Towards Adaptive Elastic Distributed Software Defined Networking

Yanyu Chen[†‡], Qing Li[†], Yuan Yang[‡], Qi Li[†], Yong Jiang[†], Xi Xiao[†]
[†]Graduate School at Shenzhen, Tsinghua University, [‡]Dept. of Computer Science, Tsinghua University
chenyy14@mails.tsinghua.edu.cn, {li.qing, jiangy, xiaox}@sz.tsinghua.edu.cn, {yyang, liqi}@csnet1.cs.tsinghua.edu.cn

***Abstract*—Distributed Software Defined Networking federates multiple controllers in a network to solve the problems in single controller networks, e.g., to improve network reliability and reduce the delay between controllers and switches. However, in the current distributed SDN schemes, the mapping between SDN switches and controllers is statically configured, which may result in uneven load distribution among controllers. These schemes cannot fully benefit from the distributed SDN architecture. In order to address this issue, this paper proposes an *adaptive elastic distributed SDN architecture*. The architecture dynamically selects a minimum number of active controllers that switches attached to, and changes the mapping between switches and controllers according to the network load. Specially, a switch can migrate from one controller domain to another so that the mapping is adaptive to the network load. We formalize the controller selection problem as an optimization problem, and prove that the problem is NP-Hard. We solve the problem by using greedy algorithms. With the algorithms, controllers in a network are dynamically changed with respect to the network load and the number of active controllers is close to the optimal value. We validate the algorithms and evaluate the performance by simulations. In particular, the simulation results show that the number of active controllers is reduced by 60% when the whole network load decreases.**

I. Introduction

Software Defined Networking (SDN) [1], [2] enables network programmability and easy management [3], [4], [5]. Since it achieves a centralized control plane architecture, it brings up some issues of scalability and reliability. Multiple distributed SDN controllers [6], [7], [8], [9] are proposed to address these issues. Most of existing work focuses on the state consistency issue among multiple controllers. The mapping between switches and controllers is statically configured, which may result in uneven load distribution among controllers and controller crash by packet burst [10]. Therefore, these approaches cannot fully benefit from the distributed architecture.

In order to address the issues above, Advait et al. [11] proposed an elastic distributed SDN controller scheme that dynamically shrinks and expands the controller pool with respect to the network load. To realize the elastic distributed SDN controller, a switch migration protocol is proposed to enable a switch migration from one controller domain to another. Unfortunately, the design only includes a basic elastic SDN controller architecture and a migration proposal. It does not discuss the key question in the elastic architecture, i.e., how to make a controller be adaptive to runtime network load with changes of network load and topology. The key challenge in the distributed SDN architectures remains unresolved. If switches cannot correctly select controllers for migration with network changes, namely *controller selection problem*, the scheme will still fail in balancing the controller load.

In this paper, we formalize the controller selection problem as an optimization problem and prove that it is NP-Hard. We solve the problem by using greedy algorithms with which controllers are dynamically selected according to their loads. In particular, switches can migrate to different controllers with respect to the network load, and the controllers can be activated and inactivated. When the whole network load falls below a given lower threshold, switches will re-select controllers and migrate to the new controllers to reduce the number of active controllers. After switch migrations, controllers without any load can be inactivated. Once the loads of controllers exceed a given upper threshold, some switches attached to them migrate to re-selected controllers that have less load or that are inactive. Thereby, the controller pool is dynamically shrinking and expanding. Note that, since switches change their corresponding controllers only when there is controller crash or significant control load change, the network stability is ensured with the proposed distributed architecture.

In summary, this paper makes the following contributions:

- We propose an adaptive elastic SDN controller architecture, and formally define the problem of controller selection in the elastic architecture. We prove that the controller selection problem is NP-Hard.
- We develop greedy algorithms to solve the controller selection problem under different network loads. Our algorithms enable dynamic controller selection by migrating switches in runtime.
- We use simulations to evaluate the performance of the proposed scheme. When the network load decreases, the simulation results show that the number of inactive controllers achieved by our scheme is not less than 90% of the optimal results, and the number of active controllers is significantly reduced by 60% when the total network load is light. When the network load increases, the simulation results show that the controller pool can expand accurately and quickly, but only incurs 15.6% more controllers than the optimal algorithm. These results demonstrate that controllers in our architecture can be effectively adaptive to network load.

The rest of this paper is structured as follows. Section II

978-1-4673-8590-9/15/$31.00 ©2015 IEEE

reviews background and related work. Section III presents the model and problem formulation. In Section IV, we develop greedy algorithms to solve the controller selection problem. The evaluation of the algorithms is presented in Section V. Section VI concludes the paper.

II. Background And Related work

The SDN controller architecture has been developed from a centralized one to a distributed one. The centralized system experienced single-threaded design [12] and multi-threaded design [13], [14], [15] in recent years. For instance, Yeganeh et. al. [14] proposed Kandoo which used two layers of controllers to control the whole network. The bottom layer is a set of controllers disconnected to each other, and the top layer is the logically centralized controller. The design of Kandoo is much similar to the distributed system.

However, the centralized controller has limitations in scalability and reliability. Thus, researchers proposed to use multiple distributed SDN controllers [6], [7], [8]. For example, Tootoonchian et. al. [7] proposed HyperFlow, which is a logically centralized physically distributed control plane. HyperFlow replicates the events at all distributed nodes so that each node can process events and update their states.

