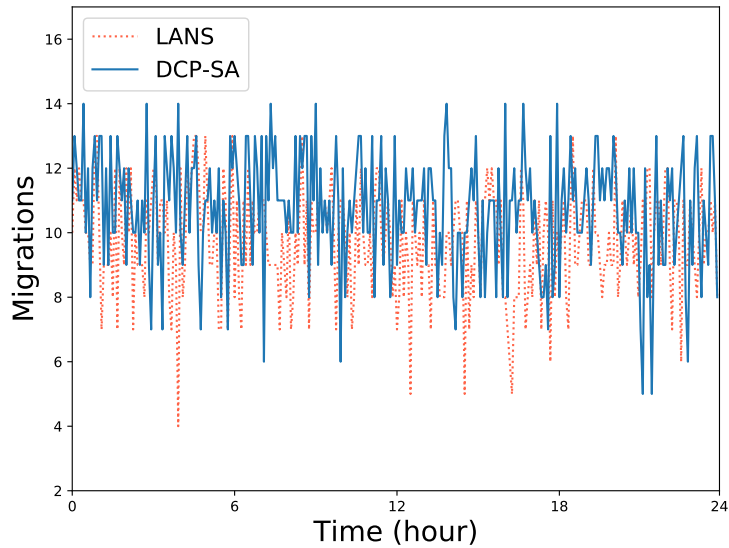


Figure 6.17: Number of migrations in case 2

The cost of switch migration is an inevitable overhead in the multiple controllers situations. Switches must be migrated as rarely as possible to reduce the overhead of migration. The number of migrations of LANS and DCP-SA is shown in Figure 6.16 and 6.17. Given the same initial assignment, different algorithms lead to different new assignments. Therefore, the two algorithms may start from different assignments in the execution. It is not necessary to compare every point in the figure. The mean number of migrations under LANS and DCP-SA are 8.3 and 9.1 in case 1, 9.7 and 10.7 in case 2. LANS makes 0.8 migrations less than DCP-SA on average and 228 migrations less than DCP-SA in total in case 1, 1 migration less than DCP-SA on average and 288 migrations less than DCP-SA in total in case 2.

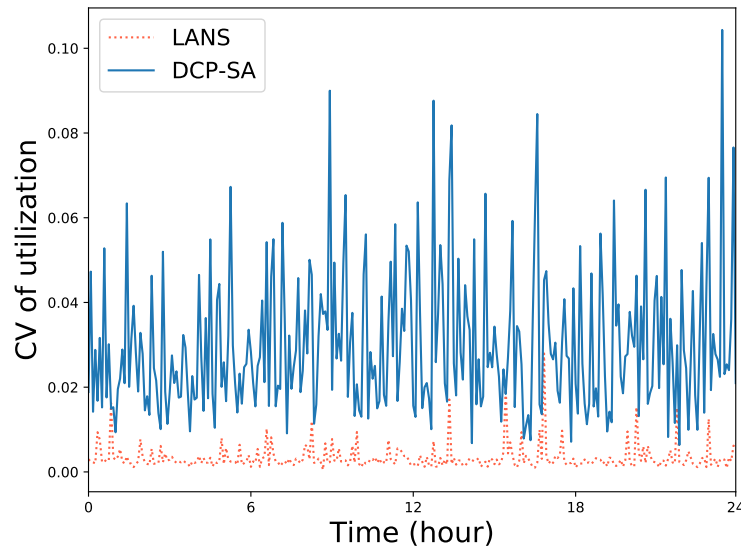


Figure 6.18: Case 1: CV of utilization

The load balancing of controllers impacts the performance of controllers significantly in the multiple controllers situations. If the load is well balanced across the controllers, the balanced controllers keep a network in high performance. We use CV of utilizations of the controllers to measure how balanced the controller are. The smaller the CV is, the better is the controllers balanced. The CV of the utilization of the controllers is shown in Figure 6.18 and 6.19. We can see that the CV under LANS is lower than DCP-SA, which indicates that LANS is more efficient to balance the controllers in a network. The balanced controllers lead to a fast flow setup time as we see in Figure 6.14 and 6.15.

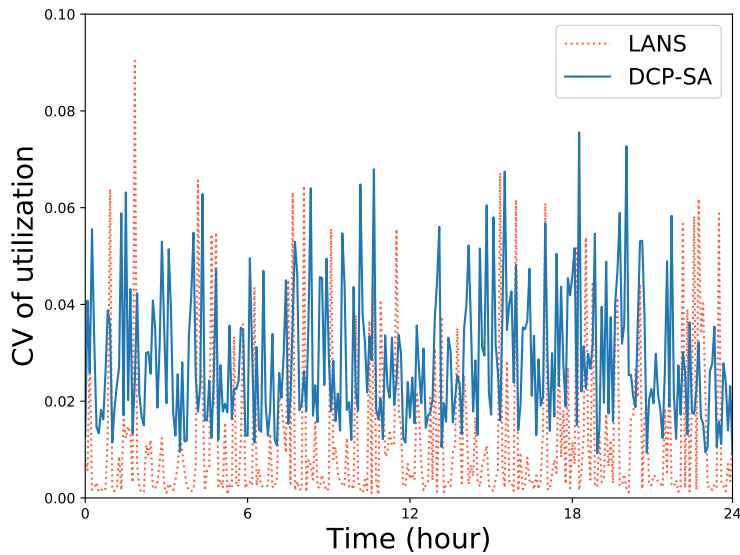


Figure 6.19: Case 2: CV of utilization

6.3 Summary

In this chapter, we have determined the optimal number of controllers in OpenFlow networks. We built a queueing model to evaluate the performance of multiple controllers. There are three types of jobs in the controllers, and there are two service rates in the controllers for different jobs. The average service rate of the controllers increases with the number of controllers and the rate of ascent is slower and slower. The arrival rate at a controller decreases with the number of controllers when the number of controllers is small, and increases with the number of controllers when the number of controllers is large. We develop a tool to measure the response time of our prototype and fit the response time to a hyper-Erlang distribution. We use the fitted distribution in the queueing analysis. The queueing analysis shows the optimal number of controllers decreases with the rate of synchronization messages and increases with the rate of packet-in messages.

We study on the controller assignment problem in the scenario of multiple controllers. Since the controller assignment problem is NP-complete, we proposed a heuristic to solve it. We also design a greedy algorithm to generate a feasible assignment as the input of the heuristic. We measure the flow setup time of a controller and model each controller as an $M/PH/1$ queue to capture its performance. The queueing model is used in the heuristic for the fitness function. The heuristic avoids a local minimum by keeping a candidate list. It accepts a candidate if the candidate has a

lower cost after a certain number of steps. We evaluate our solution in Mininet. The results show that our solution can balance the controller better, reduce the flow setup time and make less migrations less than DCP-SA.

Chapter 7

Buffer management

The burst packet-in message may decrease the performance of a controller even degrade the performance of the whole network. In this chapter, we describe the reason of burst packet-in messages and avoid the burst packet-in message using the buffer in OpenFlow switches. We model our method and the general OpenFlow switch using queueing models, and evaluate the performance of our method in simulations.

7.1 The Limitations of Existing Buffer Management

When a packet arrives at an OpenFlow switch, the switch tries to find a flow entry that matches the packet. If the switch finds one, instructions in the flow entry will be executed and the packet may be forwarded, modified or dropped. If no matching flow entries are found, the switch will check whether the buffer is full. If the buffer is not full, the switch stores the packet in the buffer, and sends a packet-in message to the controller, which contains the header of the packet and a buffer ID. If the buffer is full, the switch sends a packet-in message with the whole packet to the controller. After the controller receives the packet-in message, it will install flow entries in the switch, so that the switch can deal with packets with the same header without consulting the controller.

As we introduced in Chapter 2, the buffer can reduce the load of switches and the traffic in OpenFlow channel, but it only reduces the size of a packet-in message. Considering that a user accesses a video stream over UDP, the video server sends many packets to a switch. When the first packet in the flow arrives at the switch, there is no suitable flow entry for the packets in the flow, the switch must request a forwarding rule from the controller. The other packets in this flow will continuously arrive before the controller replies. The switch has to forward all the packets to the

controller until a flow entry is installed in the switch.

We use Mininet and deploy a Ryu controller on a laptop to analyze the number of packet-in message in one flow arriving at the Ryu controller when burst packets happen. We simulate a switch and two hosts in Mininet, the topology is shown in Figure 7.1. We run iPerf on host A generating UDP traffic at rate $10M/s$. Figure 7.2 shows the distribution of the number of packet-in messages in a burst. There are about 8 packet-in messages for one flow, but the controller can install a flow into the switch by receiving the first packet-in message in a flow. The other packet-in messages are useless for the controller. The burst of packet-in messages will degrade the performance of an OpenFlow network, because it increases the workload for both switches and controllers. A switch usually does not have a strong CPU, creating and sending so many packet-in messages exhaust their processing capacity. The controller only needs one packet-in message to install a forwarding rule, most of the packet-in messages are unnecessary in this case.

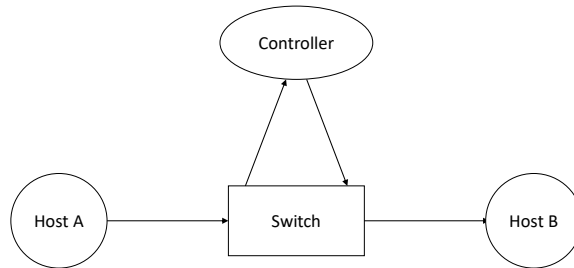


Figure 7.1: Topology for burst messages test

Burst packets happen not only in UDP streams but also in TCP streams. TCP flows transmit packets after the three-way handshake, so the forwarding rules are installed during the establishment of the connection. However, a switch can see 10^5 flows per second [15] and the forwarding rules for connected TCP flows may be replaced based on the limited size of flow tables. Besides, Benson also pointed out that when there is an ON/OFF mode in data center traffic [15], the flow entries for established TCP connections may be removed in the OFF time. The burst packets can significantly degrade the performance of OpenFlow networks, even turning the controllers into a bottleneck of the OpenFlow network. We need a mechanism to minimize the effect of burst packet.

7.2 The Proposed MPT Model

The burst of packet-in messages can significantly degrade the performance of OpenFlow networks, even the controllers become the bottleneck of the OpenFlow network. In order to overcome

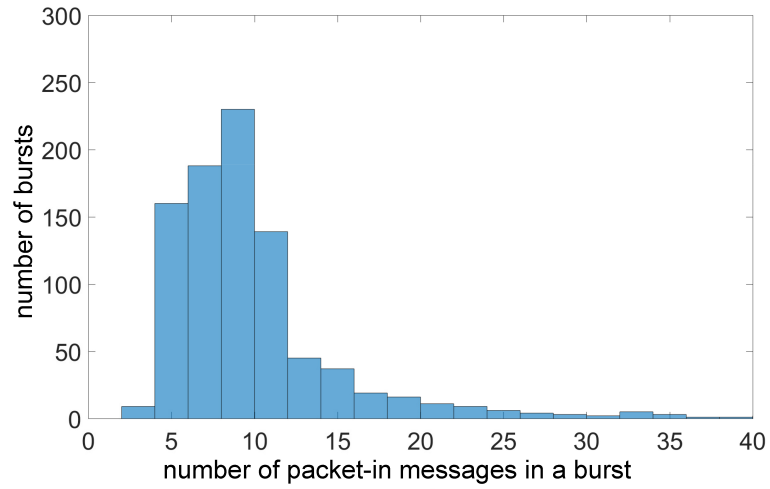


Figure 7.2: Burst of packet-in on the controller

the deficiency of buffer management in existing OpenFlow switches, we propose a novel model of buffer management to avoid bursts of packet-in messages, named MPT (Mismatched Packets Table). We add a packet table in the switches. The MPT stores the headers of mismatched packets. With MPT, the OpenFlow switch sends the first packet to the controller, while other packets in the same flow are buffered at granularity of a flow. Considering that the packet-in message may be lost in the connection between the switch and the controller, the switch will send a second packet-in message to the controller after a random time.

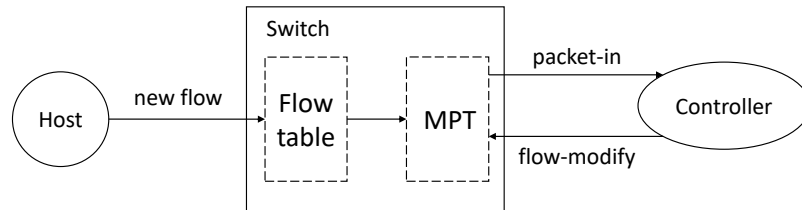


Figure 7.3: Overview of the MPT

Figure 7.3 shows how MPT works. A switch tries to find a match the MPT before the mismatched packet is sent to the controller. If the switch cannot find a matching entry, it forwards the packet to the controller and creates an entry in the MPT. The entry contains the header of the mismatched packet. If the switch finds a matching entry in the MPT, it means that a packet with the same header has been sent to the controller. The switch keeps the packet in buffer and sends it to the controller

after an exponential random time. The MPT also modifies the processing of flow-modify messages. When the switch receives a flow-modify message that adds a flow entry into the switch, the switch should install a flow entry, delete all the entries in MPT that match the flow entry and apply the instructions to all the packets in the buffer that match the flow entry. The entries in the MPT should be removed after a certain time, and all the packets in the buffer that have the same header with the expired entries should be also removed. The workflow of MPT is illustrated in Figure 7.4.

