RESEARCH ARTICLE OPEN ACCESS

Monitoring Managing and Co-ordinating Mega Data Centers With Multiple Controller

S.Ravi Kumar [1], Yarraguntla Jayalakshmi [2]
Department of Computer Science and Engineering
Visakha Institute of Engineering and Technology
Andhra Pradesh, India

ABSTRACT

In most existing cloud services, a centralized controller is utilized for resource management and coordination. Nevertheless, such infrastructure is gradually not sufficient to match the rapid growth of mega data centers. In recent literature, a new approach named devolved controller was proposed for scalability concern. This approach separates the whole network into various regions, each with one controller to monitor and reroute a portion of the flows. This technique relieves the trouble of an overloaded single controller, but brings other problems such as unbalanced work load among controllers and reconfiguration complexities. In this report, we establish an exploration on the use of devolved controllers for mega data centers, and design some new strategies to overcome these defects and improve the functioning of the organization. We first formulate Load Balancing problem for Devolved Controllers (LBDC) in data centers, and establish that it is NP-complete. We then design an f -approximation for LBDC, where f is the largest act of potential controllers for a switch in the mesh. Furthermore, we propose both centralized and distributed greedy approaches to solve the LBDC problem effectively. The numerical results validate the efficiency of our strategies, which can become a solution to monitoring, managing, and coordinating mega data centers with multiple controllers working in concert.

Keywords:- LBDC

I. INTRODUCTION

In recent years, data center has emerged as a common base that supports thousands of servers and supports many cloud applications and helps such as scientific computing, group collaboration, computer memory, financial applications, etc. This fast proliferation of cloud computing has promoted a rapid development of mega data centers used for commercial uses. Fellowships such as Amazon, Cisco, Google, and Microsoft have made immense 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 combine the traffic statistics and reroute the flows for better load balancing.

Driven by the unprecedent objectives of improving the performance and scale of DCNs, researchers try to deploy multiple controllers in such networks [4] [8]. The concept of devolved controllers is thereby brought out for the inaugural 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 a single omniscient controller, as a similar scheme in [9].

However, none of the controllers have the complete info on the whole web.. Rather, every controller only maintains a lot of the pairwise multipath information beforehand, hence cutting the work load significantly.

Recently, software-defined networking (SDN) as proposed by OpenFlow [10] has been touted as one of the most promising solutions for the 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 characteristics, we can divide the DCN flow control systems 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 dispenses with the load imbalance problem in a hierarchical fashion.

For the lower layer control, there are adult and well-evolved methods to treat the current control and VM migration at present [12] –[15]. Spell for the upper layer control, managing the DCNs by devolved controllers gradually becomes a live issue in recent years due to the enlargement of the scale of DCNs.

Many relevant studies emphasis on the imbalanced load problem for devolved controllers [4], [11], [16], only none of them break a clear conceptualization of the controller imbalance problem and examine the execution of their resolutions. This contributes to our concern on the imbalanced load issue for devolved controllers to better control the traffic and bring off the network.

Prompted by these fears, in this report we propose a novel system to manage devolved controllers. In our system, each controller monitors the traffics of a constituent of the switches locally. When traffic load imbalance occurs, some of them will migrate a lot of their supervised work to other controllers so that the workload can be kept balanced dynamically. We limit this problem as a Load Balancing problem for Devolved Controllers (LBDC). Then we design mul- tiple solutions for LBDC, including a linear programming with rounding approximation, three centralized greedy algorithms, and one distributed greedy algo- rithm. Using these results, we can dynamically Bal- once the traffic load among controllers. Such methods can cut down the natural event of traffic hot spots significantly, which will degrade network performance. These strategies can also improve the availability and throughput of DCN, supporting horizontal scaling and enhancing responsiveness of clients' requests. In whole, the main contributions of this report are as follows:

1) We design and implement a traffic load balancing scheme using devolve controller, which Elim- inates the scalability problem and balances the traffic load among manifold controllers. All these controllers are configured based on their physical placements, which is more sensible and takes in the whole network more efficient and dependable.
2) We prove the NP-totality of LBDC, and design an f -approximation algorithm to obtain the solution. We also get up with both concentrated and distributed heuristics for workload migration between controllers in dynamic sites. 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 experts- ailments. Arithmetical results validate our design's competence. To the best of our knowledge, we are the first to discuss workload balancing problem among multi-controllers in DCNs, which has both hypothetical and nononsense consequence.

