Doctoral Dissertation

# Efficient Control Plane Management for Software-Defined Networks

Woojoong Kim (김 우 중)

Department of Computer Science and Engineering

Pohang University of Science and Technology

2019




POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

# 소프트웨어 정의 네트워크를 위한 효율적인 제어 평면 관리

# Efficient Control Plane Management for Software-Defined Networks

# Efficient Control Plane Management for Software-Defined Networks

by

Woojoong Kim
Department of Computer Science and Engineering
Pohang University of Science and Technology

A dissertation submitted to the faculty of the Pohang University of Science and Technology in partial fulfillment of the requirements for the degree of Doctor of philosophy in the Computer Science and Engineering

Pohang, Korea
December. 26. 2018
Approved by
Young-Joo Suh (Signature)
Academic advisor

# Efficient Control Plane Management for Software-Defined Networks

Woojoong Kim

The undersigned have examined this dissertation and hereby certify that it is worthy of acceptance for a doctoral degree from POSTECH

December. 26. 2018

| | | |
|---|---|---|
| Committee Chair | Young-Joo Suh | (Seal) |
| Member | Sunggu Lee | (Seal) |
| Member | James Won-Ki Hong | (Seal) |
| Member | Jae-Hyoung Yoo | (Seal) |
| Member | Gwangsun Kim | (Seal) |

DCSE 김 우 중. Woojoong Kim
20120737
Efficient Control Plane Management for Software-Defined Networks,
소프트웨어 정의 네트워크를 위한 효율적인 제어 평면 관리
Department of Computer Science and Engineering, 2019, 106p, Advisor: Young-Joo Suh. Text in English.

# ABSTRACT

The Elastic Control Plane (ECP) is an efficient way to control large-scale software-defined networks which experience the rapid control traffic fluctuation. Proposed distributed control planes have utilized the static number of controllers to manage innumerable network devices and user data flows. With a surge or plunge in the control traffic load, the Control Plane (CP) utilizes insufficient or dissipative controllers, respectively. Consequently, the CP delays the rule installation time or squanders the CPU resources. In order to solve those problems, the ECP adjusts the number of active controllers according to the control traffic load.

However, existing ECPs have suffered from the immediacy and the computing overhead problems. Indeed, existing ECPs can be categorized into two types – the first type of ECPs (ECP-1) powers on all active and even inactive controllers; the second type of ECPs (ECP-2) shuts down all inactive controllers. Due to all controllers in operation, ECP-1 encounters the high computing overhead. In contrast, ECP-2 initially switches on an inactive controller for activation, which

causes the low immediacy.

In this dissertation, we propose two new ECPs, a Hybrid ECP (H-ECP) and a Threshold-based Dynamic Controller Resource Allocation (T-DCORAL). To begin with, H-ECP switches off inactive controllers, except for standby controllers which are inactive controllers powered on. When the CP exploits insufficient active controllers, a standby controller is activated. On the other hand, H-ECP transits an active controller to a standby controller if the CP overuses active controllers. To maintain the predefined number of standby controllers, H-ECP runs the background thread which manages the power of inactive and standby controllers. As a result, H-ECP reduces the computing overhead by switching off unnecessary controllers. Also, H-ECP increases the immediacy since there is no need to boot up inactive controllers for activation. Meanwhile, T-DCORAL deals with the control traffic fluctuation by dynamically allocating virtual CPUs (vCPUs) to each controller in runtime. T-DCORAL only takes advantage of minimal controllers to reduce the computing overhead. Furthermore, T-DCORAL never switches on or off controllers for improving the immediacy.

For the performance evaluation, we used ONOS the well-known controller and Mininet to emulate software-defined networks. As a result, H-ECP and T-DCORAL outperform proposed ECPs in terms of the CPU usage and the network bandwidth. Moreover, both ECPs achieves higher immediacy and faster rule installation time than previous ECPs.


POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

# Contents

List of Tables . . . . . V
List of Figures . . . . . VI
List of Algorithms . . . . . VIII

**I. Introduction** **1**

**II. Background and Related Work** **7**
2.1 Software-Defined Networking . . . . . 7
2.2 OpenFlow . . . . . 10
2.3 Roles of Controllers in OpenFlow 1.2 and Beyond . . . . 12
2.4 Distributed Control Plane . . . . . 14
2.5 Static Control Plane . . . . . 15
2.6 Elastic Control Plane . . . . . 17
2.7 Data Center Infrastructure . . . . . 21

**III. Immediacy and Computing Overhead Analysis** **24**
3.1 Types of ECP . . . . . 24
3.2 What is the Immediacy and Computing Overhead? . . . 26
3.3 Immediacy Analysis . . . . . 27
3.4 Computing Overhead Analysis . . . . . 30
3.5 Discussion . . . . . 32

**IV. Proposed Schemes: H-ECP and T-DCORAL** **33**
4.1 System Model . . . . . 33
4.2 Problem Formulation and Goal . . . . . 37
4.3 Hybrid ECP . . . . . 38

4.3.1 Overview . . . . . 38
4.3.2 Detailed Description of H-ECP . . . . . 40
4.3.3 Challenges . . . . . 51
4.3.4 An Operational Example of H-ECP . . . . . 53
4.4 Threshold-based Dynamic Controller Resource Allocation 55
4.4.1 Overview . . . . . 55
4.4.2 Detailed Description of T-DCORAL . . . . . 56
4.4.3 Challenges . . . . . 59
4.4.4 An Operational Example of T-DCORAL . . . . . 64
4.5 Implementation . . . . . 66
4.5.1 Overview . . . . . 66
4.5.2 Requirement List . . . . . 68
4.5.3 Design of the Orchestrator . . . . . 69
4.5.4 Design of OFMon . . . . . 73

**V. Performance evaluation** **76**
5.1 Evaluation of H-ECP and T-DCORAL . . . . . 76
5.1.1 Environment . . . . . 76
5.1.2 Results . . . . . 78
5.2 Evaluation of the deployment scheme of network devices in T-DCORAL . . . . . 88
5.2.1 Environment . . . . . 88
5.2.2 Results . . . . . 89
5.3 Discussion . . . . . 91

**VI. Conclusion and Future Work** **94**

**Summary (in Korean)** **96**

**References** **98**

# List of Tables

4.1 Requirement list of the orchestrator . . . . . 68
4.2 Types of OpenFlow messages which OFMon can monitor . . . . . 75

5.1 Time for each operation [ms] . . . . . 79

# List of Figures


1.1 Problem description of CAP, Elasticity, and ECP . . . . . 2

2.1 An SDN reference model . . . . . 8
2.2 An overview of OpenFlow components . . . . . 10
2.3 The structure and flow of *Role-request* and *Role-response* messages 12
2.4 The role assignment scheme in ONOS controllers . . . . . 13
2.5 Overviews of SCPs . . . . . 16
2.6 An overview of ECP . . . . . 18
2.7 An overview of data center infrastructure . . . . . 21

3.1 Comparison of ECP-1 and ECP-2 to resize the controller pool . . . 25
3.2 Examples of ECP-1 and ECP-2: Insufficient active controllers . . . 28
3.3 Examples of ECP-1 and ECP-2: No-operation period . . . . . 29
3.4 The computing performance with the various CPU burden . . . . . 32

4.1 System architecture . . . . . 34
4.2 Overviews of previous ECPs and H-ECP . . . . . 39
4.3 The flowchart of H-ECP . . . . . 41
4.4 An operational example of H-ECP . . . . . 53
4.5 Overviews of previous ECPs and T-DCORAL . . . . . 56
4.6 The flowchart of T-DCORAL . . . . . 57
4.7 The concept of deployment scheme with the various number of Head nodes . . . . . 63
4.8 An operational example of T-DCORAL . . . . . 64
4.9 An overview of the ECP framework . . . . . 67
4.10 High-level design of the ECP framework . . . . . 72
4.11 An architecture of ONOS . . . . . 73
4.12 A design of OFMon in ONOS . . . . . 74

5.1 An experimental environment for the evaluation of H-ECP and T-DCORAL . . . . . 77
5.2 The average rule installation time in both scenarios . . . . . 80
5.3 The CPU usage in both scenarios . . . . . 81
5.4 The number of vCPUs being occupied . . . . . 83
5.5 The network load . . . . . 85
5.6 An experimental environment for the evaluation the deployment scheme of the network devices in T-DCORAL . . . . . 88
5.7 The average rule installation time when deploying OpenVSwitches into the various number of Head nodes . . . . . 90

# List of Algorithms


4.1 Main procedure of H-ECP . . . . . 44
4.2 The background thread for the power management algorithm in H-ECP . . . . . 45
4.3 The foreground thread for the CP adjustment algorithm in H-ECP 46
4.4 The function to shrink the controller pool in H-ECP . . . . . 47
4.5 The function to expand the controller pool in H-ECP . . . . . 48
4.6 The function to rebalance the CP in H-ECP . . . . . 49
4.7 The initialization function in H-ECP . . . . . 52
4.8 Main procedure in T-DCORAL . . . . . 58


# I

# Introduction

Software-Defined Networking (SDN) is a promising approach to manage networks, nowadays [1]. The main feature of SDN decouples the control plane (CP) from the data plane (DP). The CP contains software controllers (e.g., NOX [2], Floodlight [3], Ryu [4], OpenDayLight [5], and ONOS [6]) which include all intelligence of network devices in the DP; each network device becomes a simple packet forwarder. Arriving a user data flow, each network device needs to request a flow rule to its controller through control protocols, such as OpenFlow [7], LISP [8], and CAPWAP [9]. Currently, several SDN use-cases have arisen to manage campus networks [10], Wide-Area Networks (WANs) [11, 12], and Data Center Networks (DCNs) [13].

The distributed CP is an effective way to control large-scale networks such as Software-Defined DCNs. There are innumerable network devices and data flows in a large-scale network. To manage the large-scale network, the primer CP, which contains a single controller, experiences the following issues: (i) low reliability, (ii) low responsiveness, and (iii) low scalability [14]. First, the primer CP is impossible to control network devices if the controller crashes, which causes low reliability. Next, the controller slowly processes each control message, because only one controller must handle a number of control messages (low responsiveness). The controller also encounters the scalability problem because there is a

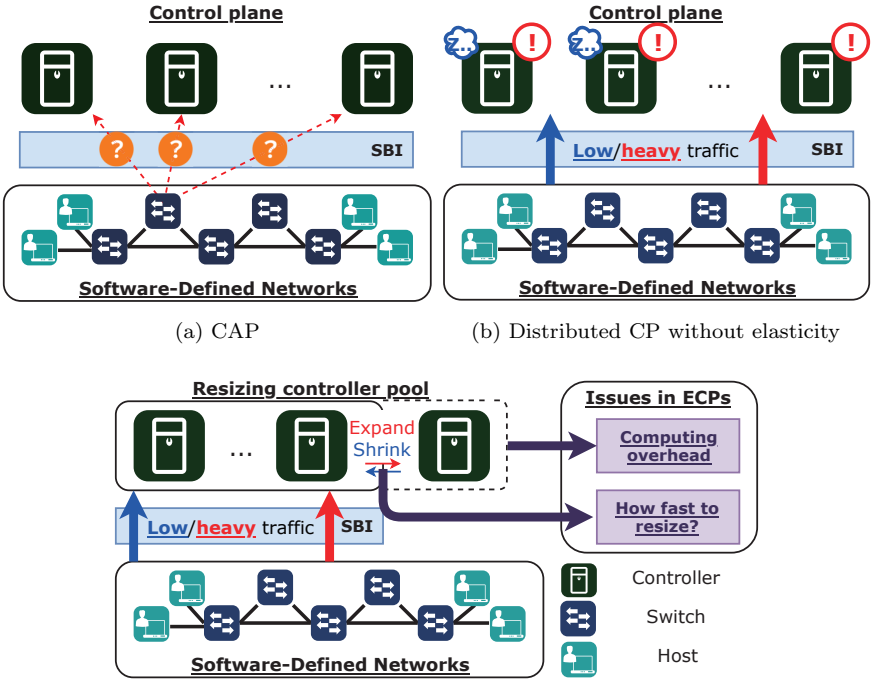



Figure 1.1: Problem description of CAP, Elasticity, and ECP

limit to the number of control messages the controller can deal with. The distributed CP is possible to resolve the above issues by utilizing multiple controllers to manage large-scale networks [14, 19].

However, the distributed CP faces the controller assignment (or placement) problem (CAP). The CAP is the problem to assign the master controller for each network device. Figure 1.1a shows an example of the CAP, where some controllers try to manage five network devices. It is essential to determine the master controller for each network device with an effective CAP solution. An inept CAP solution makes some controllers become overloaded controllers, which processes control messages, tardily. For instance, each network device requests flow rules to its master controller for data packet forwarding. Then, overloaded controllers install flow rules into network devices, belatedly. Therefore, an effective CAP solution is necessary to rapidly install flow rules. Fortunately, more than 30 solutions have been proposed in the last decade [16], such as heuristic

CAP solution [17], DSCP [18], and HeS-CoP [19].

Even if there are many effective CAP solutions, the distributed CP requires the ***elasticity*** to handle the abrupt change of the control traffic. The elasticity is the ability to resize the CP, dynamically (e.g., adjusting the number of controllers). Most distributed CPs utilize the static number of controllers, which brings about two problems: (i) the lack of controllers and (ii) the dissipation of controllers (Figure 1.1b). With a surge in the control traffic, static controllers are insufficient to process all control traffic on time, i.e., controllers become overloaded controllers. Those controllers are late to set the flow rules into network devices. On the contrary, some of the controllers are excessive when the control traffic plunges. The distributed CP then squanders the CPU resources which depict the CPU capacity and the number of virtual CPUs (vCPUs). Eventually, wasting the CPU resources curtails the opportunity that other services running on the same physical machine (PM) with controllers will occupy CPU resources. Furthermore, those two problems might be intensified in SD-DCNs which has a random traffic pattern [48–56]. Thus, the elasticity is fundamental in the distributed CP, especially for SD-DCNs.

The Elastic Control Plane (ECP) is the distributed CP supporting the elasticity by rescaling the controller pool the set of active controllers (Figure 1.1c). Previous ECPs define two types of controllers: the active controller which services at least one network device and the inactive controller which manages no network device. When controllers experience the high computing or networking load, the ECP expands the controller pool by activating inactive controllers. On the other hand, the controller pool dwindles by deactivating active controllers when the load decreases.

Despite resizing the controller pool, existing ECPs face two issues: (i) how much CPU resources an ECP wastes; (ii) how fast an ECP adapts the CP (e.g.,

resizing the controller pool). We categorize two types of ECPs, ECP-1 (DCP [33] and EAS [36]) and ECP-2 (ElastiCon [34, 35], PDDCP [37], EDES [38], and DCA-Online [39]). ECP-1 maintains all controllers in operation, even including inactive controllers, while ECP-2 pauses or shuts down inactive controllers. To activate an inactive controller, ECP-1 and ECP-2 perform different steps. ECP-1 just migrates appropriate network devices from active controllers to the inactive controller. ECP-2 requires one more step to switch on the inactive controller before the same migration step as ECP-1. Similarly, both ECP types have different ways to deactivate an active controller. ECP-1 also reattaches all network devices from the active controller to the other active controllers for deactivation. ECP-2 has to shut down the active controller after the same reattachment step as ECP-1. To sum up, ECP-1 utilizes more CPU resources than ECP-2, whereas ECP-2 spends a longer time to resize the controller pool than ECP-1.

In the recent Data Center (DC) infrastructure, the above two issues should be mitigated. The recent DC contains multiple Head nodes which operate controllers as well as other control services to manage DC [40–45]. When controllers waste the CPU resources, the other control services utilize insufficient CPU resources; control services tardily process control requests from users or operators (e.g., an instantiation of services, starting virtual instances, and resource monitoring). Besides, the DC faces the lack of controllers, when an ECP expands the controller pool slowly. This ECP installs flow rules into network devices belatedly, which is unable to guarantee the Service Level Agreement (SLA). Therefore, the above two issues must be resolved to process control requests on schedule and to ensure the SLA.

In this dissertation, we propose two new ECPs, a Hybrid ECP (H-ECP) and a Threshold-based Dynamic Controller Resource Allocation (T-DCORAL) to mitigate the above issues in SD-DCNs. H-ECP defines three types of con-

trollers, the active controller, the inactive controller, and the standby controller. The active controller services at least one network devices, whereas the inactive controller and the standby controller manage no network device. H-ECP switches off all inactive controllers and retains all standby controllers in operation. To avoid excessive or insufficient standby controllers, H-ECP maintains the predefined number of standby controllers in operation by switching on or off inactive or standby controllers, respectively. H-ECP imposes upon active and standby controllers to resize the controller pool. Thus, H-ECP achieves to preserve the CPU resources because of the inactive controllers being powered off. Furthermore, H-ECP accomplishes the fast rule installation time due to standby controllers. T-DCORAL leverages the minimum number of controllers to conserve the CPU resources. For the fast rule installation time, T-DCORAL never switches on/off any controller. T-DCORAL accelerates or decelerates each controller experiencing the high or low CPU load to process all control messages.

The contribution of this dissertation is four-fold.

- We initially categorize existing ECPs into two types and analyze them into two perspectives: (i) how fast to resize the CP resources (e.g., the controller pool) and (ii) how much CPU resources an ECP squanders.
- We propose new ECPs, H-ECP and T-DCORAL, to mitigate the issues which existing ECPs encountered.
- We design and implement the ECP framework to run H-ECP, T-DCORAL and the other ECPs for SD-DCNs.
- We evaluate our proposed ECPs with the ECP framework and popular controller software.

The remainder of this dissertation is organized as follows. Chapter II introduces background and related work. Then, we explain the immediacy and

computing overhead analysis in Chapter III. Chapter IV proposes H-ECP and T-DCORAL. In Chapter V, we evaluate our ECPs with existing ECPs. Finally, Chapter VI concludes this dissertation and suggests the future work of this dissertation.

This dissertation has the following assumptions. Controllers support distributed approach defined in OpenFlow 1.2 and beyond. Each controller operates in a VM running on Head nodes inside a DC. Each VM can exploit multiple virtual CPUs (vCPUs) which can share physical CPUs (pCPUs) with an N:1 vCPU to pCPU ratio. Activating all controllers, ECPs can process peak control messages. We describe H-ECP and T-DCORAL focusing on SD-DCNs in this dissertation because the SD-DCN is the most widely used and researched large-scale network nowadays. Of course, H-ECP and T-DCORAL are able to run in the other large-scale networks.

# II

# Background and Related Work

In this chapter, we introduce the background and related work of this dissertation. First, we describe the concept of SDN and OpenFlow which is one of the *de facto* standards of SDN. Next, we explain the roles of controllers defined in OpenFlow 1.2 and beyond specifications. Then, we explain what is the distributed control plane, the static control plane, and the ECP. Finally, we introduce the data center infrastructure.

## 2.1 Software-Defined Networking

Recently, SDN is a spotlighted technology to manage various networks. The main characteristics of SDN are two-fold: (i) decoupling the CP from the DP and (ii) the programmability. To decouple the CP from the DP, SDN initially deprives all control logic of network devices in the DP. Next, SDN defines software controllers which posses the control logic. Eventually, network devices become simple packet forwarder managed by controllers through control protocols (e.g., OpenFlow [7], LISP [8], and CAPWAP [9]). The next main feature is to support the programmability due to software controllers which are open-source software mostly. Thereby, network operators or programmers are able to implement appropriate network management applications by themselves, easily.

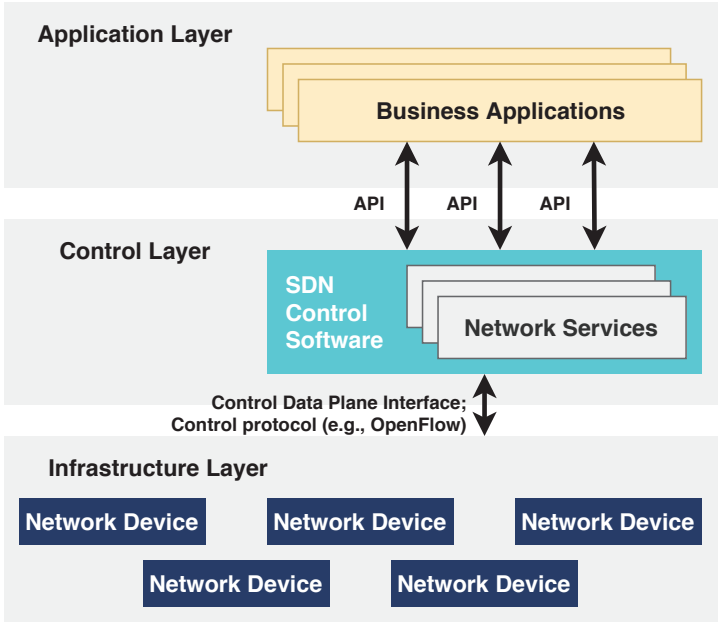



Figure 2.1: An SDN reference model

Figure 2.1 depicts the SDN reference model including three layers: (i) infrastructure layer, (ii) control layer, and (iii) application layer [20]. The infrastructure layer is the DP which contains countless network devices, e.g., OpenFlow switches and CAPWAP-enabled wireless access points. Each network device has two functionalities, the function to record statistics and the function to forward data packets. The control layer consists of SDN control software, i.e., controllers. This layer monitors the DP such as the network status, the global network view, and network statistics. Besides, this layer performs to command the packet forwarding to network devices and to process requests from the application layer. The application layer has multiple business applications implemented by network operators or programmers for their purposes. For example, an operator can implement the network monitoring application, the network virtualization application, and the load balancing application. To communicate with adjacent layers, this model defines two interfaces, the Application Programming Interface (API) and the control-data plane interface (called the south-bound interface in

general; SBI). The SBI connects between the control layer and the infrastructure layer, whereas API links the control layer with the application layer.

To manage the network with SDN, there are several benefits below [20].

- SDN enhances network configurations compared with the previous network. A network operator needs to configure some system parameters inside network devices. To configure each network device in the previous network, the network operator manually sets some parameters (e.g., IP address, VLANs, and DNS) through configuration tools implemented by device vendors. The manual configuration, however, causes the human error, i.e., the error-prone configuration. In SDN, controllers have all control logic even including network configuration functions [21]. Therefore, the network operator can configure all network devices through a single point which is the controller.

- SDN optimizes the network performance. The previous network contains different types and vendors of network devices. Then, the network operator is tough to optimize the network performance with consideration for all network vendors and types. Meanwhile, the network operator is possible to observe the global network view via controllers in SDN. Consequently, the network operator globally optimizes the network performance, such as data traffic scheduling [22], end-to-end congestion control [23], load balanced packet routing [24], energy efficient operation [25], and Quality of Service (QoS) support [26].

- SDN easily and rapidly deploys new services and applications for the network management. The previous network consists of various network devices from different device vendors. If the network operator tries to deploy a new service (e.g., network monitoring service), the network operator asks to implement the new service available toward all device vendors. After that,

those vendors start to implement and test the new service, which spends a long time and huge money. In SDN, controllers are software; the network operator or programmer can implement the new service which they want on top of controllers. Thus, the network operator deploys new services in SDN easier and faster than the previous network.

## 2.2 OpenFlow

OpenFlow is one of the open control protocols to communicate between controllers and network devices. Currently, this protocol is a *de facto* standard for SDN. OpenFlow contains two essential elements, the controller and the OpenFlow switch as shown in Figure 2.2 [7]. The controller is software running on a general-purpose computer to manage OpenFlow switches. The OpenFlow switch contains the secure channel and the flow tables. The secure channel is the interface to communicate with controllers. Through this channel, controllers command some actions, e.g., packet forwarding and network monitoring. The flow table comprises the flow entries also known as flow rules. Each flow entry consists of three fields: match field, action field, and statistics field. The match field is to compare the data packet information, while the action field constitutes the set of operations like packet forwarding or packet header modification. The statistics field is the

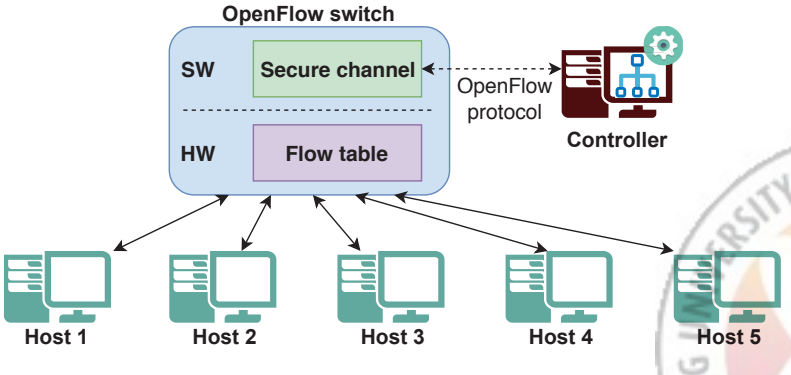



Figure 2.2: An overview of OpenFlow components

counter of the flow entry. If an OpenFlow switch receives a data packet which corresponds one of the match fields (e.g., matched the source IP address), the OpenFlow switch operates actions described in the action field (e.g., forwarding the packet to the port 1). When the packet hits the match field, the OpenFlow switch records it into the statistics field.

In the OpenFlow specification [7], controllers send *Stat-request* messages (OpenFlow 1.0-1.2) or *Multipart-request* messages (OpenFlow 1.3 and beyond) to collect the statistics information for each flow entry. Receiving one of two messages, an OpenFlow switch responses with *Stat-response* (OpenFlow 1.0-1.2) message or *Multipart-response* messages (OpenFlow 1.3 and beyond). Those response messages involve the detailed statistics information.

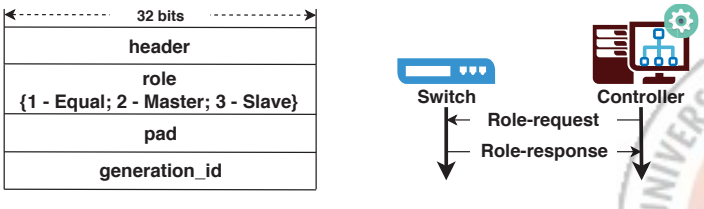
To install a new flow rule, the OpenFlow specification defines three types of messages: *Packet-in* message, *Packet-out* message, and *Flow-mod* message. Receiving a new data packet, the OpenFlow switch initially looks the correct flow rule up in the flow table. The OpenFlow switch will operate actions described in the action field if the flow table contains the flow rule for the new data packet. Otherwise, the OpenFlow switch buffers the data packet temporally and transmits a *Packet-in* message to the controller. The controller calculates the path to arrive the data packet to the destination host after receiving the *Packet-in* message. Since the *Packet-in* message has information about the data packet (e.g., source IP address, destination IP address, and port number), the controller can find the optimal or appropriate path according to the global network view. After calculating the path, the controller acknowledges the OpenFlow switch with the *Flow-mod* message including the forwarding action to follow the path. The OpenFlow switch then adds a new flow rule with regard to *Flow-mod* message. Also, the controller sends back the *Packet-out* message after the transmission of *Flow-mod* message. The OpenFlow switch finally forwards the buffered data

packet by means of the new flow rule when receiving the *Packet-out* message.

## 2.3 Roles of Controllers in OpenFlow 1.2 and Beyond

OpenFlow 1.2 and beyond specifications allow multiple distributed controllers to manage OpenFlow switches. To utilize multiple controllers, those specifications define three roles of the controller for each OpenFlow switch: (i) *Equal*, (ii) *Master*, and (iii) *Slave* [7]. The Equal role is the default role of a controller for an OpenFlow switch, which signifies the privilege to install flow rules to the OpenFlow switch. Likewise, the controller with the Master role (called the master controller) also manages the OpenFlow switch with the same privilege. The difference between both roles is the number of controllers which can occupy each role for a single OpenFlow switch. Multiple controllers can possess the Equal role for the OpenFlow switch, whereas each OpenFlow switch has only one master controller. Last, the Slave role depicts the read-only privilege to manage an OpenFlow switch. The controller with the Slave role (called the slave controller) monitors the OpenFlow switch; the slave controller is impossible to install flow rules into the OpenFlow switch.

