Volume XXIII 2020
ISSUE no.2
MBNA Publishing House Constanta 2020

Scientific Bulletin of Naval Academy

SBNA PAPER • OPEN ACCESS

Implementing a Software Load Balancer with a Genetic Algorithm

To cite this article: Alexandru Curcă and Eugen Petac, Scientific Bulletin of Naval Academy, Vol. XXIII 2020, pg.177-184.

Available online at www.anmb.ro

ISSN: 2392-8956; ISSN-L: 1454-864X

doi: 10.21279/1454-864X-20-I2-024

Implementing a Software Load Balancer with a Genetic Algorithm

A. Curcă[1], E. Petac[2]

[1,2]"Ovidius" University of Constanța, 124, Mamaia Blvd., Constanta, Romania
[1]alexandrucurca23@gmail.com, [2]epetac@univ-ovidius.ro

Abstract. In the context of network evolution, concepts like Software Defined Networking (SDN) and Network Functions Virtualisation (NFV) appeared on the market. Network virtualization permits the implementation of routers, switches and load balancers in software and separation of control plane and data plane brings easier configuration, implementation and scalability. The monolithic design of traditional network devices can be changed by implementing new algorithms which will improve the overall system performance. An example is our Software Load Balancer with a Genetic Algorithm. The code written in Python is functional through the POX Controller and the advantages of evolutionary algorithms make this implementation an innovative solution for dynamically modified topologies.
Key words: Load Balancer, Genetic Algorithm, Software Defined Network, Mininet.

1. Introduction

In the field of computer science, load balancing [1] uses multiple servers to support a single service, such as a high-volume web site. It can increase the availability of web sites and web-based applications by optimizing resource use, maximizing throughput and minimizing response time. The network component responsible for this task is the load balancer, which can be hardware or software implemented. In terms of network security, its location is usually set in Demilitarized Zone (DMZ) [2]. The distribution of traffic is made based on factors such as the number of current connections to the server farm, the processor utilization, the network connection or the server performance. In a network topology, a load balancer uses a public virtual IP. This one is the destination of client requests, which are redirected to one of the web farm servers. Servers in web farm have private IP addresses [3].

The high rate of traffic in cloud systems brought the concepts of Software Defined Network (SDN) and Network Function Virtualization (NFV) [4]. The control plane of the network devices (switches or routers) is implemented in independent controllers by using protocols like OpenFlow [5] as interface. At the same time, the network devices can be fully implemented in software for managing traffic between virtual machines. With these new concepts, a wide range of advantages appeared, among which we list: cost reduction, easier network management (a new policy can be implemented with a single script), easier implementation of new protocols.

There are a number of algorithms and solutions for implementing a Load balancer in a SDN environment among which we list: Weighted Least Connection [6], Round Robin [7], Weighted Round Robin [8] or random. A comparison between the current solution and those listed above is depicted in Section 3 of this paper. Genetic algorithms are used in various fields, like timetabling, scheduling problems or engineering [9]. In the field of computer networks is worth mentioning the research in finding the shortest path in packet switching networks [10]. The biggest advantage of our solution (presented in Section 2) comes from the ability to adapt to a dynamic environment in which the change of nodes and connections is difficult to anticipate.

2. Proposed Solution

The current solutions is made possible with Mininet [11], which is a framework capable of emulating SDN compatible network devices and Linux hosts. Those network devices can be programmed by an SDN controller. There are various solutions [12]-[13], one of them being POX, a Python based open source implementation for on Windows, Mac OS, or Linux used primarily for research and education [14]. The topology used for the current paper is shown in figure 1. The following equipment was emulated within the topology: 5 hosts with HTTP client role (h11-15 i), 10 hosts with HTTP server role (h1-10), one switch (s1) and one load balancer (s2). Mininet permits the manual setting of bandwidth and delay for every connection between devices, thus we set different values for the connections between load balancer and servers for the purpose of analyzing the evolution of the genetic algorithm. It should be noted that, both the switch and the load balancer are simple OpenFlow switches, the desired functionality being obtained from the interaction with the POX controller (c0). S1 is assigned the virtual IP address 10.0.1.1, which is the public address of the server farm. In the server farm, h1-h10 are assigned the IP address in range 10.0.0.1-10.0.0.10.

