Traffic Load Balancing Schemes for Devolved Controllers in Mega Data Centers

Xiaofeng Gao, *Member, IEEE*, Linghe Kong, Weichen Li, Wanchao Liang, Yuxiang Chen, and Guihai Chen, *Senior Member, IEEE*

Abstract—In most existing cloud services, a centralized controller is used for resource management and coordination. However, such infrastructure is gradually not sufficient to meet the rapid growth of mega data centers. In recent literature, a new approach named devolved controller was proposed for scalability concern. This approach splits the whole network into several regions, each with one controller to monitor and reroute a portion of the flows. This technique alleviates the problem of an overloaded single controller, but brings other problems such as unbalanced work load among controllers and reconfiguration complexities. In this paper, we make an exploration on the usage of devolved controllers for mega data centers, and design some new schemes to overcome these shortcomings and improve the performance of the system. We first formulate *Load Balancing problem for Devolved Controllers* (LBDC) in data centers, and prove that it is NP-complete. We then design an f-approximation for LBDC, where f is the largest number of potential controllers for a switch in the network. Furthermore, we propose both centralized and distributed greedy approaches to solve the LBDC problem effectively. The numerical results validate the efficiency of our schemes, which can become a solution to monitoring, managing, and coordinating mega data centers with multiple controllers working together.

1 Introduction

In recent years, data center has emerged as a common infrastructure that holds thousands of servers and supports many cloud applications and services such as scientific computing, group collaboration, storage, financial applications, etc. This fast proliferation of cloud computing has promoted a rapid growth of mega data centers used for commercial purposes. Companies such as Amazon, Cisco, Google, and Microsoft have made huge investments to improve Data Center Networks (DCNs).

Typically, a DCN uses a centralized controller to monitor the global network status, manage resources and update routing information. For instance, Hedera [1] and SPAIN [2] both adopt such a centralized controller to aggregate the traffic statistics and reroute the flows for better load balancing.

However, for large-scale DCN with thousands of racks (usually in a mega data center), the utilization of a centralized controller suffers from many problems such as the issues of scalability[3] and availability. Driven by the unprecedent objectives of improving the performance and scale of DCNs, researchers try to deploy multiple controllers in such networks [4], [5], [6], [7], [8]. The concept of *devolved controllers* is thereby introduced for the first time in [4], in which they used dynamic flow [5] to illustrate the detailed configuration. Devolved controllers are a set of controllers that collaborate as an single omniscient controller, as a similar scheme in [9]. However, none of the controllers has the complete information of the whole network. Instead, every controller only maintains a portion of the pairwise multipath information beforehand, thus reducing the workload significantly.

Recently, software-defined networking (SDN) as proposed by OpenFlow [10] has been touted as one of the most promising solutions for future Internet. SDN is characterized by two distinguished features: decoupling the control plane from the data plane and providing programmability for network application development [11]. From these features we can divide the DCN flow control schemes into two layers: the lower layer focuses on traffic management and virtual machine (VM) migrations, which could relieve the intensive traffic in hot spots; the upper layer coordinates the control rights of switches among controllers, which deals with the load imbalance problem in a hierarchical manner. Combining the two layers together, we could better improve the system performance and reduce the load imbalance problem greatly.

For the lower layer control, there are mature and well-developed methods to handle the flow control and VM migration at present [12], [13], [14], [15]. While for the upper layer control, managing the DCNs by devolved controllers gradually becomes a hot topic in recent years due to the expansion of the scale of DCNs. Similarly, if switches are relatively busy regionally, then the controller monitoring this region becomes a hot spot, which could be harmful for the system. Many relevant studies emphasis on the imbalanced load problem for devolved controllers [4], [11], [16], but none of them give a clear formulation of controller imbalance problem and analyze the performance of their solutions. This leads to our concern on the imbalanced load issue for devolved controllers to better control the traffic and manage the network.

- *X. Gao, L. Kong, W. Li, and G. Chen are with the Department of Computer Science and Engineering, Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: {gao-xf, linghe.kong, eizo.lee, gchen}@cs.sjtu.edu.cn.*
- *W. Liang and Y. Chen are with the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15289. E-mail: {wanchaol, yuxiang1}@cs.sjtu.edu.cn.*

Manuscript received 2 July 2015; revised 1 June 2016; accepted 1 June 2016. Date of publication 9 June 2016; date of current version 18 Jan. 2017. Recommended for acceptance by X. Wang.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2579622

Motivated by these concerns, in this paper we propose a novel scheme to manage devolved controllers. In our scheme, each controller monitors the traffics of a part of the switches locally. When traffic load imbalance occurs, some of them will migrate a portion of their monitored work to other controllers so that the workload can be kept balanced dynamically. We define this problem as *Load Balancing problem for Devolved Controllers* (LBDC). We prove that LBDC is NP-complete, which might not be easily solved within polynomial time. Then we design multiple solutions for LBDC, including a linear programming with rounding approximation, three centralized greedy algorithms, and one distributed greedy algorithm. Using these solutions, we can dynamically balance the traffic load among controllers. Such methods can reduce the occurrence of traffic hot spots significantly, which will degrade network performance. These schemes can also improve the availability and throughput of DCN, supporting horizontal scaling and enhancing responsiveness of clients' requests. In all, the main contributions of this paper are as follows:

1) We design and implement a traffic load balancing scheme using devolved controllers, which eliminates the scalability problem and balances the traffic load among multiple controllers. All these controllers are configured based on their physical placements, which is more realistic and makes the whole network more effective and reliable.
2) We prove the NP-completeness of LBDC, and design an f-approximation algorithm to obtain the solution. We also come up with both centralized and distributed heuristics for workload migration between controllers in dynamic situations. The distributed algorithm is scalable, stable, and more appropriate for real-world applications, especially for large-scale DCNs.
3) We evaluate our algorithms with various experiments. Numerical results validate our design's efficiency. To the best of our knowledge, we are the first to discuss workload balancing problem among multi-controllers in DCNs, which has both theoretical and practical significance.

This paper is the extended version of our conference version [17]. Based on the short conference version, we add a randomized rounding for the linear programming, as well as two novel centralize migration algorithms under limited conditions. Additionally, we develop a new evaluation section and obtain more reliable and precise results by various numerical experiments.

The rest of the paper is organized as follows. Section 2 presents the system architecture and problem statement; Sections 3 and 4 give our solutions to LBDC. Section 5 exhibits our performance evaluation and proves the effectiveness of our algorithms. Section 6 introduces the related works; Finally, Section 7 concludes the paper.

2 Problem Statement

Traffic in DCN can be considered as Virtual Machine communication. VMs in different servers collaborate with each other to complete designated tasks. In order to communicate between VMs, communication flow will go through several switches.

TABLE 1
Definition of Terms

Term	Definition
S, s_i	switch set with n switches: S=$\{s_1, \ldots, s_n\}$
$w(s_i)$	weight of s_i, as the no. of out-going flows.
$PC(s_i)$	potential controllers set of the ith switch.
$rc(s_i)$	the real controller of the ith switch.
C, c_i	controller set with m controllers: $C = \{c_1, \ldots, c_m\}$
$w(c_i)$	weight of c_i, as the sum of $RS(c_i)$'s weight.
$PS(c_i)$	potential switches set of the ith controller.
$RS(c_i)$	real Switches set of the ith controller.
$AN(c_i)$	adjacent node set (1-hop neighbors) of c_i.

Based on the concept of OpenFlow [10], there is a flow table in each switch, storing the flow entries to be used in routing. One responsibility of a controller is to modify these flow tables when communication occurs. Every controller has a corresponding routing component and it may be composed of several hierarchical switches, including Top of Rack (TOR) Switches, Aggregation Switches, and Core Switches. These switches are used for communication within the data center. Furthermore, every rack has a server called *designated server* [18], which is responsible for aggregating and processing the network statistics for the rack. It is also in charge of sending the summarized traffic matrices to the network controller, using a mapping program which converts the traffic of this rack (server-to-server data) into ToR-to-ToR messages. Once a controller receives these data, it will allocate them to a routing component which computes the flow reroute and replies to the new flow messages sent to the controller. Then the controller installs these route information to all associated switches by modifying their flow tables. Since this paper is not concerned with routing, we omit the details of table computing and flow rerouting.

Now we will define our problem formally. In a typical DCN, denote s_i as the ith switch, with the corresponding traffic weight $w(s_i)$, which is defined precisely as the number of out-going flows. Note that this weight does not include the communication within the ToRs. Next, given n switches $S = \{s_1, \cdots, s_n\}$ with their weights $w(s_i)$ and m controllers $C = \{c_1, \cdots, c_m\}$, we want to make a weighted m-*partition* for switches such that each controller will monitor a subset of switches. The weight of a controller $w(c_i)$ is the weight sum of its monitored switches. Due to physical limitations, assume every s_i has a potential controller set $PC(s_i)$ and it can only be monitored by controller in $PC(s_i)$. Every c_i has a potential switch set $PS(c_i)$ and it can only control switches in $PS(c_i)$. After the partition, the real controller of s_i is denoted by $rc(s_i)$ and the real switch subset of c_i is denoted by $RS(c_i)$. The symbols used in this paper are listed in Table 1.

To keep the performance of network management, each controller should finally have almost the same amount of workload. Otherwise, if the hot switches always require routing information from the same controller, it will become the bottleneck of the network. To precisely quantify the balancing performance among devolved controllers, we define *Standard Deviation* of the partitions' weights as the metric, denoted by

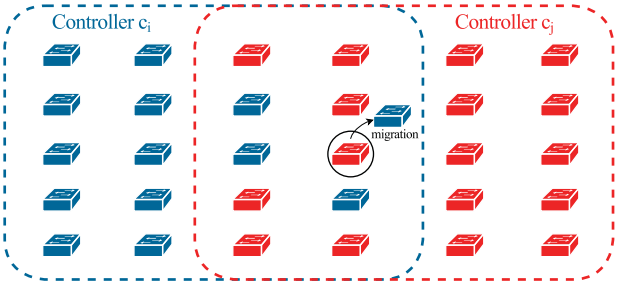


Fig. 1. An example of regional balancing migration.