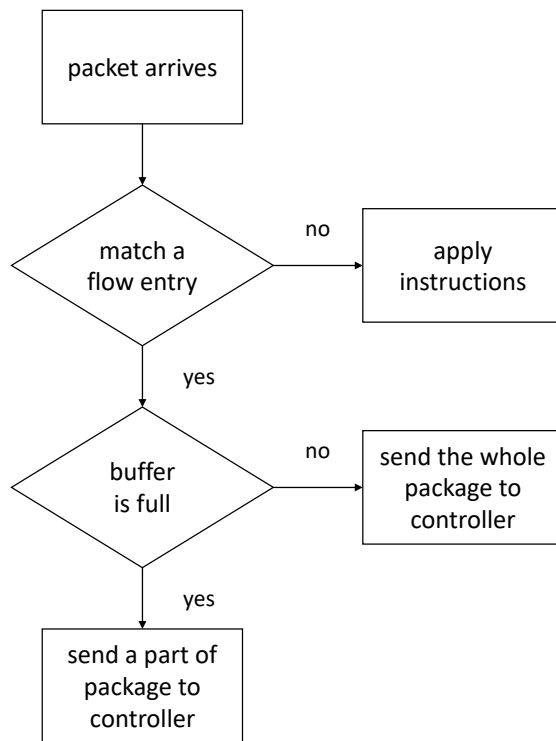


Figure 7.4: Workflow of MPT

The MPT aims to reduce the packet-in messages, it makes the controller not being overloaded by the burst packets. The workload of a controller with MPT is lower than the general switch. Therefore, the controller can process the requests from switches faster, which decreases the delay in the controller and the flow completion time.

7.3 Queuing Model of the OpenFlow Controller

We introduce queuing models of an OpenFlow controller with general switches and MPT switches.

7.3.1 Controller Performance of General Buffer

If the time interval between packets arrivals is shorter than the flow setup time, packets burst will happen. The forwarding process of burst packets is shown in Figure 7.5. When the first packet of a flow arrives, the switch cannot find a matching entry and sends it to the controller. Many other packets in the same flow arrive before the switch receives a response from the controller. The switch still cannot find any matched flow entries for the packets, and has to send all of them to the controller. So the controller receives many packet-in messages in a very short time, and replies a flow-modify message for each one. The switch creates a flow entry when it receives the first flow-modify message, so the packets in the same flow can be forwarded directly. The switch receives many flow-modify messages, because it sends many packet-in messages to the controller. The switch should replace the installed flow entry for every flow-modify message from the controller.

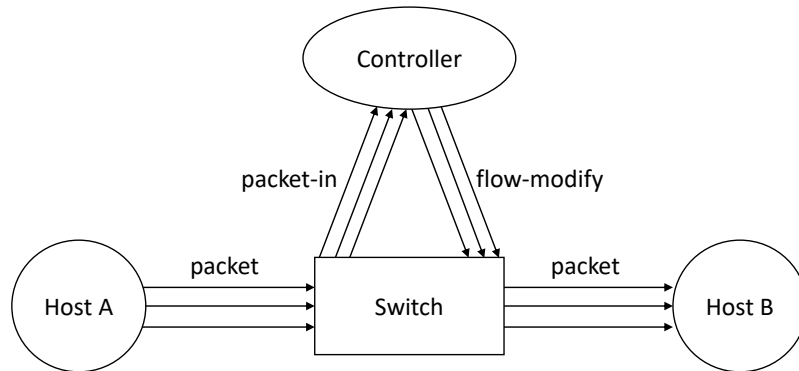


Figure 7.5: Burst packets forwarding process with general switch

Some analytical models of OpenFlow networks suppose that the packet-in messages arrive at a controller following a Poisson process [59] because network traffic measurements have indicated that flow arrival in packet switching networks follows a Poisson process [146] and they assume the switches only send the first packet of a flow to the controller. These models are suitable for the modeling of the controller performance only if the second packet of a flow arrives after the flow entry installation. However, in the burst packets scenario, the time interval between packets is much shorter than the flow setup time. Many packets arrive before a flow entry is installed. Previous studies showed that packets in networks arrive as batches [31]. Some packets arrive at a switch at

the same time as a batch. Xiong's work showed that the arrival of packet batches following Poisson process [148].

We assume a packet arrives at a switch at time t and it is sent to the controller via a packet-in message. The controller receives the packet-in message after an additional time Δt . The time difference Δt consists of flow table lookup time and the transmission time of the packet-in message from the switch to the controller. The lookup time depends on the size of the flow table and the transmission time depends on the distance between the switch and the controller. Both are fixed in a given network. So Δt tends to be constant in a given OpenFlow network. Consequently, it can be assumed that the packet-in message stream from switches is equivalent to the packets arrival process in terms of a stochastic process. We assume that the number of packet-in messages in a batch follows a Poisson distribution. At the same time, other flows are arriving at the switches that will not cause a burst of packet-in messages. For simplicity, we assume that the processing time of the packet-in messages can be modelled as an independent, identically random variable following an exponential distribution. Then we can formulate a class based queueing model to analyze the behavior of an OpenFlow controller. There are five characteristics of this queueing model: (1) there are two types of packet-in messages in the system, (2) the two types of packet-in messages both arrive at a controller following a Poisson process, (3) the first type of packet-in messages arrives at a controller as a batch, (4) the number of packet-in messages in a batch follows a Poisson distribution, (5) the processing time of the controller conforms to exponential distribution. The queueing model is shown in Figure 7.6.

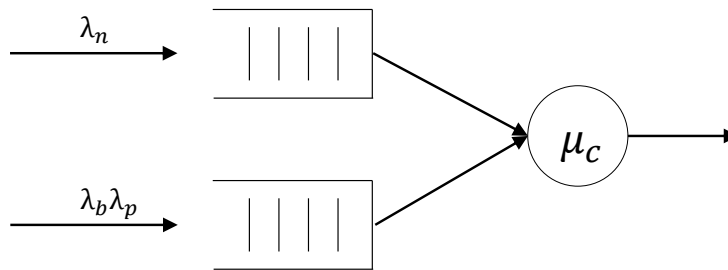


Figure 7.6: The queueing model of the controller

We denote the rate of batch arrivals λ_b , the mean number of packet-in messages in a batch λ_p , the rate of single packet-in arrivals λ_n , the processing time of the controller μ_c . If a single packet-in message arrives at the controller, we also consider it as a batch arrival, but there is only one packet-in message in the batch. The batch arrival rate is $\lambda_b + \lambda_n$. With probability $\frac{\lambda_n}{\lambda_b + \lambda_n}$, there is one packet-in message in the batch. With probability $\frac{\lambda_b}{\lambda_b + \lambda_n}$, the number of packet-in messages in the

batch following Poisson distribution.

Suppose a batch with m packet-in messages arrives at a controller, where n packets are waiting for the service. Then the l th ($l = 1, 2, \dots, m$) packet-in message in this batch has to wait until the $n + l - 1$ packet-in messages before it is processed. Then the controller will spend $1/\mu_c$ time to process it. Let the total time that the l th packet-in message in the batch spends in the controller be T_l . We get

$$\begin{aligned} T_l &= \frac{n + l - 1}{\mu_c} + \frac{1}{\mu_c} \\ &= \frac{n + l}{\mu_c}. \end{aligned} \quad (7.1)$$

The average packet-in message processing time of the batch is T_b . We can get $E[T_b]$ from Eq. 7.2.

$$\begin{aligned} E[T_b] &= \frac{1}{m} \sum_{l=1}^m T_l \\ &= \frac{n}{\mu_c} + \frac{1}{m} \sum_{l=1}^m \frac{l}{\mu_c} \\ &= \frac{n}{\mu_c} + \frac{m + 1}{2\mu_c} \end{aligned} \quad (7.2)$$

Furthermore, we can calculate the average packet-in message processing time of the controller $E[T_p]$.

$$\begin{aligned} E[T_p] &= \sum_{n=0}^{\infty} \sum_{m=1}^{\infty} T_b P_m P_n \\ &= \frac{1}{\mu_c} \sum_{n=0}^{\infty} n P_n + \frac{1}{2\mu_c} \left(\sum_{m=1}^{\infty} m P_m + 1 \right), \end{aligned} \quad (7.3)$$

where P_m is the probability that there are m packet-in messages in a batch, P_n is the probability that there are n packet-in messages in the controller when a batch arrives.

With probability $\frac{\lambda_b}{\lambda_b + \lambda_n}$, the number of packet-in messages in a batch conforms to a Poisson distribution with parameter λ_p . With probability $\frac{\lambda_n}{\lambda_b + \lambda_n}$, there is one packet-in messages in a batch.

We can obtain

$$\sum_{m=1}^{\infty} mP_m = \frac{\lambda_b\lambda_p + \lambda_n}{\lambda_b + \lambda_n}. \quad (7.4)$$

The relationship between the mean queue length $E[Q]$ and the mean service time is shown in Eq. 7.5.

$$\begin{aligned} E[Q] &= \sum_{n=0}^{\infty} nP_n \\ &= E[T_p] \left(\frac{\lambda_b}{\lambda_b + \lambda_p} \lambda_b\lambda_p + \frac{\lambda_n}{\lambda_b + \lambda_p} \lambda_n \right). \\ &= E[T_p] \frac{\lambda_b^2\lambda_p + \lambda_n^2}{\lambda_b + \lambda_n}. \end{aligned} \quad (7.5)$$

Substituting Eq. 7.4 and Eq. 7.5 into Eq. 7.3, the average packet-in message processing time of the controller can be computed as Eq. 7.6.

$$\begin{aligned} E[T_p] &= \frac{1}{\mu_c} E[T_p] \frac{\lambda_b^2\lambda_p + \lambda_n^2}{\lambda_b + \lambda_n} + \frac{1}{2\mu_c} \left(\frac{\lambda_b\lambda_p + \lambda_n}{\lambda_b + \lambda_n} + 1 \right) \\ &= \frac{1}{2} \frac{\lambda_b\lambda_p + 2\lambda_n + \lambda_b}{\mu(\lambda_b + \lambda_n) - \lambda_b^2\lambda_p - \lambda_n^2}. \end{aligned} \quad (7.6)$$

7.3.2 Controller Performance of the MPT

The packet-in message arrival process would be different if the switches in an OpenFlow network use MPT. When a batch arrives at a switch, the switch sends the first packet of this batch to the controller and add its header to the MPT. After that, the switch can find an MPT entry for all the packets in this batch. The other packets are stored in the buffer and sent to the controller following a Poisson process. If a flow arrives at a switch that will not cause the burst of packet-in messages, the switch will send one packet-in message to the controller. In this case, switches with MPT behave the same as general switches. The controller receives packet-in messages following a Poisson process with MPT. The forwarding process of a burst of packets in an OpenFlow network with MPT is shown in Figure 7.7. We can adopt a queueing model with job classes to analyze the behavior of an OpenFlow controller. There are two types of packet-in messages in the system, both arrive at the switches following a Poisson process. As we assume before, the service time of the controller

follows an exponential distribution. The queuing model is shown in Figure 7.8.

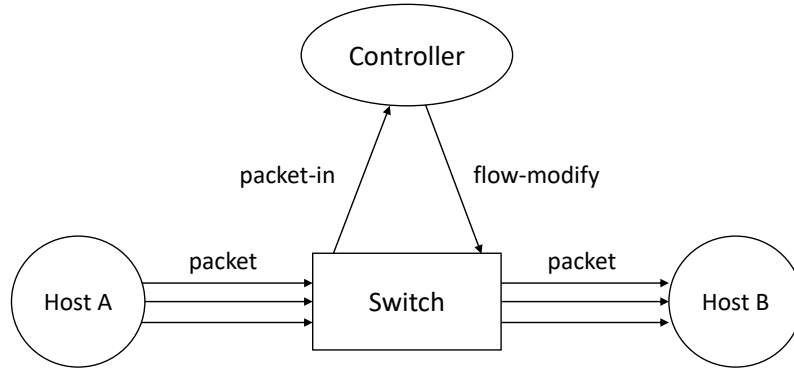


Figure 7.7: Burst packets forwarding process with MPT

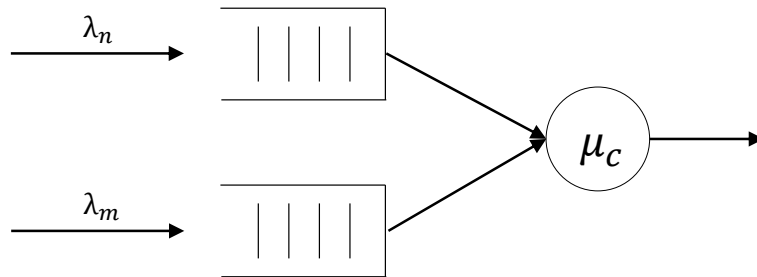


Figure 7.8: The queuing model of the controller

We denote the rate of single packet-in arrival as λ_n , the rate that MPT sends packet-in messages to the controller is λ_m , the processing time of the controller as μ_c . Merging the two types of the packet-in messages, we can derive the packet-in messages arriving at the controller at rate $\lambda_n + \lambda_m$. Let p_i be the probability that there are i packet-in messages in the controller, and the mean number of packet-in messages in the controller be $E[N]$. $E[N]$ can be computed using Eq. 7.7.

$$\begin{aligned} E[N] &= \sum_{i=0}^{\infty} ip_i \\ &= \sum_{i=0}^{\infty} i \left(\frac{\lambda_n + \lambda_m}{\mu_c} \right)^i \left(1 - \frac{\lambda_n + \lambda_m}{\mu_c} \right). \\ &= \frac{\lambda_n + \lambda_m}{\mu_c - \lambda_n - \lambda_m} \end{aligned} \quad (7.7)$$

The time a packet-in message spends in the controller is:

$$E[T] = \frac{E[N]}{\lambda_n + \lambda_m} = \frac{1}{\mu_c - \lambda_n - \lambda_m} .. \quad (7.8)$$

7.4 Evaluation

We evaluate the general buffer management and our proposed method with different parameters by queueing analysis and simulations in this section.

