Noname manuscript No.
(will be inserted by the editor)

A Predictive Load Balancing Technique for Software Defined Networked Cloud Services

Chao-Tung Yang · Shuo-Tsung Chen · Jung-Chun Liu · Yi-Wei Su · Deepak Puthal · Rajiv Ranjan

Received: date / Accepted: date

Abstract With the advent of OpenFlow, the concept of Software-Defined Networking (SDN) becomes much popular. In the past, SDN had often been used for network virtualization; however, with the rise of OpenFlow, which speeds up network performance by separating the control layer from the data layer, SDN can be further used to manage physical network facilities. In recent years, many researchers focus on using OpenFlow to substitute regular networks and look for adding value-added services with OpenFlow, such as the load balancer or the Firewall. Currently, some OpenFlow controller providers have already provided users with load balancer packages in their controllers for virtual networks, such as the Neutron package in OpenStack; nevertheless, the existing load balancer packages work in the old fashion that causes extra delay since they poll controllers for every new coming connection. In this paper, we use the wildcard mask to implement the load balance method directly on switches or routers and add a user prediction mechanism to change the range of the wildcard mask dynamically. In this way, the load balance mechanism can be applied conforming to real service situations. In our experiment, we test the accuracies of flow prediction for different predicted algorithms and compare the delay times and balance situations of the proposed method with other load balancers. With the popularity of cloud computing, the demand for cloud infrastructure also increases. As a result, we also apply our load balance mechanism on cloud services and prove that the proposed method can be implemented to varieties of service platforms.

C-T Yang, S-T Chen, J-C Liu, Y-W Su
Department of Computer Science, Tunghai University, Taichung, Taiwan

D. Puthal
University of Technology Sydney, Australia

R. Ranjan
Newcastle University, United Kingdom
Chinese University of Geosciences, China

Keywords SDN · OpenFlow · Load Balance · Cloud Services · Prediction Mechanism

1 Introduction

Since 2008, Software-Defined Networking (SDN) has become more and more popular. SDN is a new approach to handle data forwarding and control separately. We can use this architecture to manage network easily [19]. In SDN, we can control a network which is mix physical switch and a virtual switch, and we can send the policy to individual switch [17]. Many researchers focus on services and applications derived from its concept on OpenFlow, on which load balancing is one of the most valuable value-added services. Additionally, to load balancing, there is other research such as big data with SDN [16] or SDN-based wireless [18]. There are many method and mechanism to implement load balancing. With the advent of OpenFlow, which speeds up network performance by separating the control layer from the data layer, SDN can be further used to manage physical network facilities. Recently, many researchers focus on using OpenFlow to substitute regular networks and look for adding value-added services with OpenFlow, such as the load balancer or the Firewall. Currently, some OpenFlow controller providers have already provided users with load balancer packages in their controllers for virtual networks, such as the Neutron package in OpenStack; nevertheless, the existing load balancer packages work in the old fashion that causes extra delay since they poll controllers for every new coming connection. In this paper, we use the wildcard mask to implement the load balance method directly on switches or routers and add a user prediction mechanism to change the range of the wildcard mask dynamically. In this way, the load balance mechanism can be applied conforming to real service situations. In our experiment, we test the accuracies of flow prediction for different predicted algorithms and compare the delay times and balance situations of the proposed method with other load balancers.

In this study, we aim to forecast load balancing technique in Software Defined Networked Cloud Services. Specific contributions are:

1. To propose wild-card prediction of network-load-balancing based on neural networks and k-means machine learning techniques.
2. To conduct an extensive experimental evaluation based on real-world workload trace to prove the feasibility of the proposed method.
3. To test the accuracies of flow prediction for different predicted algorithms and compare the delay times and balance situations of the proposed method with other load balancers.

The rest of the paper presented as follows: Section 2 discusses the related works. Section 3 explains the proposed system design and implementation. Section 4 describes the experimental results and conclusions and future works are drawn in Section 5.

2 Related Work

For services in unstructured networks N. Handigol et al. [31] proposed an effective load-balancing system that minimizes response time by controlling loads on the network and servers using customized flow routing. H. Uppal and D. Brandon [1] implemented a general load balancing architecture using OpenFlow and test its performance. There is still load balance issue on those researchers, and we will try to solve it in this work [33]. P. Wang, et.al. [34] implemented Back-propagation Neural Network to Predict Bus Traffic. In their paper, they found out the best network structure and parameters suiting to bus traffic prediction using BPNN. S. Sharma et al. [8] used two recovery mechanisms, such as restoration and protection, to achieve recovery requirements using OpenFlow. They found that a lot of time is consumed if the switch contacts the controller too often; given this, they applied the Group Table function provided in OpenFlow v1.1 to eliminate contacting the controller to modify the flow entries. They also found that memory of the switch is not enough to store flow entries if they increase without limit. To reduce the frequency for accessing the controller and number of flow entries, we propose to allocate servers use the wildcard mask, which is a mask of bits that indicate which part of an IP address is available for examination. A wildcard mask can be thought of as an inverted subnet mask. For example, a subnet mask of 255.255.254.0 (binary equivalent is 11111111.11111111.11111110.00000000) inverts to a wildcard mask of 0.0.1.255. The mechanism can solve access frequency and many flow entries issues. After surveying literature, we found similar research by R. Wang et al. [20]. But those papers have an issue that they assumed a nonrealistic situation in which the clients uniformly come from all IP addresses. This presumption causes some weak results. We try to find the solution to improve this situation, such as done in [9]. Security in the advancing domain such as software defined networks and cloud computing become a primary focus [24][25][26][27]. In our work, we establish a prediction mechanism to predict the traffic of future users. Using this mechanism, we can dynamically update the wildcard rule and enhance load balancing all over the whole service [32].

3 System Design and Implementation

In this section, we will begin by presenting our system architecture, following by design flow and the explanation of prediction mechanism used in our system. With the details as follow:

3.1 System Architecture

We used the wildcard mask to implement load balance, and we also built a prediction mechanism that predicts according to the network traffic. As shown in Figure 1, individual user packet must go through the load balancer in the

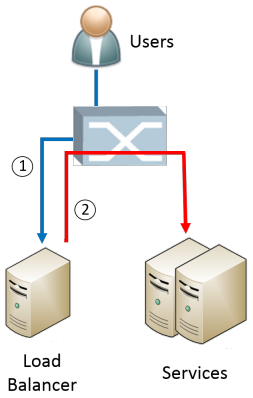

Fig. 1 General Load Balancing

general load balancing process. In this case, the load balancer needs to check each packet no matter the packet is set to be dropped or accessed, and thus extra time is spent to check the packet. Several researchers want to improve this condition with OpenFlow. Some researchers use a mechanism to check the first of packets and send the new flow entry to switch [2] [3] [7]. In other words, it checks the first packet instead of the whole set of packets and thus extra time is spent just at the first connection as shown in Figure 2. When a new user accesses the virtual IP of service, OpenFlow finds it is a new flow and asks the OpenFlow controller to determine a server to provide service for this flow. The controller will send a flow entry to the OpenFlow switch. After the switch updating the flow table, users do not need to ask the controller for the service path at the next time.

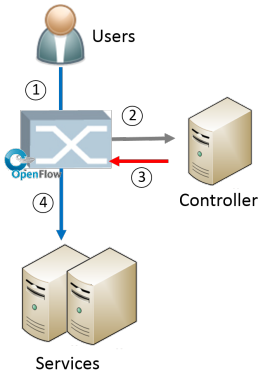

Fig. 2 OpenDayLight and FloodLight Load Balance

Using the mechanism with OpenFlow like in [3,2], extra time is still spent at the first connection. In the work of R Wang et al. [20], a new mechanism is

established to reduce communication frequency for the controller. Before users connecting to servers, a network wildcard is used to disperse source IP and thus load balancing is achieved. This mechanism works like pre-configuration. To obtain good experimental results, they assume their services clients come from all range of IP in a uniform distribution. In the real world, it is difficult to have all clients come uniformly from all IP addresses for service.

