Load Balancing Strategy for SDN Multi-controller ClustersBased on Load Prediction

Junbi Xiao (xiaojb@upc.edu.cn)
China University of Petroleum (East China)
Xingjian Pan
China University of Petroleum (East China)
Jianhang Liu
China University of Petroleum (East China)
Jian Wang
China University of Petroleum (East China)
Peiying Zhang
China University of Petroleum (East China)
Laith Abualigah
Al al-Bayt University

Research Article

Keywords: Software-defined networking, Deep Learning, SDN Multi-controller Cluster, Load Balancing, Switch Migration

Posted Date: May 3rd, 2023

DOI: https://doi.org/10.21203/rs.3.rs-2867519/v1

Additional Declarations: No competing interests reported.

Load Balancing Strategy for SDN Multi-controller Clusters Based on Load Prediction

Junbi Xiao[1*], Xingjian Pan[1], Jianhang Liu[1], Jian Wang[2], Peiying Zhang[1], Laith Abualigah[3]

[1*]Qingdao Institute of Software, College of Computer Science and Technology, China University of Petroleum (East China), Qingdao, 266580, China.
[2]College of Science, China University of Petroleum (East China), Qingdao, 266580, China.
[3]Computer Science Department, Al al-Bayt University, Jordan.

*Corresponding author(s). E-mail(s): xiaojb@upc.edu.cn;
Contributing authors: s21070007@s.upc.edu.cn; liujianhang@upc.edu.cn; wangjiannl@upc.edu.cn; zhangpeiying@upc.edu.cn; aligah@ammanu.edu.jo;

Abstract

Software-defined networking (SDN) separates the control layer from the data layer, and decisions to manage the network are issued through a controller. The distributed SDN architecture is an effective solution addressing modern WAN SDN architectures and allows multiple controllers to manage different parts of the network to ensure efficient and stable operation. To solve the problems of high switch migration cost, load imbalance, and inefficient load balancing in SDN multi-controller environments, we propose a deep learning-based controller load prediction switch migration (LPSM) strategy that uses a migration switch selection algorithm, target controller selection algorithm, and switch migration decision algorithm. Then, we propose a load balancing algorithm based on this decision algorithm. The final experimental results show that the LPSM reduces the migration cost by 16% and 8%, respectively, compared with time-sharing switch migration (TSSM) and distributed decision migration (DDM) strategies, reduces load variance from 0.02 to 0.004 compared with the DDM strategy, and improves load balancing efficiency by 27.6% compared with the TSSM strategy.

Keywords: Software-defined networking, Deep Learning, SDN Multi-controller Cluster, Load Balancing, Switch Migration.

1 Introduction

SDN provides dynamic attributes and professional programmable configurations, separates the control plane from the data plane, and transfers the network forwarding rules decision-making power to a centralized unit. This centralized unit is a controller, and it ensures that the network device only has a forwarding function; moreover, the controller makes corresponding flow rule forwarding actions according to the control plane [1]. The control plane acts as an intermediary between the data plane and the application plane, processes traffic in the network, and issues corresponding flow forwarding policies. The application plane is located above the control plane to implement customized application logic. The emergence of SDN reduces the operation and maintenance cost of the network and increases the network scalability.

Starting with the OpenFlow1.2 protocol, SDN has supported the OpenFlow switch to connect to multiple controllers at the same time. With the emergence of this feature and the increasing data requirements of SDN, SDN multi-controller clustering technology has emerged and gradually become a popular research topic in the field of SDN. It is proposed in [2] that the logical centralized control of SDN clusters can be achieved using physically distributed multiple controllers to improve the scalability and reliability of the control plane. However, in the traditional multi-controller SDN architecture, the connection between controllers and switches is static, and the load on the controllers rises as the number of switches managed by a particular controller in the cluster increases. The controller is the core of the SDN, and a surge in its load results in an elevation of the latency of the whole network, the severe waste of resources, and a reduction in fault tolerance.

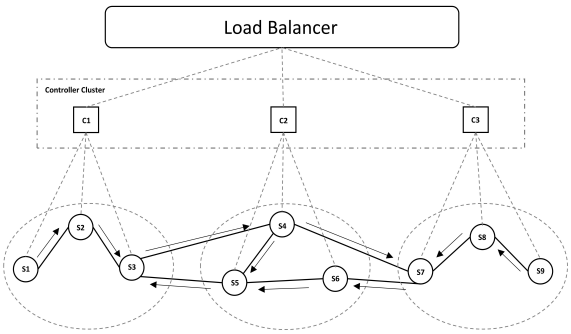


Fig. 1 Load calancer for SDN multi-controller cluster

SDN multi-controller load balancing technology shown in Figure 1 can be implemented via switch migration. Switch migration technology migrates some switches from overloaded controllers to other underloaded controllers to solve the problem of an uneven controller load. However, traditional switch migration processes generate a large amount of migration information flow, adding additional overhead to the system and increasing the migration cost, which greatly burdens the SDN. In addition,

traditional switch migration algorithms randomly select switches for migration among the set of switches managed by overloaded controllers, but they do not consider the number of migrations between switches and source controllers or the migration cost and distance, which increases the additional resource utilization and migration cost of controller nodes, aggravates the instability of controller clusters, reduces the efficiency of load balancing, and brings an additional burden to SDN. Therefore, it is extremely important to design an efficient switch migration mechanism and controller cluster load balancing strategy.

Our contributions are summarized below.

- We conducted extensive simulation training and found that the proposed LPSM strategy can reduce the migration cost by 16% and 8%, respectively, compared with the TSSM and DDM strategies.
- LPSM strategy reduces load variance from 0.02 to 0.004 compared with the DDM strategy, thus enhancing the resource stability of multi-controller clusters.
- Compared with the TSSM and DDM strategies, LPSM has a 27.6% improvement in load balancing efficiency.

The rest of the article is organized as follows: section 2 introduces prior research on this subject, section 3 introduces the network model and data model of the proposed algorithms, section 4 details the proposed algorithms, section 5 introduces the experimental environment and results, and section 6 presents a summary of our work and proposes future research directions.

2 Related Work

Previous studies considered the issue of controller load balancing from different perspectives, and thus developed many different load balancing algorithms. These approaches are described below.

Considering the load of the controllers themselves, Pang et al. [3] designed a network load balancing strategy based on SDN data centers that uses SDN controllers to centrally monitor the entire network and select paths to forward according to the real-time load of the network, which can achieve load balancing of the controllers. Singh et al. [4] designed the round robin algorithm to balance the load between SDN controller servers. To obtain better results in terms of latency, network speed, fault tolerance, and resource usage, Varalakshmi et al. [5] designed a mechanism using round robin, randomized and weighted round robin, and minimum link algorithms to reduce the throughput of SDN controllers. Thajeer et al. [6] used a hybrid algorithm as a server load balancing technique with a minimum link load balancing algorithm and weighted round robin load balancing algorithm, which flexibly balances the load among distributed SDN controllers.