To tackle these issues, [16] proposed Load Variance-based Synchronization (LVS) to improve the load-balancing performance, and [11] proposed an elastic distributed SDN controller system, which attract much attention. In a distributed SDN network, each switch can attach to more than one controller. Only one of the controllers can act as the master controller, while others act as equal or slave controllers [17], [18]. With a switch migration protocol [11], switches that controlled by a controller can be seamlessly migrated to another. However, the design does not discuss how to achieve the architecture adaptive to runtime network load, and only includes a basic elastic SDN controller architecture and a switch migration proposal.

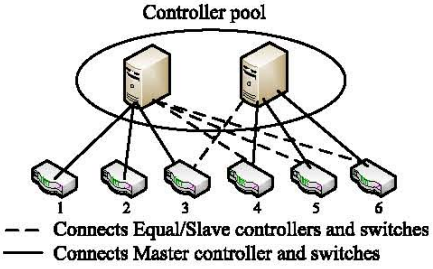

Figure 1: An illustration of the network

Fig.1 shows an example of a network topology. The solid line connects switches and a master controller, and the dotted line connects switches and an equal or slave controller. Two controllers are selected and active in the controller pool now. The selected controllers will be changed with the change of network load. For example, when the load of one controller falls below a given threshold and meets some conditions (see Section III), it can be inactivated. Note that, since both equal and slave controllers do not respond to controller messages, e.g., Packet-In packet, for simplicity, we do not distinguish them in this paper.

III. Problem Statement and Modeling

We present the controller selection problem in this section. In a network, each SDN switch (switch for short) has one or more connections to the controllers. Each switch has a traffic demand, which reflects the traffic volume (control and configuration messages, etc.) between the switch and the controller. On the one hand, a controller cannot hold a traffic volume that exceeds the capacity. More strictly, the traffic volume should not exceed a threshold to improve the resilience against a sudden traffic increasing or switch migrations. On the other hand, a controller whose traffic volume is zero can be inactivated. Our target is to minimize the number of active controllers, and achieve load balancing. The methodology and model can be applied to the scenarios that the network load increases. In these scenarios, switches can migrate and reattach to controllers if controller load exceeds the threshold.

Formally, let $\mathcal{C}$ denote the set of controllers and $|\mathcal{C}| = n$. For each controller $c_i \in \mathcal{C}$ $(1 \leq i \leq n)$, let T_i be the capacity, i.e., the maximum traffic volume that c_i can hold, and α $(0 < \alpha \leq 1)$ be a threshold that the ratio of the actual traffic volume to T_i must not exceed. Let B_i be a binary variable that indicates whether controller c_i is active. Let $\mathcal{S}$ denote the set of switches and $|\mathcal{S}| = m$. For each $c_i \in \mathcal{C}$ $(1 \leq i \leq n)$ and $s_j \in \mathcal{S}$ $(1 \leq j \leq m)$, $R_{i,j}$ is a ternary variable that indicates whether there is a connection between switch s_j and controller c_i. If there is a connection between the master controller and switch, $R_{i,j} = 1$, or there is a connection between a slave/equal controller and switch, $R_{i,j} = -1$, or else $R_{i,j} = 0$. Let F_j be the traffic demand of switch s_j, and $\delta_{i,j}$ be the fraction of traffic demand F_j that is carried by the connection between c_i and s_j. We model the controller selection problem as follows.

$$\min \sum_{i=1}^{n} B_i \tag{1}$$

$$\text{subject to:} \quad B_i \in \{0,1\}, \qquad \forall i \in \{1,\ldots,n\} \tag{2}$$

$$\sum_{i=1}^{n} \delta_{i,j} B_i = 1, \qquad \forall j \in \{1,\ldots,m\} \tag{3}$$

$$\delta_{i,j} \in [0,1], \quad \forall i \in \{1,\ldots,n\}, \quad \forall j \in \{1,\ldots,m\} \tag{4}$$

$$\sum_{i=1}^{n} \delta_{i,j} \mid R_{i,j} \mid = 1, \qquad \forall j \in \{1,\ldots,m\} \tag{5}$$

$$\sum_{j=1}^{m} \delta_{i,j} \mid R_{i,j} \mid F_{i,j} \leq \alpha T_i. \quad \forall i \in \{1,\ldots,n\} \tag{6}$$

Note that, Eq. (1) is our objective, i.e., minimizing the number of active controllers. Eq. (2) means that each controller is either active or inactive. Eq. (3) means that the traffic demand of each switch must be fully satisfied by active controllers. Eq. (4) means that the traffic demand of each switch must be delivered as a whole. Eq. (5) means that the

traffic demand of each switch must be satisfied through only one existing connection[1]. Eq. (6) means that the total traffic volume on a controller must be less than a given threshold. The input variables of the model are m, n, $R_{i,j}$, F_j, α, and T_i, while the decision variables are B_i and $\delta_{i,j}$. Now we analyze the complexity of the problem.

Theorem 1. *The controller selection problem is NP-Hard.*

Proof: We prove the theorem by a polynomial time reduction from the bin packing problem, which is known to be NP-hard [19], to the controller selection problem.