$\sigma = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(w(c_i) - \overline{w(c)})^2}$, where $\overline{w(c)}$ is the average weight of all controllers. If the traffic flow varies as the system running, the weight of controller c_i may grow explosively, making it unbalanced comparing with other controllers. Then in this condition, we must regionally migrate some switches in $RS(c_i)$ to other available controllers, in order to reduce its workload and keep the whole network traffic balanced.

Then our problem becomes balancing the traffic load among m partitions in real time environment, and migrating switches among controllers when the balance is broken. We define this problem as *Load Balancing problem for Devolved Controllers*. In our scheme, each controller can dynamically migrate switches to or receive switches from logically adjacent controllers to keep the traffic load balanced.

Fig. 1 illustrates the migration pattern. Here Controller c_j dominates 17 switches (as red switches) and Controller c_i dominates 13 switches (as blue switches). Since the traffic between c_i and c_j is unbalanced, c_j is migrating one of its switches to c_i.

Let $x_{ij} = \begin{cases} 1 & \text{If } c_i \text{ monitors } s_j \\ 0 & \text{otherwise} \end{cases}$. Then the LBDC problem can be further formulated as the following programming:

$$\min \quad \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(\sum_{j=1}^{n} w(s_j)\cdot x_{ij} - \overline{w(c)}\right)^2} \tag{1}$$

$$s.t. \quad \overline{w(c)} = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n} w(s_j)\cdot x_{ij} \tag{2}$$

$$\sum_{i=1}^{m} x_{ij} = 1, \quad \forall 1 \le j \le n \tag{3}$$

$$x_{ij} = 0, \quad \text{if } s_j \notin PS(c_i) \text{ or } c_i \notin PC(s_j), \forall i, j \tag{4}$$

$$x_{ij} \in \{0,1\} \quad \forall i, j. \tag{5}$$

Here, Eqn. (1) is the objective standard deviation. Eqn. (2) calculates the average weight of all controllers. Eqn. (3) means that each switch should be monitored by exactly one controller. Eqn. (4) is the regional constraints, and Eqn. (5) is the integer constraints.

Theorem 1. *LBDC is NP complete.*

Proof. We will prove the NP completeness of LBDC by considering a decision version of the problem, and showing a reduction from PARTITION problem [19]. An instance of PARTITION is: given a finite set A and a $size(a) \in \mathbb{Z}^+$ for each $a \in A$, is there a subset $A' \subseteq A$ such that $\sum_{a\in A'} size(a) = \sum_{a\in A\setminus A'} size(a)$? Now we construct an instance of LBDC. In this instance there are two controllers c_1, c_2 and $|A|$ switches. Each switch s_a represents an element $a \in A$, with weight $w(s_a) = size(a)$. Both controllers can control every switch in the network ($PS(c_1) = PS(c_2) = \{s_a \,|\, a \in A\}$). Then, given a YES solution A' for PARTITION, we have a solution $RS(c_1) = \{s_a \,|\, a \in A'\}$, $RS(c_2) = \{s_a \,|\, a \in A\setminus A'\}$ with $\sigma = 0$. The reverse part is trivial. The reductions can be done within polynomial time, which completes the proof. □

Next we presents our solutions for the LBDC. We implement the schemes within OpenFlow framework, which makes the system comparatively easy to configure and implement. It changes the devolved controllers from a mathematical model into an implementable prototype. Furthermore, our schemes are topology free, which is scalable for any DCN topology such as Fat-Tree, BCube, Portland, etc.

3 Linear Programming and Rounding

Given the traffic status of the a current DCN with devolved controllers, we can solve the LBDC problem using the above programming. To simplify this programming, we will then transfer it into a similar integer programming. Firstly, we can convert the standard deviation to average of absolute values:

$$\min \frac{1}{m}\sum_{i=1}^{m}\left|\sum_{j=1}^{n} w(s_j)\cdot x_{ij} - \overline{w(c)}\right|. \tag{6}$$

We rewrite Eqn. (6), and obtain an integer programming as follows:

$$\min \quad \frac{1}{m}\sum_{i=1}^{m} y_i \tag{7}$$

$$s.t. \quad y_i \ge \sum_{j=1}^{n} w(s_j)\cdot x_{ij} - \overline{w(c)} \tag{8}$$

$$y_i \ge \overline{w(c)} - \sum_{j=1}^{n} w(s_j)\cdot x_{ij} \tag{9}$$

$$\overline{w(c)} = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n} w(s_j)\cdot x_{ij} \tag{10}$$

$$\sum_{i=1}^{m} x_{ij} = 1, \quad \forall 1 \le j \le n \tag{11}$$

$$x_{ij} = 0, \text{ if } s_j \notin PS(c_i) \text{ or } c_i \notin PC(s_j), \forall i, j \tag{12}$$

$$x_{ij} \in \{0,1\} \quad \forall i, j. \tag{13}$$

In general, integer programmings may not be easily solved in polynomial time, so we adopt *relaxation* to transfer our integer programming into a linear programming (LP). Then we can acquire a fractional solution and then round it

to a feasible solution of the original programming. To obtain the linear programming, we replace Eqn. (13) with $x_{ij} \geq 0$ $(\forall i, j)$.

After solving this LP, we can discover a feasible solution to LBDC by a deterministic rounding [20], which is stated in Algorithm 1.

Algorithm 1. Deterministic Rounding (LBDC-DR)

```
1 foreach switch s_j do
2     Search the solution space of LP:
3     Let ℓ = arg max_i{x_ij | 1 ≤ i ≤ m};
4     if ∃ several maximal x_ij then
5         Let ℓ = arg min_i{w(c_i) | each max x_ij}
6     Round x_ℓj = 1;
7     for c_i ≠ c_ℓ do
8         Round x_ij = 0;
```

For instance, if a switch s_j has $x_{1j} = 0.2, x_{2j} = 0.7$, $x_{3j} = 0.1$ in the solution space of LP, then according to Algorithm 1, we can round $x_{2j} = x_{\ell j} = 1$, and $x_{1j} = x_{3j} = 0$. Next, we prove that this solution is feasible for LBDC.

Theorem 2. *LBDC-DR (Algorithm 1) results in a feasible solution for the integer programming of LBDC.*

Proof. According to LBDC-DR, for each s_j, we only round the maximum $x_{ij} = 1, \forall 1 \leq i \leq m$, and all other x_{ij}'s are equal to 0. Then each switch is monitored by only one controller and no switches are in the idle state. Thus we can get a feasible solution for the integer programming. □

Now let us analyze the performance of LBDC-DR. We define Z^*, Z^{LP}, and Z^R as the solutions of the integer programming, the solution of the linear programming, and the solution after the rounding process respectively. Then define f as the maximum number of controllers in which any switch potentially appears. More formally, $f = \max_{i=1,\ldots,n} |PC(s_i)|$.

We claim that LBDC-DR is an f-approximation. To prove it, we first prove the following two lemmas.

Lemma 1. $\overline{w(c)^{LP}} = \overline{w(c)^*} = \overline{w(c)^R}$

Proof: From the definition of the original $\overline{w(c)}$, the ideal weight of each controller is the sum of the weight of all switches divided by the number of controllers. This definition is suited for all the solution space, thus we can conclude that $\overline{w(c)^{LP}} = \overline{w(c)^*} = \overline{w(c)^R} = \frac{1}{m}\sum_{i=1}^{n} w(s_i)$. □

Lemma 2. $x_{ij}^R \leq x_{ij}^{LP} \cdot f$

Proof. We have the constraint $\sum_{i=1}^{m} x_{ij}^{LP} = 1\,(\forall 1 \leq j \leq n)$. Also according to LBDC-DR, $x_{\ell j}^{LP}$ is the largest of all $x_{ij}^{LP} (\forall 1 \leq i \leq m)$, then by the Pigeonhole principle, we must have $x_{\ell j}^{LP} \cdot f \geq 1$. Because for each switch s_j, $x_{\ell j}^R$ equals to 1 and others equal to zero, which is less than or equal to the corresponding LP solution times the f factor. Then for any controller c_i, we have $x_{ij}^R \leq x_{ij}^{LP} \cdot f$. □

According to all above lemmas, we can then obtain the following theorem:

Theorem 3. *LBDC-DR is an f-approximation algorithm.*

Proof. Since the linear programming is a relaxation of the integer programming, we have $Z^{LP} \leq Z^*$. Also we have $Z^* \leq Z^R$ because the solution of LBDC-DR is feasible according to Theorem 2, while Z^* denotes the optimal solution.

Because $\overline{w(c)}$ represents the ideal weight of each controller, it must be the same in all the solutions according to Lemma 1. Therefore we let $\overline{w} = \overline{w(c)}$. From $Z^{LP} \leq Z^*$ we can derive

$$\frac{1}{m}\sum_{i=1}^{m}\left|\sum_{j=1}^{n} w(s_j)\cdot x_{ij}^{LP} - \overline{w}\right| \leq \frac{1}{m}\sum_{i=1}^{m}\left|\sum_{j=1}^{n} w(s_j)\cdot x_{ij}^{*} - \overline{w}\right|.$$

Since we already know the inequality $|x| - |y| \leq |x - y| \leq |x| + |y|$, we can get the following relationship:

$$\frac{1}{m}\sum_{i=1}^{m}\left|\sum_{j=1}^{n} w(s_j)\cdot x_{ij}^{LP}\right| \leq \frac{1}{m}\sum_{i=1}^{m}\left|\sum_{j=1}^{n} w(s_j)\cdot x_{ij}^{*}\right| + 2\overline{w}.$$

Then the approximation ratio can be obtained by the following inequations:

$$\begin{aligned}
\frac{1}{m}\sum_{i=1}^{m}\left|\sum_{j=1}^{n} w(s_j)\cdot x_{ij}^{R} - \overline{w}\right| &\leq \frac{1}{m}\sum_{i=1}^{m}\left(\left|\sum_{j=1}^{n} w(s_j)\cdot x_{ij}^{R}\right| + \overline{w}\right) \\
&\leq \frac{1}{m}\sum_{i=1}^{m}\left|\sum_{j=1}^{n} w(s_j)\cdot x_{ij}^{LP}\cdot f\right| + \overline{w} \\
&\leq f\cdot\frac{1}{m}\sum_{i=1}^{m}\left|\sum_{j=1}^{n} w(s_j)\cdot x_{ij}^{*}\right| + (1+2f)\overline{w} \\
&= f\cdot OPT + (1+2f)\overline{w}.
\end{aligned}$$

Thus LBDC-DR is an f-approximation. □

Another idea for rounding an optimal fractional solution is to view the fractions as probabilities, flipping coins with these biases and rounding accordingly. We will show how this idea leads to an $O(\log n)$ factor randomized approximation for the LBDC problem. We then present our LBDC-Randomized Rounding (LBDC-RR) algorithm as described below.

