Received: 15 April 2018 | Revised: 28 August 2018 | Accepted: 5 September 2018

DOI: 10.4218/etrij.2018-0188

ORIGINAL ARTICLE

WILEY ETRI Journal

SD-WLB: An SDN-aided mechanism for web load balancing based on server statistics

Kiarash Soleimanzadeh[1] | Mahmood Ahmadi[1] | Mohammad Nassiri[2]

[1]Computer Engineering and Information Technology Department, Razi University, Kermanshah, Iran.

[2]Faculty of Engineering, Bu-Ali Sina University, Hamedan, Iran.

Correspondence
Mahmood Ahmadi, Computer Engineering and Information Technology Department, Razi University, Kermanshah, Iran.
Email: M.ahmadi@razi.ac.ir

Software-defined networking (SDN) is a modern approach for current computer and data networks. The increase in the number of business websites has resulted in an exponential growth in web traffic. To cope with the increased demands, multiple web servers with a front-end load balancer are widely used by organizations and businesses as a viable solution to improve the performance. In this paper, we propose a load-balancing mechanism for SDN. Our approach allocates web requests to each server according to its response time and the traffic volume of the corresponding switch port. The centralized SDN controller periodically collects this information to maintain an up-to-date view of the load distribution among the servers, and incoming user requests are redirected to the most appropriate server. The simulation results confirm the superiority of our approach compared to several other techniques. Compared to LBBSRT, round robin, and random selection methods, our mechanism improves the average response time by 19.58%, 33.94%, and 57.41%, respectively. Furthermore, the average improvement of throughput in comparison with these algorithms is 16.52%, 29.72%, and 58.27%, respectively.

KEYWORDS
Load balancing, OpenFlow, SDN, Server response time, Switch port traffic

1 | INTRODUCTION

Software-defined Networking (SDN) [1,2] is a new paradigm in networking, which provides manageable, scalable, and cost-effective network architecture. Software-Defined Networking decouples a network's control plane from the data plane; therefore, the data plane includes only forwarding devices and the control plane is implemented in a logically centralized entity called controller (or network operating system). Simple network (re)configuration, evolution, and policy enforcement are merits of such decoupling [3]. Controllers are categorized into two types: centralized and distributed. Initially, centralized controllers, as a single entity, were extended to control all networking devices. However, centralized controllers suffer from problems related to scalability, single point of failure, and resource limitation; therefore, distributed controllers, which control networks cooperatively, were proposed. On the other hand, SDN utilizes networking applications such as web load balancers and deep packet inspection. A web load balancer distributes the web request workload among several web servers. Traditional load balancers operate based on the web servers' load and cannot make decisions based on the response time of the servers and the current traffic of the switches. The current SDN load balancers do not take into account the response time of the servers and traffic of the switch ports in their decisions. However, the response time of each server and the current traffic of each switch port are two critical factors that need to be considered while deciding and selecting the most suitable server for processing the current request. Nowadays, web applications are executed on multiple servers in order to handle the increasing number of users' requests demanding web content. For this reason and due to the capacity limitation of servers, server-based clusters (also called server farms) are becoming more popular. Hence, load-balancing mechanisms are utilized to distribute incoming requests among the servers. In comparison to traditional networks, server load balancing in SDN can enhance the performance of the load-balancing server and can help overcome the problems related to its implementation. Load balancing provides many advantages such as scalability, availability, manageability, and security of websites. In particular, load balancing ameliorates the problems associated with the scalability of an application or server cluster by distributing the load across multiple servers. Load balancing can also redirect the traffic to alternate servers if a server or application fails. Load balancing allows network and server administrators to move an application from one server to another easily; hence, manageability is improved. In addition, load-balancing solutions protect server clusters from multiple forms of Denial-of-Service (DoS) attacks; thus, security is enhanced, and the incoming request to a Web system can be analyzed before sending it to the corresponding server [4]. It should be noted that the functions of load balancers in datacenters or IP networks are different from those in SDN networks. In datacenters or IP networks, the function of load balancing is controlled by a load-balancing device. When a new flow enters the network, it first goes through the load-balancing device, and the load-balancing device selects the most appropriate server as the destination based on the current network status. Then, the IP address of the selected destination server is written in the header of the new flow. Clearly, the routing decision is made only at the time of selection of the destination server, while the network status is changed over time. In such conditions, there may be other servers in the network that are more suitable compared to the destination server selected by the load-balancing device. Therefore, the load-balancing decision may not result in an optimal selection. In an SDN load balancer, when a new flow enters the switch, it will be forwarded to the controller. The controller selects the destination server (similar to that in the case of IP networks) only for the first time, and is enabled to act in real-time based on the network status the next time. Therefore, the SDN load balancer can select the destination server over time in an optimal way [5].