This theme is the expanded variant of our conference version [17]. Established along the short symposium version, we add a randiest rounding for the linear programming, as well as two novel centralize migration algorithms under limited circumstances. To boot, we generate a new appraisal fragment and get more honest and exact results by several numerical experiments.

The remainder of the paper is formed as follows. Part 2 shows the system architecture and problem statement; Section 3 and Section 4 give our solutions to LBDC. Section 5 presents our performance evalue- action and demonstrates the potency of our algorithms. Section 6 introduces the related works; Finally, Section 7 concludes the paper.

II. PROBLEM STATEMENT

Traffic in DCN can be considered as Virtual Machine (VM) announcement. VMs in different servers collab- orate with each other to complete specified tasks. In order to communicate between VMs, communication flow will go through several switches.

Founded along the concept of OpenFlow [10], there is a flow table in each toggle, store the flow entries to be used in routing. One duty of a accountant is to modify these flow tables when announcement takes place. Every controller has a corresponding routing component and it may be composed of several graded switches, including Top of Rack (TOR) Switches, Aggregation Switches, and Core Switches. These substitutions are utilized for communication within the data center. Furthermore, every rack has a server called elected server [18], which is responsible for aggregate and processing the network statistics for the rack. It is also in charge of sending the summarize traffic matrices to the network controller, using a mapping program which wins over the traffic of this rack (server-to-server data) into ToR-to-ToR messages. In one case a controller receives these data, it will distribute them to a routing element which works out the flow reroute and reply to the new flow messages sent to the accountant. Then the checker installs these route in turn to all associated switches by switch over their flow tables. Since this report is not concerned with routing, we overlook the details of table computing and flow rerouting.

Today we will fix our problem formally. In a typical DCN, denote so as the earth switch, with the corresponding traffic weight w (psi), which is determined precisely as the bit of outgoing streams. Note that this weight does not take account of the communication within the ToRs. Next, given and switches S = s1, seen with their weights w (psi) and m controllers C = c1, cm, we desire to create a weighted m-partition for switches such that each contrailer will monitor a subset of switches. The weight of a controller w (chi) is the weighted summation of its monitored switches. Due to physical precincts, 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 *key* has a potential switch set $PS\ (chi)$ and it can only control switches in $PS\ (chi)$.

Later on the division, the real controller of sin is denoted by RC (psi) and the real switch subset of chi is denoted by RS (chi). The symbols employed in this theme are listed in Table 1.

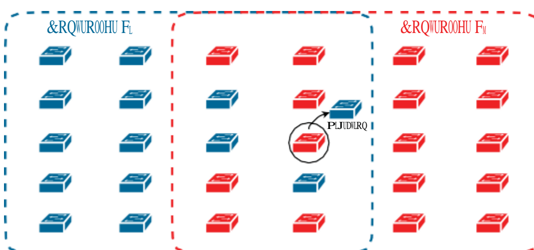
TABLE 1

Term	Definition
S, s_i	switch set with n switches: $S=\{s_1, \cdots, s_n\}$
$w(s_i)$	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: $C=\{c_1, \cdots, c_m\}$
$w(c_i)$	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.

The system running, the weightiness of the controller I may grow exclusively, making it unbalanced comparing with other controllers. Then in this condition, we must marginally migrate some switches in RS (chi) to other available controllers, in society to reduce its workload and preserve the whole network traffic balanced.

Then our problem becomes balancing the traffic load among my partition in real time upbringing, and migrate switches among controllers when the symmetry is gone. We limit this problem as a Load Balancing problem for Devolved Controllers (LBDC). In our system, each controller can lethargically migrate switch to or receive switches from understandably adjacent controllers to keep up the traffic load balanced.

Image 1 illustrates the migration pattern. Here Controller cj dominates 17 switches (as red shifts) and Controller ci dominates 13 switches (as blue switches). Since the traffic between chi and cj is unbalanced, cj is migrating one of its switches to watch.



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].