7.4.1 Queueing Analysis

At first, we show how the burst of packet-in messages impacts the response time of the controller. Suppose a general OpenFlow switch receives packets at batch arrival rate $\lambda_b = 2K$, and single arrival rate $\lambda_n = 10K$ per second. The controller processes the packet-in messages at rate $\mu_c = 16M$ per second. We analyze the mean packet-in message processing time of the controller with different average batch sizes in Figure 7.9.

As illustrated in Figure 7.9, the response time of the controller increases exponentially when the average batch size increases linearly. That means the average batch size does not cause the performance problem significantly when it is small. But as the average batch size increases the response time increases severely. As we measured in Figure 7.2, the average size of an arriving batch is about 8, where the response time is twice that of a batch size two.

Suppose a general OpenFlow switch receives packets at batch arrival rate $\lambda_b = 2K$ per second and the average number of packets in a batch is $\lambda_p = 8$. The switch sends these packets to the controller with the same rate. A switch with MPT also receives packets at batch rate $\lambda_b = 2K$ per second and the average number of packets in a batch is $\lambda_p = 8$. The switch with MPT puts these packets in a buffer and sends packet-in messages to the controller following a Poisson process at rate

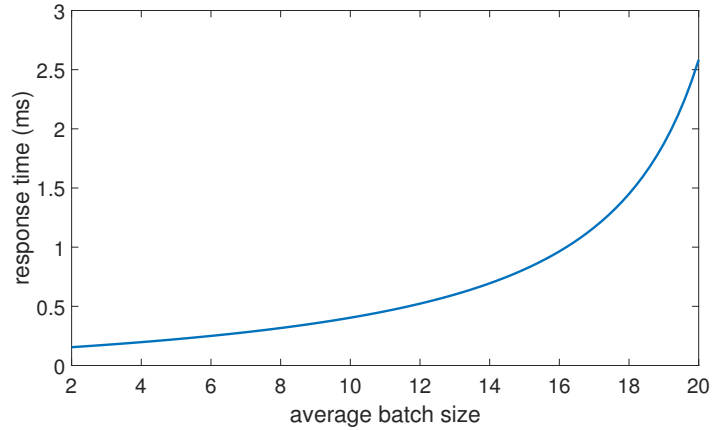


Figure 7.9: Response time with different batch sizes

$\lambda_m = 4, 8, 16K$ per second. At the same time, both switches receive packets with single arrival rate $\lambda_n = 10K$ per second. We calculate the mean packet-in message processing time of the controller with different service rates in Figure 7.10.

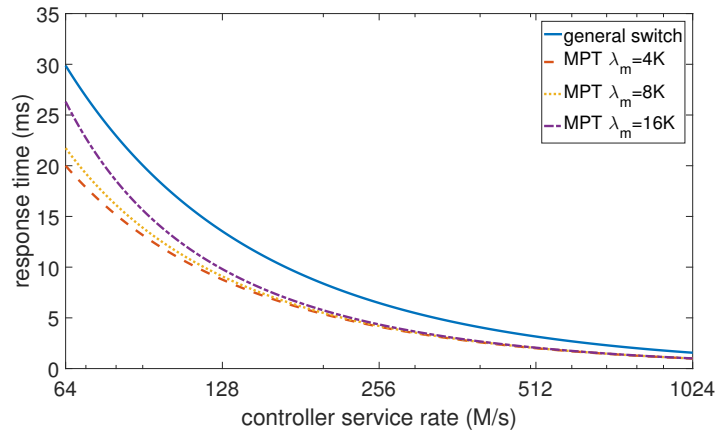


Figure 7.10: The mean response time with different service rates

As illustrated in Figure 7.10, the mean packet-in message processing time of the controller with MPT at different service rates is shorter than the general one. Because the general switch sends too many packet-in messages to the controller in a very short time, these packet-in messages consume a lot of computing resource. When the service rate is low, which means the controller is not very powerful, the difference between them is very big. The difference decreases when the service rate

of the controller is increasing. With MPT, there is no batch arrival, the mean processing time of the controller does not change much for the different arrival rates. Thus, we can say that the burst packets degrade the performance of OpenFlow networks severely when the performance of the controller is not high. The effect of burst packets may be not that important if the controller is very powerful, but it still increases the mean processing time of the controller. At a service rate $1024M$ per second, the response time of MPT is still 30% faster than the general switch. On the other hand, the difference among the response time of MPT at different arrival rate is not that big. The times are nearly the same after a service rate $256M$ per second. This is because the batch arrival is the main reason the response time increases and the MPT avoids batch arrivals.

Fixing the service rate of the controller at rate $\mu_c = 512M$ per second, the average number of packets in a batch at 8, the single arrival rate at $\lambda_n = 10K$ per second, we vary the batch arrival rate λ_b from $1K$ to $5K$ per second. The general OpenFlow switch sends all these packets to the controller, and the switch with MPT sends packet-in messages to the controller at a rate $\lambda_m = 8\lambda_b, 16\lambda_b, 32\lambda_b$ per second. We show the mean processing time of the controller with different batch arrival rates in Figure 7.11.

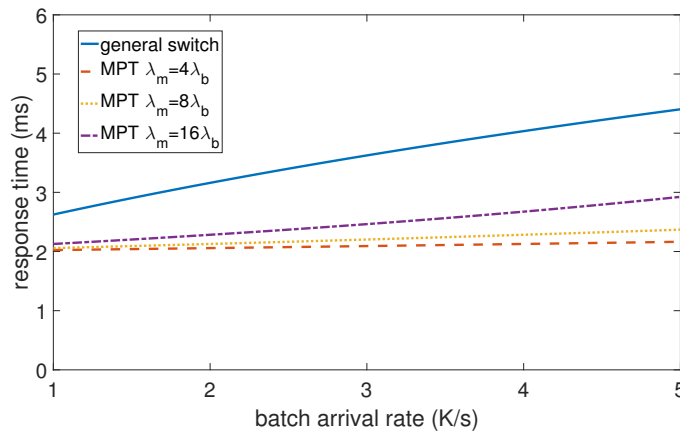


Figure 7.11: The mean response time with batch arrival rates

As we can see, the mean processing time of the controller with the general switch is longer than our proposed method at any batch arrival rates. Because the burst of packet-in messages are the main reason that increases the service time. The MPT can reduce packet-in messages very effectively in the burst packets scenario. The mean processing time of MPT is stable when the batch arrival rate is $\lambda_m = 8\lambda_b, 16\lambda_b$, the mean processing time of MPT increases when the batch arrival rate is $\lambda_m = 32\lambda_b$. It still much faster than the general switches. The response time with different batch

arrival rate increases nearly linearly. The response time of the general switch increases faster than the MPT at any batch arrival rate. With $\lambda_m = 16\lambda_b$, the response time of MPT is 20% faster than the general switch at batch arrival rate $1K$ per second, and this difference between the response increase with the batch arrival rate.

7.4.2 Simulation

To evaluate how MPT works on OpenFlow controllers, we implement a simulator to capture the behavior of controllers based on SimPy [126]. SimPy is a process-based discrete-event simulation framework. Under this framework, the measurement of an OpenFlow controller are performed with the following steps: (1) creating two processes for the switch and controller respectively; (2) creating a queue for the communication between the switch process and the controller process; (3) the switch process generates packet-in messages and puts them in the queue; (4) the controller process serves the packet-in messages in the queue using a FCFS scheduling policy; (5) recording the response time of the controller and the number of packet-in messages.

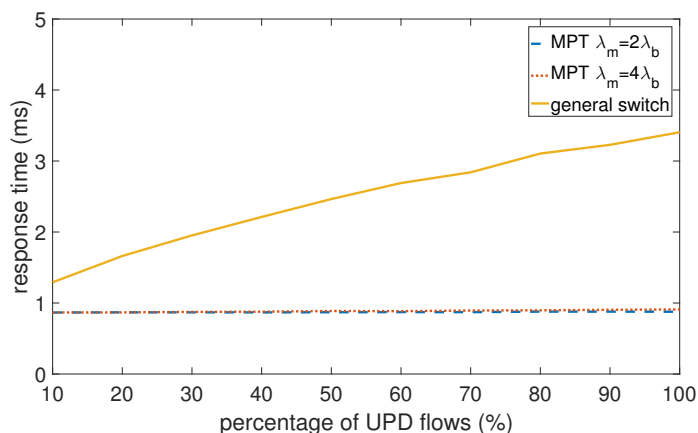


Figure 7.12: The response time of the controller

In the simulation, we use the traffic data from passive network monitors of CAIDA. There are $2.43K$ UDP flows per second. The UDP flows contribute 19% of all flows in the networks, but not all the UDP flows cause bursts of packet-in messages, so we take a percentage of the UDP flows to generate the burst of packet-in messages, and the rest of the UDP flows only generate one packet-in message. If there are n percent of UDP flows that generate the burst of packet-in messages, the general OpenFlow switch will send packet-in message to the simulated controller at rate $\lambda_b = 24.3n$, and the average number of packet-in messages in a batch is $\lambda_p = 8$. A switch

that is simulated with MPT sends packet-in messages following Poisson process at different rates. The simulated OpenFlow controller can process $128M$ packet-in messages per second. We run the simulation for 100 seconds. The mean processing time of the controller with different service rates is shown in Figure 7.12.

The mean processing time of the controller with our proposed method is 30% lower than the general switch at least. Because a switch with MPT does not send useless packet-in messages to the controller, which reduce the queue in the controller. The response time of general switch increases very rapidly when the percent of UDP flows that generate the burst of packet-in messages increases. The response time of MPT increases very slowly. Because the MPT reduce the number of packet-in messages effectively, the burst of packet-in messages do not impact the performance of the controller significantly.

In the second simulation, we use the same parameters but record how many packet-in messages arrive at the controller. The result is shown in Figure 7.13.

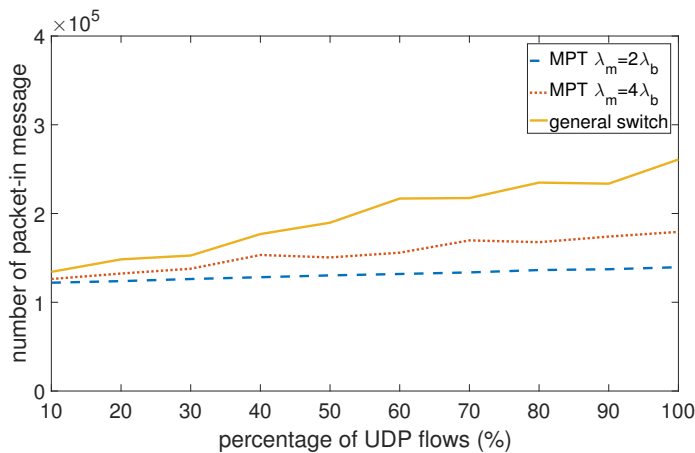


Figure 7.13: The number of packet-in messages

As we can see, the number of packet-in messages with the general switch is greater than MPT at any batch arrival rates. When the batch arrival rate is low, the difference is not that big, but it increases with the batch arrival rate. Because the a high number of packet-in messages in a batch let the number of packet-in messages increase fast. MPT can save 6% of the packet-in messages when 10% of the UDP flows generate a burst of packet-in messages, and increases the reduction with the volume of the UDP flows that generate the burst of packet-in messages.

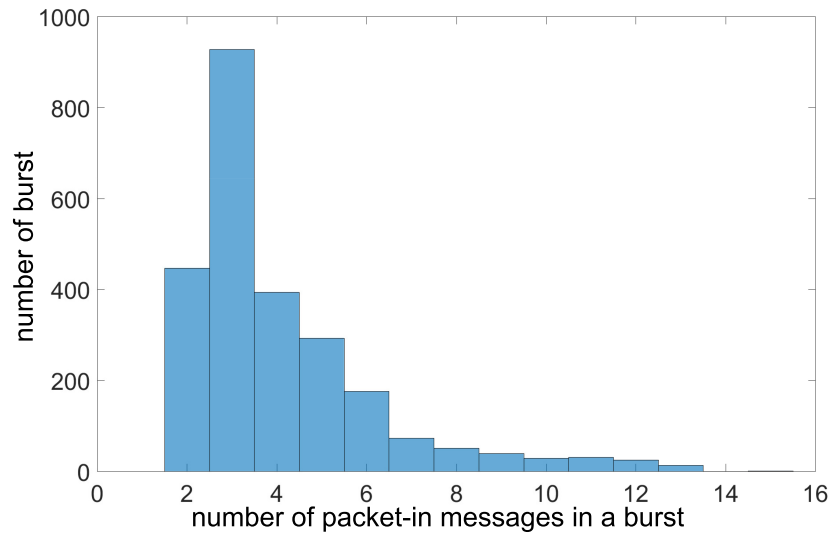


Figure 7.14: Number of packet-in message with general switch simulation

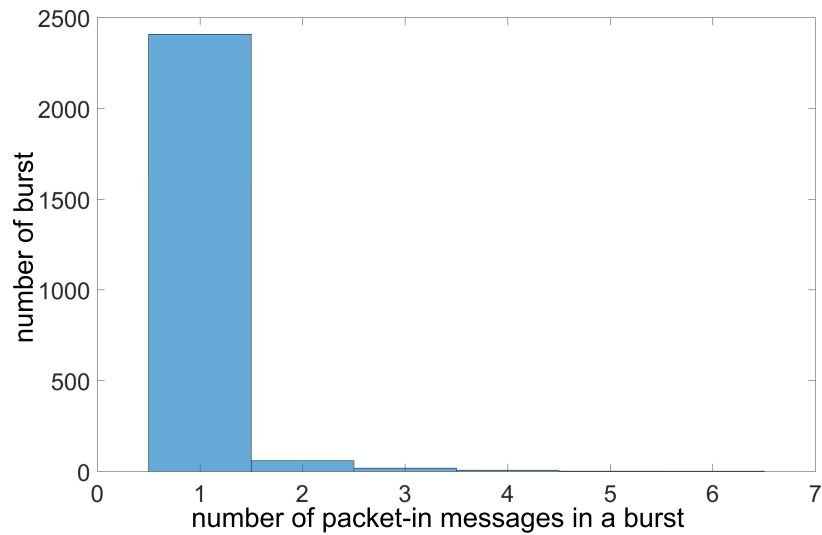


Figure 7.15: Number of packet-in message with MPT switch simulation

To evaluate how effective MPT can reduce the burst of packet-in messages, We use the virtual switch in OFCP to simulate an OpenFlow switch. To simulate the general OpenFlow switch, the tool continually sends packet-in messages to the Ryu controller until receives a flow-modify message. To simulate the MPT switch, the tool repeats the following steps: send a packet-in message, wait

an exponential random time, until it receives a flow-modify message. We do both experiments 2500 times, and analyze the number of packet-in messages arriving at the controller for one flow.