The bin packing problem is to pack a finite number of objects with weights $w_1, \ldots, w_m$ into a finite number of bins that have the same capacity W, in a way that minimizes the number of bins used. For each instance of the bin packing problem, we can construct an instance of the controller selection problem in polynomial time as follows. For each object with weight w_j, add switch s_j into $\mathcal{S}$, and set the traffic demand to w_j, i.e., $\boldsymbol{F}_j = w_j$. Then, construct a set of enough number of controllers, where each controller c_i has the same capacity $\boldsymbol{T}_i = W$. Let α be 1. Let $\boldsymbol{R}_{i,j} = 1$ for each (i, j) pair, such that the traffic demand of any switch can be satisfied by any controller. In the constructed problem, the controllers can be seen as the bins while the switches are corresponding to the objects, and we can see that the optimal solution to the bin packing problem is also the optimal solution to the constructed problem. Thus, the two problems are equivalent, and this ends our proof. ■

Due to Theorem 1, we need a heuristic algorithm to solve the controller selection problem.

IV. The Algorithm

In this section, we present our elastic adaptive SDN scheme for dynamic controller selection. We develop algorithms to shrink and expand the controller pool according to the network load, and also balance the load of controllers.

A. Overview

In our scheme, the network load is monitored by a central database [11]. The total load of a past period is used to decide whether the controller pool needs shrink or expansion. In particular, when the load of some controllers exceeds a given threshold $\alpha \boldsymbol{T}_i$, the controller pool is expanded by activating more controllers, to prevent the network from breaking down. When the average network load is less than a given threshold $\beta \boldsymbol{T}_{mean}$[2], the controller pool is shrunk by inactivating some controllers to save power. We note that uneven controller loads may incur an overload or a small mean load, but the controller pool may not need change in such cases. To avoid unnecessary controller reselection, we need to balance the controller loads when the loads are unevenly distributed. Load balancing is also needed when the network load is uneven after changing the controller pool and migrating the switches.

[1]Note that an existing connection may not connect to an active controller, so we need both Eq. (3) and Eq. (5).

[2]β is the lower threshold parameter, $\boldsymbol{T}_{mean}$ here denotes the mean of all $\boldsymbol{T}_i$, $i \in \{1, \ldots, n\}$

Table I: Notations used in this paper

Parameters	Meaning
α	the upper threshold parameter
β	the lower threshold parameter
D	the threshold of load deviation
$\boldsymbol{B}_i$	a binary variable that indicates whether the ith controller is active
$\boldsymbol{R}_{i,j}$	a ternary variable that indicates whether there is a link between the jth switch and the ith controller
$\boldsymbol{F}_{i,j}$	the traffic between the jth switch and ith controller
$\boldsymbol{T}_i$	the capacity of ith controller
$\boldsymbol{B}$	the sets of $\boldsymbol{B}_i$
$\boldsymbol{R}$	the sets of $\boldsymbol{R}_{i,j}$
$\boldsymbol{T}$	the sets of $\boldsymbol{T}_i$
$\boldsymbol{F}$	the sets of $\boldsymbol{F}_{i,j}$

The overall algorithm is shown in Algorithm 1. The inputs are $\boldsymbol{B}$, $\boldsymbol{R}$, $\boldsymbol{T}$, $\boldsymbol{F}$, D, α and β. The outputs are $\boldsymbol{B}$ and $\boldsymbol{R}$. The meanings of these parameters are summarized in Table I. We assume that a switch sends all its traffic to the master controller, so $\delta_{i,j}$ is not needed here. Step 1 checks the network load to decide further operations. When the deviation of controller load exceeds threshold D (Step 2), the network load is uneven and load balancing is performed (Step 3). When a controller overloads, i.e., the maximum load ratio of all controllers exceeds α (Step 5), the controller pool is expanded (Step 6). Finally, when the mean load ratio is less than β (Step 7), the controller pool is shrunk (Step 8).

Algorithm 1 Elastic Adaptive SDN

Input: $\boldsymbol{B}$, $\boldsymbol{R}$, $\boldsymbol{T}$, $\boldsymbol{F}$, α, β, D
Output: $\boldsymbol{B}$, $\boldsymbol{R}$

1: Check_Network($\boldsymbol{R}$, $\boldsymbol{T}$, $\boldsymbol{F}$);
2: **if** $(Mean\_Square\_Deviation > D)$ **then**
3: Balance($\boldsymbol{R}$, $\boldsymbol{T}$, $\boldsymbol{F}$, α, β);
4: **end if**
5: **if** $(Max\_Load\_Ratio > \alpha)$ **then**
6: Expand_Con_Pool($\boldsymbol{B}$, $\boldsymbol{R}$, $\boldsymbol{T}$, $\boldsymbol{F}$, α, β);
7: **else if** $(Mean\_Load\_Ratio < \beta)$ **then**
8: Shrink_Con_Pool($\boldsymbol{B}$, $\boldsymbol{R}$, $\boldsymbol{T}$, $\boldsymbol{F}$, α, β);
9: **end if**
10: return($\boldsymbol{B}$, $\boldsymbol{R}$);

B. Balancing Load Among Controllers

As discussed above, the network is dynamically changing and the loads of different controllers may be uneven. Thus, we need to balance the controller loads to improve network performance and avoid controller crash. We achieve load balancing by migrating a switch from the master controller to the slave controller, as long as the resulting traffic is much more balanced. There are two principles we must obey when migrating a switch: 1) a slave controller should not be overloaded when selected to be the new master controller of a switch, 2) the controller loads must be more balanced after a switch migration.