A. R. Curtis et al. [9] used a wildcard rule to implement load balancer. They improve the time spent on the switch by implementing the wildcard rule to let flow packets going to different ports by a round-robin method. They reduce all flows coming from the control plane to improve the load balance time. It points a new direction for load balancing and is a very worthy article.

All components in the infrastructure connected to the OpenFlow switch is shown in Figure 3. In [22] not much difference on latency is found using the general switch or OpenFlow switch. Thus, we did not test the effect of using the OpenFlow switch or the general switch in the experiments. We combined the prediction server and the OpenFlow controller on the same machine. The prediction server also can move to another machine or cluster if wanted.

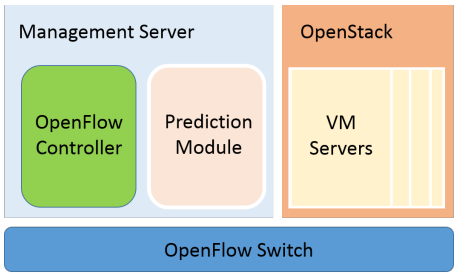

Fig. 3 System Architecture

As shown in Figure 4, we mirrored all clients' traffic and stored those traffic in the database called as the traffic database. Based on the network traffic, we trained the model to predict the future client traffic ratios and stored them in the database called as the prediction database. Afterward, those data were applied to an algorithm to create flow entries that were sent to the controller to update the switch flow table.

3.2 Design Flow

Our design consists of the client, server, and prediction module side. As shown in Figure 5, we sent the default wildcard flow entry that allocates all range of IP before the client users connecting to the server. When users past through the OpenFlow switch, we used the OpenFlow mirror method to mirror all network traffic from the client to the prediction server. We stored the network traffic in the traffic database. We would train the model to predict the future client

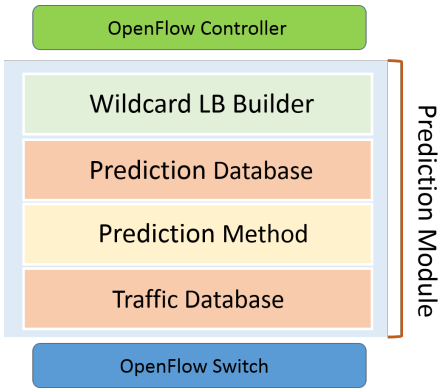

Fig. 4 Prediction Module Architecture

traffic ratios according to the traffic database with flow data stored between the last time of check and this time. We used the computed results from the prediction module to create a new wildcard flow entry command and sent it to the controller. When the controller received this command, the controller would update the flow entry on the OpenFlow switch. After the controller updating the switch, the clients connecting to the server could send packets through the new traffic path by our load balance mechanism.

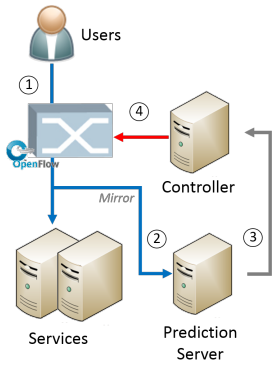

Fig. 5 Dynamic Wildcard Load Balance

3.3 Prediction Mechanism

To predict how the user would access services in the future, we used a data mining technique. Data mining is used in the analysis step of the Knowledge Discovery in Databases (KDD) process. We wanted to find out user behaviors via user traffic information. We used different prediction mechanisms to test

our module. We applied classification to predict whether a user in the future would access the switch and applied cluster analysis to cluster users of different usages. We used a neural network method and K-Mean cluster to train data and established an IP tree. We could forecast each IP appearance rate at the next checking time. In other papers [21][10][13][23], we are experimenting an approach based on a mixture of fuzzy logic and neural network, that offers positive preliminary results, but are still not entirely implemented, so their description is out of scope for this paper.

3.3.1 Neural Network

We used the back-propagation neural network (BPNN) in our mechanism. We had four input nodes, three hidden nodes, and one output node as shown in Figure 6. The first input is each IP's percentage of possession at this time calculated by the following equation:

$$Input_1 = \frac{IP_i\ Bytes}{Total\ Bytes} \tag{1}$$

We get the i_{th} user's traffic in bytes and sum of traffic in bytes and divide them at this check time set at 5 minutes of the experiment. For example, assume user A traffic of 1000 bytes and the sum of traffic of 20000 bytes at this check time, then $Input_1$ will be 0.05. There would be large variations of numbers if bytes of traffic are used; besides, the input of the neural network should be between 0 and 1, so we use the proportion of bytes for traffic. The second input is each IP's percentage of packets at this time as the following equation:

$$Input_2 = \frac{IP_i\ Packets}{Total\ Packets} \tag{2}$$

We get the i_{th} user's traffic packets and the sum of traffic packets and divide them at this check time set at 5 minutes using Equation 1. The packets and bytes are essential information for traffic, so we use these two inputs for data mining. The third input is the occurrence time for each IP as the following equation:

$$Input_3 = \frac{IP_i\ appear\ time}{Total\ check\ time} \tag{3}$$

We get the i_{th} user total access time in the past and get the entire check time of the service. For example, user B accessed the service three times in the past and the total check time of service is 10, then the $Input_3$ will be 0.3. We want to use this data to find out the user usage behavior. The last input is about how often each IP appears as the following equation:

$$\begin{aligned} &Input_4 = \frac{T_f}{T_u - T_{lu}} \\ &T_f :\ interval\ time\ of\ checking \\ &T_u :\ this\ checking\ timestamp \\ &T_{lu} :\ the\ latest\ update\ timestamp\ on\ IP_i \end{aligned} \tag{4}$$

We want to know how often each user access this service. We use the interval time of checking to get a not too small value. For example, the interval time of checking is 300 seconds and the last time user C accesses this service at 10 a.m. If the check time is at 11 a.m., then $Input_4$ will be 0.083. We chose three nodes instead of two in the hidden layer because more nodes in the hidden layer will cost more time (Table 1), and fewer nodes in hidden layer may result in errors. The output node is the probability of appearance for each IP. According to the output value, we can surmise each IP will access service in future, and we will merge all IPs' results and find the most suitable wildcard for the next time. We used the Sigmoid function:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{5}$$

on our hidden nodes and the output node.

Table 1 Different number of hidden nodes in BPNN spend time

Number of hidden nodes	Number of IP	Spend time with 5000 ecophs
3	1600	165 s
3	1680	169 s
4	1600	202 s
4	1680	208 s
6	1680	280 s
9	1680	384 s

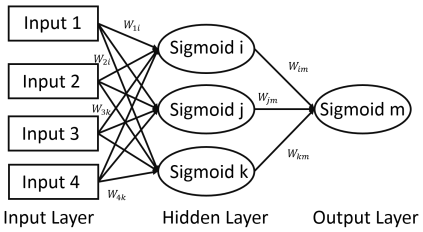

Fig. 6 BPNN 4 input, 3 hidden, 1 output

3.3.2 Traffic Classification

Many researchers studied traffic classification [11]. In this work, we used machine learning techniques to build our prediction model. We collected client's packets and traffic size to establish our model. We referenced models of Erman et al. [28] and Williams et al.[12] to implement our prediction mechanism. These two references classified protocols instead of network traffic; however, in our mechanism, we used the K-Mean cluster for traffic classification.

Clustering is the partitioning of previously unlabeled objects into disjoint groups, referred to as "clusters," such that objects within a group are similar according to chosen criteria. Formally, our clustering dataset $D = \{t_1, t_2, ..., t_n\}$ and the chosen number of clusters k, the task of clustering is to define a mapping $f : D \to \{1, 2, ..., k\}$ where each flow is assigned to only one cluster.

