Using Machine Learning Technique to Estimate TCP Round Trip Time

Dinesh Reddy Baddam (111481934)
Swetheendra Tallamraju (111401594)
Rakshith Reddy Polireddy (111498989)

Introduction

Round-trip time (RTT) is defined as time required for a single packet to travel from a specific source to a specific destination and back again. RTT depends mainly on the data-transfer rates of the source connection, speed and traffic at the intermediate nodes, number of hops to reach destination etc.. Thus several network applications and protocols use the RTT to estimate network load or congestion, and therefore it is necessary to measure it efficiently. TCP is one of the protocols which bases its error-control and congestion control functions on the estimated RTT instead of relying on feedback from the network. It uses a timeout/retransmission mechanism to recover from lost segments. This (RTO)timeout is estimated using RTT and it is necessary that RTO be higher that RTT to avoid unnecessary retransmissions. It should not be too high that the RTT transmitter not wait for too long to retransmit an actually lost packet.

Problem Statement

Current TCP Estimator uses smoothed RTT with an alpha value of 0.125 and sets the RTO to be three times that of the smoothed RTT. Smoothed RTT does not increase very fast as the alpha value is just 0.125, thus also causing the RTO to be increased slowly and in some cases this Estimator increases RTO when it should be decreased. Therefore we need new RTT estimator that estimates values effectively and quickly when there are packet losses and also get self adjusted in congested networks. Effective estimation of RTT helps us in setting proper RTO thus increasing the throughput.

Proposal

In this paper we propose Linear Regression, a Machine Learning Paradigm to estimate RTT and evaluate its performance on three TCP variants namely TCP Reno, TCP Vega and TCP Cubic on different scenarios.

Approach

Dataset needed to train the Linear regression model is generated by using hping3 and tcptraceroute on 5000 websites from Cloudflare. Hping and tcptraceroute are used as they ping using TCP packets (normal ping and traceroute uses ICMP packets). From the obtained dataset 1)Number of hops between Source and Destination 2)TTL 3)Minimum RTT 4)Maximum RTT are selected as features and are used to train the model offline and obtain the coefficients. Online training is not considered as it would incur an overhead and would slow down the RTT estimation process which is crucial. Finally performance tests are performed on the normal TCP version and also the modified TCP version.

Solution

Coefficients of the regression model are generated for 3 linux machines using a different TCP variant namely TCP Reno, TCP Vegas, TCP Cubic.

Following are linear regression outputs for all 3 different variants and their corresponding Mean Absolute Percentage Errors(MAPE).

In Kernel where **Cubic** is deployed
Coefficients: [-0.52218434, 0.01509783, 0.58899585, 0.41376639, 0.00157301]
MAPE: 3.71825

In Kernel where **Reno** is deployed
Parameters: [0.12877645, -0.01079089, 0.9933091, 0.04338126, -0.00311428]
MAPE: 4.6725

In Kernel where **Vegas** is deployed
Parameters: [-0.34314743, 0.00493765, 0.6949473, 0.3145146, 0.00192231]
MAPE: 2.8785

The prediction of rtt is now incorporated into TCP RTT estimation process. This is achieved by modifying two functions in the Linux kernel and they are “tcp_rtt_estimator” and “tcp_set_rto”.

Now because of the tight estimation of RTT and RTO, the TCP started retransmitting the packets rigorously. Thus to reduce the number of retransmissions, a running average error is calculated each time and it used in evaluating RTO. This reduced the number of retransmissions drastically. Since, we are calculating the running average unlike the normal TCP, our modified TCP variant would increase and decrease the RTO fastly. Thus our requirement of increasing and decreasing RTO with fast changes in RTT is achieved.

```
func tcp_rtt_estimator(actual_rtt):
        if first packet of connection:
                predicted_rtt = predict(hops,min,max,ttl)
                error = (predicted_rtt - actual_rtt)
                deviation_rtt = error;
        else:
                predicted_rtt = predict(hops,min,max,ttl)
                error = (error + (predicted_rtt - actual_rtt))/2
                deviation_rtt = error;

func tcp_set_rto(rtt,dev_rtt):
        rto = rtt+dev_rtt
```

Thus the code is compiled and evaluation of the following modified TCP version with their corresponding unmodified versions is performed by using tcpdump,tshark while downloading huge files.

Evaluation Setup

We used 3 Linux Centos containers for our experimental setup and deployed CUBIC, RENO and VEGAS on them.

Specifications of containers:
*Centos version**:** 4.6.0*
*RAM**:** 2GB*
*DISK**:** 50BG*

In all experiment scenarios we have analyzed performance of new algorithm by downloading files where Centos container acted as client and Azure public data set [1] acted as server.

On each of tcp variant kernel, we have downloaded data from Server and ran Tcpdump in background. We repeated this for 5 times and tabulated parameters like Download time and Number of retransmitted packets and analyzed them.

We have compared the original TCP with the changed TCP when there is no traffic, high traffic and low traffic in background while downloading a huge file from a public server. Repeated the same all the 3 variants of TCP.

While downloading the large data file from a server, to generate low/high traffic parallely on the same machine we have pinged facebook with tcp syn packets using the command below.

```
# for high traffic
$	hping3 -i u100 -c 900000 -S -p 80 facebook.com

# for low traffic
$	hping3 -i u100000 -c 900000 -S -p 80 facebook.com
```

