

Load Balancing in SDN

Team members:

Mithesh.A - 2019503533

B.Niveditha - 2019503541

Rubak Preyan.G - 2019503553

S. Yogeeswar - 2019503573

TOOLS USED

Mininet

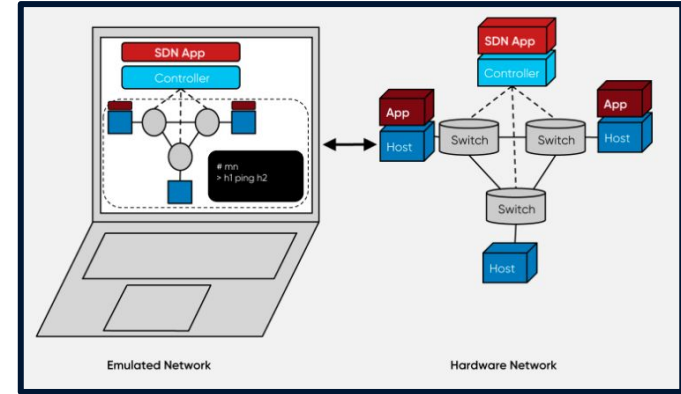
POX

Gnuplot

Apache bench

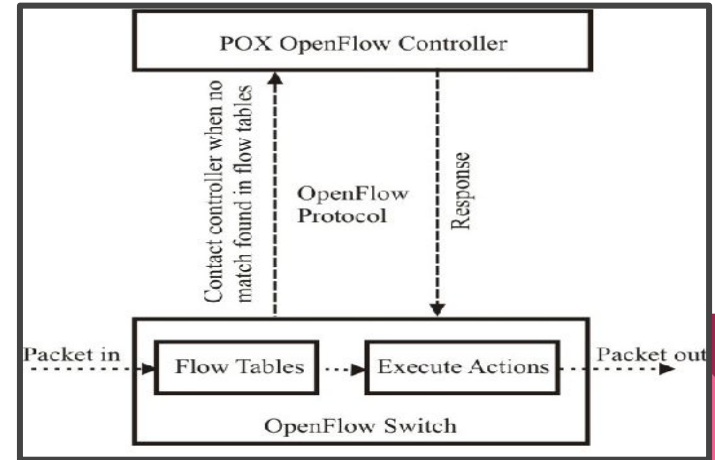
Mininet

Mininet is a software-defined network (SDN) virtual test bed and development environment (SDN).



POX

POX is a Python-based software-defined networking (SDN) control application development platform, such as OpenFlow SDN controllers.



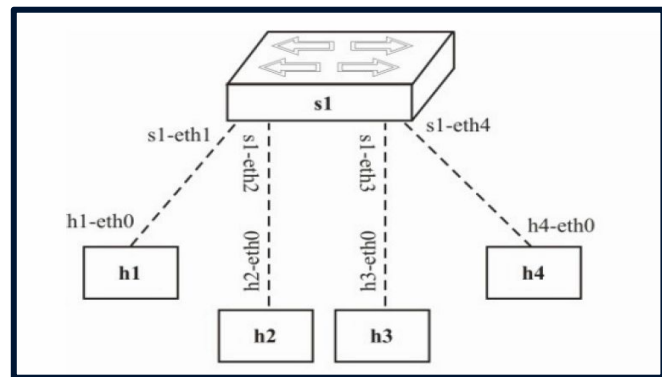
Apache bench

Apache Bench (ab) is a load testing and benchmarking tool for Hypertext Transfer Protocol (HTTP) server. It can be run from command line and it is very simple to use.

Single Topology

When there is one openflow switch and k hosts, it's called as single topology. It also establishes a connection between the switch and the k hosts.

Command: `$sudo mn --topo single,4
--controller=remote`



General Execution Order

Running pox:

`./pox.py log.level --DEBUG rand --ip=10.0.1.1 --servers=server1,server2,...`(enter as many nodes as you want)

Here the word 'rand' represents the algorithm used. This should be changed to round or least to attain the results for Round robin and Least count algorithm respectively .

Running mininet:

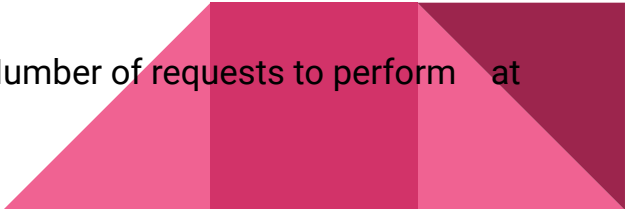
`sudo mn --topo single`(Type of topology),`8`(number of hosts) `--mac --arp --controller=remote`

`xterm h1 h2 h3 h4 h5....`(nodes)

Run the following code for nodes which are considered as servers:

`python -m SimpleHTTPServer 80`

`ab`(Apache HTTP server benchmarking tool) `-n 100`(number of requests) `-c 10`(Number of requests to perform at a time) `-g sample.csv`(gnuplot file name) `http://10.0.1.1/pdf.pdf >>filename.txt`



ALGORITHMS

Implementation and Result
Analysis

RANDOM ALGORITHM

ROUND ROBIN ALGORITHM

LEAST CONNECTION ALGORITHM

WEIGHTED ROUND ROBIN
ALGORITHM

Code can be found [here](#).

Random Algorithm

Algorithm

Input : Request, Set of available servers.

Output : Request allocation to servers.

1. Begin
2. Flowtime = 20-80 seconds
3. Get list of servers
4. Whenever a new client request comes in, pick a random server from the list of servers using:
 chosen_server =
 random.choice(SV_HOSTS)
5. Assign the request to a random chosen server
6. End

The implementation for scheduling resource requests from client nodes and allocating servers to the client nodes using Random Selection Algorithm is shown here.

In a given set of available servers, the Load Balancer arbitrarily chooses one server node for forwarding the request from client nodes.



Round Robin Algorithm

Algorithm:

Input: Request, Set of available servers.

Output: Efficient request allocation to servers.

Begin

1. Flowtime = 20-80 seconds
2. Get list of servers
3. Whenever a new client request comes in
Pick a server from the list of servers
using

```
self.last_server_idx=(self.last_server_idx+1)% len(sv_hosts)
```

```
chosen_server = sv_hosts[self.last_server_idx]
```

4. Request is forwarded to each server in
a cyclic fashion

End

Here, each server node receives the requests from client nodes in a circular pattern. Load Balancer allocates the request to several servers in a round robin manner. Also the specifications of servers such as CPU, RAM etc come into context so that the workload can be properly assigned to each of the servers.

