

SDN: Link Level Load Balancer

CSE 534 ADVANCED COMPUTER NETWORKS

Shankar Krishnamurthy, Sidharth Singh

Contents

1	Introduction	1
2	Requirements	1
3	Load Balancing Strategy	2
4	Test Topology	3
4.1	Configuration and Routing	3
5	Traffic Generation	4
6	Communicating Network Statistics to the Controller	4
6.1	Gathering Network Statistics on Traffic Shaper Nodes.....	4
6.2	Querying Network Statistics by the Controller	5
7	Control Loop Parameters	5
8	Testing	6
9	Limitations	6
10	References.....	7
11	Appendix	7

1 Introduction

The internet traffic rates for some of the popular services have exploded through the roof. Popular search engines like Google and Bing or Social Networking Applications like Twitter receive millions over millions of requests and it is practically impossible for a single server to serve all the requests and not crash down. In addition to huge loads, requests come from all across the globe and having a single server would lead to great imbalances in the latency of serving requests from different geographic locations.

Typically, such high-demand services are replicated over a bunch of servers in the server farm. A Load Balancer is usually a device that sits close to the server farm and balances incoming requests across the replicated servers. There can be several mechanisms to distribute the load across the servers ranging from trivial ones like Round Robin to more sophisticated ones like Least Connection, Least Response Time or Token Based.

While existing load balancing solutions offered by companies like Cisco, Citrix and F5 Networks scale pretty well and offer a host of other layer 2 to layer 7 functionalities, they can be prohibitively expensive for smaller ventures. Software-defined Networking (SDN) is a networking paradigm that separates control plane of a network from the data plane by using a centralised controller. A SDN controller has a view of the network wide state and it obviates the need for an expensive piece of hardware that does load balancing. In this project, we demonstrate a controller using POX that performs link level load balancing between two network paths on the basis of available bandwidth and queue size.

2 Requirements

1. **Mininet** – An emulation tool that use virtualization to create a virtual topology of hosts, openflow switches and SDN controller.
2. **POX Controller** – A framework in python that provides openflow libraries to implement a SDN controller.
3. **Wireshark** - A packet analyser used for debugging purposes.

3 Load Balancing Strategy

The Load Balancing strategy is implemented using POX controller. We have two network paths to reach the aggregator from source – via Traffic Shaper 1 and Traffic Shaper 2. We consider the following parameters in our Load Balancing Strategy:

- i) A comparison of the link utilization of the two network paths available. We consider the link between ts1-eth1 <-> aggregator-eth0 and ts2-eth1 <-> aggregator-eth1 (Figure 1). We define link utilization as the percent of allocated bandwidth utilized over those links.
- ii) A comparison of the queue depths of the outgoing (towards aggregator) interfaces of Traffic Shaper 1 and Traffic Shaper 2.

The commands used to collect these statistics are provided in section 6.1

We perform our tests using the open-source tool iperf [] which pumps TCP traffic. The benefit of using iperf is that it provides useful statistics while the traffic is running and that it can listen on multiple interfaces on the same server. The nature of TCP traffic is such that it takes a while to ramp up because of TCP slow start [] mechanism.

To account for the slow build-up of traffic, we have also implemented a simple round robin load balancing algorithm which is used until a threshold is reached. We set our thresholds to 50% of the link bandwidth for both network paths. For queues, we set the limit to 50% of the maximum queue size. Our Load Balancing algorithm works as follows –

- i) Use Round Robin Load Balancer as long as bandwidth utilization and queue sizes are below 50% for both the network paths.
- ii) If bandwidth utilization crosses the threshold on any of the network paths, we abort Round Robin Load Balancer in favour of –
 - a. Pick the network path which has greater available bandwidth.
 - b. If both network paths have equal available bandwidth, we consider queue sizes and pick the network with a lower queue size.
- iii) If queue size crosses the threshold on any of the network paths, we abort Round Robin Load Balancer in favour of –
 - a. Pick the network path which has lower queue size.
 - b. If both network paths have equal queue sizes, we consider available bandwidth and pick a network with higher available bandwidth.