To find the switches that can be migrated, we visit the most-load controller, and check whether a migration is feasible for each switch attached to the controller. If so, we do the

migration and visit the next most-load controller until no migration can be performed. Fig.2 shows an example of such migration. The left part indicates the network before load balancing, and the right part indicates the network after load balancing. Before load balancing, only one switch attaches to the left controller, which is lightly loaded, and five switches attach to right controller, which is heavily loaded. In order to balance the network, we migrate the second and the third switches from their master controller (the right one) to their slave controller (the left one), and the loads are balanced.

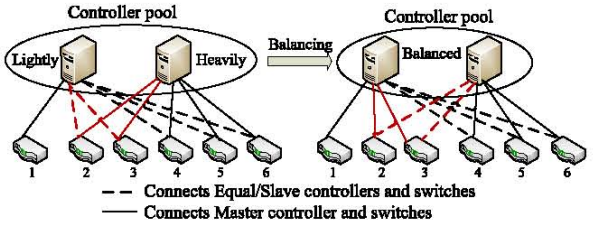

Figure 2: Balancing load among controllers

Algorithm 2 Balancing Load Among Controllers

Input: $\boldsymbol{B}$, $\boldsymbol{R}$, $\boldsymbol{T}$, $\boldsymbol{F}$, α, β, D
Output: $\boldsymbol{B}$, $\boldsymbol{R}$

1: M_i=$\sum_{j=1}^{m} \delta_{i,j} \mid \boldsymbol{R}_{i,j} \mid \boldsymbol{F}_j$; $(i \in (1, ..., n))$
2: $D1_{x,y}$ =$\mid M_x - M_y \mid$;{before balancing}
3: $D2_{x,y}$ =$\mid M_x - M_y \mid$;{after balancing}
4: $Sort\_Descending(M)$;
5: **for** $(i = 1 \rightarrow n$ && $j = 1 \rightarrow m)$ **do**
6: **if** $(M_i \geq \beta T_i)$ && $(\boldsymbol{R}_{i,j} = 1)$ **then**
7: **for** $(k = 1 \rightarrow n$ && $k! = i)$ **do**
8: **if** $((\boldsymbol{R}_{k,j} = -1)$ && $(\boldsymbol{B}_k = 1)$ &&$(M_k + \boldsymbol{F}_j \leq \alpha \boldsymbol{T}_k)$ && $(D2_{i,k} < D1_{i,k}))$ **then**
9: $\mathbf{R}_{k,j} = 1; \mathbf{R}_{i,j} = -1;$
10: Update($\boldsymbol{B}$, $\boldsymbol{R}$, $\boldsymbol{F}$);
11: **end if**
12: **end for**
13: **end if**
14: **end for**
15: return($\boldsymbol{B}$, $\boldsymbol{R}$);

Algorithm 2 shows the procedure of load balancing. The inputs include $\boldsymbol{B}$, $\boldsymbol{R}$, $\boldsymbol{T}$, $\boldsymbol{F}$, α, β, and D, and the outputs are $\boldsymbol{B}$ and $\boldsymbol{R}$. This algorithm first computes the total load amount of each controller c_i, namely $\boldsymbol{M}_i$, in Step 1. And $D1_{x,y}$ or $D2_{x,y}$ indicates the load difference value between controller x and controller y before or after load balancing (Steps 2, 3). Next, the controllers are sorted based on the load amount in descending order in Step 4. Then, according to the order of controllers, the algorithm checks the most-load controller and all the switches that are attached to the controller (Steps 5-14). For every switch, if its slave controller is not overloaded after the switch migration, and the load difference value between the master controller and the slave controller is smaller than before, then the algorithm migrates the switch to the slave controller from the master controller (Step 8). Then the algorithm updates $\boldsymbol{B}$, $\boldsymbol{R}$, $\boldsymbol{F}$ according to the migration (Step 9).

C. Shrinking Controller Pool

When the average network load is less than a given threshold, the controller pool is shrunk by inactivating some controllers to save power. To shrink the controller pool efficiently, the algorithm migrates all the switches from their master controller to their slave controllers. If one controller does not have any attached switches, it goes to an inactive state.

To find the controller that can be inactivated, we visit the least-load controller, and check whether all the switches attached to the controller can be migrated to their slave controllers. If so, we do the migration and inactivate the controller, then visit next least-load controller until no controller load is below the lower threshold βT_i. Fig.3 shows an example of shrinking the controller pool, the left part indicates the network before shrinking, in which there are two active lightly-loaded controllers in the controller pool. The right part shows the controller pool after shrinking. The third, the fourth and the fifth switches are migrated to the left controller, and the algorithm inactivates the right controller.

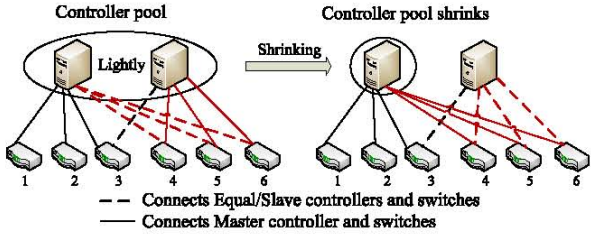

Figure 3: Shrinking controller pool

Algorithm 3 Shrinking Controller Pool

Input: $\boldsymbol{B}$, $\boldsymbol{R}$, $\boldsymbol{T}$, $\boldsymbol{F}$, α, β
Output: $\boldsymbol{B}$, $\boldsymbol{R}$, Num

