

PLS: Proactive Load Shifting for Distributed SDN Controllers

Oluwatobi Akanbi, Amer Aljaedi and Xiaobo Zhou
Department of Computer Science, University of Colorado
Colorado Springs, CO 80918

Abstract—Balancing the workload among distributed SDN controllers plays a critical role for both the network performance and the control plane scalability. Therefore, various load balancing techniques were proposed for SDN to efficiently utilize the control plane’s resources. However, such techniques suffer increased latency and packet loss resulting from load migration and intensive communication among the SDN controllers. The existing solutions adopt load migration based on CPU utilization, which are susceptible to inconsistent load spikes. In this paper, we formally define the problem and present an alternate approach called PLS that constitutes the cornerstone for addressing this problem. We then show through experimental results that our approach provides accurate responses to load migration event triggers.

I. INTRODUCTION

In Software-Defined Networking (SDN), the abstraction of control plane from the data plane has enabled flexible programmability of network functionalities, which was infeasible in traditional networks. However, this abstraction comes at a cost of introducing bottleneck in the network due to the centralized network control. Despite the promising architecture of distributed systems for SDN controller design, it introduces in practice, inconsistent switch migration triggers due to high false-negatives since it is difficult to maintain real-time statistics in SDN environment.

Most research in this area focus on solving a single point of failure problem introduced by SDN controller architecture. While these solutions have been proven efficient in distributing the workload on multiple controllers. The issue of dynamically switching the workloads amongst controllers is still an ongoing area of research. Although some works have been targeted at this area, such as ONIX [1] and NVP [2] that focused on load balancing resources within a distributed controller environment, switches are statically assigned to the controllers making it difficult to react dynamically according to network events or conditions.

The remainder of this paper is organized as follows. Section II discusses some literature and how it relates to the problem. Section III discusses the motivation to our problem formulation. Section IV discusses the system overview. Section V describes the components of our proposed load shifting technique. Section VI details the implementation and current evaluation of the system. We conclude with section VII.

II. LITERATURE REVIEW

There are two important concerns of the current distributed SDN controller architecture as follows:

A. Flow Setup Latency

In statically assigned switches-controller architecture [3], [1], [4], controllers are prone to switch request overload in addition to the increased flow setup latency (i.e. because of their inability to adapt to traffic load shifts). Besides, distributed control plane architectures that rely on remote procedural calls can typically accrue twice the normal flow setup latency.

B. Switch Assignment

Assignment of switches to controller instance can be categorized into two classes based on existing research. The first class is the statically assigned switches [3], [1], [4], [5], [6] where switch-controller association is fixed. The second is dynamically assigned switches such as the work in [2], which enables a dynamic shift in switch-controller association when certain events are triggered i.e. CPU usage. Dixit *et. al.* [2] has shown that statically assigned switches often suffer from both overload and inefficient resource utilization during traffic load shift. Such static association can cripple the SDN controller to react to dynamic network conditions. This SDN controller design can lead to imbalanced load among the controller instances. Conversely, dynamic switch to controller assignment enables flexibility in response to the volume of switch events or current controller CPU utilization to dissociate or associate switch from or to a controller.

Table I
REFERENCE TABLE.

Variables	Description
C_n	SDN controller
S_m	OpenFlow switches
H_i	Host system
r_s	Switch-controller requests
th_r	Received request threshold
$X(t)$	Association matrix at time t
X	Set of possible association
U^c	Controller resource utilization t
$Q^s(r)_c$	Switch’s local queue backlog
C_n^{load}	Controller load
M^t	Message arrival rate
$A(r)_s$	Switch’s generated packet-in request after network discovery

III. MOTIVATION AND PROBLEM FORMULATION

In SDN architecture, several switches can be connected to form an island, where the controller instructs the switches

for routing/forwarding the network traffic. However, this can result in a bottleneck if one of the distributed SDN controller becomes overwhelmed. To address this issue, multi-controller SDN architecture has been designed to spread the workload efficiently across multiple controllers.

Table I shows the variables used in our problem formulation. We consider a simple Distributed Controller SDN (DC-SDN) with its control plane comprising a set C of physically distributed controllers, while its data plane consists of a set S of switches. Each switch $m \in S$ keeps a queue backlog $Q^s(r)_c$ of maximum size $\max Q^s(r)_c$ for tracking pending openflow (OF) request sent to the controller for processing. Each controller maintains a queue backlog $Q^c(r)_s$ that buffers requests from the data plane.