This paper focuses on the advantages of using load-balancing solutions to increase the performance of the Web system in SDN-based networks. Some SDN controllers have a basic load balancer, and do not assign requests dynamically to servers based on the network status. In this paper, we propose a new load-balancing algorithm, which periodically collects information on the traffic through a switch's ports and the server response time. Then, the load balancer adapts its strategies for allocating web requests based on the collected information. In addition, we compare the proposed approach with round robin, random selection, and Load Balancing by Server Response Time (LBBSRT) [6] approaches in an SDN environment. Our proposed mechanism achieves higher throughput and lower response time in comparison to the other load-balancing methods by considering the traffic through a switch's ports and the server response times simultaneously. The contributions of this paper are as follows:

- Proposal of an SDN-based mechanism for balancing requests load among web servers. The proposed SDN-based load balancer distributes the incoming web requests according to both the traffic volume and the response time of each web server.
- Through in-depth simulations, we evaluate the performance of our proposed approach and compare it with other well-known solutions.

The remainder of this paper is organized as follows: Section 2 briefly describes related works. Section 3 presents our contribution, which relies on a statistical SDN-based mechanism for balancing web requests among web servers. Section 4 reports the evaluation results obtained by our approach, round-robin, random selection, and LBBSRT in a software-defined networking scenario. Finally, we conclude the paper in Section 5.

2 | RELATED WORKS

In this section, we review other researches related to our work. Ananta [7] is a distributed layer-4 load balancer and NAT specifically designed to meet the scale, reliability, tenant isolation, and operational requirements of multitenant cloud environments. Its design was heavily influenced by the Windows Azure public cloud [8]. While Ananta is a good engineering effort, it does not follow open source standards and cannot be used easily by other people. DUET [9] is an integrated load-balancing approach that utilizes two methods of load balancing, namely load balancing based on switching and software load balancing. DUET provides high capacity, low latency, high availability, and high flexibility. This method utilizes the capabilities of conventional switches (such as traffic splitting and encapsulation of packets) to perform hardware load balancing, and in the case of breakdown of switches, a small software load balancing is implemented. Therefore, due to the ineffective use of software capabilities, it should be considered a hardware approach that has a low degree of flexibility. In [10], Wang et al. used a binary tree to represent the space of all possible IP addresses. The *i*'th level in the binary tree corresponds to the *i*'th most significant bits of the IP address. In a subtree, the nodes corresponding to a prefix is matched from the root of the path. If it is assumed that each IP address stores equal load, a tree data structure is useful since the load is distributed uniformly in the two subtrees. An algorithm is provided to minimize the expensive TCAM entries used to represent the tree.

In [10], AppSwitch system was proposed for load balancing in key-value storage systems. AppSwitch, unlike the existing key-value load balancers, requires a single message from the client to the server. In [11], a load balancer based on automata was presented. In this approach, the selection rate of the servers is adjusted according to the performance of information retrieval.

In [12], a wildcard rule method and an enhanced parameterized host partition method were proposed. It was shown that the devised approach is more efficient when a binary tree is used. These methods do not simultaneously consider the status of the servers and the traffic of the links, which are two important factors. In [13,14], Li et al. proposed a dynamic load-balancing algorithm to schedule flows efficiently for fat-tree networks, which provide multiple alternative paths among a single pair of hosts. To search a path in a recursive manner and to evaluate the achieved real-time traffic statistics from OpenFlow protocol, the hierarchical feature of fat-tree network is utilized. In [1], a load-balancing approach based on the server response time (as LBBSRT) was proposed. This approach utilizes the advantages (flexibility) of SDN by using the real-time response time of each server measured by the controller for load balancing. In [15], a Distribution-Aware Load Balancing (FDALB) approach was proposed. It decreases the flow completion time and enables high scalability. The FDALB splits the flow into two categories, namely short flows and long flows, depending on a threshold. Subsequently, distributed and centralized algorithms are used to balance the traffic of short and long flows, respectively. In [16], a load-balancing strategy based on fuzzy logic (LBSFL) was proposed. In this approach, the correlation among several parameters that influence load balancing is analyzed and the load of multiple virtual servers is obtained using a fuzzy logic algorithm. Then, the real-time load status of the virtual servers is examined, the lightest virtual server is selected to handle the request, and if necessary, the sleep/restart policy of the server is set. Finally, to verify the correctness and effectiveness of this load-balancing algorithm, an SDN simulation platform is constructed. In [5], a novel type of controller state synchronization approach and Load Variance-based Synchronization (LVS) are proposed for multi-controller multi-domain SDN networks. This improves the performance of the proposed controller. When the load of an individual server or a domain exceeds a specific threshold, the LVS-based approach utilizes efficient state synchronization. This LVS approach reduces the synchronization overhead of controllers. The authors claim that LVS-based approaches achieve loop-free forwarding and good load-balancing performance with a much fewer number of synchronizations. In [17], a load-balancing mechanism, which implements a hierarchical control plane with both a meta-control plane and a local control plane, is proposed for use in multiple-controller SDN networks. The meta-control plane analyzes the resources and utilization of the local control plane to optimize the processing performance. This approach helps the load balancing of the local control plane to improve the data plane performance and overcome the overheads of centralized control in a network. This work analyzes the proposed load-balancing approach in multiple-controller SDN networks. The achieved results show that the proposed meta-controller, which operates based on the manager approach, monitors and deals with the loading of the overloaded local controller. In [18], a datacenter network architecture based on SDN was proposed. It utilizes the topology-aware addresses in each switch to decrease the size of the forwarding table and human errors. In this paper, we propose a new load-balancing mechanism for web server farms, relying on a centralized controller in SDN, which periodically collects information on the traffic through a switch's ports and the server response time and then assigns the incoming flow to the proper web server.