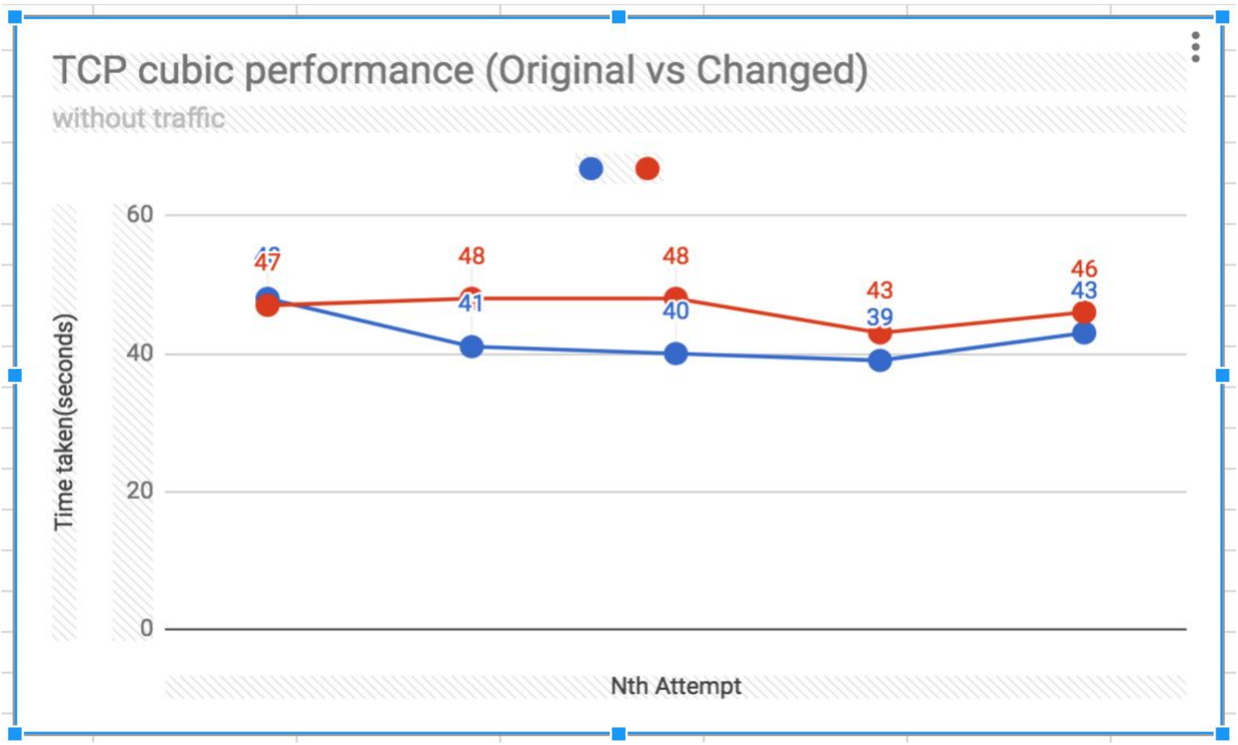
The above commands ping facebook.com with tcp packets for 900000 times at intervals of 100 microseconds and 100000 microseconds.

We wrote a script(hitavg.bash) which does the following:

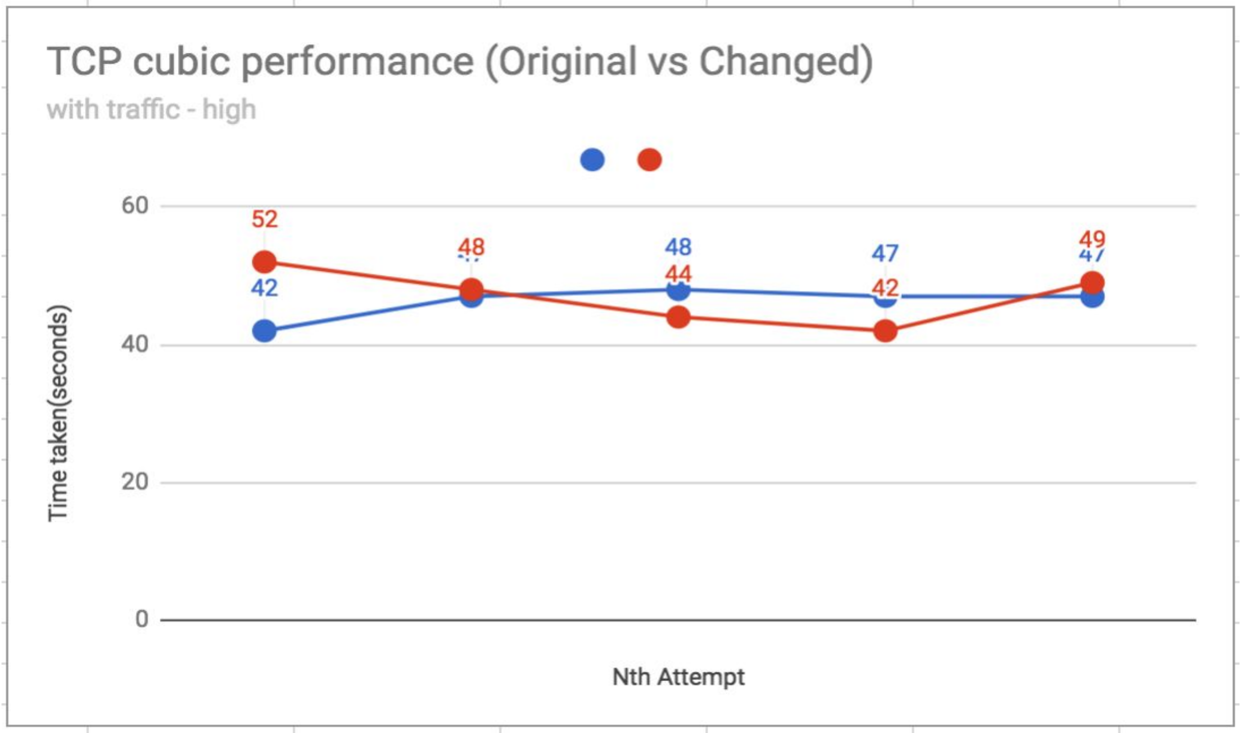
1. Sniffs packets with host=public server using “tcpdump” into a pcap file
   ```
   $	tcpdump -i $interface ip host $siteip -w data.pcap &
   ```
2. Downloads the huge data file using “wget” command
   ```
   $	wget http://$url_of_file
   ```
3. Kills the tcpdump process, as we are done downloading
   ```
   $	pid=$(ps -e | pgrep tcpdump) &&  kill -2 $pid
   ```
4. Count the number of retransmitted packets data from pcap file using “tshark”
   ```
   $	tshark -Y "tcp.analysis.retransmission" -r data.pcap
   ```
5. Repeat the above 1 to 4 steps few more times and record the results each time.
6. Do all the above steps for 3 variants of TCP

