Load Balancing Using P4 in Software-Defined Networks

Chih-Heng Ke[1], Shih-Jung Hsu[2]

[1] Department of Computer Science and Information Engineering, National Quemoy University, Taiwan
[2] Master of Information Technology and Application, National Quemoy University, Taiwan
smallko@gmail.com, s8101127@gmail.com

Abstract

Conventional software-defined networks (SDNs) use a controller to write static rules into SDN switches through OpenFlow protocol. But legacy SDN switches cannot remember the data flow processing status. When the controller fails and cannot connect with the switch, the load balance function is affected. Conventional load balancer (LB), such as Linux Virtual Server and HAProxy, must perform layer-by-layer decapsulation, retrieve the information required to execute load balancing algorithms, and add the headers back before transmitting a packet. This process is time intensive. Therefore, we use P4 language to implement the LB, analyzes the packet headers, and uses stateful objects to record data flow information. The P4 LB can process packets according to predefined rules and operating status without operations such as encapsulation or decapsulation. Based on the aforementioned characteristics, we present four packet scheduling schemes, connection hash, random, round-robin, and weighted round-robin. Therefore, this P4 LB can independently function, without a controller. However, when a controller is available, the controller can be used to monitor the health of web servers. In this case, the controller can detect a server fault and inform the P4 LB to block the request to the malfunctioning server to decrease the dispatching failure rate.

Keywords: P4 switch, Software defined network, Load balancer

1 Introduction

In recent years, with the booming of the network and the popularization of smart phones, people can receive a lot of services through mobile phones, tablet PCs or computers, for example, online shopping, online study, Internet banking, online hospital registration and network car-hailing. Under such a development circumstance, the servers need to serve a bigger number of requests at one time, which increases the possibility of server overload and collapse and motivates the development of the load balancing technology. At present, the ways to reach load balance on the Internet include software ways like LVS [1], HAProxy [2], Nginx [3] and SDN [4-6] and hardware ways such as F5 and Array load balancers. These aforesaid technologies are mainly divided into the load balancing through Layer 4 [7] and the one through Layer 7 [8].

Generally, a common Layer 4 load balancer has a virtual IP address (VIP), and all the client requests are responded through this VIP. Every web server's IP (real IP) is stored and managed in the load balancer. Load Balancer, according to information of the client packet's source IP, destination IP, source port, destination port or transport layer protocol, can be allocated to different web servers.

A load balancer using Layer 7 can provide more functions. A load balancer determines whether a client request is for a dynamic web page (such as a web page containing database query results) or a static web page (such as an image file in .jpg or .gif format) according to the packet content of the client HTTP request and transmits the request to different back-end servers, thus reducing feedback time required to respond to a client request; a load balancer can also align with the cookie information of the client packet to ensure that information requested by the same client is served by the same web server.

Moreover, load balancers can implement two basic strategies: routing client requests to different servers, or routing packets via different paths to a single server [9-10]. We will focus only on the former. The open-source P4 language (short for "programing protocol-independent packet processor") represents an evolution of OpenFlow that achieves improved elegance and flexibility in configuring software switches. Previously reported work on P4 load balancers includes no function to check on the status of the backend servers [11-12]. In this case, the load balancer might direct a packet to a server that is down, resulting in error. For conventional SDNs, a controller must be used. Though packet-handling rules are written into SDN switches under the OpenFlow protocol, the switches cannot retain the processing status of the data flow. If the controller fails before a new request packet arrives at the switch, in this case, the switch will not know how to handle the request. Then the switch can only discard

*Corresponding Author: Chih-Heng Ke; E-mail: smallko@gmail.com
DOI: 10.3966/160792642020112106009

the new packet, which disrupts the load balance in the network and results in dispatching failure. To solve the problem, the P4 language addresses this and other limitations of the OpenFlow protocol by implementing stateful objects such as the register, counter, and meter. These stateful objects allow the software switch to be dynamically configured. In this paper, we present four packet scheduling schemes, connection hash, round-robin, weighted round-robin, and random. Therefore, our proposed P4 load balancer can independently function, without a controller. However, when a controller is available, the controller can be used to monitor the health of web servers. In this case, the controller can detect a server fault and inform the P4 load balancer to block the request to the malfunctioning server to decrease the dispatching failure rate.

Furthermore, we integrated Mininet [13], a P4 software switch (behavioral model version 2, bmv2) [14], and Docker host. Using the integrated environment [15], researchers can easily evaluate load balancing algorithms for web servers that have different central processing unit (CPU) performance.

This remainder of this paper is divided as follows. Section 2 presents background knowledge. Section 3 discusses the load balancing research method and design. Section 4 provides experimental results. Finally, Section 5 presents the conclusion and future prospects.

2 Background Knowledge

This section introduces two common software load balancers (LVS and HAProxy), SDNs, and the P4 software switch.

2.1 Linux Virtual Server (LVS)

LVS [1] is a software load balancer for Linux. The load balancing software, which supports Layer 4, is highly load-resistant and requires little internal storage and few CPU resources.

Figure 1 presents an example of an LVS-NAT mode. When a client sends a TCP SYN packet to initiate a connection with the HTTP server, the destination address is the virtual IP. The TCP SYN packet first reaches the load balancer, which selects a real server as per the dispatching algorithm. The connection information is then recorded in the load balancer's hash list to ensure that follow-up HTTP request packets are sent to the same real server. When the packet is dispatched to the backend server, the server writes its IP and port number to the packet. Before the packet is returned to the client, the LVS-NAT changes the packet's source IP and port number to the virtual IP and port number of the load balancer. LVS cannot verify the health of the back-end servers by itself; therefore, if a server is down, the LVS will still dispatch the request to it, resulting in an error.