To assign one of three roles, OpenFlow 1.2 and beyond specifications define two types of OpenFlow messages: (i) *Role-request* message and (ii) *Role-response* message [7]. Figure 2.3 illustrates the message structure and message flow of both OpenFlow messages. Both messages have the same structure, which includes a




(a) Message structure (b) Message flow

Figure 2.3: The structure and flow of *Role-request* and *Role-response* messages

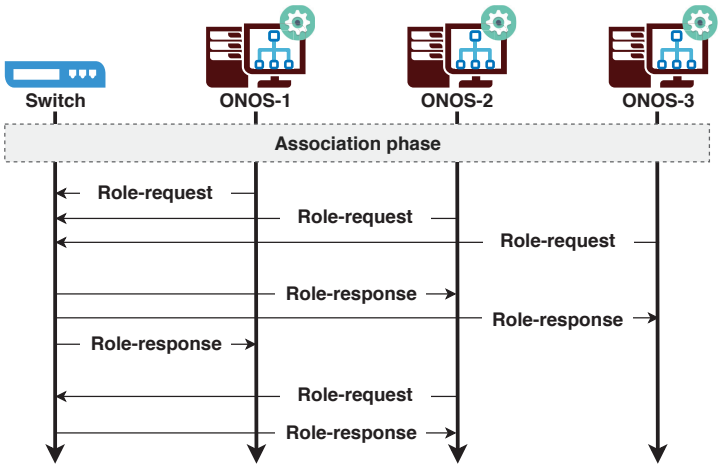



Figure 2.4: The role assignment scheme in ONOS controllers

32-bit role field in between the header field and the pad field (Figure 2.3a). The role field is set to 0x00000001, 0x00000002, and 0x00000003 to allot the Equal, Master, and Slave role to each message, respectively. Figure 2.3b depicts the flow of two messages between an OpenFlow switch and a controller. To begin with, the controller transmits the *Role-request* message with the role the controller demands to the OpenFlow switch. The OpenFlow switch modifies the role which the controller transmitted after receiving the *Role-request* message. Finally, the OpenFlow switch notifies the controller with the *Role-response* message with the role information which the OpenFlow switch recorded.

We analyzed how to assign the role of ONOS controller for each OpenFlow switch. Figure 2.4 illustrates the role assignment scheme in ONOS controllers. In this figure, there are three ONOS controllers to manage an OpenFlow switch. Before assigning the role of each controller, the OpenFlow switch associated with all ONOS controllers by leveraging OpenFlow messages, e.g., the *Hello* message. After the association phase, each ONOS controller sent *Role-request* message with the Equal role to the OpenFlow switch. Then, the OpenFlow switch acknowledged all ONOS controllers with the *Role-response* message. Next, all ONOS controllers synchronized the time which they received the *Role-response*

message. With the received time information, all ONOS controllers elected the master controller which received the *Role-response* message first. In Figure 2.4, ONOS-2 receives the *Role-response* message, primally. After the election, the master controller (ONOS-2) forwarded the *Role-request* message with the Master role. Finally, the OpenFlow switch transmitted the *Role-response* message with the Master role.

## 2.4 Distributed Control Plane

Since the centralized CP experiences the low responsiveness, the low reliability, and the low scalability [14], the distributed CP has been proposed. This CP distributes the control traffic load to other nodes or controllers, which processes control traffic load more and faster than the centralized CP. Proposed distributed CPs can be categorized into two types: (i) the flat SDN control and (ii) the hierarchical SDN control [27].

The flat SDN control defines only one flat layer, whereas the hierarchical SDN control assumes multiple layers in the CP. The flat SDN control imposes upon multiple controllers located in the flat layer, such as HyperFlow [29], OpenDayLight [5], Onix [28], and ONOS [6]. Each controller is responsible for some of network devices as a master controller. However, the hierarchical SDN control locates controllers or nodes on different layers, such as DevoFlow [30], DIFANE [31], and Kandoo [32]. First, DevoFlow tries to reduce the load of each controller. DevoFlow controllers install elephant flow rules, while mice flow rules are set up by network devices. Therefore, each DevoFlow controller encounters the control load lower than the controller installing all flows. DIFANE decreases an overhead of each controller by utilizing authority switches. Arriving the first packet of a data flow, the authority switch notifies it to a controller. Afterward, the controller installs the flow rule to the switch which then caches the flow rule. Receiving

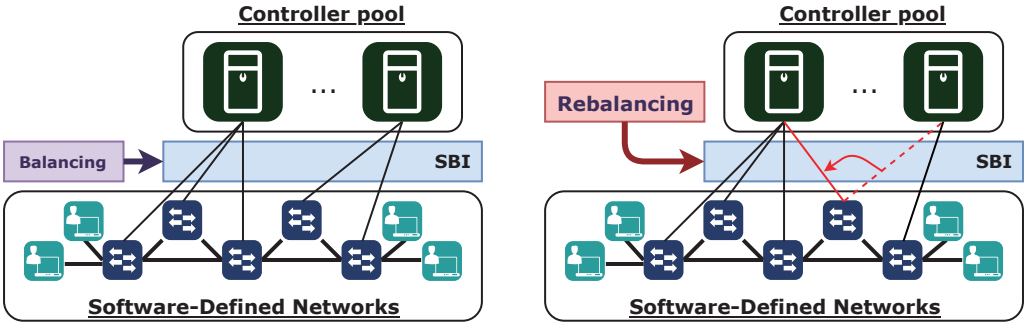
packets of the cached flow, the authority switch forwards the packets according to the cached flow rule. Kandoo exploits two types of controllers which are a root controller and local controllers located in two layers. The root controller deals with global operations, whereas local controllers process local operations.

## 2.5 Static Control Plane

The initial distributed CP takes advantage of the static number of controllers, which is called the Static Control Plane (SCP) [14, 15, 17–19]. The SCP starts up all static controllers in the CP to manage network devices in the DP. For example, the CP contains $n$ static controllers for $m$ network devices. The SCP boots up $n$ controllers and then assigns $m/n$ network device to each controller, which is the straightforward way. In order to use the SCP, we need to define (i) which controller is the master controller for each network device (the controller assignment problem; CAP) and (ii) how many controllers the SCP should leverage [15].

The CAP needs to be solved with an effective solution for the SCP, i.e., each controller should service appropriate network devices as a master controller. In the imbalance of the control traffic load due to an ineffective solution, the controllers become overloaded controllers. Since overloaded controllers process control messages tardily, the rule installation time delays [15, 16, 18, 19, 33]. Fortunately, many solutions have been proposed in the last decade [16]. Among the previous solutions, we categorized them into two types: the SCP with the static configuration (SCP-S) and the SCP with the dynamic configuration (SCP-D).

The SCP-S configures the CP topology, statically. The master controller for each network device is stationary, not changing the master controller over time. Figure 2.5a shows an overview of SCP-S. Starting the SCP-S, each controller acquires Master roles for some network devices in an effective solution. After




(a) SCP-S (b) SCP-D

Figure 2.5: Overviews of SCPs

that, the master controller for each network device never changes, i.e., balancing network devices one time at the very beginning. To assign the master controller, Heller *et al.* [15] place the master controller for each network device according to the latency between controllers and each network device. The ONOS controller [6] has the master controller election scheme as explained Section 2.3 and Figure 2.4. With the election scheme, each network device is managed by its master controller, statically. After operating the election scheme, the CP topology never changes. Only operators can change the CP topology, manually.

The SCP-D reforms the CP topology dynamically over time because the control traffic load varies [16–19, 33]. In the current network (e.g., campus networks, enterprise networks, WANs, and especially SD-DCNs), the number of data flows always fluctuates over time in DP. As the number of data flows changes over time, the more (less) data flows each network device faces, the more (less) requests each network device transmits to its master controller. Consequently, each controller becomes an overloaded controller or an underutilized controller, which will change over time. Since overloaded controllers delay the rule installation time, the load balancing is necessary for the SCP. Figure 2.5b shows an overview of SCP-D. The SCP-D has the rebalancing scheme to migrate appropriate network devices from overloaded controllers to underutilized controllers, dynamically.

We introduce some of the SCP-Ds. Li *et al.* [18] have proposed the Dynamic Switch-to-Controller Placement scheme (DSCP). According to the number of OpenFlow messages between each network device and its master controller, DSCP moves some network devices from over-subscribed controllers to under-subscribed controllers at every timeslot. Finally, each controller suffers from the similar number of OpenFlow messages. Kim *et al.* which is our previous study [19], have proposed the Heuristic Switch-Controller Placement scheme (HeS-CoP). The HeS-CoP considers both the CPU load of each controller and the number of OpenFlow messages. According to both metrics, the HeS-CoP dynamically changes the CP topology to balance the OpenFlow messages across controllers. Lange *et al.* [17] have suggested the new heuristic controller placement scheme to handle the link failure. Except for the above schemes, more than 30 solutions have been proposed in the last decade [16].

## 2.6 Elastic Control Plane

Although the SCP-D balances the control traffic load well, there is one more issue that is the reasonable number of controllers to manage network devices. Heller *et al.* [15] define the reasonable number of controllers as "it depends." Of course, the more controllers the CP uses, the shorter latency to install flow rules the CP achieves. However, this is not a linearly proportional relationship. For example, one CP exploits the fair enough number of controllers, and the other CP overuses controllers. In this example, those two CPs achieve the very similar rule installation time, which is very short rule installation time [15]. Meanwhile, the CP spends the high cost (e.g., the computing resource and the energy consumption) when overusing the controllers [35,36,38]. Thus, we need to define the reasonable number of controllers.

For instance, we defined the number of controllers for the SCP-D with an

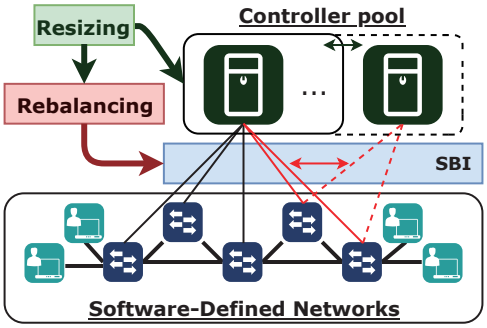



Figure 2.6: An overview of ECP

effective way, very reasonably. In this situation, the number of data traffic flows could exceed the capacity which all running controllers can process. Then, all controllers become overloaded controllers with a surge in data traffic flows, which causes the extremely slow rule installation time. To reduce the delay, we can overuse controllers with respect to the estimated peak flow rules. However, this solution is inefficient because of the high cost to keep all controllers in operation in terms of the computing resource, the networking resource, and the power consumption [35, 36, 38].

To solve the above problem, the Elastic Control Plane (ECP) has been proposed, which adapts the number of controllers being used (Figure 2.6) [33–39]. The ECP defines two types of controllers, the active controller and the inactive controller. The active controller manages at least one network device as a master controller, while the inactive controller services no network device with the Master role. With two types of controllers, the ECP dynamically adjusts the number of active controllers in conformity with the CPU or network load. The ECP expands the controller pool if active controllers may become overloaded controllers. On the other hand, the ECP shrinks the controller pool when overusing active controllers. Otherwise, the ECP rebalances the CP load by reforming the CP topology. As a result, the ECP is able to install flow rules on time even when data traffic increases. The ECP also utilizes efficient resources and reduces the

energy consumption.

From 2013, six ECPs have been proposed as follows:

- Bari *et al.* [33] have proposed the dynamic controller provisioning (DCP). The goal of DCP minimizes the rule installation time with low communication overhead by adapting the number of active controllers. DCP defines the communication overhead as the combination of four costs: (i) statistics collection cost; (ii) flow setup cost; (iii) synchronization cost; (iv) switch assignment cost. The statistics collection cost is the number of messages to collect statistics at each controller, whereas the flow setup cost is the cost to set up the flow rules across the end-to-end path. The synchronization cost and the switch assignment cost are the numbers of inter-controller messages and the cost to assign a network device to a new controller, respectively. To achieve the goal, DCP has two proposed scheme, a greedy knapsack approach (DCP-GK) and a simulated annealing based meta-heuristic approach (DCP-SA). For those approaches, DCP assumes that all controllers are running in appropriate servers and DCP focuses on the WANs.

- Dixit *et al.* [34, 35] have proposed ElastiCon, an elastic distributed SDN controller. The CPU load represents the controller load because the CPU resource is the performance bottleneck. The CPU load each network devices causes is the controller's CPU load proportional to the number of control messages each network device transceives. With the load definition, ElastiCon migrates appropriate network devices from overloaded controllers to underutilized controllers to make all controllers suffer from a similar CPU load (i.e., to avoid the load imbalance situation). In addition, ElastiCon expands (or shrinks) the controller pool by activating (or deactivating) an active (or inactive controller), when the CPU load of controllers is over (or under) the threshold. To activate or deactivate controllers, ElastiCon

powers on or off target controllers, respectively.

- Chen *et al.* [36] have proposed an elastic adaptive SDN (EAS). With the greedy approach, EAS dynamically activates and deactivates controllers to resize the controller pool. Also, EAS migrates network devices from over-subscribed controllers to under-subscribed controllers. EAS decides the resizing and rebalancing step according to the network load. If the network load is over (under) the upper (lower) threshold, EAS expands (shrinks) the controller pool. With EAS, all controllers even including inactive controllers are in operation at appropriate servers.

- Mattos *et al.* [37] have proposed a profile-based dynamic distributed controller provisioning (PDDCP). Based on the Markov chain, PDDCP creates a network profile of each controller according to the number of requests which each network device communicates with its master controller. Then, PDDCP categorizes the three controller states, increasing, decreasing, and stable flow setup request load. If the load increases, PDDCP activates one more controller. In the decreasing state, PDDCP shrinks the controller pool by deactivating an active controller. To activate/deactivate controllers, PDDCP switches on/off controllers.

- Prathyusha *et al.* [38] have proposed an efficient DHT-based elastic SDN (EDES). EDES is the enhanced scheme of ElastiCon. In fact, ElastiCon has the high time complexity in the rebalancing algorithm. EDES tries to reduce the time complexity with the DHT-based efficient algorithm. Of course, EDES powers off all inactive controllers.

- Wang *et al.* [39] have proposed an efficient online algorithm for dynamic SDN controller assignment (DCA-Online), which is highly similar to DCP. Indeed, DCP is the heuristic manner and encounters the high time com-

plexity. However, DCA-Online tries to reduce the time complexity with the optimal online algorithm. To decide the expansion and shrinkage of the controller pool, DCA-Online defines the operating cost and switching cost. The operating cost includes the delay cost (the control plane response time) and the maintenance cost (the synchronization cost in DCP). The switching cost is the cost to change the controller state from active to inactive or vice versa in between timeslots.

## 2.7 Data Center Infrastructure

Nowadays, the DC has been widely used already, e.g., Google Cloud Platform (GCP), Amazon Web Service (AWS), and Microsoft Azure. Figure 2.7 depicts an overview of the DC infrastructure proposed by CORD (one of the popular open-source edge-cloud platform) [40] and the many solution vendors such as Cisco [41], Microsoft [42], IBM [43], Google [44], and Joyent [45]. Inside this infrastructure, there are three essential components: (i) DCNs, (ii) Compute nodes, and (iii) Head nodes (or called Master nodes).

The DCN is the network to interconnect among nodes and to connect with the network outside the DC as well. The DCN consists of physical networks and

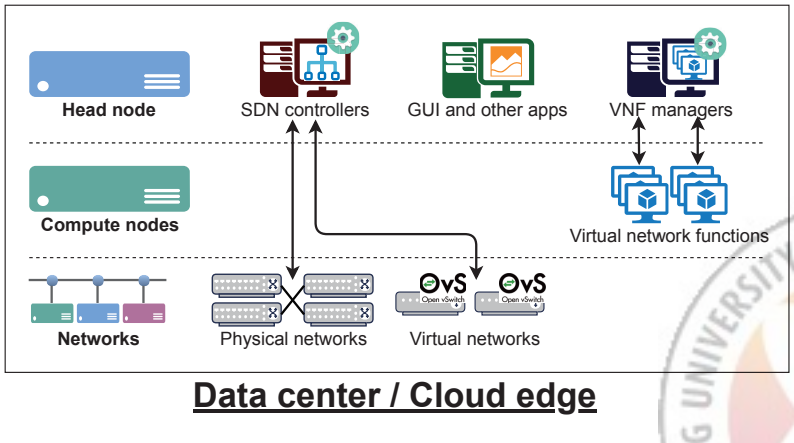



Figure 2.7: An overview of data center infrastructure

virtual networks. The physical network is the network to connect nodes inside or outside the DC, physically. The physical network consists of countless fabric switches in general, which is organized with various network topologies such as Fat-tree [47] and VL2 [54]. The virtual network interconnects virtual services, e.g., VMs and containers, on top of the physical network.

Compute node is the server to run user services such as VMs, containers, cloud applications (e.g., Google Gmail and docs), and Virtual Network Functions (VNFs) such as the virtual router, vMME, vHSS, vSGW, and vPGW. In a single DC, there are innumerable Compute nodes to run countless user services.

Head node is the server to manage the DC which includes several components: (i) controllers to manage DCNs (e.g., ONOS [6] and OpenDayLight [5]); (ii) VNF managers (e.g., Synchronizers in CORD [40] to control user services, OpenStack [62] to manage VMs, and Kubernetes [63] to control containers); (iii) Graphical User Interface (GUI); (iv) other useful applications. Controllers manage DCNs with control protocols such as OpenFlow [7]. In the CORD platform [40], Head node has two ONOS controllers: ONOS-VTN and ONOS-Fabric. ONOS-VTN controls OpenVSwitches on the virtual network, whereas ONOS-Fabric manages fabric OpenFlow switches on the physical network. VNF managers monitor and control user services. For instance, a user wants to make new VMs as a service in the DC. VNF managers then receive the user demand, launch new VMs, and make appropriate virtual networks. To visualize the DC or to interact with operators, Head node contains GUI. Except for those components, there are other useful applications in Head node such as Database and WEB server.

DCNs have both temporal and spatial traffic variations. First, DCNs comprise multiple network devices which locate in one of the different layers. Network devices in DCNs suffer from extremely different flow arrival rates [48,53] and traf-

fic fluctuation [49–51] because network devices forward data packets on one of the different layers. Next, DCNs experience the different data traffic over time. In the daytime, DCNs transmit and receive more data traffic than the nighttime [48, 52]. Besides, DCNs suffer from the data traffic fluctuation even in the short term [55]. Due to the spatial and temporal data traffic characteristics, the control traffic also changes. To sum up, DCNs have the traffic variation due to the above reasons; we consider that the traffic pattern in DCNs is the random traffic pattern.

# III

# Immediacy and Computing Overhead Analysis

In this chapter, we analyze existing ECPs in terms of the immediacy and the computing overhead. First of all, we categorize proposed ECPs and then define the immediacy and the computing overhead. With two perspectives, we analyze each type of ECPs in detail.

## 3.1 Types of ECP

Existing ECPs [33–39] consist of two algorithms in general: (i) the rebalancing algorithm and (ii) the resizing algorithm. The rebalancing algorithm is to balance the control traffic load across controllers by changing the CP topology. If there are overloaded controllers, the rebalancing algorithm reforms the CP topology by moving some network devices from overloaded controllers to the other controllers. The resizing algorithm is to adjust the size of the controller pool by activating/deactivating target controllers. When insufficient active controllers manage a number of network devices and data flows, the resizing algorithm activates inactive controllers to grow the controller pool. On the contrary, the resizing algorithm deactivates active controllers to shrivel the controller pool, if the CP overuses active controllers due to the decrease of the control traffic load.

We classify existing ECPs into two types considering with the resizing algo-

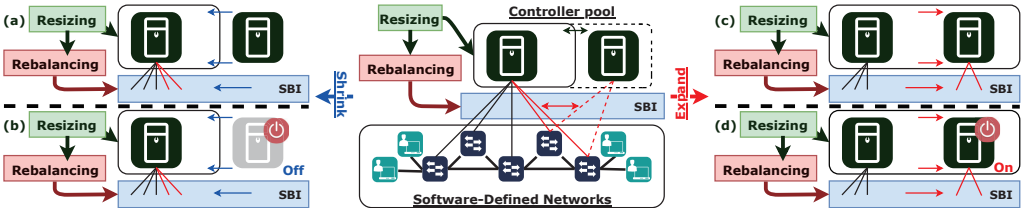



Figure 3.1: Comparison of ECP-1 and ECP-2 to resize the controller pool

rithm. In ECP-1 which is the first type of ECPs, the resizing algorithm keeps all controllers in operation. The resizing algorithm in ECP-1 includes the only one step which changes the CP topology to activate or deactivate target inactive or active controllers, respectively. Consequently, there is no need to switch on/off target controllers. On the contrary, the resizing algorithm in the second type of ECPs (ECP-2) maintains all inactive controllers turned off. To activate or deactivate a controller, the resizing algorithm comprises two steps: (i) the step to change the CP topology same as ECP-1 and (ii) the step to power on/off the controller. When ECP-2 wants to leverage one more active controller, the resizing algorithm initially switches on the active controller before changing the CP topology. In contrast to the activation, the resizing algorithm in ECP-2 needs to power off an inactive controller right after altering the CP topology.

Figure 3.1 shows the comparison of ECP-1 and ECP-2 in which there are two controllers to manage five OpenFlow switches. To begin with, we assume the scenario to shrink the controller pool by deactivating the right controller: The left controller manages three OpenFlow switches (black line) and the right controller services two OpenFlow switches (red line), initially. The resizing algorithm in ECP-1 migrates two OpenFlow switches from the right controller to the left controller (Figure 3.1a). Likewise, the resizing algorithm in ECP-2 initially reattaches all OpenFlow switches from the right controller to the left controller. After that, this resizing algorithm shuts down the right controller (Figure 3.1b). Next, we define the scenario to expand the controller pool by activating the right con-

troller: The left controller services all OpenFlow switches and the right controller is an inactive controller. To activate the right controller, the resizing algorithm in ECP-1 reconnects two OpenFlow switches to the right controller (Figure 3.1c). However, the resizing algorithm in ECP-2 firstly switches on the right controller. Then, this resizing algorithm reassociates two OpenFlow switches from the left controller to the right controller (Figure 3.1d).

Among proposed ECPs, DCP [33] and EAS [36] correspond ECP-1, while ElastiCon [34,35], PDDCP [37], EDES [38], and DCA-Online [39] are in the type ECP-2. DCP and EAS all controllers in operation, whereas ElastiCon, PDDCP, EDES, and DCA-Online shut down all inactive controllers. Since DCP and EAS power on/off no controllers, they come under ECP-1. The other ECPs switch off inactive controllers. Those ECPs need to switch on and off target controllers when enlarging and dwindling the controller pool, respectively. As a result, ElastiCon, PDDCP, EDES, and DCA-Online are applicable to ECP-2.

## 3.2 What is the Immediacy and Computing Overhead?

We analyze each type of ECP with two perspectives: (i) the immediacy and (ii) the computing overhead. The immediacy is the latency to adapt the CP, such as resizing and rebalancing the CP. If an ECP resizes the controller pool and/or modifies the CP topology tardily, we called that the ECP meets with the low immediacy. In contrast, an ECP with the high immediacy scales the controller pool and/or alters the CP topology, rapidly. The computing overhead represents the dissipation of CPU resources which indicate the CPU capacity and the number of vCPUs. Experiencing the high computing overhead, an ECP squanders the CPU resources, i.e., the ECP exploits more vCPUs and/or suffers from more CPU load. On the other hand, the ECP with the low computing

overhead imposes upon the CPU resources less than other ECPs.

## 3.3 Immediacy Analysis

Concerning the immediacy, ECP-1 outperforms ECP-2. The resizing algorithm in ECP-1 has only one step to change the CP topology, whereas the resizing algorithm in ECP-2 includes two steps, (i) the step to change the CP and (ii) the step to switch on/off target controllers. In particular, the second step in ECP-2 operates longer than the first step. The first step spends around 3 seconds while the second step devotes around 46 seconds and 13 seconds for switching on and off a target controller, respectively[1]. Therefore, ECP-1 meets with higher immediacy than ECP-2.

Resizing the controller pool, ECP-2 delays the rule installation time due to lower immediacy than ECP-1. ECP-1 makes the preparation of sufficient active controllers timely because of the high immediacy. However, ECP-2 undergoes the following two practical problems when resizing the controller pool: (i) the lack of active controllers and (ii) the no-operation period. Let the control traffic load increase in SD-DCNs. ECP-2 should switch on an inactive controller to expand the controller pool. Since ECP-2 expands the controller pool in procrastination, the number of active controllers is still deficient while switching on the inactive controllers. Eventually, ECP-2 weights each active controller down with the heavy control traffic load; each active controller turns into an overloaded controller. Also, we assume that the control traffic load decreases in SD-DCNs. Due to the low control traffic load, both ECP-1 and ECP-2 shrink the controller pool by deactivating an active controller. Even if ECP-2 still leverages sufficient active controllers without the active controllers which they are being deactivated, ECP-2 performs no operation until finishing the shrinkage (we called this period

[1]The performance time of each operation will show in Chapter VIII, detailedly.

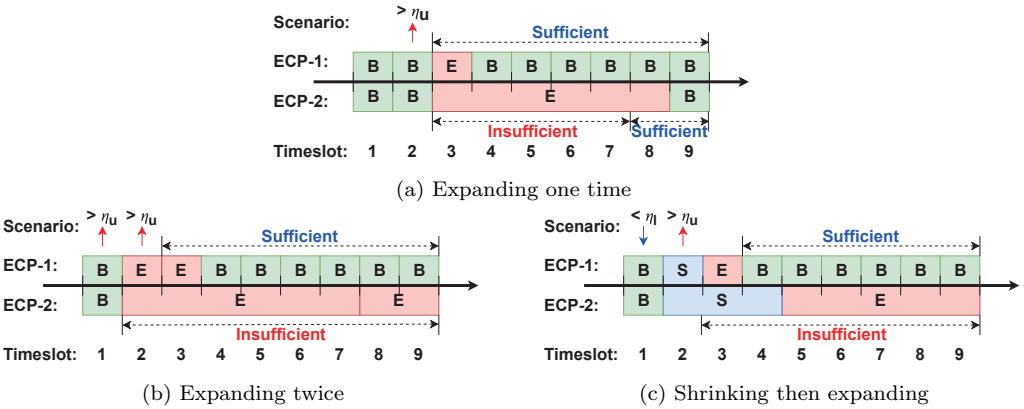



Figure 3.2: Examples of ECP-1 and ECP-2: Insufficient active controllers

as a no-operation period). Consequently, ECP-2 is impossible to resize the controller pool or even to rebalance the CP in the no-operation period, which makes some controllers become overloaded controllers. Finally, ECP-2 installs flow rules slower than ECP-1 due to the overloaded controllers.

Figure 3.2 and Figure 3.3 show examples of ECP-1 and ECP-2. In those figures, both ECPs perform one of three operations as follows: (i) If a load of active controllers exceeds the upper threshold ($\eta_u$), ECPs expand the controller pool (E within a red box); (ii) If a load of active controllers is below to the lower threshold ($\eta_l$), ECPs shrink the controller pool (S within a blue box); (iii) Otherwise, ECPs rebalance the CP (B within a green box). We assume that ECP-1 finishes all kinds of operations in a single timeslot. On the other hand, ECP-2 needs more timeslots than ECP-1 to expand and shrink the controller pool because of the low immediacy. The rebalancing operation of ECP-2, however, consumes only one timeslot, because ECP-2 powers on and off no controllers in this step, which is the same operation in ECP-1.