Results

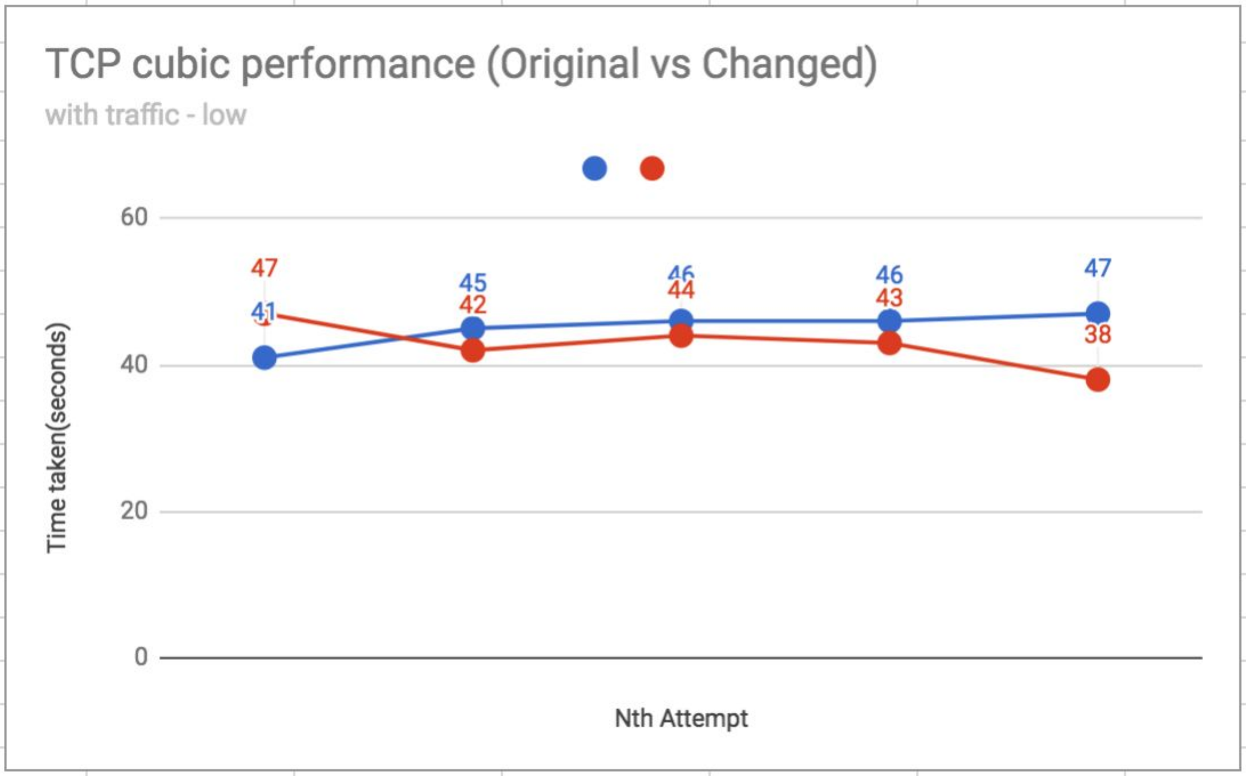
TCP Cubic - Time Taken Analysis without any traffic



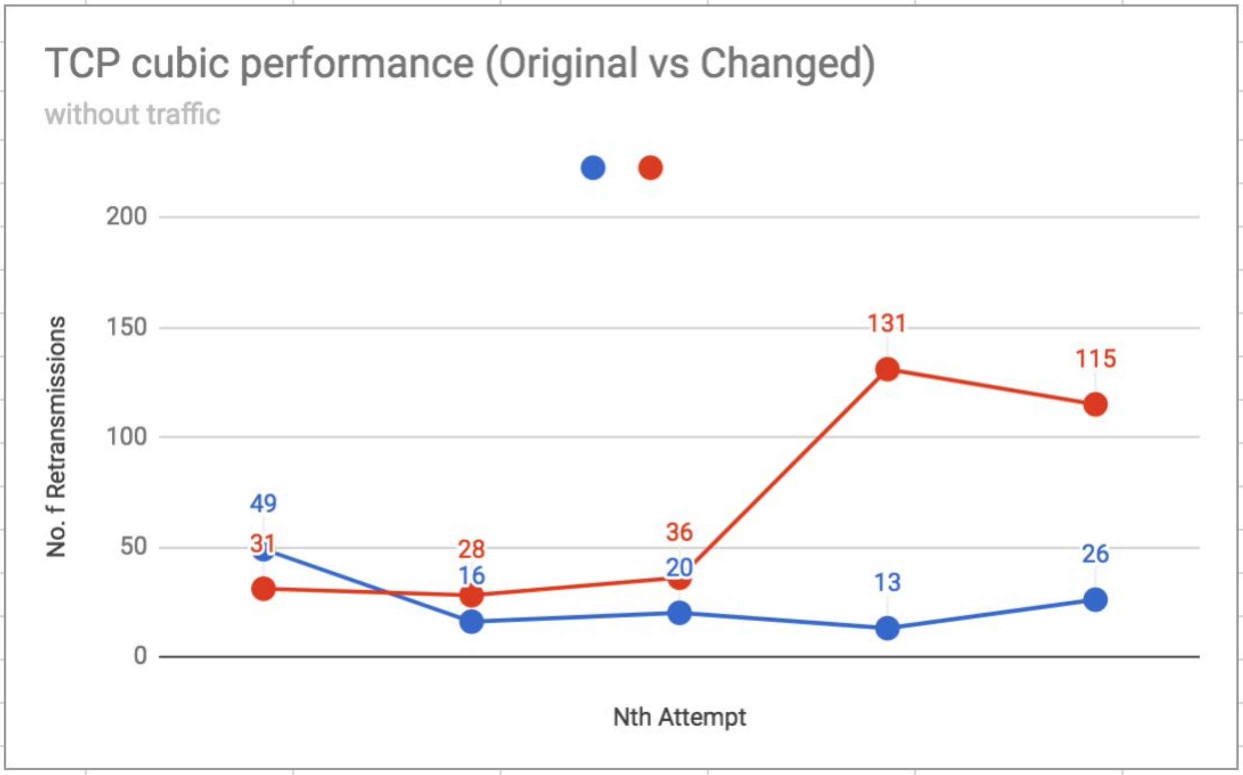
TCP Cubic - Time Taken Analysis with high traffic



