TSSM: Time-Sharing Switch Migration to Balance Loads of Distributed SDN Controllers

Wei-Kuang Lai, You-Chiun Wang, Yi-Chien Chen, and Zong-Ting Tsai

Abstract—*Software-defined networking (SDN)* makes network management easier by using a controller to govern all switches, but the controller may become a performance bottleneck. *Distributed SDN control* is a promising solution, which lets multiple controllers divide the work, where each controller manages a part of the network. *Switch migration* is one common means to the load balance of controllers, which transfers some switches to different subnets based on the workloads of their controllers. The paper proposes a *time-sharing switch migration (TSSM)* scheme to provide more refined load sharing for controllers, which allows two controllers to share a switch's load sequentially in the same period. When a controller is overloaded, TSSM finds assistant controllers to share its workload by selecting proper switches for migration and also deciding the time to perform migration. In this way, the workload of each controller can be kept below a given threshold. We implement the TSSM scheme on the *open network operating system (ONOS)* to attest to its feasibility. Experimental results show that TSSM can reduce 98% of the occurrences of overload for controllers as compared with the original OpenFlow method. Moreover, TSSM can save about 78% of the migration cost than the churn-triggered migration method.

Index Terms—load balance, ONOS, switch migration, SDN, time-sharing.

1 Introduction

The rapid growth of traffic demands and also diversified network applications bring an extraordinary challenge to network management. In conventional network architectures, the *control plane* (i.e., management and decision making) and the *data plane* (i.e., packet processing) are coupled up in every switch. Consequently, it costs much effort to apply new policies or algorithms in large networks, because administrators have to reconfigure involved switches one by one [1].

The *software-defined networking (SDN)* technique provides another view of network management by displacing the control plane from switches to a central entity called the *controller*. In this way, administrators can easily wield switches and monitor the network status. In particular, they can write programs on the controller to apply their policies or algorithms. Then, the controller sets rules into the switches spontaneously to carry out the policies or algorithms [2]. The controller can also query each switch about its state by the OpenFlow protocol, like the number of packets processed and their types [3]. SDN bears a variety of applications, for example, identifying malicious access points [4], managing data centers [5], providing anonymous authentication [6], and resisting cyber-attacks [7].

If a switch gets some packets without corresponding rules to handle, the switch has to ask the controller for the instructions. It can be expected that the controller of a large network with many switches will easily become the performance bottleneck. One promising solution is the *distributed SDN control (DSC)* [8], which allows multiple controllers cooperating to coordinate the network. More concretely, each controller manages a subset of switches in the network, which we call a "subnet" for short below. In addition, controllers can exchange information with each other for the purpose of collaboration.

The authors are with the Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung 80424, Taiwan (e-mail: wklai@cse.nsysu.edu.tw; ycwang@cse.nsysu.edu.tw; chenyichientw@gmail.com; e0989369793@gmail.com).

One may allocate the subnet of which each controller takes charge to fairly distribute the workload to them (also known as *controller placement*) [9]. Some dynamical placement methods are proposed, which regularly check each subnet and reassign its switches when necessary. The work [10] groups controllers such that the loads between groups are balanced. In a group, a master controller copes with the reassignment of switches for member controllers. In [11], the deep reinforcement learning technique is applied to controller placement, which considers flow fluctuation, data latency, and load balance. However, these methods may substantially change member switches in a subnet, making the subnet unstable. Besides, its controller would incur a high message overhead to perform the reassignment of switches (referring to Section 2.2 for details). Even worse, impulse or distributed denial-of-service (DDoS) flows usually generate numerous packets but last for a short while [12]. As a result, some controllers are assigned with just a few switches that handle such short-lasting flows. After the flows disappear, these controllers would become almost idle, which causes load imbalance (until switch reassignment is performed again).

Switch migration supports a more fine-grained adjustment of subnets with a smaller period and mitigates the above problem. In each period, a migration method checks if some controllers become busy and some other controllers can help share their workloads. If so, the migration method then transfers a switch from the subnet of a busy controller to another subnet whose controller is at a light load, until there is no busy controller or no available light-load controller. As will be discussed later in Section 3, the existing methods of switch migration regard one single switch as the smallest slice of migration. After a switch migrates, it stays in the new subnet until the switch is selected for migration in a subsequent period.

However, when a flow carries many packets (e.g., elephant flow [13]), a switch s_k that processes the flow brings heavy loads to its controller c_i. After s_k migrates to another subnet

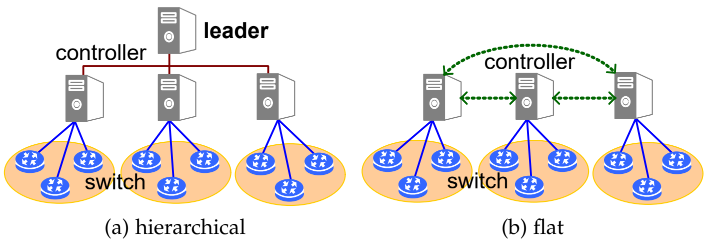


Fig. 1: Two paradigms for the DSC architecture.

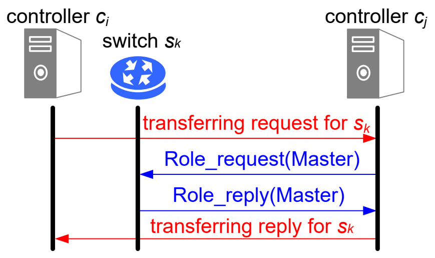


Fig. 2: The transferring procedure for switches defined in OpenFlow.

managed by a controller c_j, s_k will drastically increase c_j's load. On the other hand, c_i's load is substantially decreased as s_k leaves its subnet. In the next period, c_j may ask c_i to take over s_k again. Thus, s_k repetitively migrates between the two subnets managed by c_i and c_j, but either c_i or c_j is still overloaded. Such a phenomenon is called the *controller ping-pong difficulty*, which brings out the deficiency of the current switch migration methods. In Section 4.2, we will give an example to further explain this difficulty.

In this paper, we propose a *time-sharing switch migration (TSSM)* scheme, which lets two controllers share the load of a switch in the same period by splitting the switch's work with them sequentially. More concretely, TSSM pairs an *overloaded* controller c_i (whose workload overtakes the threshold) and an *assistant* controller c_j (which is capable of sharing the load of others). Afterward, it chooses a switch s_k from c_i's subnet to migrate. Instead of relocating s_k at the beginning of the period inflexibly, TSSM will find an adequate point in time within the period to transfer s_k to c_j's subnet. In this way, s_k's work can be appropriately shared by both c_i and c_j, thereby conquering the controller ping-pong difficulty.

Our contributions are threefold. First, unlike previous migration methods that always ask a controller to take over one switch for a whole period, this paper proposes another primal point of view to accomplish switch migration in a time-sharing manner, which provides far more flexibility. Second, the design of the TSSM scheme prevents switches from changing their subnets too frequently. Thus, TSSM's migration cost is contained, and controllers need not exchange many messages with switches to perform the migration. Third, we implement the TSSM scheme on the *open network operating system (ONOS)* [14] to attest to its feasibility. Experimental results show that TSSM can significantly reduce the occurrences of overload for controllers, efficiently balance the workloads of all controllers, and keep a low migration cost.

This paper is organized as follows: Section 2 gives background knowledge and Section 3 surveys related work. The system model is discussed in Section 4. After that, Section 5 details the TSSM scheme, followed by performance evaluation in Section 6. Finally, Section 7 concludes this paper.

2 Preliminary

In this section, we discuss the DSC architecture, the transferring procedure for switches proposed in OpenFlow, and the ONOS platform for controller implementation.

2.1 DSC Architecture

There are two common paradigms for the DSC architecture [8]. In the *hierarchical* paradigm, a "leader" is responsible for coordinating all other controllers, as shown in Fig. 1(a). Except for the leader, each controller directs a subset of switches and reports its status to the leader. Because the leader has a global view of the network, it is easy to apply network-wide policies through the leader. However, once the leader is broken down, a new leader should be elected [15]. On the other hand, in the *flat* paradigm, controllers have their respective authorization of subnets, as Fig. 1(b) shows. Since each controller has only a local view of the network, when inter-controller functions are required, the involved controllers need to carry out message exchanges of local views in a distributed manner.

In this paper, we adopt the hierarchical paradigm. Specifically, the leader keeps monitoring the status of each controller. When some controllers become overloaded, the leader executes the TSSM scheme to select assistant controllers to share their workloads and schedule the duration where a controller should take over each selected switch. Then, the leader notifies the involved controllers of the result of switch migration.

2.2 Transferring Procedure for Switches in OpenFlow

To realize the transfer of switches among different subnets, OpenFlow permits a switch s_k establishing relationships with multiple controllers. Based on s_k's perspective, each associated controller c_i can have one of the following roles:

- `OFPCR_ROLE_EQUAL` (Equal): This default role lets c_i have full access to s_k and be equal to other controllers in the same role. Specifically, c_i can send commands to s_k and also obtain its status report.
- `OFPCR_ROLE_SLAVE` (Slave): If the role of c_i changes to Slave, the access control of c_i on s_k is read-only. Thus, c_i can no longer send commands to s_k.
- `OFPCR_ROLE_MASTER` (Master): Similar to the Equal role, c_i also has full access to s_k. However, each switch can have at most one Master controller. In this case, the roles of other controllers will be switched to Slave.

Fig. 2 shows the transferring procedure for switches defined in OpenFlow. Since the Master role is exclusive, this procedure must be initiated by a Master controller. Suppose that c_i and c_j are the Master and target controllers of switch s_k, respectively. In the first place, c_i sends to c_j the transferring request for s_k. Afterward, c_j asks s_k to change c_j's role to be s_k's Master via a `Role_request(Master)` message, and s_k makes a confirmation by answering c_j with a `Role_reply(Master)` message. Finally, c_j notifies c_i of the successful transfer for s_k, and c_i's role (in terms of s_k) will be altered to Slave.

Switch migration can be accomplished by employing the transferring procedure in Fig. 2, which is supported by OpenFlow with versions 1.2, 1.3, 1.4, and 1.5 (the latest version). The OpenFlow standard only indicates how to exchange messages between controllers and switches to alter the roles of involved

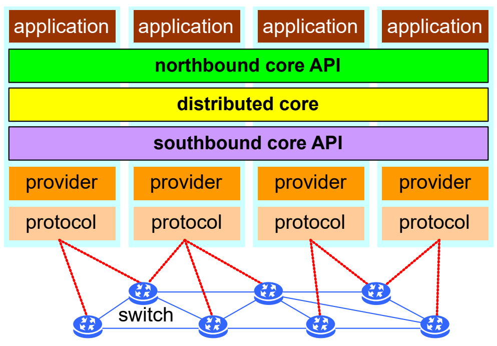


Fig. 3: The software framework of ONOS.

controllers. However, the selection of target controllers and switches for transfer is not specified in the standard. That is why different switch migration methods are proposed (as discussed in Section 3). Our TSSM scheme finds appropriate controllers and decides when to perform switch migration by using the procedure in Fig. 2. Thus, TSSM can function well in all the above versions of OpenFlow. In this paper, OpenFlow version 1.5 is adopted to implement TSSM.