The goal of clustering is to group objects that are similar users. We used the Euclidean distance to measure the similarity between two flow vectors f_i and f_j:

$$sim(f_i, f_j) = \sqrt{\sum_{k=1}^{n}(x_{ik} - x_{jk})^2} \tag{6}$$

In this work, we used the K-Means [30] algorithm. The K-Means algorithm is one of the quickest and simplest algorithms for clustering and some researchers also use K-Means for clustering of Internet flows [28,29]. The algorithm begins by randomly choosing cluster centroids from within the training sample. Our training data set are the same as used in the neural network, so we will get a four-dimensional dataset and use simple Euclidean distance Equation 6 to assign the remaining instances to their cluster center. K-Means iteratively computes new centers of the clusters that are formed and then repartitions the flows based on the new centers. The complexity of the K-Means is $O(knm)$ where k is the number of clusters, n is the number of training flows, and m is the number of iterations.

The first step in the K-Means is to choose value K. We choose four Ks and test the spent time. In Table 2, we can see most time is spent when K = 25 and the time spent reduces when K is greater than 25. We think the reason for this situation is due to the number of data set. In our programming, when value K is larger than the number of data set, we will not execute K-Mean clustering. According to Table 2, we choose 10 for value K.

Table 2 Different K in K-Mean spend time

K	K-Mean cost average time(s)	Data set average number
10	0.16224944753	104
25	0.31429123688	104
50	0.292544601609	104
75	0.279896007556	104
100	0.24670831782	104

After K-Means clustering, we sorted the cluster with the percentage of bytes in this training. We define the value as the user usage rate by the result of sorting as follows:

$$value = \frac{i}{K},\ i \in \{1, 2, ..., K\} \tag{7}$$

This value will be used on wildcard load balance builder to implement load balance.

3.3.3 Wildcard Load Balance Builder

The output of each prediction mechanism is stored in the prediction database as shown in Figure 4. After the prediction process, the flow entries are established. Our flow entries are finer than those used in R. Wang, [20] and numbers of flow entries are less than those used in the OpenDayLight Load Balancer [3]. We established a wildcard tree to record the predictive output value as shown in Figure 8. Each node records sum of children value, and the leaf is the output of each IP. This tree has five layers. The first layer is the root layer, which records all predictive outputs and our algorithms do not consider these value. The second layer spans the first 8-bit of the IP address, and it has 255 nodes. The third layer spans the second 8-bit of the IP address, and it has 255×255 nodes. The fourth layer spans the third 8-bit of the IP address, and it has $255 \times 255 \times 255$ nodes. And the fifth layer is the full IP address, and it has $255 \times 255 \times 255 \times 255$ nodes. At first glance this tree is huge; however, we need to use part of the tree in our experiments. In the real tests of services, less than 100 nodes on the first layer and less than 4000 total number of IPs are found to be used. We used algorithms to establish flow entries as shown in Fig. 7.

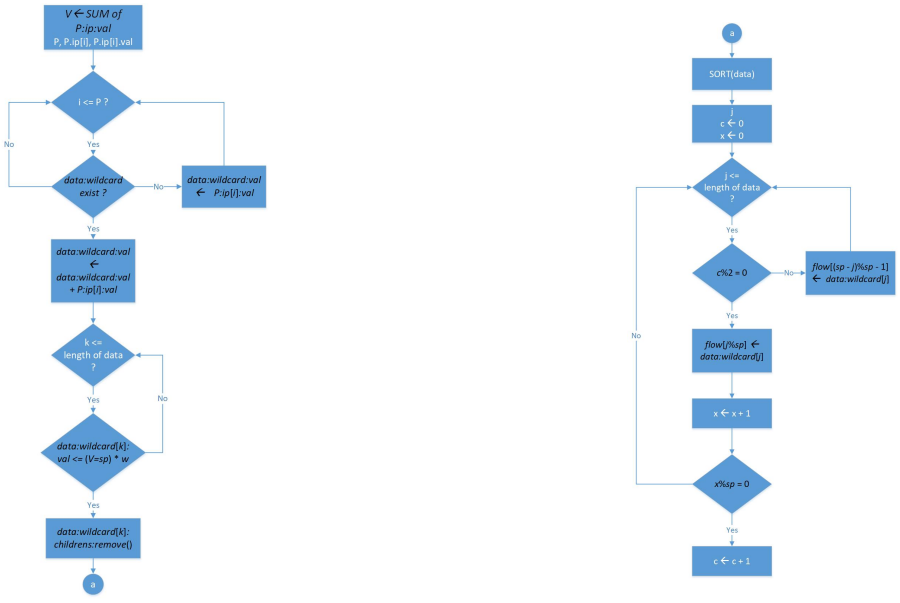

Fig. 7 Flowchart of Wildcard Flow Entries

The detail of Pseudocode is shown in Algorithm 3.1. We obtained all data for P that is an object, include IP array $P.ip[]$ and IP predictive output value $P.ip[].val$. In our algorithms, we also received the number of load balance servers sp and loaded weight of each server w. To start the algorithms, we

need to build a wildcard tree. After building the wildcard tree, we integrate the tree: we remove the children nodes when parent node's value is enough to allocate to load balance servers. After integration, we sort this tree according to predictive values, and then we assign the wildcard to load balance servers in descending order with S-type. If there is extra IP traffic, we will assign IP to the minimum load server. If there is one IP with traffic bigger than each load balance server loading, we will assign a load balance server to provide service for this IP only. After allocation, we will change those data array into Ryu controller commands, and send those commands to web REST API of the Ryu controller to control the OpenFlow switch.

Algorithm 3.1: ESTABLISH WILDCARD FLOW ENTRIES(P, sp, w)

$V \leftarrow SUM\ of\ P.ip.val$
for $i \leftarrow 0$ **to** $length\ of\ P$
do **if** $data.wildcardexsit$
then $\{data.wildcard.val \leftarrow data.wildcard.val + P.ip[i].val$
else $\{data.wildcard.val \leftarrow P.ip[i].val$
for $k \leftarrow 0$ **to** $length\ of\ data$
do **if** $data.wildcard[k].val \leq (V/sp) * w$
then $\{data.wildcard[k].childrens.remove()$
SORT$(data)$
$c \leftarrow 0$
$x \leftarrow 0$
for $j \leftarrow 0$ **to** $length\ of\ data$
do **if** $c\%2 = 0$
then $\{flow[j\%sp] \leftarrow data.wildcard[j]$
else $\{flow[(sp-j)\%sp-1] \leftarrow data.wildcard[j]$
$x \leftarrow x + 1$
if $x\%sp = 0$
then $c \leftarrow c + 1$
CHANGE_TO_RESTAPI$(flow)$

4 Experimental Environment and Results

In this section, we describe our experimental methodology in 3 phases. First, we depict our experimental environment including hardware and software. Second, we present our experimental results consists of prior experiments, prediction experiments, web service delay experiments, and load balancers experiments. Finally, we discuss the experimental results and comparison of four Different Methods of Load Balancing.

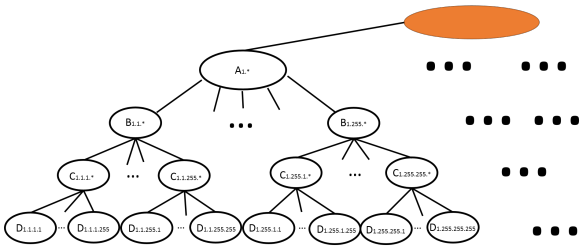

Fig. 8 Part of Wildcard Tree

4.1 Experimental Environment

The setting of our experiment is divided as the client side, the server side, and three testbeds. As shown in Figure 9, the clients connect to the server through the OpenFlow switch that is implemented with a wildcard-based load balancer. The client side consists of four groups, and each group has the unique behavior pattern as shown in Table 3. First, we have the always online users group that still connects to the server. Second, we have regular users group that connects to the server at a regular time. For example, the department staff work between 8 a.m. to 6 p.m. and they may play an online game between 7 p.m. to 11 p.m. after working hours, so those users connect to the company's service between 8 a.m. to 6 a.m. and connect to the game service between 7 p.m. to 11 p.m. regularly. Third, we have random users group which connects to the server at random time. For example, many people have mobile phones and use the web service such as Facebook on them. They could access web services any time and any place instead of at a regular time. Last, we have new users group which connects to the server at the first time. We predict future traffic based on user's traffic connected to the server in the past; however, services must have new users, so we add this new users group in our experiment. In the server side, we created load balance servers by OpenStack. We created eight servers and compared the results of R. Wang [20].