The load balancing will be done by distributing every packet that came from a new TCP connection [3] to a server in the server farm after the learn phase of the genetic algorithm. The output of the genetic algorithm will be made up of a list with the right order of task distributions to servers. The protocol used between load balancer(S2) and POX Controller is OpenFlow.

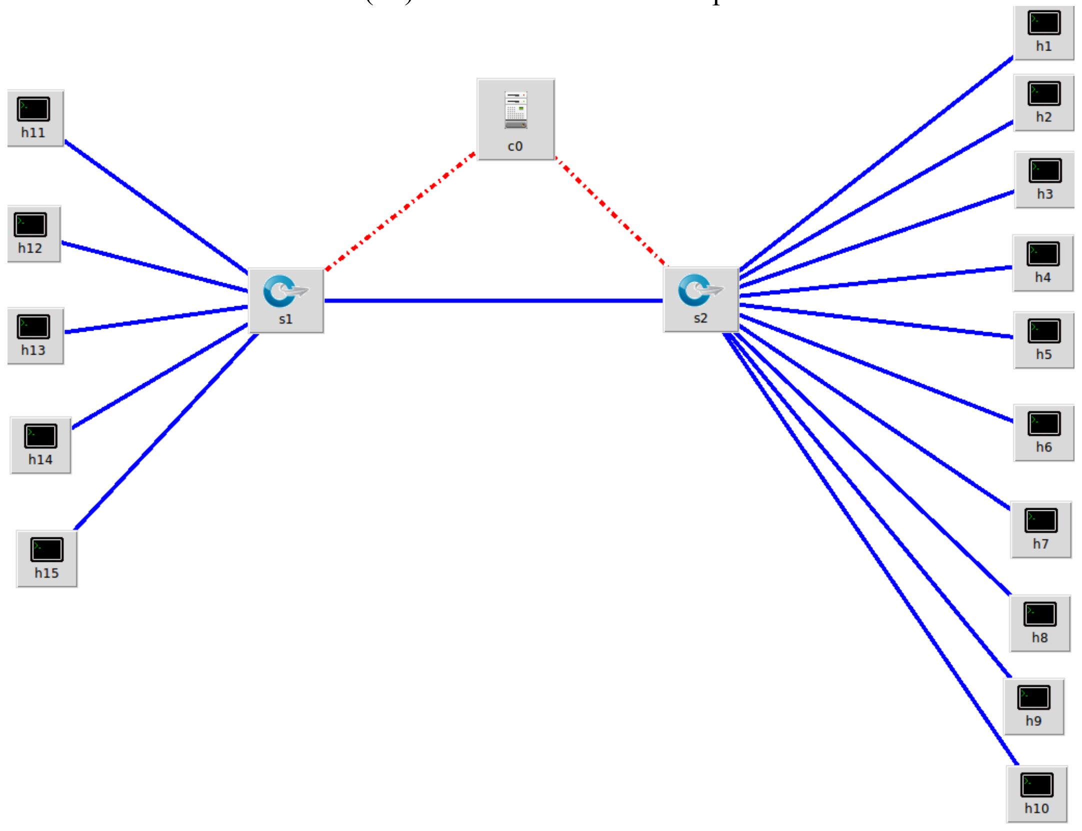


Figure 1. Used topology. Source: Authors' contribution.

The distribution will be done by sending every packet that came from a new TCP connection to a server in the server farm after the learn phase of the genetic algorithm. The output of the genetic algorithm will be made up of a list with the right order of task distributions to servers.