Here, a single device is capable enough to process the same number of requests. Round Robin algorithm is considered best for clusters consisting of servers with identical specs.



Least Connection Algorithm

Algorithm:

Input : Request, Set of available servers.

Output : Request allocation to servers.

1. Begin
2. Flowtime = 20-80 seconds
3. Get list of servers
4. Get number of active connections of respective servers.
5. Whenever a new client request comes in, pick a server with least number of active connections from the list of servers.
6. After choosing a server number of connection is incremented.
7. End

Least Connection, otherwise known as Least Count load balancing is a dynamic load balancing algorithm where client requests are distributed to the application server with the least number of active connections at the time the client request is received.



Weighted Round Robin Algorithm

Algorithm:

Input: Request, weight, Set of available servers.

Output: Efficient request allocation to servers.

Begin

1. Flowtime = 20-80 seconds
2. Get list of servers
3. Whenever a new client request comes in Pick a server from the list of servers using round robin strategy but to follow the weights, i.e. Server with higher weight get higher number of request at a time.
4. Request is forwarded to each server in a cyclic fashion

End

A more complex load balancing option is Weighted Round Robin. This strategy, like standard Round Robin, allows you to point records to several IP addresses, but it also allows you to distribute weight based on the needs of your domain.

This is accomplished by establishing DNS record pools that are cycled based on the values.

Multiple passive packet sources are connected to a single active packet sink using the Weighted Round-Robin Scheduler module.



Execution and Results of Existing Solutions

Various constraints have been modified and the outcomes are compiled and plotted into graphs.

Data for the graphs is found [here](#)

Throughput Vs Number Of Request

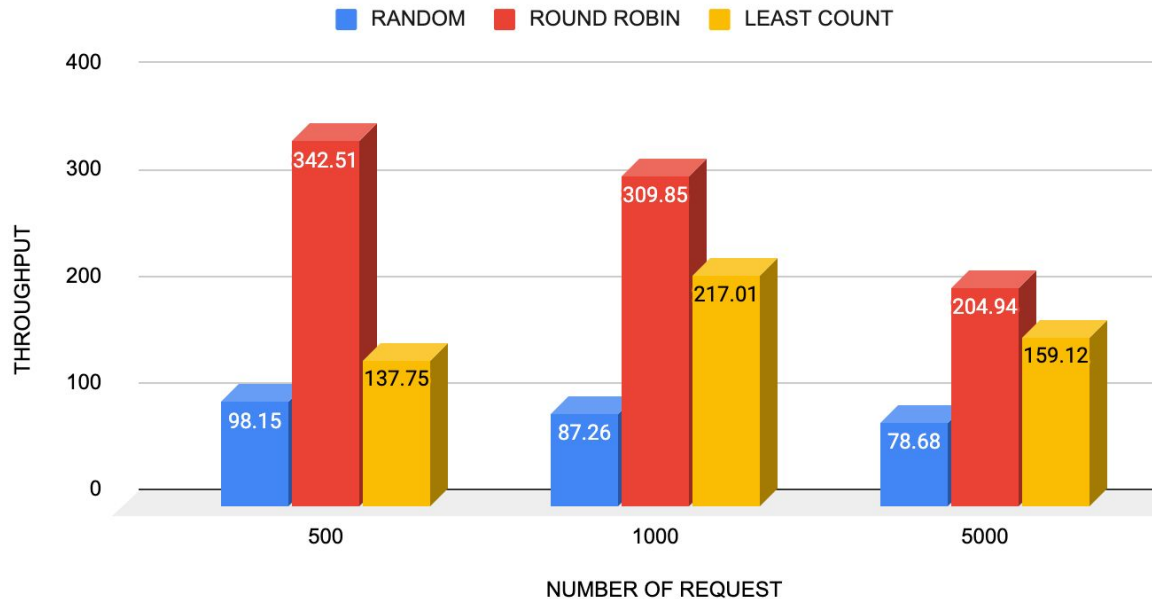
- Throughput is basically the number of request per second. Throughput is calculated for 3 algorithms by changing number of requests.
- Throughput is obtained using apache bench load testing tool using the code below, where the number_of_request denotes the number of requests:

```
ab -n number_of_request -c 10 http://10.0.1.1/requestfile.txt >> data_algoname.txt
```

- All the data are redirected to text files which is used to plot graphs.



Throughput Vs Number of Requests



From the graph we can observe that though throughput of all the algorithms decreases with increase in number of request, round robin has higher throughput followed by least count and random algorithms in order.

Link to Data: [random](#), [round](#), [least](#)

Throughput Vs File Size

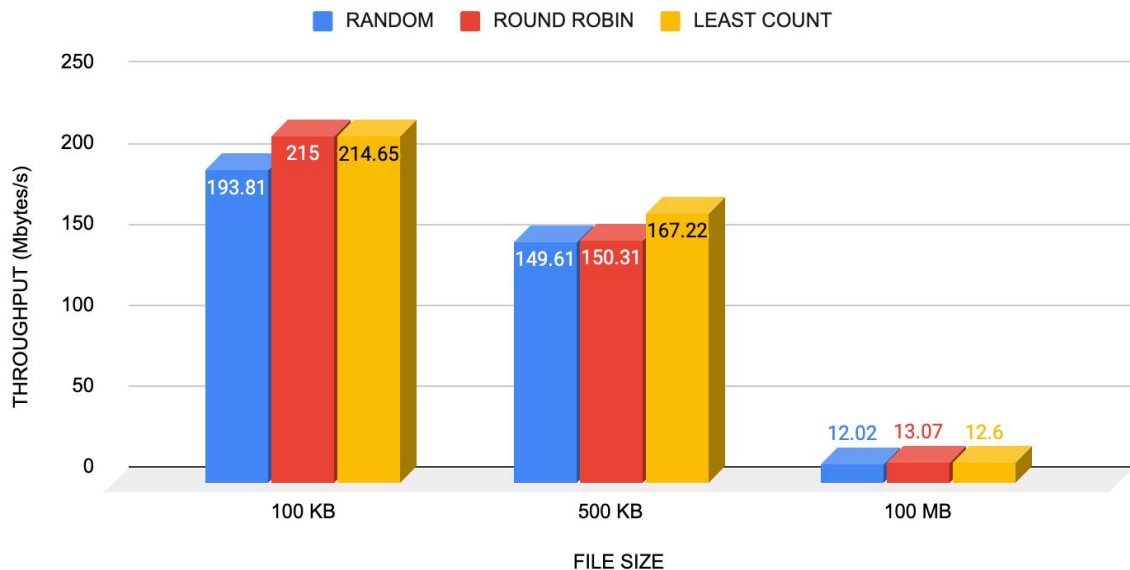
- Throughput is calculated for 3 algorithms by changing number of requests.
- 3 files of different sizes 100kb, 500kb, 100mb are created for the test.
- Throughput is procured using apache bench load testing tool using the following code, where the number_of_request denotes number of request and size in file_size denotes (100kb, 500kb, 100mb):

```
ab -n number_of_request -c 10 http://10.0.1.1/file_size.txt >> data_algoname.txt
```