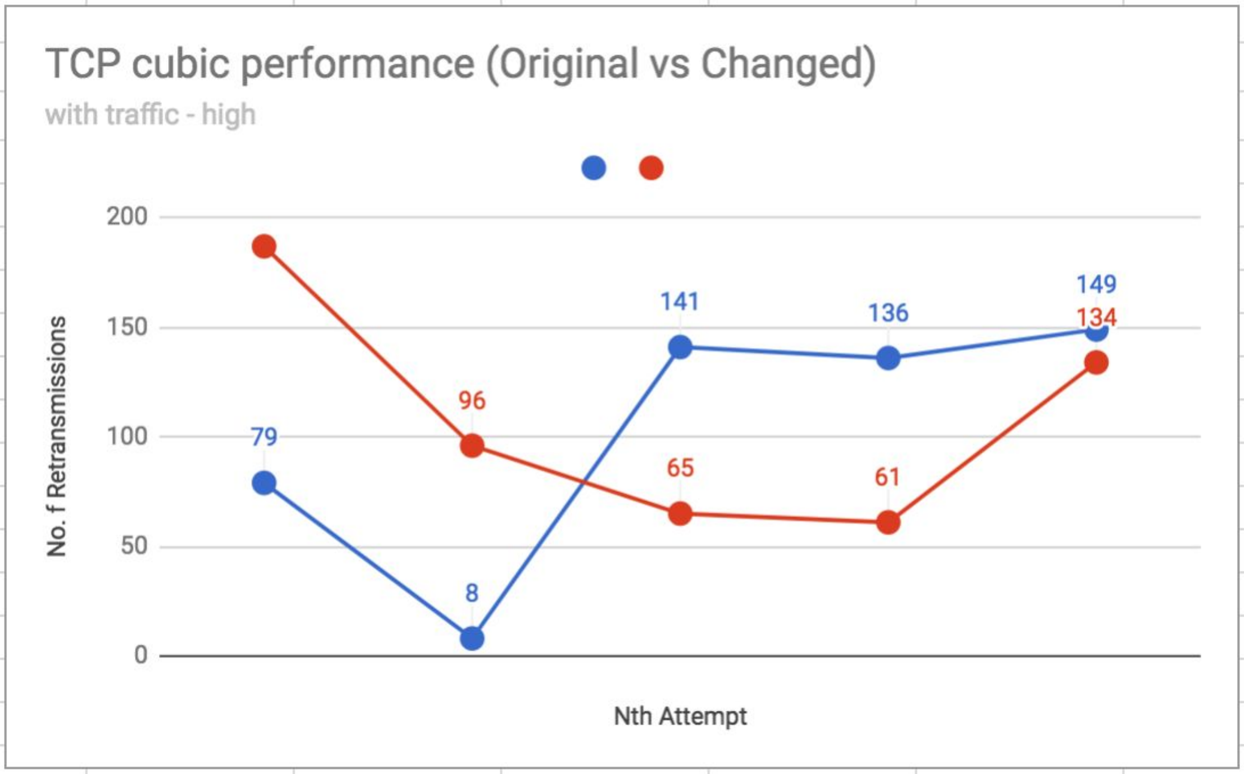
TCP Cubic - Time Taken Analysis with low traffic



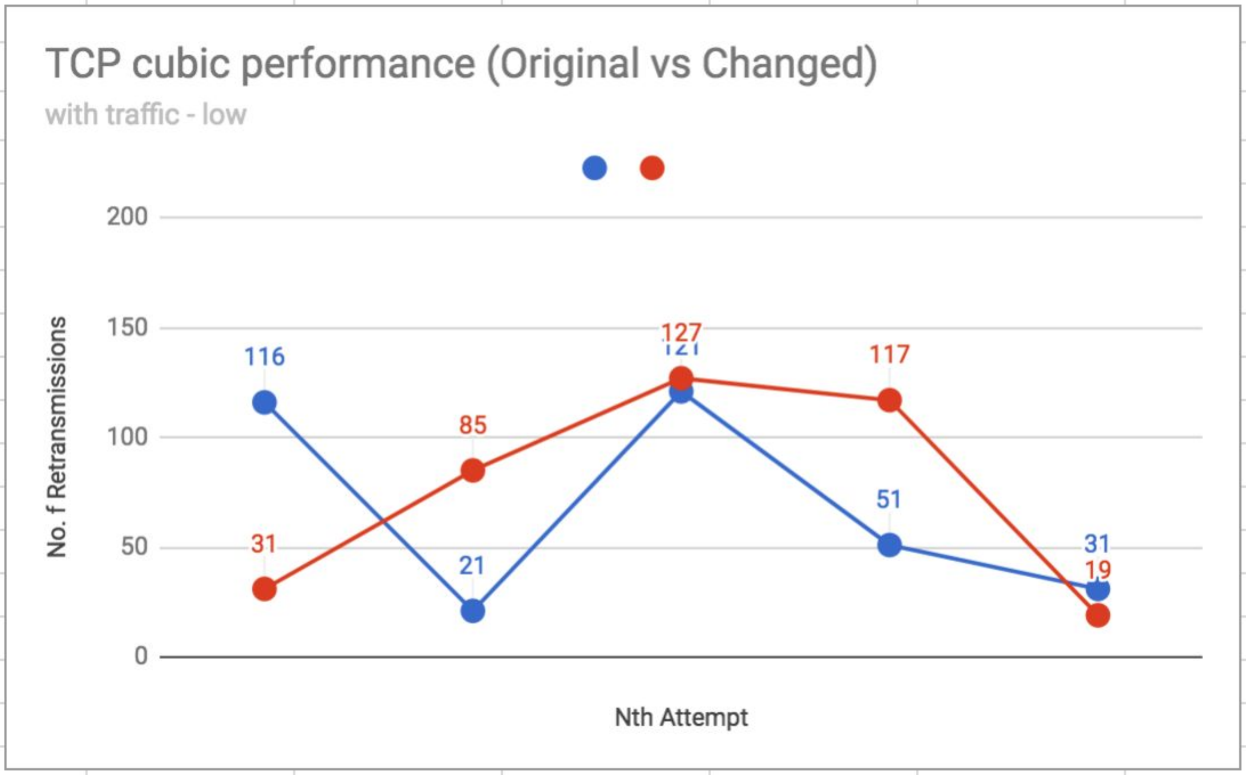
TCP Cubic - No. of Retransmitted packets without traffic