The pseudocode of the genetic algorithm is as follows:

```
INITIALIZATION  generate_P(n) // P(n)-population with n chromosomes
                evaluate_P(n)
ITERATIONS      begin t:=0
                while (t < generations_nr) //number of generations
                    if (cross_condition)
                            cross_P(n)
                    endif
                    if(mutate_condition)
                            mutate_P(n)
                    endif
                 t:=t+1
                 endwhile
```

The function `generate_P(n)` initializes the population P(n) with n chromosomes and the function `evaluate_P(n)` applies the fitness function for every member in the initial population. The variable `generation_nr`, represents the number of iterations the algorithm will run. The `cross_condition/mutate_condition` establishes if during the current generation a cross/mutation will be made. The two functions `cross_P(n)` and `mutate_P(n)` will do the cross or mutation operation. These two operations include the evaluation of the current population and the selection of chromosomes.

During the cross phase, two chromosomes are selected and crossed resulting a new member of population. This one will be evaluated and will replace the worst chromosome from population. The mutation phase will select a single chromosome on which we will replace a server with a randomly chosen one. The resulting chromosome will also be evaluated and will replace the worst chromosome from population.

The components of the genetic algorithm, adapted to the current solution, are as follows:

- CHROMOSOME REPRESENTATION
  The coding of a chromosome is a list $c_j = [s_{i1}, s_{i2}, \dots, s_{in}]$ each s_i is a server from the server farm. The fitness value [15] for a chromosome is the time, in seconds, it takes a host to get a sequence of consecutive packets from the respective servers. For this implementation, the length of a chromosome is 30 servers.
- POPULATION REPRESENTATION
  The population $P(N) = \{(c_1, t_1,), (c_2, t_2,) \dots (c_N, t_N,)\}$, is a Python dictionary, each tuple $(c_i, t_i,)$ representing a chromosome mapping the fitness value t_i. N is the number of chromosomes in the population. The purpose of the algorithm is to find the chromosome with the smallest responding time (the best fitness value). This implementation's population includes 50 chromosomes.
- INITIAL POPULATION
  The initial population is made with 50 chromosomes random generated.
- CROSSOVER:
  By picking two chromosomes, father chromosome and mother chromosome, from the population and generating a random number between 0 and chromosome's length, it will result a new chromosome (child chromosome) as shown in figure 2. First part of the result is made with father's servers (from 0 to random value -1) and the second part with mother's (from random value to length of chromosome-1). During both phases, crossover and mutation, the worst chromosome of the population will be replaced by the resulting one.

- MUTATION
  The mutation is possible by changing a random chosen server from a picked chromosome with a new random server, as shown in figure 3.
- SELECTION:
  For a *N*-chromosome population, there will be made 5 distinct classes with "*N*/5" members (if *N* is not divisible by 5, the rest of the chromosomes will be distributed one-by-one starting with the first class). Every chromosome is captured in one of the five classes for selection.

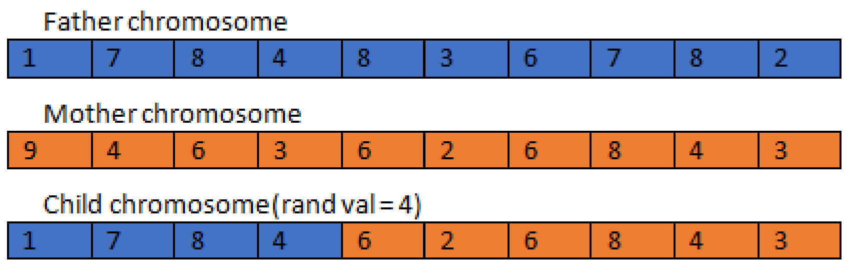


Figure 2. Crossover example. Source: Authors' contribution.

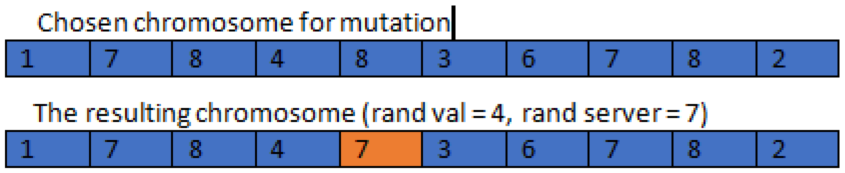

Chosen chromosome for mutation
1 7 8 4 8 3 6 7 8 2
The resulting chromosome (rand val = 4, rand server = 7)
1 7 8 4 7 3 6 7 8 2

Figure 3. Mutation example. Source: Authors' contribution.