Figure 3.2 depicts examples of ECP-1 and ECP-2 which experiences the lack of active controllers. First, Figure 3.2a is an example that a load of active controllers exceeds the $\eta_u$ at the timeslot 2, where both ECPs start to expand the controller pool at the timeslot 3. ECP-1 finishes the expanding operation at

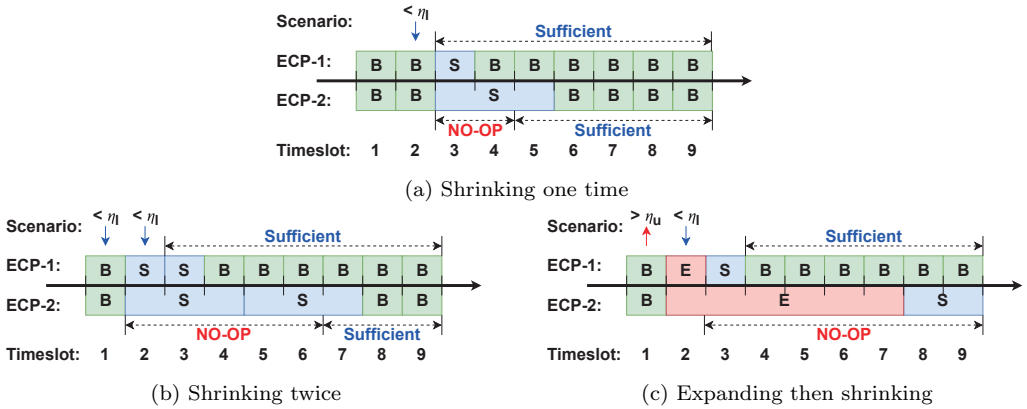



Figure 3.3: Examples of ECP-1 and ECP-2: No-operation period

the timeslot 3, whereas ECP-2 expands the controller pool from the timeslot 3 to 8. Eventually, ECP-2 takes advantage of insufficient active controllers to process all control messages in the timeslot 3 through 7. Figure 3.2b shows an example of the load upwards the $\eta_u$ at the timeslot 1 and 2, sequentially. ECP-1 ends to expand the controller pool at the timeslot 2 and 3. ECP-2 completes the first expanding operation and then starts to perform the second expanding operation. As a sequence, ECP-2 faces the lack of active controllers from the timeslot 2. Finally, Figure 3.2c draws an example of the load under the $\eta_l$ and over the $\eta_u$ at the timeslot 1 and 2, respectively. ECP-1 terminates the shrinking operation and the expanding operation at the timeslot 2 and 3, respectively. However, ECP-2 delays to expand the controller pool due to the first shrinking operation, which brings about the lack of active controllers.

We draw examples of ECP-1 and ECP-2 which suffers from a no-operation period in Figure 3.3. Figure 3.3a assumes that the load of active controllers is beneath the $\eta_l$ at the timeslot 2; ECP-1 and ECP-2 start to shrink the controller pool at the timeslot 3. On the one hand, ECP-1 finishes the shrinking operation at the same timeslot on schedule. On the other hand, ECP-2 ends the shrinking operation at the timeslot 5, which causes the no-operation period from the timeslot 3 to 4. Next, Figure 3.3b depicts an example of the load under the $\eta_l$ at the

timeslot 1 and 2, continuously. Of course, ECP-1 completes the shrinking operation at the timeslot 2 and 3, punctually. In contrast to ECP-1, ECP-2 terminates each shrinkage step at the timeslot 2 and 7. Accordingly, ECP-2 is impossible to perform any operation, e.g., rebalancing and resizing operations, in the timeslot 2 through 6. Last, Figure 3.3c illustrates that the load of active controllers is over the $\eta_u$ and subsequently under the $\eta_l$ at the timeslot 1 and 2, respectively. In this situation, ECP-1 terminates each operation timely, while ECP-2 delays the shrinking operation due to the expanding operation. Thus, ECP-2 meets with the no-operation period from the timeslot 3 to 9.

## 3.4 Computing Overhead Analysis

ECP-2 surpasses ECP-1 regarding the computing overhead since ECP-2 makes all inactive controllers shut down. Each controller is software running inside a VM which occupies one or more vCPUs. Moreover, a running inactive controller should operate the following modules: user interface modules; subsystems to handle control protocols (e.g., OpenFlow [7], CAPWAP [9], and LISP [8]); modules to synchronize with other controllers; other useful modules (e.g., traffic monitoring, deep packet inspection, and router). The running inactive controller which manages even no network device leads to the additional CPU load because of the above modules. Shutting down VMs which performed inactive controllers, ECP-2 conserves the CPU resources. Therefore, ECP-2 encounters the CPU load and leverages the number of vCPUs lower than ECP-1; ECP-2 outshines ECP-1 in terms of the computing overhead.

The ECP with the high computing overhead (ECP-1) influences the other services (e.g., applications, VMs, and containers) performing in the same PM. On account of the limited CPU resources, all services including controllers share pCPUs in the same PM. When controllers dissipate the CPU resources, the other

services gain the low chance to utilize the CPU resources, which causes the service performance degradation. Recently, DC architectures (e.g., CORD and the commercial DC architectures proposed by Cisco [41], Microsoft [42], IBM [43], Google [44], and Joyent [45]) have two types of nodes as we introduced in Chapter II: (i) lots of Compute nodes and (ii) a few Head nodes which perform controllers and other control services. If controllers squander the CPU resources, the other control services exploit the CPU resources with the low opportunity. Eventually, the performance of the other control services degrades, which is the critical problem to manage DC.

For example, a user wants to make a virtual Evolved Packet Core (vEPC) service or network infrastructure, when an ECP wastes CPU resources like ECP-1. Then, some control services deploy VNFs into appropriate Compute nodes and define virtual networks, tardily. As a result, the DC architecture transgresses the SLA due to the high computing overhead. Therefore, an ECP should pursue the low computing overhead like ECP-2.

We conducted experiments to evaluate the computing performance with three, five, and seven ONOS controllers, when each controller generated the various CPU burden. For the evaluation, we deployed the Mobile CORD (M-CORD) in the single PM (i.e., M-CORD-in-a-Box) which contained 26 CPU cores. The PM had one virtual Head node with 16 CPU cores and one virtual Compute nodes with 16 CPU cores. The Head node ran at least 35 control services (e.g., database, GUI, many synchronizers to manage VNFs, and OpenStack) as Docker [57] containers. In the Head node, we deployed seven Ubuntu Linux VMs to perform ONOS controllers and each VM contained two vCPUs according to the ONOS recommendation. Then, we switched on three, five, or seven VMs and generated the various CPU burden with *stress-ng* tool [58] in each VM. Inside a synchronizer in this environment, we used *Intel MKL benchmark* tool [59] to measure

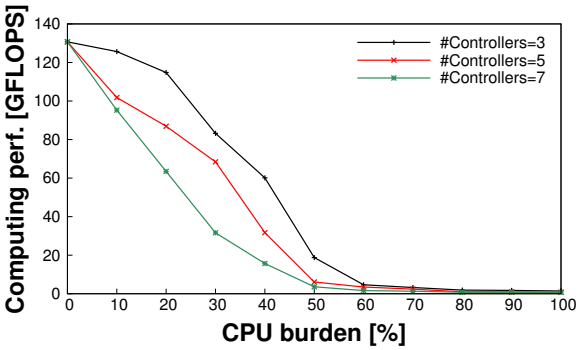



Figure 3.4: The computing performance with the various CPU burden

the Giga floating point operations per seconds (GFLOPS) which is one of the well-known computing performance metrics.

Figure 3.4 shows the experimental result with three, five, and seven VMs which generated the various CPU burden from 0 to 100%. When the CPU burden increased, the computing performance of the synchronizer decreased. The more controllers an ECP exploited in operation, the less computing performance the synchronizer achieved. Since controllers occupied more vCPUs and/or used more CPU capacity, other control services encountered the low computing performance like the synchronizer. Therefore, an ECP should preserve to use the CPU resources to minimize the influence on the other control services.

## 3.5 Discussion

To mitigate the problems faced in ECP-1 and ECP-2, a new type of ECP is necessary. According to the analysis, each ECP type encounters one of two issues: the low immediacy and the high computing overhead. The low immediacy leads to overloaded controllers, which delays the rule installation time. On the contrary, the high computing overhead makes other control services process their operations, slowly. Those problems degrade the performance of DC and breaches SLA. Therefore, new ECPs are essential to resolve the both drawbacks.

# IV

# Proposed Schemes: H-ECP and T-DCORAL

In this chapter, we propose two ECPs, H-ECP and T-DCORAL. First, we define the system architecture and model to propose new ECPs. Next, we formulate the problem which our ECPs try to solve. Then, we propose a Hybrid ECP (H-ECP) and a Threshold-based Dynamic Controller Resource Allocation (T-DCORAL) in detail. Finally, we describe how to implement ECP framework to run H-ECP, T-DCORAL, and the other ECPs.

## 4.1 System Model

Our system considers cutting-edge DC architectures very similar to CORD [40] and the commercial DC architectures proposed by Cisco [41], Microsoft [42], IBM [43], Google [44], and Joyent [45]. In our system, the system architecture contains the following elements: (i) the DP, (ii) the CP, (iii) the management network, (iv) the orchestrator, and (v) the external network. Figure 4.1 shows the system architecture in detail. In this architecture, each element works for the following purposes.

- The DP: all networks and servers (Compute nodes) to run user services (e.g., applications, VMs, and containers) and VNFs (e.g., virtual router, MME, HSS, SGW, and PGW).

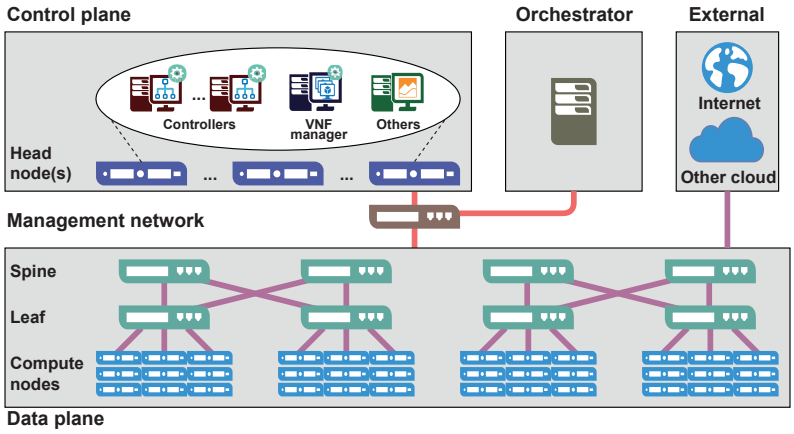



Figure 4.1: System architecture

- The CP: the control logic to manage all elements in the DP by utilizing controllers, VNF managers, and the other control applications.
- Management network: the network to connect among the CP, the DP, and the orchestrator.
- Orchestrator: the orchestration software to control all controllers and Head nodes.
- External networks: the networks outside the DC like Internet or other clouds.

The DP and the CP have several servers and switches as shown in Figure 4.1. The DP consists of OpenFlow switches with the leaf-spine topology like Fat-tree and Compute nodes running countless user services and VNFs. Those devices can communicate with the external networks through gateways or routers. The CP has a few Head nodes containing the following control services: (i) several controllers to manage OpenFlow switches in DP; (ii) VNF manager to manage services running in Compute nodes (e.g., user applications, VMs, containers, and VNFs); (iii) GUI; and (iv) other useful control applications.

The orchestrator manages controllers and Head nodes through the manage-

ment network. The orchestrator monitors the CPU and network load in each controller and each Head node. Also, the orchestrator runs ECPs to rebalance the CP and to resize the controller pool. The orchestrator is possible to switch on and off VMs running controllers in Head nodes. Finally, the orchestrator can even modify the configuration of controller VMs, e.g., the vCPU allocation.

We modeled discrete time series: $T = \{t_1, t_2, \dots\}$. where each element $t_c$ denotes a timeslot and the duration of $t_c$ is $\tau$. Next, we defined the set of $k$ Head nodes $\mathbf{M} = \{M_1, M_2, \dots, M_k\}$, the set of $n$ controllers $\mathbf{C} = \{C_1, C_2, \dots, C_n\}$, and the set of $m$ network devices $\mathbf{S} = \{S_1, S_2, \dots, S_m\}$. Since each VM which runs a controller locates in one of Head nodes, we designed the set of relationship matrices between each Head node and each controller: $\mathbf{R} = \{\mathbf{R_1}, \mathbf{R_2}, \dots, \mathbf{R_k}\}$, where $i$th element matrix $\mathbf{R_i} \in \mathbf{R}$ is defined as

$$\mathbf{R_i} = \begin{bmatrix} r_{i,1} & \cdots & r_{i,n} \end{bmatrix}, \quad r_{i,j} = \begin{cases} 1 & \text{if } C_j \text{ runs in } M_i \\ 0 & \text{otherwise} \end{cases}. \tag{4.1}$$

Likewise, we defined the CP topology, the set of connection matrices between each network device and its master controller: $\mathbf{E} = \{\mathbf{E_1}, \mathbf{E_2}, \dots, \mathbf{E_n}\}$, where $j$th element matrix $\mathbf{E_j} \in \mathbf{E}$ is defined as

$$\mathbf{E_j} = \begin{bmatrix} e_{j,1} & \cdots & e_{j,m} \end{bmatrix}, \quad e_{j,l} = \begin{cases} 1 & \text{if } C_j \text{ is the master controller of } S_l \\ 0 & \text{otherwise} \end{cases}. \tag{4.2}$$

Also, the matrix $\mathbf{O}$ means the number of transmitted and received control messages, which is defined as

$$\mathbf{O} = \begin{bmatrix} \mathbf{O_1} \\ \mathbf{O_2} \\ \vdots \\ \mathbf{O_n} \end{bmatrix} = \begin{bmatrix} o_{1,1} & o_{1,2} & \dots & o_{1,l} \\ o_{2,1} & o_{2,2} & \dots & o_{2,l} \\ \vdots & \vdots & \ddots & \vdots \\ o_{n,1} & o_{n,2} & \dots & o_{n,l} \end{bmatrix}, \tag{4.3}$$

where $\mathbf{O_j}$ denotes the number of transmitted and received control messages only for the controller $C_j$ and $o_{j,m}$ means the number of control messages between the controller $C_j$ and the network device $S_m$.

Let $\mathbf{V} = [v_1 \quad \cdots \quad v_k]$ represent the maximum number of vCPUs for each Head node and $\mathbf{P} = \{\mathbf{P^{(1)}}, \mathbf{P^{(2)}}, \ldots, \mathbf{P^{(k)}}\}$ be the set of vCPU pools for all Head nodes. The $i$th vCPU pool for $M_i$ is defined as

$$\mathbf{P^{(i)}} = \begin{bmatrix} \mathbf{P_1^{(i)}} \\ \mathbf{P_2^{(i)}} \\ \vdots \\ \mathbf{P_n^{(i)}} \end{bmatrix} = \begin{bmatrix} p_{1,1}^{(i)} & p_{1,2}^{(i)} & \cdots & p_{1,v_i}^{(i)} \\ p_{2,1}^{(i)} & p_{2,2}^{(i)} & \cdots & p_{2,v_i}^{(i)} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n,1}^{(i)} & p_{n,2}^{(i)} & \cdots & p_{n,v_i}^{(i)} \end{bmatrix}, \tag{4.4}$$

$$where\ p_{j,u}^{(i)} = \begin{cases} 1 & \text{if } C_j \text{ has vCPU } u \text{ in } M_i \text{ and } r_{i,j} = 1 \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

and $\mathbf{P_j^{(i)}}$ means the vCPU subpool for $C_j$ in $M_i$. Each $C_j$ should occupy at least $\delta$ vCPUs. The vCPU bitmap $\mathbf{B}(M_i)$ which represents the occupation of the vCPUs in $M_i$ is calculated by $\mathbf{B}(M_i) = \mathbf{R_i} \times \mathbf{P^{(i)}}$. Next, we defined the number of vCPUs for $C_j$ as

$$n(vCPUs|C_j) = \sum_{i=1}^{k} (\mathbf{P_j^{(i)}} \times \mathbf{J_{v_i \times 1}}), \tag{4.6}$$

where $\mathbf{J_{v_i \times 1}}$ is a $v_i \times 1$ size all-one matrix, i.e., $[1 \quad \cdots \quad 1]$. Likewise, the number of vCPUs in $M_i$ is defined as

$$n(vCPUs|M_i) = \mathbf{B}(M_i) \times \mathbf{J_{v_i \times 1}}. \tag{4.7}$$

Last, we defined the CPU load, and the network load. First, let $L_{cpu}(\mathbf{C}) =$

$\{L_{cpu}(C_j)|C_j \in \mathbf{C}\}$ be the CPU load for each controller defined as

$$L_{cpu}(C_j) = \frac{\sum_{\text{each vCPU}} vCPU\_load\ [\%]}{n(vCPUs|C_j)}\ [\%], \tag{4.8}$$

where $vCPU\_load$ is a percentile load of each vCPU in $C_j$.

## 4.2 Problem Formulation and Goal

Our goal is three-fold: (i) to enhance the immediacy, (ii) to reduce the computing overhead, and (iii) to minimize the rule installation time. Combining three goals, we formulate the problem which we will solve based on the weighted sum model:

$$min(\overbrace{\omega_1 \sum_{a_k \in \mathbf{A}} T_{a_k}}^{\text{term 1}} + \overbrace{\omega_2 \sum_{M_i \in \mathbf{M}} n(vCPUs|M_i)}^{\text{term 2}} + \overbrace{\omega_3 \sum_{C_j \in \mathbf{C}} L_{cpu}(C_j)}^{\text{term 3}} + \overbrace{\omega_4 \sum_{f \in \mathbf{F}} T_f}^{\text{term 4}}), \tag{4.9}$$

$$\text{s.t. } \omega_1 + \omega_2 + \omega_3 + \omega_4 = 1,$$

where $\omega$ is the weight value, $\mathbf{A}$ is the set of ECP's actions (e.g., rebalancing the CP, expanding the controller pool, shrinking the controller pool, etc.), $a_k$ means one of the ECP's actions, $T_{a_k}$ is the latency to operate the action $a_k$, $\mathbf{F}$ denotes the set of installed flow rules, $f$ represents one flow rule in $\mathbf{F}$, and $T_f$ is the latency to install the flow rule $f$. The term 1 minimizes the latency to adjust the CP, which increases the immediacy. The term 2 and 3 minimize the number of used vCPUs and the CPU load (reducing the CPU resources being used), which is for reducing the computing overhead. Finally, the term 4 minimizes the rule installation time.

## 4.3 Hybrid ECP

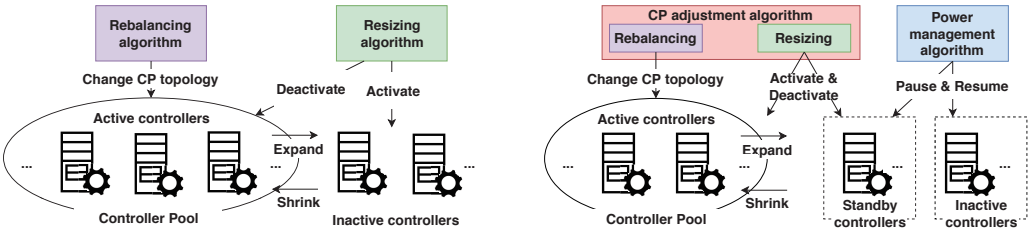
In this section, we propose a Hybrid ECP (H-ECP). To begin with, we introduce an overview of H-ECP and then show the detailed description of H-ECP. Next, we solves some challenges: (i) how to power on/off controllers and (ii) the initial setup of H-ECP in detail. Finally, we explain an operational example of H-ECP.

### 4.3.1 Overview

We define new policies for a new ECP which mitigates the problem we designed in Equation 4.9. (i) The new ECP needs to turn off unnecessary controllers as many as possible (for term 2 and 3 in Equation 4.9). (ii) The new ECP should activate controllers without the step to switch on controllers when the CP demands new active controllers (for term 1 in Equation 4.9). (iii) The new ECP processes control messages as rapid as possible to reduce the rule installation time (for term 4 in Equation 4.9).

According to the above policies, we propose a new ECP, a Hybrid ECP (H-ECP). H-ECP switches off most inactive controllers to achieve the first policy. Among inactive controllers, H-ECP maintains the predefined number of inactive controllers switched on (called as standby controllers). H-ECP activates standby controllers, not inactive controllers, to expand the controller pool without a power-on step, which rapidly adjusts the CP (for the second policy). The rapid adjustment of the CP enables H-ECP to utilize sufficient active controllers. Consequently, H-ECP processes all control messages on schedule, which reduces the rule installation time (for the third policy).

H-ECP forks two threads, the foreground thread to adjust the CP and the background thread to control the power of standby and inactive controllers. To adapt the CP, the foreground thread runs the CP adjustment algorithm which




(a) Previous ECPs (b) H-ECP

Figure 4.2: Overviews of previous ECPs and H-ECP

rebalances the CP and resizes the controller pool. The CP adjustment algorithm considers both active and standby controllers, not inactive controllers. If the load of active controllers exceeds the upper threshold $\eta_u$ due to the increase of the control traffic load, the CP adjustment algorithm activates a standby controller. In the opposite situation, i.e., the load is beneath the lower threshold $\eta_l$, the CP adjustment algorithm transforms from an active controller to a standby controller. Note that the number of standby controllers varies due to the CP adjustment algorithm. To retain the predefined number of standby controllers, H-ECP monitors the number of standby and inactive controllers. With the monitoring results, the background thread performs the power management algorithm, which switches on or off inactive or standby controllers, respectively.

Figure 4.2 illustrates the overviews of existing ECPs and H-ECP. Previous ECPs consist of two algorithms, the rebalancing algorithm and the resizing algorithm (Figure 4.2a). The rebalancing algorithm reforms the CP topology to balance the control traffic load across active controllers, whereas the resizing algorithm activates/deactivates inactive/active controllers for expanding/shrinking the controller pool, respectively. In contrast to the previous ECPs, H-ECP comprises the CP adjustment algorithm and the power management algorithm (Figure 4.2b). The CP adjustment algorithm includes the rebalancing and resizing functions in the foreground thread. The rebalancing function works the same as the rebalancing algorithm in the previous ECPs. The resizing function activates

standby controllers, not inactive controllers, or transmute active controllers to standby controllers. The power management algorithm operates in the background thread to keep the predefined number of standby controllers. This algorithm shuts down standby controllers or boots up inactive controllers, when standby controllers are excessive or insufficient, respectively.

### 4.3.2 Detailed Description of H-ECP

**Types of controllers**

H-ECP defines three types of controllers, (i) the active controller, (ii) the inactive controller, and (iii) the standby controller as follows.

- Active controller: the controller which is powered on and services at least one network device.
- Inactive controller: the controller which is powered off and services no network device.
- Standby controller: the controller which is powered on but services no network device.

According to the definition, the active controller in H-ECP is the same as the active controller defined in previous ECPs, whereas the inactive controller is equal to the inactive controller in ECP-2 [34, 35, 37–39]. Lastly, the standby controller in H-ECP and the inactive controller in ECP-1 [33, 36] are equivalent.

H-ECP should define the minimum number of active controllers $min(Active)$. Exploiting one or two active controllers, H-ECP might experience the low reliability. Aforementioned before, controllers install flow rules into network devices when network devices receive new data flows. In this situation, if the one or two controllers crash, network devices forward no user packet. Therefore, H-ECP

needs to perform more than one or two controllers; we should define $min(Active)$, reasonably.

### Flowchart of H-ECP

Figure 4.3 depicts the flowchart to show how H-ECP works. In fact, we design H-ECP running on the discrete time series including infinite timeslots. We determine the duration of timeslot $\tau$ as the latency to rebalance the CP. Since the rebalancing step spends the shortest time among all operations in H-ECP, we define the duration to finish the rebalancing step as a unit timeslot.

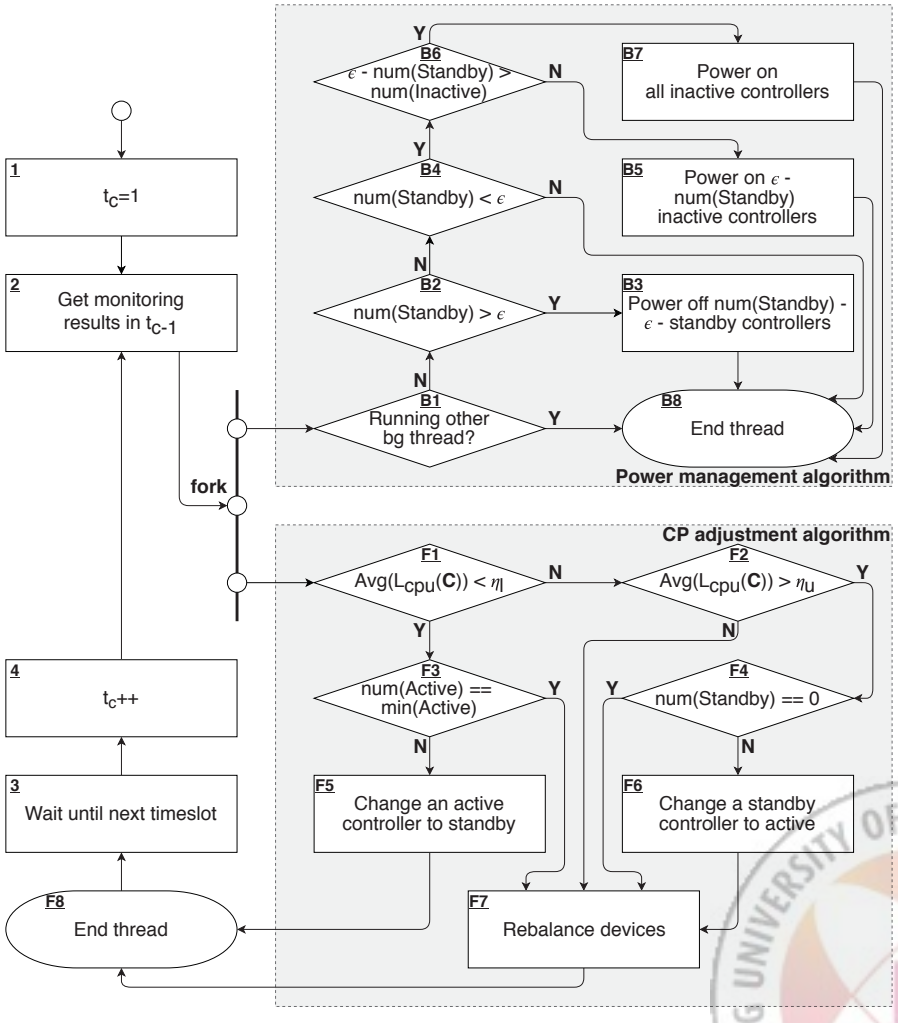



Figure 4.3: The flowchart of H-ECP

In the flowchart, the timeslot $t_c$ is initialized to 0 at the beginning (1). Then, H-ECP gets the monitoring results in $t_c - 1$ which is the timeslot right before the current timeslot (2). The monitoring results include the CPU load of each controller and the number of control messages between each network device and its master controller. Then, H-ECP forks one background thread for the power management algorithm (from B1 to B8) and one foreground thread for the CP adjustment algorithm (from F1 to F7).

The background thread for the power management algorithm checks whether the other background thread is still running or not (B1). If the previous background thread is still running, the new background thread exists (B8). Otherwise, the background thread proceeds the next step. Without this step, switching on inactive controllers or switching off standby controllers occur, redundantly. The power-on and power-off steps spend more than one timeslot. The current background thread then still recognizes insufficient or excessive standby controllers even if the other background threads are running to power on or off controllers. Finally, the current background thread boots up or shuts down inactive or standby controllers, superfluously. To avoid this situation, we make the running background thread singular.

The next step is to perform the power management algorithm as follows:

- If the number of standby controllers ($num(Standby)$) exceeds the predefined number of standby controllers $\epsilon$ (B2), the algorithm starts to power off standby controllers as many as the difference between $\epsilon$ and the $num(Standby)$ (B3).

- If $num(Standby)$ is lower than $\epsilon$ (B4) and the number of inactive controllers ($num(Inactive)$) is insufficient to maintain $\epsilon$ controllers (i.e., $\epsilon - num(Standby) \leq num(Inactive)$ in B6), the algorithm switches on all inactive controllers (B7).

- If $num(Standby)$ is lower than $\epsilon$ (B4) and $num(Inactive)$ is sufficient to keeps $\epsilon$ standby controllers (i.e., $\epsilon - num(Standby) > num(Inactive)$ in B6), the algorithm powers on $\epsilon - num(Standby)$ inactive controllers.

- Otherwise, the algorithm powers on or off no inactive controllers;

Finally, the background thread quits (B8).