1: $Num = 0$; {the number of controller to be inactivated}
2: $Sort\_Ascending(M)$;
3: **for** $(i = 1 \rightarrow n, j = 1 \rightarrow m)$ **do**
4: **if** $(M_i < \beta \boldsymbol{T}_i \&\& \boldsymbol{B}_i = 1 \&\& \boldsymbol{R}_{i,j} = 1)$ **then**
5: **for** $(k = 1 \rightarrow n \&\& k! = i)$ **do**
6: **if** $(\boldsymbol{R}_{k,j} = -1 \&\& \boldsymbol{B}_k = 1 \&\& M_k + \boldsymbol{F}_j \leq \alpha \boldsymbol{T}_k)$ **then**
7: Mark(j,k); {controller k has the highest load}
8: Update($\boldsymbol{B}$, $\boldsymbol{R}$, $\boldsymbol{F}$);
9: **end if**
10: **end for**
11: **if** $(All\_Switch\_Is\_Marked() = true)$ **then**
12: $Num = i$;
13: $\boldsymbol{B}_i = 0$;
14: Update($\boldsymbol{B}$, $\boldsymbol{R}$, $\boldsymbol{F}$);
15: **else** Roll_Back();
16: **end if**
17: **end if**
18: **end for**
19: return($\boldsymbol{B}$, $\boldsymbol{R}$, Num);

Algorithm 3 shows the pseudo-code of the shrinking procedure. The inputs include $\boldsymbol{B}$, $\boldsymbol{R}$, $\boldsymbol{T}$, $\boldsymbol{F}$, α and β, and the

outputs include $\boldsymbol{B}$, $\boldsymbol{R}$ and Num. Num indicates the controller number to be inactivated every time. When it equals to 0, it means that switch migration is finished or the network has no need to change (Step 1). According to the amount of the controller load, function $Sort\_Ascending(M)$ sorts the controllers in ascending order (Step 2). Then, the algorithm checks the least-load controller and all switches to it (Steps 3-18). In these steps, if the load of the controller c_i is less than $\beta \boldsymbol{T}_i$, the switches attached to the controller can be migrated to their slave controllers, and these slave controllers are not overloaded after switch migration, the algorithm marks these switch numbers and the slave controllers numbers (Steps 4-10). Note that, the $Mark()$ function selects the switch to be migrated and the controller that the switch should be reattached to (Step 7). If all the switches attached to the controller are marked, the controller can be inactivated (Steps 11-14), otherwise, the algorithm returns back to the initial state (Step 15).

D. Expanding Controller Pool

When the load of some controllers exceeds a given threshold, the controller pool is expanded by activating more controllers, to prevent the network from breaking down. We develop an algorithm that automatically activates controllers and allows switches to be migrated to these controllers. To expand the controller pool, the algorithm migrates the switches from the heavily-load controller to one inactivated controller.

In order to expand the controller pool, we visit the controller with the biggest load ratio, and check whether a migration to an inactivated slave controller is feasible for some switches attached to the controller. If so, we do the migration and activate the slave controller, then visit next controller with the biggest load ratio until no controller load ratio exceeds its upper threshold α. Fig.4 presents an example of expanding the controller pool, the left part indicates the network before expanding, in which there is just one active controller in the controller pool, which is overloaded. The right part shows the controller pool after expanding. The right controller has been activated, and the last two switches are migrated to the left controller.

Algorithm 4 shows the pseudo-code of the procedure. The inputs include $\boldsymbol{B}$, $\boldsymbol{R}$, $\boldsymbol{T}$, $\boldsymbol{F}$, α and β, and the outputs include $\boldsymbol{B}$ and $\boldsymbol{R}$. When a controller overloads (Step 1), the algorithm sorts the controllers in descending order according to the controller load ratio (Step 2). Then, it checks the controller with the biggest load ratio and all switches attached to the controller to find a feasible migration (Steps 3-14). If the load of the controller exceeds the upper threshold, we mark the most-load switch attached to the controller (Steps 4, 5). Note that, the switch which has the highest load is selected by the $Mark()$ function. For the switch marked, the algorithm checks its slave controller to find an inactivated slave controller (Steps 7-9).

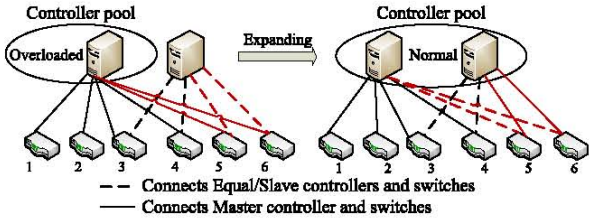

Figure 4: Expanding controller pool

Algorithm 4 Expanding Controller Pool

Input: $\boldsymbol{B}$, $\boldsymbol{R}$, $\boldsymbol{T}$, $\boldsymbol{F}$, α, β
Output: $\boldsymbol{B}$, $\boldsymbol{R}$

1: **if** $(Max\_Load\_Ratio > \alpha)$ **then**
2: $Load\_Ratio\_Sort\_Descending(M)$;
3: **for** $(i = 1 \rightarrow n, j = 1 \rightarrow m)$ **do**
4: **if** $((\boldsymbol{R}_{i,j} = 1))$ **then**
5: Mark(j); {switch j has the highest load}
6: **for** $(k = 1 \rightarrow n \;\&\&\; k! = i)$ **do**
7: **if** $((\boldsymbol{R}_{k,j} = -1)\&\&(\boldsymbol{B}_k = 0))$ **then**
8: $\boldsymbol{R}_{k,j} = 1; \boldsymbol{R}_{i,j} = -1; \boldsymbol{B}_k = 1;$
9: Update($\boldsymbol{B}$, $\boldsymbol{R}$, $\boldsymbol{F}$);
10: **end if**
11: **end for**
12: **end if**
13: **end for**
14: **end if**
15: return($\boldsymbol{B}$, $\boldsymbol{R}$);

