Increasing Revenue by Applying Machine Learning to Congestion Management in SDN

Nabarun Jana
nxj2778@rit.edu

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Increasing Revenue by Applying Machine Learning to Congestion Management in SDN

Submitted by:
Nabarun Jana
nxj2778@rit.edu
(585)286-7223

Under Guidance of:
Prof Joseph Nygate, PhD

Thesis submitted in partial fulfillment of requirements of Degree of Master of Science in Telecommunication Engineering Technology
Department of Electrical, Computer & Telecom Engineering Technology
College of Engineering Technology
Rochester Institute of Technology
Rochester, NY 14623
December 17, 2018
Committee Approval

Joseph Nygate, Ph.D.  
Associate Professor, Thesis advisor

William P. Johnson, J.D.  
Graduate Program Director, MS Telecommunications Engineering Technology Program

Mark Indelicato  
Associate Professor
# Table of Contents

1. Abstract.......................................................................................................................... 1

2. Literature Review ........................................................................................................... 2

3. Architecture .................................................................................................................... 3
   3.1. Mininet ....................................................................................................................... 4
   3.2. POX SDN Controller ................................................................................................ 4
   3.3. IPerf ........................................................................................................................... 5
   3.4. Network Topology .................................................................................................... 6

4. Methodology ................................................................................................................... 8
   4.1. Creating a topology ................................................................................................... 8
   4.2. Generating flows ..................................................................................................... 9
   4.3. Collecting data ......................................................................................................... 10
   4.4. Parsing the results ................................................................................................. 11
   4.5. Dumping to the database ....................................................................................... 11
   4.6. Generating graphs ................................................................................................. 12

5. Experiments .................................................................................................................. 13
   5.1. Intervals .................................................................................................................. 13
   5.2. Duration ................................................................................................................... 13
   5.3. Bandwidth ............................................................................................................... 14

6. Results .......................................................................................................................... 15
Figures

Figure 1: Architecture diagram ................................................................. 3
Figure 2: Mininet ................................................................. 4
Figure 3: POX SDN Controller ................................................................. 5
Figure 4: Iperf ................................................................. 6
Figure 5: Virtual network topology ................................................................. 6
Figure 6: Code to create topology ................................................................. 9
Figure 7: Joins in database for creating graphs ........................................... 12
Figure 8: Graph showing value of coefficients over time ................................................................. 19
Figure 9: Graph showing value of coefficients over long range ...................................................... 20
Figure 10: Graph showing value of coefficients for duration: 8sec bandwidth: 64 Mb/sec .... 21
Figure 11: Graph showing value of coefficients for duration: 16sec bandwidth: 32 Mb/sec ....... 22
Figure 12: Graph showing value of coefficients for duration: 16sec bandwidth: 64 Mb/sec .... 23

Tables

Table 1: h1001-h2040.iperf.dat ........................................................................................................ 10
Table 2: h1001-h2040.ping.txt ........................................................................................................ 10
Table 3: Duration of flows with time ................................................................................................. 14
Table 4: SLA Comparison for high load flows ............................................................................... 15
Table 5: SLA Comparison for low load flows ................................................................................. 16
Table 6: SLA Comparison for double SLA ...................................................................................... 17
Table 7: Delay comparison ............................................................................................................... 18
1. Abstract

With the advent of 5G, IoT and 4k videos, online gaming, movie streaming and other data intensive applications, the demand for data is sky rocketing. Due to this surge in data, the load on the network increases. This heightened network load causes degradation in network performance. Which can lead to the customer Service Provider (CSP)’s loosing revenue if the Service Level Agreement (SLA) are not met.

This report describes how machine learning techniques such as tit for tat can be applied to telecom networks. Machine learning applied to telecom networks help detect congestion and maintain SLAs while increasing yield (revenue).

Several experiments are run with varying conditions on the network, such as low, medium and high loads; different levels of SLA for bandwidth and delay. Once the original conditions are tested without applying any smart blocking techniques, machine learning is applied to detect congestion in the network and block flows to maintain SLA and increase the number of flows that generate revenue.
2. Literature Review

The 5G system would increase the traffic volumes by at least a 100-fold (Olwal et al., 2016) with varying use of the network and QoS of multimedia applications. It is expected that the internet download speeds for each user would be a hundred Mbps and there would be about a hundred billion devices connected to the network (Olwal et al., 2016).

Video is one of the major bandwidth (Chen et al., 2014) utilizing application in the network. Dropping low priority packets prove to provide a better-Quality experience (Caenegem et al., 2008) to customers as in case of videos, which are the major bandwidth utilizers in the network. Caenegem et al. use an algorithm to assign priorities to packets based on the content i.e. packets containing video information would be considered more important than others like image data in the video stream. By using this approach, they were able to obtain better Quality of experience.

Congestion control is one of the poor performing areas (Sharma & Kumar, 2016) in ad-hoc wireless mobile communication systems. With growing traffic, the delay increases and there is a significant increase in the number of packets dropped (Sharma & Kumar, 2016), which leads to a poor QoS.

Performance analysis of delay and latency (Sadeghi & Barati, 2012) is a critical aspect for providing a good QoS. Sadeghi & Barati compare the network queueing performance based on Poisson distribution and Exponential distribution on OPNET (a simulation tool). It was observed that queuing the traffic in an exponential distribution proved to provide better QoS (Sadeghi & Barati, 2012) than letting the traffic flow in a normal fashion. It is expected that routing the traffic according to an exponential distribution should provide a better performance as well.
3. Architecture

The following diagram shows the process flow of the experiments, data collection and analysis involved in this report. It consists of experiment components i.e. the network, the controller and the tools used (Iperf and ping) showed in blue. While the transition phase of data collection and storing is showed in grey. The analysis phase is shown in green.