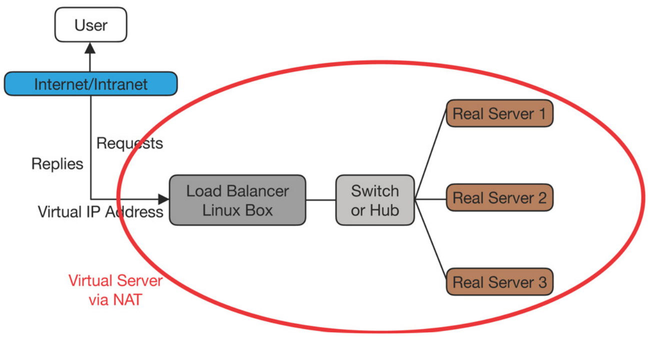


Figure 1. LVS-NAT mode

2.2 HAProxy

HAProxy [2] is load balancing software that can support virtual hosts and supports Layer 4 and Layer 7. HTTP is stateless, meaning that neither the server nor the client retains session information or connection status during multiple requests. Therefore, the server does not know the status of the client (i.e., whether the client is logged in) and has access to the user information in only the session record stored on the web server. Session mechanisms store required user information after the user completes identity authentication then produce a session ID and store it in a response packet before transmission to the user side. The next time the user side sends the request, the web server validates the request and identifies the session ID, thereby validating the user and confirming the connection status. Simultaneously, user data flow is guided to the same server.

HAProxy can verify the status of backend servers. For this purpose, HAProxy sends a TCP SYN packet to the backend server. If the back-end server returns TCP SYN+ACK packet, HAProxy replies with the TCP RST+ACK packet to terminate the connection, and logs the fact that the server is up. If no ACK packet is received, HAProxy knows that the server is down and will redirect traffic to other backend servers.

2.3 Software Defined Networks

Nick McKeown's research team introduced SDNs, a new network structure based on the characteristics of OpenFlow. SDNs separate the network control plane and data plane and realize network functions inside an SDN controller in a software approach intended to help network managers plan and manage networks. This structure has solved the problem of the network manager resetting every device when a network strategy changes. Moreover, if a traditional network structure requires a new network function (firewall or flow rate limitation), a new device must be purchased for onsite deployment and testing. However, in an SDN environment, users can create or purchase the required functional software module, send it to the appointed device through OpenFlow, and complete the setup of the new network functions. Certain SDN controllers such as the Open Network Operating

System (ONOS) are equipped with complete web management interfaces and can monitor transmissions on the network. These monitoring interfaces allow network managers to adjust routing strategies on the fly to reroute traffic around jammed channels. OpenFlow has become considerably more complicated as networking technology has advanced, going from implementing 12 matching fields in version 1.0 to 44 matching fields in version 1.5 [16]. OpenFlow no longer offers the flexibility and interoperability needed with state-of-the-art networks, and the high-level P4 language was introduced to develop network managers with even greater flexibility and hardware interoperability with SDNs.

2.4 P4 Switch

Bosshart et al. introduced the P4 language to address the OpenFlow's inability to analyze and process any field in the packet header. P4 switches can forward packets according to any protocol because they can be reconfigured flexibly by a parser. P4 was designed to offer three advantages over OpenFlow: (1) reconfigurability in the field, (2) protocol independence, and (3) target independence [17]. The compiler for a P4 switch automatically translates the P4 program into whatever machine code the target hardware requires. Figure 2 shows Bosshart et al.'s abstract model of packet forwarding. When a packet reaches the switch, the parser first extracts specific fields from the packet header. These extracted header fields are then operated upon by match+action steps in series or in parallel. In these steps, the packet may be forwarded, copied, or dropped in the ingress pipeline, and its header can be modified in the egress pipeline.

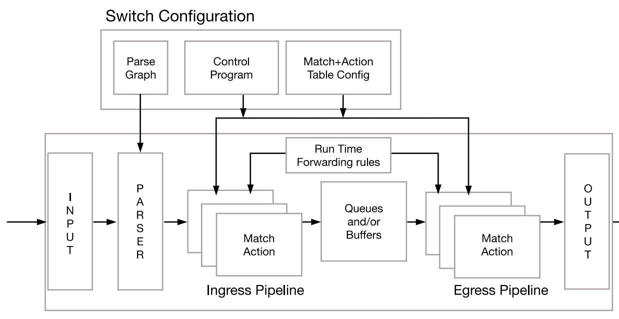


Figure 2. P4 switch abstract forwarding model [17]

The differences between OpenFlow and P4 are that OpenFlow is a protocol allowing us to add, modify, or delete forwarding entries for 44 matching fields in version 1.5 on the switches that have fixed functions. And P4 is a language that allows us to define the packet header, how to match the header, and what actions should be taken on each header. More information can refer to [18].

3 Research Methods and Design

We implemented four packet-scheduling algorithms in a P4 switch to test their performances: connection hash, round-robin, weighted round-robin, and random. We assume the network includes n backend servers, denoted by S_1, …, S_i, …, S_n, where $n \geq 2$. Moreover, we assume that the weights for S_1, …, S_i, … S_n are W_1, …, W_i, …, W_n, where W_i is an integer and $W_i \geq 1$ when using the weighted round-robin algorithm.