2.3 ONOS Platform

ONOS is a popular open-source platform that supports hosts with the complete SDN functions, including the control plane, switches, links, and communication services. The kernel, core services, and applications of ONOS are all written in Java as bundles to be loaded into the OSGi container, which is a Java component system that allows function modules to be installed and run dynamically in a single Java virtual machine. In this way, ONOS can run on different underlying operating systems.

Fig. 3 gives the software framework of ONOS. Specifically, the *northbound core API* provides the network's information (e.g., topology and paths) based on the representational state transfer (REST) style [16]. On the other hand, the *distributed core* is responsible for computation, status management, and notification. The *southbound core API* helps an ONOS controller interact with switches. It communicates with switches through *providers*, which can use different *protocols* such as OpenFlow, P4Runtime, NETCONF, and OVSDB[1].

Generally speaking, the TSSM scheme can be implemented on any SDN controller platform compliant to OpenFlow, such as ONOS, NOX, Floodlight, and OpenDaylight. Considering that ONOS provides a useful platform to facilitate programmers to develop their SDN applications, we adopt the ONOS platform to implement TSSM and attest to its feasibility.

3 Related Work

Various issues for DSC have been addressed in the literature. Chan et al. [17] discuss how to smoothly hand over the work of an impaired controller to another controller to minimize the service interruption time. The work [15] picks a controller as the leader to coordinate other controllers. Once the leader fails, a controller whose load is low and throughput is high will be the new leader. Lu et al. [9] survey the existing solutions to the controller placement problem, whose objective is to keep fairness between controllers in terms of workloads. A reliable deployment approach for controllers is proposed in [18] to improve network stability and reduce packet loss. Kim et al. [19] increase the throughput of distributed datastore in an OpenDaylight controller cluster by evenly distributing the shard leaders to cluster members. In [20], when some switches become busy, controllers cooperate to reroute their traffic flows to mitigate congestion. The work [21] proposes a software-defined cyber foraging framework, which is a hybrid controller including a control plane for local networks and cloudlets. In [22], layer-2 controllers predict the network's load, and layer-3 controllers perform device migration based on the prediction.

Except for the dynamical controller placement methods [10], [11] discussed in Section 1, a number of studies aim to balance loads of distributed SDN controllers, which can be classified into three categories.

Switch migration: These methods make a busy controller transfer the control of some switches to low-load controllers to reduce its workload. The study [23] performs switch migration when the CPU and memory utilization of a controller exceeds the threshold, but how to choose target controllers for switch migration is not addressed. The work [24] employs Q-learning to relocate switches, which minimizes the standard deviation of the workloads of controllers. Cui et al. [25] execute switch migration based on the response time of each controller. They pick the controller with the longest response time and transfer its switch with the largest load to a controller with the shortest response time. In [26], the criteria to select the target controller for switch migration include its memory size, CPU utilization, bandwidth, and hop count. Hu et al. [27] discuss the cost of switch migration, and propose a simulated-annealing algorithm to select target controllers to reduce the migration cost.

Flow migration: Instead of transferring a whole switch, these methods carry out the migration at the granularity of the flows. In [28], one super controller takes charge of monitoring the status of every controller and adjusting the flow requests handled by them. The work [29] allows switches to detect the workload of each controller and choose their target controllers for flow management. A migration method based on the game theory is proposed to improve network performance. However, to let a switch select its favored controller, custom rules for the control plane must be installed in its memory, but this feature is not supported by OpenFlow.

Flow splitting: In this category of methods, a switch is supervised by more than one controller simultaneously. Gorkemli et al. [30] use a virtual overlay on the data plane for switches to negotiate with their controllers to realize flow splitting. The study [31] maps each switch to multiple controllers for sharing the requests of flow setup, which saves the time for the switch to handle the flows. A convex quadratic programming problem is defined in [32] to model the mapping between switches and controllers, which attempts to balance the loads of controllers and also reduce new switch-controller assignments. However, if multiple controllers have full access to a switch, we have to guarantee the *synchronization* among these controllers in terms of the switch (for example, two controllers cannot install rules into the switch at the same time). This issue would complicate the design of a flow-splitting method.

Table 1 gives a comparison between the prior load-balancing methods and our TSSM scheme. As discussed in Section 2.2, OpenFlow supports the transfer of a switch's control from a controller to another. Dynamical controller placement

1. API: application program interface, REST: representational state transfer, NETCONF: network configuration protocol, OVSDB: open vSwitch database.

TABLE 1: Comparison between the prior load-balancing methods and our TSSM scheme.

method	category*	OpenFlow compliant	time sharing	controller implementation
work [10]	DCP	✓		n/a
work [11]	DCP	✓		n/a
work [23]	SM	✓		n/a
work [24]	SM	✓		NOX
work [25]	SM	✓		Floodlight
work [26]	SM	✓		Floodlight
work [27]	SM	✓		OpenDaylight
work [28]	FM			n/a
work [29]	FM			n/a
work [30]	FS			n/a
work [31]	FS			n/a
work [32]	FS			n/a
TSSM	SM	✓	✓	ONOS

*DCP: dynamical controller placement, SM: switch migration, FM: flow migration, FS: flow splitting

methods and switch migration methods (including TSSM) are therefore OpenFlow-compliant. Among all studies mentioned in Table 1, only those methods proposed in [24], [25], [26], [27] and our TSSM scheme are implemented on real controller platforms. TSSM is peculiarly implemented on ONOS to show its practicability. To the best of our knowledge, TSSM is the first work to carry out switch migration in a time-sharing manner, which provides more refined and flexible load sharing for controllers.

4 System Model

This section presents the network model, gives an example to show the controller ping-pong difficulty, and then formulates the switch migration problem.

4.1 Network Model

Let us consider an SDN-based network composed of a set $\hat{\mathcal{S}}$ of switches and a set $\hat{\mathcal{C}}$ of controllers. For each controller $c_i \in \hat{\mathcal{C}}$, any switch in $\hat{\mathcal{S}}$ is *administrable*, which means that c_i is capable of acting as the switch's Master controller. On the other hand, each switch $s_k \in \hat{\mathcal{S}}$ has one Master controller, which would change due to migration. Let $\hat{\mathcal{S}}_i \subseteq \hat{\mathcal{S}}$ be the set of switches managed by c_i. Then, we have $\bigcup_{c_i \in \hat{\mathcal{C}}} \hat{\mathcal{S}}_i = \hat{\mathcal{S}}$ and $\hat{\mathcal{S}}_i \cap \hat{\mathcal{S}}_j = \emptyset$ for any two controllers c_i and c_j in $\hat{\mathcal{C}}$.

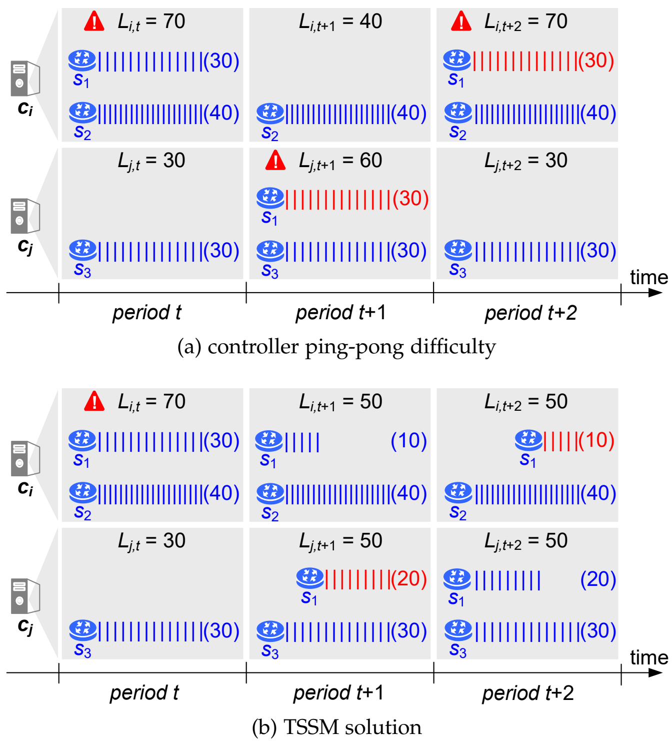
Many studies [23]–[26] point out that processing the *Packet_In messages (PIMs)* sent from switches accounts for most of a controller's workload. Because of this, we adopt the number of PIMs to evaluate the load generated by a switch. Specifically, the time axis is divided into fixed-length periods. Let $\zeta_{k,t}$ be the number of PIMs produced by a switch $s_k \in \hat{\mathcal{S}}$ in period t. Since s_k may migrate to the subnet of a different controller at any time in a period, we denote by $\zeta^{(i)}_{k,t}$ the number of PIMs that s_k submits to its Master controller c_i in period t. Therefore, we have $\sum_{\forall c_i \in \hat{\mathcal{C}}} \zeta^{(i)}_{k,t} = \zeta_{k,t}$. The capacity Γ_i of a controller c_i is defined by the maximum number of PIMs that it can process in a period. Furthermore, let $L_{i,t}$ be the *amount of workload of controller c_i in period t* (below, we call it "c_i's workload" for short), which is calculated by

$$L_{i,t} = \sum\nolimits_{\forall s_k \in \hat{\mathcal{S}}_i} \zeta^{(i)}_{k,t}. \tag{1}$$

Thence, the workload of c_i is the sum of PIMs that each switch in c_i's subnet sends to c_i during period t.

If $L_{i,t} > \delta_i$, c_i is viewed as overloaded in period t, where $\delta_i \leq \Gamma_i$ is a threshold. In particular, we adopt δ_i instead of Γ_i due to two reasons. First, doing so provides flexibility, as δ_i can be adjusted based on the application demand. Second, each controller can reserve a (small) portion of its processing power to cope with additional PIMs (e.g., abruptly generated by switches within a period).

Similar to the existing solutions, parameters $\zeta_{k,t}$ and $L_{i,t}$ are given. Since the hierarchical paradigm of DSC architecture is adopted (as mentioned in Section 2.1), there exists a leader controller to collect these parameters from all other controllers in each period. Then, the leader directs the decision result of switch migration (based on the proposed scheme) to them.



(a) controller ping-pong difficulty

(b) TSSM solution

Fig. 4: An example to illustrate the controller ping-pong difficulty and how TSSM solves this difficulty.

4.2 Controller Ping-Pong Difficulty

We use an example in Fig. 4 to explain the controller ping-pong difficulty. Suppose that there are two controllers, $\hat{\mathcal{C}} = \{c_i, c_j\}$, and three switches, $\hat{\mathcal{S}} = \{s_1, s_2, s_3\}$ in a network, where $\delta_i = \delta_j = 50$ PIMs. Moreover, s_1, s_2, and s_3 produce 30, 40, and 30 PIMs per period, respectively. In period t, c_i manages s_1 and s_2 (i.e., $\hat{\mathcal{S}}_i = \{s_1, s_2\}$) and c_j takes charge of s_3 (i.e., $\hat{\mathcal{S}}_j = \{s_3\}$). Because $L_{i,t} = \zeta_{1,t} + \zeta_{2,t} = 30 + 40 > \delta_i$, c_i is overloaded and thus switch migration is necessary.