- All the data are redirected to text files which is used to plot graphs.



Throughput Vs File Size



It is very obvious from the graph that throughput of all the algorithms decreases with increase in file size. All the algorithms seem to have same throughput but least count has bit higher throughput than other algorithms

Link to data: [random](#), [round](#), [least](#)

Variation in Concurrency Level

Concurrency Level is the number of requests that are done simultaneously. Variation in concurrency levels affects the value of throughput.


This can be done by changing the values in the command as follows:

```
ab -n 100 -c 10 -g con10rand.csv http://10.0.1.1/testfile.txt >>concurrency_level_10.txt
```

```
ab -n 100 -c 20 -g con20rand.csv http://10.0.1.1/testfile.txt >>concurrency_level_20.txt
```

```
ab -n 100 -c 30 -g con30rand.csv http://10.0.1.1/testfile.txt >>concurrency_level_30.txt
```

These commands are run for all 3 algorithms for each value of concurrency level and the results are stored in a text file. The values obtained after simulation are condensed into a graph.



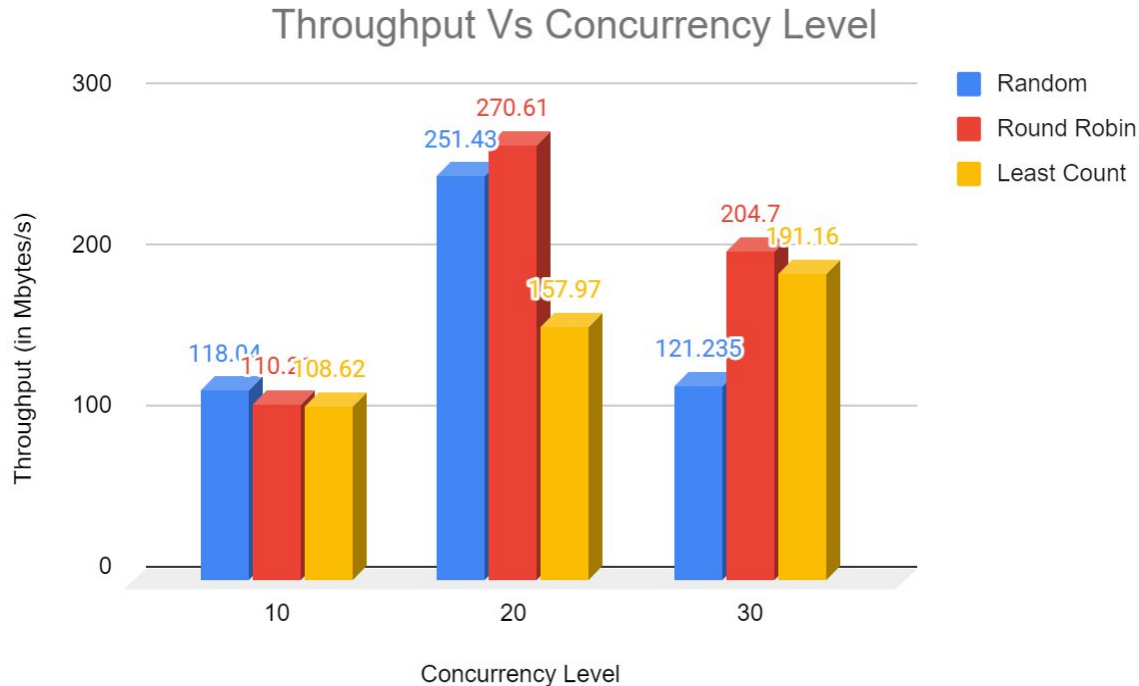

```
"Node: h5"
me/niveditha/pox# ab -n 100 -c 10 -g demo.csv http://10.0.1.1/testfile.pdf >>concurr
me/niveditha/pox#

"Node: h1" "Node: h2"
ct/2021 20;34;43] "GET /testfile.pdf HTTP/ 10.0.0.5 - - [24/Oct/2021 20;34;43] "GET /te
ct/2021 20;34;43] "GET /testfile.pdf HTTP/ 10.0.0.5 - - [24/Oct/2021 20;34;43] "GET /te
ct/2021 20;34;43] "GET /testfile.pdf HTTP/ 10.0.0.5 - - [24/Oct/2021 20;34;43] "GET /te
ct/2021 20;34;43] "GET /testfile.pdf HTTP/ 10.0.0.5 - - [24/Oct/2021 20;34;43] "GET /te
ct/2021 20;34;44] "GET /testfile.pdf HTTP/ 10.0.0.5 - - [24/Oct/2021 20;34;43] "GET /te
ct/2021 20;34;44] "GET /testfile.pdf HTTP/ 10.0.0.5 - - [24/Oct/2021 20;34;43] "GET /te

"Node: h3" "Node: h4"
ct/2021 20;34;43] "GET /testfile.pdf HTTP/ 10.0.0.5 - - [24/Oct/2021 20;34;43] "GET /
ct/2021 20;34;43] "GET /testfile.pdf HTTP/ 10.0.0.5 - - [24/Oct/2021 20;34;43] "GET /
ct/2021 20;34;43] "GET /testfile.pdf HTTP/ 10.0.0.5 - - [24/Oct/2021 20;34;43] "GET /
ct/2021 20;34;43] "GET /testfile.pdf HTTP/ 10.0.0.5 - - [24/Oct/2021 20;34;43] "GET /
ct/2021 20;34;44] "GET /testfile.pdf HTTP/ 10.0.0.5 - - [24/Oct/2021 20;34;44] "GET /
```