3.1 Connection Hash Load Balancer

We designed a P4 program for load balancing that uses the novel algorithm described in detail in this subsection. The connection hash load balancer algorithm uses five tuples in the IP and TCP header to produce hash values for the n servers in the backend stack. The tuples are the source IP, destination IP, source port, destination port, and the protocol value. When the client sends a TCP SYN packet to the P4 load balancer, the Parser gains the information of the Ethernet header, the IPv4 header, and the TCP header, and then runs the Verifychecksum function to evaluate whether the checksums in the IPv4 and TCP headers are correct. If so, the packet is forwarded to the ingress pipeline. Pseudocode for the Connection Hash Load Balancer is shown in Algorithm 1. Because this is a TCP SYN packet, a hash function is used to hash the five tuples to one of the n backend servers, i (line 9). Another hash function is used to hash five tuples as an index to access the register flow_select (line 10). The register flow_select stores the chosen server i in the corresponding index for the new connection (line 11). Next, health status of chosen server S_i should be checked (line 13-19). If server S_i is down, the algorithm needs to send the packet to the next functioning server S_j. Initially, all servers are assumed to be up. While the network is running, the P4 controller can dynamically change the statuses of each server. To route the TCP SYN packet, the load balancer needs to change the MAC and IP address to the selected server's MAC and IP address and send it to the corresponding port (line 20-24). The packet is not operated upon in the egress pipeline. When the server responds with a TCP SYN+ACK packet to the load balancer, the packet's MAC address is changed to the client's MAC address and sent to the corresponding port in the ingress pipeline (line 4-7). At the Egress Pipeline, the source IP address needs to be changed to the virtual IP (VIP) (line 25-27). Finally, the client will send out a TCP ACK packet to finish the three-way handshake. The five tuples of this TCP ACK packet are the same as those of the first TCP SYN packet. Therefore, the five tuples are used to simply identify the serving server in flowlet_select (line 1-3), and no further hash function is required to choose the serving server. Then, this TCP ACK packet is sent out to the

Algorithm 1. Connection Hash Load Balancer

```
Ingress Pipeline
1.  if a packet destined to virtual IP and not a TCP
    SYN packet
2.    using five tuples to get a hash value as the index
      in the flowlet_select to get serving server
3.  end
4.  if a packet destined to client
5.    change the destination MAC address to the
      client's MAC
6.    send this packet to the corresponding port
7.  end
8.  if this is a TCP SYN packet destined to virtual IP
9.      using five tuples to get a hash value i (i is 1~n)
10.   using five tuples to get a hash value as an index
      to flowlet_select
11.   store i in the flowlet_select with the corresponding
      index
12. end
13. if Si is down
14.    find the next functioning well server with index j
15.    store j in flowlet_select
16.    if all servers are down
17.       drop this packet
18.    end
19. end
20. if a packet destined to virtual IP
21.   change the destination MAC address to the
      chosen Server's MAC
22.   change the destination IP address to the chosen
      Server's IP
23.   send this packet to the corresponding output port
24. end
Egress Pipeline
25. if a packet destined to the client
26.   change the source IP address to VIP
27. end
```

selected server. The subsequent operations on the HTTP request packet sent from client are similar to those performed on the TCP ACK packet, and the operations on the HTTP response sent from the server are similar to those performed on the TCP SYN+ACK packet. We use a cURL script in the controller for health checks to avoid introducing an agent on the server side. If the cURL script fails to execute, the corresponding server is assumed to be down. The controller will then change the status of the corresponding server in the P4 code, and the P4 load balancer will not dispatch the request to the down server. If the execution of the cURL script is successful, the corresponding server is healthy and remains on the server list. To maintain the separation of the control plane from the data plane, we use a separate communication channel to send and receive health check packets between controller and servers. Moreover, the health check function in our controller can monitor a specific webpage to verify whether it contains a predefined keyword. If the fetched webpage does not include this keyword, this webpage may have been hacked. The server hosting this webpage should be shut down for a detailed checkup and removed from the load balancer's server table. For the complete code and a model of this load balancer, refer to [19].

3.2 Random Load Balancer

The random load balancer behaves similar to the connection hash load balancer, except that it uses a pseudorandom number generator to select server S_i. This difference is reflected in lines 8-12 of Algorithm 1. All other operations are the same. For the complete code and model, please refer to [20].

3.3 Round Robin (RR) Load Balancer

For the round robin (RR) load balancer, the P4 program requires another register, i.e., myselect in our implementation, to remember the last serving server index. When a TCP SYN packets arrives at the ingress pipeline, the last-used server index i is obtained from myselect. The load balancer then uses i and n to move through servers in round-robin fashion, storing the indices with the myselect and flowlet_select objects. The health check operation and packet routing to the client are identical to those in the connection hash load balancer. For the complete code and model, please refer to [21].

3.4 Weighted Round Robin (WRR) Load Balancer

The primary difference between the RR and WRR load balancers is that the last serving server index is not saved in the myselect register. Instead, a value between 0 and $\sum_{i=1}^{n} W_i$ is saved. If the value is less than or equal to W_1, the server 1 is selected. If the value is greater than $\sum_{i=1}^{p} W_i$ and less than or equal to $\sum_{i=1}^{q} W_i$, server q is selected. As each new connection (a TCP SYN packet) arrives at the Ingress Pipeline, the value k stored in myselect is incremented up by 1. The corresponding server is then extracted as per the above rules. When k reaches n, it is set back to 0. After selecting the serving server, the selected server and index k will be stored back into flowlet_select and myselect. The other operations are identical to those of the connection hash algorithm. For the complete code and model, please refer to [22].