Additionally, some studies implemented cluster load balancing strategies from the traffic and link perspectives. Wang et al. [7] combined SDN and link load algorithms to schedule elephant flows by analyzing the traffic characteristics of SDN data center networks, and used a scheduling algorithm to calculate re-routing to achieve load balancing of link traffic. Dafda et al. [8] used a genetic algorithm (GA) and Ant

Colony Optimization (ACO) load balancing for energy-aware routing to achieve link load balancing and minimize energy consumption. Given that traditional load balancing methods cannot effectively obtain statistics in network devices or consider the impact of many single load balancing factors, Jun et al. [9] proposed a path-server ant colony optimization (JPSACO) algorithm that considers the server level and customizes the performance metric server busyness (SBD) to quantify the real-time status of servers. Sun et al. [10] proposed the Yen-M model to address the challenges of load balancing and traffic scheduling in traditional networks based on the SDN network architecture and Yen algorithm, which optimizes the link bandwidth and effectively improves the load balancing efficiency compared to traditional load balancing algorithms. This model reduces the traffic scheduling delay and greatly improves the network transmission efficiency.

Regarding switch migration, most previous studies have been conducted on the dynamic deployment of switches. Liu et al. [11] proposed an efficient switch migration (HESM) strategy to balance the controller load for static controller-switch deployment facing load imbalance. HESM defines multiple load metrics to measure the controller load. The best target controller with the maximum remaining resources and the switch with the minimum migration cost are selected for migration, which improves the controller load balancing performance. Since flow table capacity is limited and unreasonable rule placement will lead to a flow table overflow problem, Yue et al. [12] proposed a controller load balancing scheme (CAR) based on rule placement and switch migration from the flow table perspective. The CAR algorithm reduces and balances the controller load, achieves lower latency than FlowStats algorithms, and reduces packet latency. Yao et al. [13] proposed a deep Q-learning (DQN)–based switch migration scheme by combining the powerful sensing capability of deep learning and the decision-making capability of Q-learning, and it outperformed the traditional approach in terms of controller server resource utilization and load balancing capability. Babbar et al. [14] proposed a dynamic QoS-aware load balancing switch migration algorithm (LBSMT) in combination with a traffic-only system, which showed a large improvement in CPU utilization, memory utilization, throughput, and response time compared to traditional algorithms. Filali et al. [15] proposed a multi-step ARIMA prediction model, that can be used to predict long-term controller load, and in which switch migration operations are scheduled in advance based on the prediction results.

As can be seen, most previous studies considered only one or a few factors; however, there are many aspects that need to be considered to make SDN controller clusters achieve load balancing using switch migration techniques. For load balancing and switch migration issues, existing research results are not satisfactory, and there are many factors that have not been considered. Therefore, it is very important to design sound and efficient load balancing and switch migration algorithms.

3 System Model

This section presents the network model. First, we abstract the network model into a general data structure model and define some parameters in the network. Then, we describe the controller load balancing problem and give a load balancing example.

Finally, we analyze the switch migration protocol. For ease of reading, we summarize the variables used in this section in Table 1.

3.1 Network Model

3.1.1 Network Data Structure

SDN involves many nodes and links, and thus we use data structure graph $G = (V, E)$ to represent an SDN topology, where V denotes nodes and E denotes edges. Nodes are composed of a set of switches S and a set of controllers C. For each controller $C_i \in C$ in the network, all of the switches in S are connected to one controller, which means that each controller can act as the master controller for each switch in S. This is called logically separated and physically centralized control. Each switch in S has a master controller, and their master controller may change later in the migration strategy. We also define $S_j^{C_i}$ as switch S_j whose master controller is C_i. Each switch must have a unique controller so that $\sum S_j^{C_i} = S$, $S_k^{C_i} \bigcap S_l^{C_j} = \varnothing$ for any two controllers C_i and C_j in C.

3.1.2 Protocol Introduction

In the OpenFlow protocol, the controller load primarily comes from communication with the switch, which mainly includes $Packet\_In$ messages and $Packet\_Out$ messages. The purpose of the $Packet\_In$ message is to send packets arriving at the OpenFlow switch to the OpenFlow controller. $Packet\_In$ messages can be sent in the following two cases:

- if there is no item that matches the flow table entry (Table-miss), OFPR_NO_MATCH;
- if the action recorded in the matching flow table entry is "Send to OpenFlow controller", OFPR_ACTION.

The $Packet\_Out$ messages are sent from the OpenFlow controller to the OpenFlow switch and contain packet delivery commands. When a controller wishes to send a $Packet\_Out$ message through the datapath, it uses the OFPT_PACKET_OUT message through a specific port of a switch. Therefore, the main load on the controller comes from the processing of $Packet\_In$ messages [16, 17]. When a large number of $Packet\_In$ messages are sent to the controller, a corresponding number of $Packet\_Out$ messages need to be sent from the controller, which takes up significant controller resources.

We define α_{S_i} as the number of $Packet\_In$ messages received by the controller from switch $S_j^{C_i}$ in a period of time, and β_i as the number of $Packet\_Out$ messages sent by controller C_i in a period of time. Within a certain threshold (which varies for each controller and is defined as δ), the rate of $Packet\_Out$ messages generated by the controller increases linearly with the rate of received $Packet\_In$ messages over a period of time. After this threshold is exceeded, the rate of $Packet\_Out$ messages no longer trends linearly and the derivative of the rate decreases as the rate of $Packet\_In$ messages is boosted due to the limited processing power of the controller. Load $L_t^{C_i}$

Table 1 Summary of variables

Symbol	Description
S, C	Set of all switches and controllers
S_i, C_i	A switch or controller in a switch or controller set
$S_j^{C_i}$	Switch S_j whose master controller is C_i
α_{S_i}	Packet_In messages received rate
β_i	Packet_Out messages received rate
δ	Controller's Packet_In process capacity
$L_t^{C_i}$	Load of a controller C_i in a period of time t
C_{tar}	Migration target controller
C_{ori}	Migration origin controller
S_{mir}	Switch to be migrated
T	Average propagation delay between switch and controller
μ	Mean value of $L_t^{C_i}$ (used in Equation (2))
σ	$L_t^{C_i}$ standard deviation (used in Equation (2))
λ	Poisson distribution parameter (used in Equation (2))
net_i	Network traffic size between switch and controller
d_i	Load balancing index of a controller
Δf	Change in the number of flow request messages
$cost_{mir}$	Consumption of migration message transmission
$f(S_i, C_j)$	Received rate of flows from switch S_{C_i} to controller C_i
ω_{rule}	Number of Flow mod messages sent by a controller
$h_{i,j}$	Minimum number of hops from switch $S_i^{C_j}$ to controller C_j
Γ_i	Switch migration cost
ξ_i, τ_i, k_i	Corresponding weights parameters (used in Equation (6,8,9))
U_{cpu}	CPU utilization
U_{mem}	Physical memory usage
U_{net}	Network bandwidth usage
n	Number of switches in network
r	Mean link latency from switch to controller
T_{out}	Network throughput
L_{sw}	Switch mean load
U_{res}	Average remaining resource utilization of a controller
η_{C_i}	Actual load factor of a controller C_i
W_{C_i}	Load weight of a controller, target controller parameters
W_t	Weight threshold
$State_{C_i}$	Controller overload state
ρ_{C_i}	Controller resource utilization

in a period of time t of controller C_i is the sum of $L_t^{C_i}$.