Fig. 1. An example of regional balancing migration.

We make an instance of LBDC. In this case there are two controller c1, c2 and A switches. Each switch *so* represents an element *a A*, with weight *w* (*SA*) = *size* (*a*). Both controllers can control every switch in the network ($PS(c_1) = PS(c_2) = SA$ *an A*).

Then, given a YES solution A' for PARTITION, we have a solution $RS(c_1) = s_a\{ a \mid \in A' \}$ $RS(c_2) = \{s_a \mid a \in A \setminus A'\}$ with $\sigma = 0$. The reverse part is trivial. The reductions can be done within polynomial time, which completes the proof. Next we demonstrate our results for the LBDC. We carry out the schemes within OpenFlow framework, which clears the system fairly easy to configure and enforce. It changes the devolved controllers from a statistical model into an implementable prototype. besides, our schemes are topology free, which is scalable for any DCN topology such as Fat-Tree, BCube, Portland, etc.

III. LINEAR PROGRAMMING AND ROUNDING

Given the traffic status of the a current DCN with devolved controllers, we can solve the LBDC prob- lem using the above programming. To simplify this programming, we will then transfer it into a similar integer programming

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

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 prop- erties of set cover: Given a universe U of n switch elements, S is a collection of subsets of U , and S= S_1, , S_n . And there is a cost assignment func- tion c : S 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) OPT_f O(log n) OPT , where OPT_f is the cost of an optimal solution to the LP-relaxation. $\leq$

Alg. 2 shows the formal description of LBDC-RR.

Algorithm 1: Randomized Rounding (LBDC-RR)

1 Let $\mathbf{x} = \mathbf{p}$ be an optimal solution to the LP;
2 **foreach** *set* $S_i \in$ **S do**
3 Pick S_i with probability x_{S_i}
4 **repeat**
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 U is covered by C. Suppose that a occurs in k sets of S. Let the probabilities associated with these sets be p , · · · , p . Since a is fractionally covered in the optimal solution, $p_1 + p_2 + + p_k \geq 1$. 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 inde- pendently pick c log n such subcollections. And then we compute their union, say C' , where c is a constant such that $\left(\frac{1}{e}\right)^{c \log n} \leq \frac{1}{4n}$.

Then we can obtain the following probability,

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

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

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

IV. ALGORITHM DESIGN

Using Linear programming and rounding, we can perfectly solve LBDC theoretically. However, it is usu- ally 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 Central- ized Migration, which includes Centralized Initialization and Centralized Regional Balanced Migration.

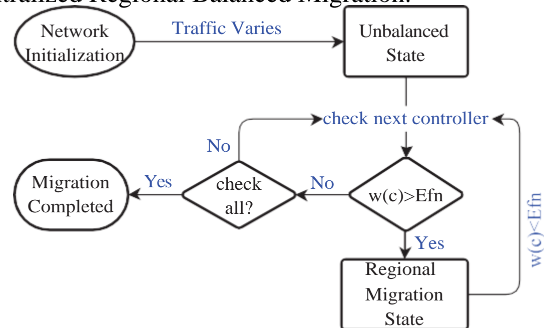


Fig. 2. Dynamic Load Balancing Workflow of LBDC

Centralized Initialization: First we need to initial- ize the current DCN, and assign switches to the con- trollers in its potential controller set. We design a centialization 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.

Break Tie Law: (1) When choosing s_i from S, we se- lect 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 candi-

dates, 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

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

Algorithm 2: 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={ $s_1, s_2, \cdots, s_n$ };
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 de- volved 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 con- troller 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 de- fine *Thd* as the threshold and *Efn* as the effluence. If

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

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

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

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

Algorithm 3: Centralized Migration (LBDC-CM)

Input: S with $w'(s_i)$; C with $w'(c_i)$; PendList $=$ OverList $= \{\emptyset\}$;

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

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 pa- rameters are used together to decide when to start

22 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 un- til the OverList is empty and makes the PendList become stable, which takes $O(m^2n)$. Step 4 and Step 5 balance the PendList locally as Step 2 and 3. In the worst case, the running time is $O(m^2n)$. 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 unlimitedprocessing 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% 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 param- eter values, and we use two sets to store them: $ThdList = \{Thd_1, \cdots, Thd_m\}$ and $EfnList = \{Efn_1,$

age 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 $\cdots, Efn_m\}$. 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

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 lim- ited LBDC-CM is very similar as the naive LBDC- CM in Alg. 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.