The foreground thread runs for the CP adjustment algorithm. In the beginning, the algorithm calculates the average CPU load among active controllers ($L_{cpu}(\mathbf{C})$). After that, the foreground thread operates the CP adjustment algorithm as follows:

- If $L_{cpu}(\mathbf{C})$ is beneath $\eta_l$ the lower threshold (F1) and the number of active controllers ($num(Active)$) is over $min(Active)$ we defined (F3), the algorithm changes an active controller to a standby controller (F5). In this step, all network devices reconnects from the standby controller to the other active controllers.

- If the $L_{cpu}(\mathbf{C})$ is lower than $\eta_l$ (F1) and $num(Active)$ is the same as $min(Active)$, the algorithm runs the rebalancing function (F7).

- If the $L_{cpu}(\mathbf{C})$ is more than $\eta_u$ (F2) and $num(Standby)$ exceeds zero (F4), the algorithm changes a standby controller to an active controller (F6). Then, the algorithm rebalances network devices among active controllers (F7).

- If the $L_{cpu}(\mathbf{C})$ is more than $\eta_u$ (F2) and $num(Standby)$ is zero (F4), the algorithm operates the rebalancing function (F7).

- Otherwise, the algorithm activates or deactivates no controller and just performs the rebalancing function (F7).

Then, the foreground thread exits (F8). After the foreground thread, H-ECP waits until the next timeslot (3) and increase the timeslot index (4).

**Detailed algorithm of H-ECP**

Performing H-ECP with regard to the flowchart Figure 4.3, H-ECP needs the main procedure of H-ECP, the function for the power management algorithm, and the function for the CP adjustment algorithm.

• *Main procedure of H-ECP:*

**Algorithm 4.1** Main procedure of H-ECP

```
1:  boolean isActiveThreadPowerMgmt = false;
2:
3:  procedure run_HECP:
4:  G_a, G_s, G_i ← init(C);
5:  for every timeslot t_c ∈ T do
6:     G_a, G_s, G_i, L_cpu(C), O ← getMonitorResults(t_c − 1);
7:     thread bg_thread = thread_Power_Mgmt(G_s, G_i);
8:     thread fg_thread = thread_CP_Adj(L_cpu(C), O, G_a, G_s);
9:     fork bg_thread and fg_thread;
10:    join fg_thread;
11: end for
```

Algorithm 4.1 shows the pseudo code of the H-ECP main procedure. For the main procedure, H-ECP defines the global variable *isActiveThreadPowerMgmt* to check whether the background thread is operating or not (line 1). When the background thread starts, this variable is set to true. Otherwise, it goes to false. The main procedure *run_HECP* runs the initialize function and each algorithm as a thread for every timeslot (line 3-11). The main procedure initially calls *init* the initialize function. The *init* sets the initial environment of H-ECP and returns $\mathbf{G_a}$, $\mathbf{G_s}$, and $\mathbf{G_i}$, which are the set of active controllers, standby controllers, and inactive controllers, respectively (line 4). At the timeslot $t_c$, *run_HECP* gets monitoring results including the type for each controller ($\mathbf{G_a}, \mathbf{G_s}, \mathbf{G_i}$), the CPU load ($L_{cpu}(\mathbf{C})$), and the number of control messages $\mathbf{O}$ for all controllers, in timeslot

$t_c - 1$ the right before timeslot (line 6). After that, the main procedure makes a new thread *bg_thread* for the power management algorithm as a background thread (line 7). Then, *run_HECP* defines a new thread *fg_thread* for the CP adjustment algorithm as a foreground thread (line 8). After the definitions of two threads, the main procedure forks two threads and performs them, simultaneously (line 9). Finally, the *run_HECP* joins the *fg_thread*, which waits until finishing the *fg_thread*.

• *Power management algorithm:*

**Algorithm 4.2** The background thread for the power management algorithm in H-ECP

```
1: thread thread_Power_Mgmt(G_s, G_i):
2: if isActiveThreadPowerMgmt == true then
3:    return; // exit this thread
4: else
5:    isActiveThreadPowerMgmt =true;
6:    if num(G_s) > ε then
7:       Ĉ ← pick any num(G_s) − ε standby controllers in G_s;
8:       G_s = G_s − Ĉ; G_i = G_i ∪ Ĉ;
9:       power off Ĉ;
10:   else if num(G_s) < ε && ε − num(G_s) ≥ num(G_i) then
11:      Ĉ ← pick all inactive controllers in G_i;
12:      G_i = φ; G_s = G_s ∪ Ĉ;
13:      power on Ĉ;
14:   else if num(G_s) < ε && ε − num(G_s) < num(G_i) then
15:      Ĉ ← pick any ε − num(G_s) inactive controllers in G_i;
16:      G_i = G_i − Ĉ; G_s = G_s ∪ Ĉ;
17:      power on Ĉ;
18:   end if
19: end if
```

When the main procedure calls the thread *thread_Power_Mgmt*, Algorithm 4.2 starts. To begin with, the thread checks whether the global variable *isActiveThreadPowerMgmt* is true or false. If it is true, i.e., the other background thread is running due to the long delay to power on/off controllers, the thread exits (line 3). Otherwise, the thread runs the power management algorithm (line 4-19). At the beginning, the thread sets *isActiveThreadPowerMgmt* to true (line

5). Then, the thread checks how many standby controllers are running now. If the size of $\mathbf{G_s}$ ($num(\mathbf{G_s})$) is over the $\epsilon$, the thread picks $num(\mathbf{G_s}) - \epsilon$ standby controllers in $\mathbf{G_s}$ (line 7). The thread pops those selected controllers from $\mathbf{G_s}$ and pushes them into $\mathbf{G_i}$ (line 8). Then, the thread powers off the selected standby controller (line 9). Next, if $num(\mathbf{G_s})$ is under $\epsilon$ and $\epsilon - num(\mathbf{G_s})$ is over $num(\mathbf{G_i})$, the thread selects all controllers in $\mathbf{G_i}$ (line 11). The reason is that inactive controllers are insufficient to keeps $\epsilon$ standby controllers although all inactive controllers become standby controllers. The thread sets $\mathbf{G_i}$ to the empty set $\phi$ and then adds all inactive controllers into $\mathbf{G_s}$ (line 12). After that, the thread switches on all inactive controllers (line 13). Last, if $num(\mathbf{G_s})$ is under $\epsilon$ and $\epsilon - num(\mathbf{G_s})$ is lower than $num(\mathbf{G_i})$, i.e,. inactive controllers are sufficient to maintain $\epsilon$ standby controllers, the thread selects $\epsilon - num(\mathbf{G_s})$ inactive controllers in $\mathbf{G_i}$ (line 15). The thread deletes the selected inactive controllers from $\mathbf{G_i}$ and pushes them into $\mathbf{G_s}$ (line 16). Finally, the thread boots up the selected inactive controllers (line 17) and then quits.

• *CP adjustment algorithm:*

**Algorithm 4.3** The foreground thread for the CP adjustment algorithm in H-ECP

```
1: thread thread_CP_Adj(L_cpu(C), O, G_a, G_s):
2: if avg(L_cpu)(G_a) < η_l && num(G_a)! = min(Active) then
3:    run_Shrink(L_cpu(C), O, G_a, G_s);
4: else if avg(L_cpu)(G_a) > η_u && num(G_s)! = 0 then
5:    run_Expand(L_cpu(C), O, G_a, G_s);
6: else
7:    run_Rebalancing(L_cpu(C), O, G_a);
8: end if
```

Algorithm 4.3 depicts the *thread_CP_Adj* which is the foreground thread to run the CP adjustment algorithm. The thread compares the average CPU load for all active controllers ($avg(L_{cpu}(\mathbf{G_a}))$) with each threshold, $\eta_u$ or $\eta_l$. If $avg(L_{cpu}(\mathbf{G_a}))$ is under $\eta_l$ and $num(\mathbf{G_a})$ is not equal to $min(Active)$, the

**Algorithm 4.4** The function to shrink the controller pool in H-ECP

1: **function** run_Shrink($L_{cpu}(\mathbf{C}), \mathbf{O}, \mathbf{G_a}, \mathbf{G_s}$):
2: $C_{tmp} \leftarrow$ pick the controller which experiences the lowest CPU load in $\mathbf{G_a}$;
3: $\mathbf{G_s} = \mathbf{G_s} \cup \{C_{tmp}\}$; $\mathbf{G_a} = \mathbf{G_a} - \{C_{tmp}\}$;
4: $\mathbf{S_{tmp}} \leftarrow$ network devices served by $C_{tmp}$;
5: sort $\mathbf{S_{tmp}}$ as descending order according to the number of control messages;
6: **for each** $S_l \in \mathbf{S_{tmp}}$ **do**
7: $L_{cpu}(S_l) = L_{cpu}(C_{tmp}) \times (o_{tmp,l} / \sum \mathbf{O_{tmp}})$;
8: $C_{low} \leftarrow$ pick the controller suffering the lowest $L_{cpu}(\mathbf{C})$ in $\mathbf{G_a}$;
9: $L_{cpu}(C_{low}) = L_{cpu}(C_{low}) + L_{cpu}(S_l)$;
10: change the master controller of $S_l$ to $C_{low}$;
11: **end for**

thread runs *run_Shrink* function which shrivels the controller pool (line 2-3). If $avg(L_{cpu}(\mathbf{G_a}))$ is above $\eta_u$ and $num(\mathbf{G_s})$ is not zero, the thread operates *run_Expand* function which grows the controller pool (line 4-5). Otherwise, the thread performs *run_Rebalancing* function to balance network devices among active controllers. For example, $num(\mathbf{G_a})$ and $min(Active)$ are equivalent. In this situation, there is no possible active controller to become a standby controller, since H-ECP should use at least $min(Active)$ controllers. Likewise, H-ECP is impossible to activate a standby controller if $num(\mathbf{G_s})$ is zero. Thus, H-ECP runs *run_Rebalancing* function even if $avg(L_{cpu}(\mathbf{G_a}))$ is over or under each threshold.

Algorithm 4.4 shows the function *run_Shrink* to deflate the controller pool called by the foreground thread. This function initially picks the controller $C_{tmp}$ which suffers from the lowest CPU load (line 2). Then, the function pushes $C_{tmp}$ to $\mathbf{G_s}$ and then pops $C_{tmp}$ from $\mathbf{G_a}$ (line 3). After that, the function gets network devices served by $C_{tmp}$ and save it into $\mathbf{S_{tmp}}$ (line 4). The function sorts the $\mathbf{S_{tmp}}$ as a descending order with regards to the number of control messages (line 5). Next, the function runs loop statements (line 6-11) to distribute $\mathbf{S_{tmp}}$ to active controllers except for $C_{tmp}$. To reattach each network device $S_l$ in $\mathbf{S_{tmp}}$, the function calculates the estimated CPU load of $S_l$ ($L_{cpu}(S_l)$) how much $S_l$ afflicts

the CPU load to $C_{tmp}$ with the equation:

$$L_{cpu}(S_l) = L_{cpu}(C_{tmp}) \times \frac{o_{tmp,l}}{\sum \mathbf{O_{tmp}}}. \tag{4.10}$$

In this equation, $L_{cpu}(S_l)$ denotes the estimated CPU load from $S_l$, $L_{cpu}(C_{tmp})$ is the CPU load of $C_{tmp}$, $o_{tmp,l}$ means the number of control messages between $C_{tmp}$ and $S_l$, and $\sum \mathbf{O_{tmp}}$ represents the total number of control messages in $C_{tmp}$. This equation means that $L_{cpu}(S_l)$ is the CPU load of its master controller proportional to the transceived control messages. Since the number of control messages is proportional to the CPU load of a controller, each network device approximately weighs its master controller down with $L_{cpu}(S_l)$ [34, 35]. Thus, the function takes advantage of the estimated CPU load of $S_l$ calculated by the equation (line 7). Then, the function selects the controller ($C_{low}$) which experiences the lowest $L_{cpu}(\mathbf{C})$ in $\mathbf{G_a}$ (line 8). The function updates the $L_{cpu}(C_{tmp})$ and the changes the master controller of $S_l$ to $C_{low}$ (line 9-10). After traversing all network devices, the function is finished.

To expand the controller pool, the CP adjustment algorithm runs *run_Expand* function shown in Algorithm 4.5. At first, *run_Expand* function selects a controller $C_{tmp}$ in $\mathbf{G_s}$ (line 2). Next, the function removes $C_{tmp}$ from $\mathbf{G_s}$ and then pushes $C_{tmp}$ to $\mathbf{G_a}$ (line 3). The function sets the CPU load of $C_{tmp}$ ($L_{cpu}(C_{tmp})$) to zero. Finally, the function calls *run_Rebalancing* function which rebalances network devices among active controllers in $\mathbf{G_a}$.

The last function in the CP adjustment algorithm is *run_Rebalance* func-

**Algorithm 4.5** The function to expand the controller pool in H-ECP

1: **function** run_Expand($L_{cpu}(\mathbf{C}), \mathbf{O}, \mathbf{G_a}, \mathbf{G_s}$):
2: $C_{tmp} \leftarrow$ pick one controller in $\mathbf{G_s}$;
3: $\mathbf{G_s} = \mathbf{G_s} - \{C_{tmp}\}$; $\mathbf{G_a} = \mathbf{G_a} \cup \{C_{tmp}\}$;
4: $L_{cpu}(C_{tmp}) = 0$;
5: run_Rebalancing($L_{cpu}(\mathbf{C}), \mathbf{O}, \mathbf{G_a}$);

**Algorithm 4.6** The function to rebalance the CP in H-ECP

```
1: function run_Rebalance(L_cpu(C), O, G_a):
2: G_over ← controllers in G_a which experiences L_cpu > avg(L_cpu(G_a));
3: G_under ← controllers in G_a which experiences L_cpu < avg(L_cpu(G_a));
4: sort G_over as descending order with regard to L_cpu(G_a);
5: sort G_under as ascending order with regard to L_cpu(G_a);
6: for each C_tmp/over ∈ G_over do
7:    S_tmp ← network devices served by C_tmp;
8:    sort S_tmp as descending order regarding the number of control messages;
9:    for each C_tmp/under ∈ G_under do
10:      for each S_l ∈ S_tmp do
11:         L_cpu(S_l) = L_cpu(C_tmp/over) × (o_tmp/over,l / ∑ O_tmp);
12:         if L_cpu(C_tmp/over) − L_cpu(S_l) < avg(L_cpu(G_a)) then
13:            continue;
14:         else if L_cpu(C_tmp/under) + L_cpu(S_l) > avg(L_cpu(G_a)) then
15:            continue;
16:         else if L_cpu(C_tmp/under) + L_cpu(S_l) ≤ avg(L_cpu(G_a)) then
17:            change the master controller of S_l to C_tmp/under;
18:            L_cpu(C_tmp/under) = L_cpu(C_tmp/under) + L_cpu(S_l);
19:         end if
20:      end for
21:   end for
22: end for
```

tion which distributes network devices to active controllers in $\mathbf{G_a}$ (Algorithm 4.6) Rebalancing network devices, the function tries to experience the same or similar CPU load among controllers. To achieve this, the function estimates the CPU load for each network device. Then, the function migrates network devices managed by over-subscribed controllers to under-subscribed controllers.

The detailed description of *run_Rebalance* operates as follows. First, the function categorizes active controllers into two groups $\mathbf{G_{over}}$ and $\mathbf{G_{under}}$, where active controllers are over and under $avg(L_{cpu}(\mathbf{G_a})$, respectively (line 2-3). Controllers in $\mathbf{G_{over}}$ are over-subscribed controllers, while controllers in $\mathbf{G_{under}}$ are under-subscribed controllers. Then, the function sorts $\mathbf{G_{over}}$ as a descending order and $\mathbf{G_{under}}$ as an ascending order according to $L_{cpu}(\mathbf{G_a})$ (line 4-5). With two groups, the function starts to rebalance network devices with triple loops (line

6-22). The first loop in the function sequentially selects the controller $C_{tmp/over}$ in $\mathbf{G_{over}}$, repeatedly (line 6-22). In the first loop, the function assigns network devices controlled by $C_{tmp/over}$ to the variable $\mathbf{S_{tmp}}$ (line 7) and sorts $\mathbf{S_{tmp}}$ as a descending order with regard to the number of control messages (line 8). After that, the function executes the second loop with $C_{tmp/over}$ and $\mathbf{S_{tmp}}$ (line 9-21). The second loop selects the controller $C_{tmp/under}$ in $\mathbf{G_{under}}$, iteratively (line 9). After selecting $C_{tmp/under}$, the function starts the last loop to traverse $\mathbf{S_{tmp}}$ (line 10-20). The third loop sequentially gets the network device $S_l$ in $\mathbf{S_{tmp}}$ (line 10), and then calculates the estimated CPU load from $S_l$ by using Equation 4.10 (line 11). Next, the third loop has three branches:

- If $C_{tmp/over}$ in $\mathbf{G_{over}}$ becomes an under-subscribed controller due to the detachment of $S_l$ from $C_{tmp/over}$ (i.e., $L_{cpu}(C_{tmp/over}) - L_{cpu}(S_l) < avg(L_{cpu}(\mathbf{C}))$), there is no operation and continue to get the next network device in $\mathbf{S_{tmp}}$ (line 12-13).

- If $C_{tmp/under}$ in $\mathbf{G_{under}}$ becomes an over-subscribed controller because of the attachment of $S_l$ to $C_{tmp/under}$ (i.e., $L_{cpu}(C_{tmp/under}) + L_{cpu}(S_l) > avg(L_{cpu}(\mathbf{G_a}))$), there is no operation and continue to get the next network device in $\mathbf{S_{tmp}}$ (line 14-15).

- If $C_{tmp/under}$ in $\mathbf{S_{tmp}}$ is still an under-subscribed controller with servicing sl (i.e., $L_{cpu}(C_{tmp/under}) + L_{cpu}(S_l) \leq avg(L_{cpu}(\mathbf{G_a}))$), the third loop changes the master controller of $S_l$ to $C_{tmp/under}$ (line 16-17). Then, the third loop updates $L_{cpu}(C_{tmp/under})$ (line 18).

After the third loop, the finish to traverse $\mathbf{S_{tmp}}$ (line 19), the function goes to the second loop and picks a next controller in $\mathbf{G_{under}}$ (go to line 9). Of course, the functions come back to the first loop when finishing the second loop (line 22); the first loop selects the next controller in $\mathbf{G_{over}}$ (go to line 6).

### 4.3.3 Challenges

**How to Power On/Off Controllers?**

Essentially, we need to define the way to power on and off controllers to maintain the $\epsilon$ standby controllers. However, previous ECPs in ECP-2 [34, 35, 37–39] suggest no way how to switch on/off a controller. Therefore, we define the straightforward process to switch on and off a controller. To shut down a controller, H-ECP switches off controller software, and then pauses the VM which operated controller software. On the contrary, H-ECP resumes the paused VM and then brings up controller software.

Of course, we can use other ways to manage the power of each controller. The first way is to explicitly power on and off a VM, not pausing and resuming. Exploiting the first way, H-ECP needs to boot up the VM with (i) booting Operating System (OS); (ii) starting controller software. However, the first way spends more time than the way we proposed, because booting up OS takes longer than the process to resume the VM. The second way is to pause and resume the VM like the way we proposed before. Comparing with our way, the second way directly pauses and resumes the VM without switching on and off controller software, respectively. However, the second way causes the inconsistency problem among controllers. To see what happens with the second way, we tested to directly pause and then resume the software controller ONOS version 1.10. After resuming the directly paused VM running ONOS controller, this ONOS controller included older state than the other ONOS controllers. Finally, the resumed ONOS controller became an inconsistent controller; the controller encountered many unexpected errors. Therefore, we used our way rather than the above two ways.

**Initial Setup of H-ECP**

To run H-ECP, we initialize the environment first. Before starting H-ECP, we need to define $\epsilon$ and $min(Active)$. The orchestrator which operates the main procedure of H-ECP is possible to pause and resume VMs and to switch on and off controller software in each VM. On top of the above environment, the main procedure of H-ECP *run_HECP* calls the function *init* (Algorithm 4.7). The *init* categorizes all controllers into three sets, the set of active controllers ($\mathbf{G_a}$), the set of standby controllers ($\mathbf{G_s}$), and the set of inactive controllers ($\mathbf{G_i}$). In the ($\mathbf{G_a}$) and ($\mathbf{G_s}$), there are $min(Active)$ controllers and $\epsilon$ controllers, respectively (line 2-3). The third set ($\mathbf{G_i}$) involves the other controllers (line 4). If controllers in ($\mathbf{G_a}$) and ($\mathbf{G_s}$) are paused, the function resumes the controllers. On the contrary, the function powers on the controllers when the controllers are shut down (line 5). Next, the function changes the state of controllers in ($\mathbf{G_i}$) to the pause state (line 6). After that, the function attaches the same number of network devices to each controller in ($\mathbf{G_a}$) (line 7-9). After the initialization, the function returns ($\mathbf{G_a}$), ($\mathbf{G_s}$), and ($\mathbf{G_i}$).

**Algorithm 4.7** The initialization function in H-ECP

```
1: function init(C):
2: G_a ← min(Active) controllers in C;
3: G_s ← ε controllers in C;
4: G_i ← the other controllers;
5: resume or power on G_a and G_s controllers;
6: pause G_i;
7: for each C_tmp ∈ G_a do
8:    S_tmp ← num(S)/min(Active) network devices;
9:    assign S_tmp to C_tmp as a master controller;
10: end for
11: return G_a, G_s, G_i;
```

### 4.3.4 An Operational Example of H-ECP

Figure 4.4 shows an operational example of H-ECP with nine controllers during 23 timeslots. We define that $min(Active)$ is three and $\epsilon$ is four. In this figure, the green box B, the blue box S, and the red box E mean the step to rebalance the CP, the step to shrink the controller pool, and the step to expand the controller pool, respectively. The yellow box R and the purple box P represent the step to resume controllers and the step to pause controllers, respectively. Besides, the numbers of active, standby, and inactive controllers are *#A ctrls*, *# SB ctrls*, and *# I ctrls*, respectively. Finally, *CP Adj* means the foreground thread running the CP adjustment algorithm, whereas *Pwr Mg* denotes the background thread running the power management algorithm.

The first phase of the example in Figure 4.4 operates as follows (timeslot from 1 to 8). First, the average CPU load for all active controllers exceeds $\eta_u$ from the timeslot 1 to 4. Since standby controllers are sufficient in the time 1 through 5, H-ECP expands the controller pool from the timeslot 2 to 5. At the timeslot 2, the number of standby controllers is three; H-ECP starts to resume one inactive controller. Next, the average CPU load for all controllers is over $\eta_u$ again at the timeslot 5. Even if H-ECP should activate one standby controller, H-ECP performs the rebalancing function at the timeslot 6. The reason is that there is no standby controller at timeslot 6. Next, the step to resume one controller is

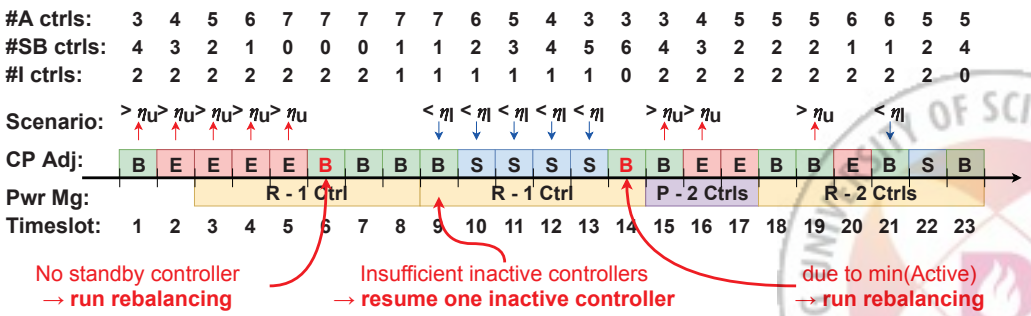



Figure 4.4: An operational example of H-ECP

terminated at the timeslot 8; H-ECP has one more standby controller.

The second phase of the example in Figure 4.4 runs as follows (the timeslot 9 to 17). At the timeslot 9, H-ECP needs to switch on three standby controllers, because H-ECP has one standby controller. However, H-ECP resumes only one inactive controllers due to insufficient inactive controllers. From the timeslot 9 to 12, the average CPU load is beneath $\eta_l$. Consequently, H-ECP shrinks the controller pool during the timeslot 10 to 13. H-ECP also has to shrink the controller pool at the timeslot 14 due to the average CPU load of active controllers lower than $\eta_l$ at the timeslot 13. H-ECP, however, rebalances the controller pool because $min(Active)$ active controllers should be running. At the timeslot 14, the step to resume one inactive controller is completed, which adds one more standby controller. The number of standby controllers is over $\epsilon$ by two at the timeslot 14. As a result, H-ECP pauses two standby controllers at the timeslot 15. During the step to pause some standby controllers, H-ECP cannot use the standby controllers. Therefore, the number of inactive and standby controllers immediately increases and decreases at the timeslot 15, respectively. Similar to the previous timeslots, H-ECP expands the controller pool at the timeslot 16 and 17 because of the excessive average CPU load greater than $\eta_u$ at the timeslot 15 and 16. At the timeslot 17, the step to pause two standby controllers completes.

The third phase of the example in Figure 4.4 is executed as follows (the timeslot 18 to 23). At the timeslot 18, there are only two standby controllers; H-ECP resumes two inactive controllers. Next, H-ECP expands and shrinks the controller pool at the timeslot 20 and 22, since the average CPU load is over and under each threshold, respectively. After finishing the step to resume two inactive controllers (the timeslot 23), H-ECP grabs two more standby controllers.

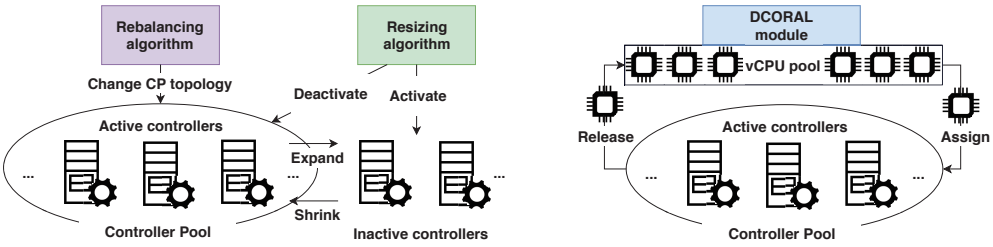
## 4.4 Threshold-based Dynamic Controller Resource Allocation

In this section, we propose Threshold-based Dynamic Controller Resource Allocation (T-DCORAL). First, we explain an overview of T-DCORAL and then depict how T-DCORAL performs in detail. Next, we describe challenges: (i) how to assign a vCPU to a controller in runtime, (ii) the initial setup of T-DCORAL, and (iii) how to deploy network devices across multiple controllers running on multiple Head nodes. Finally, we explain an operational example of T-DCORAL.

### 4.4.1 Overview

To solve the problem we formulated in Equation 4.9, we define the new policies for a new ECP. (i) The ECP should not switch on/off controllers for minimizing the latency to adjust the CP (for term 1 in Equation 4.9). (ii) The ECP imposes upon the minimum number of controllers to reduce the computing overhead (for term 2 and 3 in 4.9). (iii) Those minimal controllers should have the performance possible to process all control messages as fast as possible (for term 4 in 4.9).

Based on the above policies, we propose a new EPC, T-DCORAL. Basically, T-DCORAL exploits the minimal controllers that are not switched on/off (for the first and second policies). The crux of T-DCORAL is to accelerate/decelerate the CP, not resizing the controller pool to handle an abrupt change of the control traffic (for the third policy). For an acceleration/deceleration, T-DCORAL assigns/reclaims a vCPU to/from a controller in runtime. In fact, the more vCPUs are on each controller, the more control messages can be processed (i.e., the higher controller throughput) [34, 35]. That is the reason why T-DCORAL allocates vCPUs in runtime.