4 Experiment

4.1 Settings

We designed three scenarios to compare the performance of the load balancing algorithms running on a P4 software switch. We adopted the Mininet emulator [13] and a P4 software switch (bmv2) [14] for our experimental environment. We use Docker containers to act as backend servers and Apache for web service. In Scenarios 1 and 2, the P4 switch is implemented with the four load balancing algorithms, namely, connection hash, random, round-robin, and weighted round-robin. ApacheBench [23] is used to simulate a scenario of 10 users sending 10,000 requests to fetch a 300-KB web page. Each experiment is repeated with 30 trials for each load balancing algorithm to obtain a confidence interval of 95%. Scenario 3 includes four back-end web servers, but server 4 is down. The weights for servers 1-4 are 1, 2, 3, and 4, respectively, for the weighted round robin algorithm. If a controller is available, it performs health checks at 1-s intervals. Moreover, we use a bash script that uses cURL to access the web pages 1000 times such that we can compare the load balancers' dispatching failure rates.

4.2 Scenarios

Scenario 1. Servers with identical CPU performance

Assume that the CPU performance of the Docker web server is 50,000 µs for the CPU period and 5000 µs for the CPU quota. The CPU quota specifies the time that Docker has access to CPU resources during the time specified by the CPU period [24]. With the settings we implemented, this quota limits each Docker web server to using 10% of the common CPU resources. The experiments with identical server CPU performance use two, three, and four servers, as illustrated in Figure 3.

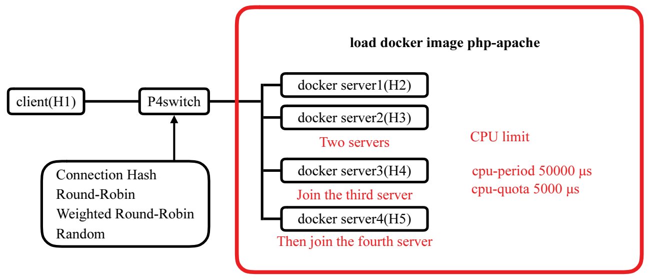


Figure 3. Scenario 1

Scenario 2. Servers with different CPU performanc

Assume that the CPU period of server 1 is 50,000 µs, and its CPU quota is 10,000 µs. The CPU periods of servers 2, 3, and 4 are 50,000 µs, and the corresponding CPU quota is 5000 µs. The Scenario 2 settings are presented in Figure 4.

Scenario 3. Dispatching failure rate

The failure rate is defined as the ratio of how many times the cURL program fails to fetch a web page to the total number of cURL requests sent. This metric lets us compare the dispatching failure rate of our proposed P4 load balancer and a conventional

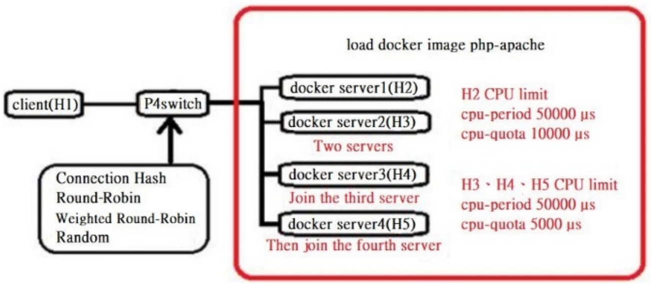


Figure 4. Scenario 2

OpenFlow-based switch with or without a controller. Moreover, the legacy LVS and HAProxy switches, which adopt the round-robin algorithm, are also compared with our P4 load balancer.

4.3 Results

As shown in Figure 5, in Scenario 1, each addition of a back-end server effectively reduces the response time. Among the four load balancing algorithms, the round-robin load balancer is most efficient when the CPU performance of the back-end servers is the same.

Figure 6 shows the distributions of packets among the two servers after collecting 240 packet requests in the P4 switch log. As expected, the round-robin load balancer distributes the packets evenly. The weighted round-robin load balancer distributes packets to the backend servers in a 2:1 ratio, overloading in Server 1 and increasing the response time. With the random load balancer, packets can be distributed to the same server continually. Therefore, every 80 packets will be distributed differently, which increases the response time. Finally, the connection hash load balancer tends to overload one or the other of the servers. Therefore, when the CPU performance of the servers is the same and equal requests are sent to the servers, the load balance is superior with the round-robin algorithm.

As shown in Figure 7, when the back-end servers have distinct CPU speeds (Scenario 2), the weighted round-robin load balancer is most efficient. For the random load balancer and connection hash load balancer, every time the number of servers increases by one set, the reduction in response time is considerably small, although the greater number of servers should reduce response time. These algorithms seem to send many requests to servers with slower CPU performance, explaining this effect.

Figure 8 shows the distribution of 240 successive packet requests between the two web servers in the P4 switch log. With the round-robin load balancer, the packets are equally distributed between the back-end servers; however, Server 1 has higher CPU performance and receives the same number of requests as Server 2. Thus, server 1 has idle resources. If certain requests were redistributed from Server 2 to 1, the response time would be shorter. With the weighted round-robin load balancer, the packets are distributed to the back-end servers in a 2: 1 ratio, according to the relative