In the current switch migration methods, a busy controller will request another controller to take over some of its switches for a *whole* period, as shown in Fig. 4(a). More concretely, c_i lets s_1 migrate to c_j's subnet in period $t+1$. Since $L_{j,t+1} = \zeta_{1,t+1} + \zeta_{3,t+1} = 30 + 30 > \delta_j$, c_j will be overloaded. Thus, c_j asks c_i to take over s_1 again in period $t+2$, so the situation is the same as that in period t, which causes a ping-pong effect. As can be seen, no matter how switches are transferred, either c_i or c_j will be overloaded. This is called the controller ping-pong difficulty.

On the other hand, TSSM performs switch migration in a time-sharing manner, which splits the workload from the same switch between two controllers within a period, as Fig. 4(b) shows. In period $t+1$, c_i handles the first 10 PIMs of s_1 and leaves s_1's residual 20 PIMs to c_j (by migration). Therefore, c_i's workload is $L_{i,t+1} = \zeta^{(i)}_{1,t+1} + \zeta_{2,t+1} = 10 + 40 \leq \delta_i$ and c_j's workload is $L_{j,t+1} = \zeta^{(j)}_{1,t+1} + \zeta_{3,t+1} = 20 + 30 \leq \delta_j$, so both controllers are not overloaded in period $t+1$. Similarly, in period $t+2$, c_j first processes 20 PIMs of s_1 and lets s_1 migrate to c_i's subnet for the residual work. Thus, the workloads of both controllers are below their thresholds. In this way, TSSM can efficiently conquer the controller ping-pong difficulty.

4.3 Switch Migration Problem

Suppose that the network operation time is divided into N periods. Given sets $\hat{\mathcal{C}}$ and $\hat{\mathcal{S}}$, the *switch migration problem* asks how to transfer some switches in $\hat{\mathcal{S}}$ among different subnets to balance workloads of controllers in $\hat{\mathcal{C}}$ in each period, such that the number of occurrences of overload for controllers in these N periods is minimized. In particular, let $f(c_i, t)$ denote an indicator to reveal the occurrence of overload for a controller $c_i \in \hat{\mathcal{C}}$ in period t, where $f(c_i,t) = 1$ if $L_{i,t} > \delta_i$ or $f(c_i,t) = 0$ otherwise. Then, the objective function can be expressed by

$$\text{minimize} \sum\nolimits_{t=1}^{N} \sum\nolimits_{\forall c_i \in \hat{\mathcal{C}}} f(c_i, t), \tag{2}$$

Definition 1 restates the switch migration problem as a decision problem, and Theorem 1 shows that it is NP-hard. Table 2 summarizes our notations.

Definition 1. Given the set $\hat{\mathcal{S}}_i \subseteq \hat{\mathcal{S}}$ of switches (i.e., subnet) managed by a controller c_i during period t, the *switch migration decision (SMD) problem* asks whether $L_{i,t}$ can be equal to δ_i for every controller in $\hat{\mathcal{C}}$ by transferring some switches among different subnets.

Theorem 1. The SMD problem is NP-hard.

Proof: To prove this theorem, we reduce an NP-complete problem, the *subset sum problem* [33], to the SMD problem. Given a set $\hat{\mathcal{X}} = \{x_1, x_2, \cdots, x_m\}$ of integers and one target sum λ, the subset sum problem determines whether any subset of the integers in $\hat{\mathcal{X}}$ will sum to λ. For the reduction, let us construct an instance of the SMD problem as follows: Suppose that there are two controllers c_i and c_j, where $\delta_i = \lambda$ and $\delta_j = \sum_{\forall x_k \in \hat{\mathcal{X}}} x_k - \lambda$. Moreover, let $\hat{\mathcal{S}} = \{s_1, s_2, \cdots, s_m\}$, where every switch s_k produces x_k PIMs in period t (i.e., $\zeta_{k,t} = x_k$, for $k = 1, 2, \cdots, m$). Then, we show that if there exists a solution to the subset sum problem, there must also exist a solution to the SMD instance problem, and vice versa.

Assume that the solution of the subset sum problem is $\hat{\mathcal{Y}} \subseteq \hat{\mathcal{X}}$. In this case, we can exchange some switches between $\hat{\mathcal{S}}_i$ and $\hat{\mathcal{S}}_j$ such that $\hat{\mathcal{S}}_i$ contains each switch s_k whose load $\zeta_{k,t}$ is equal to $x_k \in \hat{\mathcal{Y}}$ and $\hat{\mathcal{S}}_j$ includes all residual switches in $\hat{\mathcal{S}}$. Thus, a solution is found for the SMD instance problem.

On the other hand, suppose that the solution to the SMD instance problem is $\{\hat{\mathcal{S}}_i, \hat{\mathcal{S}}_j\}$, where $\hat{\mathcal{S}}_i \cup \hat{\mathcal{S}}_j = \hat{\mathcal{S}}$ and $\hat{\mathcal{S}}_i \cap \hat{\mathcal{S}}_j = \emptyset$. Then, for each switch s_k in $\hat{\mathcal{S}}_i$, we choose the integer x_k $(= \zeta_{k,t})$ from $\hat{\mathcal{X}}$ and add it to $\hat{\mathcal{Y}}$. Since $\sum_{\forall s_k \in \hat{\mathcal{S}}_i} \zeta_{k,t} = \delta_i = \lambda$, $\hat{\mathcal{Y}}$ must be a solution to the subset sum problem.

Apparently, the above reduction takes polynomial time, so the SMD problem is NP-hard. □

TABLE 2: Summary of notations.

notation	definition
$\hat{\mathcal{C}}, \hat{\mathcal{S}}$	the sets of all controllers and all switches in the network
$\hat{\mathcal{C}}_\mathbf{O}, \hat{\mathcal{C}}_\mathbf{A}$	the subsets of overloaded and assistant controllers in $\hat{\mathcal{C}}$
$\hat{\mathcal{S}}_i$	the set of switches managed by a controller $c_i \in \hat{\mathcal{C}}$
$\zeta^{(i)}_{k,t}$	PIMs that a switch $s_k \in \hat{\mathcal{S}}_i$ sends to c_i in period t
$\zeta_{k,t}$	total PIMs produced by s_k in period t
Γ_i	the capacity of c_i
$L_{i,t}$	the workload of c_i in period t
δ_i	If $L_{i,t} > \delta_i$, c_i is an overloaded controller. ($\delta_i \leq \Gamma_i$)
φ	If $L_{i,t} < \varphi \times \delta_i$, c_i is an assistant controller. ($0 < \varphi < 1$)
$f(c_i, t)$	an indicator to reveal whether c_i is overloaded in period t
τ	the amount of time after which s_k will migrate
q	the number of s_k's PIMs that the new controller will handle
Δ	a load threshold used in Algo. 3

Algorithm 1: TSSM Scheme

```
1  $\hat{\mathcal{C}}_\mathbf{O} \leftarrow \emptyset$ and $\hat{\mathcal{C}}_\mathbf{A} \leftarrow \emptyset$;
2  foreach $c_i \in \hat{\mathcal{C}}$ do
3      $L_{i,t} \leftarrow 0$;
4      foreach $s_k \in \hat{\mathcal{S}}_i$ do
5          $L_{i,t} \leftarrow L_{i,t} + \zeta^{(i)}_{k,t}$;
6      if $L_{i,t} > \delta_i$ then
7          $\hat{\mathcal{C}}_\mathbf{O} \leftarrow \hat{\mathcal{C}}_\mathbf{O} \cup \{c_i\}$;
8      else if $L_{i,t} < \varphi \times \delta_i$ then
9          $\hat{\mathcal{C}}_\mathbf{A} \leftarrow \hat{\mathcal{C}}_\mathbf{A} \cup \{c_i\}$;
10 if $\hat{\mathcal{C}}_\mathbf{O} \neq \emptyset$ and $\hat{\mathcal{C}}_\mathbf{A} \neq \emptyset$ then
11     Use Algo. 2 to balance loads between $\hat{\mathcal{C}}_\mathbf{O}$ and $\hat{\mathcal{C}}_\mathbf{A}$;
```

5 The Proposed TSSM Scheme

In the beginning, each switch $s_k \in \hat{\mathcal{S}}$ chooses one controller to be its Master controller, which can be done by the manual configuration or using any controller placement method [9]. In this way, the initial subnet $\hat{\mathcal{S}}_i$ of each controller $c_i \in \hat{\mathcal{C}}$ can be obtained. As discussed in Section 3, the previous solutions of controller placement and switch migration change the subnets of which switches belong merely at the beginning of a period, so the connections between switches and controllers are static in a whole period. Unlike the previous solutions, the TSSM scheme allows switches to migrate in a time-sharing manner. Thus, the switches can *dynamically* change their connections with controllers within a period (i.e., from busy controllers to light-load controllers). Consequently, TSSM not only performs switch migration more flexibly but also conquers the controller ping-pong difficulty (as mentioned in Section 4.2).

Algo. 1 presents the pseudocode of TSSM, which first finds out *overloaded* and *assistant* controllers from $\hat{\mathcal{C}}$, whose sets are denoted by $\hat{\mathcal{C}}_\mathbf{O}$ and $\hat{\mathcal{C}}_\mathbf{A}$, respectively. According to Eq. (1), the code in lines 3–5 calculates the workload $L_{i,t}$ of every controller $c_i \in \hat{\mathcal{C}}$ by summing up the loads of all switches in its subnet $\hat{\mathcal{S}}_i$. Then, we check if $L_{i,t}$ overtakes the threshold δ_i. If so, c_i is overloaded and thus it is added to $\hat{\mathcal{C}}_\mathbf{O}$ by lines 6–7. On the other hand, c_i is categorized as an *assistant* when its workload $L_{i,t}$ is below $\varphi \times \delta_i$, where φ is a coefficient whose value is less than but close to one (for example, we can set φ to 0.95), as shown in lines 8–9. The reason for using coefficient φ is to exclude those non-overloaded controllers that almost use up their resources (that is, $L_{i,t} < \delta_i$ and $L_{i,t} \approx \delta_i$). Such controllers in the end have no extra resources to help partake

Algorithm 2: Load-Balanced Migrating Module

```
1  SORT($\hat{\mathcal{C}}_{\mathbf{O}}$, $L_{i,t} - \delta_i$);
2  SORT($\hat{\mathcal{C}}_{\mathbf{A}}$, $\delta_j - L_{j,t}$);
3  foreach $c_i \in \hat{\mathcal{C}}_{\mathbf{O}}$ do
4      SORT($\hat{\mathcal{S}}_i$, $\zeta^{(i)}_{k,t}$);
5      while $L_{i,t} > \delta_i$ do
6          if $\hat{\mathcal{C}}_{\mathbf{A}} = \emptyset$ then
7              Terminate this module;
8          Pick the first controller $c_j$ from $\hat{\mathcal{C}}_{\mathbf{A}}$;
9          $(s_k, \tau, q) \leftarrow$ Algo. 3$(c_i, c_j)$;
10         Transfer $s_k$ to $c_j$'s subnet after $\tau$ units of time;
11         $L_{i,t} \leftarrow L_{i,t} - q$;
12         $L_{j,t} \leftarrow L_{j,t} + q$;
13         if $L_{j,t} \geq \varphi \times \delta_j$ then
14             $\hat{\mathcal{C}}_{\mathbf{A}} \leftarrow \hat{\mathcal{C}}_{\mathbf{A}} \setminus \{c_j\}$;
15         else
16             SORT($\hat{\mathcal{C}}_{\mathbf{A}}$, $\delta_j - L_{j,t}$);
```

the workloads of other controllers, so they will not be included in $\hat{\mathcal{C}}_{\mathbf{A}}$ to avoid unnecessary calculations and shifts.