First, we claim that our LBDC problem can be described in another way as the definition and properties of set cover: Given a universe U of n switch elements, S is a collection of subsets of U, and $S = \{S_1, \ldots, S_n\}$. And there is a cost assignment function $c : S \to \mathbb{Z}^+$. Find the subcollection of S with the minimum deviation that covers all the switches of the universal switch set U.

We will show that each switch element is covered with constant probability by the controllers with a specific switch set, which is picked by this process. Repeating this process $O(\log n)$ times, and picking a subset of switches if it is chosen in any of the iterations, we get a set cover with high probability, by a standard coupon collector argument. The expected minimum deviation of cover (or say controller-switch matching) picked in this way is $O(\log n) \cdot OPT_f \leq O(\log n) \cdot OPT$, where OPT_f is the cost of an optimal solution to the LP-relaxation.

Algorithm 2 shows the formal description of LBDC-RR.

Algorithm 2. Randomized Rounding (LBDC-RR)

1 Let $\mathbf{x} = \mathbf{p}$ be an optimal solution to the LP;
2 **foreach** *set* $S_i \in \mathbf{S}$ **do**
3 Pick S_i with probability x_{S_i}
4 **repeat** ▷ get $c \log n$ subcollections
5 Pick a subcollection as a min-cover
6 **until** *execute* $c \log n$ *times*
7 Compute the union of subcollections in C.

Next let us compute the probability that a switch element $a \in U$ is covered by C. Suppose that a occurs in k sets of S. Let the probabilities associated with these sets be $p_1, \ldots, p_k$. Since a is fractionally covered in the optimal solution, $p_1 + p_2 + \cdots + p_k \geq 1$. Using elementary calculus, it is easy to show that under this condition, the probability that a is coverd by C is minimized when each of the p_i's is $1/k$. Thus,

$$\mathbf{Pr}[a \text{ is covered by } C] \geq 1 - \left(1 - \frac{1}{k}\right)^k \geq 1 - \frac{1}{e},$$

where e is the base of natural logarithms. Hence each element is covered with constant probability by C.

To get a complete switch set cover, we can independently pick $c \log n$ such subcollections. And then we compute their union, say $C^{'}$, where c is a constant such that $(\frac{1}{e})^{c \log n} \leq \frac{1}{4n}$.

Then we can obtain the following probability,

$$[\mathbf{Pr}[a \text{ is not covered by } C^{'}] \leq \left(\frac{1}{e}\right)^{c \log n} \leq \frac{1}{4n}.$$

Summing up all switch elements $a \in U$, we get

$$\mathbf{Pr}[C^{'} \text{ is not a valid switch set cover}] \leq n \cdot \frac{1}{4n} \leq \frac{1}{4}.$$

Therefore the LBDC-RR algorithm is efficient and we can solve the LBDC problem using linear programming and randomized rounding.

4 Algorithm Design

Using Linear programming and rounding, we can perfectly solve LBDC theoretically. However, it is usually time consuming and impractical to solve an LP in real-world applications. Thus, designing efficient and practical heuristics for real systems is essential. In this section, we will propose a centralized and a distributed greedy algorithm for switch migration, when the traffic load becomes unbalanced among the controllers. We then describe OpenFlow based migration protocols that we use in this system.

4.1 Centralized Migration

Centralized Migration is split up into two phases. The first phase is used for configuring and initializing the DCN. As the traffic load changes due to various applications, we have to come to the second phase for dynamical migration among devolved controllers.

Fig. 2 illustrates the general workflow of Centralized Migration, which includes Centralized Initialization and Centralized Regional Balanced Migration.

Centralized Initialization. First we need to initialize the current DCN, and assign switches to the controllers in its potential controller set. We design a centralized initialization algorithm (LBDC-CI) for the initialization process. In order to get rid of the dilemma where we have to select from conflict switches or controllers, we first present the *Break Tie Law*.

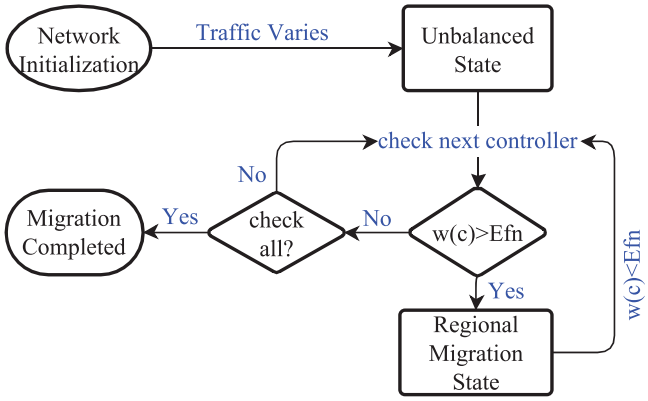


Fig. 2. Dynamic load balancing workflow of LBDC.

Break Tie Law. (1) When choosing s_i from S, we select the one with the largest weight. If several switches have the same weight, the one with the smallest $|PC(s_i)|$ is preferred. If there are still several candidates, we randomly choose one. (2) When choosing c_i from C, we select the one with the minimum weight. If several controllers have the same weight, the one with the smallest $|RS(c_i)|$ is preferred. If there are still several candidates, we choose the closer controller by physical distance. Finally, if we still cannot make a decision, just randomly choose one.

Then we design LBDC-CI as shown in Algorithm 3.

Algorithm 3. Centralized Initialization (LBDC-CI)

Input : S with $w(s_i)$; C with $w(c_i)$;
Output: An m-Partition of S to C
1 $RemList = \{ s_1 , s_2, \ldots , s_n\}$;
3 **while** $RemList \neq \emptyset$ **do**
4 Pick s_i from $RemList$;
5 Let $\ell = \arg\min_j \{w(c_j) \mid c_j \in PC(s_i)\}$;
6 Assign s_i to c_ℓ (by break Tie Law);
7 Remove s_i from $RemList$;

LBDC-CI needs to search the $RemList$ to assign the switches. This process takes running time $O(n)$. *While* loop will be executed once for each switch in $RemList$, which takes $O(m)$. Hence in the worst case the running time is $O(mn)$. If we use a priority heap to store the $RemList$, we can improve the performance and reduce the overall running time to $O(m \log n)$.

As the system runs, traffic load may vary frequently and will influence the balanced status among devolved controllers. Correspondingly, we have to begin the second phase and design the centralized migration algorithm (LBDC-CM) to alleviate the situation.

Centralized Regional Balanced Migration. During the migration process, we must assess when the controller needs to execute a migration. Thus we come up with a threshold and an effluence to judge the traffic load balancing status of the controllers. Here we define Thd as the threshold and Efn as the effluence. If the workload of a controller is lower than or equal to Thd, it becomes relatively idle and available to

receive more switches migrated from those controllers with workload overhead. If the workload of a controller is higher than Efn, it is in an overload status and should assign its switches to other idle controllers. Some measurement studies [21] of data center traffic have shown that data center traffic is expected to be linear. Thus we set the threshold according to the current traffic sample and the historical records, by imitating Round-Trip Time (RTT) and Timeout of TCP [22]. This linear expectation uses two constant weighting factors α and β, depending on the traffic features of the data center, where $0 \leq \alpha \leq 1$ and $\beta > 1$.

(1) *Naive LBDC-CM*. We will first raise a naive algorithm for LBDC-CM. We will run naive LBDC-CM periodically and divide the running time of the system into several rounds. We use Avg_{last} and Avg_{now} to represent the average workload of the last sample round and the current sample round. These two parameters are used together to decide when to start and stop the migration. In each round, we sample the current weight of each controller, and calculate $Avg_{now} = \sum_{i=1}^{m} w(c_i)/m$. In all, the Linear Expectation can be computed as follows:

$$\begin{cases} Thd = \alpha \cdot Avg_{now} + (1-\alpha) \cdot Avg_{last} \\ Efn = \beta \cdot Thd. \end{cases} \tag{14}$$

The core principle of LBDC-CM is migrating the heaviest switch to the lightest controller greedily. Algorithm 4 describes the details. Note that $AN(c_i)$ denotes the neighbor set of c_i.