E. Computational Complexity

Now we analyze the computational complexity of algorithms. The balancing procedure has three parts. The first part is to check all controllers whose load is more than βT one by one, so that we can to find a possible controller that can be balanced. The second part is to traverse all switches attached to the controller, which examines whether some switches attached to the controller can migrate to their slave controllers. The third part is to check all slave controllers to find new master controllers of these switches. For the three parts, the time complexity is $O(n^2m)$. The shrinking and expanding procedures are similar to the balancing procedure. The complexity of these algorithms is $O(n^2m)$ as well. However, during the execution of the three procedures, some circles never be executed according to the relationships between the controllers and switches, so the overall complexity is smaller than $O(n^2m)$.

V. Evaluation

We present the details of our algorithms implemented by c++ and report experimental results with the prototype.

A. Implementation

In order to enable the controller pool can expand and shrink dynamically, we develop an overall algorithm (see Section IV), which includes three procedures: balancing, shrinking and expanding procedure. When the network load is unevenly distributed, we need to balance the load of controllers. If the

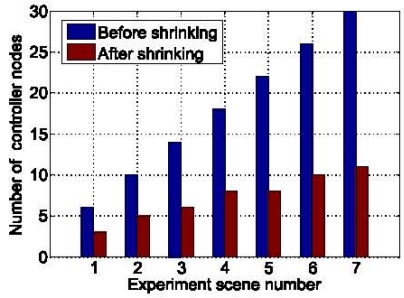

Figure 5: The controller number before and after shrinking the controller pool

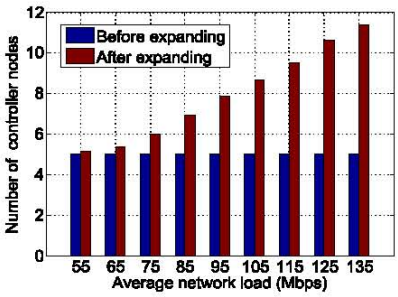

Figure 6: The controller number before and after expanding the controller pool

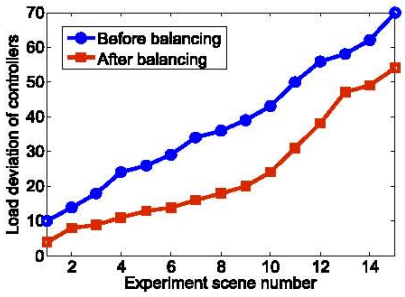

Figure 7: Load deviation of controllers

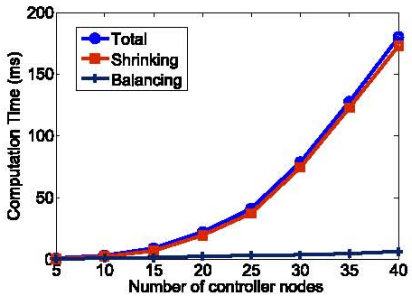

Figure 8: The initial number of controller nodes and computation time

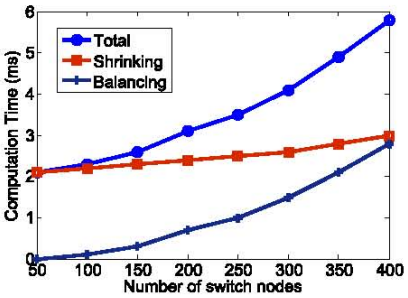

Figure 9: The initial number of switch nodes and computation time

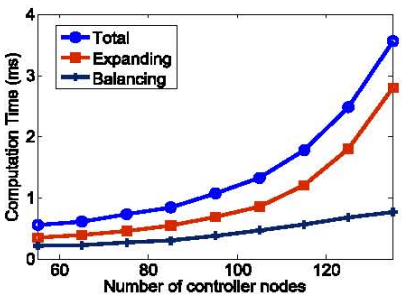

Figure 10: Expanding time

loads of some controllers exceed a given upper threshold, the controller pool expands according to the balancing results. And when the mean network load is less than a lower threshold, the controller pool shrinks by inactivating some controllers to save power.

B. Experiment Setup

Since Mininet with distributed SDN controllers cannot support runtime switch migration [11] now, we can only evaluate our algorithms with experiments conducted by our own simulators. In order to ensure the accuracy of the results, each experiment is performed more than thirty times. Since the mapping between controller number and switch number is not publicly available or the network is too small (i.e., Stanford backbone network just has two controllers and fourteen switches, which is too small for our experiments), topologies and traffic of the network are randomly generated in our experiments.

Next, we generate topologies and traffic matrixes by using the following two methods, respectively. 1) To begin with, we generate the network topologies randomly and ensure that each switch attaches to only one master controller. Then, we generate slave controllers randomly for each switch and guarantee that the number of slave controllers is less than $(n-1)$. 2) We generate traffic matrixes and then distribute flows randomly according to the generated topologies. The sum flow of each controller is not larger than αT. Here, we set α to 0.8, which ensures that controllers have enough capacity to handle traffic burst. Besides, the capability of all the controllers is the same, and we use small controller capability to validate our algorithm, and set it to 100Mbps [3]. The experimental results will be similar if we set a larger controller capability.