3 | DESIGN AND IMPLEMENTATION OF THE PROPOSED SDN-BASED LOAD BALANCER

In this paper, we propose a load-balancing algorithm that works based on the traffic through the ports of a switch and the server response time. Figure 1 depicts a simplified view of the SDN architecture, whereas Figure 2 depicts our proposed load balancer system.

Our load balancer module is configured in the floodlight SDN controller, along with the other modules. Clients send their requests over the Internet to the datacenter. The gateway switch sends the first packet of each flow through an OpenFlow protocol to the controller in the form of a Packet_In message. Packet_In message is an OpenFlow message, which is issued from the switch to the controller. The controller places the Packet_In message on its modules for processing. Finally, after the load balancer selects the best server, a new rule for this flow is installed in the switch, and the flow is forwarded to the

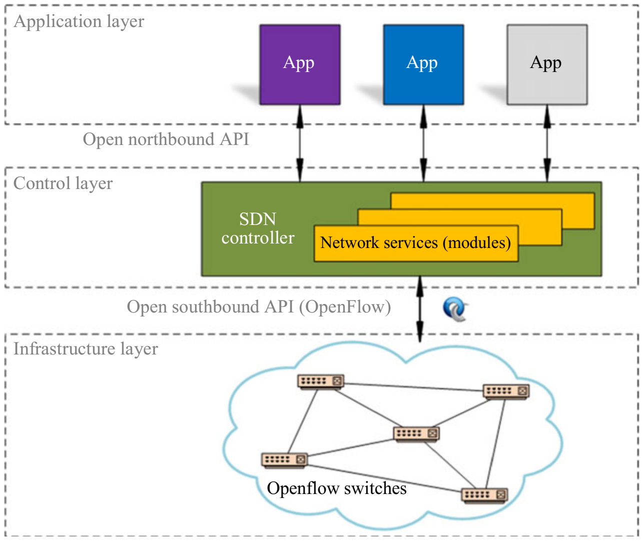

FIGURE 1 Software-defined networking architecture

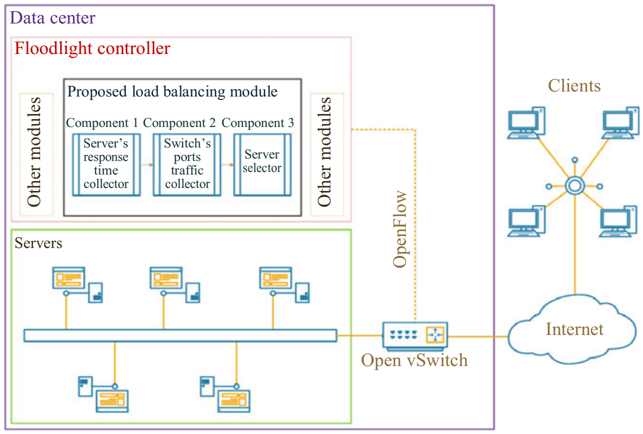

FIGURE 2 Proposed load balancer system

most efficient server by the switch. The proposed algorithm consists of three components: *Server response time collector component*, *switch's ports traffic collector component*, and *the best server selector component*. The *server response time collector* component and the *switch's ports traffic collector* component are responsible for collecting the server response time and the traffic through the switch's ports, respectively, and *the best server selector* component selects the most suitable server based on the obtained statistics.

3.1 | Server response time collector component

The operation of this component is depicted in Algorithm 1. After starting up the system (line 1), this component sends an http request to the server periodically in the uniform time slots (lines 2 and 3), and stores the sending time (line 4). Subsequently, the server response time is calculated by subtracting the sending time from the time of receiving the response (line 6), and the calculated response time is stored in the database (line 7). Moreover, to increase the performance and to decrease the response time, Head http messages (which only have a header and no body) are used. Owing to its performance, the SQlite database is used to store the responses. The server's response time collector agent in Figure 2 depicts this component.

3.2 | Switch's ports traffic collector component

This component receives the switch's port traffic periodically in uniform time slots, and stores it in a database. To extract the switch's port traffic, either the REST API feature of the floodlight controller or the OpenFlow port stat message can be used. In this paper, the REST API is used. The switch's port traffic collector component in Figure 2 depicts this component. The operation of this component is given in Algorithm 2. After starting up the system (line 1), the component sends a REST request to the controller (line 3) in the uniform time slot (line 2). Then, the response of REST is requested for each switch's port (line 4): To calculate the switch's port traffic in the next round of execution of Algorithm 2, two temporary variables, namely "previous total seconds" and "previous total bytes" are initialized (lines 5 and 6). These variables are used to store the values of time and traffic obtained by the REST request. In the next round, these variables are used to calculate the amount of time spent and the transferred traffic from the previous time slot until now (lines 8 and 9). Subsequently, the value of the traffic is calculated by dividing the volume of data by the time of transferring of that data (line 10). After updating the values of "previous total byte" and "previous total time" to the current values for the next round (lines 11 and 12), the port's traffic is stored in the database (line 13).