(a) Previous ECPs (b) T-DCORAL

Figure 4.5: Overviews of previous ECPs and T-DCORAL

Literally, T-DCORAL accelerates/decelerates the CP according to thresholds. T-DCORAL has two threshold values: the upper threshold ($\eta_u$) and the lower threshold ($\eta_l$). With the CPU load or network load more than the upper threshold, most existing ECPs, e.g., ElastiCon, EDES, and EAS, expand the controller pool. In contrast, those existing ECPs shrink the controller pool when the CPU load or network is beneath the lower threshold. Similarly, T-DCORAL accelerates the CP by assigning a vCPU to appropriate controllers which experience the CPU load more than $\eta_u$. On the contrary, T-DCORAL decelerates the CP by bringing back a vCPU from suitable controllers which suffers from the CPU load lower than $\eta_l$.

Figure 4.5 shows overviews of existing ECPs and T-DCORAL. Existing ECPs consist of the rebalancing algorithm and the resizing algorithm (Figure 4.5a). However, T-DCORAL comprises only one module, T-DCORAL module. This module is responsible for the management of the vCPU pool which includes spare vCPUs. Also, this module assigns/reclaims spare vCPUs to/from target controllers to accelerate/decelerate the CP. The orchestrator involves T-DCORAL module same as H-ECP algorithms.

### 4.4.2 Detailed Description of T-DCORAL

Figure 4.6 shows the flowchart of T-DCORAL. At the very beginning, the timeslot index $t_c$ is set to 1 (1). After that, T-DCORAL runs the first step which

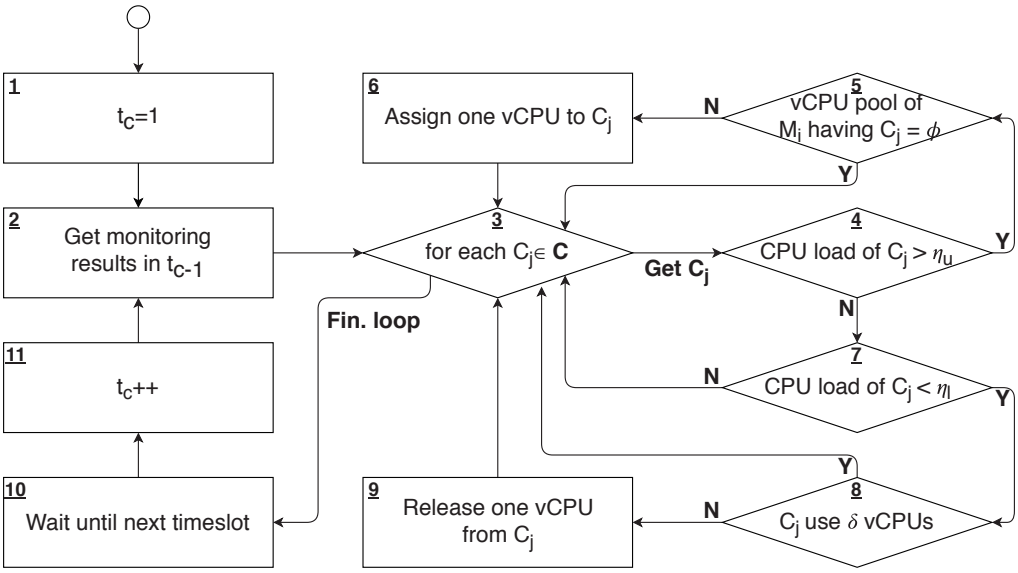



Figure 4.6: The flowchart of T-DCORAL

gets monitoring results in the right before timeslot $t_c - 1$ (2). With the monitoring results, T-DCORAL operates the following steps for all controllers (3):

- If the CPU load of $C_j$ exceeds $\eta_u$ (4) and the vCPU pool in $M_i$ which runs the $C_j$ is not empty (5), T-DCORAL assigns a vCPU to $C_j$ (6).
- If the CPU load of $C_j$ is beneath $\eta_l$ (7) and $C_j$ occupies vCPUs more than $\delta$, which is the minimum number of vCPUs each controller should occupy (8), T-DCORAL reclaims a vCPU from $C_j$ (9).
- Otherwise (i.e., if the CPU load of $C_j$ exceeds $\eta_u$ and the vCPU pool in $M_i$ which runs $C_j$ is empty; if the CPU load of $C_j$ is below to $\eta_l$ and $C_j$ utilizes $\delta$ vCPUs; if the CPU load of $C_j$ is in between two thresholds), T-DCORAL runs no action.

After running the above steps for all controllers, T-DCORAL waits until the next timeslot (10) and increases the timeslot index (11). Finally, T-DCORAL goes to the first step and run the above all steps again.

Algorithm 4.8 shows the pseudo code of T-DCORAL to explain how to im-

**Algorithm 4.8** Main procedure in T-DCORAL

```
1: procedure run_TDCORAL:
2: for every timeslot t_c ∈ T do
3:    L_cpu(C) ← getMonitorResults(t_c − 1); init D ← [d_1...d_n];
4:    for each C_j ∈ C do
5:       d_j ← (L_cpu(C_j) > η_u) ? 1 : (L_cpu(C_j) < η_l) ? −1 : 0;
6:    end for
7:    for each R_i ∈ R do
8:       release_vCPU(R_i, D XOR (−1) · R_i);
9:       assign_vCPU(R_i, D XOR R_i);
10:   end for
11: end for
12:
13: function release_vCPU(R_i, D̂):
14: for each d_j ∈ D̂ & d_j = −1 & n(vCPUs|C_j) > δ do
15:   u = Last 1 in B(M_i); Release u of C_j in M_i; p^(i)_{j,u} ← 0;
16: end for
17:
18: function assign_vCPU(R_i, D̂):
19: for each d_j ∈ D̂ & d_j = 1 & n(vCPUs|M_i) < v_i do
20:   w = First 0 in B(M_i); Assign w to C_j in M_i; p^(i)_{j,w} ← 1;
21: end for
```

plement T-DCORAL in detail. The pseudo code runs on the T-DCORAL module in Figure 4.5b. The procedure *run_TDCORAL* runs T-DCORAL in every timeslot $t_c \in T$ (line 2-11). At $t_c$, this procedure initially gets the CPU load of each controller monitored at the right before timeslot $t_c - 1$, and defines an array $\mathbf{D} = [d_1...d_n]$ (line 3). Next, T-DCORAL classifies controllers into three groups: (i) $L(C_j)$ exceeds $\eta_u$; (ii) $L(C_j)$ is beneath $\eta_l$; (iii) the others. If a $C_j$ is corresponds to the first (second) group, T-DCORAL marks $d_j$ as 1 (-1). Otherwise, $d_j$ is 0 (line 5). Finally, T-DCORAL calls *release_vCPU* and *assign_vCPU* for each controller which corresponds to the first ($d_j = 1$) and second group ($d_j = -1$), respectively (line 8-9).

The *release_vCPU* and *assign_vCPU* release and assign a vCPU from and to target controllers, respectively (line 13-21). Releasing a vCPU from $C_j$ which owns vCPUs more than $\delta$, the *release_vCPU* searches the vCPU $u$ which is the

last 1 in $\mathbf{B}(M_i)$. Then, the function releases the vCPU $u$ from $C_j$ and updates the vCPU pool (line 15). On the other hand, the *assign_vCPU* operates to assign a vCPU to $C_j$, when the vCPU pool is not empty, i.e., $n(vCPUs|M_i) < v_i$. To assign a vCPU to $C_j$, the function figures out the vCPU $w$ which is the first 0 in $\mathbf{B}(M_i)$. Finally, the function assigns the vCPU $w$ to $C_j$ and updates the vCPU pool (line 20).

### 4.4.3 Challenges

**How to Assign a vCPU in Runtime?**

To allocate vCPU in runtime, we found two ways. The first way is to use *CPU HotPlug* functions supported by VM Monitors (VMMs) such as Oracle VM VirtualBox [66]. The next way is to change Linux system files, which specify target vCPUs as online/offline in each VM. For example, let the vCPU pool contains $n$ vCPUs for $m$ VMs running Ubuntu. Then, we initially allow that each VM can take advantage of all $n$ vCPUs. To exploit $k$ vCPUs for each VM ($mk \leq n$), we then specify $n - k$ vCPUs as offline and $k$ vCPUs as online. For specifying vCPUs as online or offline, we modify Linux system file *online*[1]. In this dissertation, we use the second way to support all VMMs.

**Initial Setup of T-DCORAL**

Starting T-DCORAL, we need to set up an initial environment. To begin with, we should decide the minimum number of controllers, and then turn on those controllers which are always running. Next, the orchestrator is possible to allocate vCPUs to controllers in runtime, since this is the basic operation of T-DCORAL. In addition, each controller initially occupies the minimum number of vCPUs $\delta$. When deploying controllers into multiple Head nodes, we considered the number of vCPUs for each Head nodes. The more vCPUs a Head node has,

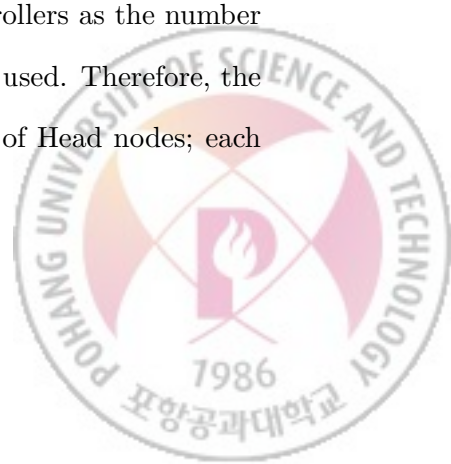
[1]For Ubuntu OS, those files are located in */sys/devices/system/cpu/cpu<index>* directory.

the more controllers the Head node runs. Finally, we need to assign each network device to an appropriate controller as a master controller. The deployment schemes of controllers and network devices will explain as belows.

**How to Deploy Controllers into Multiple Head Nodes?**

To deploy multiple controllers to multiple Head nodes, we define the following policies.

- Case 1 - there is one Head node: Simply, the Head node runs all minimal controllers.
- Case 2 - the minimum number of controllers exceeds the number of Head nodes: We fairly deploy controllers to Head nodes according to the number of vCPUs. If Head nodes have the same number of vCPUs, each Head node runs the same number of controllers. After the assignmnet, the remaining controllers operate in the randomly selected Head nodes. If Head nodes have the different number of vCPUs, we figure outs the number of vCPUs for each Head node, and then deploy controllers to Head nodes according to the proportion to the number of vCPUs.
- Case 3 - the minimum number of controllers is equal to the number of Head nodes: Each Head node operates each controller.
- Case 4 - the minimum number of controllers is lower than the number of Head nodes: We define the minimum number of controllers as the number of Head nodes. If not, some of Head nodes cannot be used. Therefore, the minimum number of controllers is set to the number of Head nodes; each Head node performs each controller.



**How to Deploy Network Devices into Multiple Head Nodes?**

The goal of this scheme is to fairly distribute network devices according to the number of vCPUs each controller can occupy. In the unfair distribution situation, some controllers experience more CPU load than other controllers due to the increase of the control traffic load. Of course, T-DCORAL can assign more vCPUs to the controllers to accelerate them. However, if the CPU pool of the Head node which runs the controllers is empty, the rule installation time from the controllers may delay. Consequently, some network devices managed by the controllers forward data packets more slowly than the other network devices serviced by the other controllers. To avoid this situation, this scheme deploys network devices fairly according to the number of vCPUs and controllers which each Head node includes.

Assigning network devices to appropriate controllers, T-DCORAL leverages the metric, the number of network devices for each controller. In SD-DCNs, each network device has a random traffic pattern [48–56], which changes the control traffic load from the network device to its master controller, rapidly. Because of this problem, we assume that each network device transmits and receives the control traffic with the same random distribution. In other words, each network device weighs its master controller with the stochastically same control traffic load. Thus, we used the number of network devices as the metric to deploy network devices to each controller.

When deploying network devices according to the above metric, we initially calculate how many vCPUs each controller can fairly utilize. When controllers perform on multiple Head nodes for T-DCORAL, each controller can occupy a different number of vCPUs. For instance, Controller A and Controller B are operating in the Head node 1 and the Head node 2, respectively. When the Head node 1 and the Head node 2 includes five and seven CPU cores, Controller A and

the Controller B can use five and seven vCPUs, respectively. If controllers are running in the same Head node, we assume that each controller exploits the same number of vCPUs for the fairness. Let a single Head node perform two controllers with eight CPU cores. Then, we simply assume that each controller utilizes four vCPUs for each, even if T-DCORAL can allocate a different number of vCPUs for the controller fairness. Since the fairness makes each controller encounter the same or similar CPU load, we consider it when deploying network devices.

Next, we finally calculate how many network devices each controller should service, i.e., the deployment of network devices. In fact, the more vCPUs each controller utilizes, the more throughput each controller can achieve [34,35]. Therefore, we distribute network devices according to the proportion of vCPUs each controller can utilize. The number of network devices for a controller $C_j$ is calculated by

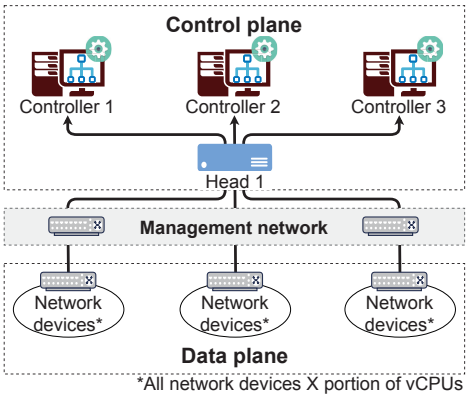
$$n(\mathbf{S}|C_j) = m \times \frac{max(n(vCPUs|C_j))}{\sum_{M_i \in \mathbf{M}} v_i}, \tag{4.11}$$

where $n(\mathbf{S}|C_j)$ denotes the number of network devices served by $C_j$, $\sum_{M_i \in \mathbf{M}} v_i$ means the total number of vCPUs in all Head nodes, and the term $max(n(vCPUs|C_j))$ represents the number of vCPUs which can be occupied by $C_j$. To calculate $max(n(vCPUs|C_j))$, we defined following equation:

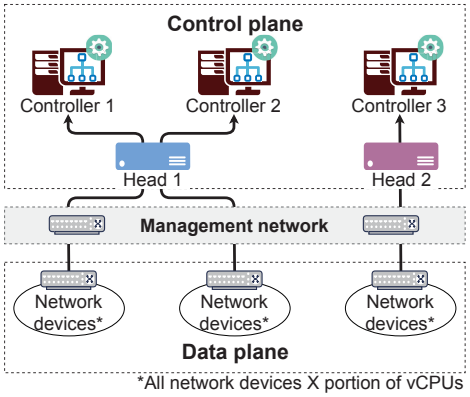
$$max(n(vCPUs|C_j)) = \sum_{i=1}^{k}(v_i \times r_{i,j}) \times \frac{1}{\mathbf{R_i} \times \mathbf{J_{1 \times n}}}, \tag{4.12}$$

where $\sum_{i=1}^{k}(v_i \times r_{i,j})$ represents the total number of vCPUs in the Head node $\mathbf{M_i}$, $\mathbf{J_{1 \times n}}$ is a $1 \times n$ size all-one matrix, $\mathbf{R_i}$ is the relationship matrix between each controller and the Head node $\mathbf{M_i}$, and $\mathbf{R_i} \times \mathbf{J_{1 \times n}}$ denotes the number of controllers running in the Head node $\mathbf{M_i}$.

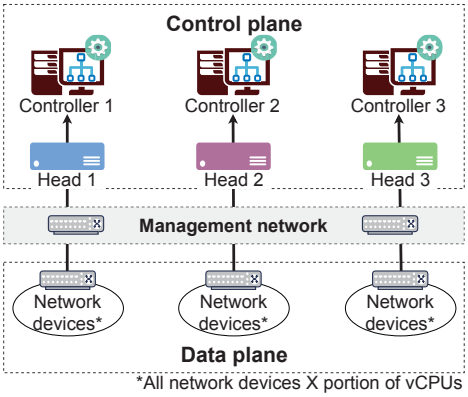
Figure 4.7 depicts cases of the deployment scheme with the various number of Head nodes. In this case, we decided that T-DCORAL takes advantage of




(a) With one Head node




(b) With two Head nodes




(c) With three Head nodes

Figure 4.7: The concept of deployment scheme with the various number of Head nodes

three controllers to manage $m$ OpenFlow switches. When the CP has only one Head node, each controller is able to fairly occupy $n(vCPUs|Head1)/3$ vCPUs (Figure 4.7a). Thus, each controller services $m \times \frac{n(vCPUs|Head1)/3}{n(vCPUs|Head1)} = \frac{m}{3}$. Figure 4.7b shows the case with two Head nodes, where Head 1 runs the controller 1 and the controller 2, and Head 2 operates the controller 3, respectively. Since each controller running in Head 1 can fairly have $n(vCPUs|Head1)/2$ vCPUs, we assign $m \times \frac{n(vCPUs|Head1)/2}{n(vCPUs|Head1)+n(vCPUs|Head2)}$ network devices to each controller. On the contrary, the controller 3 operating in Head 2 occupies $n(vCPUs|Head2)$ vCPUs; the controller 3 services $m \times \frac{n(vCPUs|Head2)}{n(vCPUs|Head1)+n(vCPUs|Head2)}$ network devices. Next, there are three Head nodes in the CP shown in Figure 4.7c, where each Head node performs each controller. Thus, the controller 1 should service $m \times$

$\frac{n(vCPUs|Head1)}{n(vCPUs|Head1)+n(vCPUs|Head2)+n(vCPUs|Head3)}$, whereas the controller 2 should manage $m \times \frac{n(vCPUs|Head2)}{n(vCPUs|Head1)+n(vCPUs|Head2)+n(vCPUs|Head3)}$ network devices. Also, the controller 3 should control $m \times \frac{n(vCPUs|Head3)}{n(vCPUs|Head1)+n(vCPUs|Head2)+n(vCPUs|Head3)}$ network devices.

### 4.4.4 An Operational Example of T-DCORAL

Figure 4.8 shows an operational example of T-DCORAL for 23 timeslots. For this example, we assume that there are three controllers, *Ctrl #1*, *Ctrl #2*, and *Ctrl #3*, in the same Head node which has 18 vCPUs. Each controller should occupy at least two vCPUs ($\delta = 2$) In this figure, the purple box A denotes the allocation of vCPU to a controller, whereas the orange box R represents the release of vCPU from a controller. From timeslot 1 to 11 (the first phase), the CP experiences the low control traffic load; each controller suffers from the CPU load lower than $\eta_l$ at some timeslots. However, the CP undergoes the heavy control traffic load in the reminder timeslots, which causes the CPU load of each controller upward the $\eta_u$ (the second phase). Therefore, T-DCORAL tries to allocate more vCPUs to each controller.

To begin with, we explain how T-DCORAL works in the first phase. At the timeslot 1, three controllers experience the CPU load in between two threshold

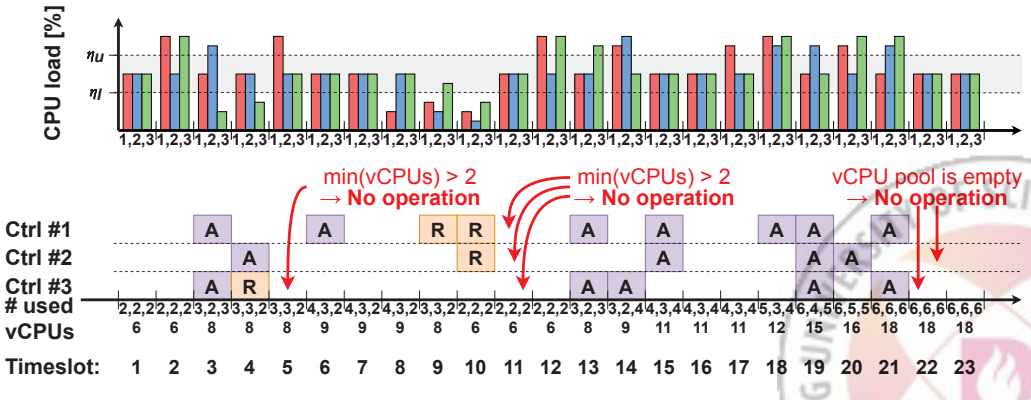



Figure 4.8: An operational example of T-DCORAL

– no operation. Since *Ctrl #1* and *Ctrl #3* suffer from the CPU load more than $\eta_u$, T-DCORAL assigns a vCPU to each controller at the timeslot 3. At the timeslot 3, the CPU load of *Ctrl #2* exceeds $\eta_u$, while the CPU load of *Ctrl #3* is beneath $\eta_l$. T-DCORAL allocates a vCPU to *Ctrl #2* and reclaims a vCPU from *Ctrl #3* at the next timeslot. Next, *Ctrl #3* experiences the CPU load lower than $\eta_l$. Even though T-DCORAL should redeem a vCPU from *Ctrl #3*, there is no operation. The reason is that *Ctrl #3* only have two vCPUs which is the minimum number of vCPUs $\delta$. Likewise, this situation happens for all controllers at the timeslot 11 – no operation.

Next, we explain T-DCORAL operation in the second phase. Obviously, each controller takes one more vCPU when the CPU load of each controller exceeds $\eta_u$. The CPU load *Ctrl #1* is over than $\eta_u$ at the timeslot 12, 14, 17, 18, and 20; T-DCORAL assigns a vCPU to *Ctrl #1* at the timeslot 13, 15, 18, and 19. Likewise, *Ctrl #2* experiences the CPU load more than $\eta_u$ at the timeslot 14, 18, and 19, while *Ctrl # 3* suffers from the CPU load over $\eta_u$ at the timeslot 12, 13, 18, and 20. As a result, *Ctrl #2* and *Ctrl #3* acquire a vCPU at the next timeslots. Since each controller obtains vCPUs, the vCPU pool is empty at the timeslot 22, finally. At this timeslot, the CPU load of *Ctrl #1* and *Ctrl #3* exceeds $\eta_u$. Thus, T-DCORAL runs no operation in this situation, although each controller needs more vCPUs.

## 4.5 Implementation

In this chapter, we propose an ECP framework for SD-DCNs. The goal of this framework is to support any ECP including previous ECPs, H-ECP, and T-DCORAL for SD-DCNs. To begin with, we introduce an overview and the requirement list of the orchestrator. Then, we describe the high-level design ot the orchestrator. Last, we explain OFMon the OpenFlow monitoring application in ONOS [6].

### 4.5.1 Overview

According to the system architecture, The ECP framework for SD-DCNs consists of the DP, the CP, and the orchestrator. Figure 4.9 depicts an overview of the ECP framework for SD-DCNs. The DP contains many Compute nodes and network devices. We decide that the type of all network devices is OpenFlow switch, especially OpenVSwith [60] in this framework since OpenFlow is *de facto* standard for SD-DCNs and OpenVSwitch is the well-known open-source OpenFlow switch. Of course, other network devices like P4 switches [61] can be applied in SD-DCNs. The CP includes some Head nodes running controllers, GUI, VNF managers (e.g., Synchronizers in CORD [40], OpenStack [62], and Kubernetes [63]), and other control applications. For controllers, we use ONOS [6] controllers which is one of the widely used distributed controllers. The orchestrator manages controllers and Head nodes in the CP. To communicate with each other, there is the management network including a management switch.

The orchestrator consists of several functions to manage controllers in the CP. First, the orchestrator has a monitoring module to monitor the computing and networking resources of each controller and each Head node. In addition, the monitoring module observes the CP topology. Second, the orchestrator has a database to store the above monitoring results. In order to manage controllers

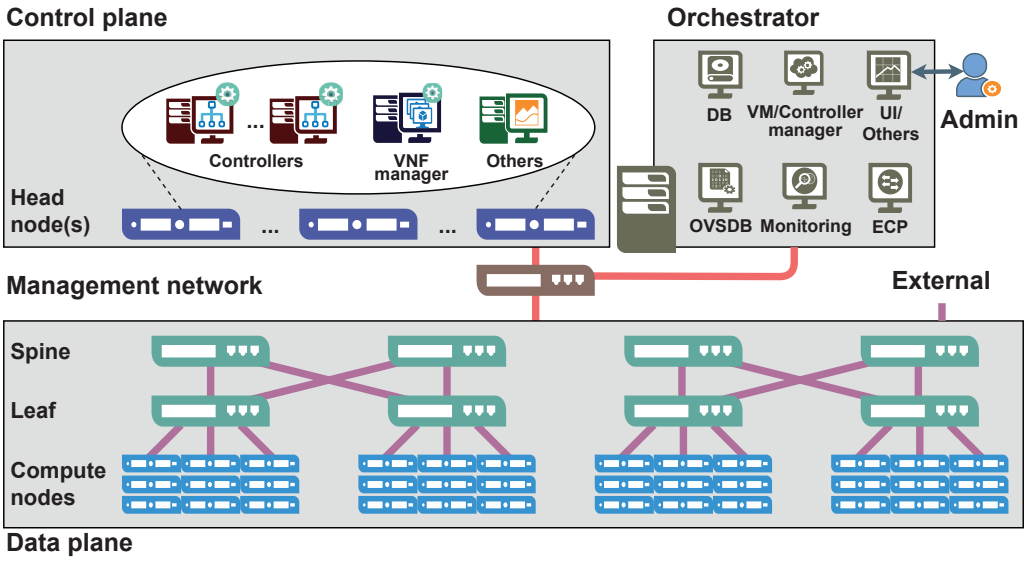



Figure 4.9: An overview of the ECP framework

and VMs which run the controllers, the orchestrator has a controller manager and a VM manager. For example, the VM manager switches on and off VMs, whereas the controller manager changes the CP topology. Next, the orchestrator contains the OpenVSwitch Database (OVSDB) which can notify switched off controllers to each OpenFlow switch in the DP. If the orchestrator refrains to notify the switched off controllers to OpenVSwitches, unnecessary control messages arise. When an OpenVSwitches misunderstand the switched off controller as the running controller, OpenVSwitches still send control messages to the controller. Since the controller transmits no response message to the OpenVSwitch, OpenVSwitch tries to reassociate with the controller again. To avoid this situation, the orchestrator let all OpenVSwitches know controllers status through OVSDB. Except for the above elements, the orchestrator has a UI to interact with network operators and other applications like ECPs. With those modules, the orchestrator also has the module ECP to run ECPs.

### 4.5.2 Requirement List

To implement software for the above orchestrator, we initially make a requirement list (Table 4.1). The orchestrator mainly requires the main controller to control all modules and threads. ECP modules are necessary to operate previous ECPs [33–39], H-ECP, and T-DCORAL. Next, the orchestrator requires monitoring modules to monitor the following metrics: (i) the number/bytes of control messages; (ii) the CPU load of each controller and each VM; (iii) the network load of each VM; (iv) the CP topology; (v) the type of each controller {active, inactive, standby}. The orchestrator needs to contain the VM manager and the controller manager. The VM manager has the power management function to switch on and off each VM and the vCPU allocator to assign and

Table 4.1: Requirement list of the orchestrator

| **Modules** | **Requirements** |
|---|---|
| **Main controller** | • to control below modules and threads |
| **ECPs** | • to operate previous ECPs, H-ECP, and T-DCORAL |
| **Monitoring** | • to monitor control messages [# control messages, bytes]<br>• to monitor the CPU load of each VM and controller [%]<br>• to monitor the network load of each VM [bps]<br>• to monitor the CP topology<br>• to monitor the type of each controller |
| **VM manager** | • power management to switch on/off each VM<br>• vCPU allocator to assign/release vCPUs |
| **Controller manager** | • mastership module to change the CP topology |
| **OVSDB** | • to register/unregister controllers in OpenVSwitch |
| **Database** | • to store monitoring results<br>• to get monitoring results for ECPs |
| **Communication** | • to communicate with VMs/controllers through SSH<br>• to communicate with controllers through REST API |
| **Parser** | • to parse received messages through SSH<br>• to parse received messages through REST API |
| **User interface** | • to interact between the orchestrator and network admins |

release vCPUs to and from each VM. On the contrary, the controller manager transmutes the CP topology. The orchestrator also requires OVSDB to register and unregister controllers to each OpenVSwitch. To store monitoring results, the orchestrator has the database. With the database, ECPs get the monitoring results to operate appropriate functions and algorithms. The orchestrator needs the communication modules to communicate with VMs and controllers through SSH or REST API. To parse the received messages through SSH or REST API, the orchestrator needs the parser. Finally, the orchestrator requires the user interface to interact between the orchestrator and network operators.