Figure 7.14 shows the result of the general switch simulation. At least, the controller receives two packet-in messages for one flow, and the mean number of the packet-in messages for one flow is 4.06. With probability 0.3, the controller receives more than four packet-in messages. The controller may even receive more than ten packet-in messages for one flow.

Figure 7.15 shows the result of the MPT switch simulation. With probability 0.95, the controller only receives one packet-in message for one flow, and the mean number of packet-in messages is 1.06. With MPT, the controller may receive more than three packet-in messages for one flow, but the probability is less than 0.01. The MPT can decrease the number of packet-in messages on the controller effectively when burst packets happen. For most cases, the controller only receives one packet-in message for one flow.

7.5 Summary

In this chapter, we have discussed how burst packets degrade the performance of the controller in OpenFlow networks and have presented a method to reduce the packet-in messages by using the built-in buffer in OpenFlow switches so as to reduce the workload of the controllers. This method moves some workload of the data plane to the control plane. A switch looks up two times for every mismatched packet, this may degrade the performance of a switch. However, switches are usually hardware. They are much faster than controllers. Therefore, it is worth moving workload to the control plane. The proposed method can reduce the unnecessary packet-in messages when there are many UDP packets in an OpenFlow network. Queueing analysis and simulation results indicate that our method can decrease the mean processing time of the controller very effectively when the performance of the controller is not high. The queueing analysis shows that the response time of MPT is 30% faster than the general switch. The simulation shows that MPT can save 6% of the packet-in messages when there are 10% of the UDP flows that generate a burst of packet-in messages. The controller receives more than four packet-in messages with probability 0.3 when a general switch is used. The controller only receives one packet-in message for one flow with probability 0.95, when a switch with MPT is used.

Chapter 8

Server Load Balance in OpenFlow Networks

Many online service providers use multiple servers to handle user requests. In order to minimize the response time of requests and enhance user experience, requests from different users are handled by different servers [114]. This reduces the amount of workload for each server. OpenFlow offers high flexibility with the programmable controller. In this chapter, we implement a dynamic load balancer based on the OpenFlow protocol. The dynamic load balancer collects the CPU and memory utilization of each server and dispatches user requests based the load of servers.

8.1 Load balance for online service

Load balancing methods can be used for web services, FTP services, business-critical applications and other online applications [24]. Traditional load balancers are expensive, and the load balancing policies need to be created in advance. The lack of flexibility leads to an inability to dealing with variable traffic. Traditional load balancers require a dedicated administrator for maintenance and do not allow the user to design flexible strategies based on their network conditions. Since all requests are passed through a single piece of hardware, any failures on the load balancer will cause the collapse of the entire service.

The various load balancing schemes have different deficiencies in modern online services. The fundamental reason is that traditional network design has some shortcomings [104]. Under the impact of new demands, the bottleneck of traditional network architecture has been reflected in many aspects. Researchers are looking for new possibilities to meet changing demands. Controller appli-

cations in an OpenFlow network can well overcome some shortcoming of traditional load balancing, and provide a simple and effective solution with high flexibility. Due to the difference between traditional network and an OpenFlow network, they are inevitable difference between traditional network and an OpenFlow network. New problems have emerged, such as the load balancing module design, monitoring the status of a server and the flexibility of load balancing. On the other hand, many businesses in enterprise networks are migrating to virtual environments because virtualization technology can simplify the management and maximize the utilization of limited resources [4]. But the load balancing technology is not mature enough in virtual environment, traditional load balancers are under restrictions in virtualization environments.

We design and implement an OpenFlow-based dynamic server load balancing in a virtualized environment. It is effective and flexible. New modules can be easily deployed in the controller for implementing customizable policies. We deploy a single controller in the experiments, but our method supports multiple controllers. Users can deploy multiple controllers to improve the performance and robustness of the architecture.

8.2 Server Load Balancing Strategy

In this section, we introduce server load balancing strategy SBLB (Status-Based Server Load Balancing). The load balancing architecture consists of an OpenFlow network with a controller and server cluster which is connected to the OpenFlow network. The controller manages the OpenFlow switches. At the same time it obtains status from the servers regularly.

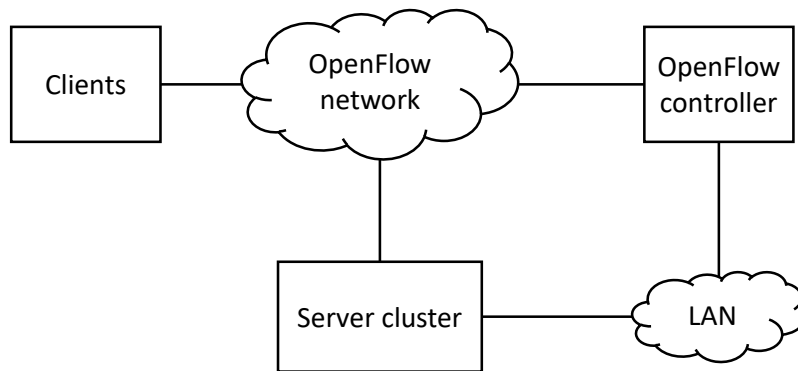


Figure 8.1: Load balance architecture based on OpenFlow

As is shown in Figure 8.1, each server has a static IP address. The controller maintains a server address pool and obtains network topology. There is a virtual IP address used by clients to send

requests. From the client's perspective, the server cluster can be considered a server with the virtual IP address. All the servers share the virtual address and all the clients will send their requests to the virtual IP address. The clients treat the server cluster as one server whose IP address is the virtual IP address. When a packet-in message arrives at the controller, the controller will check if the IP address of the packet is the same as the virtual IP address, by which the servers provide service. If not, the controller will forward the packet normally. Otherwise, the controller will select a server to respond to the client based on the load balancing strategy. It will also add a flow entry to the OpenFlow switch.

8.2.1 Load Balancing Strategy

There are three steps in the SBLB algorithm.

(1) Dynamically obtaining the current server load. We collect the status of servers. For server S_i , we collect its CPU utilization $L(C_i)$, memory utilization $L(M_i)$. This information does not directly represent the load of a server, so we use a function to indicate the load of server S_i .

$$L(S_i) = w_1L(C_i) + w_2L(M_i), \quad (8.1)$$

where $w_1 + w_2 = 1$. Because of these different type of parameters, which can have different effect on the server, we introduce the weights w_1, w_2 to combine them together.

(2) Calculate the rank. Denote the initial rank of server S_i as r_i , When the server load balancing system comes into use, r_i is set as $L(S_i)$. Then the rank is updated according to Equation 8.2.

$$r_i = r_i + A\sqrt[3]{\rho - L(S_i)}, \quad (8.2)$$

where ρ (the default value is 0.8) is the server utilization we want to achieve, and A is an adjustable coefficient (the default value is 5). If A is too small, the historical state of the servers impacts the rank effectively. Using Eq. 8.2, we can control the rank of a server in a non-linear way. This non-linear rank can describe the load of servers reasonable because the relation between the requests and response time is non-linear. If the relative load is ρ , the server's rank will not change. If the relative load is greater than ρ , it will become smaller. Otherwise, it will become larger. As the server load changes, the rank will be adjusted. In order to avoid that the weight becomes too large, we set the maximum of a rank to n , which can be adjusted. We set its default value to 5. If the rank is greater than the maximum, we set the rank to the maximum. The maximum value of rank can increase the probability of the most loaded server to be selected. If there are a lot of servers in the cluster, we can choose a big value. The controller collects the server status periodically to query the

load parameters, calculate the relative load, and update the rank of each server. This is a dynamic feedback formula that will adjust the weights to a stable value. When a system achieves the desired status, the rank will be constant. In practice, if the ranks of all the servers are very small, the server cluster is considered overloaded. In this case, a new server needs to be added to the server cluster to address this issue.

(3) Select a server. For each new incoming request, the controller selects a server to respond to the client randomly. The probability that server S_i is selected is $\frac{r_i}{\sum r_i}$. A server has a low utilization will be selected by a high probability.

8.2.2 OpenFlow Based Load Balancer Implementation

The controller forwards the arriving packet try to reach the virtual host to a server based on the SBLB. All clients use the virtual IP address to access the servers. A client should send an ARP request for the virtual address before request. When an OpenFlow switch receives the ARP request, it will send a packet-in message to the controller. The controller then send an ARP reply packet through a packet-out message, which contains a virtual MAC address associated to the virtual IP address. When a request arrives at an OpenFlow switch, the controller designates one server to serve it according to the load balancing strategy, and adds a flow entry to the flow table. The flow entry will modify the request packet's destination MAC and IP address to the selected server's MAC and IP address and forward packet to the selected server. Meanwhile, it also adds a flow entry for the response of the server, which will modify the response packet's source MAC and IP address to the virtual MAC and IP address.

We show an example of the workflow in Figure 8.2. The virtual IP address of web servers is 10.0.0.1. The client send an ARP request to obtain the associated MAC address of 10.0.0.1. When the switch receives the ARP request, it sends a packet-in message to the controller. The packet-in message contains the ARP request. The controller send a packet-out message to the switch when it receives the packet-in message. The packet-out message contains an ARP reply with a virtual MAC address $e6 : 8e : f7 : fb : c1 : 55$. The switch sends the ARP reply to the client following the packet-out message. After the client receives ARP reply, it will send HTTP request using the virtual IP address and the virtual MAC address. When the switch receives the HTTP request, it sends packet-in message to the controller. The controller will select a server and install two flow entries into the switch. One flow entry modifies the destination IP address and MAC address of HTTP request to the IP address and MAC address of selected server. In this example, the switch modifies the IP address and MAC address to 10.0.0.2 and $a8 : 8e : f7 : fb : c1 : 7e$. The other flow entry modifies the source IP address and MAC address to the virtual IP address and MAC address. After

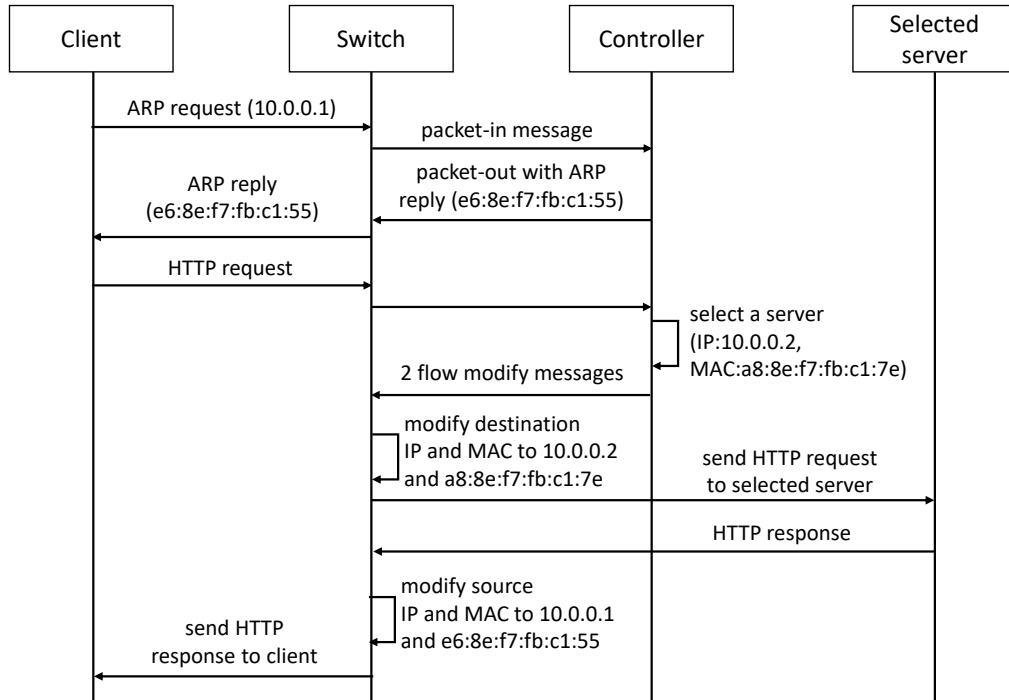


Figure 8.2: Workflow of load balancer

the two flow entries are installed, the switch modifies the destination IP address and MAC address of HTTP request and sends HTTP request to the selected server following the instructions in the first flow entry. The selected server sends the response to the switch. When the switch receives the HTTP response, it modifies the source IP address and MAC address to the virtual IP address and MAC address, and sends the response to the client following the second flow entry. Then the HTTP communication between the client and the selected server is completed.