For purposeful debugging we have incorporated several debug counters in our Load Balancer Controller that gives us a good idea of how load balancing progresses. These counters can be seen when the Controller is run with log level set to INFO. Table 1 summarises the counters and their meaning.

Counter	Purpose
ts1_path_counter	Total Number of Connections Sent via Traffic Shaper 1
ts2_path_counter	Total Number of Connections Sent via Traffic Shaper 2
ts1_bw_threshold_more_available_bw	Sent via Traffic Shaper 1 because more BW available
ts2_bw_threshold_more_available_bw	Sent via Traffic Shaper 2 because more BW available
ts1_q_threshold_lower_q	Sent via Traffic Shaper 1 due to lower queue size
ts2_q_threshold_lower_q	Sent via Traffic Shaper 2 due to lower queue size
ts1_roundrobin	Sent via Traffic Shaper 1 due to Round Robin
ts2_roundrobin	Sent via Traffic Shaper 2 due to Round Robin

Table 1: List of counters for debugging

4 Test Topology

The topology used in this experiment is shown in the Figure 1. This topology was created using a custom topology script on Mininet. The topology includes four hosts - one source/client, two traffic shaper nodes running Linux utility 'tc' to modify network characteristics like bandwidth, latency and queue depth and aggregator node that acts as a multi-homed server (see section 3.1 for more details on configuration and routing). The topology also contains an openflow virtual switch and another host connected to it that acts as the SDN controller.

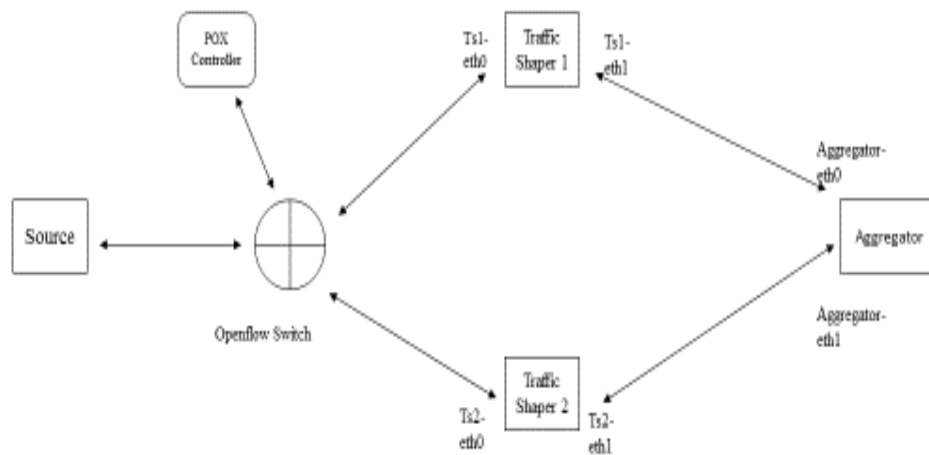


Figure 1: Load Balancer Test Topology

4.1 Configuration and Routing

While setting up configuration and routing for this experiment, four points had to be considered –

- The aggregator/server must be implemented as a multi-homed device so that it can be assigned two IP Addresses in two different networks. We use this configuration to emulate two servers and two distinct Network via Traffic Shaper 1 and Traffic Shaper 2.

We assign IP addresses in two different networks on the aggregator. Aggregator-eth0 is assigned 10.0.0.1 and Aggregator-eth1 is assigned 20.0.0.1. To ensure correct routing we create two routing tables – table 1 and table 2 and bind them to the two interfaces respectively. Table 1 has Traffic Shaper 1 (ts1-eth1: IP 10.0.0.10) as its default gateway while Table 2 has Traffic Shaper 2 (ts2-eth2: IP 20.0.0.10) as its default gateway.