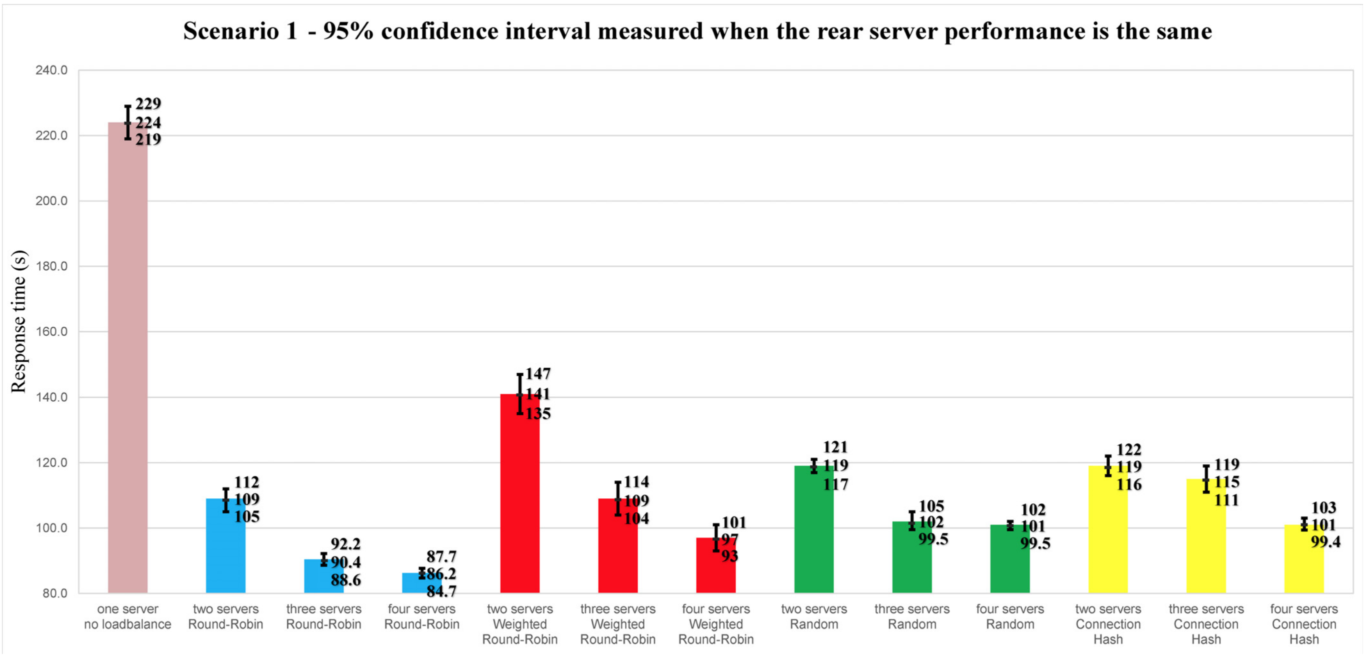


Figure 5. Results of Scenario 1

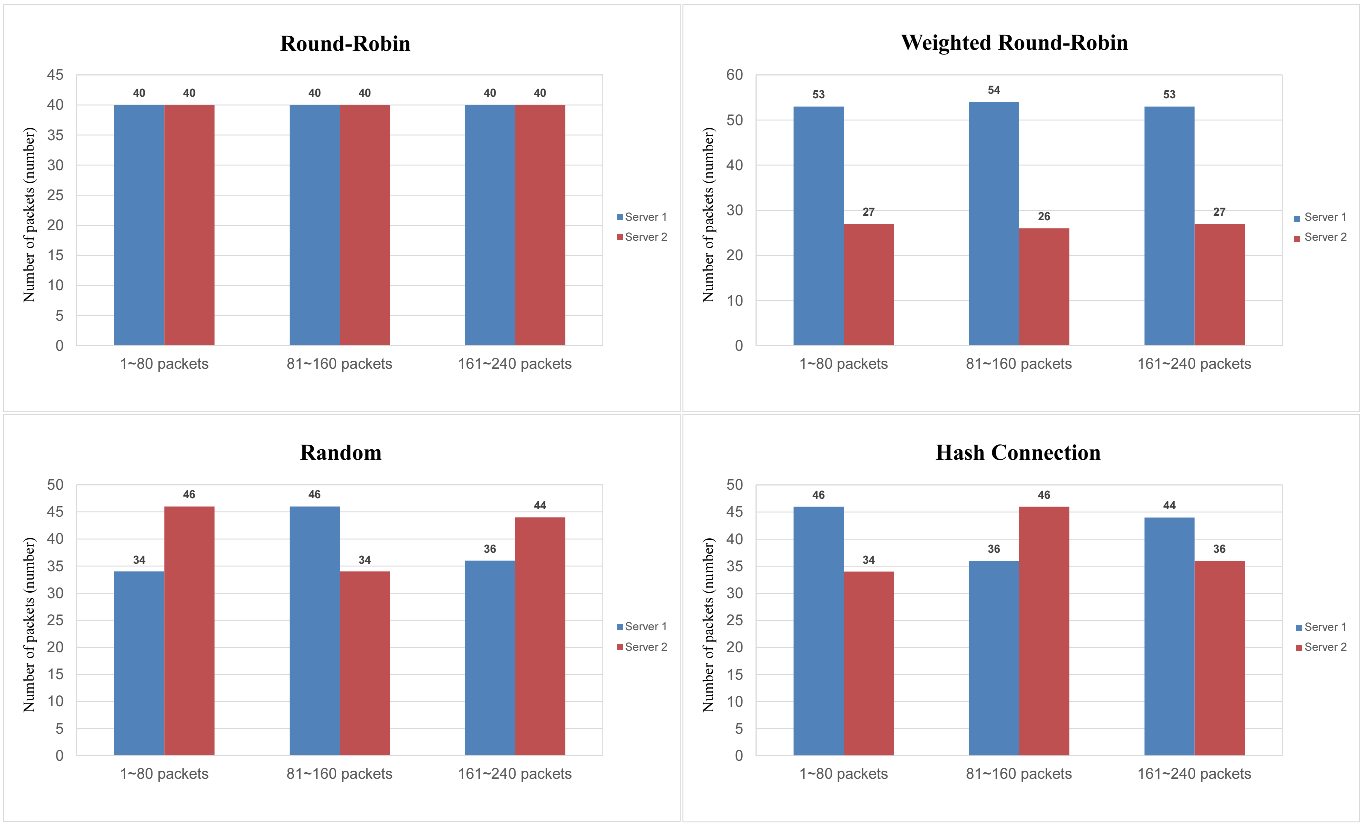


Figure 6. Distributions of packets to servers under Scenario 1

speeds of the CPUs. As expected, then, WRR is the most efficient load balancer in this scenario. With the random load balancer, more requests are distributed to Server 2, which shows lower CPU performance and increased the response time. Similarly, with the connection hash load balancer, the uneven request distribution increases the server response time.

Table 1 and Table 2 show the dispatching failure rate for SDN-based load balancer, LVS, and HAProxy. In Table 1, Legacy SDN method indicates that we use a controller that implements the round-robin algorithm to help an OpenFlow-based switch execute load balancing job. When a controller is available, our proposed P4 load balancer can achieve a failure rate of 0% because includes a controller. Since the legacy SDN has no health-check function, all requests that are directed to server 4 get no response; therefore, the failure rate is 25%. When no controller is available, the