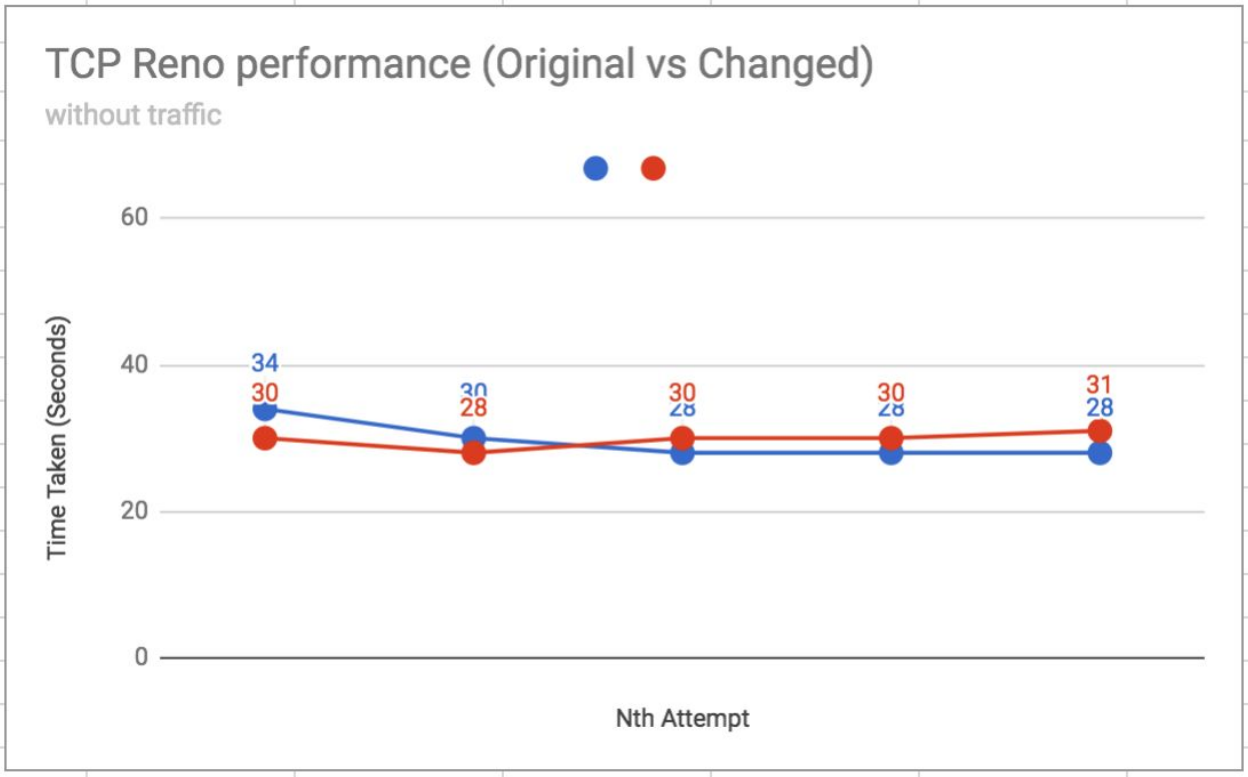
TCP Cubic - No. of Retransmitted packets with high traffic