Screenshot for
implementation
of Random
Algorithm For
Concurrency
Level 10

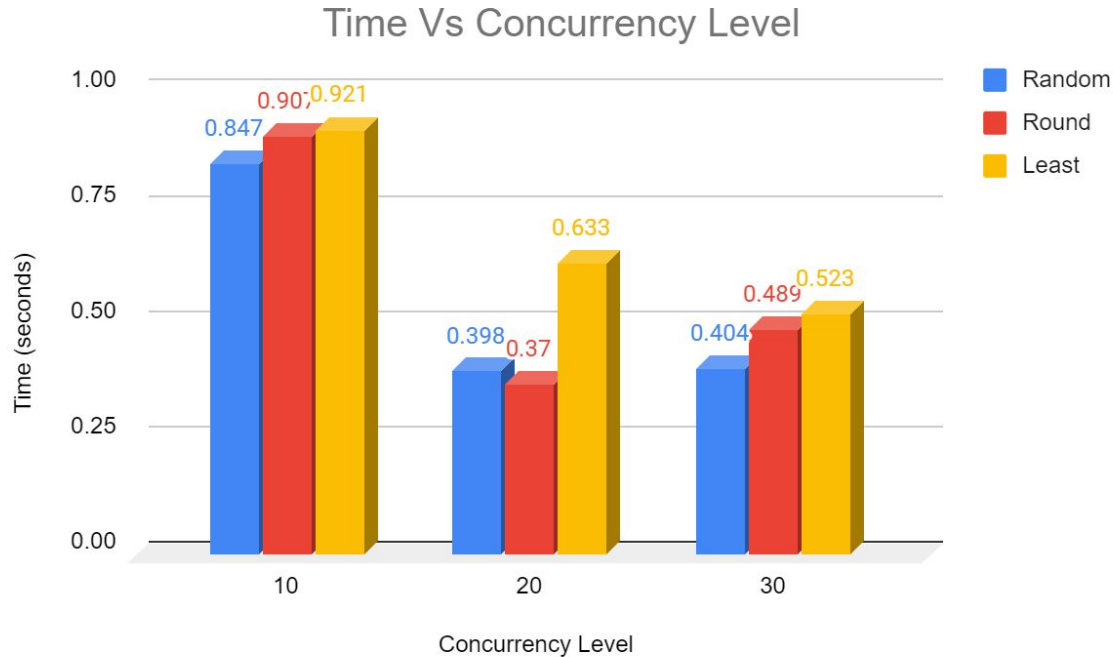
Throughput Vs Concurrency Level



The throughput values from the table has been obtained to draw a cluster bar graph as shown. As can be shown, the Round Robin approach has a higher throughput at increasing concurrency levels, whereas the Least Count strategy has the lowest throughput in all circumstances.

The information for this graph is taken from [CL10](#), [CL20](#) and [CL30](#).

Time Vs Concurrency Level



The total time from the table has been obtained to draw a cluster bar graph as shown. The y-axis represents time, while the x-axis represents the level of concurrency. In all three scenarios, the Least Count method has a greater throughput.

The information for this graph is taken from [CL10](#), [CL20](#) and [CL30](#).

Variation in Throughput Based on Number of Servers


Throughput is calculated for various algorithms by changing the number of servers. Here, 3 variations in the number of servers are taken, that is, 2 servers, 4 servers and 8 servers, all in single topology. Commands which can be used to attain the results are:

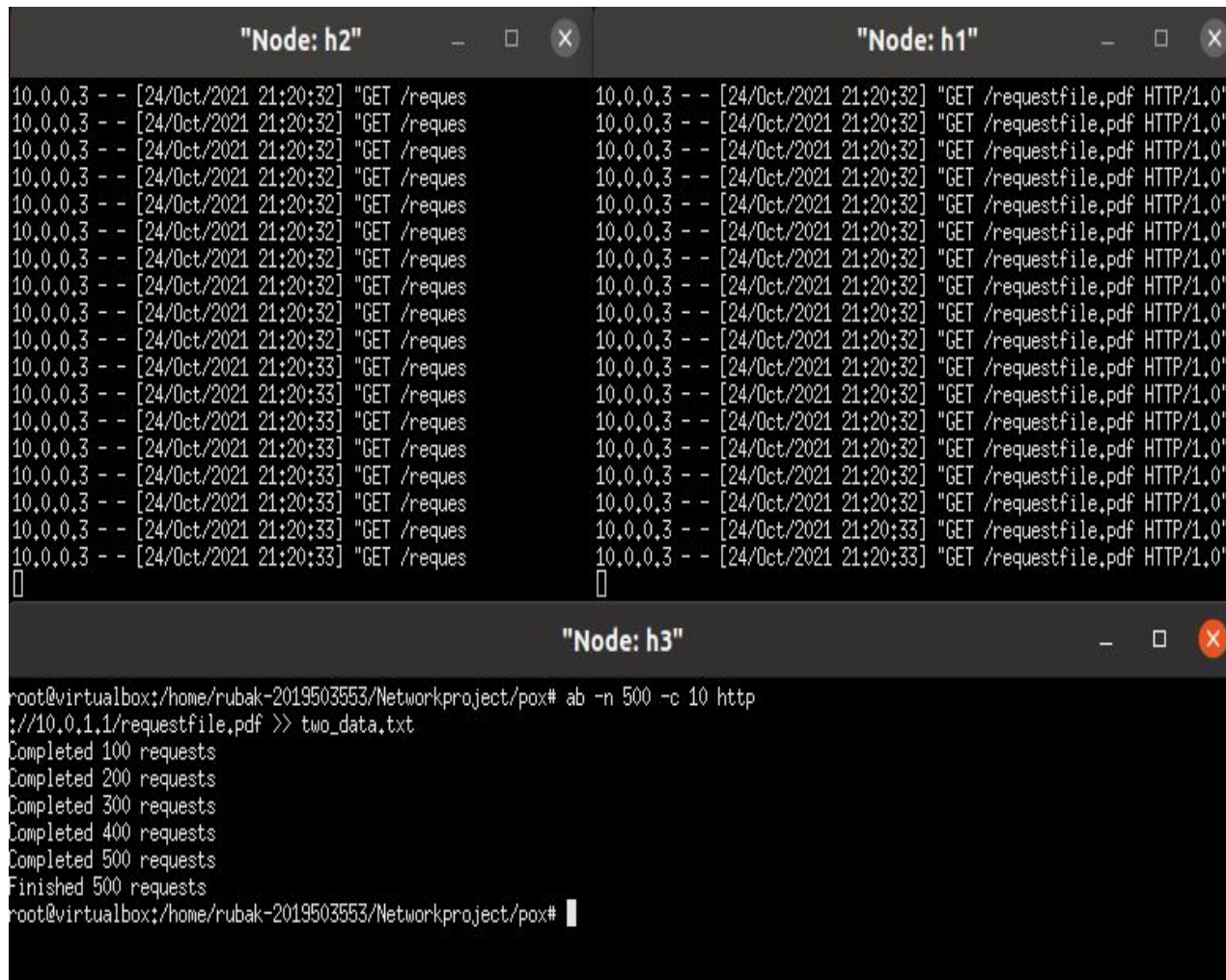
Pox Controller command: **./pox.py log.level --DEBUG rand --ip=10.0.1.1
--servers=10.0.0.1,10.0.0.2**