At the beginning of each network processing time t , each switch $m \in S$ generate some $r_s : 0 \leq A(r)_s \leq \max_a$ of requests. At this initial time t_i , no flow rules f_r have been installed in the switches yet; therefore, every flow is considered a new flow and as such should be processed at controller C which can then incur an increased RTT if controller utilization ($U^c \geq 50\%$) of available processing resources. New flow rules are then installed based on newer flow traffic inserted in the network. These flow rules are referenced as f_r^i . Next, the association of switches to controller can be modeled as follows:

First, we denote an associated matrix by $X(t)$ such that $X(t) \in \{0, 1\}^{|S| \times |C|}$. The following conditions apply:

- Cond.1 if $X(t)_{m,n} = 1$, then switch S_m is associated with controller C_n during the current processing time period,
- Cond.2 otherwise, $X(t)_{m,n} = 0$, implying no association between S_m and C_n .

Therefore, we denote the set of possible associations as $\bar{X}$.

$$\bar{X} = X \in 0, 1^{|S| \times |C|} \mid \sum_{n \in C} X_{m,n} \leq 1 \text{ for } m \in S \quad (1)$$

According to Cond.1, each switch m sends its request to associated controller n if $X_{m,n} = 1$; however, if $\sum_{n \in C} X_{m,n} = 0$ then the switch appends its request to a local queue backlog $Q^s(r)_c$.

Having introduced all the variable representation, we then dive into the main objectives and constraints of our research.

- 1) **Communication Delay:** We determine the communication delay between switch m and controller n as the estimate of its distance $d_{m,n}$. To compute the distance between m^{th} switch node to n^{th} controller $\iff \exists X \in \bar{X}$ such that the communication delay within one processing time period is

$$d_{m,n} = \sum_{j=1}^m X(t)_{m,n} \quad (2)$$

Where $X(t)_{m,n} = 1$ for some switch(s) $j \in m$ such that j is in the path from m to n .

- 2) **Computation Cost:** We determine the computational cost involved as ω_m for each request sent from switch m

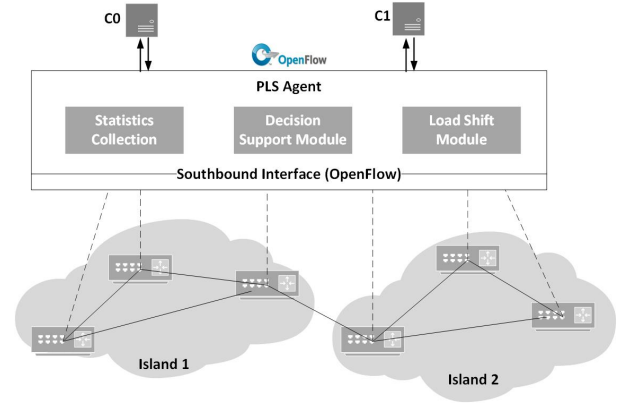


Figure 1. PLS Components.

but appended in the queue backlog for processing. For some association $X \in \bar{X}$, the computational cost within a processing time period is

$$\omega(t)_m = \hat{f}(X, \bar{X}(t)) \quad (3)$$

- 3) **Latency Cost:** The amount of latency incurred during a processing time period between switch m and controller n using the round trip time value. We first set a threshold th_r as a function of the average RTT between switch S_m and controller C_n . Then we take the current time value (c_t) recorded for every response to a request sent from the controller, and compare it with a pre-defined threshold. Latency is eminent once $c_t \geq th_r$. Therefore, the latency cost within a processing time period on the path between controller and switch denoted by $Lat(d)_{m,n}$ is

$$Lat(d)_{m,n} = \sum_{j=1}^m c_t \cdot d_{m,n} \quad (4)$$

Consequently, our problem formulation is given as follows

$$\begin{aligned} & \text{Minimize} \quad \lim_{t \rightarrow \infty} (Lat(d)_{m,n}^t + \omega(t)_m) \\ & \text{subject to} \quad (2)(3)(4) \end{aligned} \quad (5)$$

Where $Lat(d)_{m,n}^t$ is the path latency cost between some controller n and some switch m at time t , and $\omega(t)_m$ is the computational cost within a processing time t for each request sent from switch m .

IV. SYSTEM OVERVIEW

In contrast to previous works, our load shifting technique is able to efficiently balance workload on the controllers via a shim layer that serves as a middle layer between the control plane and the data plane.

The key components in our PLS design for DC-SDN controller management are shown in Fig. 1. It consists of the following three components:

- *Statistics collection* module is used to collect load information of controllers-switch association.

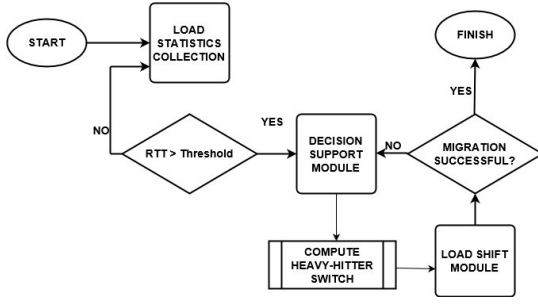


Figure 2. PLS Flowchart.