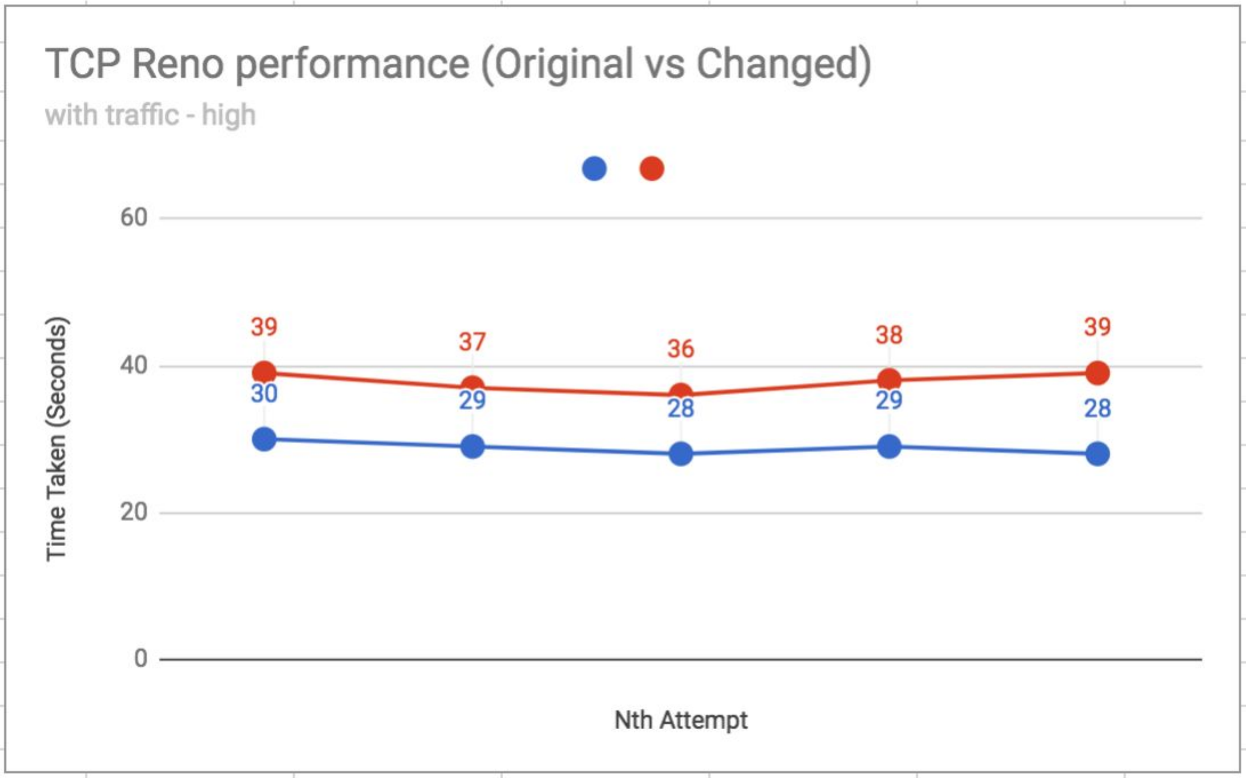
TCP Cubic - No. of Retransmitted packets with low traffic



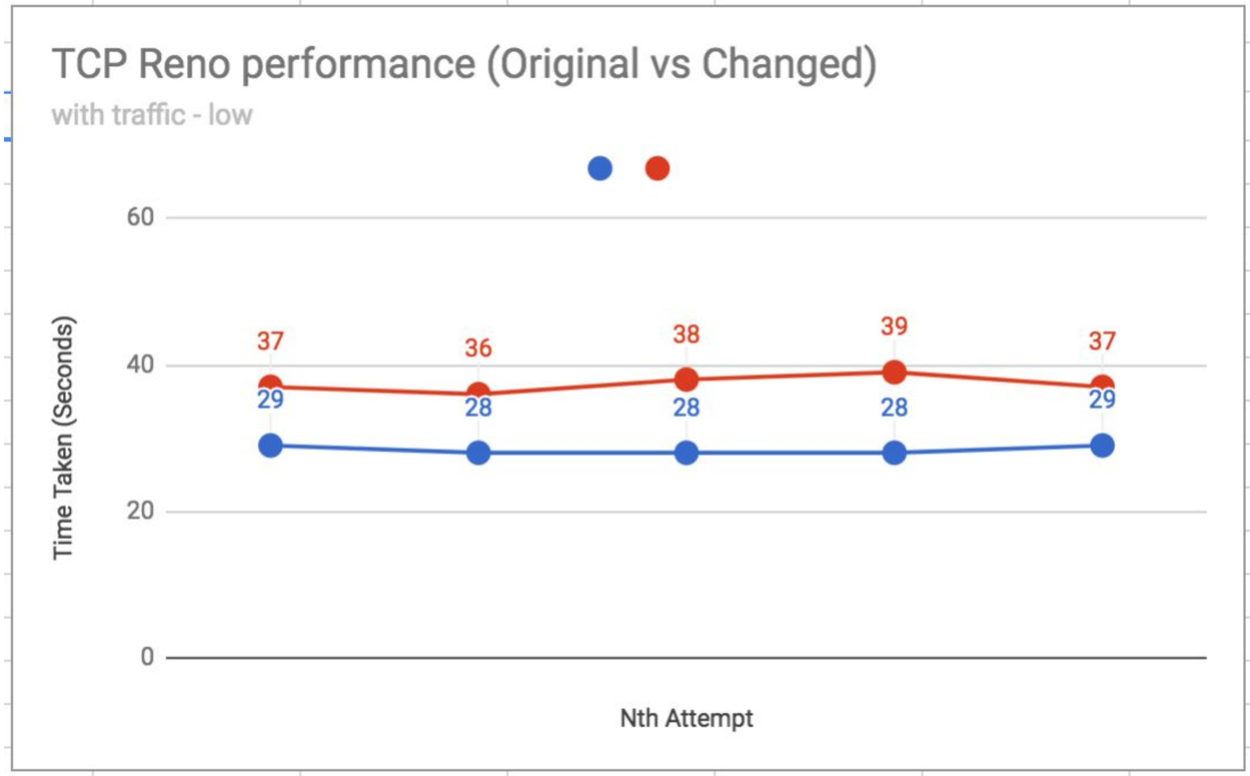
TCP Reno - Time Taken Analysis without any traffic