$$L_t^{C_i} = \sum_{S_i \in S} \alpha_{S_i} \tag{1}$$

3.2 Controller Load Balancing

An SDN-based load balancer allows for the control of multiple devices. Therefore, software-defined networks are more agile than traditional networks. Software-defined network control can be programmed directly for more responsive and efficient application services. Figure 2 shows an example of the SDN load balance problem. There are three controllers and nine switches in the cluster: $C = \{C_1, C_2, C_3\}$, and $S = \{S_1, S_2, ..., S_9\}$. Each of the four controllers manages a different sub-domain network. Controller C_1 is only responsible for processing $Packet\_In$ messages sent by switches S_1 and S_2; controller C_2 is only responsible for processing $Packet\_In$ messages sent by switches S_3, S_4, S_5, S_6, andS_7; and controller C_3 is only responsible for processing $Packet\_In$ messages sent by switches S_8, andS_9. When a $Packet\_In$ message flows into the controller, the controller will process the $Packet\_In$ message and send the corresponding $Packet\_Out$ message. When there are excessive switches under a controller, the number of $Packet\_In$ messages will become larger. Then, due to the limited processing capacity of the controller, the processing of $Packet\_In$ messages will consume a significant amount of the controller's resources, making it unable to handle other network services. This causes massive network packet congestion, which will affect the efficiency of the controller and indirectly affect the network's transmission quality.

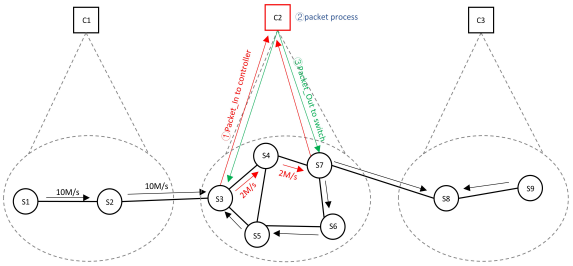


Fig. 2 Imbalanced multi-controller cluster in SDN

Figure 3 shows the solution to the load balancing problem. Originally, C_2 is an overloaded controller. Switches S_4 and S_7 on C_2 are migrated to the network by C_1 and C_3, respectively, thus reducing the load on C_2 and rebalancing the load across all of the cluster controller nodes. The load balancing strategy allows C_2 to free up many resources, the switches connected to C_2 return to a normal communication level, and the whole network returns to a stable state.

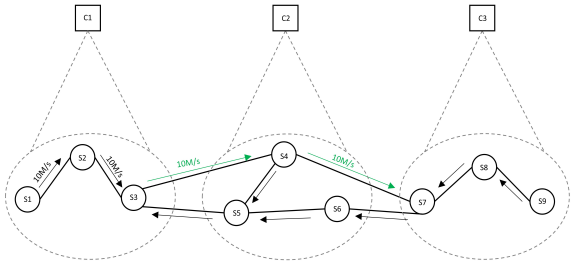


Fig. 3 Balanced SDN with a multi-controller cluster

In our model, to measure controller load, we define the network traffic size between a switch and controller in Equation (2), where T denotes the average propagation delay between the switch and controller, μ denotes the mean value of $L_t^{C_i}$, and σ denotes its standard deviation. We assume that the flow density λ in our experiments obeys a Poisson distribution.

$$net_i = e^{[-(1-\lambda)\frac{\sigma}{\mu}T]} \tag{2}$$

We use the quotient of the controller's load and controller capacity threshold δ to determine the load balancing index of controller C_i. The quotient d_i can be calculated by Equation (3).

$$d_i = \frac{net_i}{net_\delta} \tag{3}$$

In effective load balancing, the most important thing is to select the target controller set C_{tar} and the switch to be migrated S_{mir}.

Selection of migrating switches

In this paper, we select a proper set of migrating switches by minimizing the cost of switch migration. The cost of switch migration is expressed in Definition 1.

Definition 1 *The cost of switch migration: the cost of switch S_i being migrated to the sub-domain network controlled by controller C_j is defined as a linear combination of three performance metrics: the change in the number of flow request messages Δf, the cost of migration message transmissions $cost_{mir}$, and controller load balancing index d_i.*

The change in the number of flow request messages Δf can be calculated by

$$\Delta f = \sum_{S_i \in S_j^{C_i}} f(S_i, C_i), \tag{4}$$

where $f(S_i, C_j)$ represents the number of flows from switch S_{C_i} to controller C_i over a period of time.

The consumption of migration message transmissions $cost_{mir}$ can be calculated by

$$cost_{mir} = \omega_{rule} \cdot \alpha_{S_i} \cdot h_{i,j}. \tag{5}$$

When a switch is migrated, its master controller needs to deploy the migration rule message $Flow\_mod$ to the switch. We use $omega_{rule}$ to represent the resources consumed by the controller C_i to process $Flow/_mod$ messages. Meanwhile, $h_{i,j}$ represents the minimum number of hops between switch $S_i^{C_j}$ to controller C_j. When the switches are migrated far away, the change in the number of flow request messages Δf will be larger after the switches are migrated. When the network size is large and a large number of switches need to be migrated, the controller needs to install a large number of migration rules on the switches, which also results in a large migration message transmission cost $cost_{mir}$. When the load imbalance between multiple controllers is severe, the controller load balancing index d_i will be large after the switches are migrated. Therefore, from the analysis mentioned above, it can be seen that multiple cost metrics need to be measured simultaneously during switch migration to obtain the total migration cost. In a practical solution, the switch migration cost Γ_i is calculated by Equation (6), where ξ_1, ξ_2, and ξ_3 are the corresponding weights of parameters Δf, $cost_{mir}$, and d_i, respectively.

$$\begin{aligned} \Gamma_i &= \xi_1 \cdot \Delta f + \xi_2 \cdot cost_{mir} + \xi_3 \cdot d_i \\ s.t. &\begin{cases} 0 < \xi_1, \xi_2, \xi_3 < 1 \\ \xi_1 + \xi_2 + \xi_3 = 1 \end{cases} \end{aligned} \tag{6}$$