### 4.5.3 Design of the Orchestrator

Figure 4.10 shows the high-level design of the ECP framework to achieve the requirement list. In this design, there are three types of design: the orchestrator; Head nodes; and OpenVSwitches. The orchestrator can communicate with Head node, ONOS controllers running in Head nodes, and OpenVSwitches through a management network.

The orchestrator has lots of modules for ECPs. To interact with users or network administrators, the orchestrator has a user interface module (UI). They can initiate ECPs to manage the CP or configures parameters for ECPs through UI. The Main controller in the orchestrator commands to run one of ECPs and queries monitoring results from a database (DB). Literally, ECP modules contain each ECP module to operate previous ECPs, H-ECP, and T-DCORAL. Since H-ECP is designed based on multi-thread environment, the orchestrator has Thread manager which handles all threads in the orchestrator. With the Thread manager, all ECPs run their function and algorithm as a thread including the rebalancing operation, the resizing operation, and the monitoring operation. The Thread manager controls the operation sequence among multiple threads. Also, Each

ECP module basically run Monitoring module to control the CP through the Thread manager. Monitoring module gets monitoring results from ONOS controllers and Head nodes through Transceiver module which supports SSH and REST API. When receiving the monitoring results, the Monitoring module saves the results in DB. In order to change the CP topology, the orchestrator has the Mastership module. This module can change the master controller of each OpenVSwitch. Mastership module requests the change of the master controller to the ONOS controller through Transceiver. The Power manager in the orchestrator deals the power of VMs running ONOS controllers. To power on and off VMs, this manager orders to VM Monitor (VMM) in Head nodes through Transceiver. Aforementioned before, all OpenVSwitches should know the information which controllers are powered off. To notify this information, the Power manager calls the OVSDB module which requests to set controllers being running through Transceiver. Last, Transceiver in the orchestrator handles to communicate with Head nodes, ONOS controllers, and OpenVSwithches. To send messages toward a node outside the orchestrator, Transceiver takes advantage of the SSH module and REST module for the Secure Shell (SSH) and Representational State Transfer (REST) API, respectively. If the orchestrator receives responses (e.g., monitoring results) from each node, SSH module or REST module receives raw response messages. To interpret and refine the raw messages, there are two modules, the REST parser and the SSH parser.

Head nodes run ONOS controllers as VMs on top of the VMM. The VMM deals with the power of each VM and monitors the CPU load and the network load of each VM. To collect monitoring results in Head nodes, the CPU/Network monitoring agent is running in each Head node as a background process. This agent queries the CPU load and the network load to VMM periodically; this agent accumulates the CPU load and the network load over time. The orchestrator

can access the monitoring results collected by CPU/Network monitoring agent through the SSH module. Each VM has a vCPU module to monitor the number of vCPUs being used by each VM. Furthermore, this module can change the number of vCPU being used in each VM. Consequently, the orchestrator is possible to get and change the number of vCPUs by using the vCPU module in each VM through SSH module. Except for ONOS controllers, Head nodes are possible to run other applications such as VNF manager (e.g., OpenStack [62], Kubernetes [63]), GUI, and so on, which we ignore them in this figure.

An ONOS controller inside Head nodes has many functions. To communicate with the orchestrator, ONOS essentially has the REST module. Mastership manager allows the orchestrator to change the master controller of each Open-VSwitch through REST module (i.e., the change of the CP topology). OFMon which is the OpenFlow monitor counts the number and bytes of OpenFlow messages. When the transmission or reception event of OpenFlow messages happens, OpenFlow Subsystem calls OFMon to notify the event. Through the REST module, the orchestrator can get the number and bytes of OpenFlow messages. We will explain the detailed design of OFMon in the next section, Section 7.4.

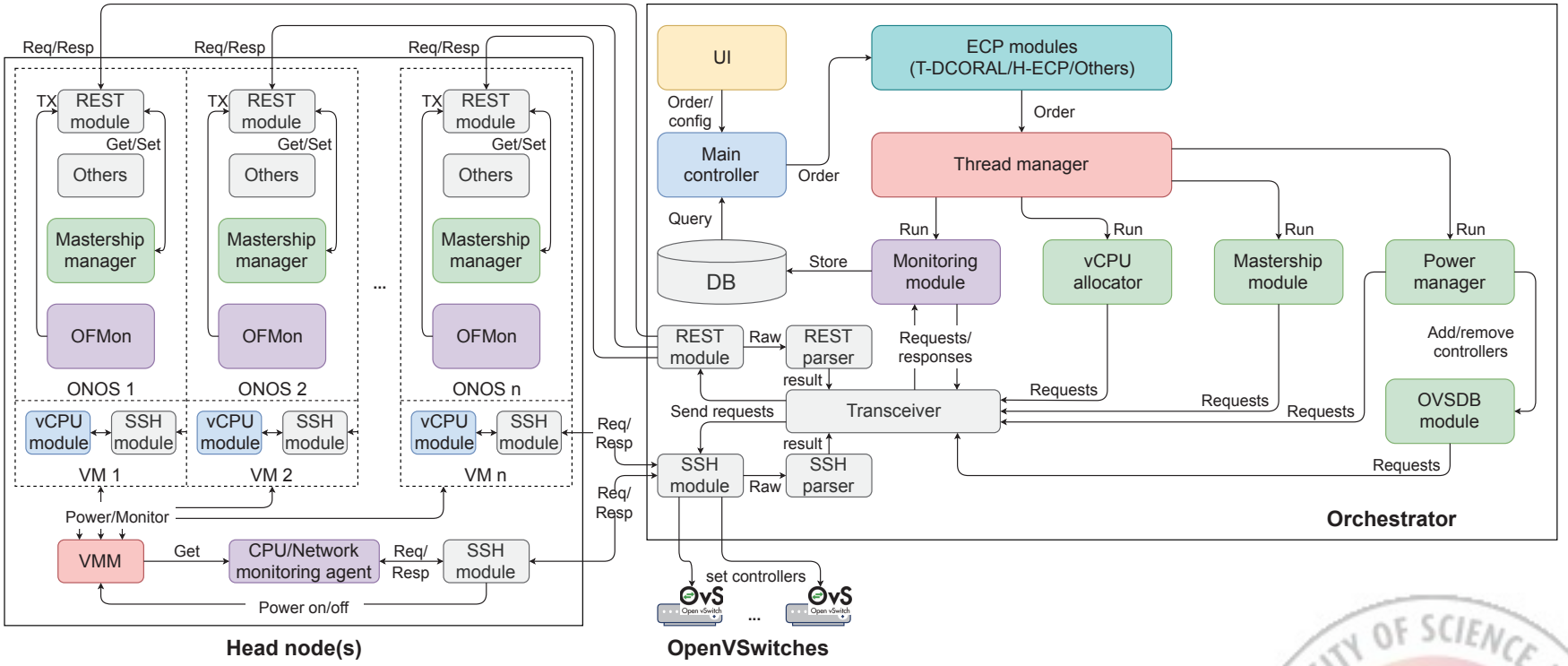



Figure 4.10: High-level design of the ECP framework

### 4.5.4 Design of OFMon

Next, we design the control plane monitor in ONOS, which is names as OpenFlow Monitor (OFMon). The OFMon the monitoring subsystem to monitor the number and bytes of OpenFlow messages in ONOS [6]. According to an architecture guide of ONOS [6], there are four layers and two APIs. Figure 4.11 shows an architecture of ONOS to manage the SDN network through control protocol. Protocols layer and Providers layer include functionalities of each control protocol such as OpenFlow, NetConf, OVSDB, and so on. It is possible to support those control protocols from each ONOS controller. Core layer contains many Managers to provide core functions for management of SDN network. Also, this layer has Store to synchronize and share the information to the other ONOS controllers. Lastly, Applications layer includes many applications which expose the functions through various interfaces such as command line interface (CLI), graphical user interface (GUI), and REST API to manage the SDN network. In order to connect between Provider layer and Core layer, each ONOS controller defines Southbound (SB) API. Moreover, there is Northbound (NB) API in each

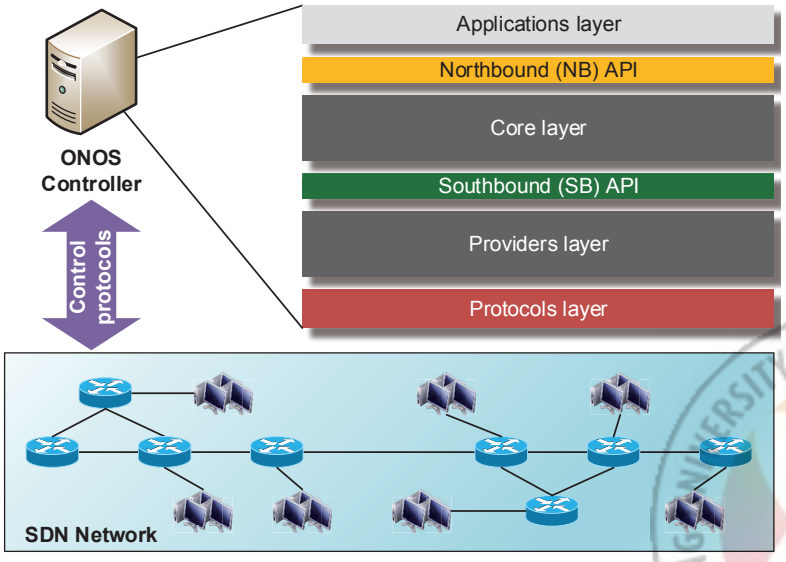



Figure 4.11: An architecture of ONOS

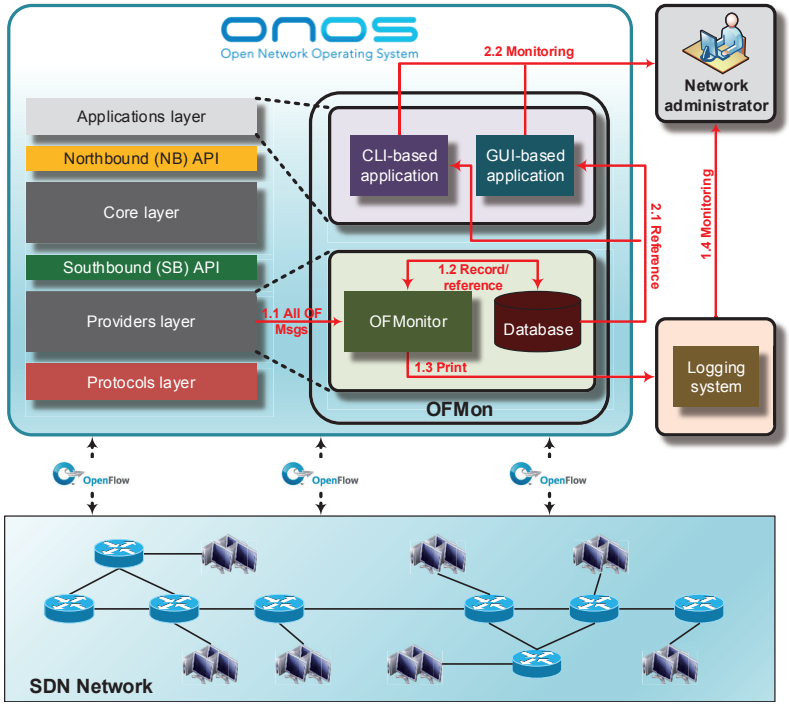



Figure 4.12: A design of OFMon in ONOS

ONOS controller to connect between Applications layer and Core layer as well.

OFMon is designed in accordance with the architecture of ONOS. Figure 4.12 illustrates a design of OFMon in a single ONOS controller. OFMon includes four components as follows:

- **Database**: this component contains the cumulative monitoring results of OpenFlow messages which are transmitted to and received from switches in the SDN network. The cumulative monitoring results classified into each switch in the SDN network. This component is located in Providers layer of ONOS.

- **OFMonitor**: this component records the monitoring results in Database, when the OpenFlow messages are transmitted and received. Moreover, this component records all sending and receiving events of OpenFlow messages into the logging system of ONOS controller in real time. Also, this compo-

Table 4.2: Types of OpenFlow messages which OFMon can monitor

| | |
|---|---|
| **Incoming messages** | Hello, Error, EchoReq, EchoRes, Experimenter, FeatureRes, GetConfigRes, PacketIn, FlowRemoved, PortStatus, MultipartRes, BarrierRes, QueryGetConfigRes, RoleRes, GetAsyncRes |
| **Outgoing messages** | Hello, EchoReq, EchoRes, Experimenter, FeatureReq, GetConfigReq, SetConfig, PacketOut, FlowMod, GroupMod, PortMod, TableMod, MultipartReq, BarrierReq, QueryGetConfigReq, RoleReq, GetAsyncReq, SetAsync, MeterMod |

nent periodically records the cumulative monitoring results from Database. This component is located in Providers layer of ONOS. This module can monitor all types of OpenFlow messages described in Table 4.2.

- **CLI-based application**: this component provides a monitoring application using CLI. When the network administrator demands to display the monitoring results, this component shows the results from Database on the terminal of ONOS controller.

- **GUI-based application**: this component provides a monitoring application using Web-based GUI. When the network administrator demands to display the monitoring results, this component shows the results stored in Database on the web page of ONOS controller.

By using OFMon, the network administrator can monitor with three types of interfaces, the ONOS logging system, CLI, and GUI. With the ONOS logging system, the network administrator can monitor all events of OpenFlow messages in real time. Also, the network administrator periodically checks the cumulative monitoring results from Database. In this situation, OFMon does not need both CLI-based application component and GUI-based application component. If the network administrator wishes to use the user-friendly interfaces, then the network administrator can exploit either CLI-based application or GUI-based application.

# V

# Performance evaluation

In this chapter, we evaluate H-ECP and T-DCORAL in the ECP framework for SD-DCNs. First, we evaluate the performance of H-ECP and T-DCORAL. Next, we evaluate the deployment scheme of network devices in T-DCORAL. Last, we discuss about the evaluation results.

## 5.1 Evaluation of H-ECP and T-DCORAL

### 5.1.1 Environment

We evaluated H-ECP and T-DCORAL with four generalized ECPs derived from ElastiCon, EDES, and EAS: (i) ECP-1 according to the CPU load (CPU1); (ii) ECP-2 according to the CPU load (CPU2) to cover ElastiCon and EDES; (iii) ECP-1 according to the network load (NET1) to cover EAS; (iv) ECP-2 according to the network load (NET2). The rebalancing algorithms in ElastiCon and EAS were used in CPU1/CPU2 and NET/NET2, respectively. We did not analyze H-ECP and T-DCORAL with DCP and PDDCP. DCP is for WAN and PDDCP considers the traffic patterns in WAN, while H-ECP and T-DCORAL aims at SD-DCNs generating the random traffic pattern [48–56].

Based on the ECP framework, we implemented the orchestrator to run all ECPs and to monitor the CPU and control traffic load with VirtualBox [66] and

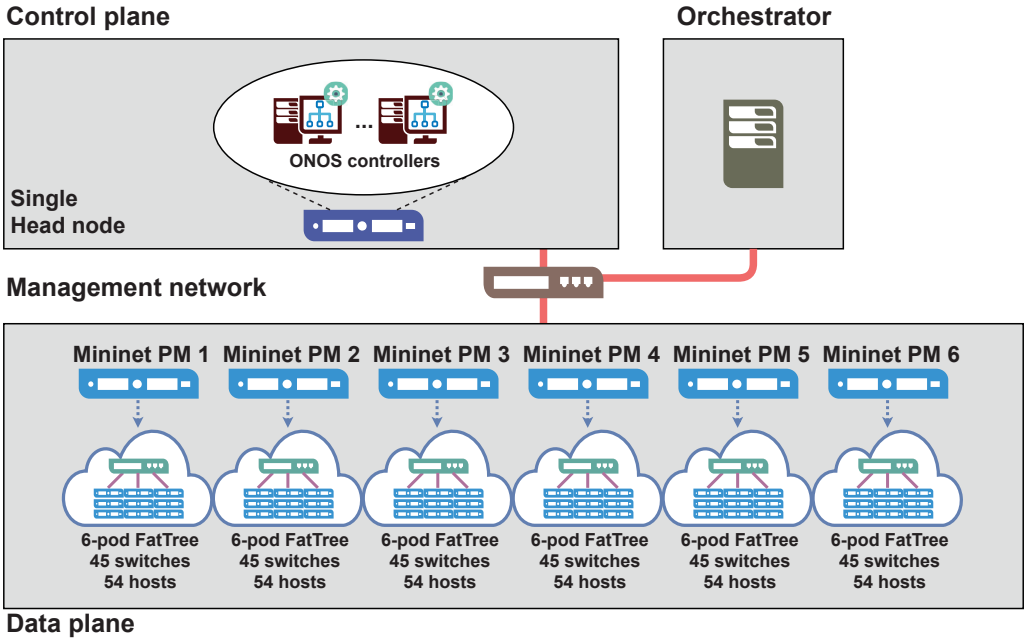



Figure 5.1: An experimental environment for the evaluation of H-ECP and T-DCORAL

OFMon, respectively. The Lines of code (LOC) was around 6,000 lines for the orchestrator and around 1,000 lines for OFMon. We conducted our experiment for 120 5-second timeslots[1]. As there was no definition to select thresholds, we experimentally determined parameters: $\eta_u = 60\%$ and $\eta_l = 40\%$ for CPU1/CPU2, H-ECP, and T-DCORAL; $\eta_u = 42$ Mbps and $\eta_l = 28$ Mbps for NET1/NET2[2].

We used a single PM to run the orchestrator and a single Head node to operate nine VirtualBox [66] VMs which performed ONOS [6] (Figure 5.1). The Head node for nine ONOS controllers ($n = 9$) contained 20 CPU cores, where two CPU cores were occupied for the OS of the Head node and 18 CPU cores can be occupied by each VM running an ONOS controller ($v_1 = 18$). The orchestrator assigned those CPUs to VMs as a vCPU and each VM should have at least two vCPUs ($\delta = 2$). Starting the experiment, we set three active controllers, which was the minimum number of active controllers ($min(Active) = 3$). Finally,

[1]Since the latency of monitoring and rebalancing is up to 5 seconds, $\tau$ is 5.

[2]In our experiment, each controller faced up to 70 Mbps. Accordingly, we defined $\eta_u = 42$ Mbps ($70 \times 60\%$) and $\eta_l = 28$ Mbps ($70 \times 30\%$).

we defined the minimum number of standby controllers as two or four for the evaluation of H-ECP ($\epsilon = \{2, 4\}$).

For the DP, we used Mininet [64, 65] to emulate six 6-pod Fat-Tree [47] topologies which included 270 OpenVSwitches and 324 hosts (Figure 5.1). We used iPerf3 [67] to generate the DC traffic with regards to [53–56], where each TCP flow had 100 KBps for 10 seconds. To observe the adjustment of the CP through ECPs, we defined the Hill scenario and the Random scenario. The Hill scenario had tho phase: (i) the phase to linearly increase the number of TCP flows from 60 to 1,800 at every 10 seconds until the middle of our experiments (60th timeslot); (ii) the phase to linearly decrease the number of TCP flows from 1,800 to 60 at every 10 seconds until the end of our experiment (120th timeslot). The Random scenario generated the random number of TCP flows [60, 1,800] at every 10 seconds. The Hill scenario we defined was the well-controlled environments, wheres the Random scenario represented more realistic environment to evaluate ECPs.

### 5.1.2 Results

Table 5.1 shows the immediacy that is the time to adjust the CP for each ECP in both the Hill and the Random scenarios. CPU1/NET1 spent around 3 seconds for all operations, because they only modified the CP topology. CPU2/NET2 expanded around 3, around 13, and around 46 seconds for the rebalancing, shrinking, and expanding operation, respectively. In particular, those ECPs spent (i) around 3 seconds to reform the CP topology, (ii) around 10 seconds to switch off the controller software, and (iii) up to 100 ms to pause the VM, when shrinking the controller pool. In contrast, CPU2/NET2 consumed (i) up to 100 ms to resume the VM, (ii) around 43 seconds to boot up the controller software, and (iii) to change the CP topology, when expanding the controller pool. We saw

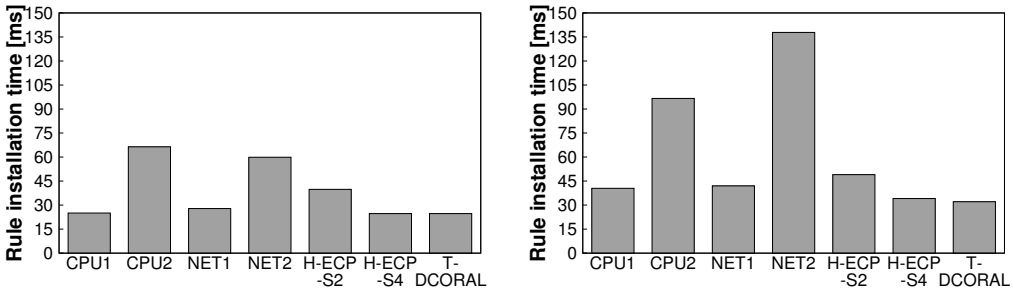
Table 5.1: Time for each operation [ms]

| ECPs | Shrinking/ Decelerating operation | Expanding/ Accelerating operation | Rebalancing operation |
|---|---|---|---|
| **CPU1** | 3179.17 | 2811.67 | 2629.63 |
| **CPU2** | 12945.50 | 46992.17 | 3032.88 |
| **NET1** | 3330.49 | 3246.73 | 3252.97 |
| **NET2** | 13228.20 | 46403.80 | 3157.04 |
| **H-ECP** | F*: 3230.12<br>B**: 10021.23 | F*: 3133.12<br>B**: 43109.04 | F*: 3340.04<br>B**: N/A*** |
| **T-DCORAL** | 32.08 | 38.55 | N/A**** |

*F means the results of the foreground thread in H-ECP.
**B means the results of the background thread in H-ECP.
***The background thread in H-ECP has no rebalancing step: no result for the rebalancing step.
****T-DCORAL has no rebalancing step: no result for the rebalancing step.

that CPU2/NET2 spent the most time to boot up the controller software due to the three substeps: (i) to switch on the controller software, (ii) to associate with other controllers, and (iii) to associate with all OpenVSwitches. However, the foreground thread in H-ECP, i.e., the CP adjustment algorithm, spent around 3 seconds, since the CP adjustment algorithm just changed the CP topology for the shrinking, expanding, and rebalancing operation. The background thread in H-ECP (the power management algorithm) consumed (i) around 10 seconds to power off standby controllers and (ii) around 43 seconds to change inactive controllers to standby controllers. Since the power management algorithm ran the step to change the CP topology, the time for the switching on and off steps spent around 3 seconds lower than the expanding and shrinking operation in CPU2/NET2. T-DCORAL required just around 30 ms to allocate vCPUs, because T-DCORAL just modified Linux system files to allocate and release vCPUs through SSH. To sum up, H-ECP had the same immediacy as CPU1/NET1, whereas T-DCORAL experienced the lowest immediacy.

Figure 5.2 illustrates the average rule installation time which is the average delay in installing flow rules from each controller into target OpenVSwitches.



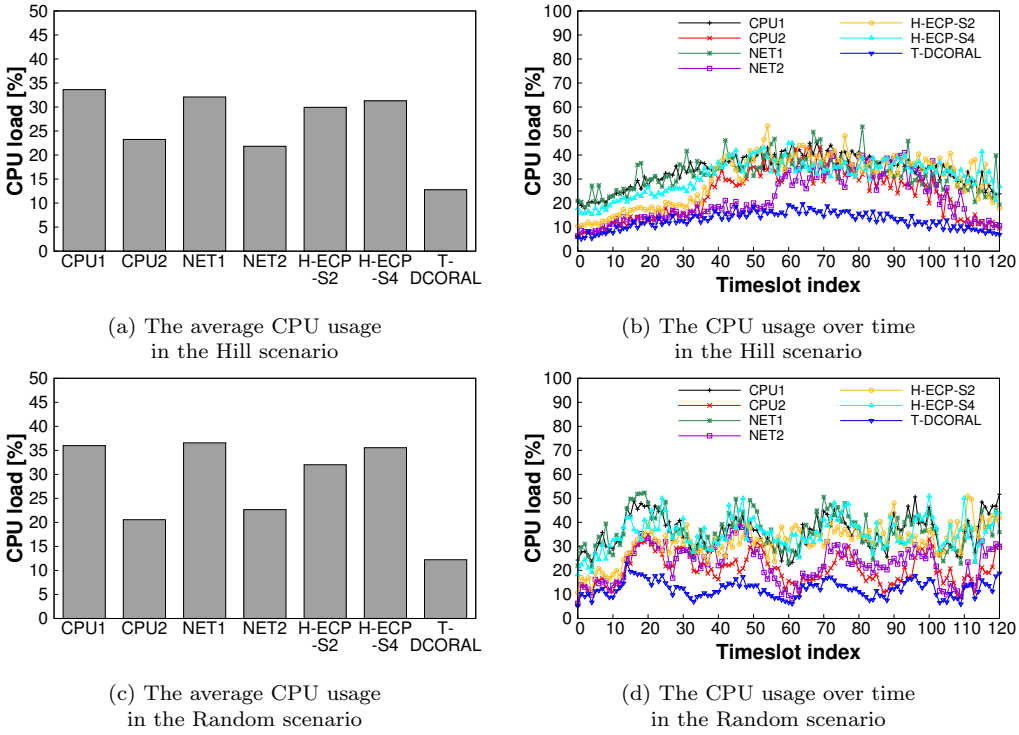

(a) In the Hill scenario

(b) In the Random scenario

Figure 5.2: The average rule installation time in both scenarios

This time can be calculated by the time difference between the transmission time of *Packet-in* message and the reception time of *Packet-out* message in each OpenVSwitch. In these experiments, CPU1/NET1 had the fastest rule installation time, since they always used the maximum number of controllers. Thus, we considered that CPU1/NET1 was the optimal performance in each experiment.

In Figure 5.2, H-ECP with $\epsilon$ of two (H-ECP-S2) experienced the average rule installation time 59% and 43% longer than CPU1 and NET1 in the Hill scenario and 21% and 16% longer than CPU1 and NET1 in the Random scenario. H-ECP-S2 had better average rule installation time than CPU2 and NET2 by 40% and 33% in the Hill scenario and by 49% and 64% in the Random scenario. H-ECP with $\epsilon$ of four (H-ECP-S4) gots the average rule installation time very similar to the CPU1 and NET1, whereas H-ECP-S4 achieved faster average rule installation time than CPU2 and NET2 by 62% and 58% in the Hill scenario and by 64% and 75% in the Random scenario. Due to the difference of $\epsilon$, H-ECP-S2 and H-ECP-S4 experienced the different average rule installation time. With H-ECP-S2 experiencing the insufficient standby controllers, the number of standby controllers might become zero frequently when rapidly increasing the control traffic load. As a result, H-ECP-S2 was impossible to expand the controller pool. On the contrary, H-ECP-S4 included sufficient standby controllers; H-ECP-S4 expanded and shrunk the controller pool on demand. Thus, H-ECP-S4




(a) The average CPU usage in the Hill scenario

(b) The CPU usage over time in the Hill scenario

(c) The average CPU usage in the Random scenario

(d) The CPU usage over time in the Random scenario

Figure 5.3: The CPU usage in both scenarios

had almost the same performance as CPU1/NET1.

Next, T-DCORAL accomplished the average rule installation time very similar to CPU1/NET1 and H-ECP-S4 (Figure 5.2). In contrast, T-DCORAL attained 38% and 52% faster average rule installation time than H-ECP-S2 in the Hill and Random scenarios, since H-ECP-S2 had insufficient standby controllers. Furthermore, T-DCORAL achieved 62% and 58% faster average rule installation time than CPU2/NET2 in the Hill scenario and 66% and 76% faster average rule installation time than CPU2/NET2 in the random scenario. Indeed, T-DCORAL also allocated sufficient vCPUs to the appropriate controller, even if there are only minimal controllers. Thus, T-DCORAL attained the fastest rule installation time among ECPs like CPU1/NET1.