We created 500 to 1000 IPs for each users group except the new user's group and selected 1 to 50 IPs from each group to connect to the server with the time based on the behavior pattern of each users group. We also created 1 to 20 new IPs for the new users' group. Each IP has 1 to 10 connections at the same time, and each connection carries data with a random size. For the server, we have the service port that responds to basic packets.

We had three testbeds. In the first testbed, we used Mininet [4] to simulate all the server side, client side, and OpenFlow switch. As shown in Figure 9, we simulated eight servers to implement load balancing. In the second testbed, we implemented cloud services on the server side with OpenStack [5] and established eight virtual machines as servers as shown in Figure 10. We used

Table 3 The client side specification on experiment

Group type	Group name	Number of IP	Used IP
Connected before	Always online	100 to 200	1 to 25
	Regular	100 to 200	1 to 25
	Random	100 to 200	1 to 25
No connected before	New	$2^{24} - 1 - 1500$ to $2^{24} - 1 - 3000$	1 to 10

a physical OpenFlow switch to connect to cloud services, and all clients must go through the physical switch for services. In the third testbed, we applied the real web service's traffic collected from our department, and we used those data to replace data on the client side of the second testbed.

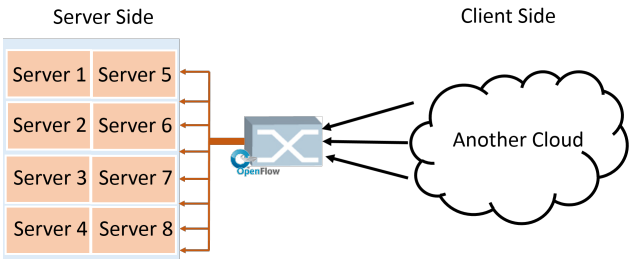

Fig. 9 Experiment Architecture

執行實例名稱	虛擬CPU數	硬碟	隨機存取記憶體	上線時間
lbserver1	2	10	1GB	4日, 21時
Norlb	2	10	1GB	4日, 21時
Lbserver2	2	10	1GB	4日, 21時
Lbserver3	2	10	1GB	4日, 21時
Lbserver4	2	10	1GB	4日, 21時
Lbserver5	2	10	1GB	4日, 21時
Lbserver6	2	10	1GB	4日, 21時
Lbserver7	2	10	1GB	4日, 21時
Lbserver8	2	10	1GB	4日, 21時

Fig. 10 VM Status in OpenStack

4.1.1 Hardware

In our experiment, we have implemented prediction mechanisms and databases on the controller. Our cloud service was built on three physical machines as shown in Table 4 and our client-side was another cloud service's virtual machine. The physical switch used was Netgear M5300.

Table 4 Hareware specification list

Target	Number	Specification
Controller and prediction server	1	CPU: I7-990 @ 3.47GHz, 6core, RAM: 8GB,
Cloud hosts	3	HDD: 2TB
VM Servers	8	CPU: 2core, RAM: 1GB, HDD: 10GB
VM Clients	4	CPU: 2core, RAM: 2GB, HDD: 20GB
OpenFlow Switch	1	Netgear M5300-52G

4.1.2 Software

We implemented the prediction mechanism and load balancing builder in Python. Our simulator used the Mininet, and we implemented OpenFlow switch with OpenvSwitch for both physical and virtual switches. Our OpenFlow controller used the Ryu controller because it is also applied in Python, comes with RESTful API, and supports multiple versions of OpenFlow. All used software versions are shown in Table 5.

Table 5 Software version list

Name	Version	Direction
OpenFlow	1.0.0 [6]	Version 1.0.0 is enough for our load balancing
Ryu	3.8.0	Our controller provider
OpenvSwitch	2.1.0	Implement our OpenFlow switch
Mininet	2.1.0	Using to simulate virtual environment
OpenStack	IceHouse	Establish our cloud services
MySQL	5.5.37	Store our traffic and training data
Python	2.7.3	Implement our code

4.2 Experimental Results

4.2.1 Prior Experiments

Two preliminary experiments were performed in the first testbed, i.e., is the virtual switch testbed. First, we tested the effect of enabling the mirror function. We created one to ten clients to ping the virtual IP which was to be implemented with load balancing. To avoid the impact from updating the flow table, we did not update wildcard rules during the test. As shown in Figure 11 to 13, the mirror function causes delays no matter how many numbers of connections are used. Although enabling the mirror function will increase time cost, it is indispensable for monitoring the network.

In the second preliminary experiment, we tested the effect of updating the flow tables. We created two clients and set IP 10.0.0.5 to client1 and IP 10.0.0.14 to client2. We update the wildcard rule which was /24 to /32 every ten seconds from 0 to 80 seconds. We then updated different destination server

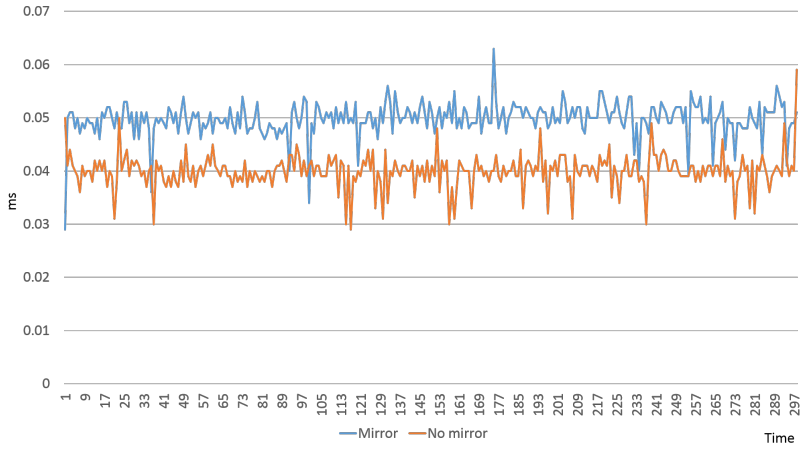

Fig. 11 Testing delay time for one connection

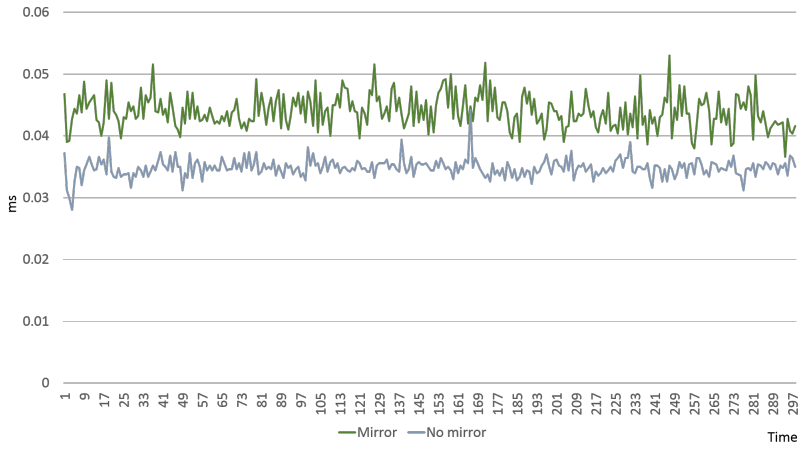

Fig. 12 Testing delay time for five connection

IPs every ten seconds from 90 to 110 seconds. At last, we updated both the wildcard rule and destination server IPs every ten seconds from 120 to 170 seconds. In the mixed test, we updated wildcard rule which was /26 to /31. Figure 14 shows that there is high latency at 29 and 39 second from both of two clients, due to that fact that the wildcard rule /27 and /28 influenced both two clients. At 56 second, only client2 was affected by the wildcard rule /30, and at 66 second client1 was affected by the wildcard rule /31. As shown in Figure 14, no perceivable effect comes from updating deferent destination server IPs. At 136, 146 and 156 seconds, we updated both the wildcard rule and destination server IPs. According to the above results, we can surmise that the effect of updating the wildcard rule is more than that of updating the