![Architecture Diagram](image)

*Figure 1: Architecture diagram*

The test is performed on a virtual Software Defined Network (SDN) topology created on Mininet. Working with POX controller set to make the switches work as Layer 2 forwarding devices.
Bandwidth and delay are 2 parameters taken into consideration for measuring network performance SLAs. IPerf is used (between two randomly selected hosts) to generate random data with varying loads in the network and measure the bandwidth. While ping is used to measure the delay in between the same two nodes (as were selected for IPerf) in the network.

3.1. Mininet

Mininet is useful for interactive development, testing, and demos, especially those using OpenFlow and SDN. OpenFlow-based network controllers prototyped in Mininet can usually be transferred to hardware with minimal changes for full line-rate execution.

Mininet creates a realistic virtual network, running real kernel, switch and application code, on a single machine (VM, cloud or native), in seconds, with a single command.

Figure 2: Mininet

3.2. POX SDN Controller

POX Controller is a simple python based open flow controller. It comes as the default controller for Mininet. Its main advantage is that it is simple and light weight. Implying it is not very useful for complex packet processing.
POX started life as an OpenFlow controller, but can now also function as an OpenFlow switch, and can be useful for writing networking software in general. POX officially requires Python 2.7 (though much of it will work fine with Python 2.6), and should run under Linux, Mac OS, and Windows.

### 3.3. IPerf

IPerf is a tool to determine the maximum bandwidth that can be achieved in between two devices. It can generate pseudo random data at a given rate and test for bandwidth.
Figure 4: Iperf

It supports tuning of various parameters related to timing, buffers and protocols (TCP, UDP, SCTP with IPv4 and IPv6). For each test it reports the bandwidth, loss, and other parameters.

3.4. Network Topology

Figure 5: Virtual network topology

**Distributed hybrid tree**- The virtual topology is built with 2 central routers (currently working as just Layer 2 devices) connected to each other. Which are in turn connected to a central switch each. Each of the central switches are in turn connected to 10 other switches. And each switch is connected to 16 hosts. Thus, creating a sub-network of 160 hosts on each side of the network (subnet). Totaling up to 320 hosts in the entire network.
The choice of the network is based on creating a balanced network with sufficient hosts to generate enough load in the network.
4. Methodology

The test is performed by generating flows that would generate load on the network to purposely degrade network performance. If there are flows that do not meet the SLA, flows would start to begin getting blocked. Thus, clearing up the network for successive flows to meet the SLA. Once flows start meeting the SLA, more flows would be generated, keeping a balance of flows meeting the SLA.

A coefficient is used in this algorithm for generating flows. The value of the coefficient is used in a probabilistic approach to generate a flow. A flow will probably be generated based on the value of the coefficient as the probability of generating the flow.

4.1. Creating a topology

A virtual topology is created in Mininet using the Mininet.Topo class as a skeleton. On which various network devices such as switches are added using the self.addSwitch() method. Followed by addition of links joining the switches as described in the topology section above. Based on the topology configuration chosen, switches are added, linked and configured. Varying parameters such as meshed network of switches, or different subnets in the network, number of networks will result in network topology being generated differently.

Finally, all the hosts are added. A suitable IP addressing scheme is chosen based the parameters chosen before initializing the topology. Once that is complete, the configuration and names are dumped on to the host and added to the network topology.
After the topology has been created, the network is started, and all the configurations are dumped on to the devices.

4.2. Generating flows

After a certain interval of time, a new thread is created that creates a new instance of an object and calls a function that selects 2 random hosts from the list of hosts in the network. One from the first network and the other from the 2nd. The first host is made a IPerf server and the other host is made as a client. This is done to stress the link between the two routers. The duration and target bandwidth is determined as the parameters while initiation of the program. Along with that ping is performed for the same durations as the IPerf.
4.3. Collecting data

The result of IPerf and ping are taken every certain interval and put into individual files along with the timestamp for the host pairs. If the hosts pair selected are h11 and h21. For IPerf the files are names as h11-h21.ipperf.dat. While the files generated by ping are named as h11-h21.ping.txt. A few sample files are attached below:

Table 1: h1001-h2040.ipperf.dat

<table>
<thead>
<tr>
<th>Time</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>20:41:42</td>
<td>Client connecting to 10.0.1.40, TCP port 5001</td>
</tr>
<tr>
<td>20:41:42</td>
<td>TCP window size: 85.3 KByte (default)</td>
</tr>
<tr>
<td>20:41:42</td>
<td></td>
</tr>
<tr>
<td>20:41:43</td>
<td>[ 3] local 10.0.0.1 port 59018 connected with 10.0.1.40 port 5001</td>
</tr>
<tr>
<td>20:41:43</td>
<td>[ ID] Interval  Transfer  Bandwidth</td>
</tr>
<tr>
<td>20:41:44</td>
<td>[ 3] 0.0-1.0 sec 1.91 MBytes 16.0 Mbits/sec</td>
</tr>
<tr>
<td>20:41:45</td>
<td>[ 3] 1.0-2.0 sec 1.91 MBytes 16.0 Mbits/sec</td>
</tr>
<tr>
<td>20:41:47</td>
<td>[ 3] 3.0-4.0 sec 1.91 MBytes 16.0 Mbits/sec</td>
</tr>
<tr>
<td>20:41:48</td>
<td>[ 3] 4.0-5.0 sec 1.91 MBytes 16.0 Mbits/sec</td>
</tr>
<tr>
<td>20:41:49</td>
<td>[ 3] 5.0-6.0 sec 1.91 MBytes 16.0 Mbits/sec</td>
</tr>
<tr>
<td>20:41:50</td>
<td>[ 3] 0.0-8.0 sec 15.3 MBytes 16.0 Mbits/sec</td>
</tr>
</tbody>
</table>