3.3 | The best server selector component

This component investigates each received *packet-in* message to check whether the received packet belongs to a new flow that needs to be processed by the servers (lines 2, and 3). Then, for each server (line 4), it reads *m* recent traffic samples and *m* recent server response time samples from the database (lines 5, and 6) for the port connected to the server. After that, for each server, the metric introduced in Eq. 1 is calculated (line 7). The server with the lowest value of this metric is selected for processing the current flow (line 8), and the packet that was received using the "packet-in" message by the controller is sent to the selected server using the "packet-out" message. Finally, the related rule is inserted into the flow table of the switch for the next incoming packets to overcome the forwarding of the new packet to the controller.

Algorithm 1 presents the *server response time collector* component, which performs the following tasks periodically for each server at system startup: An HTTP request is sent to the server, and $T_{\text{send},}$ which is the request sending time, is recorded (lines 4–5). Upon receiving the corresponding response, the time is stored in T_{arrive}. Then, the server response time, T_{response} is computed by subtracting T_{send} from T_{arrive} (line 6). Algorithm 2 presents the *switch's ports traffic collector* component, which performs the following tasks periodically: In line 3, the controller requests the switch's ports statistics through REST API. For each port, the value of *previous_total_bytes* and *previous_total_seconds* are initialized to 0. These variables store the values of *total bytes* and *total seconds* from the beginning until the previous execution of this subcomponent (lines 5–6). The response to the REST API request is parsed and *total received bytes* and *total seconds* are extracted for the current port (line 7). In lines 8–9, the amount of current received bytes for the current port is computed by subtracting *previous_total_bytes* from the current *total_bytes*. The same procedure is applied to compute the value of *current duration*. Finally, the traffic volume transferred over the current port is calculated as follows (line 10): *port_traffic* = (current_bytess × 8)/*current_seconds*. Here, *previous_total_bytes and previous_total_seconds* are also updated (lines 11–12). Algorithm 3 describes the operation of *the best server selector* component. The received Packet_In message is redirected to the other modules if this is not a user service request. Otherwise, *m* historical data for the switch's ports traffic and server response times are extracted (lines 5–6). These are collected by the *switch's ports traffic collector* and *server's response time collector* subcomponents. In line 7, the weighted average metric for each server is calculated using (1). Finally, the most appropriate server with the minimum value of weighted average in the server farm is selected, and the appropriate rules for this flow are injected to the switch (line 8).

The historical data include the recent server response time and switch's port traffic extracted using Algorithms 1 and 2 and stored in the database. To allocate a new flow, which enters a switch, to the best server, *m* recent response times for each server and *m* recent switch's port traffic are read from the database according to (1). The best server is selected to process the new flow. The best server is the server with the minimum value of the metric in (1). SQlite database, which is very fast and has high performance, is used in this study. Upon receiving a request from a web client, the switch performs a lookup in its table. If its features correspond to an existing entry, the switch forwards this incoming request to the server specified in that entry. In the case of a mismatch, the switch sends the first packet

ALGORITHM 1 Server response time collector

```
Output: The Server Response Time
1.  While system startup do
2.      If currenttime % $t == 0$ then
3.          For Each server do
4.              Send a HTTP request to server and record sending time, $T_{send}$;
5.              Record the time of receiving response, $T_{arrive}$;
6.              Calculate response time by the formula $T_{response} \leftarrow T_{arrive} - T_{send}$;
7.              Store server response time in SQLite database. ($T_{response}$);
8.          End For
9.      End if
10. End while
```

ALGORITHM 2 Switch's ports traffic collector

```
Output: The Switch's Port Traffic
1.  While system startup do
2.      If currenttime % t == 0 then
3.          Fetch switch's ports traffic by controller's REST API;
4.          For Each port of switch do
5.              previous_total_bytes ← 0;
6.              previous_total_seconds ← 0;
7.              Parse response of request and fetch total_bytes and total_seconds fields;
8.              current_bytes ← total_bytes - previous_total_bytes;
9.              current_second ← total_seconds - previous_total_seconds;
10.             port_traffic ← (current_bytess * 8) / current_seconds;
11.             previous_total_bytes ← total_bytes;
12.             previous_total_seconds ← total_seconds;
13.             Store switch's ports traffic in SQLite database, port_traffic;
14.         End For
15.     End if
16. End while
```

ALGORITHM 3 Server selector

```
Input: The Server's Response Times and Switch's Ports Traffic
Output: The most appropriate server for processing user request
1.  If controller receive a Packet_in message then
2.      Parse message;
3.      If package is for user service request then
4.          For Each server do
5.              Fetch m historical data for switch's ports traffic;
6.              Fetch m historical data for server's response time;
7.              Calculate weighted average by formula (1);
8.      Select server with minimum value of weighted average, and add appropriate rules for current flow and selected server in the switch.
9.          End For
10.     Else
11.         Send to other modules;
12.     End if
13. End if
```

of the new flow to the controller through a Packet_In message. Then, the controller dynamically selects the most appropriate server according to Algorithm 3.