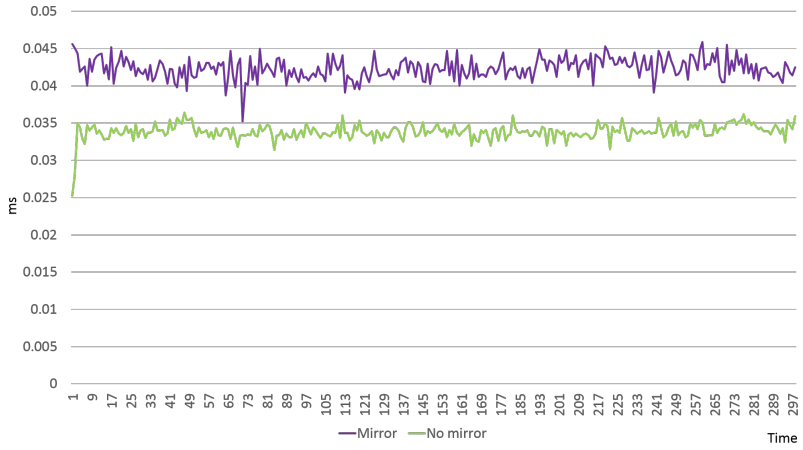

Fig. 13 Testing delay time for one connection

server IP. We used the script to send commands at above designated times automatically, and it also causes some latency.

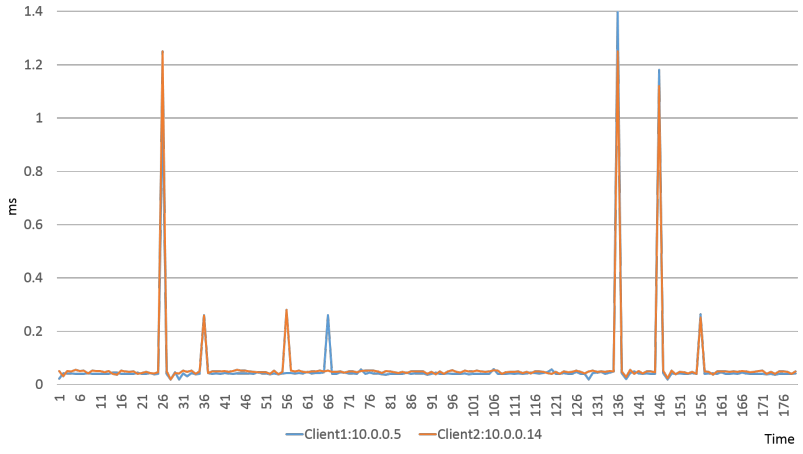

Fig. 14 Testing the effect of the update flow entries

4.2.2 Prediction Experiments

The first experiment is to test the prediction mechanism. In the experiments, we tested BPNN and K-Means clustering and observed two kinds of users with a low usage rate or high usage rate. The experiment was done on the second testbed in which OpenStack is used in the server side used, and a physical OpenFlow switch is used. In Figure 15, 16, 17, 18, the x-axis is hour in a day

and the y-axis is user traffic proportion that represents the user usage rate on this time. We obtained the user traffic proportion data at the first 5 minutes of every hour, which is set since we consider that the web service usually occupies a short period, and if the check time is too long, then the result will be inaccurate. However, performing the BPNN algorithm to predict will spend a lot of time so we cannot choose a shorter checking time. The y-axis is BPNN output value as shown in Figure 15, 16. These output values define the user usage rate on the next 5 minutes. From Figure 17, 18, the y-axis means K-Means clustering value which is defined in Equation 7.

As shown in Figure 15, 16, the BPNN output value curve is almost equal to the user usage rate. It means the BPNN prediction mechanism may not work accurately. Thus, we think it is possible to get the similar load balance result by using BPNN outputs and using traffic flows directly. Because of this, we will add direct traffic flows in load balance experiments. As shown in Figure

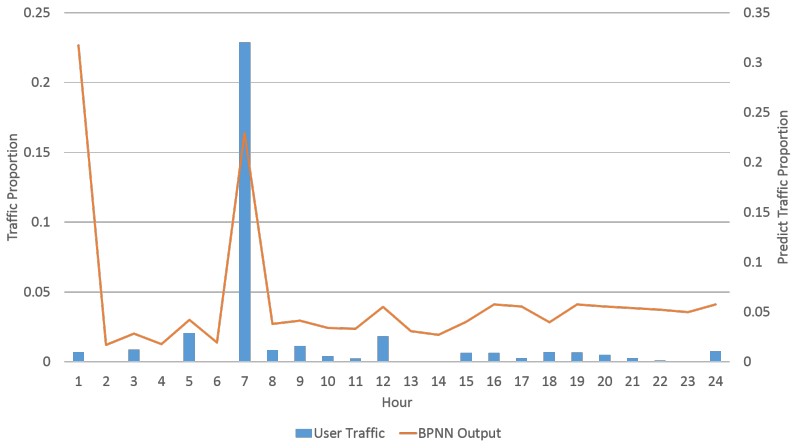

Fig. 15 BPNN Predictive Result by Lower Usage User

17,18 the K-Means classification is more inaccurate than the BPNN prediction mechanism for lower usage rate users and better than the BPNN mechanism for higher usage rate users.

4.2.3 Web Service Delay Experiments

The second experiment compares service delays of our module, the OpenDay-Light module, and the general load balancer. From Figure 19, the x-axis is the test time, and its unit is second. We got the request time every second for one minute. The y-axis is the web request delay time, and its unit is a microsecond. We get this value from the log file of Apache service. From this file, we can get every connection request time in microseconds. We used the BPNN prediction mechanism in the test and used data of high usage rate users. Fig-