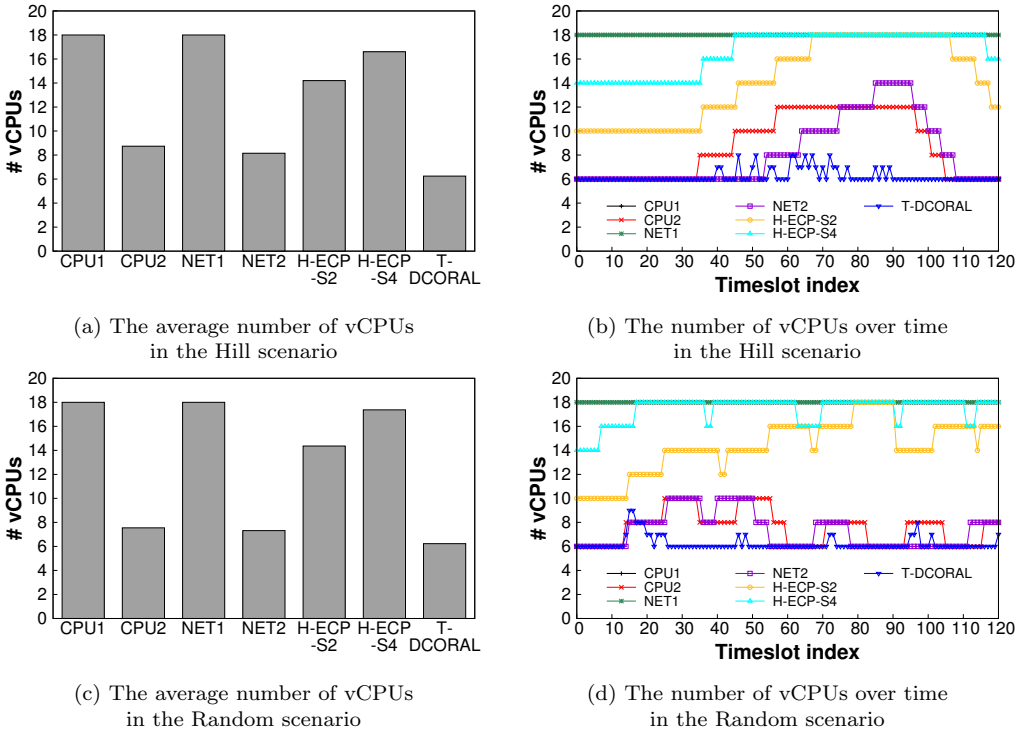
Figure 5.3 depicts the result of the CPU usage in the Hill and Random scenarios. In this figure, the average CPU usage of CPU1/NET1 exceeded the

CPU2/NET2 due to the power management. CPU1/NET1 kept all controllers in operation, whereas CPU2/NET2 switched on and off appropriate controllers on demand. Therefore, CPU2/NET2 was the target to judge whether a new ECP achieves the low CPU usage or not.

According to Figure 5.3a and Figure 5.3c, H-ECP-S2 located in between CPU1/NET1 and CPU2/NET2, while H-ECP-S4 was similar to the CPU/NET1. H-ECP-S2 attained lower average CPU usage than CPU1 and NET1 by 10% and 6% in the Hill scenario and by 11% and 12% in the Random scenario. In contrast, H-ECP-S2 experienced the average CPU usage 28% and 37% more than CPU2 and NET2 in the Hill scenario and 55% and 41% more than CPU2 and NET2 in the Random scenario, respectively. H-ECP-S4 had more average CPU usage than CPU2 and NET2 by 34% and 43% in the Hill scenario and by 72% and 56% in the Random scenario. Next, H-ECP-S4 suffered from the average CPU usage similar to CPU1/NET1. Compared with H-ECP-S4, H-ECP-S2 used less CPU capacity due to the less number of standby controllers.

Next, T-DCORAL outperformed the other ECPs even including CPU1/NET1 in terms of the average CPU usage (Figure 5.3a and Figure 5.3c) Compared to CPU1, NET1, CPU2, and NET2, T-DCORAL reduced the average CPU usage up to 61%, 45%, 60%, and 41% in the Hill scenario and up to 66%, 40%, 66%, and 46% in the Random scenario. Also, T-DCORAL achieved the 57% and 59% lower average CPU usage in the Hill scenario and 61% and 65% lower CPU usage in the Random scenario than H-ECP-S2 and H-ECP-S4. T-DCORAL had the best performance in terms of the average CPU usage since T-DCORAL only used the minimal controllers.

According to Figure 5.3b and Figure 5.3d, H-ECP-S2 suffered from the lower CPU usage than previous ECPs in the most time of the both scenario. Since H-ECP-S4 needed more standby controllers, H-ECP-S2 used less CPU usage




(a) The average number of vCPUs in the Hill scenario

(b) The number of vCPUs over time in the Hill scenario

(c) The average number of vCPUs in the Random scenario

(d) The number of vCPUs over time in the Random scenario

Figure 5.4: The number of vCPUs being occupied

than H-ECP-S4 over time. Previous ECPs and H-ECP cases always experienced the CPU usage more than the T-DCORAL. The reason was that T-DCORAL exploited minimal controllers; T-DCORAL performed fewer guest OSes, software controllers, and threads to process inter-controller and OpenFlow traffic than the other ECPs.

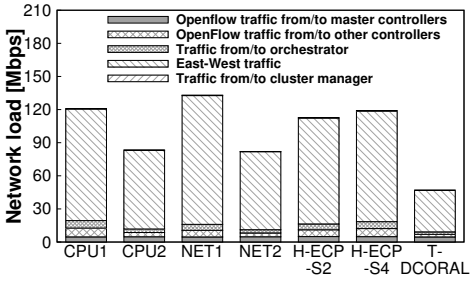
Figure 5.4 depicts the number of vCPUs being occupied in both scenarios. This figure represents that CPU1/NET1 always occupied the maximum number of vCPUs ($v_1 = 18$). On the contrary, CPU2/NET2 used less vCPUs than CPU1/NET1, because CPU2/NET2 switched on/off target controllers. As a result, CPU1/NET1 were the worst case in this result while CPU2/NET2 were the ECPs to evaluate whether new ECPs were enhanced or not regarding the number of vCPUs.

Figure 5.4a and Figure 5.4a depict the average number of vCPUs for each

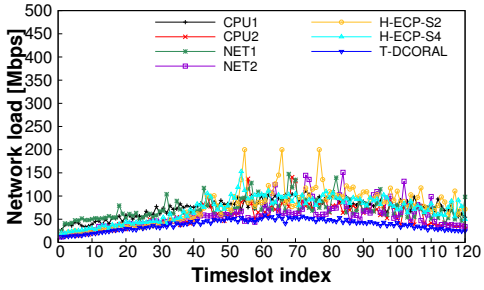
ECP. H-ECP-S2 occupied the lower average number of vCPUs than CPU1/NET1 by 21% in the Hill scenario and by 20% in the Random scenario. However, H-ECP-S2 used more vCPUs than CPU2 and NET2 by 62% and 74% in the Hill scenario and by 90% and 96% in the Random scenario, respectively. Compared with H-ECP-S2, H-ECP-S4 occupied 16% and 20% more vCPUs in the Hill and Random scenarios since H-ECP-S4 leveraged more standby controllers. Obviously, H-ECP-S4 utilized the vCPUs similar to CPU1/NET1 and more vCPUs than CPU2/NET2 by 89% and 103% in the Hill scenario and by 130% and 137% in the Random scenario, respectively. Similar to the experimental result of the CPU usage, H-ECP-S2 occupied less vCPUs than H-ECP-S4 due to the fewer standby controllers.

Next, T-DCORAL outshined the other ECPs in terms of the average number of used vCPUs. T-DCORAL occupied the less average number of vCPUs than CPU1, CPU2, NET1, and NET2 up to 65%, 28%, 65%, and 23% in the Hill scenario and up to 65%, 17%, 65%, and 14% in the Random scenario. Comparing with H-ECP-S2 and H-ECP-S4, T-DCORAL reduced the number of vCPUs being by 55% and 62% in the Hill scenario and by 56% and 64% in the Random scenario, respectively (Figure 5.4a and Figure 5.4c). The reason why T-DCORAL occupied the least average CPU load was the same appearance as the previous experimental results, i.e., T-DCORAL used minimal controllers with fewer guest OSes, software controllers, and threads to process inter-controller and OpenFlow traffic than other ECPs.

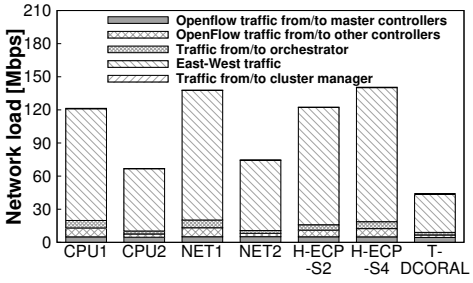
Figure 5.4b and Figure 5.4d show the number of vCPUs over time in both scenario. CPU1/NET1 always used 18 vCPUs, while CPU2/NET2 changed the number of vCPUs with the *stair-shaped* graph due to the latency for switching on and off a controller. Similar to CPU2/NET2, H-ECP-S2 and H-ECP-S4 altered the number of vCPUs with the *stair-shaped* graph. The difference between



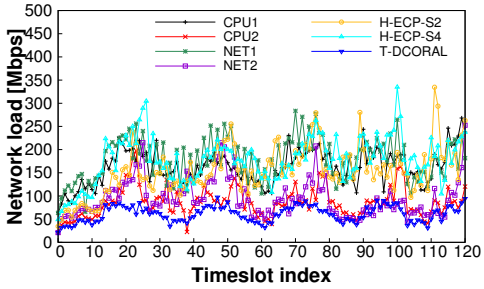

(a) The average network load in the Hill scenario




(b) The network load over time in the Hill scenario




(c) The average network load in the Random scenario




(d) The network load over time in the Random scenario

Figure 5.5: The network load

H-ECP-S2/H-ECP-S4 and CPU2/NET2 was that H-ECP-S2/H-ECP-S4 always occupied more vCPUs than CPU2/NET2 due to standby controllers. Of course, H-ECP-S4 utilized more vCPUs than H-ECP-S2 since H-ECP-S4 prepared two more standby controllers than H-ECP-S2. In contrast to previous ECPs and H-ECP-S2/H-ECP-S4, T-DCORAL adjusted the number of vCPUs with a sharp fluctuation, because of the low latency to accelerate/decelerate the CP. Thus, T-DCORAL used the CPU resources less than other ECPs; we could use more control services with T-DCORAL than other ECPs in a Head node. H-ECP utilized the slightly less CPU resource than CPU1/NET1; H-ECP could utilize a bit more control services than CPU1/NET1 not than CPU2/NET2.

Figure 5.5 shows the network load in each scenario. Similar to the result of the CPU usage, CPU2/NET2 had the network load less than CPU1/NET1, since CPU2/NET2 used less controllers in operation. In fact, even a running controller serviced no network device, the controller transceived some control messages (e.g.,

inter-controller messages). Therefore, CPU1/NET1 were the worst cases, while we compared H-ECP and T-DCORAL with CPU2/NET2 to judge whether our ECPs decreased the network load acceptably or not.

Figure 5.5a and Figure 5.5c show the average network load for each traffic type. OpenFlow traffic from/to master controller represents the average network load between each OpenVSwitch and its master controller. On the contrary, OpenFlow traffic from/to other controllers means the average network load between each OpenVSwitch and its slave controllers or controllers with Equal role[3]. Literally, Traffic from/to orchestrator is the average network load between controllers and the orchestrator and between a Head node and the orchestrator. East-West traffic means the inter-controller traffic and Traffic from/to cluster manager means the traffic between controllers and the cluster manager[4]

In these figures, the average network load of H-ECP is located in between CPU1/NET1 and CPU2/NET2. Indeed, the OpenFlow traffic from/to master controllers was similar among all ECPs, since all ECPs installed the same number of data traffic flows. With the other types of traffics especially East-West traffic, if an ECP switched on the more controllers, the other types of network traffic increased more than other ECPs. The reason was that controllers being running should synchronize with the other controllers as well as should transceive control messages (e.g., OpenFlow messages for Equal/Slave roles, the traffic to/from the orchestrator, and the traffic to/from the cluster manager). Among H-ECPs, H-

[3]With ONOS controllers, one ONOS controller is a master controller, some ONOS controllers are the controllers with Equal role, and the other controllers are slave controllers for a single OpenVSwitch. Some controllers with Equal role transceives the same OpenFlow messages as the master controller for a single OpenVSwitch. Of course, the slave controller only sends and receives the monitoring and read-only messages. As a result, the more controllers an ECP runs, the more OpenFlow messages the controllers with Equal role are transceived.

[4]The cluster manager is to maintain the ONOS cluster. This manager runs in the Head node with the statement which controllers are involved in this cluster. In the experimental environments, the statement described the information of nine ONOS controllers in detail. Even if an ECP switched off ONOS controllers, the statement was constant and the orchestrator never touched the statement. Each ONOS controller just periodically checked which controllers were located inside the cluster, which is the ONOS feature.

ECP-S2 always powered on less controllers than CPU1, NET1, and H-ECP-S4. As a result, H-ECP-S2 reduced the average network load at the maximum of 8.51%, 11.41%, and 6.71% in the Hill scenario and 4.65%, 13.94%, and 13.44% in the Random scenario, respectively. On the contrary, H-ECP-S2 encountered more average network load than CPU2 and NET2 by 36.72% and 40.68% in the Hill scenario and 78.91% and 62.41% in the Random scenario. Compared with CPU2/NET2, H-ECP-S2 switched on two more controllers, which caused more average network load. Similarly, H-ECP-S4 switched on two more controllers than H-ECP-S2, H-ECP-S4 faced higher average network load than H-ECP-S2.

Next, we analyze T-DCORAL with other ECPs in term of the average network load (Figure 5.5a and Figure 5.5c). T-DCORAL achieved the least average network load. Compared with CPU1, CPU2, NET1, and NET2, T-DCORAL reduced the average CPU load by 59.27%, 39.13%, 60.56%, and 37.37% in the Hill scenario and by 62.12%, 28.92%, 65.81%, and 35.47% in the Random scenario, respectively. Likewise, T-DCORAL encountered less average network load than H-ECP-S2 and H-ECP-S4 up to 55.48% and 58.46% in the Hill scenario and by 60.27% and 65.61% in the Random scenario, respectively. Naturally, T-DCORAL attained the least average network load, since T-DCORAL always used minimal controllers.

Figure 5.5b and Figure 5.5d depict the network load for each ECP over time. In these figures, T-DCORAL always achieved the least traffic load. Since H-ECP used controllers being running more than CPU2/NET2 but less than CPU1/NET1, H-ECP-S2 and H-ECP-S4 locate in between CPU1/NET1 and CPU2/NET2. Likewise, H-ECP-S2 powered on less standby controllers than H-ECP-S4, H-ECP-S2 met with lower network load than H-ECP-S4 in the most timeslot.

## 5.2 Evaluation of the deployment scheme of network devices in T-DCORAL

### 5.2.1 Environment

Next, we evaluated the deployment scheme of network devices in T-DCORAL with the various number of Head nodes. To evaluate our proposed scheme, we defined three cases: (i) the ideal case; (ii) the balanced case; (iii) the case with our proposed scheme. The ideal case assumed that each controller utilized the same number of vCPUs. As a result, each controller serviced the same number of OpenVSwitches. The balanced case assumed that each controller occupied the different number of vCPUs. Even if each controller leveraged the different number of vCPUs, there were the same number of OpenVSwitches for each controller (Balance). Last, each Controller managed the different number of OpenVSwitches according to the proportion of the vCPUs when each controller contained the different number of CPU cores (Proposed).

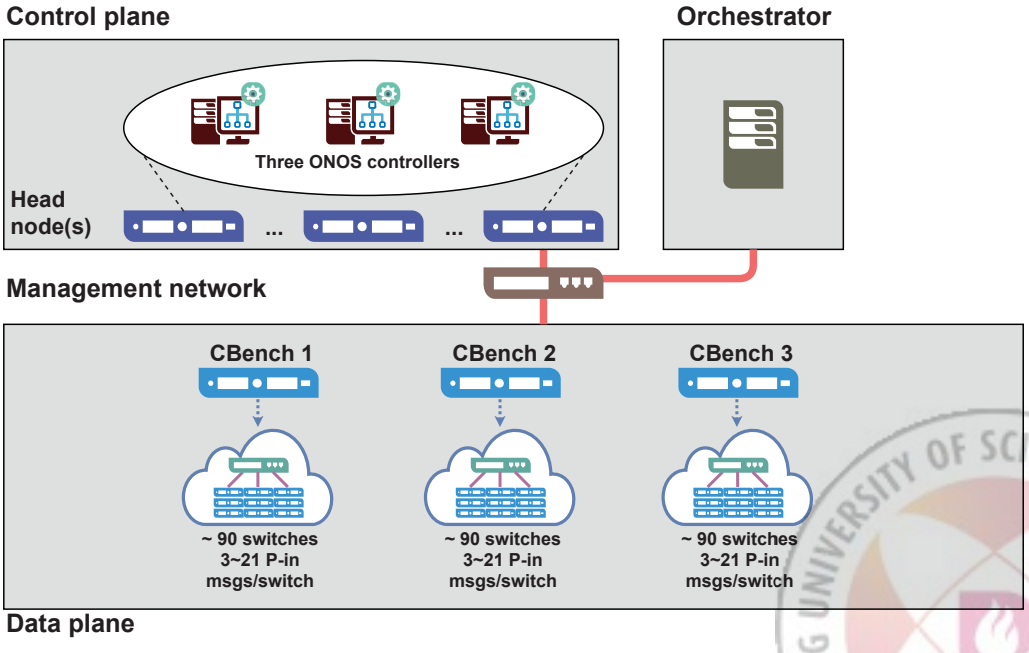



Figure 5.6: An experimental environment for the evaluation the deployment scheme of the network devices in T-DCORAL

Figure 5.6 shows an experimental environment to evaluate the deployment scheme of network devices in T-DCORAL. At first, there were the various number of Head nodes from two to three. Within those Head nodes, the CP utilized three ONOS controllers, i.e., $min(Active) = 3$. In the DP, we used three PMs to run CBench [68] which is the well-known benchmark tool for controllers. This tool generates a number of *Packet-in* messages from fake OpenVSwitches. Even though the original CBench tool sets the number of fake OpenVSwitches, it is impossible to adjust the number of *Packet-in* messages. This tool always generates the maximum number of *Packet-in* messages to check the throughput of controllers. We modified the original CBench tools to generate the number of *Packet-in* messages which we wanted. With our CBench tool, we emulated 90 fake OpenVSwitches in each PM (i.e., the total number of OpenVSwitch is 270, which is the same as the previous experimental environment) and we generated the different number of *Packet-in* message from 3 to 21 for each fake OpenVSwitch at every second. Of course, we used the orchestrator we implemented to manage controllers.

In this environment, the total number of CPUs was 12. With two Head nodes, Head node 1 and 2 had four and eight CPUs for the ideal case; each ONOS controller had four vCPUs. For the Balance and Proposed cases, each Head node exploited six CPUs; two controllers used three CPUs while one controller used six CPUs. With three Head nodes, each head node had four CPUs for the ideal case, whereas Head node 1, 2, and 3 leveraged 2, 4, and 6 CPUs in the other cases, respectively.

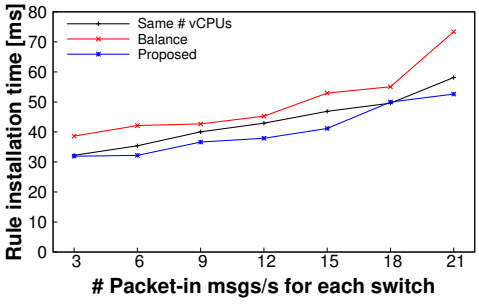
### 5.2.2 Results

Figure 5.7 depicts the average rule installation time when deploying OpenVSwitch into various Head nodes. In Figure 5.7a, the average rule installation

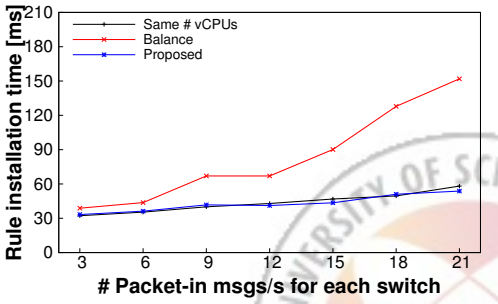
time increased when the number of # *Packet-in* messages for each OpenVSwitch increased, obviously. Comparing with the other cases, the Proposed case experienced the rule installation time similar to the ideal case. However, the Balanced case was always the worst case in terms of the average rule installation time. Comparing with the Balance case, the Proposed case always had lower average rule installation time.

With the three Head nodes (Figure 5.7b), there was the similar appearance. Both the ideal case and Proposed case had the similar rule installation time. On the contrary, Balance case experienced the slowest average rule installation time. Especially, the number of more *Packet-in* messages ONOS controllers suffered from, the more difference of rule installation time the case experienced. Also, the Balance case got worse in the environment with three Head nodes.

Comparing with two figures, the Balanced case with the three Head nodes was worse than the Balance case with the two Head nodes. The reason was that the controller with two vCPUs become the performance bottleneck in the three-Head-node scenario. Each controller used at least three and two vCPUs in the two-Head-node scenario and the three-Head-node scenario, respectively. In the two-Head-node scenario, the controller with three vCPUs spent around 55 ms to install flow rules since the controller still had quite sufficient vCPUs. However,




(a) Deploy into two Head nodes




(b) Deploy into three Head nodes

Figure 5.7: The average rule installation time when deploying OpenVSwitches into the various number of Head nodes

the controller with two vCPUs in the three-Head-node scenario consumed around 364 ms to install flow rules. Even though the difference of vCPUs was only one, the average rule installation time increased more than six times longer. On the contrary, the controller with three vCPUs in the two-Head-node case and the controller with four vCPUs in the three-Head-node case achieved the similar rule installation time (around 55 ms), although the difference of vCPUs is also one. In addition, the controller with six vCPUs in the three-Head-node case suffered from the rule installation time not quite differently (around 36 ms). Since the controller with two vCPUs delayed too longer than other controllers, the Balanced case with the three Head nodes was worst.

## 5.3 Discussion

First, we analyzed the time complexity of all ECPs. Let $n$, $m$, and $k$ be the number of controllers, OpenVSwitches, and Head nodes, respectively. The time complexity of ElastiCon, EDES, and EAS are $O(n^m)$, $O(nm)$, and $O(n^2m)$, respectively. Likewise, the time complexity of H-ECP is $O(n^2m)$ which is the same as the time complexity of EAS. Those ECPs spent the most time in the rebalancing function/algorithm, not the resizing algorithm. Since T-DCORAL has no rebalancing algorithm and contains the loop with $n$ and $k$, the time complexity of T-DCORAL is $O(kn)$. Indeed, T-DCORAL utilizes the minimal controllers, and $k$ is up to $n$ (one controller on one Head node) and less than $m$. As a result, the time complexity is lower than other ECPs.

Second, we investigated about H-ECP. Since there were standby controllers, H-ECP resized the controller pool faster than existing ECPs. Due to the thread-based design, H-ECP ran the rebalancing step more frequently than ECP-2. In fact, ECP-2 was impossible to rebalance the CP when resizing the controller pool. On the contrary, H-ECP can rebalance the CP in the foreground thread although

the background thread were switching on and off ONOS controllers. To sum up, H-ECP was possible to provide an active controller faster than ECP-2 and to avoid both the unbalanced control traffic load among controllers. Consequently, H-ECP suffered from the rule installation time less than ECP-2. Also, H-ECP used the less CPU resources, i.e., the less CPU usage and the less number of vCPUs being occupied than ECP-1. The reason was that H-ECP shut down unnecessary ONOS controllers as many as possible.

However, H-ECP has an open issue about the number of standby controllers. With the less number of standby controllers, H-ECP exploited less CPU resources and experienced lower immediacy which leads to slower rule installation time. On the other hand, H-ECP with the more number of standby controllers used more CPU resource and attained higher immediacy which causes the faster rule installation time. Thus, we must decide the number of standby depending on the situation, when we use H-ECP. If Head nodes have the sufficient CPU resource, we can use many standby controllers. However, if there are the less CPU resources in each Head node with lots of control services, we need to use the less number of standby controllers.

Next, we analyzed T-DCORAL with existing ECPs. Comparing with previous ECPs, T-DCORAL always had better performance than previous ECPs in terms of the CPU resources, the immediacy, and the rule installation time. In fact, T-DCORAL only exploited minimal ONOS controllers; the redundant and unnecessary modules (e.g., guest OS and ONOS controller software) was not performed. Also, T-DCORAL experienced lower inter-controller traffic to synchronize other ONOS controllers and OpenFlow messages than other ECPs because T-DCORAL took advantage of minimal controllers. In addition, T-DCORAL just changed the Linux system file to accelerate and decelerate each ONOS controller through SSH, which finished rapidly (the high immediacy). Thus, T-DCORAL

achieves faster rule installation time than previous ECPs.

Finally, we compared T-DCORAL with H-ECP. Obviously, T-DCORAL outperformed H-ECP in terms of all performance indicators. However, T-DCORAL should consider vCPUs in Head nodes and VMs, unlike the other ECPs including H-ECP. Initially, T-DCORAL needs to monitor vCPUs in Head nodes and VMs. To allocate vCPUs in runtime, T-DCORAL should exploit the function in VMM to assign/release vCPUs in runtime or the process to change the Linux system file. With the first function, there is the dependency of VMM to support the function like HotPlugCPU in VirtualBox [66]. With the second way, T-DCORAL needs to get the permission to change the Linux system file. Also, there is the OS dependency in each VM to change the system file. To sum up, T-DCORAL needs more information and permissions for leveraging vCPUs than the other ECPs.

# VI

# Conclusion and Future Work

In this dissertation, we proposed two new ECPs, H-ECP and T-DCORAL to improve the immediacy and to reduce the computing overhead. Previous ECPs can be categorized by two types of ECPs, ECP-1 and ECP-2 according to the resizing algorithm. Since all controllers even including inactive controllers were running, ECP-1 wasted the CPU resources (the high computing overhead). On the contrary, ECP-2 switched off inactive controllers, which delayed to resize the controller pool. ECP-2 activated inactive controllers tardily; the rule installation time of ECP-2 lagged (the low immediacy).

To mitigate the computing overhead and immediacy issues, we proposed H-ECP first. H-ECP used three types of controllers: (i) active controller to service at least one network device; (ii) inactive controller to service no network device and powered off; (iii) standby controller to service no network device but switched on. With those types of controllers, H-ECP always maintained the predefined number of standby controllers. H-ECP then resized the controller pool by using active controllers and standby controllers, not inactive controllers, according to the average CPU load. To retain the predefined number of standby controllers, H-ECP used the background thread. Due to the resizing function in H-ECP, the number of standby controllers was changed. In the background thread, H-

ECP powered on inactive controllers when standby controllers are insufficient. When H-ECP overused standby controllers, the background thread shut down standby controllers. Since H-ECP switched off all inactive controllers, H-ECP outperformed ECP-1 in terms of the computing overhead. Compared with ECP-2, H-ECP achieved higher immediacy because H-ECP resized the controller pool without the step to switch on and off controllers.

T-DCORAL was the other ECP to resolve the computing overhead and immediacy issues. In order to mitigate the computing overhead issue, T-DCORAL exploited the minimum number of controllers. In addition, T-DCORAL had no operation to switch on and off minimal controllers for the high immediacy. With minimal controllers, T-DCORAL accelerated/decelerated the CP by allocating each controller to process all control messages timely, when the control traffic was rapidly fluctuated.