The above code is for 2 servers which is to be changed to 4 servers and 8 servers later.

Xterm Command to connect with servers: **ab -n 500 -c 10 http://10.0.1.1/requestfile.txt
>>two_server_data.txt**

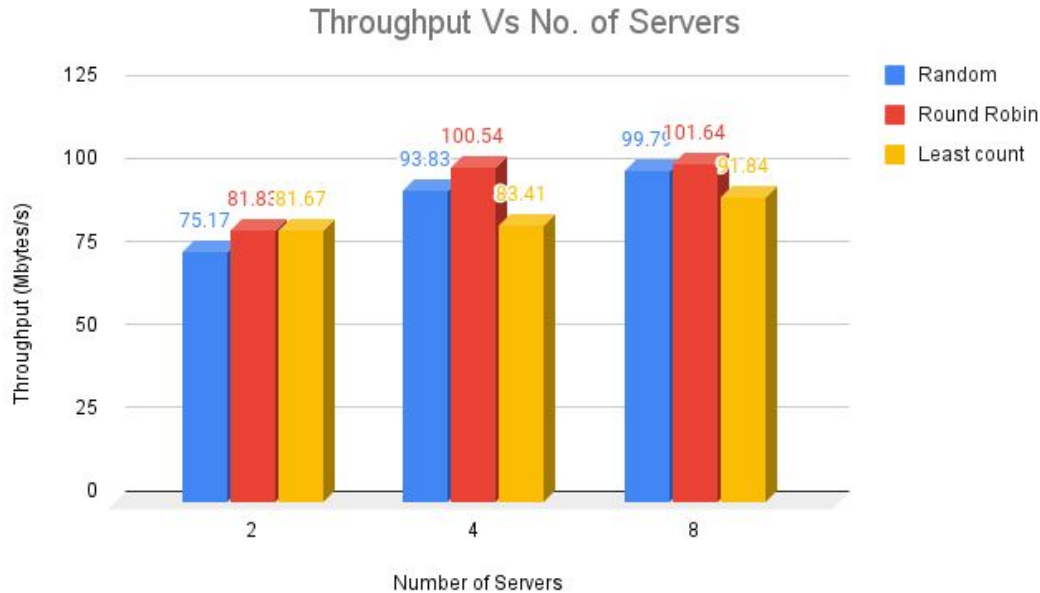
The above commands are used to save the throughput values which are calculated by using various combinations of servers and algorithms into a text file. The information obtained is used to plot graph for throughput.





Screenshot for the implementation of Random algorithm with two servers active. The topology used is Single topology