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 aver-

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 in- tense 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

to distribute s_i to c_2, if the current load of both con- trollers 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 V alue, and if c_j

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.

Algorithm 4: LBDC-CM with switch priority

1 **Step 2:** Find $c_m \in OverList$ with max $\frac{w'(c_m)}{w_m(c_m)}$;
2 **if** $\exists c_n \in AN(c_m) : w'(c_n) < Thd_n$ **then**
3 **repeat**
4 **if** $\exists c_f \in AN(c_m)$: $w'(c_f) < Thd_f$ **then**
5 Sort $PS(c_m)$ by v_{if}: $s_i \in PS(c_m)$;
6 Pick s_k with max v_{kf} in c_m, and pick max s_k to break tie;
7 Send $s_k \rightarrow c_f$;
8 **until** $w'(c_m) \leq Thd_m$ *or all* $w'(c_f) \geq Thd_f$;
9 **if** $w'(c_m) > Efn_m$ **then** move c_m
10 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 environ- ment 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 ran- domly. By sending control messages to the controller's potential switch set, the controller can determine the correct assignment. Alg. 6 shows the distributed ini- tialization 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 migra- tion 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

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 trans- action to transfer heavy switches to nearby nodes. Alg. 7 shows the distributed migration procedure.

Algorithm 5: 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;

IdleMode: (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 battle between the federal and the strewn migration is that the earlier can produce in sequence in a global position and grow to better decision, but it will also perform more meting out times and will become a potential classified access of the system. On the dissimilar, for the private eye in the broadcast version, each governor will simply gather data from its environs and can only make proper migration within this region. Though the distributed adaptation cannot obtain a global most advantageous balancing grade, it is more practical to deploy in real organizations. In the intervening time, it can resourcefully avoid the dilemma in the centralized scheme that the crumple or mistake of the central processor will affect problem of the system.

Their quarrel is also indicate in the characterization of the brink (Thd). In the federal version, the porch is moved by the utilizing ratio of the whole system, which is the same for each public accountant in the centralized system. A enchantment in the distributed version, the threshold of each controller is premeditated by its local information instead of the global in sequence, and the deserved utilizing ratio of each controller is in actuality different from each other.

By using our scattered scheme, for the state of affairs depicted in Fig. 1, regulator c_i and controller c_j will get the information about each other, estimate its Thd and Efn value, and make your mind up its status. If controller c_i is in the conveyance mode and controller c_j is in the being paid mode, then c_i will move around some of its dominate switches to c_j vouchsafe to Alg. 7.

4.3 OpenFlow based Migration Protocol

To keep up a well balanced in commission mode when a peak flow appear, switches should change the roles of their modern controllers while controllers should change their typescript by sending Role demand messages to the replacement. These operations stipulate the system to achieve a switch migration operation. Nevertheless, there is no such instrument provided in the OpenFlow standard. OpenFlow 1.3 defines three equipped modes for a controller master, work fingers to bone, and equal. Both superior and equal controllers can transform switched state and receive nonparallel messages from the substitution. Side by side, we design a unambiguous code of behavior to transfer a switch from its initial controller to a novel controller.

It is taken for granted that we are not able to stage-manage the switch in our resettlement protocol design, while it is theoretically reasonable to update the OpenFlow standard to put through our system. Nevertheless, there are two extra events. Foremost, the OpenFlow standard without a doubt says that a toggle may process messages not automatically in the same guild as they are picked up, first and foremost to allow multi-threaded implementations. Second, the measure does not specify unambiguously whether the order of messages sent by the switch remains regular between two controllers that are in master or equal mode.

Our code of behavior is built on the vital idea that we need to first make a distinct trigger event to stop message dispensation in the first controller and start a same significance in the second one. We can exploit the fact that Flow-Removed messages are sent to all controllers in commission in the equatorial mode. We therefore simply insert a dummy flow into the switch from the first controller and then remove the flow, which will offer a single trigger event to both the controller in equatorial mode to signal handoff . Our anticipated migration protocol for migrating switch some of the initial organizer key to target controller cog works in four forms as indicate under.