Table 2: h1001-h2040.ping.txt

<table>
<thead>
<tr>
<th>Time</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>20:41:42</td>
<td>PING 10.0.1.40 (10.0.1.40) 56(84) bytes of data.</td>
</tr>
<tr>
<td>20:41:42</td>
<td>64 bytes from 10.0.1.40: icmp_seq=1 ttl=64 time=0.091 ms</td>
</tr>
<tr>
<td>20:41:43</td>
<td>64 bytes from 10.0.1.40: icmp_seq=2 ttl=64 time=0.189 ms</td>
</tr>
<tr>
<td>20:41:44</td>
<td>64 bytes from 10.0.1.40: icmp_seq=3 ttl=64 time=0.139 ms</td>
</tr>
<tr>
<td>20:41:45</td>
<td>64 bytes from 10.0.1.40: icmp_seq=4 ttl=64 time=0.077 ms</td>
</tr>
<tr>
<td>20:41:46</td>
<td>64 bytes from 10.0.1.40: icmp_seq=5 ttl=64 time=0.048 ms</td>
</tr>
<tr>
<td>20:41:47</td>
<td>64 bytes from 10.0.1.40: icmp_seq=6 ttl=64 time=0.067 ms</td>
</tr>
<tr>
<td>20:41:48</td>
<td>64 bytes from 10.0.1.40: icmp_seq=7 ttl=64 time=0.059 ms</td>
</tr>
<tr>
<td>20:41:49</td>
<td>64 bytes from 10.0.1.40: icmp_seq=8 ttl=64 time=0.200 ms</td>
</tr>
<tr>
<td>20:41:50</td>
<td>--- 10.0.1.40 ping statistics ---</td>
</tr>
<tr>
<td>20:41:50</td>
<td>8 packets transmitted, 8 received, 0% packet loss, time 7126ms</td>
</tr>
<tr>
<td>20:41:50</td>
<td>rtt min/avg/max/mdev = 0.048/0.108/0.200/0.057 ms</td>
</tr>
</tbody>
</table>
4.4. Parsing the results

Once all the data collection is complete, all the files generated are processed to extract relevant information and stored along with the exact timestamp as a list.

For example, in the first file for reading iperf.dat it will extract the host pair from the file name and store the names in memory. It will read the time data and store it in a new object created for that row, set the host pair to the host pair for that file which is in memory. Next it will extract the bandwidth and map it to the same object. It will add the object to the list and then move on to the next row.

Similarly, for the ping.txt file, it extracts the host pair from the file name. Get the timestamp and rtt from the last part and add them to a list.

4.5. Dumping to the database

Once the data is extracted, the java program when called with the command line argument of load, generates a unique BATCH_ID for that batch, loops through all the data that was extracted, attaches the unique BATCH_ID and prepares SQL insert statements. The statements are executed as a batch for quick loading into the database. A count of the rows inserted for each file is accumulated. At the end of the batch, the accumulated counts for each table is kept in a separate table along with the BATCH_ID to keep track of the number of rows inserted in each batch. Upon completion, the java program returns the BATCH_ID.

Each batch has a set value of link bandwidth, SLA, session parameters (duration and target bandwidth).

Once all the batches for a session are complete, they are inserted into another table called sessionmap along with the session link bandwidth, SLA, session name made from the session.
parameters (duration, target bandwidth) and the number of time the session ran (i.e. no
coefficient, learning or applying phase).

After loading to the database, the data is normalized to the closest 1 second timestamp.

4.6. Generating graphs

The data that is loaded is normalized to a 1 second interval. This gives flexibility to compare the
values in a sequential manner.

![Diagram](image)

*Figure 7: Joins in database for creating graphs*

Views are created to get the required information from each table. Which are joint based on
batch_id.

BandwidthSLA is a view that contains the max, min bandwidths, the number of flows that met 3
levels of SLA for bandwidth. Similarly, DelaySLA contains the max, min delays, the number of
flows that met 3 levels of SLA for delay.

The table batch_run is populated with the number of rows inserted into each table, the number of
flows dropped, and the number of flows blocked. While sessionmap contains a mapping of each
batch to a session, e.g. a session might be DoubleSLA2x1000 indicating that the session was run
with the SLA set to double of what was usually considered. 2 indicated that the test was run for
the 2\(^{nd}\) time i.e. the applying phase after 0 (no coefficient) and 1(learning phase). While 1000
indicated that there were 1000 flows generated in the session.
5. Experiments

The experiments were different values of intervals, bandwidth of the links, target bandwidth, durations etc.

5.1. Intervals

Initially it was anticipated that there might be delays in collecting metrics and the interval was set to be at 10s. Implying a new flow would be generated after 10 seconds. Network statistics would be collected every half interval time, i.e. 5 seconds.

After carefully inspecting all the data collected, it was found that the data collected was precise to 1 second (with variations in a few micro seconds). So, the interval was set to 2 seconds and the network statistics were taken every 1 second.