Selection of target controllers

The selection of the target controllers requires the consideration of not only the load of the overloaded controller but also the CPU utilization U_{cpu} and network bandwidth usage U_{net} of the controller. This is because on the physical machine running the controller, the CPU, memory, and actual network bandwidth affect how efficiently the controller can process network information. We assume that n is the number of switches, r is the mean link latency from switch to controller, and T_{out} is the network throughput. We define the switch mean load in Equation (7).

$$L_{sw} = \sum_{S_i \in S_j^{C_i}} r/n \tag{7}$$

Controller resource utilization U_{res} is defined as

$$U_{res} = \tau_1 \cdot U_{cpu} + \tau_2 \cdot U_{net} + \tau_3 \cdot U_{mem}. \tag{8}$$

The actual load factor η of a controller C_i is defined as

$$\eta_{C_i} = k_1 \cdot L_{sw} + k_1 \cdot U_{res} + k_1 \cdot T_{out}. \tag{9}$$

During switch migration, the controller's resource consumption mainly includes the processing of flow request messages, migration consumption, and state synchronization consumption between controllers. The formula for calculating the controller resource utilization is

$$\rho_{C_i} = \frac{1}{\eta_{C_i} \cdot |S_j^{C_i}|} \cdot \sum_{S_j^{C_i} \in S} (\eta_{C_i} - L_t^{C_i}). \tag{10}$$

According to parameters $U_{res}, \eta_{C_i}, and \rho_{C_i}$, we define variable W_{C_i} to represent the weight of a controller. The higher the weight, the more likely controller C_i is to be selected to be in C_{tar}:

$$W_{C_i} = \frac{L_t^{C_i}}{\eta_{C_i}}. \tag{11}$$

In addition, the weight W_{C_i} of controller C_i needs to be judged. If the judgment standards are high, the controller may be in high load all the time without reaching the overloaded threshold. If the judgment standards are low, the controller may keep migrating switches, add migration consumption, and increase resource utilization. Therefore, in this paper, we additionally set a weight threshold W_t; only if the calculated weight W_{C_i} is always above this threshold in time t, the migration operation will be executed. Here, a value of 0 means controller C_i is overloaded, while a value of 1 means controller C_i is not overloaded. We abstract this into the mathematical model presented in Equation (12).

$$State_{C_i} = \begin{cases} 0, & W_{C_i} < W_t \\ 1, & W_{C_i} > W_t \end{cases} \tag{12}$$

3.3 Switch Migration Process Analysis

Switch migration is the process of selecting switch S_i to be migrated from switch set S_{C_i} of the overloaded controller C_i and migrating it to the sub-domain network of the underloaded controller C_j to achieve a load-balanced distribution cluster. The main step is to use OpenFlow role messages between the source controller, migrating switch, and target controller to change the controller's control over the switch. Based on this feature of OpenFlow protocol, we can improve the switch migration mechanism.

In this paper, we divide the switch migration process into four stages as shown in Figure 4.

Stage 1: Change the target controller to the Equal controller of the switch to be migrated.

The source controller C_{ori} receives the start migration message and sends the *Role_Request* message to the switch to be migrated S_{mir}, requesting to become its Equal controller. The target controller C_{tar} receives the *Role_Request* message and sents a *Role_Reply* message indicating to C_{ori} that it is ready to migrate. At this stage, C_{ori} remains the only master controller of the switch to be migrated because C_{tar} does not respond to asynchronous messages from S_{mir} after it becomes the Equal controller of S_{mir}, thus ensuring both security and activation characteristics.

Stage 2: Determine precise migration times by inserting and deleting flow table entries.

C_{ori} inserts an empty flow table entry into S_{mir} via the $Flow\_Mod$ command, determines that this $Flow\_Mod$ message is completed, and then removes the flow table entry using the $Flow\_Mod$ command. The $Packet\_In$ messages sent by S_{mir} before the $Flow\_Remove$ messages are processed by C_{ori}, and those sent afterward are processed by C_{tar}, using the time of receipt of the $Flow\_Removed$ messages as the separation point. This ensures that all of the $Packet\_In$ requests from the migrating switch are processed.

Stage 3: Origin controller processes the switch legacy $Pakcet\_In$ requests.

C_{ori} asks S_{mir} via a $Barrier\_Request$ message if the previous $Packet\_In$ message has been processed, and after S_{mir} sends a $Barrier\_Reply$ message to C_{ori}, C_{ori} sends an end-of-migration message to C_{tar}.

Stage 4: Make C_{tar} become the master controller of S_{mir}.

After controller C_{tar} sends a $Role\_Request$ message to S_{mir} to become its master controller, S_{mir} sends a $Role\_Reply$ message to controller C_{tar} so that controller C_{tar} becomes the master controller of S_{mir} and C_{ori} becomes the slave controller of S_{mir}.

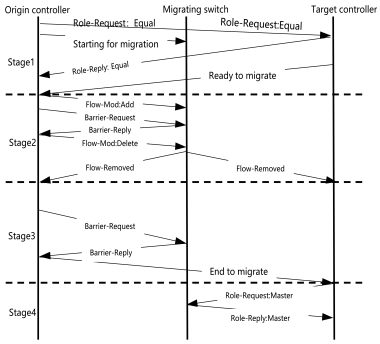


Fig. 4 Switch Migration Process

Both the SDN multi-controller load balancing problem and the switch migration problem have been proven to be NP-hard problems[18]. An NP-hard problem is difficult to solve in polynomial time. Therefore, we propose a target controller selection algorithm, a migration switch selection algorithm, a switch migration algorithm, and a load balance algorithm to solve these two problems.

4 Algorithm Design

We propose four algorithms to solve the SDN multi-controller cluster load balancing problem. First, we propose a target controller selection algorithm, which selects suitable controllers as a migration target based on parameters such as switch load, CPU utilization, network throughput, and controller processing power. Then we propose a

migration switch selection algorithm, which selects the switches to be migrated based on parameters such as Poisson distribution variables, quotient variables, number of flow requests, and migration costs. Next, we propose a switch migration algorithm to achieve controller cluster load balance. Finally, we propose a load balance algorithm to make the multi-controller cluster achieve a load-balanced state.

4.1 Selection of target controller

As we mentioned earlier, all of the algorithms and experiments need to be based on the fact that all of the switches will be connected to every controller in the cluster, but each switch has one and only one master controller while all of the other controllers are slaves. This is the logical connectivity physical blocking that we mentioned earlier. The purpose of this selection step is to facilitate our subsequent switch migration actions.