Phase 1: Change the role of intention c_j to equal mode. Here, controller c_j is first transitional to the equal mode for switch s_m. Initially master c_i initiate 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 dressmaker's dummy flow. To find an exact split second for the voyage, cue sends a dummy Flow-Mod domination to seem to append a new flow table entry that does not check any incoming packets. We presume that all controller know this dummy flow entry a priori as part of the protocol. And so, it sends another Flow-Mod command to delete this entry. In reaction, the switch sends a Flow-Removed message to both controllers since cog is in the equatorial mode. This Flow-Removed event provide a time point to transfer the tenure of switch s_m from c_i to c_j, after which only c_j will process all messages transmit by s_m. An supplementary barrier message is taken after the insertion of the dummy flow and before the dressmaker's dummy flow is deleted to prevent any chance of processing the delete message before the interleave. Note that we do not assume that the *Flow-Removed* message is received by c_i and c_j concurrently, since we assume that the message order is unswerving between c_i and c_j after these controllers enter the equal mode, meaning that all messages before *Flow-Removed* will be process 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 tenure of s_m in the subsequent phase, the protocol is not complete unless c_i is connected from s_m. However, it cannot just be isolated 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.

Even so, in that respect is no unambiguous appreciation from the switch that these messages are obtainable. Thus, in order to certification all these messages are steadfast, c_i hand on a Barrier Request and waits for the Barrier Reply, only after which it signal end migration to the final master c_j.

Phase 4: Dispense 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 information store to argue this. The switch sets c_i to slave when it receive the Role-Request communication from c_j. Then c_j remains active and process all messages from s_m for this segment.

The above movement protocol requires 6 round-trip times to complete the migration. But notice that we need to trigger migration only once in a while when the load conditions change, as we discussed in the algorithm design subsections.

V. PERFORMANCE EVALUATION

In this segment, we assess the execution of our centralized and disseminated protocols. We look at the case where traffic stipulate changes and test whether the metric of evenhanded workload controllers is minimize. We likewise consider the number of migrated switches into thoughtfulness. Furthermore, we look into how different parameters will determine the outcomes.

5.1 Environment Setup

We construct simulations by Python 2.7 to assess the routine of our intentions. We place 10,000 switches and 100 controllers in a 100 100m^2 square. Switches are evenly isolated in this square, sound out, each switch is 1m away from any of its neighbors. The controllers are too similarly spread and each controller is 10m away from its fellow citizen. Each controller can check all the switch within 30m, and can exchange a few words with other controllers within the range of 40m. We take on the weight of each switch follows the Pareto distribution with its parameter $\alpha p = 3$. We make a small simulation to choose the most appropriate α, β and γ, so that the environment we make can be very near to the actual situation, in terms of the traffic condition, workload of controllers, and migration frequency, etc. [13] –[15], [24]. Thus we set $\alpha = 0.7$, $\beta = 1.5$, $\gamma = 1.3$ as default configuration.

5.2 System Performance Visualization Results

We employ the default configuration described above to examine the operation of the organization. We first apply initialization and change the traffic demands dynamically to emulate unpredictable user requests. And so we apply naive LBDC-CM and other variants to alleviate the spot congestion. We use relative weight deviation to evaluate the functioning of our algorithms.

We analyze the performance of our four algorithms.. Consider a DCN with 10 10 controllers locating as a straight array. At the start of a time slot, the weights of switches are updated and then we play the migration algorithms.The weight of switches follows Pareto distribution with $\alpha_p = 3$. Figure 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. Figure 4, Figure 5, Figure 6 and Figure 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 lim- ited. This phenomenon is attributed to the system setting that one controller can only cover switches within 30m. When the number of controllers is few, more switches should be controlled by one particular controller without many choices. As the 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 Sec. 5.4.