- *Decision support* module is used to compare all the distributed controllers' load statistics to estimate the least utilized controller within the DC-SDN.
- *Load shift* module is responsible for migrating heavy hitter switches' [7] to the least utilized controller, then adjust the new threshold.

PLS runs between the control and data planes which allows an unbiased view of both planes in real-time. Fig. 2 illustrates the flowchart of PLS and the representative module relationship.

V. PLS COMPONENTS

Here, we describe the components of PLS in detail with the reference algorithms to attest to the effectiveness of our proactive load shifting strategy for distributed SDN controllers.

A. Statistics Collection

Purposefully, our statistic collection module is set to collect three types of statistics: (1) forwarding rules sent by the controller for each switch as flow table entries (FTE), (2) Average control message arrival rate ($avg(M)^t$), and (3) round trip time (RTT) from controller n to switch m . Our shim layer uses these statistics to calculate the load C_n^{load} on each controller. Hence, the following equation:

$$C_n^{load} = w_1 * FTE + w_2 * RTT + w_3 * (1/n \sum_{i=1}^n M^t) \quad (6)$$

Where w_1, w_2, w_3 are weighted coefficients of the three factors respectively with a total sum of 1. This module forwards the collected statistics to the *decision support* module for further processing.

For reliable group communication between our shim layer and the controllers, we implemented JGroups toolkit [8] which also incorporates fail-over, high availability and performance measure. This toolkit also handles message triggers such as when a controller load is greater than the set threshold to initiate statistic collection control message on other controllers.

B. Decision Support Module

This module decides which load to migrate from one controller to another. In order to know which load to migrate from the overloaded controller and also ensuring that state

conditions are kept stable, we call our heavy hitter detection sub-routine within the decision support module.

In view of the entire system, we define a load balancing rate κ as a function of the average load on all the controllers with $U^c \leq 50\%$ to the total load incurred by the system (i.e. all the controllers active in the network). Therefore, we define κ as follows:

$$\kappa = \frac{\frac{1}{m} \sum_{i=1}^m \{C_i^{load}\}}{\text{maximum}\{C_i^{load}\}_{i=1 \text{ to } m}} \quad (7)$$

For some integer $i \leq m : i, m \in S$. We can deduce the value of κ from Eq. (7) such that the co-domain is of the range $[0,1]$. Hence, the closer κ is to 1 the more evenly distributed the load in our system. We set the default value of κ to 0.6; any value less than our default value for a particular controller will imply that the controller's load is non-uniform in view of the whole system. We restrict migration to occur only if the following conditions are satisfied:

- Cond.1 the controller's $\kappa \leq 0.6$, if multiple controllers have the same κ , a FIFO queue is initiated ensuring that the first controller that triggers a migration event is considered first,
- Cond.2 the controller has the largest load as compared to other distributed controllers.

Table II
CUSTOM TOPOLOGY.

Component	Island 1	Island 2	Island 3
Switches	32	64	48
Hosts per Switch	500	1000	750

C. Load Shift Module

This module is responsible for switch migration from one controller to another based on the output from the *decision support* module. As stated in Eq. (6), the weight assigned to these factors are statically assigned to normalize our data and further shows the trends in our experimental result. In a best-case scenario, the load reduction for an overloaded controller must be equal to that of the load migrated to other controllers.

Theorem 1: Resources fair share. In a matching resource fair sharing θ of a group of DC-SDN network G , the load C_n^{load} of the load shift receiver to the the previously overloaded controller must be such that the new load for the previously overloaded controller (NC_s^{load}) is equal to or slightly less than the new load for the receiving controller (NC_r^{load}) following Eq. 8, where integer $s, r \in \{C\}$.

$$NC_s^{load} \leq 0.1 \times NC_r^{load} \quad (8)$$

In addition, we state our *Resource Fair Share* in Theorem 1 for our load shift module to ensure that our system does not initiate a migration process that overloads another controller contrary to our hypothesis.

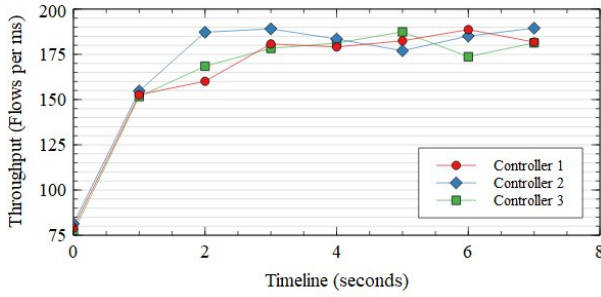


Figure 3. Controller benchmark using Cbench with 8 tests, each lasting for 10 seconds on 32 fake switches per controller with 1 warmup and 0 cooldown loop

VI. IMPLEMENTATION AND EVALUATION

To evaluate the load shifting efficiency of PLS, we implement DC-SDN based on Floodlight, which offers high modularity SDN control plane.