Switch migration will be carried out only when both $\hat{\mathcal{C}}_{\mathbf{O}}$ and $\hat{\mathcal{C}}_{\mathbf{A}}$ are non-empty, and this condition is checked in line 10. If so, TSSM uses the *load-balanced migrating module* in Algo. 2 to let the controllers in $\hat{\mathcal{C}}_{\mathbf{A}}$ take over some switches managed by the controllers in $\hat{\mathcal{C}}_{\mathbf{O}}$ (i.e., sharing their workloads). Then, Lemma 1 analyzes the time complexity of Algo. 1.

Lemma 1. Given ξ_C controllers in $\hat{\mathcal{C}}$ and ξ_S switches in $\hat{\mathcal{S}}$, the time complexity of Algo. 1 is $O(\xi_C + \xi_S) + T_2$, where T_2 is the computation time of Algo. 2.

Proof: In Algo. 1, line 1 takes a constant time to initialize both $\hat{\mathcal{C}}_{\mathbf{O}}$ and $\hat{\mathcal{C}}_{\mathbf{A}}$. Then, the outer for-loop in lines 2–9 has ξ_C iterations, where lines 3, 6, 7, 8, and 9 all require $O(1)$ time. As mentioned in Section 4.1, since $\bigcup_{\forall c_i \in \hat{\mathcal{C}}} \hat{\mathcal{S}}_i$ is equal to $\hat{\mathcal{S}}$, the inner for-loop in lines 4–5 (along with the outer for-loop) checks every switch in $\hat{\mathcal{S}}_i$. Therefore, the outer for-loop takes time of $\xi_C \times O(1) + O(\xi_S) = O(\xi_C + \xi_S)$. After that, line 11 executes Algo. 2 and consumes T_2 time. To sum up, the time complexity of Algo. 1 is $O(\xi_C + \xi_S) + T_2$. □

5.1 Load-Balanced Migrating Module

Given overloaded controllers in $\hat{\mathcal{C}}_{\mathbf{O}}$ and assistant controllers in $\hat{\mathcal{C}}_{\mathbf{A}}$, this module iteratively finds a pair of controllers (c_i, c_j), where $c_i \in \hat{\mathcal{C}}_{\mathbf{O}}$ and $c_j \in \hat{\mathcal{C}}_{\mathbf{A}}$, and then requests one switch of c_i migrating to c_j's subnet to reduce c_i's workload, until $\hat{\mathcal{C}}_{\mathbf{O}}$ or $\hat{\mathcal{C}}_{\mathbf{A}}$ becomes empty. Algo. 2 shows the pseudocode of the load-balanced migrating module.

Let SORT($\hat{\mathcal{X}}$, ϵ) denote a function which sorts the elements in a set $\hat{\mathcal{X}}$ according to the measure ϵ in decreasing order. In line 1, $\hat{\mathcal{C}}_{\mathbf{O}}$ is sorted decreasingly by the amount of *overload* of controllers (i.e., $L_{i,t} - \delta_i$), so the controller with the maximum overload will be served first. On the other hand, line 2 sorts $\hat{\mathcal{C}}_{\mathbf{A}}$ decreasingly based on the amount of *remaining capacity* of controllers (i.e., $\delta_j - L_{j,t}$). Thus, the controller that has the most remaining capacity will be considered first to share the workload of an overloaded controller.

The for-loop in lines 3–16 handles each controller $c_i \in \hat{\mathcal{C}}_{\mathbf{O}}$ in turn (from the most overloaded one to the least overloaded one). Specifically, line 4 sorts the set $\hat{\mathcal{S}}_i$ of switches managed by c_i based on their PIMs sending to c_i (i.e., $\zeta^{(i)}_{k,t}$) decreasingly. The while-loop in lines 5–16 keeps diminishing c_i's workload by switch migration, until $L_{i,t}$ is below threshold δ_i. However, if $\hat{\mathcal{C}}_{\mathbf{A}}$ is empty (i.e., there is no assistant controller to help), Algo. 2 terminates, as shown in lines 6–7. Otherwise, we pick the first controller c_j from $\hat{\mathcal{C}}_{\mathbf{A}}$ to share c_i's workload. Since $\hat{\mathcal{C}}_{\mathbf{A}}$ has been already sorted by line 2, c_j is the (assistant) controller with the most remaining capacity in the network and the one who can most share the burden of other controllers.

Then, we execute the *time-to-migration calculating module* in Algo. 3 with parameters c_i and c_j. It returns a three-tuple result (s_k, τ, q), where s_k is the selected switch (originally managed by c_i) for migration, τ is the amount of time after which s_k will migrate, and q is the number of s_k's PIMs that c_j expects to process (i.e., the amount of c_i's workload that c_j will handle). Afterward, the code in lines 10–12 performs switch migration and updates the workloads of c_i and c_j (i.e., $L_{i,t}$ and $L_{j,t}$). If c_j almost uses up its remaining capacity due to the above migration (i.e., $L_{j,t} \geq \varphi \times \delta_j$, as indicated in line 13), c_j is removed from $\hat{\mathcal{C}}_{\mathbf{A}}$. Otherwise, $\hat{\mathcal{C}}_{\mathbf{A}}$ should be sorted again since c_j's workload changes, as shown in line 16.

Theorem 2 proves the convergence of Algo. 2 and Lemma 2 analyzes its time complexity.

Theorem 2. Given a finite number of controllers in $\hat{\mathcal{C}}$, Algo. 2 must converge.

Proof: Because $\hat{\mathcal{C}}_{\mathbf{O}}$ and $\hat{\mathcal{C}}_{\mathbf{A}}$ are subsets of $\hat{\mathcal{C}}$, the numbers of overloaded controllers (in $\hat{\mathcal{C}}_{\mathbf{O}}$) and assistant controllers (in $\hat{\mathcal{C}}_{\mathbf{A}}$) are finite. Moreover, since $\hat{\mathcal{C}}_{\mathbf{O}}$ and $\hat{\mathcal{C}}_{\mathbf{A}}$ are disjoint, there will be no controller in $\hat{\mathcal{C}}$ that is an overloaded controller and also an assistant controller at the same time. By the code in lines 8–12, an overloaded controller $c_i \in \hat{\mathcal{C}}_{\mathbf{O}}$ must transfer a part of its workload to an assistant controller $c_j \in \hat{\mathcal{C}}_{\mathbf{A}}$. In this way, c_i's workload $L_{i,t}$ will always decrease, while c_j's workload $L_{j,t}$ will only increase. According to lines 13 and 14, c_j is removed from $\hat{\mathcal{C}}_{\mathbf{A}}$ when its workload overtakes $\varphi \times \delta_j$, which implies that $\hat{\mathcal{C}}_{\mathbf{A}}$ can only shrink. Because $\hat{\mathcal{C}}_{\mathbf{A}}$ contains a finite number of controllers, $\hat{\mathcal{C}}_{\mathbf{A}}$ will eventually become empty, which forces Algo. 2 to terminate by lines 6–7. On the other hand, since $L_{i,t}$ will only reduce, the while-loop in lines 5–16 must be eventually terminated because of $L_{i,t} \leq \delta_i$ (line 5) or $\hat{\mathcal{C}}_{\mathbf{A}} = \emptyset$ (line 6). Since $\hat{\mathcal{C}}_{\mathbf{O}}$ has a finite number of controllers, the for-loop in lines 3–16 will never run forever. Consequently, Algo. 2 must converge, which proves this theorem. □

Lemma 2. Let $|\hat{\mathcal{C}}_{\mathbf{A}}| = \xi_A$ and $|\hat{\mathcal{S}}| = \xi_S$. Algo. 2 takes time of $\xi_S(T_3 + O(\xi_A + \log_2 \xi_S))$ in the worst case, where T_3 is the computation time of Algo. 3.

Proof: In Algo. 2, lines 1 and 2 require $O(\xi_O \log_2 \xi_O)$ and $O(\xi_A \log_2 \xi_A)$ time to sort sets $\hat{\mathcal{C}}_{\mathbf{O}}$ and $\hat{\mathcal{C}}_{\mathbf{A}}$, respectively, where $\xi_O = |\hat{\mathcal{C}}_{\mathbf{O}}|$. In the for-loop, we choose an overloaded controller c_i (i.e., line 3), select an assistant controller c_j (i.e., line 8), and transfer the load of a switch s_k from c_i to c_j (i.e., lines 9–12). The worst case occurs when all switches in $\hat{\mathcal{S}}$ are initially managed by the controllers in $\hat{\mathcal{C}}_{\mathbf{O}}$. In this case, the for-loop together with the while-loop will repeat at most ξ_S times (i.e., checking every switch in $\hat{\mathcal{S}}$). Then, let us take a look at the for-loop. Except for lines 4, 9, and 16, each of the residual statements inside the for-loop consumes $O(1)$ time. Then, line

Algorithm 3: Time-to-migration Calculating Module

```
1  $\Delta \leftarrow \min\{L_{i,t} - \delta_i, \delta_j - L_{j,t}\}$;
2  $\hat{\mathcal{S}}_i^\alpha \leftarrow \emptyset$ and $\hat{\mathcal{S}}_i^\beta \leftarrow \emptyset$;
3  foreach $s_k \in \hat{\mathcal{S}}_i$ do
4      if $\zeta_{k,t}^{(i)} \geq \Delta$ then
5          $\hat{\mathcal{S}}_i^\alpha \leftarrow \hat{\mathcal{S}}_i^\alpha \cup \{s_k\}$;
6      else
7          $\hat{\mathcal{S}}_i^\beta \leftarrow \hat{\mathcal{S}}_i^\beta \cup \{s_k\}$;
8  if $\hat{\mathcal{S}}_i^\alpha \neq \emptyset$ then
9      $s_k \leftarrow$ the last switch of $\hat{\mathcal{S}}_i^\alpha$;
10     $\tau \leftarrow [(\zeta_{k,t}^{(i)} - \Delta)/\zeta_{k,t}^{(i)}] \times P$ and $q \leftarrow \Delta$;
11 else
12     $s_k \leftarrow$ the first switch of $\hat{\mathcal{S}}_i^\beta$;
13     $\tau \leftarrow 0$ and $q \leftarrow \zeta_{k,t}^{(i)}$;
14 $\zeta_{k,t}^{(i)} \leftarrow \zeta_{k,t}^{(i)} - q$ and $\zeta_{k,t}^{(j)} \leftarrow q$;
15 return $(s_k, \tau, q)$;
```