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 hetero- geneous 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. Firstly, 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 Fig. 8 and Fig. 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 system performance significantly com-

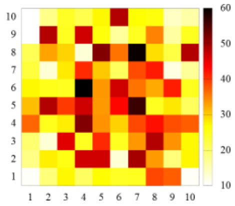


Fig. 3. Initial state

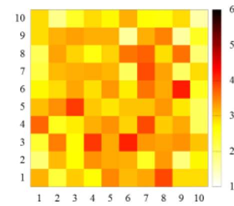


Fig. 4. Naive LBDC-CM migration

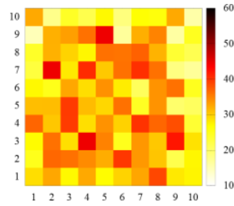


Fig. 5. Limited LBDC-CM migration

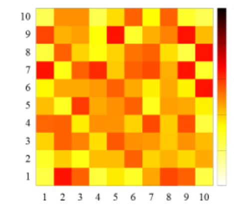


Fig. 6. Priori LBDC-CM migration

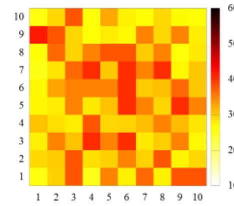


Fig. 7. LBDC-DM migration

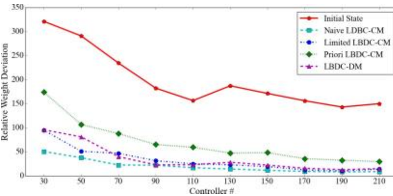


Fig. 8. Relative weight deviation protocol comparison

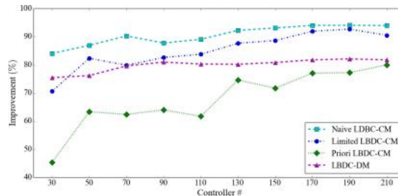


Fig. 9. Performance improvement for different protocols

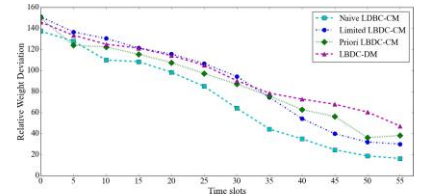


Fig. 10. Relative weight deviation without traffic changes

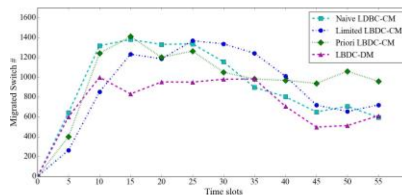


Fig. 11. Migrated switch without traffic changes

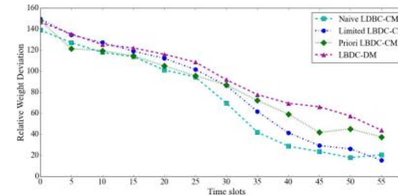


Fig. 12. Relative weight deviation as traffic changes

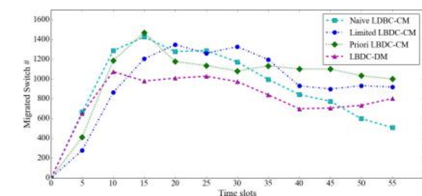


Fig. 13. Migrated switch as traffic changes

pared 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, lim- ited 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 dis- tance, 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.

Figure 10 and Figure 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. Figure 12 and Figure 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 bal- ancing among switches to better improve the sys- tem performance. Since we consider the balancing problem among controllers, which is like the "higher level" of balancing problem among switches, we can implement some load balancing strategies among switches [25]–[28] and combine the two-layers to- gether to achieve a better solution.

5.4 Parameter Specification

Next we explore the impact of the threshold parame- ters α, β, γ. Here α is a parameter to balance conser- vativeness 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 improve- ment rate and the number of migrated switches is mostly decreasing as β increases, which is actually correct according to the definition of the threshold.

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	6344
0.25	1.15	1.35	188.54	16.90	91.04	6236
0.25	1.35	1.15	193.81	11.83	93.90	6536
0.25	1.35	1.35	182.76	16.18	91.15	6224
0.75	1.15	1.15	196.73	12.01	93.90	6705
0.75	1.15	1.35	187.62	17.29	90.79	6244
0.75	1.35	1.15	178.77	15.46	91.35	6305
0.75	1.35	1.35	181.01	14.29	92.11	6231

VI. RELATED WORK

As data center becomes more important in industries, there have been tremendous interests in designing efficient DCNs [1], [2], [29]–[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, us- ing a centralized controller makes the design simpler and sufficient for a fairly large DCN.