Algorithm 4. Centralized Migration (LBDC-CM)

Input: S with $w'(s_i)$; C with $w'(c_i)$;
$PendList = OverList = \{\emptyset\}$;
Step 1: Add $c_i \rightarrow OverList$ if $w'(c_i) > Efn$;
Step 2: Find c_m of max weight in $OverList$;
if $\exists c_n \in AN(c_m) : w'(c_n) < Thd$ **then**
  repeat
    Pick $s_m \in RS(c_m)$ of max weight;
    if $\exists c_f \in AN(c_m) \cap PC(s_m) : w'(c_f) < Thd$ **then** Send $s_m \rightarrow c_f$
    else Ignore the current s_m in c_m
  until $w'(c_m) \leq Thd$ *or all* $w'(c_f) \geq Thd$;
  if $w'(c_m) > Efn$ **then** move c_m to $PendList$
  else remove c_m from $OverList$
else
  Move c_m from $OverList$ to $PendList$;
Step 3: Repeat Step 2 until $OverList = \{\emptyset\}$;
Let $OverList = PendList$, Repeat Step 2 until $PendList$ becomes stable;
Step 4: Now $PendList$ has several connected components CC_i $(1 \leq i \leq |CC|)$;
foreach $CC_i \in CC$ **do**
  Search the $\bigcup_{c_j \in CC_i} AN(c_j)$;
  Compute $avg_{local} = \frac{w'(CC_i \cup AN(CC_i))}{|CC_i| + |AN(CC_i)|}$;
  while $w'(c_j) \geq \gamma \cdot avg_{local} : c_j \in CC_i$ **do**
    Migrate $s_{max} \in RS(c_j)$ to $c_{min} \in AN(CC_i)$;
  remove $c_j \in CC_i$ from $PendList$;
Step 5: Repeat Step 4 until $PendList$ is stable.

The naive LBDC-CM consists of five steps. In Step 2, it searches the $OverList$ to find c_m, which takes $O(m)$. Next, it repeatedly migrates switches from the $OverList$ to corresponding controllers, which takes $O(mn)$. Step 3 invokes Step 2 for several times until the $OverList$ is empty and makes the $PendList$ become stable, which takes $O(m^2 n)$. Step 4 and Step 5 balance the $PendList$ locally as Step 2 and 3. In the worst case, the running time is $O(m^2 n)$. By using a priority heap to store the $OverList$ and $PendList$, we can reduce the time complexity to $O(mn \log m)$.

(2) Limited LBDC-CM. In our naive version, we simply suppose that all controllers have unlimited processing abilities. However, in real conditions, the performance of each controller will vary a lot. Thus, although naive LBDC-CM balances every controller with almost the same traffic load after several rounds, some of them will work in an overloaded state. For example, consider the following condition: there are three controllers c_1, c_2, c_3. The maximum capacity for c_1 is λ, for c_2 is 2λ and for c_3 is 4λ. The total weight of all switches in this system is 6λ. If our naive LBDC-CM works perfectly, then each controller will have a load of 2λ in the end. Definitely, c_1 works in an overloaded status, and will become the bottleneck of the system. Yet c_3 only makes use of 50 percent of its maximum abilities. Thus in fact, the naive LBDC-CM only balances the value of load among devolved controllers, instead of balancing the performance of processing traffic load.

Correspondingly, we design an improved algorithm as *limited LBDC-CM*. To reconfigure the system when it is unbalanced, we still need a threshold parameter and an effluence parameter for each controller. But now different controllers will have different parameter values, and we use two sets to store them: $ThdList = \{Thd_1, \ldots, Thd_m\}$ and $EfnList = \{Efn_1, \ldots, Efn_m\}$. For controller c_i, we use Des^i_{now} to denote its deserved workload of the current round, and use Des^i_{last} to denote the deserved workload of the last round. Then these parameters are computed as follows:

$$\begin{cases} Des^i_{now} = \frac{\sum_{i=1}^{n} w'(s_i)}{\sum_{j=1}^{m} w_m(c_j)} \cdot w_m(c_i) \\ Thd_i = \alpha \cdot Des^i_{now} + (1-\alpha) \cdot Des^i_{last} \\ Efn_i = \beta \cdot Thd_i. \end{cases} \tag{15}$$

Here the maximum load that controller c_i can hold is denoted as $w_m(c_i)$. Meanwhile, we modify the definition of standard deviation, and define σ' as Relative Weight Deviation: $\sigma' = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(\sum_{j=1}^{n} w(s_j) \cdot x_{ij} - Des^i_{now}\right)^2}$. We believe this reference index is more appropriate. We use Des^i_{now} in Relative Weight Deviation to evaluate limited LBDC-CM and LBDC-CM with switch priority. We use Avg_{now} to replace Des^i_{now} in RWD to evaluate naive LBDC-CM and LBDC-DM.

According to Eqn. (15), the procedure of the limited LBDC-CM is very similar as the naive LBDC-CM in Algorithm 4. The only difference comes from the comparison steps, when to judge whether a controller is overloaded, we need to compare $w'(c_i)$ to its local Efn_i

and Thd_i. Furthermore, in Step 4 (Line 18 of Algorithm 4), we need to calculate $\epsilon_{local} = \frac{w'(CC_i \cup AN(CC_i))}{w_m(CC_i \cup AN(CC_i))}$, and consider candidate controller c_j if $w'(c_j) > \gamma \cdot \epsilon_{local} \cdot w_m(c_j)$ instead of $\gamma \cdot avg_{local}$.

Limited LBDC-CM uses the current load ratio of each controller other than the value of the current weight, to judge whether the devolved controllers are unbalanced. Thus, we only need to calculate the average percentage of resources utilized in the system, and migrating switches from the controllers that have high percentages to those with low percentages. The time complexity is the same as naive LBDC-CM, which takes $O(m^2n)$, and can be reduced to $O(mn \log m)$ using priority heap. For space complexity, we need to use several lists to store the following parameters: the weight of a switch, the current of a controller, the maximum capacity of a controller, the threshold and effluence of each controller, as well as the PendList and the OverList. Each of them requires a linear array to store, which takes $O(n)$. We also need two matrices to store the potential mapping and real mapping between controllers and switches, which takes $O(n^2)$. Thus, the space complexity is $O(n^2)$.

(3) LBDC-CM with Switch Priority. Our scheme of limited LBDC-CM can work well in a comparative intense structure. That is to say, if the distance between a switch and all its potential controllers are close enough, so that migrating switch s_i from controller c_1 to controller c_2 will not influence the processing speed of messages, then limited LBDC-CM will have a good performance. However, in some distributed data centers that have a very sparse structure, it is better to attach a switch to its nearby controllers. Meanwhile, as we have mentioned, the performance of controllers in the network system may be very different. Some of the controllers may have strong computing capacities, and thus can process messages in a higher speed. In real network systems, sometimes we hope certain messages or certain areas can have a higher priority in the whole structure, and we want to allocate switches in this region to those strong controllers to increase the value of the system.

Thus, though for a certain switch s_i, it can be attached to any controller c_j in its potential controller set $PC(s_i)$, the performance, or the value of the whole system may vary according to the real mapping strategy. If the value we get for s_i monitored by c_1 is θ, and for s_i monitored by c_2 is 2θ, then its better to distribute s_i to c_2, if the current load of both controllers are below their thresholds. Thus we come up with *LBDC-CM with switch priorities*. In this scheme, each switch has a value list, which stores the value of each mapping between this switch and its potential controllers. We want to balance the traffic load of the network and make the whole value as large as possible. In LBDC-CM, we use v_{ij} to denote the value we can get by attaching switch s_i to controller c_j. These values are stored in a matrix *Value*, and if c_j is not in the potential controller set of s_i, then $v_{ij} = 0$. We also consider the maximum capacity of each controller as we did in the limited LBDC-CM.

The implementation of this algorithm is quite similar to limited LBDC, except that we changed the migration scheme used in Step 2 of limited LBDC-CM, which is shown in Algorithm 5.

Algorithm 5. LBDC-CM with Switch Priority

```
1  Step 2: Find c_m ∈ OverList with max w'(c_m)/w_m(c_m);
2  if ∃c_n ∈ AN(c_m) : w'(c_n) < Thd_n then
3      repeat
4          if ∃c_f ∈ AN(c_m): w'(c_f) < Thd_f then
5              Sort PS(c_m) by v_if: s_i ∈ PS(c_m);
6              Pick s_k with max v_kf in c_m, and pick
               max s_k to break tie;
7              Send s_k → c_f;
8      until w'(c_m) ≤ Thd_m or all w'(c_f) ≥ Thd_f;
9      if w'(c_m) > Efn_m then move c_m to PendList;
10     else remove c_m from OverList
11 else
12     Move c_m from OverList to PendList;
```

In this scheme, we add the process of sorting the switch list according to the value matrix, which will take $O(\log n)$ if we use heap sorting. Thus the time complexity is $O(n \log m \log n)$ if we use a priority heap to store the PendList and the OverList. And the space complexity is still $O(n^2)$ since we need some matrices to store the value and the mapping relations.

4.2 Distributed Migration

The centralized algorithm is sometimes unrealistic for real-world applications, especially for large data center with regional controller. It is time consuming and complicated for a devolved controller to get the global information of the whole system. Thus it is natural to design a practical and reliable distributed algorithm [23]. We assume a synchronous environment to deploy our algorithm. For the distributed algorithm, it is still divided into two phases.

Distributed Initialization. During this phase, we assign each switch a corresponding controller randomly. By sending control messages to the controller's potential switch set, the controller can determine the correct assignment. Algorithm 6 shows the distributed initialization process.

Algorithm 6. Distributed Initialization (LBDC-DI)

```
1 Send "CONTROL" message to my own PS(c_my)
2 s_i reply the first "CONTROL" message with "YES", all other
  messages after that with "NO".
3 Move s_i with "YES" from PS(c_my) to RS(c_my).
4 Wait until all the switches in PS(c_my) reply, and then
  terminate.
```

The correctness of LBDC-DI is easy to check. After initialization, we then design the distributed migration algorithm (LBDC-DM) to balance the workload of the system dynamically.

Distributed Regional Balanced Migration. In the second phase, the controller uses the threshold and the effluence to judge its status and decide whether it should start the migration. Since in a distributed system, a controller can only obtain the information of its neighborhood, the threshold is not a global one that suits for all the

controllers, but an independent value which is calculated by each controller locally. Also the algorithm runs periodically for several rounds. In each round, each controller samples $AN(c_i)$ and applies the Linear Expectation again:

$$\begin{cases} Avg = \frac{\sum_{c_k \in AN(c_i)+c_i} w(c_k)}{|AN(c_i)|+1} \\ Thd = \alpha \cdot Avg_{now} + (1-\alpha) \cdot Avg_{last} \\ Efn = \beta \cdot Thd. \end{cases} \quad (16)$$

LBDC-DM aims at monitoring the traffic status of itself by comparing current load with its threshold. When the traffic degree is larger than Efn, it enters the sending state and initiates a double-commit transaction to transfer heavy switches to nearby nodes.