For our future work, we will optimize H-ECP and T-DCORAL by changing system parameters in each ECP. In fact, we decided most system parameters, experimentally; those were not optimal values. Therefore, we will optimally decide the number of standby controllers for H-ECP, first. Second, we will analyze all parameters used in T-DCORAL such as threshold values and the number of controllers. Next, we will propose a new ECP based on T-DCORAL. T-DCORAL allocated no vCPU when the vCPU pools were empty. Also, T-DCORAL assigned/released only one vCPU even if the CP experienced a surge/plunge in the control traffic. Moreover, we studied no rebalancing algorithm for T-DCORAL. Thus, we will propose a new ECP to solve those drawbacks based on T-DCORAL. Finally, we will improve those ECPs for not only SD-DCNs but also other large-scale networks like WANs.

# 요 약 문

## 소프트웨어 정의 네트워크를 위한 효율적인 제어 평면 관리

탄성 제어 평면(ECP; Elastic Control Plane)은 대규모 소프트웨어 정의 네트워크 내의 제어 평면 부하가 급증하거나 급감하는 환경에서 효율적인 기법이다. 기존의 분산 제어 평면은 고정된 수의 컨트롤러로 모든 네트워크 장비들 및 사용자 트래픽 플로우들을 제어 및 관리한다. 하지만 분산 제어 평면은 제어 평면의 부하가 급격히 변화하면 컨트롤러의 수가 부족하거나 과도히 사용될 수 있다. 컨트롤러가 부족한 제어 평면은 네트워크 장비들의 플로우 규칙 설정이 지연되며, 과도한 경우에는 중앙 프로세서(CPU) 자원이 낭비되는 문제가 있다. 이러한 문제를 해결하기 위해, 탄성 제어 평면은 제어 평면의 부하 정도에 따라 활성 컨트롤러의 수를 조절하는 기법이다. 이를 통해, 제어 평면이 사용하는 컨트롤러의 수가 부족하거나 과도하지 않도록 조정한다.

하지만 기존의 탄성 제어 평면은 즉시성 문제 혹은 컴퓨팅 오버헤드 문제를 안고 있다. 이를 분석하기 위해, 우리는 먼저 제어 평면은 크게 두 개의 형태로 구분하였다. 첫 번째 형태는 활성 컨트롤러와 비활성 컨트롤러 모두 전원을 켜 두는 형태인 반면, 두 번째 형태는 비활성 컨트롤러의 전원을 꺼 두는 형태이다. 첫 번째 형태는 모든 전원을 켜 두기 때문에 컴퓨팅 오버헤드가 높은 문제가 있다. 한편 두 번째 형태는 컨트롤러를 활성화 할 때, 컨트롤러의 전원을 켜는 시간 때문에 즉시성이 낮은 문제가 있다.

본 학위 논문에서는 위의 두 문제를 완화하기 위해, 혼합 탄성 제어 평면(H-ECP; Hybrid ECP)과 동적 컨트롤러 자원 할당(T-DCORAL; Threshold-based Dynamic Controller Resource Allocation) 기술을 제안한다. 먼저 H-ECP는 비활성 컨트롤러의 전원은 종료하되, 일정 수의 비활성 컨트롤러를 대기 컨트롤러라 하여 전원을 켜 둔채로 유지한다. 만일 활성 컨트롤러가 부족하면 H-ECP는 대기 컨트롤러를 활성화하고, 반대의 경우에는 활성 컨트롤러를 대기 컨트롤러로 변환한다. 대기 컨트롤러가 과도히 사용되거나 부족한 상황을 막기 위해, H-ECP는 병렬적으로 대기 컨트롤러의 수를 일정하게 유지하도록 백그라운드 스레드를 동작시킨다. 결과적으로 H-ECP는 불필요한 비활성 컨트롤러의 전원을 끔으로 컴퓨팅 오버헤드를 낮추며, 전원이 켜진 대기 컨트롤러를 활성화 함으로 즉시성을 높인다. 한편, T-DCORAL은 제어 평면의 부하가 급격히 변화하는 상황에 대응하고자, 컨트롤러의 수를 증감하는것이 아닌 각 컨트롤러가 사용하는 가상 CPU(vCPU)의 수를 동적으로 조절하는 기법이다. 이 때, T-DCORAL은 컴퓨팅 오버헤드를 낮추기 위해 최소한의 컨트롤러만을 사용하도록 하며, 즉시성 향상을 위해 모든 컨트롤러의 전원을 켜 둔다.

본 학위 논문에서는 H-ECP와 T-DCORAL의 성능을 현재 널리 사용하는 컨트롤러인 ONOS와 데이터 평면을 에뮬레이션 하는 Mininet을 사용하여 진행하였다. 제안한 기법들은 기존에 제안된 탄성 제어 평면들에 비해 더 낮은 CPU 사용률, vCPU 점유율, 그리고 네트워크 사용률을 보였다. 또한 H-ECP와 T-DCORAL은 기존 기법 대비 높은 즉시성과 빠른 규칙 설정 시간을 보였다.

# References


[1] A. Lara, A. Kolasani, and B. Ramamurthy, "Network Innovation using OpenFlow: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 493–512, First quarter 2014.

[2] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, S. Shenker, "NOX: Towards an Operating System for Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, Jul. 2008.

[3] Floodlight. [Online]. Available: http://www.projectfloodlight.org.

[4] Ryu. [Online]. Available: http://osrg.github.io/ryu.

[5] OpenDayLight. [Online]. Available: http://opendaylight.org.

[6] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an Open, Distributed SDN OS," in Proc. *ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, Aug. 2014.

[7] Open Networking Foundation (ONF), "OpenFlow Switch Specification," *Technical Specification*, 2015.

[8] D. Farinacci,V. Fuller, D. Meyer, and D. Lewis, "The Locator/ID Separation Protocol (LISP)," *IETF RFC 6830*, Jan. 2013.

[9] P. Calhoun, M. Montemurro, and D. Stanley, "Control and Provisioning of Wireless Access Points (CAPWAP) Protocol Specification," *IETF RFC 5415*, Mar. 2019.

[10] N. McKeown, T. Anderson, G. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp.69–74, Apr. 2008.

[11] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Holzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally-Deployed Software Defined WAN," in Proc. *ACM SIGCOMM Conference*, Aug. 2013.

[12] K. Phemius, M. Bouet, and J. Leguay, "DISCO: Distributed Multi-Domain SDN Controllers," in Proc. *IEEE Network Operations and Management Symposiym (NOMS)*, May. 2014.

[13] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker, "Applying NOX to the Datacenter," in Proc. *ACM Workshop on Hot Topics in Networks (HotNets)*, Oct. 2009.

[14] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically Centralized?: State Distribution Trade-offs in Software Defined Networks," in Proc. *ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, Aug. 2012.

[15] B. Heller, R. Sherwood, and N. McKeown, "The Controller Placement Problem," in Proc. *ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, Aug. 2012.

[16] A. Singh and S. Srivastava, "A Survey and Classification of Controller Placement Problem in SDN," *International Journal of Network Management (IJNM)* vol. 28, no. 3, pp. 1–25, May. 2018.

[17] S. Lange, S. Gebert, T. Zinner, P. Tran-Gia, D. Hock, M. Jarschel, and M. Hoffmann, "Heuristic Approaches to the Controller Placement Problem in Large Scale SDN Networks," *IEEE Transactions on Network and Service Management (TNSM)*, vol. 12, no. 1, Mar. 2015.

[18] J. Li, J.-H. Yoo, and J. W.-K. Hong, "Dynamic Control Plane Management for Software-Defined Networks," *International Journal of Network Management (IJNM)*, vol. 26, no. 2, pp. 111-130, Mar. 2016.

[19] W. Kim, J. Li, J. W.-K. Hong, and Y.-J. Suh, "HeS-CoP: Heuristic Switch-Controller Placement Scheme for Disributed SDN Controllers in Data Center Networks," *International Journal of Network Management (IJNM)*, vol. 28, no. 3, pp.1–20, May. 2018.

[20] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, "A Survey on Software-Defined Networking," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 27–51, First quarter 2015.

[21] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella, "Towards Software-Defined Middlebox Networking," in Proc. *ACM Workshop on Hot Topics in Networks (HotNets)*, Oct. 2012.

[22] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks," in

Proc. *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Apr. 2010.

[23] M. Ghobadi, S. Yeganeh, and Y. Ganjali, "Rethinking End-to-End Congestion Control in Software-Defined Networks," in Proc. *ACM Workshop on Hot Topics in Networks (HotNets)*, Oct. 2012.

[24] N. Handigol, M. Flajsli, S. Seetharaman, R. Johari, and N. McKeown, "Aster*x: Load-Balancing as a Network Primitive," in Proc. *GENI Engineering Conference (Plenary)*, Nov. 2010.

[25] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "ElasticTree: Saving Energy in Data Center Networks," in Proc. *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Apr. 2010.

[26] K. Jeong, J. Kim, and Y. Kim, "QoS-aware Network operating System for Software Defined Networking with Generalized OpenFlows," in Proc. *IEEE Network Operations and Management Symposium (NOMS)*, Apr. 2012

[27] F. Bannour, S. Souihi, and A. Mellouk, "Distributed SDN Control: Survey, Taxonomy, and Challenges," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, First quarter 2018.

[28] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A Distributed Control Platform for Large-scale Production Networks," in Proc. *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2010.

[29] A. Tootoonchian and Y. Ganjali, "HyperFlow: A Distributed Control Plane for OpenFlow," in Proc. *Internet Network Management Workshop / Workshop on Research on Enterprise Networking (INM/WREN)*, Apr. 2010.

[30] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling Flow Management for High-Performance Networks," in Proc. *ACM SIGCOMM Conference*, Aug. 2011.

[31] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable Flow-based Networking with DIFANE," in Proc. *ACM SIGCOMM Conference*, Aug. 2010.

[32] S. H. Yeganeh and Y. Ganjali, "Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications," in Proc. *ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, Aug. 2012.

[33] 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 Proc. *International Conference on Network and Service Management (CNSM)*, Oct. 2013.

[34] A. Dixit, F. Hao, S. Mukherjee, T. V. Lakshman, and R. Kompella, "Towards an Elastic Distributed SDN Controller," in Proc. *ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, Aug. 2013.

[35] A. Dixit, F. Hao, S. Mukherjee, T. V. Lakshmann, and R. Kompella, "ElastiCon: An Elastic Distributed SDN Controller," in Proc. *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Oct. 2014.

[36] Y. Chen, Q. Li, Y. Yang, Q. Li, Y. Jiang, and X. Xiao, "Toward Adaptive Elastic Distributed Software Defined Networking," in Proc. *IEEE Interna-*

*tional Performance Computing and Communications Conference (IPCCC)*, Dec. 2015.

[37] D. M. F. Diogo, O. C. Duarte, and G. Pujolle, "Profiling Software Defined Networks for Dynamic Distributed-Controller Provisioning," in Proc. *International Conference on Network of the Future (NoF)*, Nov. 2016.

[38] G. Prathyusha and A. Potluri, "An Efficient DHT-based Elastic SDN Controller," in Proc. *International Conference on Communication Systems and Networks (COMSNET)*, Jan. 2017.

[39] T. Wang, F. Liu, and H. Xu, "An Efficient Online Algorithm for Dynamic SDN Controller Assignment in Data Center Networks," *IEEE/ACM Transactions on Networking (ToN)*, vol. 25, no. 5, pp. 2788–2801, Jun. 2017.

[40] L. Peterson, A. Al-Shabibi, T. Anshutz, S. Baker, A. Bavier, S. Das, J. Hart, G. Palukar, and W. Snow, "Central Office Re-architected as a Data Center," *IEEE Communications Maganize*, vol. 54, no. 10, pp. 96–101, Oct. 2016.

[41] Cisco, "Cisco Data Center Infrastructure 2.5 Design Guide," *Cisco Validated Design I*, Cisco Systems Inc., May. 2008.

[42] D. Chappell, "Windows Azure And Windows HPC Server," *Chappell & Associates White Paper*, Mar. 2012.

[43] IBM Cloud Private, [Online] Available: https://www.ibm.com/cloud/private/.

[44] Google Cloud Platform: Using Clusters for Large-Scale Technical Computing in the Cloud, [Online] Available: https://cloud.google.com/solutions/using-clusters-for-large-scale-technical-computing/.

[45] Joyent, "Triton Data Center," [Online] Available: https://docs.joyent.com/private-cloud.

[46] S. D. Lowe, "Best Practice for Oversubscription of CPU, Memory and Storage in vSphere Virtual Environment," *Dell White Paper*, 2013.

[47] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," in Proc. *ACM SIGCOMM Conference*, Aug. 2008.

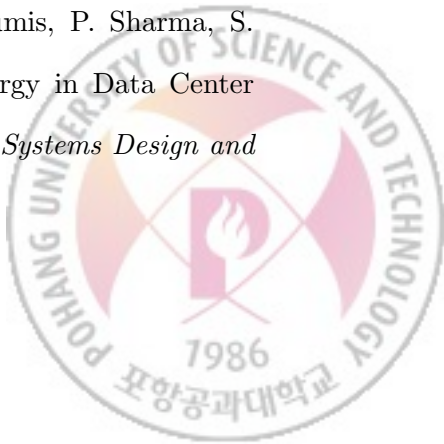
[48] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the Social Network's (Datacenter) Network," in Proc. *ACM SIGCOMM Conference*, Aug. 2015.

[49] J. Guo, F. Liu, J. C. S. Lui, and J. Jin, "Fair Network Bandwidth Allocation in IaaS Datacenters via a Cooperative Game Approach," *IEEE/ACM Transactions on Networking (ToN)*, vol. 24, no. 2, pp. 873–886, Apr. 2016.

[50] J. Guo, F. Liu, and T. Wang, "Pricing Intra-Datacenter Networks with Over-Committed Bandwidth Guarantee," in Proc. *USENIX Annual Technical Conference (ATC)*, Jul. 2017.

[51] F. Liu, J. Guo, X. Huang, and J. C. S. Lui, "eBA: Efficient Bandwidth Guarantee Under Traffic Variability in Datacenters," *IEEE/ACM Transactions on Networking (ToN)*, vol. 25, no. 1, pp. 506–519, Feb. 2017.

[52] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "ElasticTree: Saving Energy in Data Center Networks," in Proc. *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Apr. 2010.



[53] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in Proc. *ACM SIGCOMM Conference on Internet Measurement (IMC)*, Nov. 2010.

[54] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: A Scalable and Flexible Data Center Network," in Proc. *ACM SIGCOMM Conference*, Aug. 2009.

[55] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The Nature of Data Center Traffic: Measurements & Analysis," in Proc. *ACM SIGCOMM Conference on Internet Measurement*, Nov. 2009.

[56] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding Data Center Traffic Characteristics," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 1, pp. 92–99, Jan. 2010.

[57] Docker conatiner, [Online] Available: https://www.docker.com/.

[58] Stress-ng tool, [Online] Available: https://kernel.ubuntu.com/ cking/stress-ng/.

[59] Intel Math Kernel Library Benchmark, [Online] Available: https://kernel.ubuntu.com/ cking/stress-ng/.

[60] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, and P. Shelar, "The Design and Implementation of Open vSwitch," in Proc. *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* May. 2015.

[61] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schleshinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-Independent Packet Processor," *ACM SIGCOMM Computer Communications Review*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[62] OpenStack, [Online] Available: https://www.openstack.org/.

[63] Kubernetes, [Online] Available: https://kubernetes.io/.

[64] Mininet: An Instance Virtual Network on your Laptop, [Online] Available: http://mininet.org/.

[65] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible Network Experiments using Container-based emulation," in Proc. *International Conference of Emerging Networking Experiments and Technologies (CoNEXT)*, Dec. 2012.

[66] Oracle VM VirtualBox, [Online] Available: https://www.virtualbox.org/.

[67] iPerf3, "iPerf3 – The Ultimate Speed Test Tool for TCP, UDP, and SCTP," [Online] Available: https://iperf.fr.

[68] CBench - Controller benchmark tool, [Online] Available: https://github.com/mininet/oflops/tree/master/cbench.

# Acknowledgements

지금으로부터 7년 전, 제가 서울을 떠나 포항공과대학교 컴퓨터공학과 모바일 네트워킹 연구실에 처음 왔을 때가 아직도 생생히 기억 나는데 벌써 졸업을 하게 되었습니다. 제 이십대의 절반이나 이 곳에서 빛났음에도 그리 오랜 시간이 지나지 않았다고 느껴지는 건, 이 곳에서의 생활이 제게는 매우 의미있고 즐거웠을 이유겠지요. 포항공대라는 최고의 환경에서 최고의 연구를 하고, 빛나는 삶을 영위할 수 있었던 건, 제 주변에 저를 도와 주시고 응원해 주신 많은 분들 때문입니다. 이 자리를 빌어, 도움을 주시고 응원해 주신 많은 분들께 아래의 감사의 글을 올립니다.

먼저, 부족한 저를 믿고 지난 7년간 지도 해 주신, 존경하는 제 지도 교수님이신, 포항공대 서영주 교수님께 가장 먼저 감사의 말씀을 올립니다. 2011년 가을, 처음 교수님께 면접을 보고 대학원 합격 통지를 받은게 엊그제 같습니다. 지난 7년동안, 교수님께서는 제게 성공적인 인생을 살 수 있는 방법을 가르쳐주신 스승님이셨습니다. 항상 감사한 마음을 지니고, 교수님께서 가르침을 주신 것을 자산으로 삼아 살아가도록 하겠습니다. 그리고 제 연구와 진로에 많은 도움을 주신, 제게는 또 다른 지도 교수님이신, 포항공대 홍원기 교수님께도 감사의 말씀을 올립니다. 홍원기 교수님 덕분에 제가 더 큰 무대로 나아갈 수 있게 되었습니다. 진심으로 감사의 말씀을 올립니다. 또한, 연구 과제 진행 및 제 연구에 많은 코멘트를 주신 포항공대 유재형 교수님과, 논문 심사위원이신 포항공대 이승구 교수님 그리고 김광선 교수님께도 감사의 말씀을 올립니다. 마지막으로, 박사 학위를 시작하도록 독려 해 주시고 계기를 마련해 주신, 홍익대학교 컴퓨터공학과 송하윤 교수님께도 감사의 말씀을 올립니다.

다음으로, 지난 7년간 모바일 네트워킹 연구실에서 함께 동고동락한 연구실 가족

들께도 감사의 말씀을 전합니다. 7년이란 긴 시간동안 함께 한 우리 연구실 15기 동기 영덕이형, 효련이, 재국이에게 감사하는 말을 가장 먼저 전합니다. 연구실에서 힘든 일이나 즐거운 일을 항상 함께 했던 동반자였습니다. 그리고 연구실에서 저를 도와주신 김동욱 박사님, 이정윤 박사님, 정경학 박사님, 정재필 박사님, 전석성 박사님께도 감사의 말씀을 올립니다. 함께 졸업하는 상욱이형과 먼저 석사로 졸업한 승호형, 성중이, 하림이, 시영이, 상윤이형, 승은이에게도 감사하다는 말을 전하며, 앞으로 눈부신 경력을 쌓기를 기원합니다. 또한 연구실에 남아있을 연구실 후배인 총무로써 고생하는 제현이, 동덕이, 승현이, 진유, 그리고 새로 오신 형태씨에게 졸업까지 힘내라는 말과 함께 훌륭한 연구와 좋은 실적을 낼 수 있길 바랍니다. 마지막으로 저와 연구를 함께 진행했던 제 멘토인 DPNM 연구실의 이건 박사님과, 미국에서 함께 지낸 DPNM 연구실의 종환이형께도 감사의 말씀을 전합니다.

마지막으로, 제가 박사 학위를 하는데 응원과 격려를 해 주신 제 가족들께도 감사의 말씀을 올립니다. 군포에 계신 제 부모님과, 포항에 계신 부모님께 먼저 진심으로 감사의 말씀과 함께 사랑한다는 말씀을 올립니다. 포항에서 박사 학위 동안 힘이 들 때 가장 큰 힘이 되어주신 버팀목이셨습니다. 또한, 이 자리까지 올 수 있게 해 주신 사랑하는 우리 할머니께도 감사의 말씀을 올립니다. 이 자리에 올 수 있었던 것은 할머니의 사랑과 관심 덕분입니다. 박사 학위 기간동안 저를 또 응원 해 주신 우리 큰 아버지들과 작은 아버지, 그리고 다른 친지분들께도 진심으로 감사의 말씀을 올립니다. 그리고 제 동생 우희와 제 사촌 형제들, 최정석 형님, 안애랑 형수님께도 감사의 말씀을 올립니다. 마지막으로, 힘들 때나 즐거울 때 언제나 함께한 사랑하는 아내 민정이에게도 진심으로 감사의 인사를 전합니다. 감사하고, 사랑합니다.

# Curriculum Vitae

Name : Woojoong Kim

## Education

2006. 3. – 2012. 2. B.S., Department of Computer Engineering, Hongik University, Seoul, Republic of Korea.

2012. 3. – 2019. 2. Ph.D., Department of Computer Science and Engineering, Pohang University of Science and Technology (POSTECH), Pohang, Republic of Korea.

## Experience

2008. 1. – 2010. 3. Software engineer (Central Computing Center, Republic of Korea Air Force, Gyerong, Republic of Korea).

2017. 8. – 2018. 3. Research scholar (Open Networking Foundation, Menlo Park, CA, U.S).

## Affiliation

1. Mobile Networking (MoNet) Lab., Department of Computer Science and Engineering, Pohang University of Science and Technology (POSTECH).

## Publication

International Journal Papers

1. **Woojoong Kim**, James Won-Ki Hong, and Young-Joo Suh, "T-DCORAL: A Threshold-based Dynamic Controller Resource Allocation for Elastic Control Plane in Software-Defined Data Center Networks," *IEEE Communications Letters*, vol.99 (early access), pp. 1–4, Nov. 2018.
2. **Woojoong Kim**, Jian Li, James Won-Ki Hong, and Young-Joo Suh, "HeSCoP: Heuristic Switch-Controller Placement Scheme for Distributed SDN Controllers in Data Center Networks," *International Journal of Network Management (IJNM)*, vol. 28, no. 3, pp. 1-20, May. 2018.
3. **Woojoong Kim** and Ha Yoon Song, "Filtering of Erroneous Positioning Data With Iterative Application of One Class Support Vector Machine," *International Journal of Database Theory and Application (IJDTA)*, vol. 5, no. 1, pp. 67-88, Mar. 2012.

International Conference Papers

1. **Woojoong Kim**, James Won-Ki Hong, and Young-Joo Suh, "Live Distributed Controller Migration for Software-Defined Data Center Networks," in Proc *IFIP/IEEE Network Operations and Management Symposium (NOMS)*, Apr. 2018.

2. **Woojoong Kim**, Jae-Pil Jeong, and Young-Joo Suh, "Delayed Dynamic Bandwidth Channel Access Scheme for IEEE 802.11ac WLANs," in Proc. *International Conference of Information Networking (ICOIN)*, Jan. 2017.

3. **Woojoong Kim**, Seungho Ryu, James Won-Ki Hong, and Young-Joo Suh, "WLANMan: A Cloud-based Wireless LAN Management System in ONOS Controllers," in Proc. *IEEE Conference on Computer Communications (INFOCOM) Student Activities Workshop*, Apr. 2016.

4. Seungho Ryu, **Woojoong Kim**, and Young-Joo Suh, "A Block ACK Transmission Scheme for Reliable Multicast in IEEE 802.11 WLANs," in Proc. *IEEE Conference on Computer Communications (INFOCOM) Student Activities Workshop*, Apr. 2016.

5. **Woojoong Kim**, Jian Li, James Won-Ki Hong, and Young-Joo Suh, "OF-Mon: OpenFlow Monitoring System in ONOS Controllers," in Proc. *IEEE Conference on Network Softwarization (NetSoft) Conference and Workshop*, Jun. 2016.

5. **Woojoong Kim** and Young-Joo Suh, "Enhanced Adaptive Periodic Mobility Load Balancing Algorithm for LTE Femtocell Networks," in Proc. *International Conference on Network of the Future (NoF)*, Oct. 2013.

6. **Woojoong Kim** and Ha Yoon Song, "Optimization Conditions of OCSVM for Erroneous GPS Data Filtering," in Proc. *International Conference on Multimedia, Computer Graphics, and Broadcasting (MULGRAB)*, Dec. 2011.

Domestic Journal Papers (Republic of Korea)

1. **Woojoong Kim**, Jeong-Yoon Lee, and Young-Joo Suh, "Adaptive MLB Algorithm for LTE Femtocell Networks," *The Journal of Korea Information and Communications Society*, vol. 38, no. 9, pp. 764-774, Sep. 2013.

Domestic Conference Papers (Republic of Korea)

1. Seung Eun Lee, **Woojoong Kim**, and Young-Joo Suh, "A CPU Load-Based Master Controller Election Scheme for Distributed SDN Controllers," in Proc. *KICS 2017 Winter Conference*, Feb. 2017.
2. Seungho Ryu, **Woojoong Kim**, and Young-Joo Suh, ""nterference Mitigation Scheme based on Macro User Location for Cognitive Femto Base Station," in Proc. *KICS 2014 Winter Conference*, Jan. 2014.
3. **Woojoong Kim**, Jeong-Yoon Lee, and Young-Joo Suh, "A Study on the Periodic Mobility Load Balancing Algorithm in LTE Femtocell Network," in Proc. *KICS 2013 Winter Conference*, Feb. 2013.
4. **Woojoong Kim** and Ha Yoon Song, "Optimization Conditions of OCSVM for GPS Data Filtering," in Proc. *KCC 2011 Fall Conference*, Nov. 2011.
5. **Woojoong Kim** and Ha Yoon Song, "A Study on Novelty Detection of GPS Data using Human Mobility and OCSVM (One-Class SVM)," in Proc. *KIPS 2011 Spring Conference*, May. 2011.

Patents

1. **Woojoong Kim**, James Won-Ki Hong, Young-Joo Suh, and Jian Li, "Method and Apparatus for Placement of Distributed Software Defined Networking Controller (분산형 소프트웨어 정의 네트워킹 컨트롤러의 배치 방법 및 장치)," Appl. No. 10-2017-0127820 (2017.09.29), Republic of Korea Patent (Assignee: Pohang University of Science and Technology (POSTECH)).
2. Young-Joo Suh, **Woojoong Kim**, and James Won-Ki Hong, "Channel Selection Method of Communication Node in Communication Network (통신 네트워크에서 통신 노드의 채널 선택 방법)," Patent No. 10-1817862 (2018.01.05), Appl. No. 10-2017-0076433 (2016.06.20), Republic of Korea Patent (Assignee: Pohang University of Science and Technology (POSTECH)).
3. Ha Yoon Song and **Woojoong Kim**, "Location Classifying Method and Electronic Apparatus for the Same Method (위치정보 분류방법 및 이를 위한 장치)," Patent No. 10-1466668 (2014.11.24), Appl. No. 10-2011-0117949 (2011.11.13), Republic of Korea Patent (Assignee: Hongik University).

## Technical Talks & Demonstrations

1. **Woojoong Kim** and Jibum Hong, "Design and Implementation of M-CORD Monitoring System," *Open Networking Korea (ONK) 2018 Fall*, Demonstration, Oct. 2018.
2. **Woojoong Kim**, "M-CORD 5.0: From Technical Architecture to Implementation," *ONOS/CORD Working Group Seminar*, Technical Talk, Aug. 2018.
3. Open Networking Foundation (ONF), "M-CORD Demonstration in Mobile World Congress (MWC)," *Mobile World Congress (MWC) Barcelona*, Feb. 2018 (attended as a engineering staff).
4. **Woojoong Kim**, "M-CORD Connectivity: Deep Technical Dive," *CORD Build*, Technical Talk, Nov. 2017.
5. **Woojoong Kim**, "An Off-platform Orchestration Software for Switch-Controller Placement in ONOS," *ONOS/CORD Working Group Seminar*, Technical Talk, Feb. 2017.
6. **Woojoong Kim**, "A Periodic Switch-Controller Placement Software," *Open Networking Korea (ONK) 2016 Fall*, Demonstration, Nov. 2016.
7. **Woojoong Kim**, "A Design and Implementation of Control Plane Monitoring System and Mastership Manager," *Open Networking Korea (ONK) 2016 Spring*, Demonstration, Apr. 2016.

## Awards

1. Silver award at K-ICT NET Challenge Camp Season 3, Dec. 2016.


POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교