Routing Table 1 and Table 2 on the Aggregator look like this:

```
root@mininet-vm:~/mininet/custom# ip rule show
0:    from all lookup local
32764: from 30.0.0.20 lookup 2
32765: from 20.0.0.20 lookup 1
32766: from all lookup main
32767: from all lookup default

root@mininet-vm:~/mininet/custom# ip route show table 1
default via 20.0.0.10 dev aggregator-eth0

root@mininet-vm:~/mininet/custom# ip route show table 2
default via 30.0.0.10 dev aggregator-eth1
```

- b) The controller must be able to communicate with the two traffic shaper nodes in order to collect bandwidth and queue statistics for load balancing. For this we assign an IP Address on the openflow switch's bridge interface (s1: IP 10.0.0.10). The openflow switch can now bridge traffic from the controller to the Traffic Shapers because one interface of the shapers (ts1-eth0: IP 10.0.0.3 and ts2-eth0: IP 10.0.0.4) are also connected to the openflow switch.
- c) The source/client must be able to directly communicate with the two IP addresses on the Aggregator via appropriate routes on source and aggregator.

Add the following routes on the source/client to reach the aggregator via Traffic Shaper 1 and Traffic Shaper 2 respectively.

```
route add -net 20.0.0.0 netmask 255.255.255.0 gw 10.0.0.3
route add -net 30.0.0.0 netmask 255.255.255.0 gw 10.0.0.4
```

Also, enable IP Forwarding on Traffic Shaper 1 and Traffic Shaper 2.

```
sysctl -w net.ipv4.conf.all.forwarding=1
```

5. Traffic Generation

Iperf [] is used to pump traffic from source to aggregator in this experiment. We have created a script in python that spawns multiple threads and starts multiple TCP connections to the aggregator. An Iperf server running on the default 5001 port is running on the aggregator and it listens for incoming TCP connections on both interfaces of the aggregator. Traffic from the source is sent to the Switch where a Load Balancing decision is made by the controller for every flow as per the mechanism described in section 3.

6. Communicating Network Statistics to the Controller

6.1. Gathering Network Statistics on Traffic Shaper Nodes

To collect network statistics (bandwidth utilization of the link and queue length of forwarding interface), we run a python script, netinfo.py [geni] [NetInfo] on both Traffic Shapers. This script was referenced from GENI tutorials of SDN Load Balancer. The python script collects link utilization statistics from the file /proc/net/dev. For queue length, it uses the 'tc -p qdisc show' command. The script parses the output for these commands and appends them to an output file. This output file is hosted on the open source NetInfo server which we have installed on both Traffic Shaper Nodes.

6.2 Querying Network Statistics by the Controller

We have added the python code in our POX controller to query network statistics from Traffic Shapers. We use the python library 'urllib2' to send a HTTP GET for the most recent file containing statistics. A detailed description of how we query for network statistics is as follows:

- i) We have defined two python functions - TS1_info () and TS2_info (). These functions use 'urllib2' to send a HTTP GET to the two Traffic Shaper Nodes respectively.
- ii) In TS1_info () and TS2_info (), we parse the output of the HTTP GET to extract bandwidth and queue size. These statistics are assigned to variables (total_ts1_bytes, total_ts2_bytes, total_ts1_queue, total_ts2_queue). A new HTTP GET request is sent after an interval of every 5 seconds.
- iii) While assigning these network statistics to variables, we acquire locks to ensure that the Load Balancer doesn't read from these variables while they are being written to. The locks are released after we have assigned new network statistics to the variables.
- iv) TS1_info() and TS2_info() are called (only once) in the _handle_PacketIn function and they run in an infinite loop so long as the controller runs.
- v) To ensure that general packet processing on the controller does not block for communication with the traffic shaper nodes, we spawn two threads and call TS1_info() and TS2_info() within those threads.

7 Control Loop Parameters

The controller queries bandwidth and queue statistics from the traffic shaper nodes (TS1 and TS2) and applies exponential averaging while using these values in the Load Balancing decision making. For the most recent bandwidth/queue size value fetched by the HTTP GET, we assign an exponential weighted average of 0.2. We chose a low value of 0.2 to avoid instability/inaccurateness in the values of Bandwidth utilization. The nature of TCP is such that the rate at which a source can send packet often fluctuates with time.

There are two primary causes for this –

- i) TCP Slow Start – the sender begins very cautiously but doubles the rate of sending (congestion window) on receiving acknowledgements. Some of the newer TCP Congestion Algorithms like CUBIC are extremely aggressive while ramping up.
- ii) TCP Reaction to Packet Loses – most common TCP Congestion Algorithms (like Reno, Cubic) slash the congestion window by half or as much as 30-40% (depending upon the exact algorithm) as soon as a packet loss is detected via Retransmission Timeout (RTO).