Our load balancer consists of three modules as following: virtual machine management, load balancing policy module and Floodlight, as shown in Figure 8.3. In the experiments, we use libvirt [51] to implement virtual machine management module. This module is responsible for obtaining the running status of each virtual machine periodically, and providing the result to the load balancing policy module. The responsibility of the load balancing policy module is to select a server for the client according to the SBLB algorithm when a request arrives. Using the load information from virtual machine management module, the load balancing policy module calculates the aggregated load of each server and then selects a server based on the load balancing strategy. The selected server will process the client’s request. Floodlight module establishes a connection between the client and

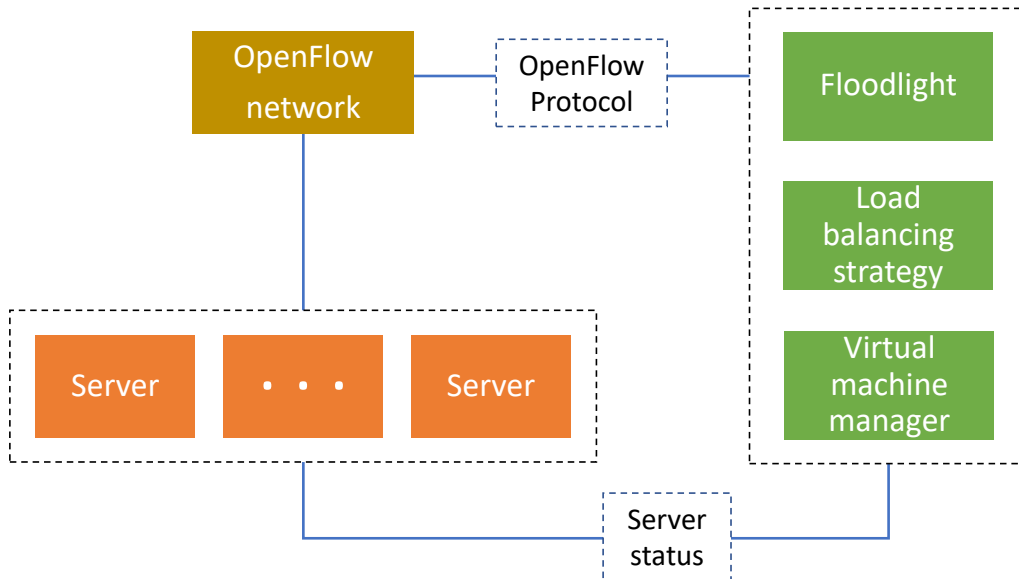


Figure 8.3: Modules of load balancer

the server by adding flow entries. When a request arrives, the Floodlight module will notify the load balancing policy module. Load balancing module will then select a server and provide the IP address of the selected server to Floodlight. The Floodlight module will establish a network connection between the server and the client so that the client and the server communicate with each other.

8.3 Experiment Results

We setup a virtualized network that contains four web servers, and one client and three OpenFlow switches. The switches are connect to a Floodlight controller. The client runs a test application that send requests to the web servers. The Floodlight controller balance the load of the web servers using three strategies, SBLB, random and round robin. We compare SBLB with, random and round robin in terms of response time, throughput, CPU utilization and memory utilization.

8.3.1 Experiment Environment

The experiments compare the performance of using different load balancing strategies in a virtualized environment. As shown in Figure 8.4, there are four virtual machines as web servers and three Open vSwitch instances as OpenFlow switches. The servers in the network map to the same virtual

address and provide the same service. Since we must guarantee the consistency of the servers' content so that they run the same application, we put the web application on the storage server and all the server connect to the storage.

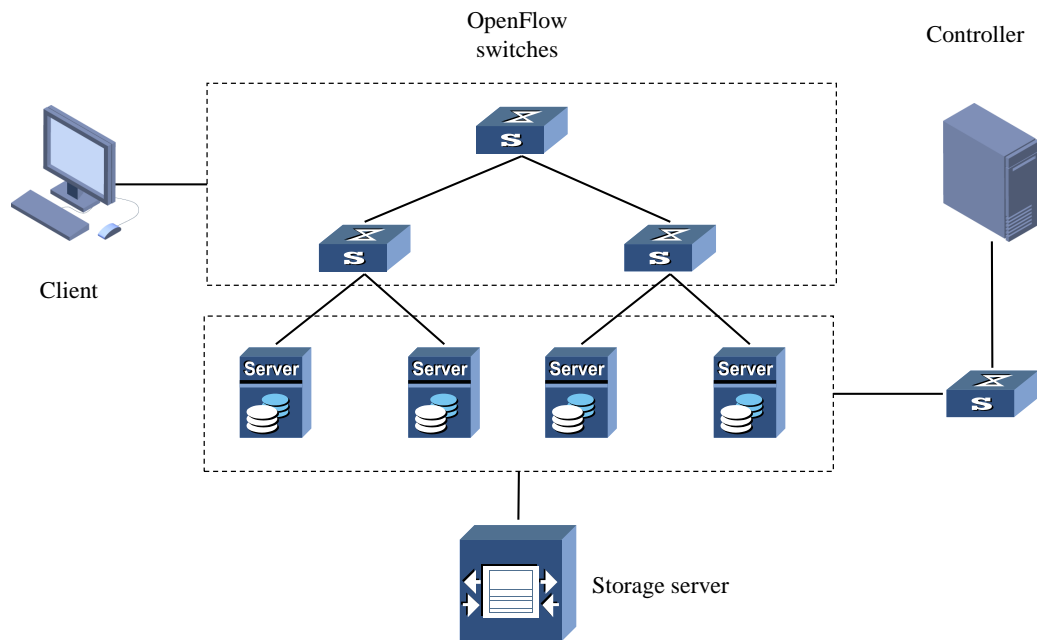


Figure 8.4: Load balancing architecture in virtualization environment

In order to store the running status of the server, we store all the results into MySQL databases that are collected by virtual machine management module. The load balancing policy module receives data from the database to calculate the current load and informs Floodlight. The Ubuntu based client runs Jmeter, a load test tool, to simulate 100 users accessing the service concurrently. And there are the same configuration and hardware for all servers in the experiment.

8.3.2 Experiment Results

Under the random strategy, the controller randomly selects a server from the server list to process the client request. Under the round robin strategy, the controller selects a server in order.

We analyze the system response time with the three different load balancing strategies, SBLB, random and round robin. Figure 8.5 shows the system response time with the three different load balancing strategies. The horizontal axis represents the number of clients accessing and the vertical axis represents the system's response time in seconds. As shown in Figure 8.5, compared to the

round robin and random, the SBLB algorithm achieves lower response time. It provides a better user experience in the experiment.

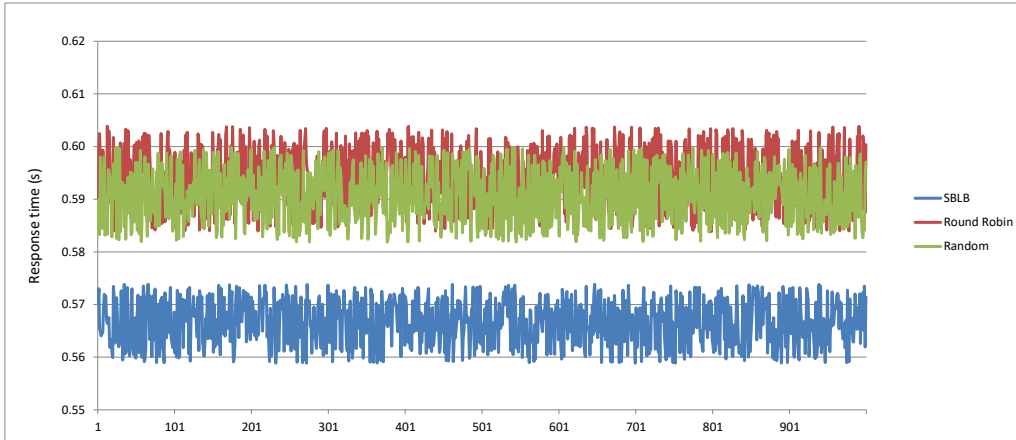


Figure 8.5: Response time with different strategies

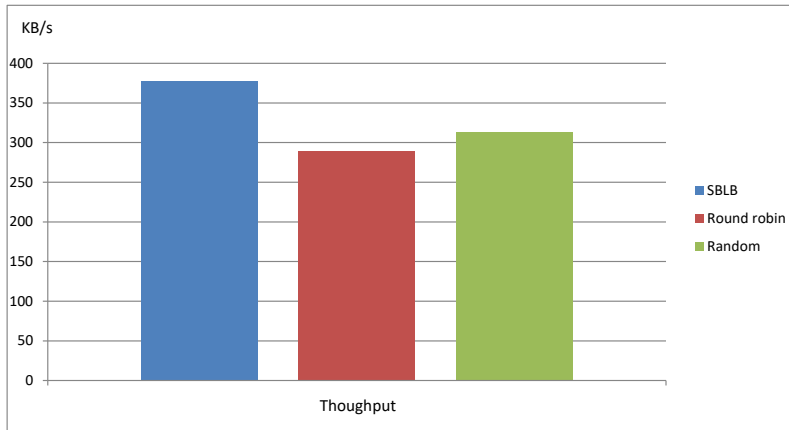
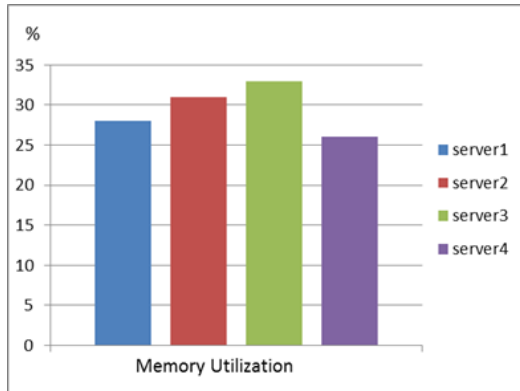


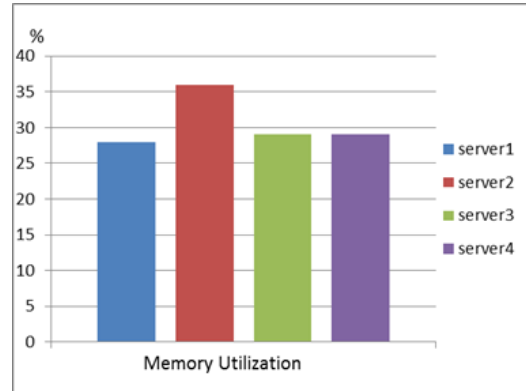
Figure 8.6: Throughput with different strategies

We analyze the throughput of the load balancing architecture when different load balancing strategies are used. Figure 8.6 shows the throughput chart of using three different load balancing strategies. The graph shows the value of the amount of received data from web server per second. As shown in Figure 8.6, the client sends about 375KB data per second when the SBLB algorithm is used. And the client sends about 300KB data when round robin or random algorithm is used. SBLB algorithm can improve the throughput of the system as compared to both other strategies.

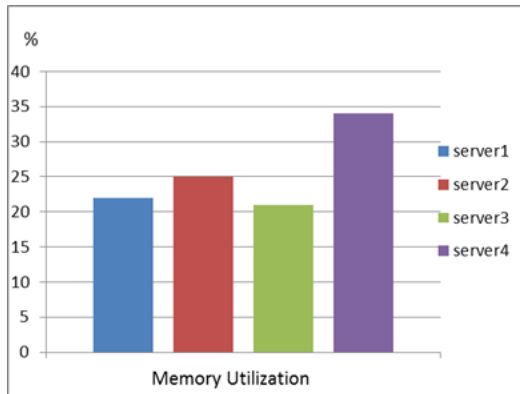
8.3. EXPERIMENT RESULTS



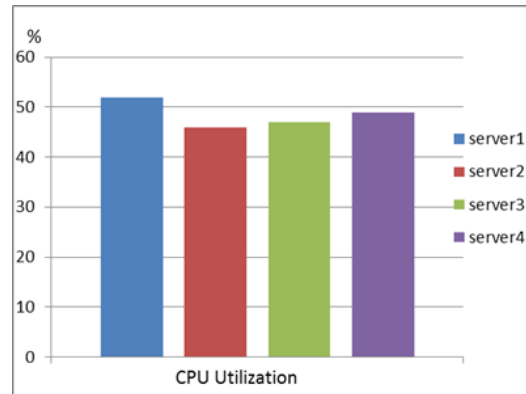
(a) Memory utilization with SBLB strategies



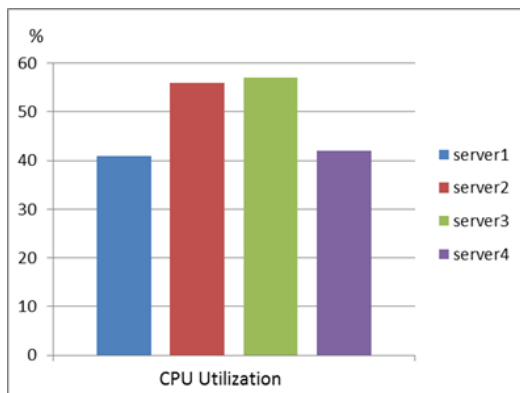
(b) Memory utilization with round robin strategies



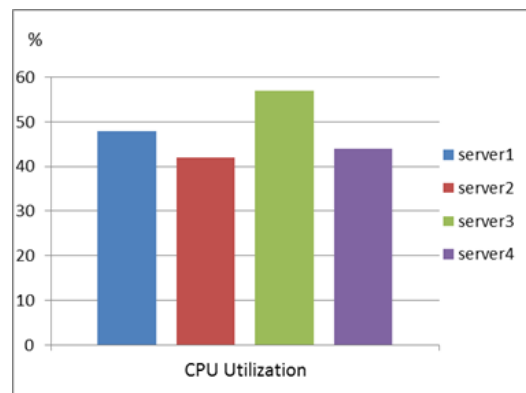
(c) Memory utilization with random strategies



(d) CPU utilization with SBLB strategies



(e) CPU utilization with round robin strategies



(f) CPU utilization with random strategies

Figure 8.7: CPU and memory utilization of each server

We also analyze the CPU and the memory utilization of each server using the three load balancing

strategies. The results are shown in Figure 8.7. Figure 8.7a shows the memory utilization of the four servers when the SBLB strategy is used. Figure 8.7b shows the memory utilization of the four servers when the round robin strategy is used. Figure 8.7c shows the memory utilization of the four servers when the random strategy is used. As can be seen from the three figures, when the SBLB strategy is used, the four servers' memory utilization are closer. When round robin or random algorithm are used, there is a gap in the memory utilization of the four servers. This means that the policy can allocate user requests more evenly, and make more effective use of memory for each server.