The distribution of selection chances is shown in figure 4 and obtained with the following rule: In a population of N chromosomes, for a fitness value f_i and a population member i, the probability of selection will be $p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$.

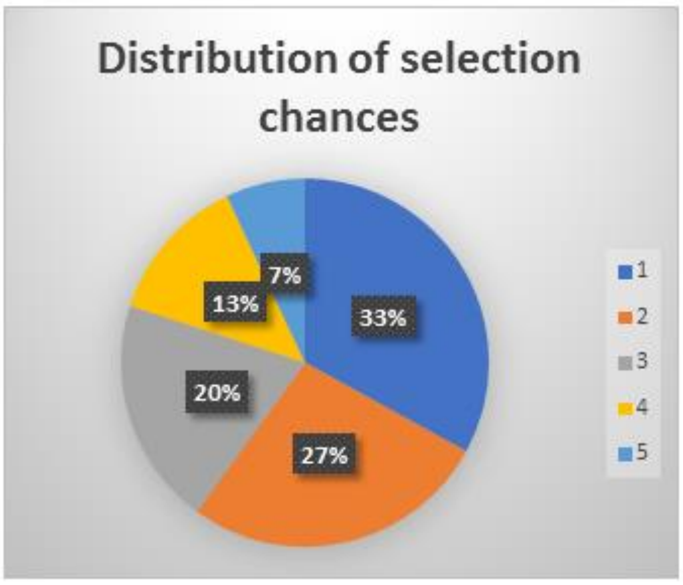
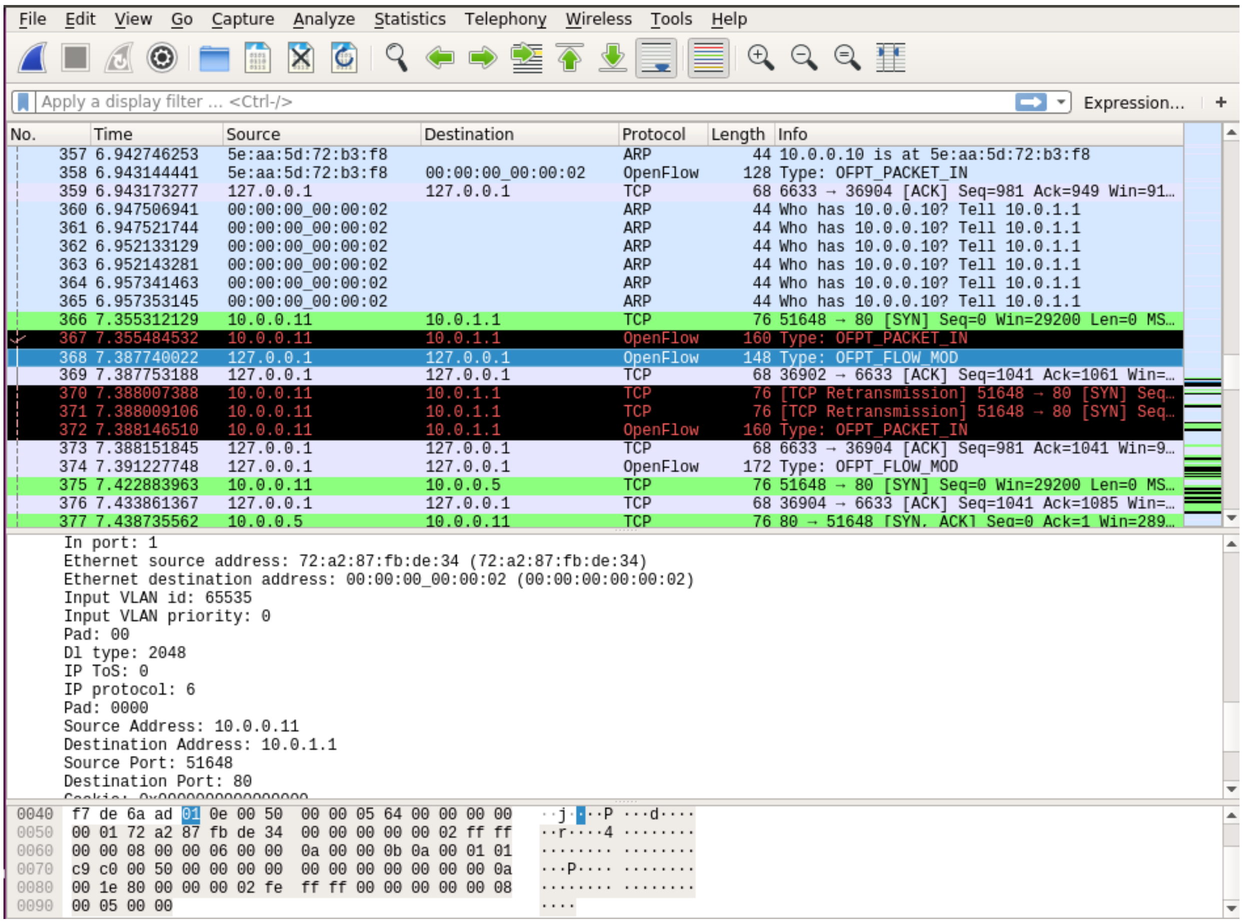