Algorithm 7 shows the distributed migration procedure.

Algorithm 7. Distributed Migration (LBDC-DM)

Sending Mode: (when $w'(c_{my}) \geq Efn$)
1 **if** $\exists c_i \in AN(C_{my})$ *in receiving or idle* **then**
2 add $c_i \rightarrow RList$ (receiving $>$ idle).
3 **repeat**
4 Pick s_{max} with max weight, refer $PC(s_{max})$, find $c_j \in RList$ with min weight, send "HELP$[c_{my}, s_{max}]$" to c_j, then check response:
5 **if** response="ACC" **then**
6 send "MIG$[c_{my}, s_{max}]$" to c_j
7 **else if** response="REJ"**then**
8 remove c_j from $RList$, find next c_j, send "HELP" again, check response.
9 Check response, delete s_{max} when receiving "CONFIRM" message, terminate.
10 **until** $w'(c_{my}) \leq Efn$;
Receiving Mode: (when $w'(c_{my}) \leq Thd$)
11 When receiving "HELP" messages:
12 **repeat**
13 receive switches for c_j and return "ACC";
14 **until** $w(c_j) + s_{max} \geq Thd$;
15 Now all "HELP" messages will reply "REJ"
16 When receiving "MIG" message:
17 $s_{max} \rightarrow c_j$, send back "CONFIRM" message;
Idle Mode: (when $Thd \leq w'(c_{my}) \leq Efn$)
18 When receiving "HELP" message:
19 **repeat**
20 receive switches for c_j and return "ACC";
21 **until** $w(c_j) + s_{max} > Efn$;
22 When receiving "MIG", migrate as above;

The main difference between the centralized and the distributed migration is that the former can get information in a global view and make better decisions, but it will also cause more processing times and will become potential bottleneck of the system. On the contrary, for the controllers in the distributed version, each controller will only collect information from its neighborhood and can only make proper migrations within this area. Though the distributed version cannot obtain a global optimal balancing status, it is more practical to deploy in real systems. Meanwhile, it can efficiently avoid the problem in the centralized scheme that the collapse or mistake of the central processor will affect problem of the system.

Their difference is also shown in the definition of the threshold (Thd). In the centralized version, the threshold is affected by the utilizing ratio of the whole system, which is the same for each controller in the centralized scheme. While in the distributed version, the threshold of each controller is calculated by its local information instead of the global information, and the deserved utilizing ratio of each controller is actually different from each other.

By using our distributed scheme, for conditions shown in Fig. 1, controller c_i and controller c_j will get the information of each other, calculate its Thd and Efn value, and decide its status. If controller c_i is in the sending mode and controller c_j is in the receiving mode, then c_i will migrate some of its dominating switches to c_j according to Algorithm 7.

4.3 OpenFlow Based Migration Protocol

To maintain a well-balanced operating mode when a peak flow appears, switches should change the roles of their current controllers while controllers should change their roles by sending Role-Request messages to the switches. These operations require the system to perform a switch migration operation. However, there is no such mechanism provided in the OpenFlow standard. OpenFlow 1.3 defines three operational modes for a controller: master, slave, and equal. Both master and equal controllers can modify switch state and receive asynchronous messages from the switch. Next, we design a specific protocol to migrate a switch from its initial controller to a new controller.

It is assumed that we are not able to manipulate the switch in our migration protocol design, while it is technically feasible to update the OpenFlow standard to implement our scheme. However, there are two additional issues. First, the OpenFlow standard clearly states that a switch may process messages not necessarily in the same order as they are received, mainly to allow multi-threaded implementations. Second, the standard does not specify explicitly whether the order of messages transmitted by the switch remains consistent between two controllers that are in master or equal mode. We need this assumption for our protocol to work, since allowing arbitrary reordering of messages between two controllers will make an already hard problem significantly harder.

Our protocol is built on the key idea that we need to first create a single trigger event to stop message processing in the first controller and start a same message in the second one. We can exploit the fact that *Flow-Removed* messages are transmitted to all controllers operating in the equal mode. We therefore simply insert a dummy flow into the switch from the first controller and then remove the flow, which will provide a single trigger event to both the controllers in equal mode to signal handoff. Our proposed migration protocol for migrating switch s_m from initial controller c_i to target controller c_j works in four phases as shown below.

Phase 1. Change the role of target c_j to equal mode. Here, controller c_j is first transitioned to the equal mode for switch s_m. Initially master c_i initiates this phase by sending a start migration message to c_j on the controller-to-controller channel. c_j sends the *Role-Request* message to s_m informing that it is an equal. After c_j receives a

Role-Reply message from s_m, it informs the initial master c_i that its role change is completed. Since c_j changes its role to equal, it can receive asynchronous messages from other switches, but will ignore them. During this phase, c_i remains the only master and processes all messages from the switch guaranteeing liveness and safety.

Phase 2. Insert and remove a dummy flow. To determine an exact instant for the migration, c_i sends a dummy *Flow-Mod* command to s_m to add a new flow table entry that does not match any incoming packets. We assume that all controllers know this dummy flow entry a priori as part of the protocol. Then, it sends another *Flow-Mod* command to delete this entry. In response, the switch sends a *Flow-Removed* message to both controllers since c_j is in the equal mode. This Flow-Removed event provides a time point to transfer the ownership of switch s_m from c_i to c_j, after which only c_j will process all messages transmitted by s_m. An additional barrier message is required after the insertion of the dummy flow and before the dummy flow is deleted to prevent any chance of processing the delete message before the insert. Note that we do not assume that the *Flow-Removed* message is received by c_i and c_j simultaneously, since we assume that the message order is consistent between c_i and c_j after these controllers enter the equal mode, meaning that all messages before *Flow-Removed* will be processed by c_i and after this will be processed by c_j.

Phase 3. Flush pending requests with a barrier. While c_j has assumed the ownership of s_m in the previous phase, the protocol is not complete unless c_i is detached from s_m. However, it cannot just be detached immediately from s_m since there may be pending requests at c_i that arrives before the *Flow-Removed* message. This appears easily since we assume the same ordering at c_i and c_j. So all c_i needs to do is processing all messages arrived before *Flow-Removed*, and committing to s_m. However, there is no explicit acknowledgment from the switch that these messages are committed. Thus, in order to guarantee all these messages are committed, c_i transmits a *Barrier-Request* and waits for the *Barrier-Reply*, only after which it signals end migration to the final master c_j.

Phase 4. Assign controller c_j as the final master of s_m. c_j sets its role as the master of s_m by sending a *Role-Request* message to s_m. It also updates the distributed data store to indicate this. The switch sets c_i to slave when it receives the *Role-Request* message from c_j. Then c_j remains active and processes all messages from s_m for this phase.

The above migration protocol requires six round-trip times to complete the migration. But note that we need to trigger migration only once in a while when the load conditions change, as we discussed in the algorithm design subsections.

5 Performance Evaluation

In this section, we evaluate the performance of our centralized and distributed protocols. We consider the case where traffic demand changes and examine whether the metric of balanced workload controllers is minimized. We also take the number of migrated switches into consideration. Furthermore, we check how different parameters will influence the results.

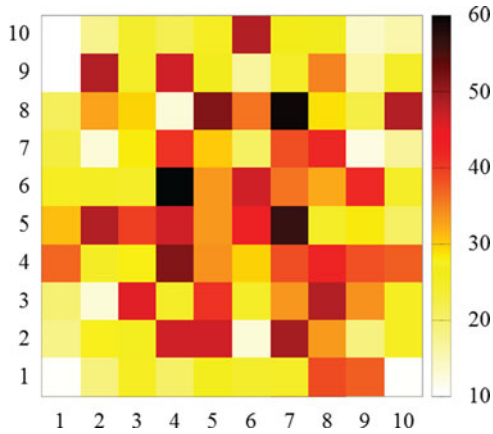


Fig. 3. Initial state.

5.1 Environment Setup

We construct simulations by Python 2.7 to evaluate the performance of our designs. We place 10,000 switches and 100 controllers in a 100×100 m^2 square. Switches are evenly distributed in this square, say, each switch is 1 m away from any of its neighbors. The controllers are also evenly distributed and each controller is 10 m away from its neighbor. Each controller can control all the switches within 30 m, and can communicate with other controllers within the range of 40 m. We assume the weight of each switch follows Pareto distribution with its parameter $\alpha_p = 3$. We build a small simulation to choose the most appropriate α, β and γ, so that the environment we build can be very close to the real situation, in terms of the traffic condition, workload of controllers, and migration frequency, etc. [13], [14], [15], [24]. Thus we set $\alpha = 0.7, \beta = 1.5, \gamma = 1.3$ as default configuration.

5.2 System Performance Visualization Results

We use the default configuration described above to test the performance of the system. We first apply initialization and change the traffic demands dynamically to emulate unpredictable user requests. Then we apply naive LBDC-CM and other variants to alleviate the spot congestion. We use relative weight deviation to evaluate the performance of our algorithms.

We examine the performance of our four algorithms. Consider a DCN with 10×10 controllers locating as an square array. At the beginning of a time slot, the weights of switches are updated and then we run the migration algorithms. The weight of switches follows Pareto distribution with $\alpha_p = 3$. Fig. 3 indicates the system initial traffic states, Different color scale represents different working state of a controller. The darker the color is, the busier the controller works. Figs. 4, 5, 6 and 7 illustrate the performance of the naive LBDC-CM, limited LBDC-CM, Priori LBDC-CM and LBDC-DM respectively. We can see that after the migration, the whole system becomes more balanced.