Figure 8.7d shows the CPU utilization of the four servers when SBLB strategy is used. Figure 8.7e shows the CPU utilization of the four servers when round robin strategy is used. Figure 8.7f shows the CPU utilization of the four servers when random strategy is used. As can be seen from the three figures, the CPU utilization of the four servers are more balanced when the SBLB algorithm is used. SBLB algorithm makes more effective utilization of CPU for each server.

SBLB can balance the memory and CPU utilization of the servers more effectively than the other algorithm because it dispatches the user requests based on the status of the servers. SBLB updates the memory and CPU utilization of the servers periodically, and calculates the rank of each server. The lower the memory and CPU utilization of a server is, the more possibility it is selected. A user request can be dispatched to a server with lower utilization.

8.4 Summary

This chapter described a dynamic load balancing method for server cluster based on OpenFlow in virtual environments to balance the load of servers in a virtualized environment. The experiments show that it is possible to make a flexible and effective load balancing in OpenFlow networks. OpenFlow provides flexibility for implementation of different load balancing strategies, which makes it convenient to use a software-defined method to use different load balancing strategies in different network environment. Experimental results show that compared with the traditional load balancing strategy, such as round robin and random, the SBLB load balancing algorithm can decrease the response time to provide a better user experience, make the utilization of server CPU and memory more efficient, and balance the load of servers more effectively.

Chapter 9

Conclusions and Outlook

9.1 Conclusions

OpenFlow is a new network architecture to overcome the shortage of traditional IP networks. The programmable controller offers high flexibility and makes network management automatic. With OpenFlow, the network operators can change the forwarding rules for traffic flows without influencing other flows. New demands can be met by updating applications running in the controller instead of changing the hardware. The controller can also provide a global view of a network and make routing decisions based on the global view. In this thesis, the main purpose has been to study the performance of the OpenFlow controllers since the controller manages all the forwarding devices in a network and it may become a bottleneck of the network. We fit PH distributions to the response time of OpenFlow controllers, and build queueing models based on the fitted distributions to model and optimize the performance of OpenFlow controllers.

PH distributions are used in performance evaluation very often since they can model the response time of many systems. If the response time of a system is correlated, MAPs are more suitable than PH distributions because PH distributions do not describe the correlation in empirical data. We implemented HyperStar2 to obtain the distribution of response time of OpenFlow controllers. HyperStar2 is a user-friendly distribution fitting tool with GUI. It helps the user, who does not understand the underlying mathematical knowledge, to fit a PH distribution or MAP to their samples. HyperStar2 uses a cluster-based algorithm. It splits samples into several clusters, fits an Erlang distribution to each cluster, and combines the Erlang distributions into a PH distribution or MAP. We provide some numerical examples to show the abilities and limitations of HyperStar2. We find that HyperStar2 can fit distributions very well but it does not provide good results if there are no obvious peaks in the samples.

There are several existing benchmark tools for OpenFlow controllers. However, they only provide the very basic performance metrics. The metrics can offer a glimpse of the performance of an OpenFlow controller but researchers need the distribution of the response time of an OpenFlow controller to build a queueing model. Queueing models can help researchers to understand the behaviors of an OpenFlow controller. We develop OFCP, a benchmark tool for OpenFlow controllers that provides the distribution of response time of an OpenFlow controller. The distribution of response time can be used in a performance model. OFCP employs a virtual OpenFlow switch to communicate with an OpenFlow controller. The virtual OpenFlow switch sends packet-in messages to an OpenFlow controller. The OpenFlow controller replies each packet-in message with a flow-modify message. OFCP records the response time of the OpenFlow controller and fits a PH distribution to the response time.

Multiple controllers are usually deployed in a large scale OpenFlow network because there are lots of requests from the switches, one controller is usually not capable. To determine the optimal number of controllers in an OpenFlow network, we evaluate the performance of multiple controllers based on a queueing model. The queueing analysis shows that the average arrival rate at a controller decreases with the number of controllers when the number of controllers is small, and the average arrival rate at a controller increases with the number of controllers when the number of controllers is large. Meanwhile, the average service rate of a controller increases with the number of the controllers, and the rate of ascent decreases with the number of controllers. We fit a PH distribution to the response time of the controller and put the distribution into a queueing model. Based on the queueing model, we can determine the optimal number of controllers that minimizes the flow setup time. The queueing analysis shows the optimal number of controllers decreases with the rate of synchronization messages and increases with the rate of packet-in messages. Therefore, we suggest to use a few powerful controller in a network if the state of the network changes frequently.

The imbalance load among multiple controllers may degrade the performance of a network. To balance the load among the controllers, we design a heuristic to solve the controller assignment problem. The heuristic starts from a feasible assignment and output an optimal assignment. We also design a greedy algorithm to generate a feasible assignment as the input of the heuristic. We model each controller as an $M/PH/1$ queue and approximate the average flow setup time of all controllers. The combination of approximated flow setup time and the number of migrations are considered as the fitness function in the heuristic. We evaluate our algorithm in Mininet and compare the algorithm with DCP-SA. The results show that our algorithm can balance the load of controller more effectively and make less switch migrations less than DCP-SA.

The burst of packet-in messages may degrade the performance of an OpenFlow network, be-

cause it increases the workload for both switches and controllers. The buffering mechanism in an OpenFlow switch can only reduce the size of a packet-in message. It can not reduce the number of packet-in messages from OpenFlow switches. Therefore, the buffering mechanism can hardly improve the performance of an OpenFlow network. We design MPT (Mismatched Packets Table), a new buffering mechanism for OpenFlow switches to avoid bursts of packet-in messages. MPT can reduce the number of packet-in messages. MPT buffers packets at a flow grain instead of packet grain. An OpenFlow switch with MPT does not send unnecessary packet-in messages to its controller. We evaluate MPT with queueing analysis and simulations. The results indicate MPT can reduce the number of packet-in messages and decrease the average flow setup time. MPT can improve the performance of an OpenFlow network effectively when the performance of the controller is not high.

Load balancing can minimize the response time of servers and improve the user experience, but traditional load balancers are expensive and not flexible. User can only choose the build-in strategies. It is difficult to add new strategies to adapt to new demands. Besides, many online services are migrated to virtualized environments, traditional load does not work well in virtualized environments. We design and implement SBLB (State-Based Load Balancing), a controller application, to balance the load of servers in virtual environments. SBLB collects server status and dispatches requests based on the load of servers. There are three modules in SBLB. Virtual machine manager gathers server status periodically. Load balancing strategy module selects a server when a request comes. Floodlight installs flow entries into switches to establish connection between the client and selected server. Our experiments show that SBLB can balance the load of servers effectively.

9.2 Outlook

Although some techniques and algorithms have been proposed in this thesis, the performance of OpenFlow controller can be further studied. We suggest several directions for future work.

The distribution of response time of an OpenFlow controller is a critical parameter in a queueing model. HyperStar2 does not provide a good fitting result if there are no obvious peaks in the histogram of samples. Improving the accuracy of HyperStar2 can help researchers to build more precise queueing models. In addition, HyperStar2 aims at helping users, who are not expert in distribution fitting, to fit a PH distribution to their samples. The number of branches is an important parameter in the fitting algorithm. It impacts the accuracy of fitting results severely. However, HyperStar2 can not determine the number of branches automatically, instead, it gets the parameter from user

input. Therefore, inexperienced users may get inaccurate fitting result. Determining the number of branches automatically can help inexperienced users and improve the accuracy of fitting result.

As we discuss in Chapter 6, the optimal number of controllers depends on the rates of flow requests and synchronization messages. The traffic load in a network may change frequently. The optimal number of controllers may change with the traffic. Since network conditions can change over time, a dynamic controller provisioning algorithm can be designed to keep OpenFlow networks always at high performance.

In the controller load balancing problem, we only consider the migrations and flow setup time. These two metrics are important but there are some others metrics, such as synchronization cost, delay between a switch and a controller. It may improve the performance of OpenFlow networks if more metrics are considered in the model. Furthermore, combining the dynamic controller provisioning problem and controller load balancing together can keep OpenFlow networks always at high performance and use as least controller as possible.

An OpenFlow controller can install flow entries in different grains. Coarse-grained flows offer less statistics information than fine-grained flows. However, coarse-grained flows can reduce the load of controllers. For example, two flow entries can match all the packets between two hosts if the two flow entries only match the source and destination MAC addresses. A trade-off can be made between fine-grained control and performance in OpenFlow network. A controller can install coarse-grained flows when it is under heavy load. Building a model for the trade-off may help controller to determine the grains of flows.

Bibliography

- [1] S. Agarwal, M. Kodialam, and T. Lakshman. Traffic engineering in software defined networks. In *2013 Proceedings IEEE INFOCOM*, pages 2211–2219. IEEE, 2013.
- [2] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou. A roadmap for traffic engineering in sdn-openflow networks. *Computer Networks*, 71:1–30, 2014.
- [3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat, et al. Hedera: dynamic flow scheduling for data center networks. In *Nsdi*, volume 10, 2010.
- [4] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the internet impasse through virtualization. *Computer*, (4):34–41, 2005.
- [5] S. Asmussen and G. Koole. Marked point processes as limits of markovian arrival streams. *Journal of Applied Probability*, 30(2):365–372, 1993.
- [6] I. I. Awan, N. Shah, M. Imran, M. Shoaib, and N. Saeed. An improved mechanism for flow rule installation in in-band sdn. *Journal of Systems Architecture*, 96:32–51, 2019.
- [7] S. Azodolmolky, R. Nejabati, M. Pazouki, P. Wieder, R. Yahyapour, and D. Simeonidou. An analytical model for software defined networking: A network calculus-based approach. In *2013 IEEE Global Communications Conference (GLOBECOM)*, pages 1397–1402. IEEE, 2013.
- [8] S. Azodolmolky, P. Wieder, and R. Yahyapour. Performance evaluation of a scalable software-defined networking deployment. In *2013 Second European Workshop on Software Defined Networks*, pages 68–74. IEEE, 2013.
- [9] M. F. Bari, A. R. Roy, S. R. Chowdhury, Q. Zhang, M. F. Zhani, R. Ahmed, and R. Boutaba. Dynamic controller provisioning in software defined networks. In *CNSM*, pages 18–25, 2013.
- [10] F. Bause, P. Buchholz, and J. Krieger. Profido-the processes fitting toolkit dortmund. In *2010 Seventh international conference on the quantitative evaluation of systems*, pages 87–96. IEEE, 2010.
- [11] A. Beben, P. Wisniewski, J. M. Batalla, and G. Xilouris. A scalable and flexible packet

BIBLIOGRAPHY

- forwarding method for future internet networks. In *2014 IEEE Global Communications Conference*, pages 1986–1992. IEEE, 2014.
- [12] F. Benamrane, R. Benaini, et al. An east-west interface for distributed sdn control plane: Implementation and evaluation. *Computers & Electrical Engineering*, 57:162–175, 2017.
- [13] F. Benamrane, M. B. Mamoun, and R. Benaini. Short: A case study of the performance of an openflow controller. In *International Conference on Networked Systems*, pages 330–334. Springer, 2014.
- [14] T. Benson, A. Akella, and D. A. Maltz. Unraveling the complexity of network management. In *NSDI*, pages 335–348, 2009.
- [15] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.
- [16] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O’Connor, P. Radoslavov, W. Snow, et al. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.
- [17] M. Bertoli, G. Casale, and G. Serazzi. Jmt: Performance engineering tools for system modeling. 36:10–15, 01 2009.
- [18] G. Bolch, S. Greiner, H. De Meer, and K. S. Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.
- [19] L. Breuer. An em algorithm for batch markovian arrival processes and its comparison to a simpler estimation procedure. *Annals of Operations Research*, 112(1-4):123–138, 2002.
- [20] P. Buchholz. An em-algorithm for map fitting from real traffic data. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 218–236. Springer, 2003.
- [21] P. Buchholz and J. Kriege. A heuristic approach for fitting maps to moments and joint moments. In *2009 Sixth international conference on the quantitative evaluation of systems*, pages 53–62. IEEE, 2009.
- [22] P. Buchholz and A. Panchenko. A two-step em algorithm for map fitting. In *International Symposium on Computer and Information Sciences*, pages 217–227. Springer, 2004.
- [23] CAIDA. Passive network monitors from caida. <http://www.caida.org/data/realtime/passive/?monitor=equinix-chicago-dirB>, 2018. [Online].
- [24] V. Cardellini, M. Colajanni, and P. S. Yu. Dynamic load balancing on web-server systems. *IEEE Internet computing*, 3(3):28–39, 1999.