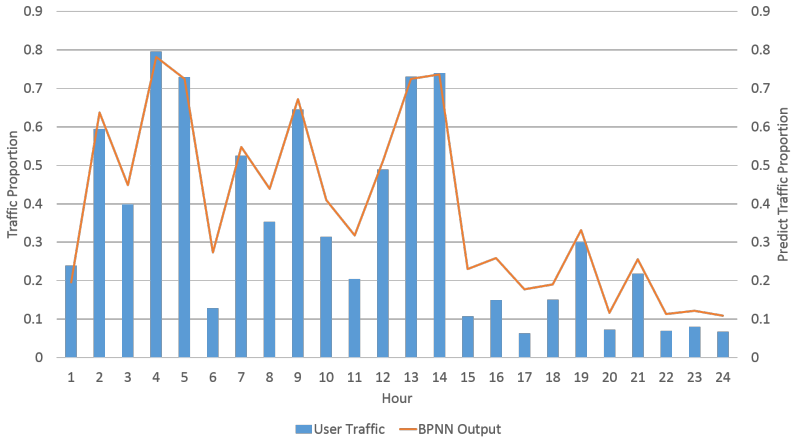

Fig. 16 BPNN Predictive Result by Higher Usage User

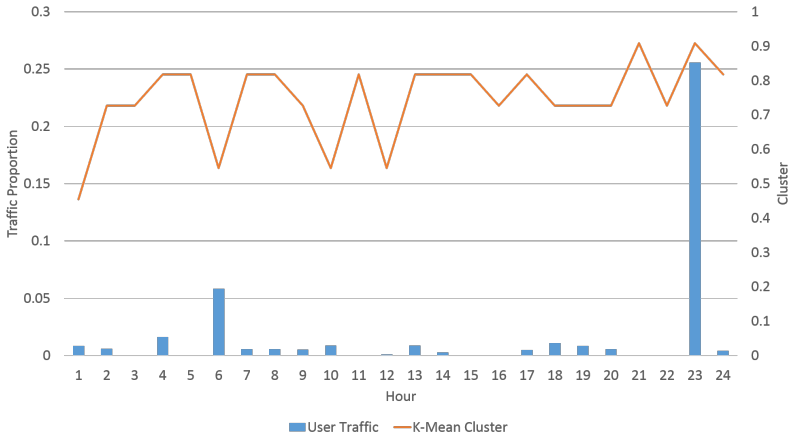

Fig. 17 K-Mean Classification Result by Lower Usage User

ure 19 shows that performances of our module and OpenDayLight module are better than that of the general load balance service and our module has better performance than the OpenDayLight module at the first connection. For the OpenDayLight module, the first flow of each IP needs to ask the controller to know which server to provide service for it and then adds or updates the flow entry to the OpenFlow switch. As shown in Table 6, the OpenDayLight adds flow entries for each IP, and our module uses about 100 flow entries for 2000 IPs with the wildcard.

4.2.4 Load Balancers Experiments

The last experiment tests performance of our load balancer by the realistic data. We compare results of our modules using three different prediction mech-

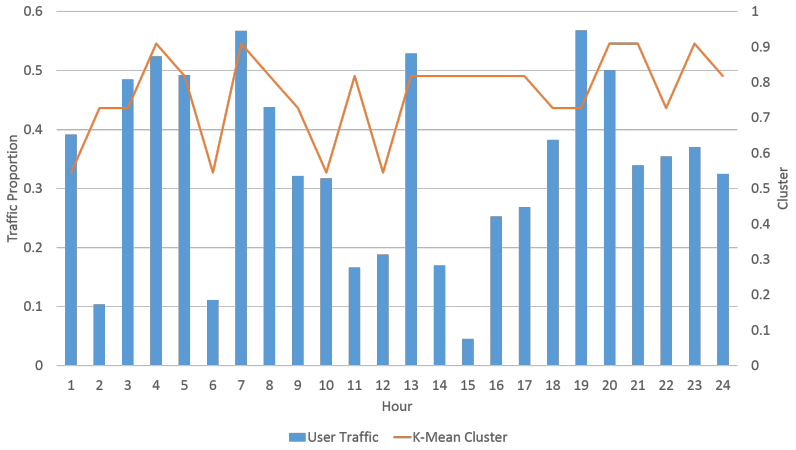

Fig. 18 K-Mean Classification Result by Higher Usage User

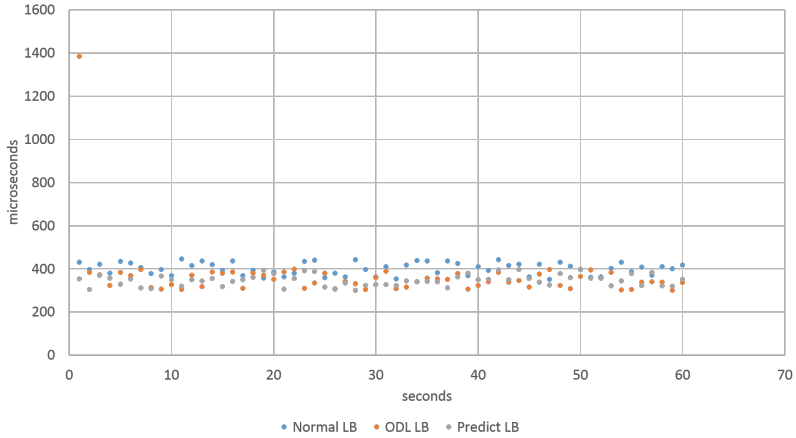

Fig. 19 Three Different Load Balance Request Time

Table 6 OpenDayLight Load Balancer and Wildcard Flow Entries Comparison

Modules	Number of flow entries	Number of IP
OpenDayLight	2000	2000
Wildcard	100	2000

anisms and the load balancer proposed by R. Wang [20]. We used the third testbed that uses real web services to test the load balancers. From Figure 20, 21, 22, 23, the x-axis represents the time in hours during April 12, 2013. The y-axis is the web network traffic, and the y-axis is the extra IP occurring rate. Each stack of bars means the flow assigned to each server. In this experiment, we resend all traffic during April 12, 2013. Our prediction mechanism uses training data collected from April 1, 2013, to April 11, 2013.

The load balancer of R. Wang et al. [20] just used four load-balanced servers in this experiment as shown in Figure 23. That means the real web service's clients only come from four kinds of wildcard rules defined by them. Figure 20, 21, 22 show our module can assign traffic to each server at some point and the best mechanism is load balance by bytes usage. We also found that the lower the extra IP occurrence rate is, the better our module performs. If the extra IP occurrence rate is too higher, the OpenDayLight module will outperform our module. We did not compare the performance of the general load balance service and OpenDayLight load balancer because they use more time delay to exchange for more load balance. However, our module is established with the aim to reduce delay time.

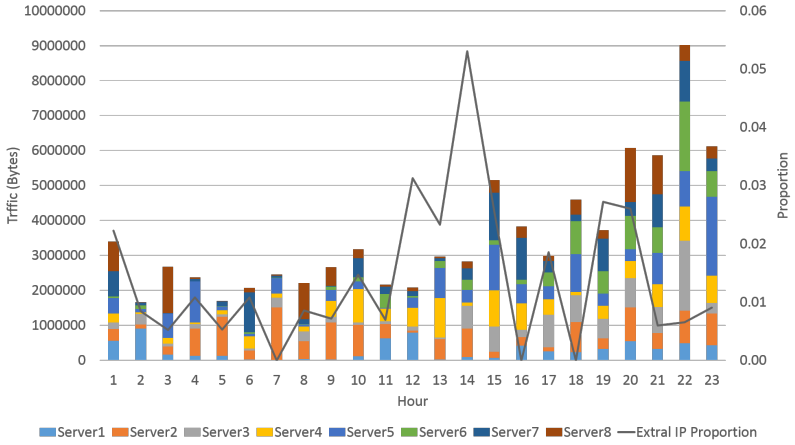

Fig. 20 Load Balance by Bytes Usage

4.3 Discussion

From the experimental results, we find that much delay is caused by updating flow entries. Thus, we must reduce the number to update flow entries. In the last experiment, the higher the extra IP occurrence rate is, the higher the entry updating frequency is. We plot the balance deviation of the four different methods used in the experiment in Figure 24. We can find our methods perform better than the load balancer of R. Wang et al. [20] and the load balance by bytes gives best results. We also combined the extra IP proportion from Figure 20, 21, 22, 23 as shown in Figure 25. We can find that performance of the load balancer of R. Wang et al. [20] is best and that of the load balance by bytes is second. From above two comparisons, we can find modules without prediction methods perform better than modules with prediction methods.