5.2. Duration

The tests were run with several different values for duration. Ranging from 4 to 32 seconds. Whenever a flow is generated it would be run for that amount of duration. Implying there would duration/interval amount of flows running at the same time during the entire test, except the initiation and tear down of the batch. For example, let’s say we take interval to be 2 and duration to be 8. At t=0 a new flow would be generated. After 2 seconds another flow would be generated, at t=2, taking the current number of flows to 2. Again after 2 seconds, another flow would be generated, at t=4, taking the total number of flows to 4, and so on. After the end of the 8 seconds, the first flow would be teared down, and a new flow be generated. Keeping the total number of flows to be constant at duration/interval.
Table 3: Duration of flows with time

<table>
<thead>
<tr>
<th>Time</th>
<th>f1</th>
<th>f2</th>
<th>f3</th>
<th>f4</th>
<th>f5</th>
<th>f6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.3. Bandwidth

The target bandwidth is varied throughout the tests to generate different amounts of load in the network. Bandwidths are kept as multiples of 16 Mbits/sec. Ranging from 16 Mbits/sec to 128 Mbits/sec.

The product of duration and bandwidth gives the constant amount of load that is always present in the network.
6. Results

6.1. Revenue Increase

The following table shows the comparison for percentage of flows that meet the SLA for different load conditions along with the flows that generate revenue with the experiments being run with the SLA set to bandwidth of 1Mb/sec and delay set to 100 msec. Along with the different phases of the experiment, i.e. without any blocking coefficient, then with learning the coefficient by applying a blocking factor to decide the probability of generating the next flow and modifying the value of the coefficient based on the success of the current flow. And that being followed by the applying phase where the coefficient is fixed to a set value, which is the average value of the coefficient in the learning phase.

Table 4: SLA Comparison for high load flows

<table>
<thead>
<tr>
<th>Load</th>
<th>LengthxBW</th>
<th>CoefficientUse</th>
<th>Delay Set</th>
<th>Revenue</th>
<th>Blocked % BW 2000</th>
<th>% BW 1000</th>
<th>% BW 500</th>
<th>% Del 50</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>8x64</td>
<td>No Coefficient</td>
<td>100</td>
<td>17.69</td>
<td>0.0</td>
<td>97.9</td>
<td>99.6</td>
<td>99.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Learning</td>
<td>100</td>
<td>52.61</td>
<td>104.0</td>
<td>97.7</td>
<td>99.4</td>
<td>99.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Applying</td>
<td>100</td>
<td>58.26</td>
<td>133.0</td>
<td>98.0</td>
<td>99.6</td>
<td>100.0</td>
</tr>
<tr>
<td>16x32</td>
<td></td>
<td>No Coefficient</td>
<td>100</td>
<td>0.0</td>
<td>0.0</td>
<td>91.1</td>
<td>98.3</td>
<td>99.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Learning</td>
<td>100</td>
<td>57.78</td>
<td>779.0</td>
<td>98.3</td>
<td>99.7</td>
<td>99.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Applying</td>
<td>100</td>
<td>77.73</td>
<td>800.0</td>
<td>99.4</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>1024</td>
<td>16x64</td>
<td>No Coefficient</td>
<td>100</td>
<td>0.0</td>
<td>0.0</td>
<td>89.8</td>
<td>98.0</td>
<td>99.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Learning</td>
<td>100</td>
<td>50.02</td>
<td>834.0</td>
<td>97.7</td>
<td>99.2</td>
<td>99.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Applying</td>
<td>100</td>
<td>61.93</td>
<td>859.0</td>
<td>98.6</td>
<td>99.4</td>
<td>99.9</td>
</tr>
</tbody>
</table>

As can be seen from the table above, with heavy loads having load factor of 512 Mb (8sec * 64 Mb/sec or 16sec * 32 Mb/sec) or higher, when no blocking is applied, only 17.69 generate revenue for 8x64 while no flows generate revenue for 16x32 and 16x64. After applying the machine learning technique, to learn an optimal value of the coefficient, the flows that generate revenue are increased to 52.61 (197.31%), 57.7(100%) and 50.02(100%) respectively. After applying this blocking coefficient, we see a further improvement in the flows that meet the SLA.
and generate revenue. The increase in the number of flows that generate revenue being 58.26 (229.33%), 77.73(100%) and 61.93(100%) respectively.

Although it works great for higher loads, it doesn’t increase significant revenue with lower loads as there isn’t enough congestion in the network. As can be seen in the table below which again compares the number of flows that meet the SLA for bandwidth and delay.