- [25] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Protocol operations for version 2 of the simple network management protocol (snmpv2). Technical report, 1993.
- [26] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin. Simple network management protocol. Technical report, 1988.
- [27] M. Cello, Y. Xu, A. Walid, G. Wilfong, H. J. Chao, and M. Marchese. Balcon: A distributed elastic sdn control via efficient switch migration. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 40–50. IEEE, 2017.
- [28] W. Chen, Z. Shang, X. Tian, and H. Li. Dynamic server cluster load balancing in virtualization environment with openflow. *International Journal of Distributed Sensor Networks*, 11(7):531538, 2015.
- [29] W. Chen, X. Tian, and Z. Shang. Design of scalable control plane via multiple controllers. In *Frontier Computing*, pages 441–449. Springer, 2016.
- [30] G. Cheng, H. Chen, Z. Wang, and S. Chen. Dha: Distributed decisions on the switch migration toward a scalable sdn control plane. In *2015 IFIP Networking Conference (IFIP Networking)*, pages 1–9. IEEE, 2015.
- [31] B. Choi, D. Choi, Y. Lee, and D. Sung. Priority queueing system with fixed-length packet-train arrivals. *IEE Proceedings-Communications*, 145(5):331–336, 1998.
- [32] M.-J. Choi, H.-M. Choi, J. W. Hong, and H.-T. Ju. Xml-based configuration management for ip network devices. *IEEE Communications Magazine*, 42(7):84–91, 2004.
- [33] Cisco. The zettabyte era: Trends and analysis. <https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html>, 2018. [Online].
- [34] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 254–265. ACM, 2011.
- [35] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an elastic distributed sdn controller. *ACM SIGCOMM computer communication review*, 43(4):7–12, 2013.
- [36] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. Network configuration protocol (netconf). Technical report, 2011.
- [37] D. Erickson. The beacon openflow controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 13–18. ACM, 2013.
- [38] A. Filali, A. Kobbane, M. Elmachkour, and S. Cherkaoui. Sdn controller assignment and load balancing with minimum quota of processing capacity. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2018.

BIBLIOGRAPHY

- [39] D. Firestone. Vfp: A virtual switch platform for host {SDN} in the public cloud. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 315–328, 2017.
- [40] C. H. Foh and M. Zukerman. Performance analysis of the ieee 802.11 mac protocol. In *Proceedings of European Wireless*, volume 2, 2002.
- [41] T. O. N. Foundation. Openflow specification 1.4.1. <https://www.opennetworking.org/sdn-resources/technical-library>, 2015. [Online].
- [42] V. S. Frost and B. Melamed. Traffic modeling for telecommunications networks. *IEEE Communications Magazine*, 32(3):70–81, 1994.
- [43] S. Gebert, R. Pries, D. Schlosser, and K. Heck. Internet access traffic measurement and analysis. In *International Workshop on Traffic Monitoring and Analysis*, pages 29–42. Springer, 2012.
- [44] A. Ghodsi, S. Shenker, T. Koponen, A. Singla, B. Raghavan, and J. Wilcox. Intelligent design enables architectural evolution. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, page 3. ACM, 2011.
- [45] H. Gong, R. Li, Y. Bai, J. An, and K. Li. Message response time analysis for automotive cyber–physicalsystems with uncertain delay: An m/ph/1 queue approach. *Performance Evaluation*, 125:21–47, 2018.
- [46] N. Gray, T. Zinner, S. Gebert, and P. Tran-Gia. Simulation framework for distributed sdn-controller architectures in omnet++. In *International Conference on Mobile Networks and Management*, pages 3–18. Springer, 2016.
- [47] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.
- [48] Z. Guo, Y. Xu, M. Cello, J. Zhang, Z. Wang, M. Liu, and H. J. Chao. Jumpflow: Reducing flow table usage in software-defined networks. *Computer Networks*, 92:300–315, 2015.
- [49] M. Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.
- [50] S. Hassas Yeganeh and Y. Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 19–24. ACM, 2012.
- [51] R. Hat. Libvirt virtualization api, 2014.
- [52] B. Heller, R. Sherwood, and N. McKeown. The controller placement problem. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 7–12. ACM, 2012.

- [53] J. Hu, C. Lin, X. Li, and J. Huang. Scalability of control planes for software defined networks: Modeling and evaluation. In *2014 IEEE 22nd International Symposium of Quality of Service (IWQoS)*, pages 147–152. IEEE, 2014.
- [54] Y. Hu, W. Wang, X. Gong, X. Que, and S. Cheng. Balanceflow: controller load balancing for openflow networks. In *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*, volume 2, pages 780–785. IEEE, 2012.
- [55] Y. Hu, W. Wang, X. Gong, X. Que, and S. Cheng. On reliability-optimized controller placement for software-defined networks. *China Communications*, 11(2):38–54, 2014.
- [56] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.
- [57] M. Jarschel, F. Lehrieder, Z. Magyari, and R. Pries. A flexible openflow-controller benchmark. In *2012 European Workshop on Software Defined Networking*, pages 48–53. IEEE, 2012.
- [58] M. Jarschel, C. Metter, T. Zinner, S. Gebert, and P. Tran-Gia. Ofcprobe: A platform-independent tool for openflow controller analysis. In *2014 IEEE Fifth International Conference on Communications and Electronics (ICCE)*, pages 182–187. IEEE, 2014.
- [59] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia. Modeling and performance evaluation of an openflow architecture. In *Proceedings of the 23rd international teletraffic congress*, pages 1–7. International Teletraffic Congress, 2011.
- [60] U. Javed, A. Iqbal, S. Saleh, S. A. Haider, and M. U. Ilyas. A stochastic model for transit latency in openflow sdns. *Computer Networks*, 113:218–229, 2017.
- [61] X. Jin, Y. Li, D. Wei, S. Li, J. Gao, L. Xu, G. Li, W. Xu, and J. Rexford. Optimizing bulk transfers with software-defined optical wan. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 87–100. ACM, 2016.
- [62] S. A. Jyothi, M. Dong, and P. Godfrey. Towards a flexible data center fabric with source routing. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 10. ACM, 2015.
- [63] 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*, pages 202–208. ACM, 2009.
- [64] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of the Symposium on SDN Research*, page 6. ACM, 2016.

BIBLIOGRAPHY

- [65] S. Kaur, J. Singh, and N. S. Ghumman. Network programmability using pox controller. In *ICCCS International Conference on Communication, Computing & Systems, IEEE*, volume 138, 2014.
- [66] J. Kennedy and R. Eberhart. Particle swarm optimization (psa). In *Proc. IEEE International Conference on Neural Networks, Perth, Australia*, pages 1942–1948, 1995.
- [67] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.
- [68] A. Koshibe, A. Baid, and I. Seskar. Towards distributed hierarchical sdn control plane. In *2014 International Science and Technology Conference (Modern Networking Technologies)(MoNeTeC)*, pages 1–5. IEEE, 2014.
- [69] D. Kotani and Y. Okabe. Packet-in message control for reducing cpu load and control traffic in openflow switches. In *2012 European Workshop on Software Defined Networking*, pages 42–47. IEEE, 2012.
- [70] A. Krishnamurthy, S. P. Chandrabose, and A. Gember-Jacobson. Pratyastha: an efficient elastic distributed sdn control plane. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 133–138. ACM, 2014.
- [71] S. Kukliński. Programmable management framework for evolved sdn. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–8. IEEE, 2014.
- [72] Y. Kyung, K. Hong, T. M. Nguyen, S. Park, and J. Park. A load distribution scheme over multiple controllers for scalable sdn. In *2015 Seventh International Conference on Ubiquitous and Future Networks*, pages 808–810. IEEE, 2015.
- [73] N. labs. Trema controller. <https://trema.github.io/trema/>, 2018. [Online].
- [74] J. H. Lam, S.-G. Lee, H.-J. Lee, and Y. E. Oktian. Tls channel implementation for onos’s east/west-bound communication. In *Electronics, Communications and Networks V*, pages 397–403. Springer, 2016.
- [75] B. Lee, S. H. Park, J. Shin, and S. Yang. Iris: the openflow-based recursive sdn controller. In *16th International Conference on Advanced Communication Technology*, pages 1227–1231. IEEE, 2014.
- [76] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically centralized?: state distribution trade-offs in software defined networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 1–6. ACM, 2012.
- [77] S. Li, K. Han, N. Ansari, Q. Bao, D. Hu, J. Liu, S. Yu, and Z. Zhu. Improving sdn scalability with protocol-oblivious source routing: A system-level study. *IEEE Transactions on Network*

- and Service Management*, 15(1):275–288, 2017.
- [78] H. Liaoruo, S. Qingguo, and S. Wenjuan. A source routing based link protection method for link failure in sdn. In *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, pages 2588–2594. IEEE, 2016.
- [79] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter. Traffic engineering with forward fault correction. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 527–538. ACM, 2014.
- [80] S. Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.
- [81] D. M. Lucantoni, K. S. Meier-Hellstern, and M. F. Neuts. A single-server queue with server vacations and a class of non-renewal arrival processes. *Advances in Applied Probability*, 22(3):676–705, 1990.
- [82] K. Mahmood, A. Chilwan, O. Østerbø, and M. Jarschel. Modelling of openflow-based software-defined networks: the multiple node case. *IET Networks*, 4(5):278–284, 2015.
- [83] K. Mahmood, A. Chilwan, O. N. Østerbø, and M. Jarschel. On the modeling of openflow-based sdns: The single node case. *arXiv preprint arXiv:1411.4733*, 2014.
- [84] J. Mao, B. Han, Z. Sun, X. Lu, and Z. Zhang. Efficient mismatched packet buffer management with packet order-preserving for openflow networks. *Computer Networks*, 110:91–103, 2016.
- [85] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.
- [86] J. Medved, R. Varga, A. Tkacik, and K. Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, pages 1–6. IEEE, 2014.
- [87] L. J. V. Miranda et al. Pyswarms: a research toolkit for particle swarm optimization in python. *J. Open Source Software*, 3(21):433, 2018.
- [88] J. Moy. Ospf version 2. Technical report, 1997.
- [89] B. S. Networks. Floodlight. <http://www.projectfloodlight.org/floodlight/>, 2018. [Online].
- [90] B. S. Networks. Openflowj. <https://github.com/floodlight/loxigen/wiki/OpenFlowJ-Loxi>, 2018. [Online].
- [91] M. F. Neuts. A versatile markovian point process. *Journal of Applied Probability*, 16(4):764–779, 1979.
- [92] M. F. Neuts. *Matrix-geometric solutions in stochastic models: an algorithmic approach*.

BIBLIOGRAPHY

- Courier Corporation, 1994.
- [93] P. Neves, R. Calé, M. R. Costa, C. Parada, B. Parreira, J. Alcaraz-Calero, Q. Wang, J. Nightingale, E. Chirivella-Perez, W. Jiang, et al. The selfnet approach for autonomic management in an nfv/sdn networking paradigm. *International Journal of Distributed Sensor Networks*, 12(2):2897479, 2016.
 - [94] V.-G. Nguyen and Y.-H. Kim. Sdn-based enterprise and campus networks: A case of vlan management. *Journal of Information Processing Systems*, 12(3), 2016.
 - [95] NTT. Ryu controller. <https://osrg.github.io/ryu/>, 2018. [Online].
 - [96] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turetli. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials*, 16(3):1617–1634, 2014.
 - [97] Y. E. Oktian, S. Lee, H. Lee, and J. Lam. Distributed sdn controller system: A survey on design choice. *computer networks*, 121:100–111, 2017.
 - [98] S. Panchen, P. Phaal, and N. McKee. Inmon corporation’s sflow: A method for monitoring traffic in switched and routed networks. 2001.
 - [99] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, et al. The design and implementation of open vswitch. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 117–130, 2015.
 - [100] K. Phemius, M. Bouet, and J. Leguay. Disco: Distributed multi-domain sdn controllers. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–4. IEEE, 2014.
 - [101] K. Phemius and M. B. Thales. Openflow: Why latency does matter. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 680–683. IEEE, 2013.
 - [102] K. G. Provan and P. Kenis. Modes of network governance: Structure, management, and effectiveness. *Journal of public administration research and theory*, 18(2):229–252, 2008.
 - [103] M. Qilin and S. Weikang. A load balancing method based on sdn. In *2015 Seventh International Conference on Measuring Technology and Mechatronics Automation*, pages 18–21. IEEE, 2015.
 - [104] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, and S. Shenker. Software-defined internet architecture: decoupling architecture from infrastructure. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 43–48. ACM, 2012.
 - [105] R. M. Ramos, M. Martinello, and C. E. Rothenberg. Slickflow: Resilient source routing