There are two issues in our module. The first issue is that prediction mechanism is not precise enough. Thus, we cannot uniformly assign flows. Figure

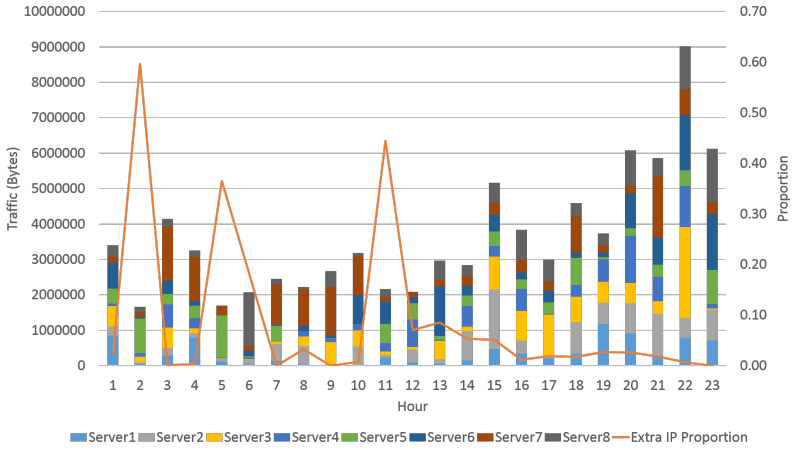

Fig. 21 Load Balance by BPNN

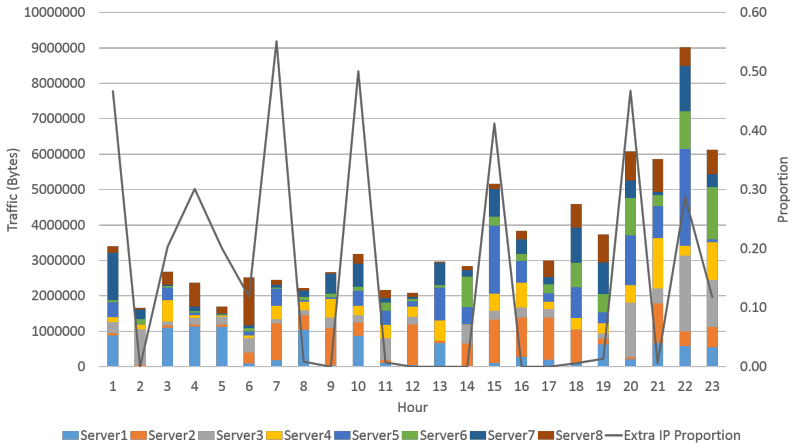

Fig. 22 Load Balance by K-Mean

24, 25 show that methods without prediction are better than methods with prediction. The second issue is that we cannot properly handle IP with suddenly large flow. In the experiment using real service data, it did include this condition, so we did not see it occur. In the future, we will try to solve these two issues.

Some methods can be included to improve our module. We can get packet data bit or TCP flag to determine if there is incoming packet at the next time. No packet data bit or TCP flag included in the specification of OpenFlow [6], but we can get that information by OpenvSwitch. In this work, we only used OpenFlow, so we did not add this policy to our module. We also can record web paths when users access. According to this information, we can improve the accuracy of the prediction mechanism. However, above information is too

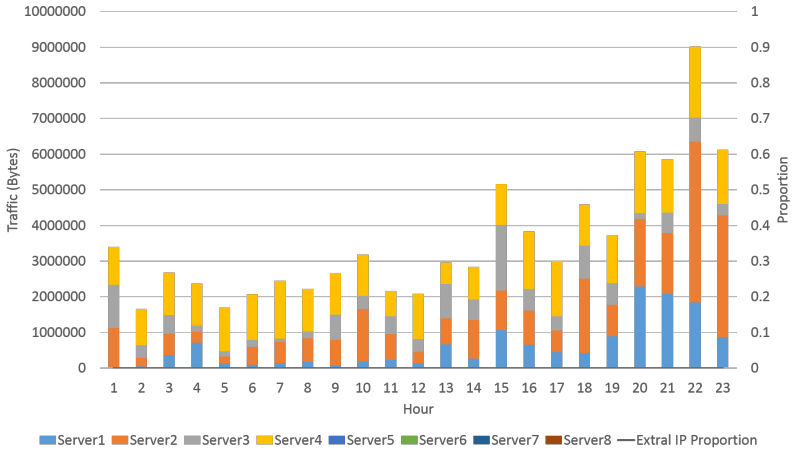

Fig. 23 Load Balance by Wildcard Only

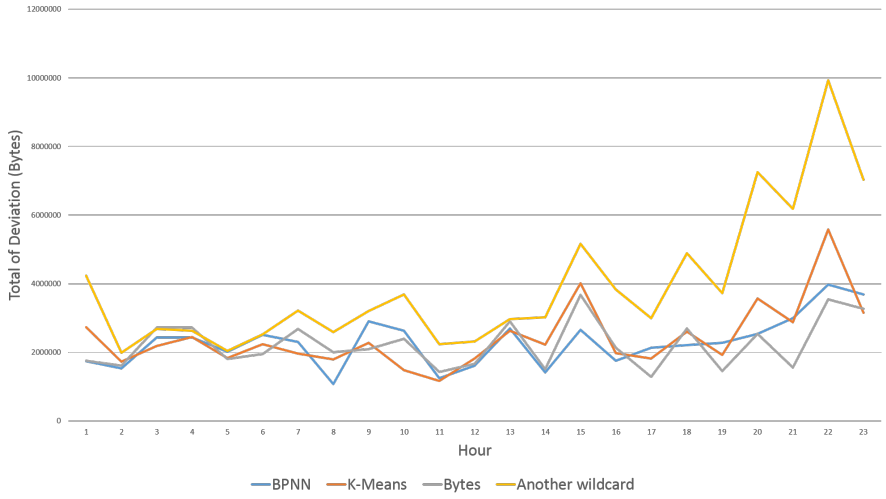

Fig. 24 Compare Four Different Methods on Balance

detailed if we want to design a module that can perform load balance for the variety of services instead of only for the web service. Thus, we focus on L2, L3 information of flow packet in this work.

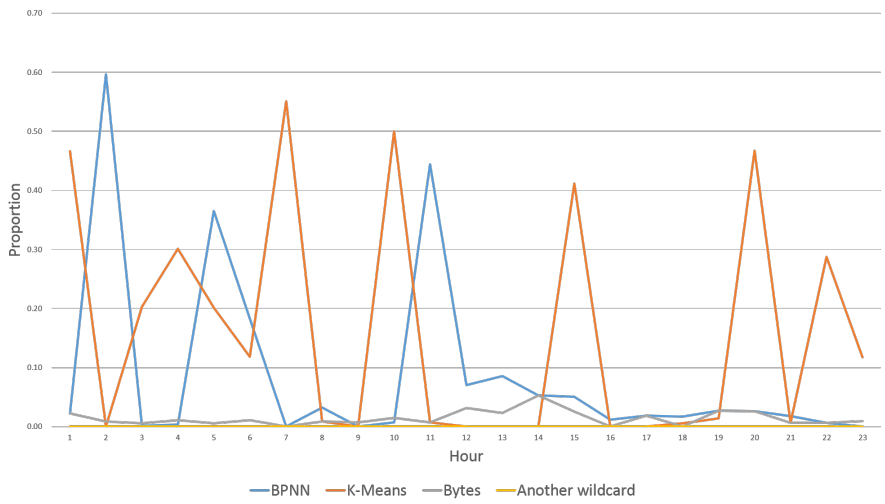

Fig. 25 Compare Four Different Methods on Extra IP Proportion

5 Conclusions and Future Work

5.1 Concluding Remarks

We establish an OpenFlow load balance module, which use different data mining algorithms based on neural networks and k-means to predict the future user traffic. In this work, we consider the problem of the delay on the general load balance service and the number of flow entries on the OpenFlow switch. We establish a new load balance module with OpenFlow and some data mining algorithms. In our experiments, we succeed in reducing average the web request time and the number of flow entries. In addition to above contribution, we also test the delay time using the mirror function with OpenFlow and updating wildcard rule on the OpenFlow switch. From the experimental results, we found our module is faster than the general load balancer and OpenDayLight load balancer. We compared results of our modules using three different prediction mechanisms: Load Balance by Bytes Usage, Load Balance by BPNN, Load Balance by K-Mean, and Load Balance by Wildcard Only. From this comparison, we can see our module can assign traffic to each server at some point, and the best mechanism is load balance by bytes usage. We also found that the lower the extra IP occurrence rate is, the better our module performs. If the extra IP occurrence rate is too higher, the OpenDayLight module will outperform our module.