Figure 3. Distribution of selection chances. Source: Authors' contribution.

- GENETIC ALGORITHM PARAMETERS
  There are various scientific opinions regarding the tuning of genetic algorithm parameters [15]. For the current project we set the following: populations size - 50 chromosomes, length of the chromosomes – 30 servers, cross rate – 50%, mutation rate – 1%, number of generations – 50.

3. Experimental Results

During the debugging phase we captured various OpenFlow packets in Wireshark [16] as shown in figure 5. Every new packet reaching the load balancer will generate an OFPT_PACKET_IN [17] sent to controller. The POX controller will update the balancer's forwarding table with an OFPT_FLOW_MOD [18] packet (packets 367 and 368 from figure 5).



```
No.     Time         Source              Destination         Protocol  Length Info
    357 6.942746253  5e:aa:5d:72:b3:f8                       ARP           44 10.0.0.10 is at 5e:aa:5d:72:b3:f8
    358 6.943144441  5e:aa:5d:72:b3:f8   00:00:00_00:00:02   OpenFlow     128 Type: OFPT_PACKET_IN
    359 6.943173277  127.0.0.1           127.0.0.1           TCP           68 6633 → 36904 [ACK] Seq=981 Ack=949 Win=91…
    360 6.947506941  00:00:00_00:00:02                       ARP           44 Who has 10.0.0.10? Tell 10.0.1.1
    361 6.947521744  00:00:00_00:00:02                       ARP           44 Who has 10.0.0.10? Tell 10.0.1.1
    362 6.952133129  00:00:00_00:00:02                       ARP           44 Who has 10.0.0.10? Tell 10.0.1.1
    363 6.952143281  00:00:00_00:00:02                       ARP           44 Who has 10.0.0.10? Tell 10.0.1.1
    364 6.957341463  00:00:00_00:00:02                       ARP           44 Who has 10.0.0.10? Tell 10.0.1.1
    365 6.957353145  00:00:00_00:00:02                       ARP           44 Who has 10.0.0.10? Tell 10.0.1.1
    366 7.355312129  10.0.0.11           10.0.1.1            TCP           76 51648 → 80 [SYN] Seq=0 Win=29200 Len=0 MS…
    367 7.355484532  10.0.0.11           10.0.1.1            OpenFlow     160 Type: OFPT_PACKET_IN
    368 7.387740022  127.0.0.1           127.0.0.1           OpenFlow     148 Type: OFPT_FLOW_MOD
    369 7.387753188  127.0.0.1           127.0.0.1           TCP           68 36902 → 6633 [ACK] Seq=1041 Ack=1061 Win=…
    370 7.388007388  10.0.0.11           10.0.1.1            TCP           76 [TCP Retransmission] 51648 → 80 [SYN] Seq…
    371 7.388009106  10.0.0.11           10.0.1.1            TCP           76 [TCP Retransmission] 51648 → 80 [SYN] Seq…
    372 7.388146510  10.0.0.11           10.0.1.1            OpenFlow     160 Type: OFPT_PACKET_IN
    373 7.388151845  127.0.0.1           127.0.0.1           TCP           68 6633 → 36904 [ACK] Seq=981 Ack=1041 Win=9…
    374 7.391227748  127.0.0.1           127.0.0.1           OpenFlow     172 Type: OFPT_FLOW_MOD
    375 7.422883963  10.0.0.11           10.0.0.5            TCP           76 51648 → 80 [SYN] Seq=0 Win=29200 Len=0 MS…
    376 7.433861367  127.0.0.1           127.0.0.1           TCP           68 36904 → 6633 [ACK] Seq=1041 Ack=1085 Win=…
    377 7.438735562  10.0.0.5            10.0.0.11           TCP           76 80 → 51648 [SYN, ACK] Seq=0 Ack=1 Win=289…
```

```
        In port: 1
        Ethernet source address: 72:a2:87:fb:de:34 (72:a2:87:fb:de:34)
        Ethernet destination address: 00:00:00_00:00:02 (00:00:00:00:00:02)
        Input VLAN id: 65535
        Input VLAN priority: 0
        Pad: 00
        Dl type: 2048
        IP ToS: 0
        IP protocol: 6
        Pad: 0000
        Source Address: 10.0.0.11
        Destination Address: 10.0.1.1
        Source Port: 51648
        Destination Port: 80
        Cookie: 0x0000000000000000
```

```
0040  f7 de 6a ad 01 0e 00 50  00 00 05 64 00 00 00 00   ··j···P ···d····
0050  00 01 72 a2 87 fb de 34  00 00 00 00 00 02 ff ff   ··r····4 ········
0060  00 00 08 00 00 06 00 00  0a 00 00 0b 0a 00 01 01   ········ ········
0070  c9 c0 00 50 00 00 00 00  00 00 00 00 00 00 00 0a   ···P···· ········
0080  00 1e 80 00 00 00 02 fe  ff ff 00 00 00 00 00 08   ········ ········
0090  00 05 00 00                                        ····
```