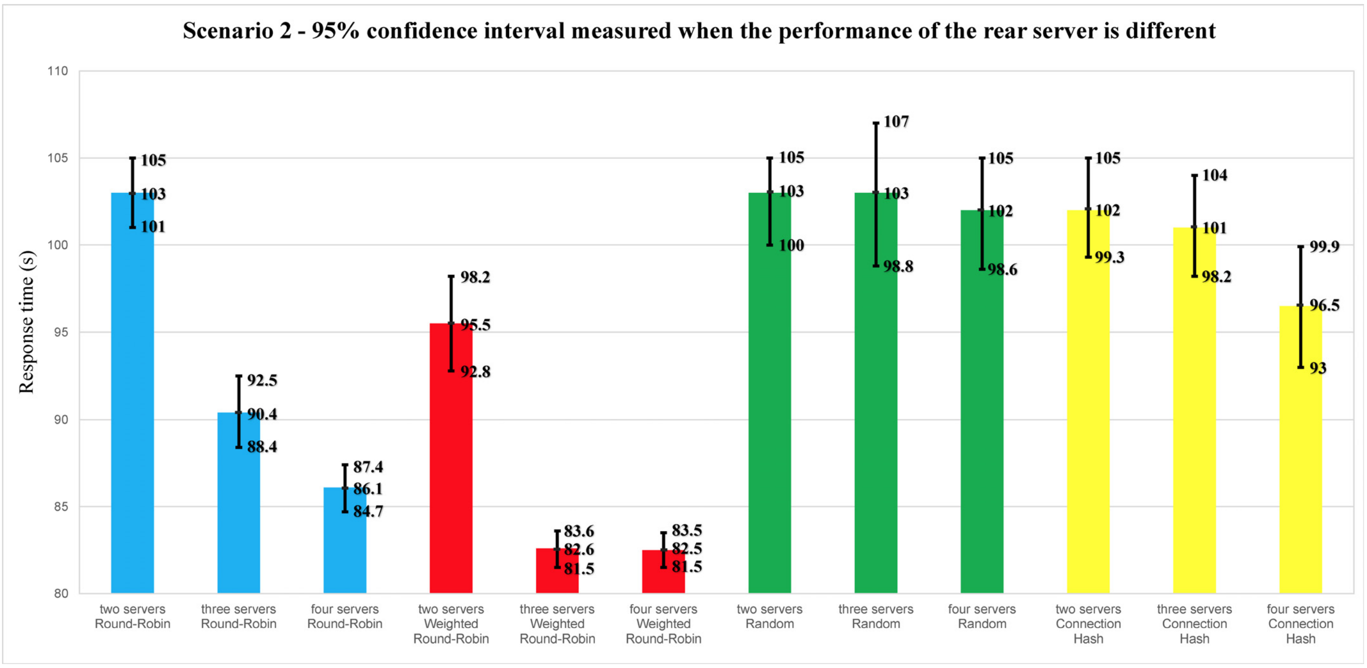


Figure 7. Results of Scenario 2

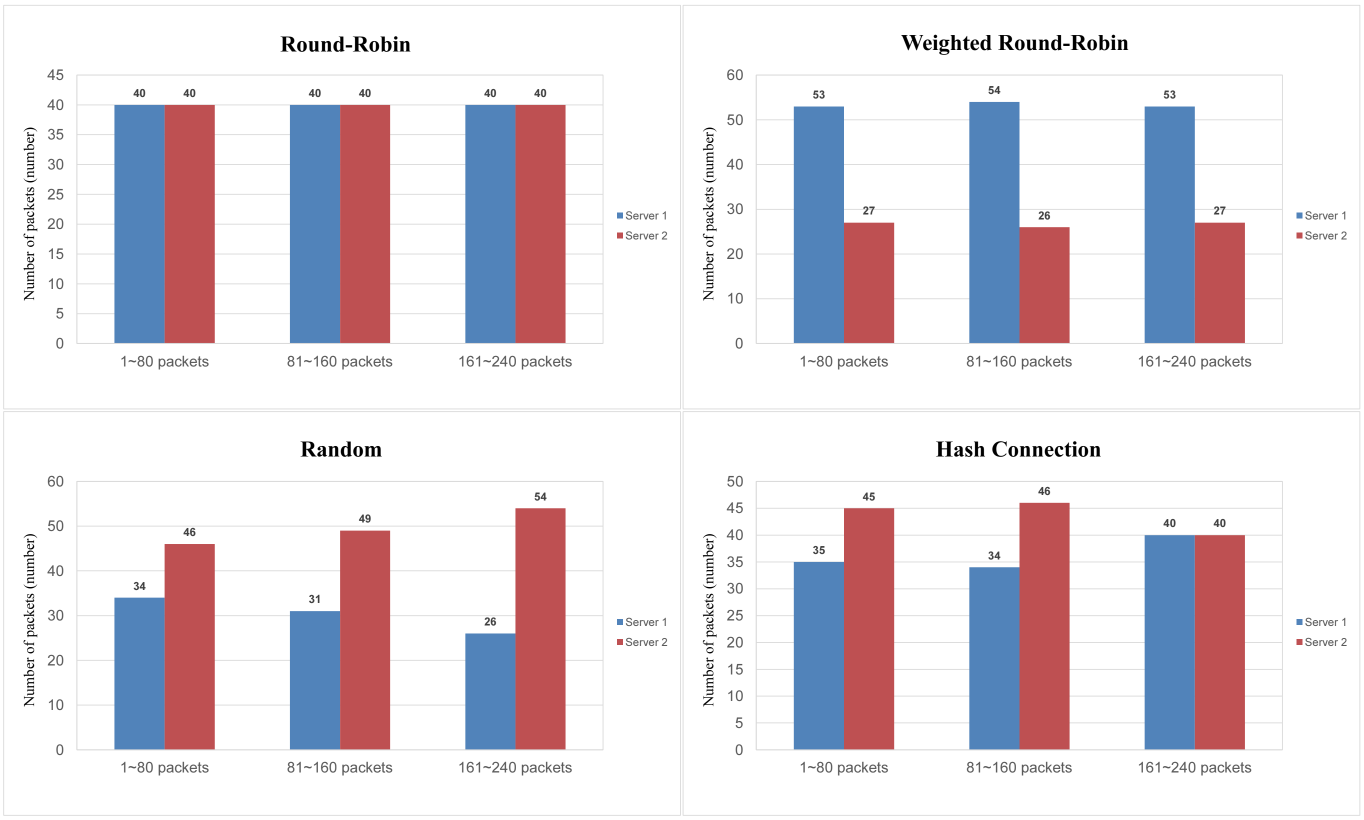


Figure 8. Distributions of packets to servers under Scenario 2

Table 1. Dispatching failure rate for SDN-based load balancer

	Connection Hash	Random	RR	WRR	Legacy SDN
w/ controller	0%	0%	0%	0%	25%
w/o controller	25.4%	24.4%	25%	40%	100%

Table 2. Dispatching failure rate for LVS and HAProxy

	LVS	HAProxy
Failure rate	25%	0.1%

dispatching failure rate is ~25% for three of our proposed P4 load balancers. Since these three algorithms dispatch requests evenly among the servers, and one of four servers is down without the controller knowing it, the failure rate is ~25%. For WRR, the weight for server 4 is 4, indicating that 40% of all requests will be dispatched to server 4; therefore, we expect the failure rate to be 40%. For the Legacy SDN method, when the controller is unavailable and the OpenFlow based switch does not have stateful objects to remember the processing status, load balancing fails entirely. The dispatching failure rate is 100%. Because the LVS has no health-check capability, all requests directed to server 4 will get no response. The failure rate for LVS is 25%. For HAProxy, the dispatching failure rate is similar to our proposed P4 load balancers, i.e., very close to 0%. However, the health-check packets (TCP SYN sent from HAProxy, TCP SYN+ACK sent from server, and TCP RST+ACK sent from HAProxy) are sent out seven times for each server while the HAProxy executes the dispatching-failure test program, adding the overhead to the data communication channel. If we want to decrease the dispatching failure rate, health checks should be executed at shorter intervals, but then the added overhead would tax the available network resources. In our P4 switch, however, we use distinct channels to monitor the health of backend servers; therefore, health-check packets will not add overhead to the data plane.