To account for these sudden fluctuations in throughput due to TCP's behaviour, we assign a low weighted average of 0.2 for the most recent bandwidth reading. Over time, our values get averaged out and the load balancer works efficiently.

8. Testing

8.1. List of Test Cases

For each Test Case, we use a python script to that spawns 5 threads and opens 50 TCP connections in each thread. TCP connections are opened using iperf as described in section 5. Each TCP connection runs for a random time period between 5sec and 25sec.

Test Case ID	#Requests	Traffic Shaper 1	Traffic Shaper 2
1	250	Bandwidth : 20Mbits/sec	Bandwidth : 20Mbits/sec
2	250	Bandwidth : 100Mbits/sec	Bandwidth : 20Mbits/sec
3	250	Bandwidth : 25Mbits/sec	Bandwidth : 20Mbits/sec
4	250	Bandwidth : 20Mbits/sec	Bandwidth : 30Mbits/sec
5	250	Bandwidth : 100Mbits/sec	Bandwidth : 125Mbits/sec
6	250	Bandwidth : 125Mbits/sec	Bandwidth : 130Mbits/sec

7	250	Bandwidth : 135Mbits/sec	Bandwidth : 140Mbits/sec
---	-----	--------------------------	--------------------------

Table 2: Test Case

8.2. Test Results

Figure 2 shows graphs for the test cases in section 8.1. These graphs show the distribution of traffic across the two network paths (via Traffic Shaper 1 and Traffic Shaper 2). There are 4 columns in each bar graph that show how many requests are load balanced (round robin) across the two paths and how many requests are load balanced based on link level statistics queried by the controller.

The Y Axis of these graphs show the number of TCP Connections. The X Axis shows the nature of Load Balancing that the TCP Connections underwent. For example: Consider the middle graph in the bottom row. The bandwidth is set to 125mbits on the network path via Traffic Shaper 1 and 130mbits on the network path via Traffic Shaper 2. The graph shows that 5 connections were sent on either network path due to Round Robin Load Balancing strategy at the start until the thresholds are reached. Furthermore, network path via Traffic Shaper 1 was chosen for 112 connections because more bandwidth was available on this path. Similarly, network path via Traffic Shaper 2 was chosen for 125 connections because more bandwidth was available on the second path. Another 12 connections chose the network path via Traffic Shaper 1 because of lower queue size.

The rest of the graphs in Figure 2 can be interpreted in a similar manner.