C. Results

*1) **The Effectiveness of Algorithm:*** Now we evaluate the effectiveness of our algorithm. First, the shrinking procedure, in this experiment, we set the mean network load to 25Mbps, which is less than βT. As shown in Fig.5, the blue bar shows the number of active controllers in the controller pool before shrinking, and the red one refers to the number of active controllers after the pool shrinks. The horizontal axis shows the experiment number, and vertical axis shows the number of active controllers in the controller pool. It can be observed that the number of the inactive controller increases with the increasing of the number of controllers in a network. In the experiment 7, the pristine active controller number is 30, but after shrinking, there are just 11 active controllers in the controller pool, and the number of active controller drops 63% compared with the initial number of the controllers.

Second, the expanding procedure, in this experiment, the controller number before expanding is 5, and the average network load increases slowly. Fig.6 gives the result of the expanding procedure. The blue bar shows the number of active

[3]Note that, this value is just for simplicity, it can be set to any appropriate value.

Table II: The results of optimal algorithm and greedy algorithm

Scene NO.	Controller number	Switch number	The number of controllers inactivated by optimal algorithm	The number of controllers inactivated by greedy algorithm	Optimal time (ms)	Greedy time (ms)
1	5	8	3	2.8	155.4	0
2	6	9	3.833	3.5	781	0.183
3	7	10	4.8	4.6	6644	0.4
4	8	11	5.333	4.833	100402.8	0.5
5	9	12	6	5.3	602601.4	0.8
6	9	13	6	5.4	1189108.3	1.1

controllers in the controller pool before expanding, and the red one stands for the number of active controllers after the pool expands. It can be observed that with the increasing of the average network load, the active controller number increases. When the average network load is 55Mbps, the controller number after expanding is just 5.15, and when the average network load is 135Mbps, the controller number after expanding is 11.39, which re-actives 6.39 controllers.

Third, the balancing procedure, in this experiment, we test the average network load before and after the balancing process. In Fig.7, the blue line shows the mean square deviation of controller load before balancing process, and the red line shows the same value after balancing process. The deviation is much small after balancing process. The largest reduction in mean square deviation that we observed in our experiments is around 25, and the average is 14. The largest deviation reduction in the experiment is 24, and the average reduction in the mean square deviation is 11. The results show that the balance part can make the network more balanced.

*2) **The Computation Overhead**:* In order to compute the computation overhead of the algorithm, we conduct experiments with different number of controllers and switches. First, we test the computation overhead in shrinking procedure, and then the expanding procedure.

In the first part of experiments in shrinking procedure, we test the relation between the controller number and the computation time. The number of switches is set to 200. It can be observed that the shrinking and total time grows faster than the speed of controller number growing (see Fig.8), but the balancing time grows slower. When the number of the controllers is 5, the shrinking time is just 0.2 ms, the balancing time is 0.2 ms and the total time is 0.8 ms. When the controller number increases to 40, the shrinking time reaches 173 ms, the balancing part is 3.4 ms, and total time is 177 ms. On average, one more controller in the network will incur 5 ms delays in shrinking the controller pool. In the second part of the experiments, we test the relation between the switch number and the computation time, and we set the number of controllers 10. As shown in Fig.12, all three lines grow slower than the speed of the switch number growing. When the number of the switches is 50, the shrinking time is just 2.1 ms, the balancing time is 0.01 ms and the total time is 2.11 ms. When the switch number increases to 400, the shrinking time is just 2.6 ms, the balancing part is 2.8 ms, and total time is 5.8 ms. On average, one more switch in the network will incur 10.5 us delays in shrinking the network. Overall, it can be observed that, in the shrinking procedure, the controller number is the most important to the computation time, and all the experiments can be finished in a few hundreds of milliseconds. The results are completely consistent with our analysis in Section IV, the computational complexity of the shrinking algorithm is $O(n^2m)$ (n is the controller number, and m is switch number).

Then we conduct experiments to compute the overhead of the expanding procedure. In the experiment, the initial number of the controller is 5, and the average network load increase from 55Mbps to 135Mbps, we compute the time of each expanding. Fig.10 shows the results of this experiment. We can find that, with the increasing of the average network load, the expanding and total time grows faster, while the balancing time grows slower. When the average network load is 55Mbps, the expanding time and balancing time are 0.35 ms and 0.21 ms , and when the average network load is 135Mbps, the expanding time and balancing time are 2.8 ms and 0.76 ms, respectively.

*3) **Comparison with Optimal Algorithm**:* In this part, we evaluate the performance of our algorithm by comparing with an optimal algorithm with simulations. The optimal algorithm lists all possible situations, and finds out the best solution. First we test the shrinking procedure, and then the expanding procedure.

We conduct an experiment to make comparison with the optimal algorithm in shrinking procedure. As shown in Table II[4], the forth column lists the number of the inactive controllers achieved by the optimal algorithm, and fifth column lists the number of inactive controllers computed by the greedy algorithm. It is obvious that the greedy result is very close to the optimal result (see Fig11), and the difference is bounded by 8%. As shown in Fig.12, the red line shows the time that the greedy algorithm consumes, and the blue one shows the delay that the optimal algorithm consumes. Since the time consumed by the optimal algorithm grows very quickly, we just compare two algorithms with some small scales of network topologies. When the number of controllers or switches increases, the time consumed by the optimal algorithm is several thousand times bigger than that consumed by our greedy algorithm. In particular, the difference enlarges significantly with expansion of the network. For example, if a network has ten controllers and forty switches, the time consumed by the optimal algorithm is more than one day, which is not acceptable for

[4]As the optimal algorithm is too time consuming, the experiment network is small.