Actually, the performance of LBDC-DM is poor when the number of the controllers is relatively limited. This phenomenon is attributed to the system setting that one controller can only cover switches within 30 m. When the number of controllers is few, more switches should be controlled by one particular controller without many choices. As the

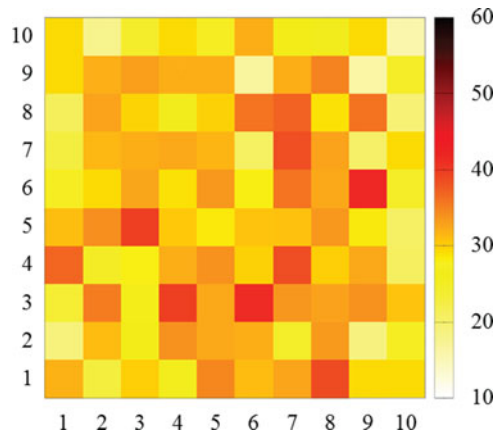


Fig. 4. Naive LBDC-CM migration.

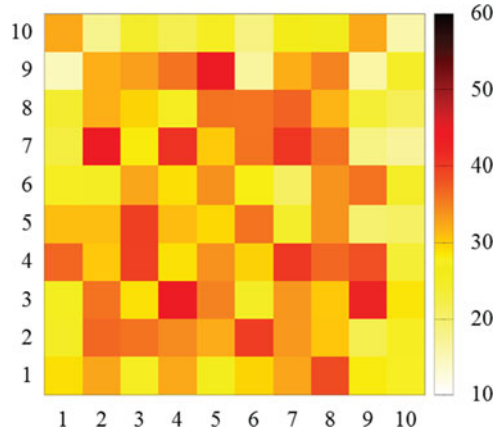


Fig. 5. Limited LBDC-CM migration.

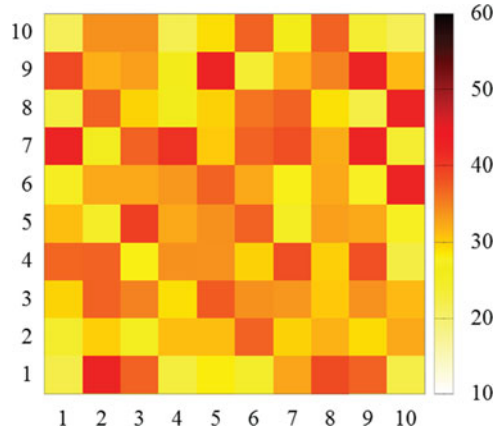


Fig. 6. Priori LBDC-CM migration.

number of the controller increases, LBDC-DM can achieve a better performance and a higher improvement ratio.

Intuitively, increasing the number of controllers may increase the deviation, but it may lead to less migration frequency. To balance the number of controllers and the migration frequency, we need to carefully set α, β, and γ values. If there are sufficient controllers to manage the whole system, we can adjust the three parameters such that the system will maintain a stable state longer. While if the number of controller reduces, we have to raise the migration threshold to fully utilize controllers. The effect on these parameters are further discussed in Section 5.4.

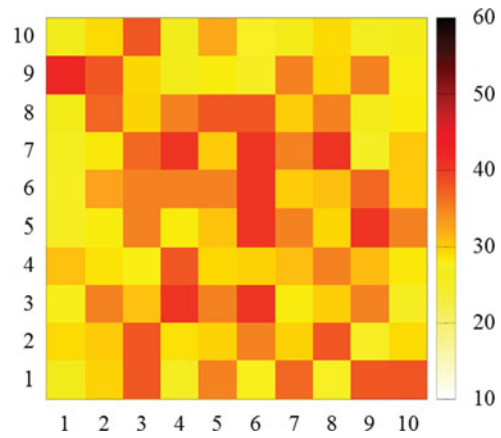


Fig. 7. LBDC-DM migration.

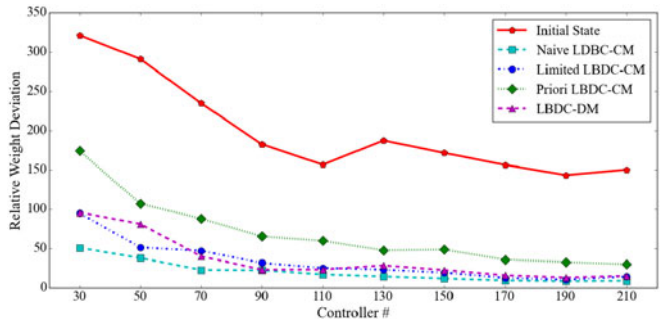


Fig. 8. Relative weight deviation protocol comparison.

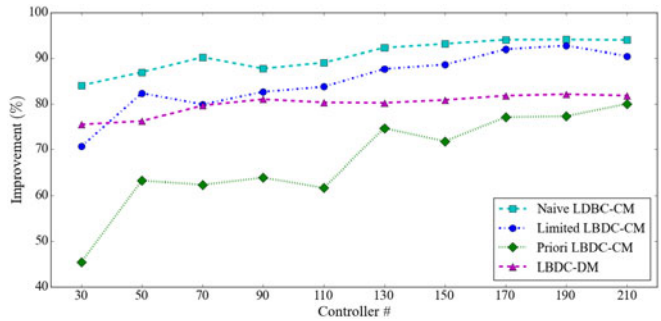


Fig. 9. Performance improvement for different protocols.

5.3 Horizontal Protocol Performance Comparison

We designed three variations of LBDC-CM: naive LBDC-CM, which is the simplest and applicable to most of the cases. While if the controllers are heterogeneous or the switches have a space priority to its closest controller physically, then we can implement limited LBDC-CM or LDBC-CM with switch priority respectively. Finally we have a distributed LBDC-DM protocol. Now let us compare the performance of the four migration protocols.

Comparison on Number of Controllers. First, we vary the number of controllers from 30 to 210 with a step of 20 and check the change of relative weight deviation of the system. The simulation results are shown in Figs. 8 and 9. We compare the relative weight deviation of the initial bursty traffic state and the state after the migration. We find that after the migration, the relative weight deviation of all the controllers decreases. It depicts that our four protocols improve the

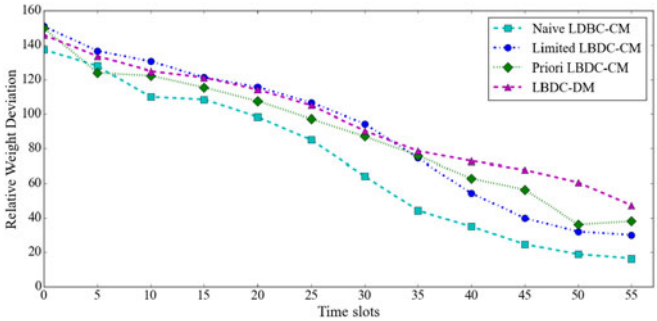


Fig. 10. Relative weight deviation without traffic changes.

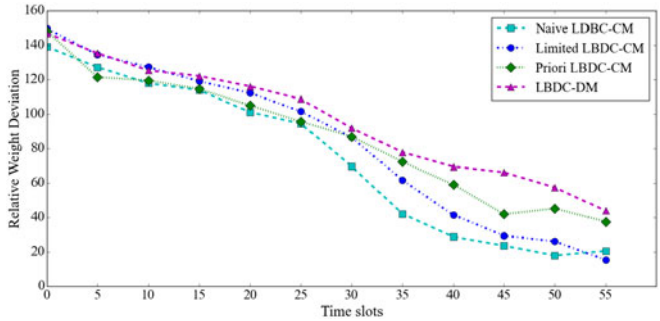


Fig. 12. Relative weight deviation as traffic changes.

system performance significantly compared with the initial state, whether in the relative weight deviation part or in the improvement part. As the number of controllers increases, the improvement ratio is also increasing. It is quite intuitive that more controllers will share jobs to reach a balanced state. Both figures show that our algorithm has a pretty good performance when the number of controllers grows, which indicates that our scheme is suitable for mega data centers.

The naive LBDC-CM performs the best because it considers all possible migrations from a global prospective. It is even better than the performance of the LBDC-DM, but the difference between them is decreasing as the number of the controllers increases. It is better if we add more controllers to the network to achieve a balanced traffic load. In reality we may only run the other protocols such as the LBDC-DM, limited LBDC-CM and LBDC-CM with switch priority. For the limited LBDC-CM, the maximum workload of controllers also follows Pareto distribution with $\alpha_p = 3$, and we amplify it with a constant to make sure the total traffic load not exceed the capacity of all controllers. For LBDC-CM with switch priority, we allocate a value to each mapping of a switch and a controller, which is inversely proportional to their distance, we can also see that it has a significant growth as the number of controller increases. Overall we can conclude that all of the four protocols performs quite well in balancing the workload of the entire system.

Run-Time Performance w.r.t Static Traffic Loads. Figs. 10 and 11 show the relative weight deviation and migrated switch number w.r.t. the four protocols at different time slot under the condition that the global traffic load is not changed all the time (the weight of each switch is constant). We can see that the relative weight deviation is decreasing, but the values of limited LBDC-CM and LBDC-CM with switch priority are higher than that of the naive LBDC-CM. This is because through limited LBDC-CM and LBDC-CM with switch priority, each controller has a different upper bound, which will influence the migration. For example, if some switches can only be monitored by a certain controller, and that controller is overloaded, then it will cause a high relative weight deviation since we cannot remove the switches to other controllers. In addition, controllers in LBDC-CM with switch priority even have a preference when choosing potential switches. In terms of migrated switch numbers, we can see that with time goes by, all four protocols remain stable on the number of migrated switches. LBDC-DM has the lowest number of migrated switches because of its controllers can only obtain a local traffic situation, resulting in the relatively low frequency in migrating switches.