*Table 5: SLA Comparison for low load flows*

<table>
<thead>
<tr>
<th>Load</th>
<th>LengthxBW</th>
<th>CoefficientUse</th>
<th>Delay Set</th>
<th>Revenue Use</th>
<th>CoefficientUse</th>
<th>% BW 2000</th>
<th>% BW 1000</th>
<th>% BW 500</th>
<th>% Del 50</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>4x16</td>
<td>No Coefficient</td>
<td>100</td>
<td>994.75</td>
<td>No Coefficient</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>96.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Learning</td>
<td>100</td>
<td>994.50</td>
<td>Learning</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>99.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Applying</td>
<td>100</td>
<td>991.79</td>
<td>Applying</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>99.1</td>
</tr>
<tr>
<td>128</td>
<td>4x32</td>
<td>No Coefficient</td>
<td>100</td>
<td>713.38</td>
<td>No Coefficient</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>73.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Learning</td>
<td>100</td>
<td>730.00</td>
<td>Learning</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>72.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Applying</td>
<td>100</td>
<td>734.05</td>
<td>Applying</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>73.3</td>
</tr>
<tr>
<td>8x16</td>
<td>No Coefficient</td>
<td>100</td>
<td>173.21</td>
<td>No Coefficient</td>
<td>99.9</td>
<td>100.0</td>
<td>100.0</td>
<td>17.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Learning</td>
<td>100</td>
<td>199.54</td>
<td>Learning</td>
<td>99.7</td>
<td>100.0</td>
<td>100.0</td>
<td>20.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Applying</td>
<td>100</td>
<td>213.03</td>
<td>Applying</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>21.5</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>8x32</td>
<td>No Coefficient</td>
<td>100</td>
<td>15.48</td>
<td>No Coefficient</td>
<td>98.7</td>
<td>99.7</td>
<td>100.0</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>Learning</td>
<td>100</td>
<td>45.89</td>
<td>Learning</td>
<td>98.5</td>
<td>99.7</td>
<td>100.0</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Applying</td>
<td>100</td>
<td>40.92</td>
<td>Applying</td>
<td>98.3</td>
<td>99.6</td>
<td>99.9</td>
<td>4.6</td>
<td></td>
</tr>
</tbody>
</table>

From the table above, it is seen that with light loads having load factor of 64 Mb (4sec * 16 Mb/sec) the, when no blocking is applied, almost all flows meet the SLA 994.75 generate revenue. For 4x32 there are 713.38 flows that generate revenue and for 8x16 there are 173.21 flows that generate revenue. For 8x32 there are only 15.48 flows that generate revenue. After applying the machine learning technique, to learn an optimal value of the coefficient, the flows that generate decreased to 991.79(-0.02 %) while for 4x32 the flows that generate revenue increased to 730(2.33 %). While for 8x16 the flows that generate revenue increased to 199.54(15.20 %) and 40.92(196.45 %) for 8x32. After applying this blocking coefficient, we see a improvement in the flows that meet the SLA and generate revenue for 4x32 and 8x16. The increase in the number of flows that generate revenue being 734.05 (2.90 %) and 213.03 (22.99 %).
respectively. While for 4x16 and 8x32, it decreased from the learning phase, to 991.79(-0.30 %) for 4x16 and 40.92(164.34 %) for 8x32.

With heavy loads we see more flows meeting the SLA. After the algorithm was applied, we see a significant improvement.

6.2. Doubling SLA

Like the previous table, the following table shows the comparison for percentage of flows that meet the SLA for different load conditions along with the flows that generate revenue with the experiments being run with the SLA set to bandwidth of 1Mb/sec but the delay set to 50 msec.

Table 6: SLA Comparison for double SLA

<table>
<thead>
<tr>
<th>SLA Comp</th>
<th>Revenue</th>
<th>Blocked</th>
<th>% BW 2000</th>
<th>% BW 1000</th>
<th>% BW 500</th>
<th>% Del 50</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>994.01</td>
<td>0.0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td>993.77</td>
<td>0.0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td>992.77</td>
<td>0.0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td>729.37</td>
<td>0.0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>72.0</td>
</tr>
<tr>
<td></td>
<td>742.22</td>
<td>19.0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>75.6</td>
</tr>
<tr>
<td></td>
<td>748.62</td>
<td>0.0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>74.8</td>
</tr>
<tr>
<td></td>
<td>152.82</td>
<td>0.0</td>
<td>99.8</td>
<td>100.0</td>
<td>100.0</td>
<td>15.3</td>
</tr>
<tr>
<td></td>
<td>366.0</td>
<td>99.9</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>67.0</td>
</tr>
<tr>
<td></td>
<td>569.05</td>
<td>998.0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>94.4</td>
</tr>
<tr>
<td>128</td>
<td>12.03</td>
<td>0.0</td>
<td>99.0</td>
<td>99.8</td>
<td>100.0</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td>171.62</td>
<td>689.0</td>
<td>99.7</td>
<td>99.9</td>
<td>100.0</td>
<td>55.0</td>
</tr>
<tr>
<td></td>
<td>189.68</td>
<td>688.0</td>
<td>99.6</td>
<td>99.9</td>
<td>100.0</td>
<td>60.6</td>
</tr>
<tr>
<td></td>
<td>19.13</td>
<td>0.0</td>
<td>97.9</td>
<td>99.5</td>
<td>99.9</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td>105.13</td>
<td>792.0</td>
<td>99.4</td>
<td>99.9</td>
<td>100.0</td>
<td>50.3</td>
</tr>
<tr>
<td></td>
<td>127.82</td>
<td>702.0</td>
<td>99.2</td>
<td>99.8</td>
<td>100.0</td>
<td>42.7</td>
</tr>
<tr>
<td>256</td>
<td>0.00</td>
<td>0.0</td>
<td>91.3</td>
<td>98.4</td>
<td>99.9</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>74.46</td>
<td>847.0</td>
<td>99.5</td>
<td>100.0</td>
<td>100.0</td>
<td>48.3</td>
</tr>
<tr>
<td></td>
<td>67.37</td>
<td>872.0</td>
<td>99.6</td>
<td>100.0</td>
<td>100.0</td>
<td>52.2</td>
</tr>
<tr>
<td>512</td>
<td>0.00</td>
<td>0.0</td>
<td>90.5</td>
<td>98.2</td>
<td>99.7</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>50.51</td>
<td>885.0</td>
<td>98.8</td>
<td>99.7</td>
<td>99.9</td>
<td>43.5</td>
</tr>
<tr>
<td></td>
<td>52.57</td>
<td>876.0</td>
<td>98.2</td>
<td>99.3</td>
<td>99.9</td>
<td>42.1</td>
</tr>
</tbody>
</table>

After increasing the set SLA for delay, we see similar results. The number of flows that are successful have increased significantly whenever the flows run for more than 8seconds. When the flows are run without any blocking coefficient, the number for flows that generate revenue
are 994.01 for 4x16, 729.37 for 4x32, 152.82 for 8x16, 12.03 for 8x32 and 19.13 for 8x64. While for 16x32 and 16x64 there are no flows that generate revenue when no blocking is used.