Algorithm 1: Selection of target controller algorithm

Input: $S, C, L_{sw}, \eta_{C_i}, \omega_{C_i}$
Output: C_{ov}, C_{tar}
1 initialize: the number of $S_j^{C_i}$ ($|S_j^{C_i}| \leftarrow N_S$); the number of C ($|C| \leftarrow N_C$); underloaded controller set $C_u \leftarrow 0$; overloaded controller set $C_{ov} \leftarrow 0$;
2 **for** $1 \leq i \leq N_C$ **do**
3 **for** $1 \leq j \leq N_S$ **do**
4 calculate $State_{C_i}$ based on input parameters;
5 **if** $State_{C_i} == 1$ **then**
6 $C_u.add(C_i)$;
7 **else**
8 $C_{ov}.add(C_i)$;
9 **end**
10 **end**
11 **end**
12 **for** $C_i \in C_u$ **do**
13 calculate ρ_{C_i};
14 **end**
15 $sorted(max(\rho_{C_i}))$;
16 choose $max(C_i) \leftarrow C_{tar}$;
17 $sorted(min(\rho_{C_i}))$;
18 choose $min(C_i) \leftarrow C_{ov}$;
19 **return** C_{ov}, C_{tar};

The target controller algorithm first needs to input the set of switches and set of controllers into the network, and then initialize the number of switches controlled by each controller. It also initializes the set of overloaded controllers and underloaded controllers. Lines 2–11 of algorithm 1 iterate through the switches on each controller and calculate the controller parameters based on Equations (7)(8)(9). $State_{C_i}$ is derived from these parameters, and controllers with a $State_{C_i}$ of 1 are added to the set of

underloaded controllers; controllers with $State_{C_i}$ of 0 are added to the set of overloaded controllers. Lines 12–14 of algorithm 1 calculate ρ_{C_i} for each controller in the set of overloaded controllers, and finally sort C_{ov} and C_{tar} by ρ_{C_i} from largest to smallest and smallest to largest, respectively. The reason for sorting the sets by size is that the controller with the largest ρ_{C_i} (i.e., the controller with the most remaining resources) will be served first, and vice versa. Finally, the algorithm 1 returns the source and destination controllers, and the target controller is calculated based on this returned information.

4.2 Selection of migrating switch

The migrating switch selection algorithm first needs to input the result of algorithm 2 to obtain the overloaded controllers C_{ov} and C_{tar}, and then must determine the minimum number of hops from switch $Si^{C}tar$ to controllers C_{tar} and $cost_{mir}$. Line 1 of algorithm 2 initializes the set of all of the switches controlled by the overloaded controller C_{ov} and the set of migration switches S_{mir}.

Algorithm 2: Selection of migrating switch algorithm

Input: $C_{ov}, C_{tar}, h_{i,j}, cost_{mir}$
Output: S_{mir}
1 initialize: The number of switches controlled by C_{ov} $|S_{ov}| \leftarrow N_{ov}, S_{mir} \leftarrow 0$;
2 **for** $1 \leq i \leq N_{ov}$ **do**
3 calculate net_i, d_i, andΔf;
4 calculate Γ_i under the constraints in equation(6);
5 **end**
6 build migrating cost set S_{cost};
7 $sort(min(S_{cost}))$;
8 **for** $1 \leq j \leq N_{ov}$ **do**
9 **if** $State_{C_{ov}} \neq 1$ **then**
10 update $\rho_{C_{ov}}$;
11 $S_{mir}.add(S_{cost})$;
12 $N_{ov} \leftarrow N_{ov} - 1$;
13 **end**
14 **end**
15 **return** S_{mir};

Lines 2–6 of algorithm 2 are a for loop that iterates through the parameters $d_i, \Delta f$, and $cost_{mir}$ of the switch controlled by the overloaded controller; these parameters are then used to calculate the migration cost for each switch and finally employed to construct the switch migration cost set S_{cost}. Lines 7–14 of algorithm 2 start selecting migration switches. Specifically, line 7 sorts S_{cost} according to migration cost per switch from smallest to largest; then, the for loop from lines 8–14 determines the current overloaded controller load based on the overload status. The switch with

the smallest migration cost in S_{cost} is added to the set S_{mir}, and then the remaining resource utilization of the overloaded controller is updated until the overloaded controller is no longer overloaded.

4.3 Switch migration

The switch migration algorithm requires an LSTM prediction model to predict the load of the controllers in the cluster. For this purpose, we monitor the load variation of all of the controllers in real time, preprocess the load data first, and then pass it forward in batches into the model to derive the corresponding load prediction results.

Algorithm 3: Switch migration algorithm

Input: $\xi_1, \xi_2, \xi_3, U_{cpu}$
Output: Migration Action$(C_{ori}, C_{tar}, S_{mir})$

```
1  initialize: |C_tar| ← null, |S_mir| ← null;
2  LSTMPrediction() ← S_candi, C_candi;
3  calculate the network traffic size net_i between S_candi and C_candi;
4  for 1 ≤ i ≤ |C_candi| do
5      for 1 ≤ j ≤ |S_candi| do
6          Dijkstra(hop_{i,j}) ← h_{i,j};
7          calculate migration cost ← cost_mir;
8          Algo2(C_ov) ← S_mir;
9      end
10     calculate network throughput ← T_out;
11     calculate controller resource utilization ← U_res;
12     get W_{C_candi} and State_{C_candi};
13     Algo1(S_candi, C_candi, L_sw, η_{C_i}, ω_{C_i}) ← C_ov, C_tar;
14     record average remaining resource utilization of each controller in C_tar;
15 end
16 sort(max(C_tar));
17 for S_i ∈ S_mir do
18     match S_mir and C_tar;
19     MigrationAction(C_ori, S_mir, C_tar);
20 end
21 return C_ori, S_mir, C_tar;
```

In line 2 of the switch migration algorithm, we first obtain the candidate migration switches and candidate target controller in the cluster based on the LSTM prediction result. Then, in line 3, we calculate the network traffic size net_i between S_{candi} and C_{candi}. Lines 4–15 present a nested for loop. Specifically, in line 6, we use the Dijkstra algorithm to calculate the minimum number of hops of the corresponding elements in sets S and C, and then we calculate the migration cost in line 7 and use algorithm 2 to obtain the migrating switch set. Lines 10–12 calculate the network throughput and controller resource utilization of each candidate controller; we use these two parameters

to get controller load weight $W_{C_{candi}}$ and controller overload state $State_{C_{candi}}$. Line 13 uses algorithm 1 to get origin target controller set C_{ov} and target controller set C_{tar}. Line 14 calculates the average remaining resource utilization of each controller in C_{tar} and sorts controllers in C_{tar} in descending order so that the controller with the most remaining resources will be served first. Lines 17–20 contain a for loop to match each element in S_{mir} and C_{tar} to facilitate Migration Action. The result of the switch migration algorithm is a series of migration actions.

We created a thread pool specifically for the switch migration algorithm to perform multiple switch migration actions in parallel and thus handle multiple overloaded controllers. At each execution of a migration action, the algorithm creates a new thread task in the thread pool. Parallel processing of multiple overloaded controllers reduces the migration implementation time and improves migration efficiency.

4.4 Load balance algorithm

Since the load balance algorithm has to continuously detect the network state, it is an infinite loop.