5 Conclusion and Future Prospects

This study has presented the performance of a P4 switch running four different load-balancing algorithms. These implementations demonstrate that the P4 language's stateful objects such as registers allows a load balancer to function without the requirement of the controller for p4 switches. Conventional OpenFlow-based switches will fail in load balancing if the connection to the control plane fails. If a controller is available, moreover, the controller can perform health checks on the backend servers in conjunction with our P4 load balancer. If a back-end server is down, the P4 load balancer can reroute the request to a functioning backend server. Compared to LVS, a P4 load balancer can provide a lower dispatching failure rate. The P4 load balancer achieves similar performance to that of HAProxy. The P4 load balancer offers the advantage of separating health-check and data packets on different channels, while HAProxy uses the same channel for both types of packets. The P4 load balancer therefore integrates health checks without increasing overhead in the data plane.

Our experimental results also show that if the CPU speeds of the back-end servers are equal, the round-robin algorithm is most efficient. If the back-end servers have disparate CPU speeds, the weighted round-robin load balancer is most efficient. Although the load balancer presented above is compatible only with Layer 4 schemes, we plan to design one for Layer 7 in future work. Moreover, because of the poor performance of the bmv2 software switch [25], we intend to port the code to the NetFPGA hardware platform and measure how much latency is introduced by running P4 code on the hardware. Finally, further design work and comparisons with HAProxy, LVS, and Nginx are expected to improve P4 load balancers in the future.