Run-Time Performance w.r.t. Dynamic Traffic Loads. Figs. 12 and 13 show the relative weight deviation and migrated switch number w.r.t. the four protocols at different time slot under the condition that the global traffic load is changed dynamically (the weight of each switch is dynamic). Even if the traffic load is changing at different time slots, the migrated switch number stays in a relatively stable status. If controller c_1 is overloaded, it will release some dominating switches to its nearby controllers. However, if in the next round, the switches that monitored by those controllers gain higher traffic load and make the nearby controllers overloaded, then the switches may be sent back to controller c_1. Thus, to avoid such frequent swapping phenomenon, we can set an additional parameter for each switch. If its role has been changed in the previous slot, then it will be stable at current state.

We may also consider the deviation of load balancing among switches to better improve the system performance. Since we consider the balancing problem among

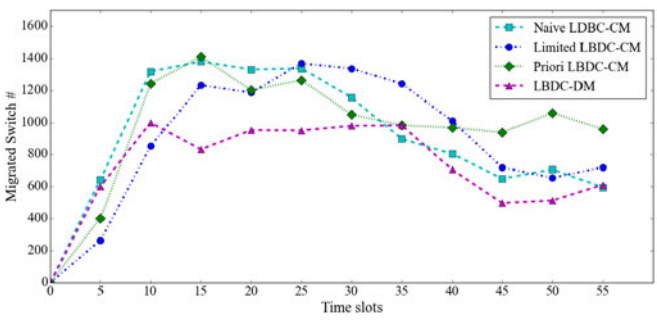


Fig. 11. Migrated switch without traffic changes.

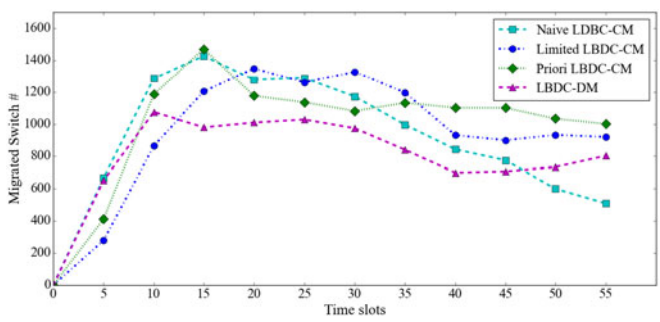


Fig. 13. Migrated switch as traffic changes.

TABLE 2
Influence of α, β and γ Factor

α	β	γ	Initial	LBDC-CM	Rate	Switch #
0.25	1.15	1.15	181.87	14.41	92.08	6,344
0.25	1.15	1.35	188.54	16.90	91.04	6,236
0.25	1.35	1.15	193.81	11.83	93.90	6,536
0.25	1.35	1.35	182.76	16.18	91.15	6,224
0.75	1.15	1.15	196.73	12.01	93.90	6,705
0.75	1.15	1.35	187.62	17.29	90.79	6,244
0.75	1.35	1.15	178.77	15.46	91.35	6,305
0.75	1.35	1.35	181.01	14.29	92.11	6,231

controllers, which is like the "higher level" of balancing problem among switches, we can implement some load balancing strategies among switches [25], [26], [27], [28] and combine the two-layers together to achieve a better solution.

5.4 Parameter Specification

Next we explore the impact of the threshold parameters α, β, γ. Here α is a parameter to balance conservativeness and radicalness, β is a crucial parameter which decides whether to migrate switches or not in four protocols, and γ is used in Step 4 of LBDC-CM. We examine the impact of changing α, β and γ altogether. Table 2 lists the statistics for α ranging between 0.25 and 0.75, β ranging between 1.15 and 1.35, γ ranging between 1.15 and 1.35. The improvement rate and the number of migrated switches is mostly decreasing as β increases, which is actually correct according to the definition of the threshold.

6 Related Work

As data center becomes more important in industries, there have been tremendous interests in designing efficient DCNs [1], [2], [29], [30], [31], [32]. Also, the effects of traffic engineering have been proposed as one of the most crucial issues in the area of cloud computing.

The existing DCN usually adapts a centralized controller for aggregation, coordination and resource management [1], [2], [10], [31], which can be energy efficient and can leverage the failure of using a global view of traffic to make routing decisions. Actually, using a centralized controller makes the design simpler and sufficient for a fairly large DCN.

However, using a single omniscient controller introduces scalability concerns when the scale of DCN grows dramatically. To address these issues, researchers installed multiple controllers across DCN by introducing *devolved controllers* [4], [5], [6], [7], [8], [33] and used dynamic flow as an example [5] to illustrate the detailed configuration. The introduction of devolved controllers alleviates the scalability issue, but still introduce some additional problems.

Meanwhile, several literatures in devising distributed controllers [6], [7], [8] have been proposed for SDN [34] to address the issues of scalability and reliability, which a centralized controller suffers from. Software-Defined Networking is a new network technology that decouples the control plane logic from the data plane and uses a programmable software controller to manage network operation and the state of network components.

The SDN paradigm has emerged over the past few years through several initiatives and standards. The leading SDN protocol in the industry is the OpenFlow protocol. It is specified by the Open Networking Foundation (ONF) [35], which regroups the major network service providers and network manufacturers. The majority of current SDN architectures, OpenFlow-based or vendor-specific, relies on a single or master/slave controllers, which is a physically centralized control plane. Recently, proposals have been made to physically distribute the SDN control plane, either with a hierarchical organization [36] or with a flat organization [7]. These approaches avoid having a SPOF and enable to scale up sharing load among several controllers. In [34], the authors present a distributed NOX-based controllers interwork through extended GMPLS protocols. Hyperflow [7] is, to our best knowledge, the only work so far also tackling the issue of distributing the OpenFlow control plane for the sake of scalability. In contrast to our approach based on designing a traffic load balancing scheme with well designed migration protocol under the OpenFlow framework, HyperFlow proposes to push (and passively synchronize) all state (controller relevant events) to all controllers. This way, each controller thinks to be the only controller at the cost of requiring minor modifications to applications.

HyperFlow [7], Onix [34], and Devolved Controllers [4] try to distribute the control plane while maintaining logically centralized using a distributed file system, a distributed hash table and a pre-computation of all possible combinations respectively. These approaches, despite their ability to distribute the SDN control plane, impose a strong requirement: a consistent network-wide view in all the controllers. On the contrary, Kandoo [36] proposes a hierarchical distribution of the controllers based on two layers of controllers. Meanwhile, DevoFlow [37] and DAIM [38] also solve these problems by devolving network control to switches.

In addition, [39] analyzes the trade-off between centralized and distributed control states in SDN, while [40] proposes a method to optimally place a single controller in an SDN network. Authors in [41] also presented a low cost network emulator called Distributed OpenFlow Testbed (DOT), which can emulate large SDN deployments. Recently, Google has presented their experience with B4 [42], a global SDN deployment interconnecting their data centers. In B4, each site hosts a set of master/slave controllers that are managed by a gateway. The different gateways communicate with a logically centralized Traffic Engineering (TE) service to decide on path computations. Authors in [6] implemented migration protocol on current OpenFlow standard. Thus switch migration become possible and we are able to balance the workload dynamically by presenting the following schemes to overcome the shortcomings as well as improve system performance from many aspects.

7 Conclusion

With the evolution of data center networks, the usage of a centralized controller has become the bottleneck of the entire system, and the traffic management problem also

becomes serious. In this paper, we explored the implementation of *devolved controllers*, used it to manage the DCN effectively and alleviate the imbalanced load issues.

We first defined the *Load Balancing problem for Devolved Controllers* and proved its NP-completeness. We then proposed an f-approximation solution, and developed applicable schemes for both centralized and distributed conditions. The feature of traffic load balancing ensures scaling efficiently. Our performance evaluation validates the efficiency of our designs, which dynamically balances traffic load among controllers, thus becoming a solution to monitor, manage, and coordinate mega data centers.

Acknowledgments

This work has been supported in part by the China 973 project (2014CB340303), the Opening Project of Key Lab of Information Network Security of Ministry of Public Security (The Third Research Institute of Ministry of Public Security) Grant number C15602, the Opening Project of Baidu (Grant number 181515P005267), National Natural Science Foundation of China (Nos. 61472252, 61133006, and 61303202), the Open Project Program of Shanghai Key Laboratory of Data Science (No. 201609060001) and China Postdoctoral Science Foundation (Nos. 2014M560334 and 2015T80433).

References

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. 7th USENIX Conf. Netw. Syst. Des. Implementation*, 2010, pp. 19–19.

[2] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. Mogul, "Spain: Cots data-center ethernet for multipathing over arbitrary topologies," in *Proc. 7th USENIX Conf. Netw. Syst. Des. Implementation*, 2010, pp. 265–280.

[3] S. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *Commun. Mag.*, pp. 136–141, 2013.

[4] A.-W. Tam, K. Xi, and H. Chao, "Use of devolved controllers in data center networks," in *Proc. IEEE Conf. Comput. Commun. Workshops*, 2011, pp. 596–601.

[5] A.-W. Tam, K. Xi, and H. Chao, "Scalability and resilience in data center networks: Dynamic flow reroute as an example," in *Proc. Glob. Telecommun. Conf. IEEE GLOBECOM*, 2011, pp. 1–6.

[6] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an elastic distributed SDN controller," in *Proc. 2nd ACM SIGCOMM Workshop Hot Top. Softw. Defined Netw. 2013*, pp. 7–12.

[7] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *USENIX INM/WREN (NSDI Workshop)*, 2010, pp. 1–6.

[8] C. Macapuna, C. Rothenberg, and M. Magalhaes, "In-packet bloom filter based data center networking with distributed openflow controllers," in *IEEE GLOBECOM Workshops*, 2010, pp. 584–588.

[9] J. Lavaei and A. Aghdam, "Decentralized control design for interconnected systems based on a centralized reference controller," in *Proc. 45th IEEE Conf. Decis. Control*, 2006, pp. 1189–1195.