Algorithm 4: Load balance algorithm

```
1  while true do
2      if first deployment then
3          initialize network and calculate network parameters;
4          Algo1() ← (C_ov, C_tar);
5          Algo2() ← S_mir;
6      else
7          Algo1() ← (C_ov, C_tar);
8          Algo2() ← S_mir;
9      end
10     if S_mir != null && (C_ov, C_tar) != null then
11         Algo3(C_ori, C_tar, S_mir);
12     end
13 end
```

Lines 2–9 of the algorithm present a condition. If the cluster is started for the first time, then it is necessary to initialize the relevant network and calculate the network parameters; these parameters are subsequently passed to Algorithms 1 and 2 for the selection of the target controllers and migration switches. However, if the cluster has already started, then Algorithms 1 and 2 are called directly. Lines 10–12 determine whether the set returned by Algorithms 1 and 2 is empty. If it is not empty, then the cluster load is not balanced and the switch migration algorithm needs to be called to make the cluster load-balanced; if it is empty, then we continue monitoring the cluster.

5 Performance Evaluation

In this section, we describe the experimental environment and obtained results. In subsection A, we describe the controller model and topology used in the system, the network environment parameters, etc. In subsection B, we describe the experimental results in detail.

5.1 Environment Settings

We choose open source controllers ONOS1.13.0 provided by the ONOS platform [19] to build the cluster environment needed for the experiment. We use OpenFlow1.3 [20] as the communications protocol. Since there are five controllers in each topology, the controller set is $C = \{C_1, C_2, C_3, C_4, C_5\}$; moreover, there are five controllers in five virtual machines on one server. We use mininet [21] to build the network topology. The details of our system are shown in Table 2.

Table 2 System details

System version	Application deployment	IP address
Ubuntu18.04	ONOS1, ofp_sniffer	172.20.249.201
	ONOS2, ofp_sniffer	172.20.249.202
	ONOS3, ofp_sniffer	172.20.249.203
	ONOS4, ofp_sniffer	172.20.249.204
	ONOS5, ofp_sniffer	172.20.249.205
	Mininet	172.20.249.206

5.1.1 Selection of topology

We use the widely adopted Internet2 OS3E [22] topology (Figure 5) to evaluate the performance of the LPSM strategy. OS3E is a topology with a high degree of simulation. It is the abstractions of actual backbone networks in the U.S., created jointly by several universities, research institutes, and companies for the construction of the next-generation Internet, and have a high degree of authority and recognition. The data information of nodes, links, and distances in OS3E is set according to the parameters of their referenced real networks. These specific parameters are given in Table 3. Before starting the experiment, we use the `kmeans++` algorithm to place the five controllers in positions that make the cluster load-balanced.

Table 3 Topology details

Topology	Number of nodes	Number of links
OS3E	34	42

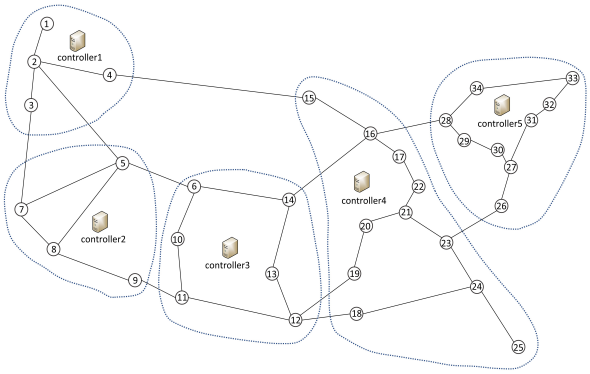


Fig. 5 OS3E topology

5.1.2 Controller performance

Figure 6 shows the *Packet_Out* response latency test result of ONOS1.13.0 with cbench [23], where the maximum, minimum, and average delays of the cluster controller are kept at a stable level for *Packet_In* rates below 1000 packets/s. After exceeding 1000 packets/s, delays start to increase. By this token, we combine the parameters mentioned in chapter 3 with Equation (3); here, we set $\delta = 1000$.

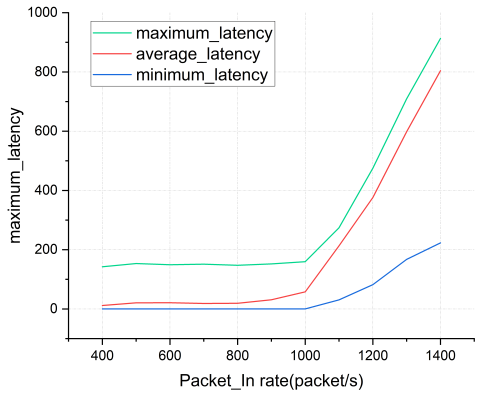


Fig. 6 Packet_Out response latency

In cluster mode, we test the controller CPU utilization percentage and physical memory usage of five ONOS controllers with different numbers of switches connected. We use the data set in [24]. From the test results, it can be concluded that as the number of switches increases, the controller CPU utilization percentage and physical memory usage increases as well, which is why we need to take these two factors into account in Equation (8).

5.2 Experimental Results

5.2.1 Switch migration cost

In the switch migration consumption experiments, we add a distribution decision mechanism (DDM) [25] and TSSM [18] for comparison since there is no migration action in the native OpenFlow environment. The core idea of the TSSM strategy is to share the load of the overloaded controller with the underloaded controller to achieve load balancing. The DDM strategy uses migration decision fields to choose migrating switches according to probability. The difference between DDM and TSSM is that DDM uses a traditional switch migration method that aims to select the best migration tuple (switch, controller). This tuple is single, i.e., performing only one migration operation at a time and not taking other influencing factors into account. Meanwhile, TSSM reduces the probability of a switch migrating back and forth between two controllers.

We evaluate the migration consumption of the best-fit migration (BFM) strategy, DDM strategy, TSSM strategy, and the LPSM strategy based on Equation (6). After many trials and parameter settings, we find that $cost_{mir}$ and Δf have a larger effect on parameter Γ_i compared with d_i. Therefore, we set $\xi_1, \xi_2 = 0.4$, and $\xi_3 = 0.2$. We also compare the average time $\sum \frac{\Gamma_i}{n}$ resource consumption of all of the migration actions, where n represents the number of migration actions to be executed. The higher the output value, the higher the cost to perform the migration action.