Figure 5. Forwarding table modification when receiving an unknown packet. Source: Authors' processing using Wireshark.

In figure 6 we analyzed the tuple (worst time, medium time, best time) per population. The X - axis represents the numbers of modification suffered by the population during the learning phase and the Y-axis corresponds to time in seconds. Even if the best chromosome didn't change from the initial population (for the current plotting), there is a visible positive variation of worst chromosome and medium time per population.

A comparison between various load In Mininet we set different connections between Load Balancer and servers in server farm. For servers 1 and 2 - a bandwidth of 2 Mbps and a delay of 25 ms were established, for servers 3 and 4 - 4 Mbps and 20 ms, for servers 5 and 6 - 6 Mbps and 15ms, for servers 7 and 8 – 8 Mbps and 10 ms , and for servers 9 and 10 – 10 Mbps and 5ms. Even if all servers were considered equal in the beginning of the algorithm, during the learning phase, the servers with a superior connection received more packets as can be seen in figure 7. It is worth mentioning server 10(10.0.0.10 – colored pink) which, in the first population received 156 packets and in the last one received 176. On the other side, server 1 (10.0.0.1 – colored brown) started with 158 packets and because of lack of performance ended with 142 in the final population.

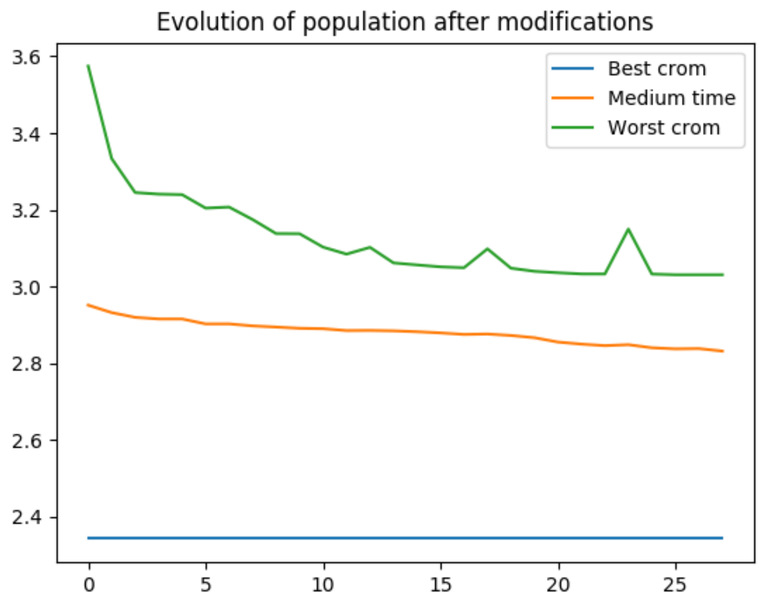


Figure 6. Time evolution of population after modifications. Source: Authors' contribution.

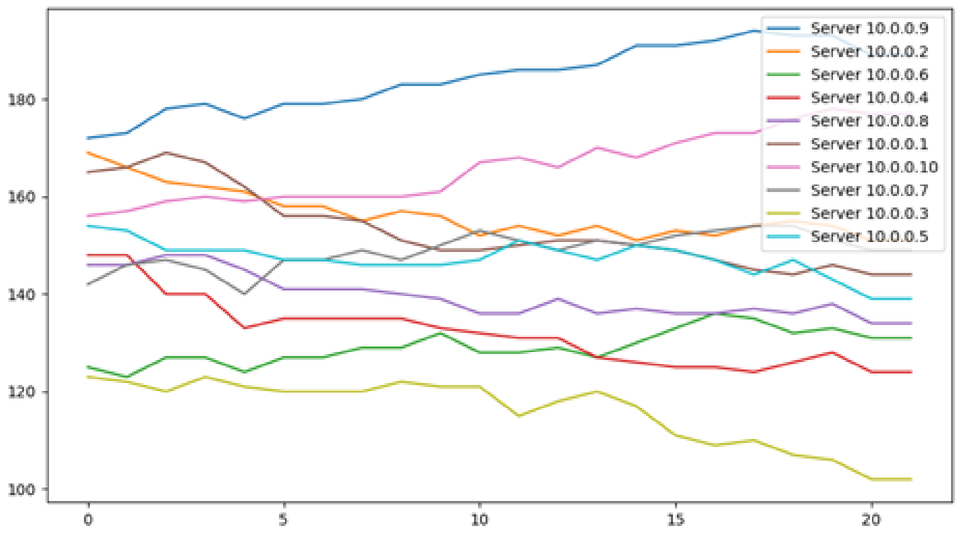


Figure 7. Evolution of packet distribution per server. Source: Authors' contribution.

A comparison between various load balancing algorithms was made by sending 1000 packets to the virtual address. The experimental results are shown in figure 8. The round-trip time of the user sent packets, where distribution is made by the genetic algorithm, is 84.7 second, while for the weighted least-bandwidth is almost 95.2 and for the round-robin is 100.

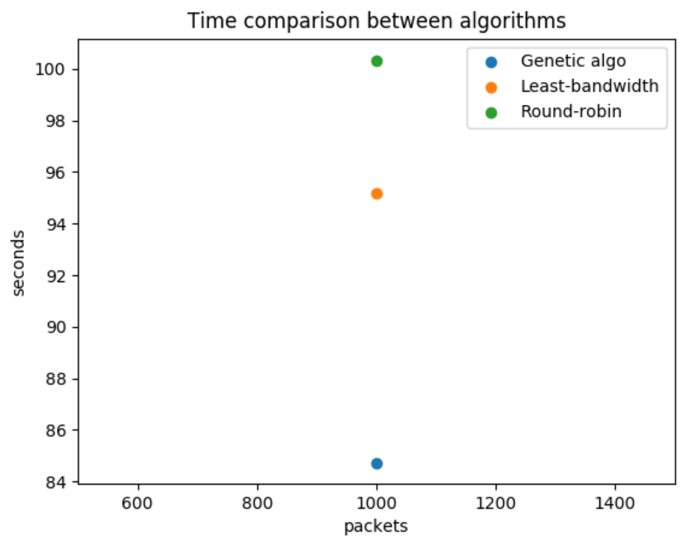


Figure 8. Time comparison between algorithms. Source: Authors' contribution.