$$M_i = \alpha \frac{\overline{\mathrm{SRT}_i} - \mathrm{Min}(\mathrm{SRT}_i)}{\mathrm{Max}(\mathrm{SRT}_i) - \mathrm{Min}(\mathrm{SRT}_i)} + \beta \frac{\overline{\mathrm{SPT}_i} - \mathrm{Min}(\mathrm{SPT}_i)}{\mathrm{Max}(\mathrm{SPT}_i) - \mathrm{Min}(\mathrm{SPT}_i)}, \quad (1)$$

where α and β are weights whose values are between 0 and 1 and are set by the network administrator. SRT_i denotes m recent response times for the i'th server, whereas $\overline{\mathrm{SRT}_i}$ is the corresponding average SRT_i for the i'th server. The first term of the formula is a normalized relative form of the average response time for the i'th server. SPT_i represents the m recent values of the traffic volume for the i'th port of the switch, and $\overline{\mathrm{SPT}_i}$ is its corresponding average. Again, we use a normalized relative form for the traffic volume on port i. M_i is our composite metric, which is a weighted average of both the normalized metric for the response time and the traffic volume. The alpha value in (1) depends on the network or datacenter infrastructure where the load balancer is located. In (1), the unit for the switch's port traffic is *bit per second*, and the unit for the server response time is *second*. The reason for normalizing the average traffic values of the ports of the switch and the server response time is to remove their units, so that we can take the sum of the two numbers without the units. The controller computes the value of (1) for each server upon the arrival of a new flow. It chooses the server with the minimum value of M_i, and then setups appropriate flow rules by OpenFlow FlowMod message for the selected server in the switch.

Figure 3 shows the messages that are exchanged among the client, Open vSwitch, load balancer, and server in the proposed method. First, the client sends a request to a Virtual IP address, due to the lack of a proper rule for flow in the switch. The switch sends the first packet of flow with OpenFlow `Packet_In` message to the controller. The controller executes the proposed load-balancing method, writes the appropriate rules for current flow in the switch, and sends the first packet of flow to the selected server by using the `Packet_Out` message. For subsequent requests belonging to the same flow, the same rule is applied. Then, the server processes the requests and returns the response to the Virtual IP address, and the switch redirects the response (with changes in the source address of the requests) to the client based on matched rule.

In the load balancer, we use Destination NAT (DNAT) to distribute the incoming request to the appropriate server according to the algorithm. In addition, we use Source NAT (SNAT) for the responses. Source NAT (SNAT) changes the source address in the IP header of a packet. It may also change the source port in the TCP/UDP headers according to the conditions. Typically, it is used to change a private address/port into a public address/port for the packets leaving the network. Destination NAT (DNAT) changes the destination address in the IP header of a packet. It may also change the destination port in the TCP/UDP headers according to the conditions. Typically, it is used to redirect the incoming packets with a destination public address/port to a private IP address/port inside the network.

4 | PERFORMANCE EVALUATION

To implement our method, we use Floodlight [13], which is an SDN controller offered by BigSwitch networks that works with the OpenFlow protocol to orchestrate traffic flows in an SDN environment.

We configured our topology with Mininet [19] simulator in a Python script. Mininet creates virtual networks using process-based virtualization and network namespaces. In our topology, as shown in Figure 2, we have multiple client nodes, all of which are connected to an OpenvSwitch [20]. We configured a custom HTTP server in our servers, which listens to port 80 and responds to the HTTP requests. A load balancer module exists in the controller, which is composed of three components: server response time collector component, switch's port traffic collector component, and best server selector. For generating HTTP requests, we have written a custom application in .NET Core [21]. .NET Core is a cross-platform, free and open-source program that manages the software framework introduced by Microsoft. In addition, we have written code for our servers in .NET Core using C#, which is supported by Microsoft for multiplatform applications. Furthermore, the server has been programmed and configured using the same platform. As existing traffic generating tools did not meet our conditions and often crash at the time of high-traffic creation, we decided to create a customized application for traffic generation. Each user requests 500 objects, having sizes between 100 KB and 100 MB. The aim of this experiment is the evaluation of a load balancer, which needs high throughput with the guarantee of the lowest possible download time. Table 1 depicts the evaluation parameters.

In Table 1, α and β are the weights of the weighted arithmetic mean used to select the best servers. t denotes the assumed time slot to extract the server's response time and the switch's port traffic in a periodic manner. m denotes the recent values of response time of the servers and port traffic of the switches that is used to select the best server when a new flow is entered. The m recent values are retrieved from a database. These values have been experimentally achieved and can be set by the administrator. In the meantime, in the evaluation of the proposed approach, four servers are used. The topology of the proposed load balancer system is constructed in a virtual machine (VM). Table 2 depicts the configuration of our proposed system topology. Our VM has a CPU with 8 cores and 10-GB RAM. The links between clients-switch have a 1000-Mbps bandwidth and switch-servers have a 100-Mbps bandwidth. The load balancer has a VIP address and all client's requests are sent to this virtual IP address. Therefore, the server's IP address is hidden from the clients. Indeed, the clients only know the VIP address. The virtual IP (VIP) is the load-balancing instance where the world points its browsers to get to a site. A VIP has an IP address, which must be publicly available to be useable. Usually, a TCP or UDP port number is associated with the VIP, such as TCP port 80 for web traffic. Usually, there are multiple real servers, and the VIP will spread traffic among them using an algorithm. Indeed, a VIP address maps one external IP address and one external port to multiple possible IP addresses and ports. It can also translate an external port to a different internal port. The VIP addresses map traffic received at one IP address to another address based on the destination port number in the TCP or UDP segment header. Therefore, clients do not need to know server IP addresses, and server-side changes are very convenient.