After applying the machine learning algorithm, to learn the value of the coefficient, the number of flows that generate revenue is decreased by a small fraction to 993.77(-0.02 %) for 4x16 while for the rest of the cases, the number of flows that generate revenue are increased to 742.22(1.76 %), 425.41(167.86 %), 171.62(1326.60 %), 105.13(449.56 %), 74.46(100.00 %), 50.51(100.00 %) respectively.

After applying this learned coefficient, the number of decreased for the 4x16 to 992.77(-0.12 %), while the number of flows that generate revenue for the rest of the cases increased to 748.62(2.64 %), 569.05(258.30 %), 189.68(1476.72 %), 127.82(568.17 %), 67.37(100.00 %), 52.57(100.00 %) respectively.

6.3. Delay comparison

The following table shows the comparison of average bandwidth; and minimum, average and maximum delay in the network for different cases of length and bandwidth.

<table>
<thead>
<tr>
<th>Load</th>
<th>Lengthx..</th>
<th>CoefficientUse</th>
<th>Avg BW</th>
<th>Avg Delay</th>
<th>Max Delay</th>
<th>Min Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>8x64</td>
<td>No Coefficient</td>
<td>15,190,754</td>
<td>195</td>
<td>420</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Learning</td>
<td>16,881,700</td>
<td>182</td>
<td>450</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Applying</td>
<td>17,559,207</td>
<td>186</td>
<td>502</td>
<td>24</td>
</tr>
<tr>
<td>16x32</td>
<td></td>
<td>No Coefficient</td>
<td>7,658,952</td>
<td>325</td>
<td>603</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Learning</td>
<td>23,421,709</td>
<td>181</td>
<td>573</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Applying</td>
<td>27,466,573</td>
<td>148</td>
<td>532</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>16x64</td>
<td>No Coefficient</td>
<td>7,644,985</td>
<td>326</td>
<td>626</td>
<td>134</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Learning</td>
<td>33,546,927</td>
<td>205</td>
<td>812</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Applying</td>
<td>39,716,017</td>
<td>162</td>
<td>717</td>
<td>0</td>
</tr>
</tbody>
</table>

As seen from the table above, the average delay has gone down with blocking in both cases of learning the coefficient and applying the value of the coefficient in the algorithm.
6.4. Coefficient Values

The following diagrams show the charts for values of coefficient for 3 different scenarios.

The first chart shows the values of coefficient stacked on top of each other for different values of length and bandwidth.

Figure 8: Graph showing value of coefficients over time

<table>
<thead>
<tr>
<th>LengthxBW</th>
</tr>
</thead>
<tbody>
<tr>
<td>8x64</td>
</tr>
<tr>
<td>16x32</td>
</tr>
<tr>
<td>16x64</td>
</tr>
</tbody>
</table>
This graph shows the values of the coefficients at the initial few times of the experiment being performed. Looking at the graph in the learning phase, the value of the coefficient keeps on decreasing as the load keeps on increasing over time.

However, if we look over a wide range of time as in the graph below:

![Graph showing value of coefficients over long range](image)

*Figure 9: Graph showing value of coefficients over long range*

After looking at the coefficient in the learning phase, over a wide range of time, the coefficient eventually saturates to a stable value, average of which is applied in the 3\textsuperscript{rd} phase which is the applying phase. Where the average value of the coefficient from the learning phase is used as the probability of generating new flows.

For low load such as 8x64, value of the coefficient fluctuates from the mid to the higher end indicating that the network frequently changes clears up and gets blocked again.
We can see the graph as below.

**Figure 10:** Graph showing value of coefficients for duration: 8sec bandwidth: 64 Mb/sec

For higher loads such as 16x32 the graph is below:
In this case the value of the coefficients dips to a low value as there are more flows running simultaneously in the network. And as we keep blocking the flows, the value of the coefficient increases and then comes down again.

A similar case is observed for loads such as 16x64
6.5. Extreme cases

The following table shows the comparison of the number of flows that meet the SLA for bandwidth and delay for extreme cases of length and target bandwidth.

<table>
<thead>
<tr>
<th>SLA Comp</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LengthxBW</strong></td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>1x256</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>256x1</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

When the experiment is repeated with extreme values of length and bandwidth, it is seen that the coefficient doesn’t help much.
The number of flows that generate revenue without applying blocking are 500.5 for 1x256 and 91.45 for 256x1. While learning the optimal value of the coefficient, the number of flows that generate revenue are 500.5 for 1x256 which is the same without applying any blocking. While for the 256x1 no flows generate any revenue. While applying the coefficient to the network, the number of flows that generate revenue are increased to 101.96(9.1 \%) for 256x1 while it remains the same for 1x256.

It is seen that fewer the flows running simultaneously in the network with very high target bandwidth, the network performs better. Unlike in the case of very high number flows running with very low target bandwidth. In which case the network performance is completely degraded.
7. Discussion

7.1. Limitations

The experiments were performed on a Linux machine installed on a Personal Computer (PC) which have limited amount of resources at its disposal.