However, using a single omniscient controller in- troduces scalability concerns when the scale of DCN grows dramatically. To address these issues, re- searchers installed multiple controllers across DCN by introducing devolved controllers [4]–[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 dis- tributed controllers [6]–[8] have been proposed for SDN [34] to address the issues of scalability and reliability, which a centralized controller suffers from. Software-Defined Networking (SDN) is a new net- work 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 net- work service providers and network manufacturers. The majority of current SDN architectures, OpenFlow- based or vendor-specific, relies on a single or mas- ter/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 or- ganization [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 knowl- edge, 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 Open- Flow 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 Con- trollers [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 em- ulate 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 Engi- neering (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.

VII. CONCLUSION

With the evolution of data center networks (DCNs), 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 (LBDC) 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.

REFERENCES

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *USENIX NSDI*, 2010, pp. 19–19.

[2] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. Mogul, "Spain: Cots data-center ethernet for multipathing over arbi- trary topologies," in *USENIX NSDI*, 2010, pp. 265–280.

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

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

[5] ——, "Scalability and resilience in data center networks: Dy- namic flow reroute as an example," in *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 *ACM SIG- COMM*, 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*, 2010, pp. 584– 588.

[9] J. Lavaei and A. Aghdam, "Decentralized control design for interconnected systems based on a centralized reference con- troller," in *IEEE CDC*, 2006, pp. 1189–1195.

[10] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Open- flow: enabling innovation in campus networks," in *ACM SIGCOMM*, 2008, pp. 69–74.

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

[12] F. P. Tso, K. Oikonomou, E. Kavvadia, and D. P. Pezaros, "Scalable traffic-aware virtual machine management for cloud data centers," in *ICDCS*, 2014, pp. 238–247.

[13] J. Cao, R. Xia, P. Yang, C. Guo, G. Li, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, and D. Maltz, "Per-packet load-balanced, low-latency routing for clos-based data center networks," in *CoNEXT'13*, 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 *CoNEXT'13*, 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 *ICDCS'14*, 2014.

[16] D. Wu, D. I. Arkhipov, E. Asmare, Z. Qin, and J. A. McCann, "Ubiflow: Mobility management in urban-scale software de- fined iot," in *INFOCOM*, 2015.

[17] W. Liang, X. Gao, F. Wu, G. Chen, and W. Wei, "Balancing traffic load for devolved controllers in data

center networks," in *GLOBECOM*, 2014, pp. 2258–2263.

[18] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: fine grained traffic engineering for data centers," in *ACM CoNEXT*, 2011, pp. 1–12.

[19] R. Karp, *Reducibility among Combinatorial Problems*. Springer US, 1972.

[20] D. Williamson and D. Shmoys, *The design of approximation algorithms*. Cambridge University Press, 2011.

[21] T. Benson, A. Akella, and D. Maltz, "Network traffic charac- teristics of data centers in the wild," in *ACM SIGCOMM*, 2010, pp. 267–280.

[22] D. Corner, *Internetworking with TCP/IP (Vol.1 Principles, Proto- cols, and Architecture)*. Prentice Hall (4th edition), 2000.

[23] N. Lynch, *Distributed algorithms*. Morgan Kaufmann, 1996.

[24] S. Brandt, K. Foerster, and R. Wattenhofer, "On consistent migration of flows in sdns," in *INFOCOM*, 2016.

[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 *IEEE INFOCOM*, 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 *IEEE INFORCOM*, 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 *ICNP*, 2013.

[28] J. Guo, F. Liu, J. Lui, and H. Jin, "Fair network bandwidth al- location in iaas datacenters via a cooperative game approach," in *IEEE/ACM Transactions on Networking*, 2015.

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

S.Ravi Kumar is presently pursuing M.Tech (CSE) Department of Computer Science Engineering from Visakha Institute of Engineering and Technology ,Visakhapatnam.

Yarraguntla Jayalakshmi , M.Tech ,(Phd) is working as an Assistant Professor in the Department of Computer Science and Engineering in Visakha Institute of Engineering and Technology ,Visakhapatnam.