4 takes $O(|\hat{\mathcal{S}}_i| \log_2 |\hat{\mathcal{S}}_i|)$ time to sort $\hat{\mathcal{S}}_i$. Line 9 finds a switch and the time for migration by Algo. 3, which requires T_3 time. In line 16, we specifically add c_j to $\hat{\mathcal{C}}_\mathbf{A}$ which has been sorted by line 2. This can be done by using the insertion sort, whose time complexity is $O(\xi_A)$. Thus, the overall time complexity is $O(\xi_O \log_2 \xi_O) + O(\xi_A \log_2 \xi_A) + \xi_S(O(1) + T_3 + O(\xi_A)) + \sum_{\forall c_i \in \hat{\mathcal{C}}_\mathbf{O}} O(|\hat{\mathcal{S}}_i| \log_2 |\hat{\mathcal{S}}_i|)$. Here, the last term is the total time taken by line 4 in the for-loop, which can be simplified to $O(\xi_S \log_2 \xi_S)$. Since there are more switches than controllers (i.e., $\max\{\xi_O, \xi_A\} < \xi_S$), by doing some algebra, the above complexity can be simplified to $\xi_S(T_3 + O(\xi_A + \log_2 \xi_S))$. □

5.2 Time-to-migration Calculating Module

Given two controllers c_i and c_j, where c_i is overloaded and c_j is an assistant, this module performs three tasks: 1) select a target switch s_k from c_i's subnet $\hat{\mathcal{S}}_i$, 2) decide the time τ when s_k should migrate, and 3) estimate the number q of s_k's PIMs that c_j will process. Algo. 3 gives its pseudocode. Let Δ be the minimum of c_i's overload (i.e., $L_{i,t} - \delta_i$) and c_j's remaining capacity (i.e., $\delta_j - L_{j,t}$), as shown in line 1. Then, $\hat{\mathcal{S}}_i$ is divided into two subsets $\hat{\mathcal{S}}_i^\alpha$ and $\hat{\mathcal{S}}_i^\beta$, where $\hat{\mathcal{S}}_i^\alpha$ contains those switches whose loads are no less than Δ and $\hat{\mathcal{S}}_i^\beta$ includes the residuary switches in $\hat{\mathcal{S}}_i$. The corresponding code is given in lines 2–7. Since $\hat{\mathcal{S}}_i$ has been sorted decreasingly based on the load $\zeta_{k,t}^{(i)}$ of each switch (by line 4 in Algo. 2), both $\hat{\mathcal{S}}_i^\alpha$ and $\hat{\mathcal{S}}_i^\beta$ are thus sorted accordingly.

To reduce the frequency of switch migration, we choose a switch s_k to migrate such that $\zeta_{k,t}^{(i)} \geq \Delta$ and $\zeta_{k,t}^{(i)}$ is as close to Δ as possible. Because $\hat{\mathcal{S}}_i^\alpha$ is a sorted set, one good choice for s_k is patently the *last* switch in $\hat{\mathcal{S}}_i^\alpha$. The corresponding code is presented in lines 8–9. Then, line 10 decides the migrating time τ and the transferred load q. Since $\zeta_{k,t}^{(i)} \geq \Delta$, we set q to Δ, which means that c_i and c_j process $(\zeta_{k,t}^{(i)} - \Delta)$ and Δ PIMs of s_k, respectively. Supposing that s_k produces PIMs at a constant rate, the migrating time can be estimated as follows:

$$\tau = [(\zeta_{k,t}^{(i)} - \Delta)/\zeta_{k,t}^{(i)}] \times P, \tag{3}$$

where P is the period length. This part realizes the substance of time-sharing migration for switches.

However, if $\hat{\mathcal{S}}_i^\alpha$ is empty, we will choose a switch from $\hat{\mathcal{S}}_i^\beta$ whose load is the closest to Δ, where the code is presented in lines 11–12. Similarly, since $\hat{\mathcal{S}}_i^\beta$ has been sorted, the best candidate for s_k must be the *first* switch in $\hat{\mathcal{S}}_i^\beta$. Because $\zeta_{k,t}^{(i)} < \Delta$, we can adopt the traditional migration, where c_j completely takes over s_k. Thus, line 13 sets $\tau = 0$ (i.e., switch migration occurs at the beginning of the period) and $q = \zeta_{k,t}^{(i)}$ (i.e., c_j should handle all of s_k's PIMs).

After that, line 14 updates $\zeta_{k,t}^{(i)}$ by $\zeta_{k,t}^{(i)} - q$ and $\zeta_{k,t}^{(j)}$ by q. In other words, c_i first handles $(\zeta_{k,t}^{(i)} - q)$ PIMs generated by s_k. After τ units of time, c_j will process q PIMs of s_k. Finally, line 15 returns the result (s_k, τ, q) to Algo. 2. Lemma 3 analyzes the time complexity of this module.

Lemma 3. Given ξ_S switches in $\hat{\mathcal{S}}$, Algo. 3 requires $O(\xi_S)$ time in the worst case.

Proof: In Algo. 3, the first two lines take a constant time to do the initialization. Because $\hat{\mathcal{S}}_i \subseteq \hat{\mathcal{S}}$, the for-loop in lines 3–7 repeats at most ξ_S times, in which every statement spends $O(1)$ time. Evidently, each statement in lines 8–15 takes $O(1)$ time. To sum up, the time complexity of Algo. 3 is $O(1) + \xi_S \times O(1) + O(1) = O(\xi_S)$. □

5.3 Discussion

Let us discuss the essence of TSSM. Algo. 1 first picks out overloaded controllers (i.e., $\hat{\mathcal{C}}_\mathbf{O}$) and assistant controllers (i.e., $\hat{\mathcal{C}}_\mathbf{A}$) from $\hat{\mathcal{C}}$. Note that those controllers whose workloads meet the condition of $\varphi \times \delta_i \leq L_{i,t} \leq \delta_i$ are skipped, because they have just enough resources to handle their current works (i.e., not overloaded) but their unexpended capacities (i.e., $\delta_i - L_{i,t}$) are too small to help other controllers. In this way, we can reduce unnecessary calculations and excess migrations.

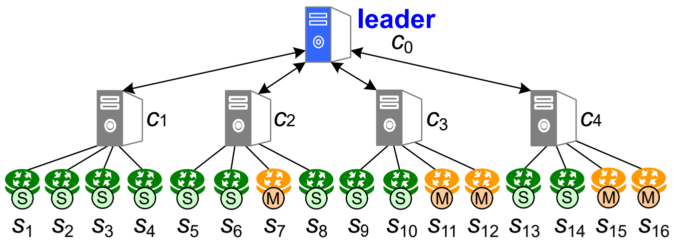
Since the amount of total overloads may exceed the amount of available spare capacities (in other words, $\sum_{\forall c_i \in \hat{\mathcal{C}}_\mathbf{O}} L_{i,t} - \delta_i > \sum_{\forall c_j \in \hat{\mathcal{C}}_\mathbf{A}} \delta_j - L_{j,t}$), the load-balanced migrating module in Algo. 2 gives a top priority to the controller c_i in $\hat{\mathcal{C}}_\mathbf{O}$ with the most overload to serve. Moreover, it selects the controller c_j in $\hat{\mathcal{C}}_\mathbf{A}$ that has the most remaining capacity to share c_i's workload. Then, the time-to-migration calculating module in Algo. 3 takes the *best-fit* policy to find an appropriate switch s_k managed by c_i to migrate such that its load $\zeta_{k,t}^{(i)}$ is as close to Δ as possible, where Δ considers not only the amount of c_i's overload but also the amount of c_j's remaining capacity. If $\zeta_{k,t}^{(i)} > \Delta$, the time-sharing migration is applied to let both c_i and c_j co-process s_k's PIMs in sequential order. Otherwise, we adopt the traditional migration by asking c_j to take over s_k at the beginning of the period.

Through the time-sharing migration, multiple controllers are able to share the load of a switch in the same period. However, the *migration cost*, which is defined by the number of times to perform switch migration, may increase when more controllers share the switch's load, as shown in Theorem 3. Hence, in TSSM, we restrict at most two controllers to participate in the time-sharing migration for a switch to save the cost. Theorem 4 gives an analysis on TSSM's time complexity.

Theorem 3. Let $\hat{\mathcal{S}}_\mathbf{O}$ be the set of switches managed by the overloaded controllers in $\hat{\mathcal{C}}_\mathbf{O}$. Suppose that at most m controllers can share the load of a switch in the same period. In the worst case for both the time-sharing migration and the traditional migration, the time-sharing migration adds $[(m-2)|\hat{\mathcal{S}}_\mathbf{O}| + |\hat{\mathcal{C}}_\mathbf{O}|]$ of migration cost than the traditional

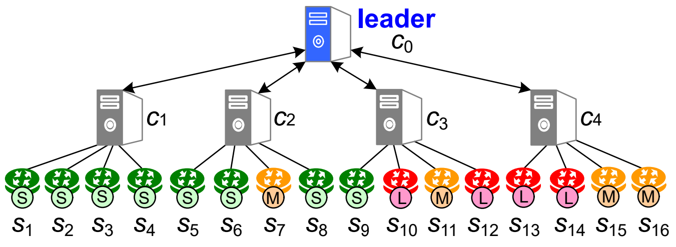
switch	load	PIM generation
S	small	15,000~16,000 PIMs/s
M	medium	31,000~32,000 PIMs/s
L	large	47,000~48,000 PIMs/s

(a) switches and their loads



leader
c_0
c_1
c_2
c_3
c_4
S S S S S S M S S S M M S S M M
s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 s_{10} s_{11} s_{12} s_{13} s_{14} s_{15} s_{16}

(b) topology at time 0



leader
c_0
c_1
c_2
c_3
c_4
S S S S S S M S S L M L L L M M
s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 s_{10} s_{11} s_{12} s_{13} s_{14} s_{15} s_{16}

(c) topology at the 11th second

Fig. 5: Network topology used in the ONOS implementation.

migration in a period. This is the comparison of the worst case for both migration manners.

Proof: In the traditional migration, a controller $c_i \in \hat{\mathcal{C}}_{\mathbf{O}}$ shifts at most $(|\hat{\mathcal{S}}_i| - 1)$ switches in a period, as c_i should keep at least one switch in its subnet. Therefore, its maximum cost is $\sum_{c_i \in \hat{\mathcal{C}}_{\mathbf{O}}} \sum_{s_k \in \hat{\mathcal{S}}_i} (|\hat{\mathcal{S}}_i| - 1) = |\hat{\mathcal{S}}_{\mathbf{O}}| - |\hat{\mathcal{C}}_{\mathbf{O}}|$. On the other hand, the time-sharing migration allows m controllers to share a switch's load, which implies that a switch will migrate at most $(m-1)$ times in a period. The maximum cost of the time-sharing migration will be $(m-1)|\hat{\mathcal{S}}_{\mathbf{O}}|$. Thus, the difference between their maximum costs is $(m-1)|\hat{\mathcal{S}}_{\mathbf{O}}| - (|\hat{\mathcal{S}}_{\mathbf{O}}| - |\hat{\mathcal{C}}_{\mathbf{O}}|) = (m-2)|\hat{\mathcal{S}}_{\mathbf{O}}| + |\hat{\mathcal{C}}_{\mathbf{O}}|$, thereby proving this theorem. □

Corollary 1. By restricting at most two controllers to participate in the time-sharing migration for a switch, in the worst case for both TSSM and the traditional migration, TSSM has no more than $|\hat{\mathcal{C}}_{\mathbf{O}}|$ of migration cost than the traditional migration in a period.