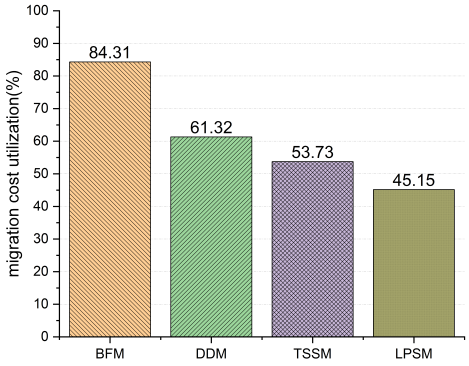


Fig. 7 Switch migration cost

From Figure 7, we can see that the BFM strategy has the highest time consumption for switch migration actions done to achieve load balancing. This is because this strategy only performs switch migration based on the current point in time and the operation between the migration actions is serial, which greatly increases the time consumption. Although the migration cost footprint of DDM is somewhat smaller than

that of BFM, DDM needs to collect network information and build a migration decision field, and then perform a load balance decision, which takes up many system resources. Meanwhile, TSSM reduces migration costs by approximately 8% compared with DDM. TSSM allows up to two controllers to participate in the time-sharing migration of a switch, which increases the flexibility of time-sharing migration but can be significantly more costly. Lastly, LPSM has a target controller selection algorithm, which relaxes migration restrictions. Multi-objective optimization is used for the selection of migration switches and target controllers, which makes this strategy better than TSSM.

5.2.2 Cluster load balancing resource efficiency

In experiments conducted to assess the average resource utilization in cluster load balancing, we randomly simulate some controllers to become overloaded and use the improved python script `iperf` command to push random bandwidth data streams into the cluster network to record the changes in controller resource utilization across the cluster over a period of time. Finally, we evaluate the controller resource utilization by computing ρ_{C_i}. We calculate the load variance of each strategy based on the load balancing result. We conclude that the load variance of BFM is 0.05318, the load variance of DDM is 0.02274, the load variance of TSSM is 0.000855, and the load variance of LPSM is 0.00460. Since BFM does not consider the impact of CPU utilization, memory utilization, and network throughput, etc., and simply performs a best-fit strategy, it has the worst resource utilization. DDM uses selection probabilities to select migrating switches and takes the controller resources occupied by the switch and the number of hops between switch controllers into account. In the selection of controllers, DDM only considers network aspects and does not consider memory and CPU utilization.

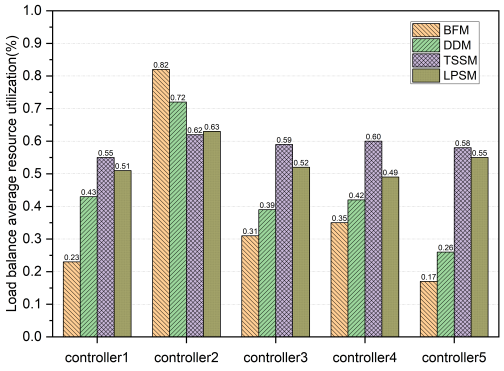


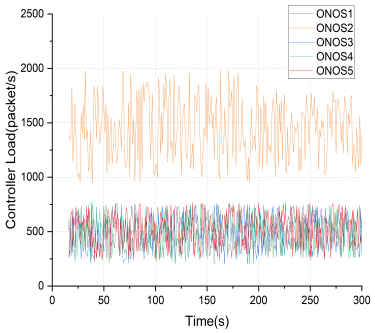
Fig. 8 Average resource utilization during load balancing

When the network traffic beccvomes larger, the memory and CPU utilization of the cluster will gradually increase. This leads to DDM having better performance but also

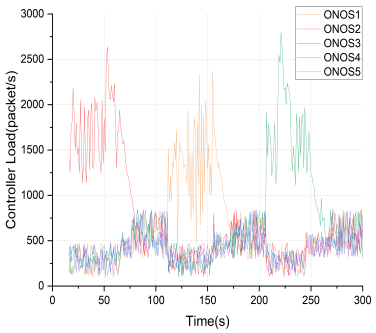
a larger load variance compared with BFM. Thanks to time-sharing migration, TSSM allows two controllers to work together to process OpenFlow messages from high-load switches, thus further balancing their workloads. Although the migration consumption of TSSM is higher than that of LPSM, its average resource usage is slightly lower. LPSM requires parallel processing of migrating switch tuples and consumes more system resources than TSSM. However, from Figure 8 we can see that the average resource consumption of LSPM is similar to that of TSSM. Moreover, compared with DDM, the resource usage of the five controllers in the LPSM strategy fluctuates less. Therefore, we can conclude that LSPM greatly reduces the load variance of system resource utilization compared with DDM.

5.2.3 Overload handling time

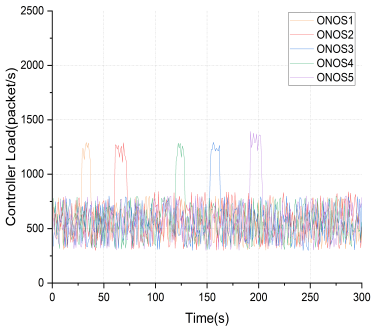
Since controller overload affects the entire multi-controller network, we compare LPSM with the OpenFlow native strategy and TSSM strategy. The results of this comparison are shown in Figure 9.



(a) OpenFlow strategy



(b) TSSM strategy



(c) LPSM strategy

Fig. 9 Controller performance in different strategies

As can be seen from Figure 9(a), the native OpenFlow strategy does not have a load balancing strategy or switch migration strategy. In our experiments, we simulate controller 2 to become overloaded and ensure that the load on controller 2 remains above threshold δ throughout the experiment. Compared with the native OpenFlow strategy, the TSSM strategy has a load balancing strategy but does not handle the overloaded controllers quickly. We successively send random traffic exceeding the threshold to controller 2, controller 1, and controller 4 to test how long it would take the TSSM strategy to handle the overloaded controllers and complete switch migration.

From Figure 9(b), we can see that the controller overload time in the cluster is around 50 seconds. When the load balancing strategy is executed, the load of the overloaded controllers will reach a balanced state only after a period of slow decline of the controller load of the whole cluster; this occurs slowly because of migration consumption, network throughput, and other factors not being considered. The LPSM strategy uses the LSTM load prediction algorithm, and after our tests on the ONOS cluster, we conclude that the CPU utilization, memory consumption, and controller processing time of other network events are not greatly affected within the first 10 seconds of the controllers being overloaded. Therefore, we set the overload time in the LSTM load prediction algorithm to 10 seconds. In our experiments, we make all five of the controllers become overloaded to observe the performance of the LPSM strategy. As we can see from Figure 9(c), the controller overload time hardly exceeds 10 seconds. Moreover, when a controller is predicted to be overloaded, the LPSM strategy immediately executes the switch migration algorithm to avoid prolonged controller overload. Thus, the controller load is quickly restored to normal after the load balancing strategy is executed.

The following equation measures the improvement in load balancing efficiency brought by this algorithm:

$$1 - \overline{ol}/\overline{total}, \tag{13}$$

According to Equation (13), the load balancing efficiency of native OpenFLow is 0% because it has no load balancing strategy. Meanwhile, the efficiency of the TSSM strategy is 60.33%, and that of the LPSM strategy is 87.93%. It is apparent that the improvement in load balancing efficiency obtained by the LPSM strategy is huge.