Throughput Vs Number of Servers

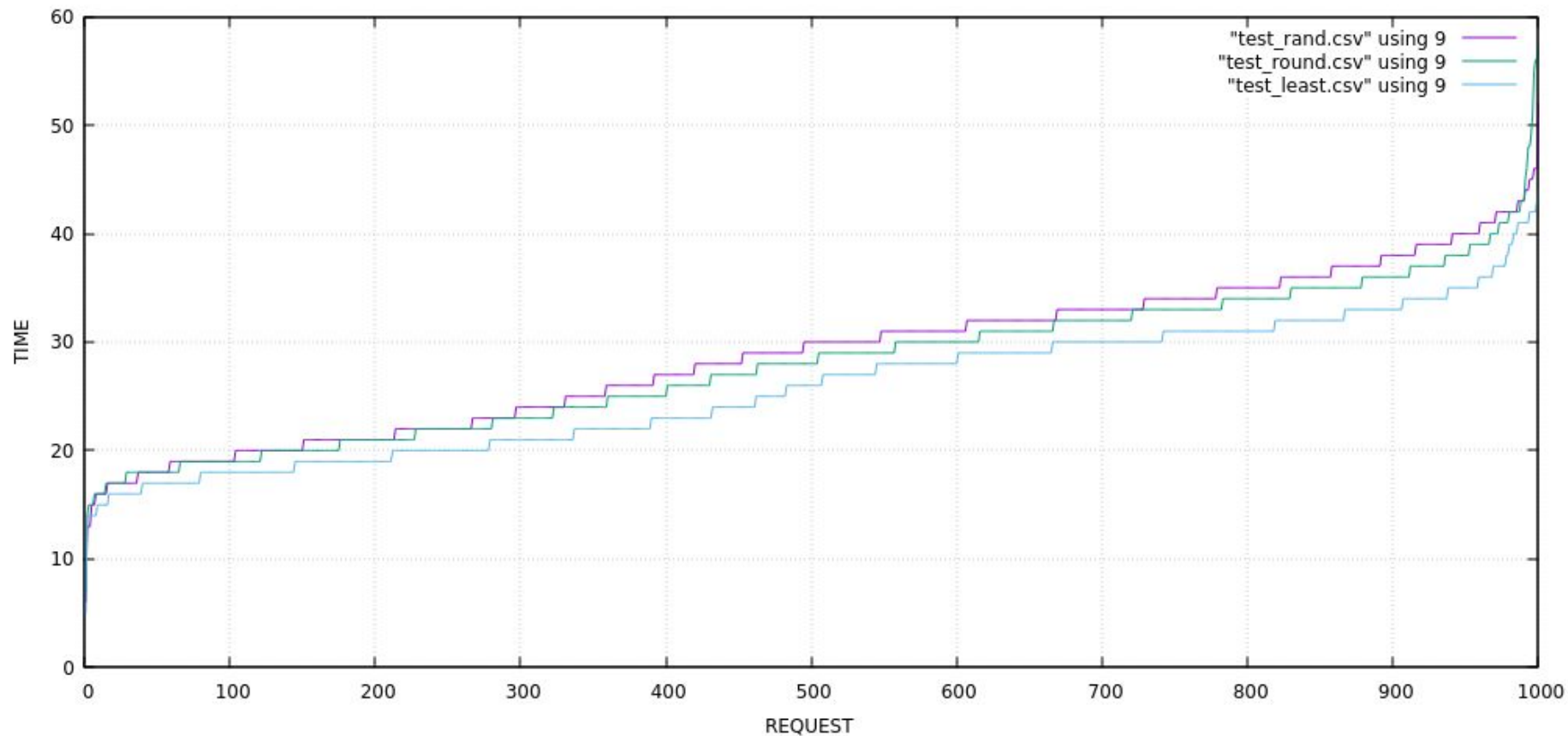


The throughput values from the table has been obtained to draw a cluster bar graph as shown. As we can observe, when the number of servers increases the throughput value also increases gradually. From the three algorithms we can notice that Least count Algorithm has the least value at high number of servers.

Corresponding data used: Two servers , Four servers , Eight servers

CONSOLIDATED OBSERVATIONS

Line graph plotted from csv files
obtained from simulation of
algorithms



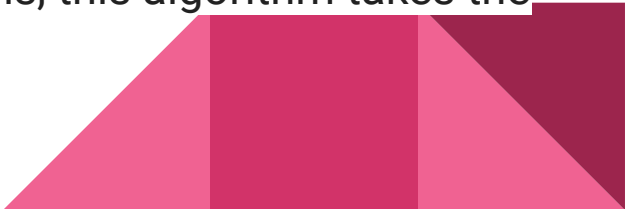
CSV Files can be found here: [test_rand](#), [test_round](#), [test_least](#).

Observations

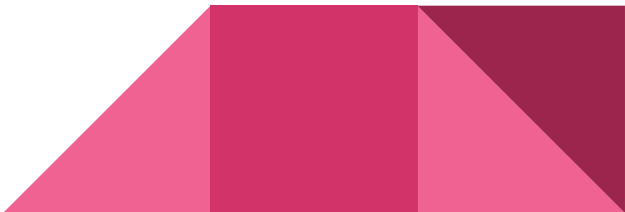
- Round robin algorithm performs better in some cases such as when there are high number of requests.
- Least Count performs good in cases where there are more number of concurrent requests.
- Least Count algorithm also takes the maximum amount of time for any concurrent level.
- Round robin is an efficient algorithm when there is an increase in the number of servers.
- From the Line graph we can see that Least count algorithm performs most efficiently compared to other algorithm in all aspects.



Gaps

- Round robin works better than random only when number of servers are higher.
 - star topology is better than single topology.
 - Most of the solution proposed till now are based on single switch and single controller, but multiple switch and controller work better
 - In case the capacities of the servers are disproportionate, the load balancer running on Random algorithm might still distribute requests without considering it. This can result in over heading of a specific server which might affect the network.
 - From all the above comparisons we can see that performance of random algorithm is poor when compared with other 2 algorithms.
 - In cases where the application servers have similar specifications, an application server may be overloaded due to longer lived connections, this algorithm takes the active connection load into consideration.
- 

Novel Solution

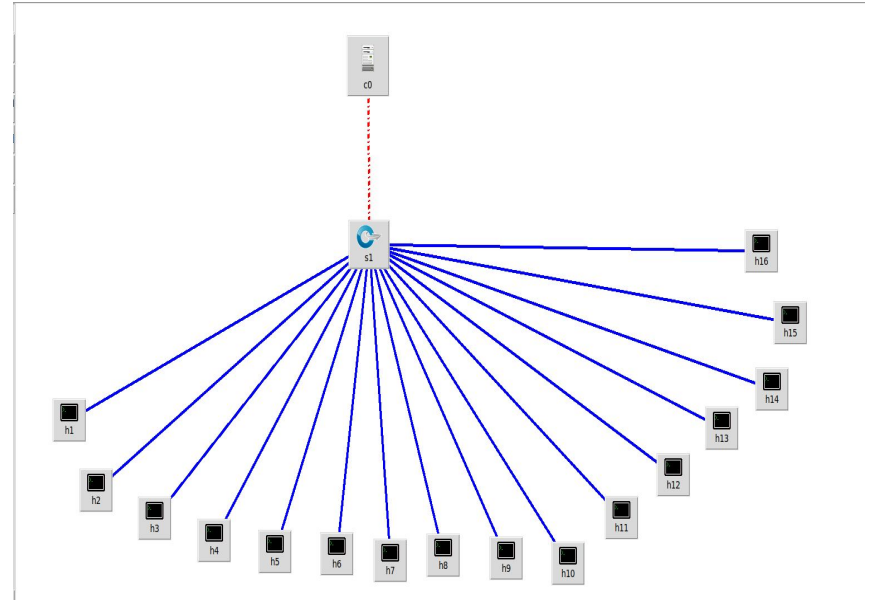
- From the above observations, we can see that each algorithm is more productive only under selective circumstances. Real-life situations might need an algorithm or topology that works efficiently even under unpredictable loads with varying nature of incoming requests as well as sudden influxes.
 - In such cases, typical topologies and load balancing algorithms may fail to prevent overloading and decrease the overall performance of the system.
 - We have developed a custom topology that uses a combination of multiple algorithms to systematically load balance client requests across server pools. We will be presenting a comparison between our solution and one of the existing methods to load balance the server-load.
- 

Execution and Results

Implementation of novel solution and comparison of its results with that of existing solutions.

Single topology - Implementation

- A single topology is when there is just one openflow switch and k hosts. It also sets up a link between the switch and the k hosts.
- The hosts from 1 to 12 serve as servers, while hosts 13, 14, 15, and 16 serve as hosts.
- The servers will fulfil requests from any of these four hosts.
- All of these hosts are linked to a single switch, which is linked to a single controller.