5.2 Future Work

The prediction mechanism of the proposed load balancer in this work still have some unsolved issues as follow:

- We will test more data mining techniques such as the fuzzy neural network [21] [10], and the decision tree [12]. Additionally, we will consider more data set. After improving our load balance module, we will continue researching for more applications using OpenFlow such as high availability or to be part of a firewall.
- We also can integrate power-saving method [14] [35] [36] in our modules. We will integrate those OpenFlow applications for ARM based cluster in which every individual ARM machine is inefficient and heavy of network requirement. The ARM based cluster is power efficient, and it will be able to improve performance, reduce the burden, and improve overall performance after clustering [15] and integrating with OpenFlow applications.
- We had also tried to combine the fuzzy logic with the neural network to form a fuzzy neural network [21][10][13][23] but not yet finishing its implementation. We would use traffic classification to cluster different usage users by user traffic and allocate flows to different servers accordingly. We wanted to observe the effects of load balance by using different data mining techniques with reduced calculate times, rather using a neural network since it needs spending the relatively long time to train.

Acknowledgements This work was funded by the Ministry of Science and Technology (MOST), Taiwan, under grant number 104-2221-E-029-010-MY3 and 106-3114-E-029-003-.

References

1. H. Uppal and D, OpenFlow Based Load Balancing, University of Washington. CSE561: Networking. Project Report, 2010.
2. FloodLight Load Balancer: http://www.openflowhub.org/display/floodlightcontroller/Load+Balancer (2014).
3. OpenDayLight Load Balancer: https://wiki.opendaylight.org/view/OpenDaylight_Controller:Load_Balancer_Service (2014)
4. Mininet: http://mininet.org/ (2014)
5. OpenStack: https://www.openstack.org/ (2014)
6. OpenFlow Switch Spec. Version 1.0.0: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf (2014)
7. D. Puthal, B. Sahoo, S. Mishra, and S. Swain. "Cloud computing features, issues, and challenges: a big picture." In International Conference on Computational Intelligence and Networks (CINE), pp. 116-123, 2015.
8. S. Sharma, D. Staessens, D. Colle, M. Pickavet and Piet Demeester, "OpenFlow: Meeting carrier-grade recovery requirements ", Computer Communications, Vol. 36, pp. 656 665, 2013.
9. L.P. Maguire, B. Roche, T.M. McGinnity and L.J. McDaid, "Predicting a chaotic time series using a fuzzy neural network ", Information Sciences, Vol. 112, pp. 125 136, 1998.
10. T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning", IEEE Communications Surveys Tutorials, Vol. 10, pp. 56-76, 2008.
11. N. Williams, S. Zander, and G. Armitage, "A Preliminary Performance Comparison of Five Machine Learning Algorithms for Practical IP Traffic Flow Classification", SIGCOMM Comput. Commun. Rev., Vol. 36, pp. 5-16, 2006.
12. M. Zhani, H. Elbiaze and F. Kamoun, "Analysis of prediction performance of training-based models using real network traffic", IJCAT, Vol. 37, pp. 10-19, 2010.

13. C-T Yang, J-C Liu, K-L Huang and F-C Jiang, "A method for managing green power of a virtual machine cluster in cloud", Future Generation Comp. Syst., Vol. 37, pp. 26-36, 2014.
14. G. Dominik, K. Dimitri, M. Geveler, D. Ribbrock, N. Rajovic, N. Puzovic and A. Ramirez, "Energy efficiency vs. performance of the numerical solution of PDEs: An application study on a low-power ARM-based cluster", Journal of Computational Physics, Vol. 237, pp. 132 - 150, 2013.
15. P. Qin, B. Dai, B. Huang, G. Xu, "Bandwidth-Aware Scheduling with SDN in Hadoop: A New Trend for Big Data", CoRR, abs/1403.2800, 2014. http://arxiv.org/abs/1403.2800
16. M. H. Raza, S. C. Sivakumar, A. Nafarieh and B. Robertson, "A Comparison of Software Defined Network (SDN) Implementation Strategies", ANT/SEIT, pp. 1050-1055, 2014.
17. G. Yi and S. Lee, "Fully distributed handover based on SDN in heterogeneous wireless networks", ICUIMC, PP. 70, 2014. http://doi.acm.org/10.1145/2557977.2558047
18. P. Smith, A. Filho, D. Hutchison and A. Mauthe, "Management patterns: SDN-enabled network resilience management", NOMS, pp. 1-9, 2014.
19. R. Wang, D. Butnariu and J Rexford, "OpenFlow-Based Server Load Balancing Gone Wild", Hot-ICE, 2011.
20. Z. Wang, T. Hao, Z. Chen and Zhuzhi Yuan, "Predicting nonlinear network traffic using fuzzy neural network", Information, Communications and Signal Processing, 2003 and Fourth Pacific Rim Conference on Multimedia. Proceedings of the 2003 Joint Conference of the Fourth International Conference on, 2003.
21. C-T Yang, W-S Chen, Y-W Su, Y-Y Yang, J-C Liu, F.-Y Leu and W. Chu, "Implementation of a Virtual Switch Monitor System Using OpenFlow on Cloud", IMIS, pp. 283-290, 2013.
22. F. A. Rouai and M. B. Ahmed, "A new approach for fuzzy neural network weight initialization", Neural Networks, 2001. Proceedings. IJCNN '01. International Joint Conference on, 2001.
23. D. Puthal, N. Malik, S.P. Mohanty, E. Kougianos, and C. Yang. "The Blockchain as a decentralized security framework." IEEE Consumer Electronics Magazine Vol. 7(2), pp. 18-21, 2018.
24. D. Puthal. "Lattice-modelled Information Flow Control of Big Sensing Data Streams for Smart Health Application." IEEE Internet of Things Journal, 2018.
25. D. Puthal, N. Malik, S. P. Mohanty, E. Kougianos, and G. Das. "Everything you wanted to Know about Blockchain." IEEE Consumer Electronics Magazine, 2018.
26. D. Puthal, S. Nepal, R. Ranjan, and J. Chen. "Threats to networking cloud and edge datacenters in the internet of things." IEEE Cloud Computing, Vol. 3(3), pp. 64-71, 2016.
27. J. Erman, A. Mahanti, M. F. Arlitt and C. L. Williamson, "Identifying and discriminating between web and peer-to-peer traffic in the network core", WWW, pp. 883-892, 2007.
28. J. Erman, M. F. Arlitt, A. Mahanti, "Traffic classification using clustering algorithms", MineNet, pp. 281-286, 2006.
29. M. H. Dunham, "Data Mining: Introductory And Advanced Topics", Pearson Education, 2006.
30. H Nikhil, S. Seetharaman, N. Mckeown and R. Johari, "Plug-n-Serve: Load-Balancing Web Traffic using OpenFlow", ACM SIGCOMM, 2009.
31. F. Pakzad, M. Portmann, W. L. Tan and J. Indulska "Efficient topology discovery in OpenFlow-based Software Defined Networks ", Computer Communications, pp. 52 - 61, 2016.
32. J. Xie, D. Guo, Z. Hu, T. Qu and P. Lv, "Control plane of software defined networks: A survey ", Computer Communications, 2015.
33. P. Wang, G.Zhao and X. Yao, "Applying back-propagation neural network to predict bus traffic", 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD), IEEE, 2016.
34. M. Tiwary, D. Puthal, K. S. Sahoo, B. Sahoo, and L. T. Yang. "Response time optimization for cloudlets in Mobile Edge Computing." Journal of Parallel and Distributed Computing, 2018.
35. S. K. Mishra, D. Puthal, B. Sahoo, P. P. Jayaraman, S. Jun, A. Y. Zomaya, and R. Ranjan. "Energy-Efficient VM-Placement in Cloud Data Center." Sustainable Computing: Informatics and Systems, 2018.