4. Conclusions

The current project could represent a solution for a dynamically-changing server farm topology, where the performance of different servers is hard to evaluate. Even if the performance is improved with the genetic algorithm, there is a drawback represented by the time of learning. There is a compromise between the two aforementioned, due to the fact that a higher performance requires larger chromosome length, population size and generations number.

Another advantage of our load balancer is OpenFlow-compatibility. Unlike the hardware solutions where algorithms are embedded in the device, the current one can be changed dynamically and easily deployed. Also, the controller can supply various network protocols for other devices in the topology, by knowing the device's Data Path ID. In this mode, the controller can be used as control plane for entire topology. On the software side of things, the application is written using well documented languages and technologies, respecting programming principles.

References

[1] Gibson, Darril. *CompTIA Security+: Get Certified Get Ahead SY0-201 Study Guide*. YCDA, LLC Publishing (2009).

[2] Webb, Jack. "Network Demilitarized Zone (DMZ)." (2014). Available at: http://infosecwriters.com/Papers/jwebb_network_demilitarized_zone.pdf. [Accessed March 2019].

[3] Petac, Eugen. "Networks and Distributed Systems." *Distributed Multimodal Virtual Environments*, Publisher: Pro Universitaria, Bucharest, Romania (2015): 103-206.

[4] Vilalta, Ricard, et al. "The SDN/NFV cloud computing platform and transport network of the ADRENALINE testbed." *Proceedings of the 2015 1st IEEE Conference on Network Softwarization* (NetSoft). IEEE (2015).

[5] Azodolmolky, Siamak. *Software defined networking with OpenFlow*. Packt Publishing Ltd (2013).

[6] Choi, DongJun, Kwang Sik Chung, and JinGon Shon. "An improvement on the weighted least-connection scheduling algorithm for load balancing in web cluster systems." *Grid and distributed computing, control and automation*. Springer, Berlin, Heidelberg (2010): 127-134.

[7] Youm, Dong Hyun, and Ravi Yadav. "Load Balancing Strategy using Round Robin Algorithm." *Asia-pacific Journal of Convergent Research Interchange* 2.3 (2016): 1-10.

[8] Sabiya, Japinder Singh. "Weighted round-robin load balancing using software defined networking." *International Journal of Advanced Research in Computer Science and Software*

Engineering (IJARCSSE) 6.6 (2016).

[9] Tomoiagă, Bogdan, et al. "Pareto optimal reconfiguration of power distribution systems using a genetic algorithm based on NSGA-II." Energies 6.3 (2013): 1439-1455.

[10] Chaudhary, Anu, and Neeraj Kumar Pandey. "GENETIC ALGORITHM FOR SHORTEST PATH IN PACKET SWITCHING NETWORKS." *Journal of Theoretical & Applied Information Technology* 29.2 (2011).

[11] Mininet. Available at: https://github.com/mininet/mininet/wiki/Introduction-to-mininet. [Accessed March 2019].

[12] OpenDaylight Foundation. Available at: https://www.opendaylight.org/. [Accessed March 2019].

[13] Beacon. Available at: https://openflow.stanford.edu/display/Beacon/Home. [Accessed March 2019].

[14] POX. Available at: https://github.com/noxrepo/pox. [Accessed March 2019].

[15] Mitchell, Melanie. *An introduction to genetic algorithms*. MIT press (1998).

[16] Wireshark. Available at: https://www.wireshark.org/. [Accessed March 2019].

[17] PacketIn. Available at: http://flowgrammable.org/sdn/openflow/message-layer/packetin/. [Accessed March 2019].

[18] FlowMod. Available at: http://flowgrammable.org/sdn/openflow/message-layer/flowmod/. [Accessed March 2019].