6 Conclusion

In this paper, we study the SDN controller cluster load balancing problem and the switch migration problem. To solve these two problems, we propose an LPSM strategy based on a deep learning LSTM algorithm, which focuses on predicting the load of each controller in a cluster and making an early decision on switch migration when controller overload is predicted. We conduct extensive simulations and experiments on the LPSM strategy, and the experimental results show that it reduces migration cost by 16% and 8%, respectively, compared with the TSSM and DDM strategies. Furthermore, it reduces load variance from 0.02 to 0.004 compared with the DDM strategy. Moreover, compared with the TSSM strategy, LPSM has a 27.6% improvement in load balancing efficiency.

However, compared with TSSM, LPSM does not have a lower average resource utilization. The reason for this is that our algorithm is too complicated for the selection of switches and controllers, and thus the algorithm requires more CPU processes and memory space. In future work, we will try to add reinforcement learning models to solve the problem of high time complexity of LPSM.

Acknowledgments. We thank LetPub (www.letpub.com) for its linguistic assistance during the preparation of this manuscript.

Data Availability. Data sharing not applicable to this article as no datasets were generated or analysed during the current study.

Funding. This work is partially supported by China University Industry-Academia-Research Innovation Fund under Grant 2021FNA02007,v partially supported by the Natural Science Foundation of Shandong Province under Grant ZR2020MF005, ZR2020MF006 and ZR2022LZH015, partially supported by the Industry-university Research Innovation Foundation of Ministry of Education of China under Grant 2021FNA01001.

Declarations

Conflict of interest. Author declares that they have no conflict of interest.

Ethics approval. Not applicable.

Consent to participate. All authors agreed to participate.

Consent for publication. Not applicable.

References

[1] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74, 2008.

[2] A TOOTOOCIAN. A distributed control plane for openflow. In *Proc. NSDI Internet Network Management Workshop/Workshop on Research on Enterprise Networking (INM/WREN), 2010*, 2010.

[3] Shuanglong Pang, Xiaodan Chen, and Desheng Zeng. Research on dynamic load balancing of data center network based on sdn architecture. In *2021 International Conference on Networking, Communications and Information Technology (NetCIT)*, pages 216–219, 2021.

[4] Irengbam Tilokchan Singh, Thounaojam Rupachandra Singh, and Tejmani Sinam. Server load balancing with round robin technique in sdn. In *2022 International Conference on Decision Aid Sciences and Applications (DASA)*, pages 503–505, 2022.

[5] P. Varalakshmi, Mithesh A, Niveditha B, Rubak Preyan G, and Yogeeswar S. Intelligent load balancing in sdn. In *2022 8th International Conference on Advanced Computing and Communication Systems (ICACCS)*, volume 1, pages 1146–1151, 2022.

[6] Thaeer Ghyadh Thajeel and Aladdin Abdulhassan. A hybrid load balancing scheme for software defined networking. In *2021 2nd Information Technology To Enhance e-learning and Other Application (IT-ELA)*, pages 106–112, 2021.

[7] Yang Wang, Ruichun Liu, Yutai Li, Zier Chen, Ning Zhang, and Bingyang Han. Sdn controller network load balancing approach for cloud computing data center. In *2022 14th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA)*, pages 242–246, 2022.

[8] Javesh Dafda and Mansi Subhedar. Dynamic load balancing in sdn using energy aware routing and optimization algorithm. In *2022 IEEE Bombay Section Signature Conference (IBSSC)*, pages 1–6, 2022.

[9] Wei Jun and Su Xiaowei. Research on sdn load balancing of ant colony optimization algorithm based on computer big data technology. In *2022 IEEE International Conference on Advances in Electrical Engineering and Computer Applications (AEECA)*, pages 935–938, 2022.

[10] Xiaoyong Sun and Guiqin Yang. Research on load balancing strategy of data center based on yen algorithm in sdn. In *2022 IEEE 5th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, volume 5, pages 1243–1247, 2022.

[11] Yong Liu, Huaxi Gu, Fulong Yan, and Nicola Calabretta. Highly-efficient switch migration for controller load balancing in elastic optical inter-datacenter networks. *IEEE Journal on Selected Areas in Communications*, 39(9):2748–2761, 2021.

[12] Gengbiao Yue, Yumei Wang, and Yu Liu. Rule placement and switch migration-based scheme for controller load balancing in sdn. In *2022 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, 2022.

[13] Lin Yao, Jia Li, Guowei Wu, and Bin Wu. New dynamic switch migration technique based on deep q-learning. In *2021 IEEE 19th International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 125–130, 2021.

[14] Himanshi Babbar, Shalli Rani, Ali Kashif Bashir, and Raheel Nawaz. Lbsmt: Load balancing switch migration algorithm for cooperative communication intelligent transportation systems. *IEEE Transactions on Green Communications and Networking*, 6(3):1386–1395, 2022.

[15] Abderrahime Filali, Soumaya Cherkaoui, and Abdellatif Kobbane. Prediction-based switch migration scheduling for sdn load balancing. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, pages 1–6, 2019.

[16] Jie Cui, Qinghe Lu, Hong Zhong, Miaomiao Tian, and Lu Liu. A load-balancing mechanism for distributed sdn control plane using response time. *IEEE transactions on network and service management*, 15(4):1197–1206, 2018.

[17] Kshira Sagar Sahoo, Deepak Puthal, Mayank Tiwary, Muhammad Usman, Bibhudatta Sahoo, Zhenyu Wen, Biswa PS Sahoo, and Rajiv Ranjan. Esmlb: Efficient switch migration-based load balancing for multicontroller sdn in iot. *IEEE Internet of Things Journal*, 7(7):5852–5860, 2019.

[18] Wei-Kuang Lai, You-Chiun Wang, Yi-Chien Chen, and Zong-Ting Tsai. Tssm: Time-sharing switch migration to balance loads of distributed sdn controllers. *IEEE Transactions on Network and Service Management*, 19(2):1585–1597, 2022.

[19] ONOS. Open operating network system. [OL]. https://opennetworking.org/onos/ Accessed 2022.

[20] OpenFlow. Openflow. [OL]. https://opennetworking.org/ Accessed 2022.

[21] mininet. Mininet. [OL]. http://mininet.org/.

[22] Internet 2. Internet 2. [OL]. http://www.internet2.edu/ Accessed 2022.

[23] Cbench. Controller bench mark. [OL]. git://gitosis.stanford.edu/oflops.git Accessed 2022.

[24] sdntopo.org. Knowledge-defined networking training datasets. [OL]. https://knowledgedefinednetworking.org/ Accessed 2022.

[25] Tao Hu, Peng Yi, Jianhui Zhang, and Julong Lan. A distributed decision mechanism for controller load balancing based on switch migration in sdn. *China Communications*, 15(10):129–142, 2018.