Theorem 4. The time complexity of TSSM is $O(\xi_S^2)$, where ξ_S is the number of switches in the network.

Proof: Based on Lemmas 1, 2, and 3, the time complexity of TSSM is $O(\xi_C + \xi_S) + \xi_S(O(\xi_S) + O(\xi_A + \log_2 \xi_S))$, where ξ_C and ξ_A are the numbers of total and assistant controllers, respectively. Because $\xi_S > \xi_C \geq \xi_A$, the above complexity can be simplified to $O(\xi_S^2)$. □

6 Performance Evaluation

The TSSM scheme is implemented on the ONOS platform [14] to attest to its feasibility and evaluate performance, which employs OpenFlow as the southbound protocol. Fig. 5 shows the network topology in our implementation, where there are 5 controllers (i.e., $\hat{\mathcal{C}} = \{c_0, c_1, \cdots, c_4\}$) and 16 switches (i.e, $\hat{\mathcal{S}} = \{s_1, s_2, \cdots, s_{16}\}$) in the network. The simulation time is 260 seconds, which is divided into 52 periods (in other words, the period length is 5 seconds). As mentioned in Section 2.1, we adopt the hierarchical paradigm of DSC architecture, so a controller (namely, c_0) serves as the leader to coordinate all other controllers. In particular, c_0 monitors the workload of every controller in $\hat{\mathcal{C}} \setminus \{c_0\}$ and decides how to perform switch migration (based on the migration method) in each period. It does not partake in switch management. Except for c_0, each controller has the capacity (i.e., Γ_i) of 10^6 PIMs per period and the threshold δ_i is set to $0.6\Gamma_i$ [34]. Thus, if a controller has to handle more than 600,000 PIMs in a period, it will be denoted as "overloaded" in that period.

In addition, we use the cbench tool [35], a popular controller benchmark for OpenFlow, to imitate the generation of PIMs by switches. Based on their loads, three types of switches are considered, as shown in Fig. 5(a):

- *Small load (SL):* Each SL switch sends 15,000 to 16,000 PIMs per second. Let Q denote the *controller affordable load (CAL)*, which is defined as the maximum number of PIMs that all controllers in $\hat{\mathcal{C}} \setminus \{c_0\}$ can handle in a period (without overloading them). CAL is estimated by $Q = \sum_{i=1}^{4} \delta_i = 2.4 \times 10^6$. When all switches in $\hat{\mathcal{S}}$ are SL switches, the *expected number of PIMs (ENP)* sent by them per period is $\sum_{i=1}^{16} 15,500 \times 5 < 0.52Q$. In other words, the total loads caused by all SL switches are below 52% of CAL, so their loads are small.
- *Medium load (ML):* An ML switch produces 31,000 to 32,000 PIMs every second. If all switches in $\hat{\mathcal{S}}$ are ML switches, the ENP per period will be $\sum_{i=1}^{16} 31,500 \times 5 = 1.05Q$. It means that the total loads produced by all ML switches are close to CAL. As compared with SL switches, loads of ML switches are relatively medium.
- *Large load (LL):* An LL switch will generate 47,000 to 48,000 PIMs per second. Suppose that $\hat{\mathcal{S}}$ contains only LL switches. Then, the ENP per period is $\sum_{i=1}^{16} 47,500 \times 5 > 1.58Q$. In this case, no matter how the switch migration is performed, some controllers must be overloaded. Evidently, loads of LL switches are large.

At time 0, the subnet of each controller (except the leader c_0) is set as follows: $\hat{\mathcal{S}}_1 = \{s_1, s_2, s_3, s_4\}$, $\hat{\mathcal{S}}_2 = \{s_5, s_6, s_7, s_8\}$, $\hat{\mathcal{S}}_3 = \{s_9, s_{10}, s_{11}, s_{12}\}$, $\hat{\mathcal{S}}_4 = \{s_{13}, s_{14}, s_{15}, s_{16}\}$. Switches s_7, s_{11}, s_{12}, s_{15}, and s_{16} are ML switches. All other switches are SL switches, as Fig. 5(b) shows. After 10 seconds, s_{10}, s_{12}, s_{13}, and s_{14} become LL switches, which may trigger switch migration, as illustrated in Fig. 5(c).

We also implement the following four methods on ONOS for comparison:

- *OpenFlow*: As mentioned in Section 2.2, OpenFlow provides the transferring procedure for switches (in Fig. 2). However, its standard does not specify how to choose target controllers and switches for migration. Thus, there is no switch migration in this method and the connections between controllers and switches are kept static in the 52 periods. The OpenFlow method is used as a baseline for performance evaluation. By comparing with OpenFlow, we can show the advantages of adaptively adjusting the connections between controllers

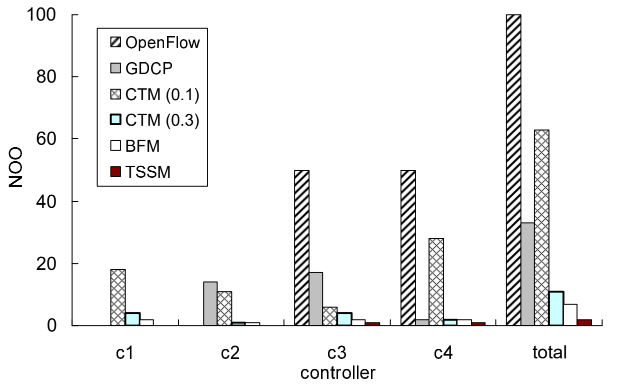


Fig. 6: Comparison on the NOO for controllers.

TABLE 3: The reduction ratio of NOO in each method.

method	GDCP	CTM (0.1)	CTM (0.3)	BFM	TSSM
ratio	67%	37%	89%	93%	98%

and switches by dynamical controller placement and switch migration.

- *Group-based dynamical controller placement (GDCP)* [10]: As mentioned in Section 1, GDCP checks each subnet in each period and reassigns its switches if necessary. To do so, GDCP groups controllers such that the workload of each group is as balanced as possible. The GDCP method is used to compare system performance between dynamical controller placement and switch migration.
- *Churn-triggered migration (CTM)* [30]: Let c_i and c_j be an overloaded controller and a target controller, respectively. CTM allows the switch in c_i's subnet that has the maximum PIMs migrating to c_j's subnet if the following condition obtains:
$$L_{j,t} \leq (1-\varepsilon) \times L_{i,t}, \tag{4}$$
where ε is set to 0.1 (for a small churn value) and 0.3 (for a large churn value). The CTM method has the same objective as our work (that is, balance the workloads of controllers), so we compare CTM with TSSM.
- *Best-fit migration (BFM)*: This method is almost the same as TSSM, except that it does not adopt the time-sharing migration, but only uses the traditional migration instead. To implement BFM, we replace line 10 with line 13 in Algo. 3. The BFM method is used to assess the effect of the time-sharing migration on our TSSM scheme.

In both BFM and TSSM, the parameter φ is set to 0.95.

6.1 Occurrences of Overload

We measure the *number of occurrences of overload* (below, it is called "NOO" for short) for controllers in the total 52 periods. As mentioned in Section 4.3, the NOO for a controller c_i is calculated by $\sum_{t=1}^{52} f(c_i, t)$, where $c_i \in \{c_1, c_2, c_3, c_4\}$. According to Eq. (2), the objective is to minimize the sum of NOO of all controllers, that is, prevent controllers from being overloaded as much as possible.

Fig. 6 gives the experimental result. The OpenFlow method does not perform switch migration. Because only the switches in subnet $\hat{\mathcal{S}}_3$ and $\hat{\mathcal{S}}_4$ are changed to high-load switches after 10 seconds, the NOO for controllers c_1 and c_2 are zero (that is, they are not overloaded). However, both controllers c_3 and c_4 are always overloaded from the 3rd period to the 52nd period. This leads OpenFlow to have the highest NOO among all methods (in terms of all controllers), which manifests the necessity of switch migration.

The GDCP method checks if the workload of each group of controllers is balanced in every period. If not, some subnets are rearranged accordingly. In this way, GDCP can greatly reduce NOO as compared with OpenFlow. However, some high-load switches repeatedly migrate between subnets $\hat{\mathcal{S}}_2$ and $\hat{\mathcal{S}}_3$. Thus, the NOO of controllers c_2 and c_3 in GDCP are higher than those in most of the other methods. This means that GDCP suffers from the controller ping-pong difficulty, which will be further explained in Section 6.2.

For the CTM method, the churn ε has a significant impact on its performance. Specifically, if the churn is set too small (i.e., $\varepsilon = 0.1$), there will be more candidate controllers to be chosen for switch migration. However, some of them may not have enough remaining capacity to help take over high-load switches (from overloaded controllers). Thus, the occurrence of controller ping-pong effect becomes more frequent, especially for controllers c_1 and c_4. The above problem can be mitigated by selecting a larger churn value (e.g., $\varepsilon = 0.3$). In this case, the NOO of total controllers in CTM will decrease from 63 to 11.

The BFM method adopts Algo. 3 (without the time-sharing migration) to find an appropriate switch s_k of an overloaded controller c_i to migrate to the subnet of an assistant controller c_j, where s_k's load is as close to Δ as possible. Since Δ is the minimum of c_i's overload and c_j's remaining capacity, c_j would be capable of attending to s_k. In this way, BFM can substantially reduce the NOO of total controllers.

Our TSSM scheme not only selects the most suitable switch for migration but also applies the time-sharing migration to let two controllers co-process the PIMs of a high-load switch in the same period. Thus, TSSM can further decrease the NOO of total controllers, as compared with BFM. Table 3 gives the reduction ratio of NOO in each method, where we take the OpenFlow method as the baseline. In particular, let R_{OpenFlow} and R_χ be the NOO of total controllers in OpenFlow and a method χ, respectively. Then, the reduction ratio of method χ is defined by

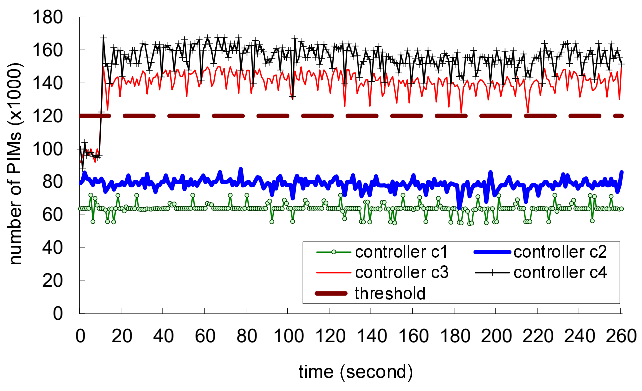
$$\frac{R_{\text{OpenFlow}} - R_\chi}{R_{\text{OpenFlow}}} \times 100\%. \tag{5}$$

As can be seen, the TSSM scheme has the highest reduction ratio of NOO, which demonstrates its high efficiency in terms of preventing controllers from being overloaded.