In the proposed system, our shim layer analyzes flow-requests every time t seconds to detect imbalances in the resource utilization of the distributed controllers.

We evaluate our work using a custom topology via mininet emulator [9] with three different islands as shown in Table II. In our simulation, we attached each island to our DC-SDN controller running on 64 bit ubuntu 16.04 operating system. Our environment is built on three servers, each has 4-cores Xeon E3-1200 processors connected via 1Gbit/s network pipeline.

A. Stress Testing

To obtain baseline parameters for our experiment, we stressed each of the distributed controllers to compute the threshold value th_r at 50% resource utilization U^c of each DC-SDN. Furthermore, to evaluate our prototype, we used a packet-in randomized generator function to push both uni-directional and bi-directional micro flows.

Purposefully, we leveraged the benchmark capability of Cbench tool [10] to benchmark our controllers using customized throughput test, and compare the results of our prototype to Cbench to show the effectiveness of our technique. Figure 3 shows the benchmark results at 10 seconds interval. Based on this result, we set our throughput threshold for flows per ms to 50% of 200 ($\approx value$). In addition, the result shows that each of the distributed controllers are closely matched in throughput.

B. PLS Evaluation

For our prototype evaluation, we used our initial setup to test the efficiency of our technique. Because of the complexity involved in modifying packet-in arrival rate, we used different switch and host sizes to emulate each controller load. We set κ value to 0.6 and calculate our load shifting success goal as shown in Eq. (8).

Success goal criteria is of two-folds. First, the load balancing rate κ must be at most 0.6 and the load on the stressed controller must be less than that of the receiving

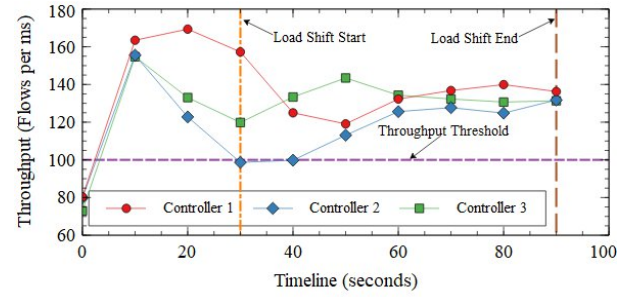


Figure 4. PLS Prototype Evaluation Outcomes. Load shifting started at 30 seconds timeline and ended at 90 seconds timeline

controller. Second, our resource fair share theorem requirement in Eq. (8) must be satisfied. PLS prototype results shown in Fig.4 demonstrate significant improvement. It can be observed that during the timeline between 0s and 29s, the load of individual DC-SDN controllers are above the configured throughput th_r , therefore, no need for load shifting. However, at timeline equals 30s, the load shifting module detects a stress condition on controller 2 causing our prototype to initialize load shifting. At which point, the module triggers a controller role change request on the heavy hitter switch which then moves to controller 1 as its new active controller and now controller 2 is set to standby role.

VII. CONCLUSION AND FUTURE WORK

In this paper, we put forward an efficient, dynamic load shift technique for DC-SDN based on the resource consumption metrics. Our solution extends the conventional SDN architecture without modifying the control plane or the data plane, as our load shift technique is embedded within a shim layer. For our future work, we plan to extend the current functionalities of our shim layer to include debugging features and more use cases.

REFERENCES

- [1] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, and T. Hama, "Onix: A distributed control platform for large-scale production networks," in *Proc. of USENIX OSDI*, 2010.
- [2] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an elastic distributed sdn controller," in *ACM SIGCOMM Computer Communication Review*, 2013.
- [3] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: a framework for efficient and scalable offloading of control applications," in *Proc. of ACM HotSDN*, 2012.
- [4] M. Monaco, O. Michel, and E. Keller, "Applying operating system principles to sdn controller design," in *Proceedings of ACM HotNets*, 2013.
- [5] S. Schmid and J. Suomela, "Exploiting locality in distributed sdn control," in *Proc. of ACM HotSDN*, 2013.
- [6] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *Proc. of ACM INM/WREN*, 2010.
- [7] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proc. of ACM SIGCOMM SOSR*, 2017.
- [8] B. Ban, "Reliable multicasting with the jgroups toolkit," *Red Hat Inc, Technical report*, 1998.
- [9] M. Team, "Mininet," 2014.
- [10] R. Sherwood and K. Yap, "Cbench controller benchmarker," <https://github.com/mininet/oflops>. Last accessed, Jan, 2018.