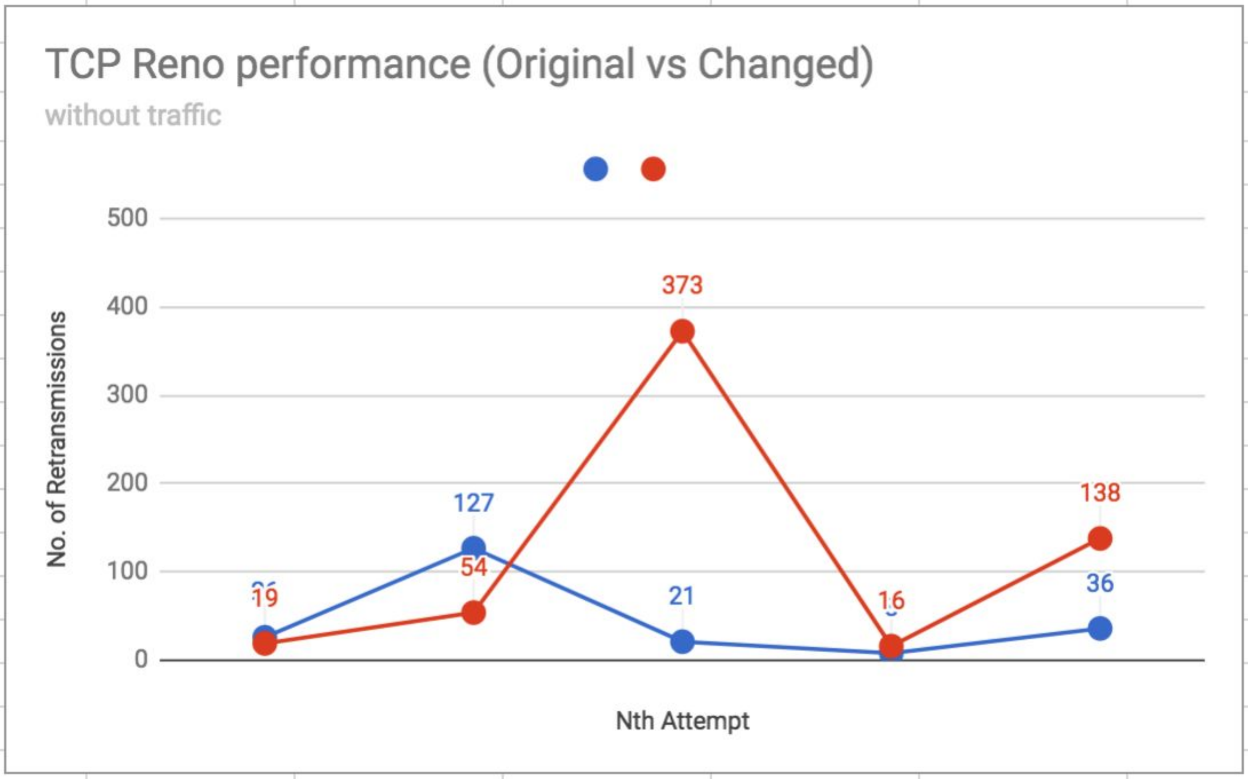
TCP Reno - Time Taken Analysis with high traffic



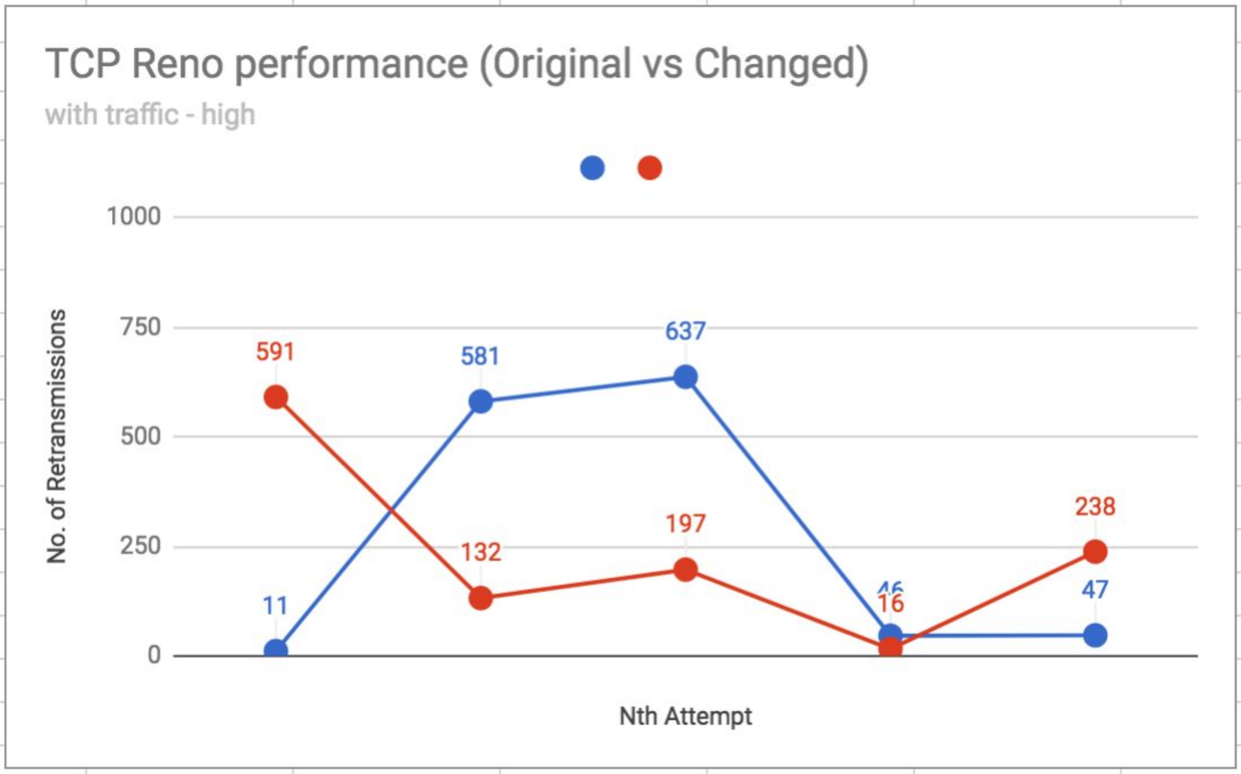
TCP Reno - Time Taken Analysis with low traffic