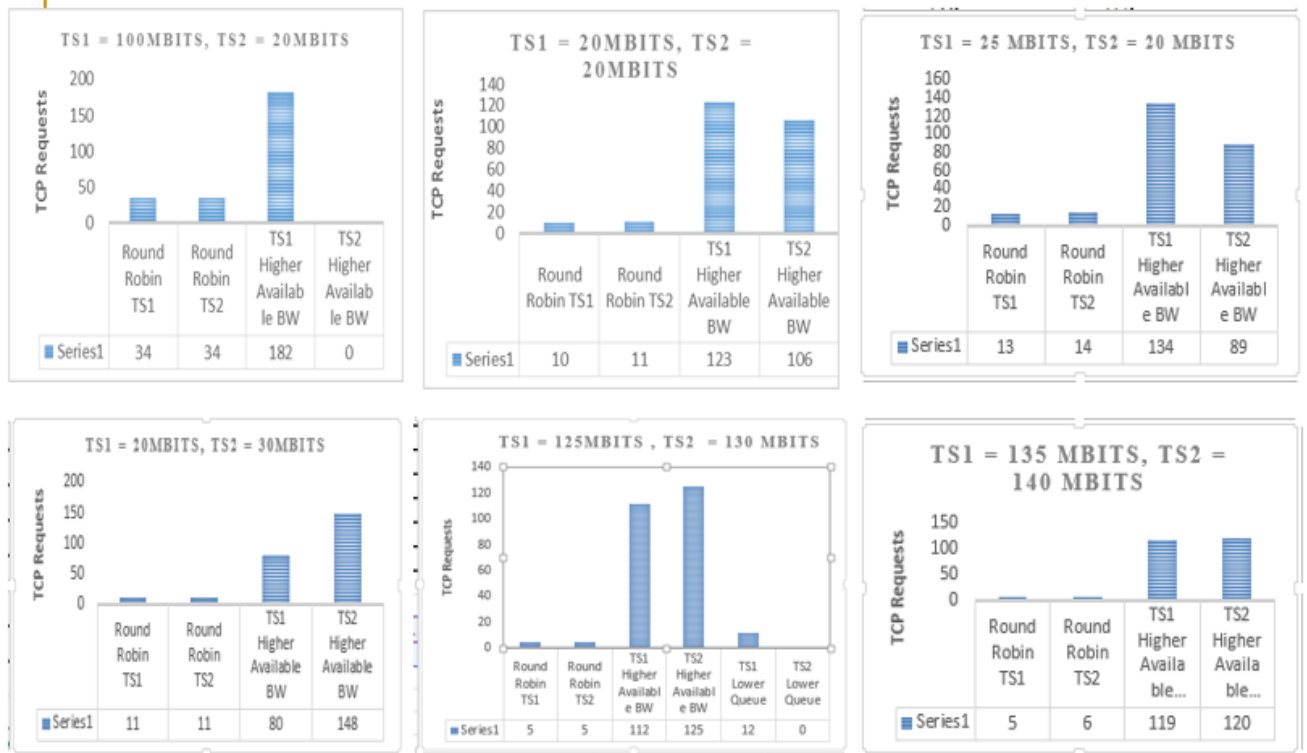


Figure 2: Test Result Graphs

In order to refer to raw data represented in graphs can be seen in raw data in the Appendix section.

9. Limitations

The following challenges were encountered during the implementation of this project:

- 1) We used 'iperf' for traffic generation. With 'iperf' we observed that the client/source usually sent a TCP RESET on the connection after a span of roughly 30 seconds. As a result of this limitation, we

could not establish long lived TCP flows. To counter this, we created a custom script that spawned multiple threads in order to utilize the network bandwidth.

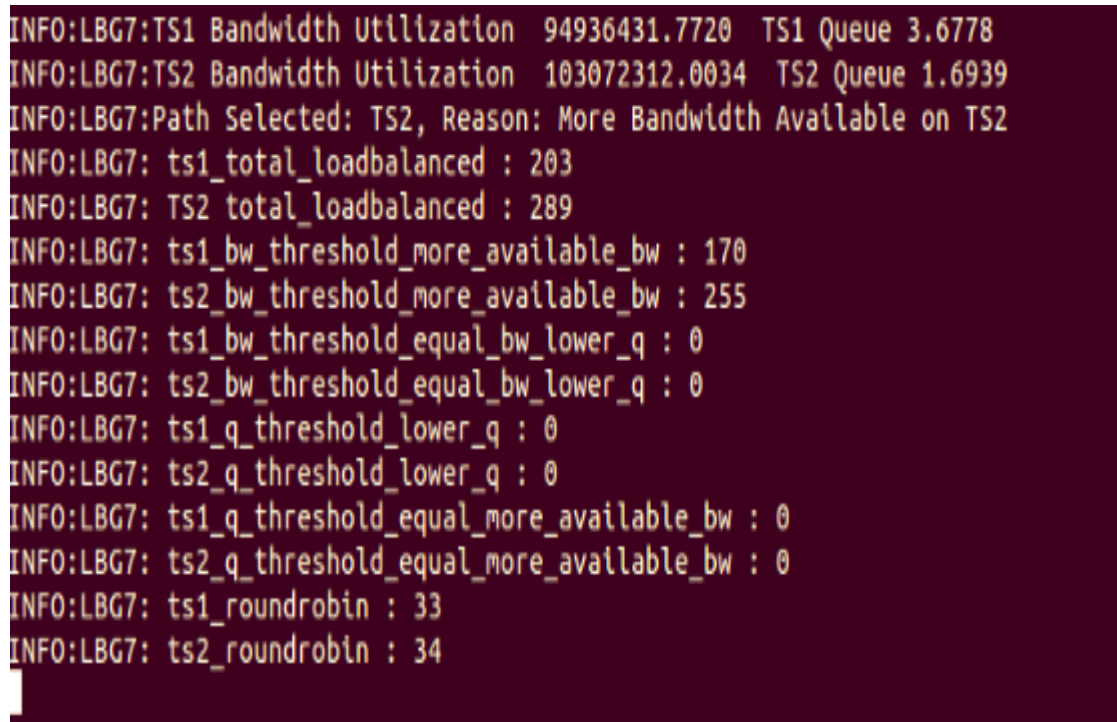
- 2) In our experiments, we found that the queue size did not increase much despite trying multiple values for the 'tb' parameters like buffer and limit. Moreover, whenever queue sizes built up on the Traffic Shaper, so did the link bandwidth utilization. As per our Load Balancer implementation, we check bandwidth utilization before checking queue size.