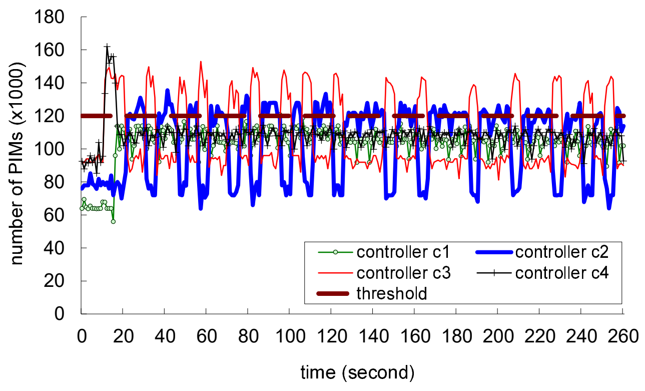
6.2 Workloads of Controllers

Next, let us evaluate the number of PIMs processed by each controller (i.e., its workload) every second. Since a controller can process at most 600,000 PIMs per period (otherwise, it is treated as one overloaded controller) and the period length is 5 seconds, if the controller's workload exceeds 120,000 PIMs in a second, there is a good possibility that the controller will be overloaded. Thus, we put a threshold line (on 120,000 PIMs) for reference in the experimental results in Fig. 7.

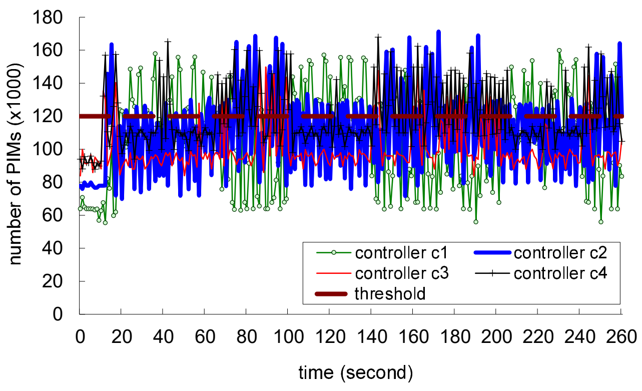
Fig. 7(a) shows the workloads of controllers in the OpenFlow method, where the connections between the 4 controllers and the 16 switches are static. Since the types of the switches in subnets $\hat{\mathcal{S}}_1$ and $\hat{\mathcal{S}}_2$ do not change, the workloads of controllers



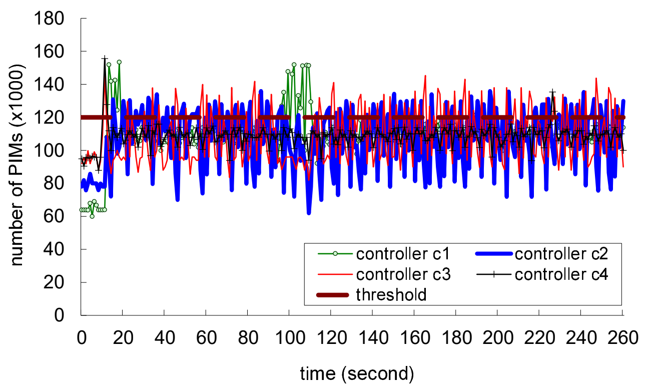
(a) OpenFlow



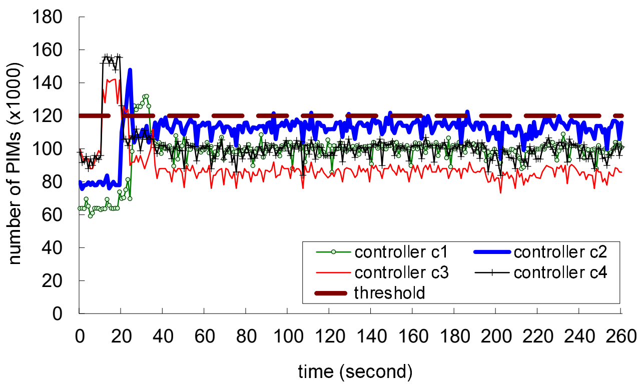
(b) GDCP



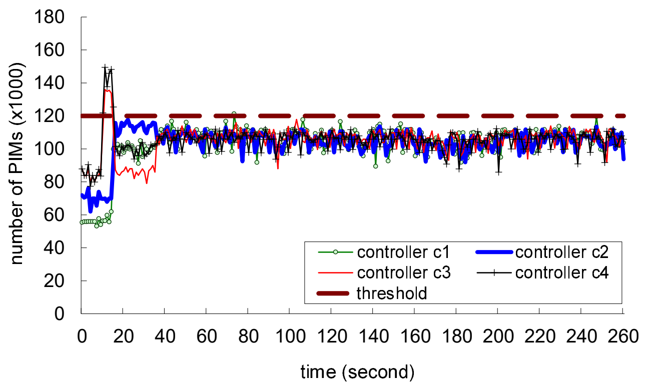
(c) CTM ($\varepsilon = 0.1$)



(d) CTM ($\varepsilon = 0.3$)



(e) BFM



(f) TSSM

Fig. 7: Comparison on the workload of each controller.

c_1 and c_2 keep steady with just small fluctuations. On the other hand, after 10 seconds, some switches in subnets $\hat{\mathcal{S}}_3$ and $\hat{\mathcal{S}}_4$ become high-load switches, which burden controllers c_3 and c_4 with heavy loads. Therefore, both c_3 and c_4 have to process many more than 120,000 PIMs per second. In other words, they are seriously overloaded. This experimental result shows the drawback of keeping static connections between controllers and switches.

Fig. 7(b) gives the workloads of controllers in the GDCP method. Interestingly, the workloads of controllers c_2 and c_3 swing significantly in a periodic manner. If c_2's workload rises, c_3's workload drops, and vice versa. This phenomenon implies that some switches are repeatedly moved between subnets $\hat{\mathcal{S}}_2$ and $\hat{\mathcal{S}}_3$, which causes c_2 and c_3 to overload in some periods. In other words, the controller ping-pong effect occurs. That also explains why GDCP has higher NOO (in terms of controllers c_2 and c_3) than other methods in Fig. 6.

Fig. 7(c) and (d) show the workloads of controllers in the CTM method by setting ε to 0.1 and 0.3, respectively. Since CTM selects a candidate controller to share the workload of an overloaded controller by Eq. (4), a smaller ε value implies that there will be more candidates for selection. However, not every candidate controller is appropriate to share the workload, as they may not have enough remaining capacity. In this case, some switches would be frequently transferred among different subnets and make their controllers overloaded. From Fig. 7(c), we can observe that CTM encounters the controller ping-pong effect when $\varepsilon = 0.1$. On the other hand, by setting a larger ε value (i.e., $\varepsilon = 0.3$), the workloads of controllers can be more close to the threshold (i.e., 120,000 PIMs), as compared with the case of $\varepsilon = 0.1$.

Fig. 7(e) gives the workloads of controllers in the BFM method. Unlike the CTM method to choose the switch with the maximum PIMs for migration, BFM adopts Algo. 3 (without

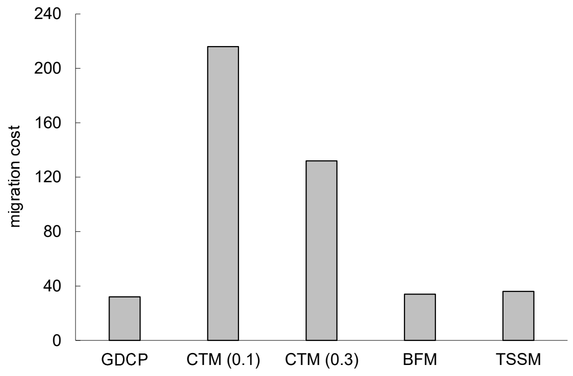


Fig. 8: Comparison on the migration cost.

the time-sharing migration) to find a switch whose load is close to Δ to realize the best-fit policy. In this way, the workload of each controller is kept below the threshold after 33 seconds.

Fig. 7(f) presents the workloads of controllers in our TSSM scheme. As compared with BFM, the time when the workloads of controllers drop below the threshold is earlier (specifically, after 15 seconds). Thanks to the time-sharing migration, TSSM allows two controllers to co-process the PIMs of a high-load switch, thereby further balancing their workloads. In particular, the workload of each controller is pretty close to each other after 40 seconds. This experimental result ensures that TSSM can achieve load balance among all controllers.

6.3 Migration Cost

Fig. 8 compares the migration cost of each method, which is defined by the aggregate number of times to transfer switches among different subnets. Since the OpenFlow method does not perform switch migration, its cost is not presented in Fig. 8.

GDCP groups controllers and reassigns their switches such that loads of groups are as balanced as possible. According to the topology in Fig. 5(c), controllers are divided into two groups $\{c_1, c_4\}$ and $\{c_2, c_3\}$, where each group manages 4 SL switches, 2 ML switches, and 2 LL switches. To achieve load balance, a controller is expected to manage 2 SL switches, 1 ML switch, and 1 LL switch. Thus, GDCP performs 4 times of switch migration for group $\{c_1, c_4\}$ (in particular, two SL switches are moved from c_1's subnet to c_4's subnet, and 1 ML switch and 1 LL switch are moved from c_4's subnet to c_1's subnet) and 2 times of switch migration for group $\{c_2, c_3\}$ (in particular, 1 SL switch is moved from c_2's subnet to c_3's subnet, and 1 LL switch is moved from c_3's subnet to c_2's subnet) at the 4th period. Moreover, since the PIMs generated by a switch will fluctuate, it may make loads of the two controllers in a group unbalanced. This phenomenon occurs in group $\{c_2, c_3\}$, as shown in Fig. 7(b). Thus, GDCP picks one switch to migrate between the subnets of c_2 and c_3 around every two periods after the 5th period. In this case, the migration cost of GDCP is $4 + 2 + \lceil (52 - 5)/2 \rceil + 2$ (as two extra switches are reassigned in some periods) $= 32$.

CTM lets a switch s_k managed by an overloaded controller c_i that has the maximum PIMs migrate to the subnet of another controller c_j if the condition in Eq. (4) holds. Let $\zeta_{k,t}$ be the number of PIMs produced by s_k. There is a good possibility that c_j will transfer s_k back to c_i in the next period when the following two conditions are both met. First, c_j becomes overloaded after taking over s_k, that is,

$$L_{j,t} + \zeta_{k,t} > \delta_j. \tag{6}$$

Second, Eq. (4) holds for c_j (after taking over s_k), that is,

$$L_{i,t} - \zeta_{k,t} \leq (1 - \varepsilon) \times (L_{j,t} + \zeta_{k,t}). \tag{7}$$

By combining Eqs. (6) and (7) and doing some algebras, we can derive that

$$\zeta_{k,t} > \max\left\{\frac{L_{i,t} - (1 - \varepsilon)L_{j,t}}{2 - \varepsilon}, \delta_j - L_{j,t}\right\}. \tag{8}$$

In Eq. (8), $\zeta_{k,t}$ may become smaller when the churn ε is smaller. Thus, s_k need not have a large load $\zeta_{k,t}$ to satisfy the condition in Eq. (8). In this case, s_k would repeatedly migrate between the subnets of c_i and c_j, thereby causing the controller ping-pong difficulty. In fact, we can observe that this difficulty does occur in Fig. 7(c) and (d). Moreover, CTM makes multiple switches migrate in each period, which increases its migration cost. The above dilemma can be mitigated by using a larger churn. In particular, the migration cost of CTM will decrease from 216 to 132 when ε is changed from 0.1 to 0.3.