Custom topology - Implementation

The number of hosts in our custom topology is 16, the number of switches is 5, and the number of controllers is four. We developed the topology in such a manner that each controller uses a different algorithm.

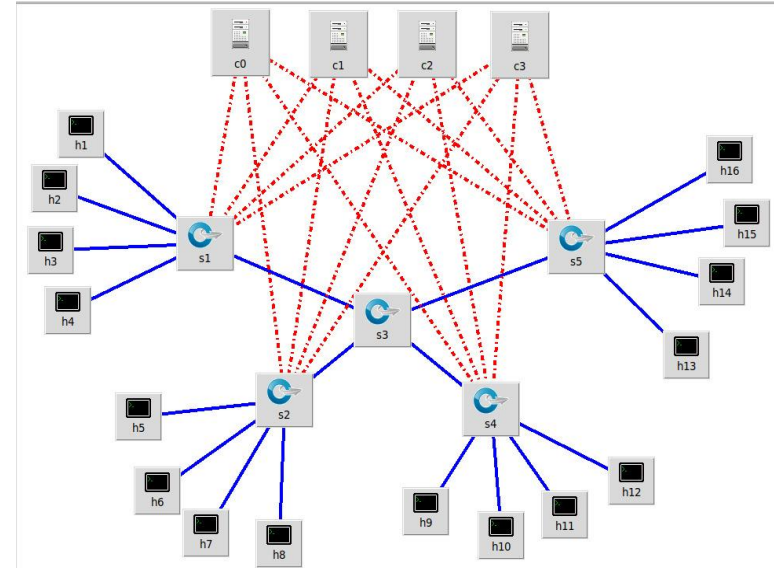
C0 - Random algorithm

C1 - Round robin algorithm

C2 - Least correction algorithm

C3 - Weighted round robin algorithm

As can be seen, the switch c1 has four hosts. The hosts h1,h2,h3 function as servers and respond to the requests made by the host h4. Every switch in the system follows this pattern. The first three hosts will function as servers, while the fourth and final host will serve as hosts.



Implementation

Shell script is used to run four pox controllers in parallel at various port numbers.

```
yogeeswar@MIT: ~/Desktop/sem5/cn/p... x  yogeeswar@MIT: ~/Desktop/sem5/cn/... x  yogeeswar@MIT: ~/Desktop/sem5/cn/w... x  ▼
1 #!/bin/sh
2 ./pox.py log.level --DEBUG rand --ip=10.1.0.0 --servers=10.1.1.1,10.1.1.2,10.1.1.3 &
3 ./pox.py log.level --DEBUG round --ip=10.1.0.0 --servers=10.1.2.1,10.1.2.2,10.1.2.3 openflow.of_01 --port=6634 &
4 ./pox.py log.level --DEBUG least --ip=10.1.0.0 --servers=10.1.3.1,10.1.3.2,10.1.3.3 openflow.of_01 --port=6635 &
5 ./pox.py log.level --DEBUG wrr --ip=10.1.0.0 --servers=10.1.4.1:1,10.1.4.2:2,10.1.4.3:3 openflow.of_01 --port=6636 &

1,1 All
```

To create the topology, miniedit is used and python script is exported, this python script is modified to run various test.

```
yogeeswar@MIT: ~/Desktop/sem5/cn/p... x  yogeeswar@MIT: ~/Desktop/sem5/cn/... x  yogeeswar@MIT: ~/Desktop/sem5/cn/... x
131     time.sleep(5)
132     info('*** Servers set\n##### Running tests from h4 h8 h12 ####\n\n')
133
134     info('##### FILE SIZE TEST (file size 100mb) #####\n\n')
135     h4.cmd('ab -n 100 -c 10 -g test_rand11.csv http://10.1.0.0/test_100kb.txt >> test_rand.txt &')
136     h8.cmd('ab -n 100 -c 10 -g test_round11.csv http://10.1.0.0/test_100kb.txt >> test_round.txt &')
137     h12.cmd('ab -n 100 -c 10 -g test_least11.csv http://10.1.0.0/test_100kb.txt >> test_least.txt &')
138     h16.cmd('ab -n 100 -c 10 -g test_wrr11.csv http://10.1.0.0/test_100kb.txt >> test_wrr.txt &')
139     time.sleep(30)
140
141     info('##### FILE SIZE TEST (file size 500kb) #####\n\n')
142     h4.cmd('ab -n 100 -c 10 -g test_rand12.csv http://10.1.0.0/test_500kb.txt >> test_rand.txt &')
133.0
```

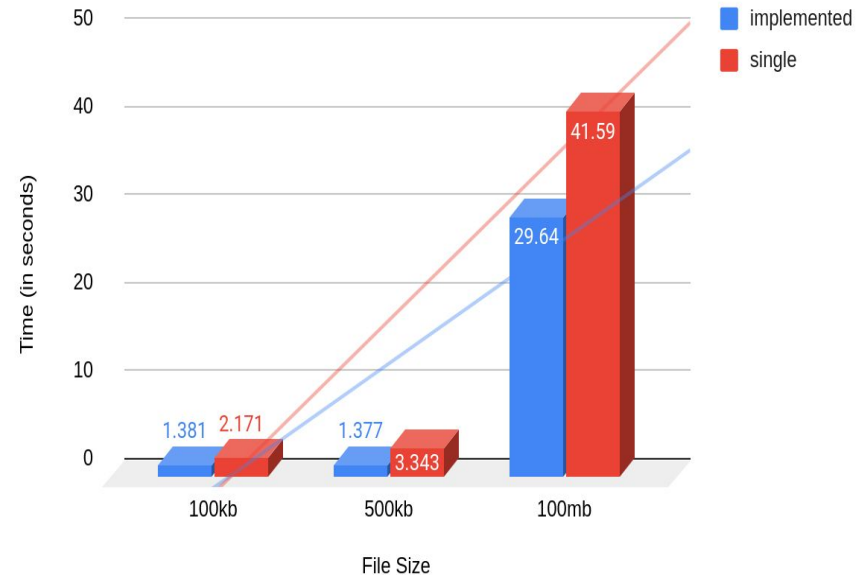
File size vs Total Time

The total time taken (in seconds) has been obtained to plot a cluster bar graph as shown. This graph represents the variation in the size of the file sent to the given topologies. The three variations are 100Kb, 500Kb and 100Mb Files.