[10] N. McKeown, et al., "Openflow: enabling innovation in campus networks," in *ACM SIGCOMM Comput. Commun. Rev.*, 2008, pp. 69–74.

[11] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, "A survey on software-defined networking," *IEEE Commun. surv. Tutorials*, vol. 17, no. 1, pp. 27–51, Jul.–Sep. 2015.

[12] F. P. Tso, K. Oikonomou, E. Kavvadia, and D. P. Pezaros, "Scalable traffic-aware virtual machine management for cloud data centers," in *Proc. IEEE 34th Int. Conf. Distrib. Comput. Syst.*, 2014, pp. 238–247.

[13] J. Cao, et al., "Per-packet load-balanced, low-latency routing for clos-based data center networks," in *Proc. 9th ACM Conf. Emerging Netw. Exp. Technol.*, 2013, pp. 49–60.

[14] S. Sen, D. Shue, S. Ihm, and M. J. Freedman, "Scalable, optimal flow routing in datacenters via local link balancing," in *Proc. 9th ACM Conf. Emerging Netw. Exp. Technol.*, 2013, pp. 151–162.

[15] L. Wang, F. Zhang, K. Zheng, A. V. Vasilakos, S. Ren, and Z. Liu, "Energy-efficient flow scheduling and routing with hard deadlines in data center networks," in *Proc. IEEE 34th Int. Conf. Distrib. Comput. Syst.*, 2014, pp. 248–257.

[16] D. Wu, D. I. Arkhipov, E. Asmare, Z. Qin, and J. A. McCann, "Ubiflow: Mobility management in urban-scale software defined iot," in *Proc. IEEE Conf. Comput. Commun.*, 2015, pp. 208–216.

[17] W. Liang, X. Gao, F. Wu, G. Chen, and W. Wei, "Balancing traffic load for devolved controllers in data center networks," in *Proc. IEEE Glob. Commun. Conf.*, 2014, pp. 2258–2263.

[18] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: fine grained traffic engineering for data centers," in *Proc. 7th Conf. Emerging Netw. Exp. Technol.*, 2011, pp. 1–12.

[19] R. Karp, *Reducibility Among Combinatorial Problems*. New York, NY, USA: Springer, 1972.

[20] D. Williamson and D. Shmoys, *The Design of Approximation Algorithms*. Cambridge, UK: Cambridge Univ. Press, 2011.

[21] T. Benson, A. Akella, and D. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, 2010, pp. 267–280.

[22] D. Corner, *Internetworking with TCP/IP (Vol.1 Principles, Protocols, and Architecture*), 4th ed., Englewood Cliffs, NJ, USA: Prentice Hall, 2000.

[23] N. Lynch, *Distributed algorithms*. San Mateo, CA, USA: Morgan Kaufmann, 1996.

[24] S. Brandt, K. Foerster, and R. Wattenhofer, "On consistent migration of flows in SDNs," in *Proc. IEEE Conf. Comput. Commun.*, 2016, pp. 1–9.

[25] J. Guo, F. Liu, X. Huang, J. Lui, M. Hu, Q. Gao, and H. Jin, "On efficient bandwidth allocation for traffic variability in datacenters," in *Proc. IEEE Conf. Comput. Commun.*, 2014, pp. 1572–1580.

[26] J. Guo, F. Liu, Z. D, J. Lui, and H. Jin, "A cooperative game based allocation for sharing data center networks," in *Proc. IEEE Conf. Comput. Commun.*, 2013, pp. 2139–2147.

[27] J. Guo, F. Liu, H. Tang, Y. Lian, H. Jin, and J. C. Lui, "Falloc: Fair network bandwidth allocation in IaaS datacenters via a bargaining game approach," in *Proc. 21st IEEE Int. Conf. Netw. Protocols*, 2013.

[28] J. Guo, F. Liu, J. Lui, and H. Jin, "Fair network bandwidth allocation in IaaS datacenters via a cooperative game approach," in *IEEE/ACM Trans. Netw.*, 2015.

[29] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM 2008 Conf. Data Commun.*, 2008, pp. 63–74.

[30] A. Greenberg, J. Hamilton, and N. Jain, "VL2: a scalable and flexible data center network," in *Proc. ACM SIGCOMM 2009 Conf. Data Commun.*, 2009, pp. 51–62.

[31] B. Heller, et al., "Elastictree: Saving energy in data center networks," in *Proc. 7th USENIX Conf. Netw. Syst. Des. Implementation*, 2010, pp. 249–264.

[32] S. Kandula, J. Padhye, and P. Bahl, "Flyways to de-congest data center networks," in *8th ACM Workshop Hot Top. Netw.*, 2009, pp. 1–6.

[33] Y. Li, L. Dong, J. Qu, and H. Zhang, "Multiple controller management in software defined networking," in *Proc. IEEE Symp. Comput. Appl. Commun.*, 2014, pp. 70–75.

[34] T. Koponen, M. Casado, N. Gude, J. Stribling, and L. Poutievski, "Onix: A distributed control platform for large-scale production networks," in *Proc. 9th USENIX Conf. Oper. Syst. Des. Implementation*, 2010, pp. 1–6.

[35] Open networking foundation (ONF). [Online]. Available: http://www.opennetworking.org/

[36] S. Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proc. 1st Workshop Hot Top. Softw. Defined Netw.*, 2012, pp. 19–24.

[37] A. R. Curtis, J. C. Mogul, J. Tourrilhes, and P. Yalagandula, "Devoflow: Scaling flow management for high-performance networks," in *Proc. ACM SIGCOMM Conf.*, 2011, pp. 254–265.

[38] A. Banjar, P. Pakawat, and B. Robin, "Daim: A mechanism to distribute control functions within openflow switches," *J. Netw.*, pp. 1–9, 2014.

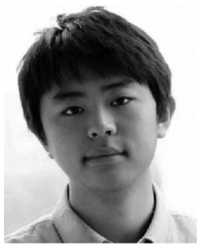
[39] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically centralized?: State distribution trade-offs in software defined networks," in *Proc. 1st Workshop Hot Top. Softw. Defined Netw.*, 2012, pp. 1–6.

[40] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *Proc. 1st Workshop Hot Top. Softw. Defined Netw.*, 2012, pp. 7–12.
[41] A. Roy, M. Bari, M. Zhani, R. Ahmed, and R. Boutaba, "Design and management of dot: A distributed openflow testbed," in *Proc. ACM SIGCOMM*, 2014, pp. 1–2.
[42] S. Jain, et al., "B4: Experience with a globally-deployed software defined wan," in *Proc. ACM SIGCOMM Conf.*, 2013.

Xiaofeng Gao received the BS degree in information and computational science from the Nankai University, China, in 2004; the MS degree in operations research and control theory from the Tsinghua University, China, in 2006; and the PhD degree in computer science from the University of Texas at Dallas, USA, in 2010. She is currently an associate professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. Her research interests include wireless communications, data engineering, and combinatorial optimizations. She has published more than 90 peer-reviewed papers and 7 book chapters in the related area, including well-archived international journals such as the *IEEE Transactions on Computers*, the *IEEE Transactions on Knowledge and Data Engineering*, the *IEEE Transactions on Parallel and Distributed Systems*, *Transactions on Circuits and Systems*, and also in well-known conference proceedings such as INFOCOM, SIGKDD, ICDCS. She has served on the editorial board of discrete mathematics, algorithms and applications, and as the PCs and peer reviewers for a number of international conferences and journals.

Linghe Kong received the BE degree in automation from Xidian University, China, in 2005, the Dipl Ing degree in telecommunication from TELECOM SudParis (ex. INT), France, 2007, and the PhD degree in computer science from Shanghai Jiao Tong University, China, 2012. He was also a joint PhD student at University of California, San Diego, 2011, and a visiting researcher in Microsoft Research Asia, 2010. He is an associate professor in the Department of Computer Science and Engineering, Shanghai Jiao Tong University. Before that, he was a postdoctoral researcher at McGill University from 2014 to 2015 and a postdoctoral researcher at Singapore University of Technology and Design in 2013. His research interests include wireless communication, sensor networks, mobile computing, Internet of Things, and smart energy systems.

Weichen Li is working toward the graduate degree in the Department of Computer Science and Engineering, Shanghai Jiao Tong University. He has been admitted by School of Computer Science, Carnegie Mellon University, USA. His research interests include data center engineering, distributed systems, data broadcasting, and distributed computing.



Wanchao Liang is currently working toward the master's degree at Carnegie Mellon University - Pittsburgh campus. He is a member of SAILING Lab at CMU, and his current research areas are distributed systems, large-scale machine learning, and cloud computing. He completed this work when he was an undergraduate student of Shanghai Jiao Tong University, China.

Yuxiang Chen is working toward the graduate degree from the School of Computer Science, Carnegie Mellon University, USA. He completed this work when he was an undergraduate student of Shanghai Jiao Tong University, China. His research interests include data center engineering and distributed computing, especially in the area of MapReduce and OpenFlow.

Guihai Chen received the BS degree in computer software from Nanjing University in 1984, the ME degree in computer applications from Southeast University in 1987, and the PhD degree in computer science from the University of Hong Kong in 1997. He is a distinguished professor of Shanghai Jiao Tong University. He had been invited as a visiting professor by Kyushu Institute of Technology in Japan, University of Queensland in Australia, and Wayne State University in USA. He has a wide range of research interests with focus on parallel computing, wireless networks, data centers, peer-to-peer computing, high-performance computer architecture, and data engineering. He has published more than 350 peer-reviewed papers, and more than 200 of them are in well-archived international journals such as the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, the *IEEE Transactions on Knowledge and Data Engineering*, the *ACM/IEEE Transactions on Networking* and the *IEEE ACM Transactions on Sensor Networks*, and also in well-known conference proceedings such as HPCA, MOBIHOC, INFOCOM, ICNP, ICDCS, CoNext, and AAAI. He has won several best paper awards including ICNP 2015 best paper award. His papers have been cited for more than 10,000 times according to Google Scholar.