BFM is similar to TSSM, except that it employs only the traditional migration. Based on Algo. 2 and Algo. 3, BFM picks a switch s_k whose load is close to Δ to migrate, where Δ is the minimum between c_i's overload (i.e., $L_{i,t} - \delta_i$) and c_j's remaining capacity (i.e., $\delta_j - L_{j,t}$). In this way, after c_i transfers s_k to c_j, c_j would not let s_k migrate soon, as c_j has enough capacity to serve s_k. Thus, BFM also chooses one switch for migration every two periods. Unlink GDCP that drastically changes some subnets at the 4th period, BFM gradually transfers one switch of an overloaded controller. This will increase BFM's migration cost, as compared with GDCP. In particular, the migration cost of BFM is 34, which is slightly higher than that of GDCP.

Like BFM, TSSM picks switches whose loads are close to Δ for migration, but it allows two controllers to share their loads in the same period (i.e., time-sharing migration). As mentioned in Corollary 1, TSSM has no more than $|\hat{\mathcal{C}}_\mathbf{O}|$ of migration cost than the traditional migration in a period, where $\hat{\mathcal{C}}_\mathbf{O}$ is the set of overloaded controllers. The result in Fig. 8 also validates Corollary 1, where the migration cost of TSSM is 36, which is slightly higher than that of BFM.

We remark that the migration cost of each method will rise by increasing the numbers of controllers and switches. In essence, there will be more overloaded controllers needed to be handled, and more switches have to migrate to help reduce their workloads. However, the behavior of each method will not significantly change when the numbers of controllers and switches increase. Thus, the trend of migration costs of these methods will be similar to the result in Fig. 8, where CTM still encounters the controller ping-pong difficulty and thereby has a much higher migration cost than GDCP, BFM, and TSSM, especially when $\varepsilon = 0.1$.

7 Conclusion and Future Work

DSC overcomes the performance bottleneck problem in an SDN-based network with one single controller, where switch migration is widely used to balance the workloads of multiple controllers. This paper proposes the TSSM scheme to let two controllers co-process the PIMs of a switch through the time-sharing migration, which holds the workload of each controller below a predefined threshold. For each overloaded controller, TSSM finds assistant controllers for help, selects switches to migrate, and decides the time to transfer each selected switch. We implement the TSSM scheme on the ONOS platform to verify its feasibility, where the experimental results show that TSSM can have smaller NOO and make the workload of each controller more balanced, as compared with the OpenFlow, GDCP, CTM, and BFM methods. Moreover, TSSM has a similar migration cost with both GDCP and BFM, which is much lower than CTM.

In TSSM, we assume that at most two controllers partake in the time-sharing migration for one single switch to save the migration cost. Although relaxing this assumption adds more flexibility to the time-sharing migration, the cost may increase significantly, as shown in Theorem 3. Therefore, we will study how to let more than two controllers participate in the time-sharing migration without overgrowing the migration cost. In addition, the protocol family in the SDN architecture with OpenFlow may be large. Thus, the P4 method is proposed as an integrated way of solution. It deserves further investigation on the switch migration problem when using the P4 method. Finally, the extensive cost-benefit analysis of different migration methods (e.g., whether there are any statistical bounds on their migration costs) will be desired in future work.

References

[1] N. Anerousis, P. Chemouil, A.A. Lazar, N. Mihai, and S.B. Weinstein, "The origin and evolution of open programmable networks and SDN," *IEEE Comm. Surveys & Tutorials*, vol. 23, no. 3, pp. 1956–1971, 2021.

[2] Y.C. Wang and H. Hu, "An adaptive broadcast and multicast traffic cutting framework to improve Ethernet efficiency by SDN," *J. Information Science and Engineering*, vol. 35, no. 2, pp. 375–392, 2019.

[3] M. Alsaeedi, M.M. Mohamad, and A.A. Al-Roubaiey, "Toward adaptive and scalable OpenFlow-SDN flow control: A survey," *IEEE Access*, vol. 7, pp. 107,346–107,379, 2019.

[4] J.H. Cox, R. Clark, and H. Owen, "Leveraging SDN and WebRTC for rogue access point security," *IEEE Trans. Network and Service Management*, vol. 14, no. 3, pp. 756–770, 2017.

[5] Y.C. Wang and S.Y. You, "An efficient route management framework for load balance and overhead reduction in SDN-based data center networks," *IEEE Trans. Network and Service Management*, vol. 15, no. 4, pp. 1422–1434, 2018.

[6] W. Iqbal, H. Abbas, P. Deng, J. Wan, B. Rauf, Y. Abbas, and I. Rashid, "ALAM: Anonymous lightweight authentication mechanism for SDN-enabled smart homes," *IEEE Internet of Things J.*, vol. 8, no. 12, pp. 9622–9633, 2021.

[7] Y.C. Wang and R.X. Ye, "Credibility-based countermeasure against slow HTTP DoS attacks by using SDN," *Proc. IEEE Annual Computing and Comm. Workshop and Conf.*, 2021, pp. 890–895.

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

[9] J. Lu, Z. Zhang, T. Hu, P. Yi, and J. Lan, "A survey of controller placement problem in software-defined networking," *IEEE Access*, vol. 7, pp. 24290–24307, 2019.

[10] H. Sufiev, Y. Haddad, L. Barenboim, and J. Soler, "Dynamic SDN controller load balancing," *Future Internet*, vol. 11, no. 3, pp. 1–21, 2019.

[11] Y. Wu, S. Zhou, Y. Wei, and S. Leng, "Deep reinforcement learning for controller placement in software defined network," *Proc. IEEE INFOCOM Workshop*, 2020, pp. 1254–1259.

[12] Y.C. Wang and Y.C. Wang, "Efficient and low-cost defense against distributed denial-of-service attacks in SDN-based networks," *Int'l J. Comm. Systems*, vol. 33, no. 14, pp. 1–24, 2020.

[13] F. Tang, H. Zhang, L.T. Yang, and L. Chen, "Elephant flow detection and load-balanced routing with efficient sampling and classification," *IEEE Trans. Cloud Computing*, vol. 9, no. 3, pp. 1022–1036, 2021.

[14] ONOS. [Online]. Available: https://opennetworking.org/onos/

[15] W.H.F. Aly, "Controller adaptive load balancing for SDN networks," *Proc. Int'l Conf. Ubiquitous and Future Networks*, 2019, pp. 514–519.

[16] L. Richardson and S. Ruby, *RESTful Web Services*. Sebastopol: O'Reilly, 2007.

[17] Y.C. Chan, K. Wang, and Y.H. Hsu, "Fast controller failover for multi-domain software-defined networks," *Proc. European Conf. Networks and Comm.*, 2015, pp. 370–374.

[18] T. Hu, J. Zhang, L. Cao, and J. Gao, "A reliable controller deployment strategy based on network condition evaluation in SDN," *Proc. IEEE Int'l Conf. Software Engineering and Service Science*, 2017, pp. 367–370.

[19] T. Kim, J. Myung, and S.E. Yoo, "Load balancing of distributed datastore in OpenDaylight controller cluster," *IEEE Trans. Network and Service Management*, vol. 16, no. 1, pp. 72–83, 2019.

[20] Y.C. Wang and E.J. Chang, "Cooperative flow management in multi-domain SDN-based networks with multiple controllers," *Proc. IEEE Int'l Conf. Smart Communities: Improving Quality of Life Using ICT, IoT and AI*, 2020, pp. 82–86.

[21] S. Nithya, M. Sangeetha, K.N.A. Prethi, K.S. Sahoo, S.K. Panda, and A.H. Gandomi, "SDCF: A software-defined cyber foraging framework for cloudlet environment," *IEEE Trans. Network and Service Management*, vol. 17, no. 4, pp. 2423–2435, 2020.

[22] K.S. Sahoo, P. Mishra, M. Tiwary, S. Ramasubbareddy, B. Balusamy, and A.H. Gandomi, "Improving end-users utility in software-defined wide area network systems," *IEEE Trans. Network and Service Management*, vol. 17, no. 2, pp. 696–707, 2020.

[23] A. Dixit, F. Hao, S. Mukherjee, T.V. Lakshman, and R. Kompella, "Towards an elastic distributed SDN controller," *ACM SIGCOMM Computer Comm. Review*, vol. 43, no. 4, pp. 7–12, 2013.

[24] Z. Min, Q. Hua, and Z. Jihong, "Dynamic switch migration algorithm with Q-learning towards scalable SDN control plane," *Proc. Int'l Conf. Wireless Comm. and Signal Processing*, 2017, pp. 1–4.

[25] J. Cui, Q. Lu, H. Zhong, M. Tian, and L. Liu, "SMCLBRT: A novel load-balancing strategy of multiple SDN controllers based on response time," *Proc. IEEE Int'l Conf. High Performance Computing and Comm.*, 2018, pp. 541–546.

[26] K.S. Sahoo, D. Puthal, M. Tiwary, M. Usman, B. Sahoo, Z. Wen, B.P.S. Sahoo, and R. Ranjan, "ESMLB: Efficient switch migration-based load balancing for multicontroller SDN in IoT," *IEEE Internet of Things J.*, vol. 7, no. 7, pp. 5852–5860, 2020.

[27] T. Hu, J. Lan, J. Zhang, and W. Zhao, "EASM: Efficiency-aware switch migration for balancing controller loads in software-defined networking," *Peer-to-Peer Networking and Applications*, vol. 12, pp. 452–464, 2019.

[28] Y. Hu, W. Wang, X. Gong, X. Que, and S. Cheng, "BalanceFlow: Controller load balancing for OpenFlow networks," *Proc. IEEE Int'l Conf. Cloud Computing and Intelligence Systems*, 2012, pp. 780–785.

[29] W. Lan, F. Li, X. Liu, and Y. Qiu, "A dynamic load balancing mechanism for distributed controllers in software-defined networking," *Proc. Int'l Conf. Measuring Technology and Mechatronics Automation*, 2018, pp. 259–262.

[30] B. Gorkemli, S. Tatlcolu, A.M. Tekalp, S. Civanlar, and E. Lokman, "Dynamic control plane for SDN at scale," *IEEE J. Selected Areas in Comm.*, vol. 36, no. 12, pp. 2688–2701, 2018.

[31] V. Sridharan, M. Gurusamy, and T. Truong-Huu, "On multiple controller mapping in software defined networks with resilience constraints," *IEEE Comm. Letters*, vol. 21, no. 8, pp. 1763–1766, 2017.

[32] F. Al-Tam and N. Correia, "Fractional switch migration in multi-controller software-defined networking," *Computer Networks*, vol. 157, pp. 1–10, 2019.

[33] J. Kleinberg and E. Tardos, *Algorithm Design*. Boston: Addison-Wesley, 2006.

[34] Y. Xu, M. Cello, I.C. Wang, A. Walid, G. Wilfong, C.H.P. Wen, M. Marchese, and H.J. Chao, "Dynamic switch migration in distributed software-defined networks to achieve controller load balance," *IEEE J. Selected Areas in Comm.*, vol. 37, no. 3, pp. 515–529, 2019.

[35] cbench. [Online]. Available: https://github.com/trema/cbench