10. References

- [1] GENI [Online] <http://www.geni.net/>
- [2] Mininet [Online] <http://mininet.org/>
- [3] GENI Resource [Online] <http://www.gpolab.bbn.com/experiment-support/OpenFlowExampleExperiment/netinfo.py>
- [4] GENI Resource [Online] <http://groups.geni.net/geni/wiki/GENIEducation/SampleAssignments/OpenFlowLoadBalancerAssignment/ExerciseLayout/Execute>
- [5] GENI Openflow [Online] <http://groups.geni.net/geni/wiki/OpenFlow>
- [6] Iperf [Online] <https://iperf.fr/>

11. Appendix

- i) Available Bandwidth is defined as total bandwidth of the link minus current weighted utilization of the link.
- ii) Screenshot of POX Controller is shown in Figure 3. This screenshot shows a sample output when network path was chosen because bandwidth threshold was hit. At this point, TS2 had a higher Available Bandwidth and was, therefore, chosen by the Load Balancer. In the POX controller output, under informational messages, various counters are updated whenever a path is chosen.



```
INFO:LBG7:TS1 Bandwidth Utilization 94936431.7720 TS1 Queue 3.6778
INFO:LBG7:TS2 Bandwidth Utilization 103072312.0034 TS2 Queue 1.6939
INFO:LBG7:Path Selected: TS2, Reason: More Bandwidth Available on TS2
INFO:LBG7: ts1_total_loadbalanced : 203
INFO:LBG7: TS2 total_loadbalanced : 289
INFO:LBG7: ts1_bw_threshold_more_available_bw : 170
INFO:LBG7: ts2_bw_threshold_more_available_bw : 255
INFO:LBG7: ts1_bw_threshold_equal_bw_lower_q : 0
INFO:LBG7: ts2_bw_threshold_equal_bw_lower_q : 0
INFO:LBG7: ts1_q_threshold_lower_q : 0
INFO:LBG7: ts2_q_threshold_lower_q : 0
INFO:LBG7: ts1_q_threshold_equal_more_available_bw : 0
INFO:LBG7: ts2_q_threshold_equal_more_available_bw : 0
INFO:LBG7: ts1_roundrobin : 33
INFO:LBG7: ts2_roundrobin : 34
```

Figure 3: POX Controller Sample Output

iii) Tabular Representation of Test Cases Run in Figure 2

#TCP Req	TS1 BW (mbits/s)	TS2 BW (mbits/s)	Total TS1	Total TS2	Round Robin TS1	Round Robin TS2	TS1 Available BW	TS2 Available BW	TS1 Queue	TS2 Queue
250	20	20	133	117	10	11	123	106	0	0
250	100	20	216	34	34	34	182	0	0	0
250	25	20	147	103	13	14	134	89	0	0
250	20	30	91	159	11	11	80	148	0	0
250	100	125	118	132	77	77	41	55	0	0
250	125	130	129	130	5	5	112	125	12	0
250	135	140	124	126	5	6	119	120	0	0

iv) In our Pox Controller, we are calling SimpleL2Learning Class which is implemented in GENI tutorial for trivial L2 switching.