The database is shifted to the cloud, which does reduce the load on the computer, but still is not very efficient to handle high loads of data being generated in the virtual network.

This experiment might be run on super computers/research computers with much higher computing capability.

7.2. Algorithm Improvements

The experiments were tested with only one strategy – tit for tat, which works in increasing revenue. But might not be the best algorithm that might be applied to increasing revenue.

One more improvement could be tit-tit-tat-tat i.e. modifying the coefficient only if there are 2 consecutive flows that don’t meet the SLA. And, increasing the coefficient if there are 2 successful flows that meet the SLA.

Similarly, more experiments can be run with extending the learning phase to more observations in network behavior.
8. Learnings

8.1. Moving from VM to local installation of Ubuntu

As the size of the virtual topology kept increasing and the load on the network started increasing, VM was unable to handle all the load by itself. There were several instances where the CPU was completely occupied. Since in a VM the resources are shared with the host PC, there leaves a significant reduction in the amount of resources available for Mininet to keep the network occupied.

8.2. Moving the local SDN controller and database to the cloud

To ease up the resources even further, the SDN controller was tried to move to the cloud on Amazon Web Services (AWS) cloud. It was a simple transition. POX controller was cloned from github onto amazon Linux installation on AWS. Security measures taken to allow only hosts from only the local PC’s IP address and the port for the POX controller that is 3366.

The database server was one bulky application that consumed a lot of processing power and memory. This was also moved to the cloud. Initially it was tried to move onto Amazon Relational Database Services (RDS), however there were issues connecting to Amazon RDS from java. This was mainly because Amazon RDS gives a single database even on Microsoft SQL Server. To overcome this issue, he database was moved to Microsoft Azure. Which allowed creating of multiples databased on a single server, just like on a physical device.

8.3. Moving back SDN controller to the local machine

It was observed that communicating with the SDN controller on the cloud introduced lot of delay in the network. So, the controller was moved back to the local machine on ubuntu alongside Mininet.
However, the database server continued to stay on the cloud, as the database interaction of the program was at the end of the program. And at times during the generation of a new flow.

8.4. Moving the installation from a computer running on i5 to i7 processor

Initial tests were run on a 4th gen i5 processor that had a 3MB cache memory. Which lead to a lot of packets being dropped. The processor couldn’t process all the packets that were generated in the network.

8.5. Deciding on the conditions for changing in the value of the coefficient

Tit for tat was applied to the value of the coefficient. In the learning phase, the coefficient was started with a value of 1. If a flow did not meet the SLA, the value of coefficient would be decreased by multiplying it with 0.9. If later a flow satisfied the SLA, the value of the coefficient would be multiplied by 1.1. This led to the value of the coefficient going down to 0 with heavy loads and no new flows being generated, which implied the value of the coefficient would never increase. So, a bottom limit of 0.1 was set and an upper limit was set to 1 for keeping a limit on the ever-increasing value of the coefficient for lower loads.
9. Summary

The virtual network topology has been set up in Mininet with 160 hosts on each side of the network, with 16 hosts connected to a switch. Totaling up to 320 hosts in the entire network. Flows are generated between a randomly chosen host from the first network to another randomly chosen host in the second network. Thus, making the all the packets flow through the link between the central routers and congesting that link.

Different sessions have been run with different values of duration and target bandwidth to test different load conditions on the network. Initially experiments were run with each simulation generating 100 flows each. It showed how machine learning helped identify congestion and help meet SLA for flows in the network and increase number of flows that generate revenue. To verify the results further, the experiments were run with 500 flows and later with 1000 flows in each simulation.

Once the session was run without any blocking coefficient, all the sessions were run again with tit for tat algorithm to learn an optimistic value of blocking coefficient. And this value was later used to apply the blocking coefficient on the network to maintain SLA and increase the number of flows generating revenue.
Applying tit for tat, the number of flows that meet the SLA, increased. This has been observed mostly with high loads which have higher changes of congestion and the network performance degrading. This is most prominently seen in the sessions with 16x64, where the revenue earned without applying the algorithm was 0, as there were no flows that met the SLA. After applying the algorithm to learn the coefficient, the number of flows that generate revenue was increased to 50.02 (100 %). And while applying this learned value of the coefficient to generate flows, the revenue increased further to 61.93 (100 %).
In extreme cases, the fewer the number of flows, the better of the performance. Like for example in the case of duration =1 second and bandwidth=256Mb/sec, 500 flows generate revenue. As the number of flows are increased, the performance degrades a lot. For example when the duration was made to 256 seconds (128 flows running simultaneously), the number of flows generating revenue reduced to 0.