As the file size increases we can see that the time taken to send the request also increases exponentially for the given single topology but the implemented topology takes comparatively lesser time. This shows that the implemented topology is more efficient.

The data for the given graph is provided [here](#).

File size vs Time

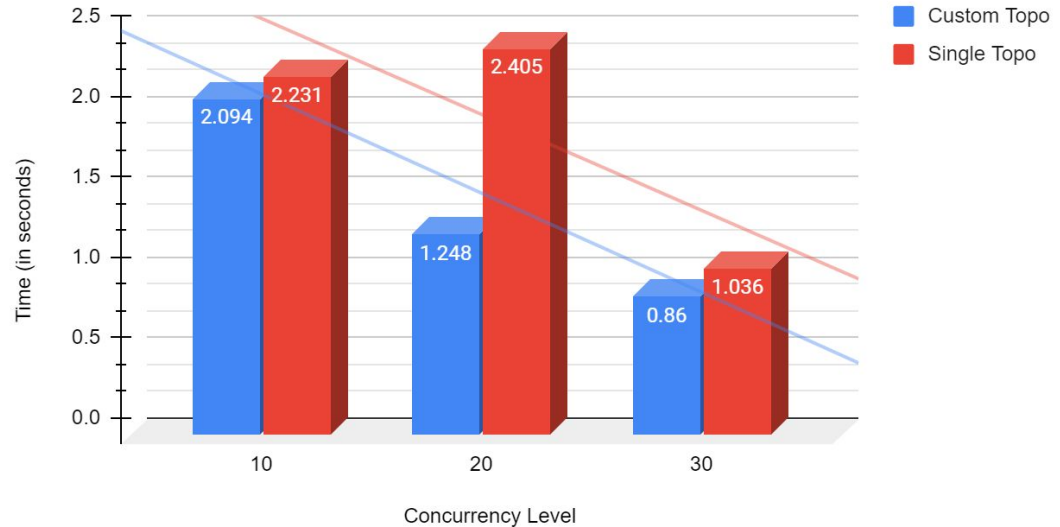


Time Vs Concurrency Level

The total time for 100 request taken (in seconds) has been obtained to plot a cluster bar graph as shown. Time is taken in y-axis and concurrency level is taken in x-axis. Three concurrency levels are taken for comparison - 10, 20 and 30

From the graph we can see that implemented solution performs better even though concurrency increases.

Concurrency Vs Time



The data for this graph is provided from [here](#).

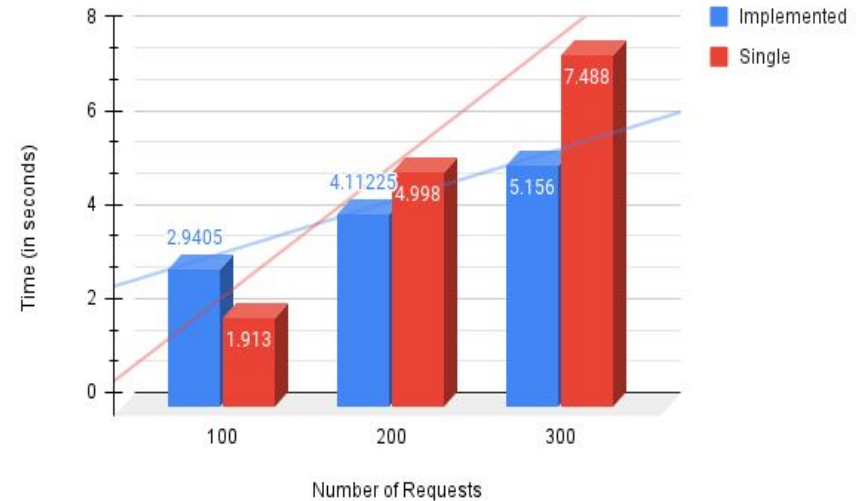
Number of Requests vs Time

The total time taken (in seconds) has been obtained to plot a cluster bar graph as shown. This graph represents the variation in the number of requests sent to the given topologies. The variations are 100, 200 and 300 requests.


From the graph we can see that as the number of requests increases the implemented topology is more efficient than the already existing Single topology.

The data for the given graph is provided [here](#).

No. of Requests Vs Time



Observations


- The topological comparisons have been made using three criteria, namely concurrency level, file size, and the number of requests from the host.
 - The graph interpretation of the criterion clearly reveals that the custom topology performs better than the single topology in terms of load balancing.
 - This is definitely better than single topology where only one switch and controller is used and can only run one particular algorithm which may not be efficient under all situations.
 - Here we are running some of the most commonly used algorithms in order to handle all types of client needs. For example round robin performs well when number of connections are high but least count is the best among others to handle huge file request.
- 

Conclusion

- The custom topology that we have created is effectual in terms of several parameters such as time, file sizes, concurrency levels, etc.
- A system should not get overwhelmed and all requests must be distributed suitably. Hence, load balancing is essential to manage the traffic follow in servers easily and efficiently.
- There are multiple ways to execute load balancing efficiently in SDN. Though the default topologies such as single, linear and tree perform well, custom topologies created in accordance with the need of the network makes the overall system productive and more cost-effective.



References

- <https://www.brianlinkletter.com/2015/04/using-the-pox-sdn-controller/>
 - https://assets.researchsquare.com/files/rs-53407/v1_covered.pdf?c=1631838773
 - https://github.com/mcanalesmayo/sdn_load_balancer
 - <https://github.com/sarthakpranesh/Mininet-Load-Balancing>
 - <https://httpd.apache.org/docs/2.4/programs/ab.html>
 - <https://noxrepo.github.io/pox-doc/html/>
 - <https://dj184dja8.blogspot.com/2016/04/mininet-fattree-topology.html>
 - <https://github.com/anand1996wani/Load-Balancing-Using-Software-Defined-Networks>
 - <https://ieeexplore.ieee.org/document/8681512>
 - https://www.researchgate.net/publication/312538681_Adaptive_Load_Balancing_Scheme_For_Data_Center_Networks_Using_Software_Defined_Network/citation/download
 - https://github.com/VamsikrishnaNallabothu/My_POX_SDN_Work
 - <https://github.com/noxrepo/pox/tree>
- 



Thank you