- in data center networks unlocked by openflow. In *38Th annual IEEE conference on local computer networks*, pages 606–613. IEEE, 2013.
- [106] H. K. Rath, V. Revoori, S. M. Nadaf, and A. Simha. Optimal controller placement in software defined networks (sdn) using a non-zero-sum game. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, pages 1–6. IEEE, 2014.
- [107] P. Reinecke, T. Krauß, and K. Wolter. Phase-Type Fitting Using HyperStar. In *Computer Performance Engineering*, pages 164–175. 2013.
- [108] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. Oflops: An open framework for openflow switch evaluation. In *International Conference on Passive and Active Network Measurement*, pages 85–95. Springer, 2012.
- [109] S. Rowshanrad, S. Namvarasl, V. Abdi, M. Hajizadeh, and M. Keshtgary. A survey on sdn, the future of networking. *Journal of Advanced Computer Science & Technology*, 3(2):232, 2014.
- [110] A. Roy, D. Bansal, D. Brumley, H. K. Chandrappa, P. Sharma, R. Tewari, B. Arzani, and A. C. Snoeren. Cloud datacenter sdn monitoring: Experiences and challenges. In *Proceedings of the Internet Measurement Conference 2018*, pages 464–470. ACM, 2018.
- [111] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network’s (data-center) network. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 123–137. ACM, 2015.
- [112] A. Sallahi and M. St-Hilaire. Optimal model for the controller placement problem in software defined networks. *IEEE communications letters*, 19(1):30–33, 2014.
- [113] S. Schmid and J. Suomela. Exploiting locality in distributed sdn control. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 121–126. ACM, 2013.
- [114] R. Schoonderwoerd, O. E. Holland, J. L. Bruten, and L. J. Rothkrantz. Ant-based load balancing in telecommunications networks. *Adaptive behavior*, 5(2):169–207, 1997.
- [115] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky. Advanced study of sdn/openflow controllers. In *Proceedings of the 9th central & eastern european software engineering conference in russia*, page 1. ACM, 2013.
- [116] Z. Shang, T. Meng, and K. Wolter. Hyperstar2: Easy distribution fitting of correlated data. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 139–142. ACM, 2017.
- [117] Z. Shang and K. Wolter. Delay evaluation of openflow network based on queueing model.

BIBLIOGRAPHY

- arXiv preprint arXiv:1608.06491*, 2016.
- [118] Z. Shang, H. Wu, G. Peng, and K. Wolter. Dynamic load balancing in the control plane of software-defined networks. In *19th IEEE International Conference on Communication Technology*. IEEE, 2019. [Accepted].
- [119] Z. Shang, H. Wu, and K. Wolter. An openflow controller performance evaluation tool. In *European Workshop on Performance Engineering*, pages 235–249. Springer, 2018.
- [120] Z. Shang, H. Wu, and K. Wolter. Buffer management for reducing packet-in messages in openflow networks. In *12th International Conference on Communication Software and Networks*. IEEE, 2019. [Accepted].
- [121] Z. Shang, H. Wu, and K. Wolter. Performance evaluation of the control plane in software defined networks. In *Proceedings of the 12th EAI International Conference on Performance Evaluation Methodologies and Tools*, pages 171–174. ACM, 2019.
- [122] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 1:132, 2009.
- [123] R. Sherwood and Y. KOK-KIONG. Cbench: an open-flow controller benchmarker. *URL <http://archive.openflow.org/wk/index.php/Oflows>*, 2010.
- [124] R. Sherwood and K. Yap. Cbench controller benchmarker. *Last accessed, Nov*, 2011.
- [125] S. Shirali-Shahreza and Y. Ganjali. Rewiflow: Restricted wildcard openflow rules. *ACM SIGCOMM Computer Communication Review*, 45(5):29–35, 2015.
- [126] T. SimPy. Simpy:discrete event simulation for python. <https://simpy.readthedocs.io/en/latest/index.html>, 2018. [Online].
- [127] M. Soliman, B. Nandy, I. Lambadaris, and P. Ashwood-Smith. Source routed forwarding with software defined control, considerations and implications. In *Proceedings of the 2012 ACM conference on CoNEXT student workshop*, pages 43–44. ACM, 2012.
- [128] M. Soliman, B. Nandy, I. Lambadaris, and P. Ashwood-Smith. Exploring source routed forwarding in sdn-based wans. In *2014 IEEE International Conference on Communications (ICC)*, pages 3070–3075. IEEE, 2014.
- [129] P. Song, Y. Liu, T. Liu, and D. Qian. Flow stealer: lightweight load balancing by stealing flows in distributed sdn controllers. *Science China Information Sciences*, 60(3):032202, 2017.
- [130] M. Team. Mininet. <http://mininet.org/>, 2018. [Online].
- [131] M. Telek and G. Horváth. A minimal representation of markov arrival processes and a moments matching method. *Performance Evaluation*, 64(9-12):1153–1168, 2007.
- [132] Z. Teo, K. Birman, and R. Van Renesse. Experience with 3 sdn controllers in an enterprise

- setting. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, pages 97–104. IEEE, 2016.
- [133] C. Thorpe, C. Olariu, A. Hava, and P. McDonagh. Experience of developing an openflow sdn prototype for managing iptv networks. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 966–971. IEEE, 2015.
- [134] A. Tootoonchian and Y. Ganjali. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, volume 3, 2010.
- [135] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On controller performance in software-defined networks. In *Presented as part of the 2nd {USENIX} Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, 2012.
- [136] M. Trevisan, I. Drago, M. Mellia, H. H. Song, and M. Baldi. Awesome: Big data for automatic web service management in sdn. *IEEE Transactions on Network and Service Management*, 15(1):13–26, 2017.
- [137] D. Tuncer, M. Charalambides, S. Clayman, and G. Pavlou. On the placement of management and control functionality in software defined networks. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 360–365. IEEE, 2015.
- [138] D. Tuncer, M. Charalambides, S. Clayman, and G. Pavlou. Flexible traffic splitting in open-flow networks. *IEEE Transactions on Network and Service Management*, 13(3):407–420, 2016.
- [139] M. T. I. ul Huque, W. Si, G. Jourjon, and V. Gramoli. Large-scale dynamic controller placement. *IEEE Transactions on Network and Service Management*, 14(1):63–76, 2017.
- [140] A. Varga. Omnet++. In *Modeling and tools for network simulation*, pages 35–59. Springer, 2010.
- [141] C. Wang, B. Hu, S. Chen, D. Li, and B. Liu. A switch migration-based decision-making scheme for balancing load in sdn. *IEEE Access*, 5:4537–4544, 2017.
- [142] R. Wang, D. Butnariu, J. Rexford, et al. Openflow-based server load balancing gone wild. *Hot-ICE*, 11:12–12, 2011.
- [143] T. Wang, F. Liu, J. Guo, and H. Xu. Dynamic sdn controller assignment in data center networks: Stable matching with transfers. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.
- [144] Y. Wang and I. Matta. Sdn management layer: Design requirements and future direction. In *2014 IEEE 22nd International Conference on Network Protocols*, pages 555–562. IEEE,

BIBLIOGRAPHY

- 2014.
- [145] J. A. Wickboldt, W. P. De Jesus, P. H. Isolani, C. B. Both, J. Rochol, and L. Z. Granville. Software-defined networking: management requirements and challenges. *IEEE Communications Magazine*, 53(1):278–285, 2015.
 - [146] C. Williamson. Internet traffic measurement. *IEEE Internet Computing*, 5(6):70–74, 2001.
 - [147] Y. Wu, G. Min, and L. T. Yang. Performance analysis of hybrid wireless networks under bursty and correlated traffic. *IEEE Transactions on Vehicular Technology*, 62(1):449–454, 2012.
 - [148] B. Xiong, K. Yang, J. Zhao, W. Li, and K. Li. Performance evaluation of openflow-based software-defined networks based on queueing model. *Computer Networks*, 102:172–185, 2016.
 - [149] G. Yao, J. Bi, Y. Li, and L. Guo. On the capacitated controller placement problem in software defined networks. *IEEE Communications Letters*, 18(8):1339–1342, 2014.
 - [150] L. Yao, P. Hong, W. Zhang, J. Li, and D. Ni. Controller placement and flow based dynamic management problem towards sdn. In *2015 IEEE International Conference on Communication Workshop (ICCW)*, pages 363–368. IEEE, 2015.
 - [151] L. Yao, P. Hong, and W. Zhou. Evaluating the controller capacity in software defined networking. In *2014 23rd International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6. IEEE, 2014.
 - [152] A. Yassine, H. Rahimi, and S. Shirmohammadi. Software defined network traffic measurement: Current trends and challenges. *IEEE Instrumentation & Measurement Magazine*, 18(2):42–50, 2015.
 - [153] V. Yazici, M. O. Sunay, and A. O. Ercan. Controlling a software-defined network via distributed controllers. *arXiv preprint arXiv:1401.7651*, 2014.
 - [154] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali. On scalability of software-defined networking. *IEEE Communications Magazine*, 51(2):136–141, 2013.
 - [155] J. Yu, Y. Wang, K. Pei, S. Zhang, and J. Li. A load balancing mechanism for multiple sdn controllers based on load informing strategy. In *2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 1–4. IEEE, 2016.
 - [156] S. Zerkane, D. Espes, P. Le Parc, and F. Cuppens. Software defined networking reactive stateful firewall. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 119–132. Springer, 2016.
 - [157] B. Zhang, X. Wang, and M. Huang. Dynamic controller assignment problem in software-defined networks. *Transactions on Emerging Telecommunications Technologies*,

- 29(8):e3460, 2018.
- [158] H. Zhong, Y. Fang, and J. Cui. Lbbsrt: An efficient sdn load balancing scheme based on server response time. *Future Generation Computer Systems*, 68:183–190, 2017.
- [159] Y. Zhou, M. Zhu, L. Xiao, L. Ruan, W. Duan, D. Li, R. Liu, and M. Zhu. A load balancing strategy of sdn controller based on distributed decision. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 851–856. IEEE, 2014.

BIBLIOGRAPHY

List of Figures

1.1	The architecture of SDN	4
1.2	The difference between traditional switches and SDN switches	5
2.1	The main components of an OpenFlow switch	14
2.2	Packet processing in an OpenFlow switch	15
2.3	The model of horizontal architecture	18
2.4	The model of hierarchical architecture	19
2.5	A CTMC with one absorbing state	21
2.6	CTMC representation of an Erlang distribution	23
2.7	CTMC representation of a hyper-Erlang distribution	24
2.8	CTMC representation of a MAP	25
2.9	$M/PH/1$ queueing model	26
4.1	User interface of HyperStar2 with fitted result	50
4.2	The pdf of the first example	53
4.3	Correlation of the first example	54
4.4	The pdf of the second example	55
4.5	Correlation of the second example	55
5.1	A benchmark result from Cbench	58
5.2	The architecture of OFCP	60
5.3	The measurement process	63
5.4	The response time of Ryu controller	63
5.5	The pdf of fitted distribution	64
6.1	Flow setup process under multiple controllers	72

LIST OF FIGURES

6.2	The queueing model for flow installation	73
6.3	The architecture of the prototype	75
6.4	The single controller response time	76
6.5	The multiple controllers response time	77
6.6	Arrival rate and service rate	78
6.7	Flow setup time with different packet-in rate	79
6.8	Flow setup time with different synchronization rate	79
6.9	The optimal number of controllers	80
6.10	OpenFlow with multiple controllers	83
6.11	Queueing model of a controller	84
6.12	Case 1: generated traffic	90
6.13	Case 2: generated traffic	90
6.14	mean flow setup time in case 1	91
6.15	mean flow setup time in case 2	91
6.16	Number of migrations in case 1	92
6.17	Number of migrations in case 2	92
6.18	Case 1: CV of utilization	93
6.19	Case 2: CV of utilization	94
7.1	Topology for burst messages test	98
7.2	Burst of packet-in on the controller	99
7.3	Overview of the MPT	99
7.4	Workflow of MPT	100
7.5	Burst packets forwarding process with general switch	101
7.6	The queueing model of the controller	102
7.7	Burst packets forwarding process with MPT	105
7.8	The queueing model of the controller	105
7.9	Response time with different batch sizes	107
7.10	The mean response time with different service rates	107
7.11	The mean response time with batch arrival rates	108
7.12	The response time of the controller	109
7.13	The number of packet-in messages	110
7.14	Number of packet-in message with general switch simulation	111
7.15	Number of packet-in message with MPT switch simulation	111

8.1	Load balance architecture based on OpenFlow	114
8.2	Workflow of load balancer	117
8.3	Modules of load balancer	118
8.4	Load balancing architecture in virtualization environment	119
8.5	Response time with different strategies	120
8.6	Throughput with different strategies	120
8.7	CPU and memory utilization of each server	121

LIST OF FIGURES

List of Tables

2.1	The components of a flow entry	14
2.2	The main components of a packet-in message	15
2.3	compare controllers	17
2.4	The main components of a flow-modify message	17
3.1	Models of OpenFlow networks	33
3.2	Models of OpenFlow networks	34
4.1	Fitting parameters	50
5.1	The detailed configuration	62
5.2	The parameters of the Erlang branches	64
6.1	Distribution of single controller response time	76
6.2	Distribution of multiple controllers response time	77

LIST OF TABLES

About the Author

Zhihao Shang received his B.S. degree from Henan University of Chinese Medicine, Zhengzhou, in 2010 and M.S degree from University of Lanzhou, Lanzhou, in 2014. Supported by China Scholarship Council, he joined in Dependable Distributed Systems at Freie Universität Berlin, supervised by Prof. Dr. Katinka Wolter since 2015. His research interests include performance evaluation, software-defined networking and queueing theory.