The system is designed to respond to client requests based on Zipf's law [22]. Thirty different clients send their requests, which is based on probabilities of Zipf's law, as shown in Table 3. Fifteen clients send their request continuously and other infrequently with a 2-s delay. Zipf's law proposes the distribution of requests in the real world. According to Zipf's law, a higher percentage of requests have lower-sized responses, and fewer requests have larger-sized responses. Table 3 shows the probability and size of different responses.

Clients request in parallel from a configured HTTP server in each of our servers. Figure 4 demonstrates the average response time (s), and Figure 5 depicts the average throughput (Mb/s) of algorithms. We have 30 clients, with an ability to send 500 requests.

Figure 4 depicts the average response time of the algorithms. Each algorithm is executed four times, and the results are reported with a confidence interval of 95%. As can be seen in the figure, the proposed algorithm has a response time lower than LBBSRT, round robin, and random algorithms. The average response time of the algorithms is 1.851 s, 2.225 s, 2.546 s, and 4.423 s, respectively. Equation (2) shows the confidence interval relation.

$$\bar{x} \pm 1.96\left(\frac{\sigma}{\sqrt{n}}\right), \tag{2}$$

where $\bar{x}$ is the average of results for each algorithm, σ is the standard deviation of results for each algorithm, and n is the number of times the desired algorithm is executed.

In addition, Figure 5 depicts the average throughput of the algorithms. Each algorithm is executed four times and a 95% confidence interval is calculated by (2). As can be seen in the figure, the proposed algorithm has a higher throughput than LBBSRT, round robin, and random algorithms. The average throughput of the algorithms is 148.19 Mb/s, 129.98 Mb/s, 117.01 Mb/s, and 89.26 Mb/s, respectively.

The average improvement of the response time in comparison with the three other algorithms—LBBSRT, round robin, and random selection methods—is 19.58%, 33.94%, and 57.41%, respectively. Moreover, the average improvement in throughput in comparison with the three other algorithms is 16.52%, 29.72%, and 58.27%, respectively. As can be seen, the response time of our proposed algorithm is lower than LBBSRT, round robin, and random

TABLE 1 Evaluation parameters

Parameter	Value
α	0.6
β	0.4
t	2 s
m	10

TABLE 2 Proposed system configuration

CPU	RAM	Clients-switch links	Servers-switch links
8 cores	10 GB	1000 Mbps	100 Mbps

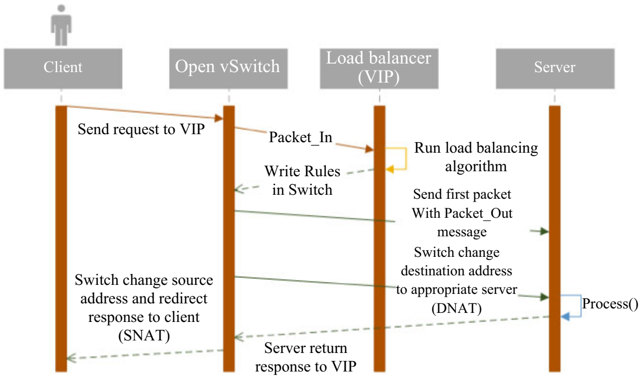
Client
Open vSwitch
Load balancer (VIP)
Server
Send request to VIP
Packet_In
Run load balancing algorithm
Write Rules in Switch
Send first packet With Packet_Out message
Switch change destination address to appropriate server (DNAT)
Process()
Switch change source address and redirect response to client (SNAT)
Server return response to VIP

FIGURE 3 Time diagram for the proposed system

TABLE 3 Size of returning objects from servers

Probability (%)	Response size
40	100k (obj1)
30	512 k (obj2)
15	2 M (obj3)
8	10 M (obj4)
4	30 M (obj5)
2	50 M (obj6)
1	100 M (obj7)