References

[1] M. Zhang, H. Yu, A New Load Balancing Scheduling Algorithm Based on Linux Virtual Server, *2013 International Conference on Computer Sciences and Applications*, Wuhan, China, 2013, pp. 737-740.

[2] J. E. C. de la Cruz, I. C. A. R. Goyzueta, Design of a High Availability System with HAProxy and Domain Name Service for Web Services, *2017 IEEE XXIV International Conference on Electronics, Electrical Engineering and Computing (INTERCON)*, Cusco, Peru, 2017, pp. 1-4.

[3] Z. Wen, G. Li, G. Yang, Research and Realization of Nginx-based Dynamic Feedback Load Balancing Algorithm, *2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, Chongqing, China, 2018, pp. 2541-2546.

[4] M. Qilin, S. Weikang, A Load Balancing Method Based on SDN, *2015 Seventh International Conference on Measuring Technology and Mechatronics Automation*, Nanchang, China, 2015, pp. 18-21.

[5] J. S. Sabiya, Weighted Round-Robin Load Balancing Using Software Defined Networking, *International Journal of Advanced Research in Computer Science and Software Engineering*, Vol. 6, No. 6, pp. 621-625, 2016.

[6] Y. Zhou, L. Ruan, L. Xiao, R. Liu, A Method for Load Balancing based on Software- Defined Network, *Advanced Science and Technology Letters*, Vol. 45 (CCA 2014), pp. 43-48, 2014.

[7] R. Miao, H. Zeng, C. Kim, J. Lee, M. Yu, SilkRoad: Making Stateful layer-4 Load Balancing Fast and Cheap Using Switching ASICs, *Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'17)*, Los Angeles, CA, USA, 2017, pp. 15-28.

[8] R. Gandhi, Y. C. Hu, M. Zhang, Yoda: A Highly Available Layer-7 Load Balancer, *Eleventh European Conference on Computer Systems (EuroSys'16),* London, UK, 2016, Article 21, pp. 1-16.

[9] J. Ye, C. Chen, Y. H. Chu, A Weighted ECMP Load Balancing Scheme for Data Centers Using P4 Switches, *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*, Tokyo, Japan, 2018, pp. 1-4.

[10] C. H. Benet, A. J. Kassler, T. Benson, G. Pongracz, MP-HULA: Multipath Transport Aware Load Balancing Using

Programmable Data Planes, *2018 Morning Workshop on In-Network Computing*, Budapest, Hungary, 2018, pp. 7-13.

[11] B. Pit-Claudel, Y. Desmouceaux, P. Pfister, M. Townsley, T. Clausen, Stateless Load-Aware Load Balancing in P4, *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, Cambridge, UK, 2018, pp. 418-423.

[12] T. Barbette, C. Tang, H. Yao, D. Kostic, G. Q. Maguire Jr., P. Papadimitratos, M. Chiesa, A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency, *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA, 2020, pp. 667-683

[13] *Emulator for Rapid Prototyping of Software Defined Networks*, https://github.com/mininet/mininet.

[14] *The Reference P4 Software Switch*, https://github.com/p4lang/behavioral-model.

[15] *myP4Dockernet (P4 switch+Mininet+Docker Host)*, http://csie.nqu.edu.tw/smallko/sdn/p4-dockernet.htm.

[16] S. Jouet, R. Cziva, D. P. Pezaros, Arbitrary Packet Matching in OpenFlow, *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*, Budapest, Hungary, 2015, pp. 1-6.

[17] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, D. Walker, P4: Programming Protocol-independent packet Processors, *ACM SIGCOMM Computer Communication Review*, Vol. 44, No. 3, pp. 87-95, July, 2014.

[18] N. Mckeown, J. Rexford, *Clarifying the Differences between P4 and OpenFlow*, https://p4.org/p4/clarifying-the-differences-between-p4-and-openflow.html, 2016.

[19] C. H. Ke, *Connection Hash Load Balancer*, http://csie.nqu.edu.tw/smallko/sdn/LBP4.htm.

[20] C. H. Ke, *Random Load Balancer*, http://csie.nqu.edu.tw/smallko/sdn/randomLB.htm.

[21] C. H. Ke, *Round-Robin Load Balancer*, http://csie.nqu.edu.tw/smallko/sdn/RRP4.htm.

[22] C. H. Ke, *Weighted Round-Robin Load Balancer*, http://csie.nqu.edu.tw/smallko/sdn/wrrlb.htm.

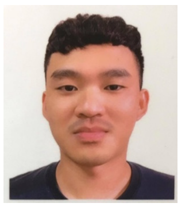
[23] *Apache HTTP Server Benchmarking Tool*, https://httpd.apache. org/docs/2.4/programs/ab.html.

[24] *Specify a Container'S Resources*, https://docs.docker.com/config/containers/resource_constraints.

[25] Y. Iozzelli, L. Rizzo, G. Lettieri, *Performance Improvements on the P4 Software Switch*, https://core.c.uk/download/pdf/79622350.pdf, 2016.

Biographies

Chih-Heng Ke received his B.S. and Ph.D. degrees in electrical engineering from National Cheng-Kung University, in 1999 and 2007. He is an associate professor at the Department of Computer Science and Information Engineering in National Quemoy University, Kinmen, Taiwan. His current research interests include multimedia communications, wireless networks, and software defined networks.



Shih-Jung Hsu received his B.S. degrees Master of Information Technology and Application, National Quemoy University, Kinmen, Taiwan, in 2019. His research interests include computer networks and software defined networks.