Although the results show an improvement, with applying the coefficient, it might not be the best approach. Future work, focusing on getting a better value of blocking coefficient might prove to be better.
10. References

https://doi.org/10.1017/S026988914000186


11. Appendix

11.1. Folder structure

<table>
<thead>
<tr>
<th>File Type</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Folder</td>
<td>archive</td>
</tr>
<tr>
<td>Python Script</td>
<td>Properties.py</td>
</tr>
<tr>
<td>Shell Script</td>
<td>call.ksh</td>
</tr>
<tr>
<td>Shell Script</td>
<td>initiateSession.ksh</td>
</tr>
<tr>
<td>Java Executable</td>
<td>monitoring.jar</td>
</tr>
<tr>
<td>Python Script</td>
<td>nIperfSessions.py</td>
</tr>
<tr>
<td>Shell Script</td>
<td>results.ksh</td>
</tr>
<tr>
<td>Text File</td>
<td>dbcon.properties</td>
</tr>
</tbody>
</table>

The package to create virtual SDN topology in Mininet, generate and collect data consists of several Python scripts, Shell scripts and a java executable file. The source codes for all the files can be found at [https://github.com/nabarunjana/mininet-test](https://github.com/nabarunjana/mininet-test).
11.2. Initialization flow

![Diagram of call.ksh, InitiateSession.ksh, sudo python niperfSessions.py, monitoring.jar load]

11.3. File Description

11.3.1. archive
The archive folder consists of archived files generated in batches with specific parameters such as duration of each flow, the target bandwidth for each flow, the type of the flow: e.g. without any coefficient, learning the value of the coefficient, applying the coefficient, etc.

11.3.2. Properties.py
This python script is written to help ease the make changes to the database the program connects to. This is like the Properties class in java. This has methods for reading a file and getting a property. This is used in the program to read database connection properties specified in the dbcon.properties file.

11.3.3. call.ksh
This is a shell script that is the starting point for the program that can keep track of the batches that run with the required session parameters and load the details later to the database into a table called sessionmap. This calls initiateSession.ksh with the values of length of the flows, the target bandwidth and the phase of the experiment.
11.3.4. initiateSession.ksh
This shell script does some maintenance jobs such as zipping all the files generated and moving zipped files to the archive directory.

Once the folder cleared, it calls the nIperfSessions.py to create the virtual topology with the parameters from call.ksh and decides upon the number of flows to generate during the experiment and start generating flows and collecting data.

11.3.5. results.ksh
This shell script keeps a track of the pairs that were selected along with the bandwidth and amount of data that were transferred between them. The results for each host pair are appended to the results.txt file.

11.3.6. nIperfSessions.py
This is the main python script that creates the virtual topology after inheriting Mininet.Topo class, and adding switches, hosts, and links. Once the topology is set up, it starts the network and dumps all the configurations.

Once the topology is set up, it will set up monitoring the network performance ad host device statistics.

It then creates new threads for each flow, which in then selects 2 random hosts (one from each subnet) and performs IPerf and ping to measure bandwidth and delay between them. The output of these tests is put into individual files for each pair that is selected.
11.3.7. **monitoring.jar**

This is a java executable file that when called with the command line argument of load, loops through all the files in the current folder and checks for the desired generated files, parses them, and loads them into the database as a batch. At the end it inserts a row into the batch_run table that has a count of the number of rows that were inserted into each table.

To create the database schema automatically, the monitoring.jar application may be called with the create command line argument. This relies on the dbcon.properties for getting connection details. To load an archive file again, the monitoring.jar application may be called with the zip command line argument followed by an optional BATCH_ID.

To delete all entries related to one batch I'd, the monitoring.jar application may be called with the delete command line argument followed by the BATCH_ID.

11.3.8. **dbcon.properties**

This file has details for the program to connect to the database, with simple parameters such as username, password, database etc. This is used by nIperfSessions.py and monitoring.jar.

e.g.

```plaintext
host=18.222.28.122
user=mydbuser
port=1433
password=passwd
database=mydb
dbserver=sqlserver
```
11.4. Setup

The whole program works in a Linux environment with Mininet, Python, Java, and SQL server.

The below commands are useful to install the necessary components in Ubuntu and to set up the environment with SNMP and necessary tools:

```bash
sudo apt-get update
sudo apt-get -y install git
cd ~
git clone git://github.com/mininet/mininet
mininet_installed=`./mininet/util/install.sh -a|tail`
echo $mininet_installed
sudo apt-get -qqy install curl gawk net-tools python-pip python-pyodbc openjdk-8-jre freetds-bin freetds-dev
pip install numpy
sudo apt-get -qqy install snmp snmpd snmp-mibs-downloader
echo "Downloading MIBs"
echo `sudo download-mibs|wc -l`
mkdir -p ~/.snmp/mibs
sudo sed -i -re 's/([a-z]{2})\(#rocommunity)rocommunity/g' /etc/snmp/snmpd.conf
sudo service snmpd restart

# Clone mininet - test
codethere=`ls ~/mininet-test|wc -l`
if [ $codethere -le 0 ]; then
git clone https://github.com/nabarunjana/mininet-test
mkdir mininet-test/archive
chmod 777 mininet-test/*sh  mininet-test/*py
fi
sudo curl https://packages.microsoft.com/keys/microsoft.asc | apt-key add -
sudo curl https://packages.microsoft.com/config/ubuntu/16.10/prod.list >
/etc/apt/sources.list.d/mssql-release.list

sudo apt-get update
sudo ACCEPT_EULA=Y apt-get -qqy --allow-unauthenticated install msodbcsql
# optional: for bcp and sqlcmd
sudo ACCEPT_EULA=Y apt-get -qqy --allow-unauthenticated install mssql-tools
echo 'export PATH="$PATH:/opt/mssql-tools/bin"' >> ~/.bash_profile
echo 'export PATH="$PATH:/opt/mssql-tools/bin"' >> ~/.bashrc
source ~/.bashrc
# optional: for unixODBC development headers
sudo apt-get -qqy install unixodbc-dev
```
11.4.1. Known issues:

Sometimes, curl fails to add the mssql source to add to the source list and needs to be added by exclusively switching to the root user (sudo su) and then executing the command and then exit back to user mode. And then continue with the further steps of installing msodbcsql and mssql-tools.