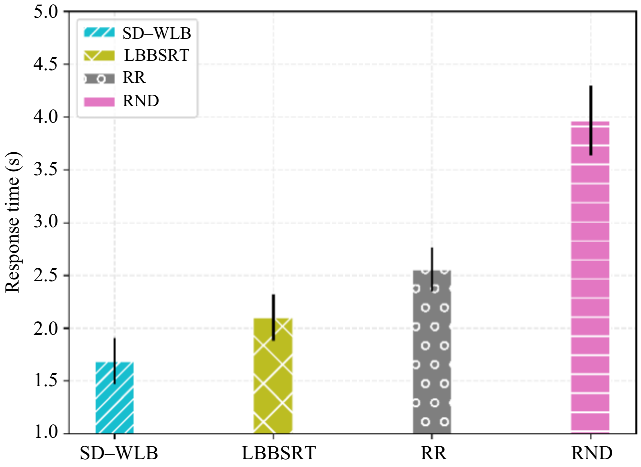

FIGURE 4 Average response time

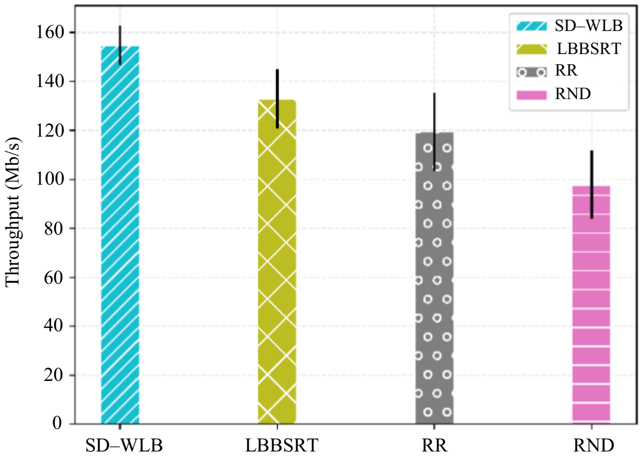

FIGURE 5 Average throughput

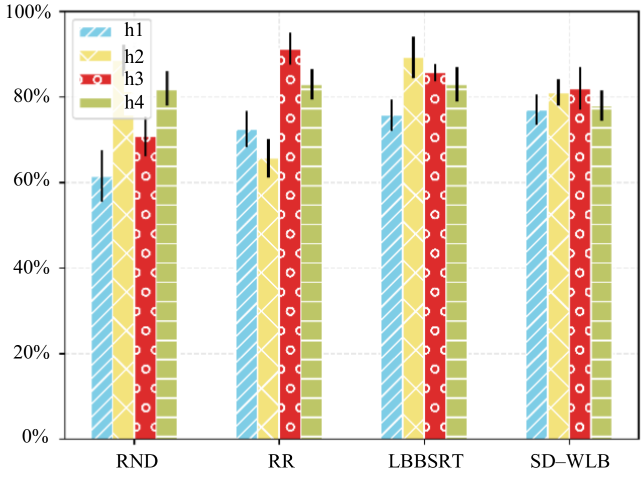

FIGURE 6 CPU utilization

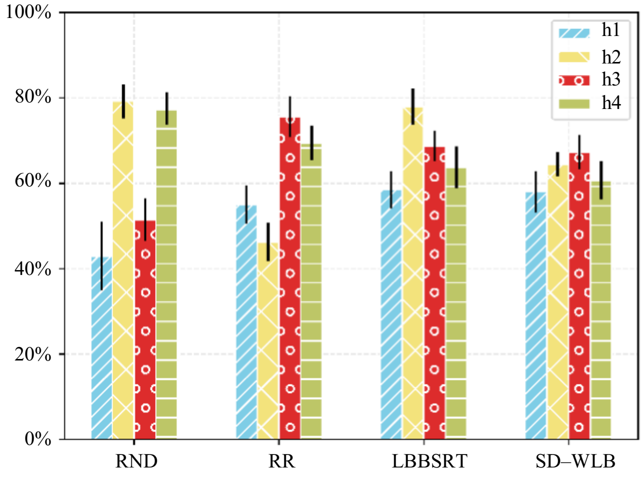

FIGURE 7 Memory utilization

selection algorithms. In addition, the throughput of our proposed algorithm is higher than LBBSRT, round robin, and random selection algorithms. To achieve a more prominent load-balancing effect in the proposed approach, we also extract the CPU and memory utilization rates for each server. Figures 6 and 7 present the CPU and memory usage graphs for the four servers—h1, h2, h3, and h4—under round robin, random, LBBSRT, and our proposed load-balancing approach. From Figures 6 and 7, we observe a slight difference in memory utilization and CPU utilization for the four servers in the proposed load-balancing approach. The reason is that in our approach, we receive the real-time traffic of switch ports and response time of each server. Thus, the resources of all servers are almost equally used and there is not a server that uses its resources much differently from the other servers. In comparison with the round robin, random, and LBBSRT approaches, our approach completely exploits the server resources and thus achieves a much better load-balancing effect. In other words, the proposed approach uses the available resources fairly.

5 | CONCLUSION

Load balancing improves the distribution of workloads across server farms in datacenters. As a result, end users will have a better experience by reducing response times and increasing throughput. In this paper, we proposed a new load-balancing algorithm in an SDN environment that periodically collects the switch's port traffic and server's response time and then chooses the most appropriate server based on these values. Based on evaluations, the proposed load-balancing approach has a higher throughput and lower response time in comparison to the LBBSRT, round robin, and random selection approaches. In future studies, we plan to consider the server's load and status and incorporate these statistics into our work for better decision making. In addition, we plan to increase the number of controllers and consider the time intervals for the collecting server's response time and switch's port traffic as well as the α and β weights in a dynamic and adaptive manner. Indeed, the stored values of server's response time, switch's port traffic, and other network statistics can be used as knowledge databases for machine learning methods.