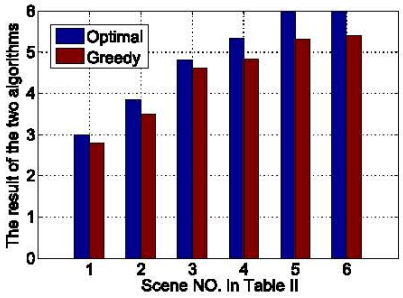

Figure 11: The controller number inactivated by two algorithms

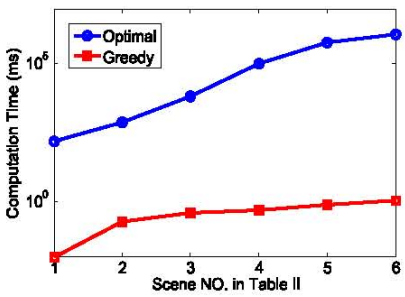

Figure 12: Computation time of two algorithms

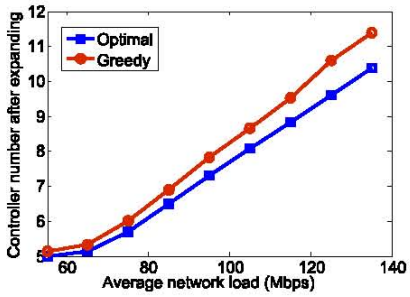

Figure 13: The controller number before and after expanding the controller pool

online controller selection. However, the time consumed by our greedy algorithm is only a few seconds.

Now we evaluate the effectiveness of our expanding procedure in activating controllers. In this experiment, the initializing active controller is 5, and with the increasing of the network load, the active controller number increases. The average network load increases from 55Mbps to 135Mbps. The Fig.13 shows that, when the average network load is 55Mbps, the expanding controller number of our procedure is 5.18, and the expanding controller number of the optimal algorithm is 5.15, which is not much different. But when then average network load is 135Mbps, the expanding controller number of the optimal algorithm is 10.384, and the expanding controller number of the greedy algorithm is 11.39, which has one controller difference. The largest difference with the optimal algorithm is 1.005, and our procedure only incurs 15.6% more controllers compared with the optimal algorithm.

VI. Conclusion

In this paper, we build a framework for an adaptive and elastic SDN network. We prove that the problem to select controllers in the elastic SDN is NP Hard, and develop a greedy algorithm to solve this problem. The experiment results show that our algorithm can effectively shrink and expand the controller pool in the networks. The number of inactive controllers computed by shrinking procedure averagely achieves around 92% of the optimal values. And the expanding algorithm can dynamically add necessary number of controllers to prevent the network breakdown, and only incurs 15.6% more controllers than the greedy algorithm. We hope that our algorithm can shed light on designing adaptive distributed SDN.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under grant No. 61402255 and No. 61202358, and in part by the R&D Program of Shenzhen under grant No. ZDSYS20140509172959989, No. JSGG20150512162853495, and No. Shenfagai[2015]986.

References

[1] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A Protection Architecture for Enterprise Networks. *USENIX*, 2006.

[2] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. *ACM SIGCOMM*, 2007.

[3] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A Clean Slate 4D Approach to Network Control and Management. *SIGCOMM CCR*, 2005.

[4] T. V. Lakshman, T. Nandagopal, R. Ramjee, K. Sabnani, and T. Woo. The SoftRouter Architecture. *ACM HOTNETS*, 2004.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling Innovation in Campus Networks. *ACM SIGCOMM*, 2008.

[6] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically Centralized? State Distribution Trade-offs in Software Defined Networks. *SIGCOMM HotSDN*, 2012.

[7] A. Tootoonchian and Y. Ganjali. HyperFlow: A Distributed Control Plane for Openflow. *INM/WREN*, 2010.

[8] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. *OSDI*, 2010.

[9] S. Schmid and J. Suomela. Exploiting Locality in Distributed SDN Control. *SIGCOMM HotSDN*, 2013.

[10] A. Akella T. Benson and D. Maltz. Network Traffic Characteristics of Data Centers in the Wild. *IMC*, 2010.

[11] A. Dixit, F. Hao, S. Mukherjee, T. V. Laskshman, and R. Kompella. Towards an Elastic Distributed SDN Controller. *SIGCOMM HotSDN*, 2013.

[12] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, and N. McKeown. Nox: Towards an Operating System for Networks. *SIGCOMM CCR*, 2008.

[13] OpenFlow White Paper: http://www.projectfloodlight.org/floodlight/.

[14] S.H. Yeganeh and Y. Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. *SIGCOMM HotSDN*, 2012.

[15] D. Erickson. The Beacon Openflow Controller. *SIGCOMM HotSDN*, 2013.

[16] Guo Z, Su M, Xu Y, Duan Z, Wang L, Hui S, and H. Jonat. Improving the performance of load balancing in software-defined networks through load variance-based synchronization. *Computer Networks*, 68(95-109), 2015.

[17] OpenFlow White Paper: https://www.opennetworking.org/sdn-resources/-sdn-library/whitepapers.

[18] OpenFlow: https://www.opennetworking.org/sdn-resources/onf-specifications/openflow.

[19] X. Feng. *Computational complexity*. University of Electronic Science and Technology: China Machine Press, 2005.