ORCID

Mahmood Ahmadi https://orcid.org/0000-0003-4110-6824

Mohammad Nassiri https://orcid.org/0000-0002-0703-5949

REFERENCES

1. H. Zhong, Y. Fang, and J. Cui, *LBBSRT: An efficient SDN load balancing scheme based on server response time*, Future Generation Comput. Syst. **68** (2017), 183–190.
2. N. Mckeown, *How SDN will shape networking*, Oct. 2011, available at http://www.youtube.com/watch?v=c9-K5O_qYgA.
3. S. Schenker, *The future of networking, and the past of protocols*, Oct. 2011, available at http://www.youtube.com/watch?v=YHeyuD89n1Y.
4. H. Kim and N. Feamster, *Improving network management with software-defined networking*, IEEE Commun. Mag. **51** (2013), no. 2, 114–119.
5. K. Gilly, C. Juiz, and R. Puigjaner, *An up-to-date survey in web load balancing*, World Wide Web **14** (2011), no. 2, 105–131.
6. P. Patel et al., *Ananta: Cloud scale load balancing*, SIGCOMM Comput. Commun. Rev. **43** (2013), no. 4, 207–218.
7. Y. Li and D. Pan, OpenFlow based load balancing for Fat-Tree networks with multipath support, *Proc. IEEE Int. Conf. Commun. (ICC'13)*, Budapest, Hungary, June 9–13, 2013, pp. 1–5.
8. R. Wang, D. Butnariu, and J. Rexford, OpenFlow-based server load balancing gone wild, *Proc. USENIX Conf. Hot Topics Manag. Internet, cloud, enterprise netw. Services*, Boston, MA, USA, 2011, pp. 12–22.
9. R. Gandhi et al., Duet: Cloud scale load balancing with hardware and software, *Proc. ACM Conf. SIGCOMM (SIGCOMM '14)*, Chicago, IL, USA, Aug. 17–22, 2014, pp. 27–38.
10. Project Floodlingt, available at http://www.projectfloodlight.org.
11. Mininet, available at http://www.mininet.org.
12. Open vSwitch, available at http://www.openvswitch.org.
13. Microsoft, *Tutorial guide*, available at https://www.microsoft.com/net/core.
14. Wikipedia, *Microsift Azure*, available at https://en.wikipedia.org/wiki/Microsoft_Azure.
15. Wikipedia, *Zipf's law*, available at https://en.wikipedia.org/wiki/Zipf%27s_law.
16. G. Velusamy and R. Lent, Smart load-balancer for web applications, *Proc. Int. Conf. Smart Digital Environ. (ICSDE '17)*, Rabat Morocco, July 21–23, 2017, pp. 19–26.
17. T. L. Lin et al., A parameterized wildcard method based on SDN for server load balancing, *Int. Conf. Netw. Network Applicat. (NaNA)*, Hakodate, Japan, July 23–25, 2016, pp. 383–386.
18. R. H. Hwang and H. P. Tseng, Load balancing and routing mechanism based on software defined network in data centers, *Int. Comput. Symp. (ICS)*, Chiayi, Taiwan, Dec. 15–17, 2016, pp. 165–170.
19. Sh. Wang et al., *Flow distribution-aware load balancing for the datacenter*, Comput. Commun. **106** (2017), 136–146.
20. G. Li et al., *Fuzzy logic load-balancing strategy based on software-defined networking*, in Wireless Internet. WiCON 2018, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 230, Springer, Cham, Switzerland, 2018, pp. 471–482.
21. Z. Guo et al., *Improving the performance of load balancing in software-defined networks through load variance-based synchronization*, Comput. Netw. **68** (2014), 95–109.
22. Y. Ma et al., *Load-balancing multiple controllers mechanism for software-defined networking*, Wireless Personal Commun. **94** (2017), no. 4, 3549–3574.

AUTHOR BIOGRAPHIES

Kiarash Soleimanzadeh was born in Hamedan, Iran. He received his BS degree in software engineering from Malayer University, Malayer, Hamedan, Iran, in 2014, and his MS degree in Department of Computer and Information Technology Engineering from the Razi University, Kermanshah, Iran, in 2018. His main research interests include computer networking, software-defined networking, load balancing, computer graphics, databases, and big data.

Mahmood Ahmadi received his BS and MS degrees in computer engineering from Isfehan University, Isfahan, Iran, in 1995 and Amirkabir University of technology of Tehran, Iran, in 1998, respectively. He joined the Razi University of Kermanshah, Iran, as a lecturer in 1998. In October 2005, he joined the Department of Computer Engineering, Delft University of Technology, Delft, the Netherlands. Currently, he works as an assistant professor at the Department of Computer Engineering, Razi University. His research interests include network processing, approximate membership query data structures, software-defined networking, high-performance computing, performance modeling, and reconfigurable architectures.

Mohammad Nassiri is an assistant professor at the Computer Department of Bu-Ali Sina University. He received his BSs and MSs degrees from Iran University of Science and Technology (IUST) and Sharif University of Technology, respectively, in 2000 and 2002. He also received his PhD in Computer Engineering from Grenoble INP, France, in 2008. His current research interests are mainly experimental design for high-throughput WLANs, energy-aware protocol design for various wireless technologies, and traffic classification.