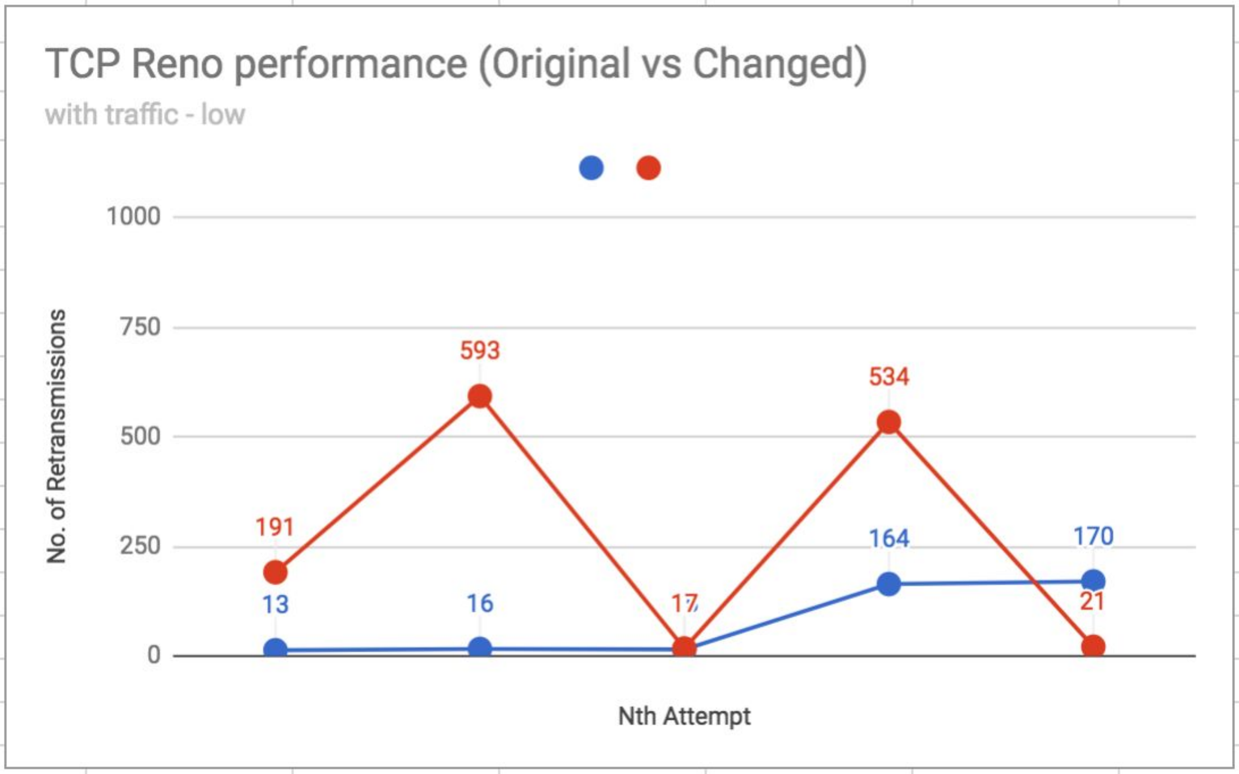
TCP Reno - No. of Retransmitted packets without traffic



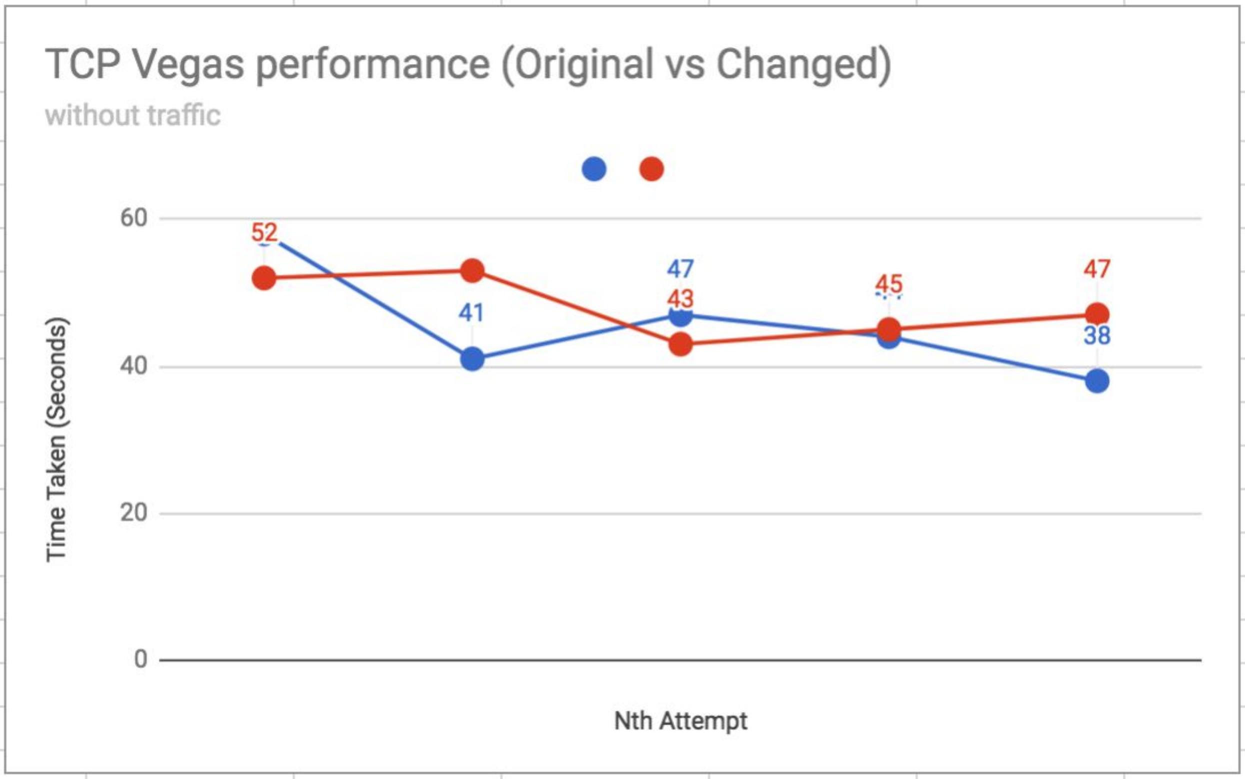
TCP Reno - No. of Retransmitted packets with high traffic



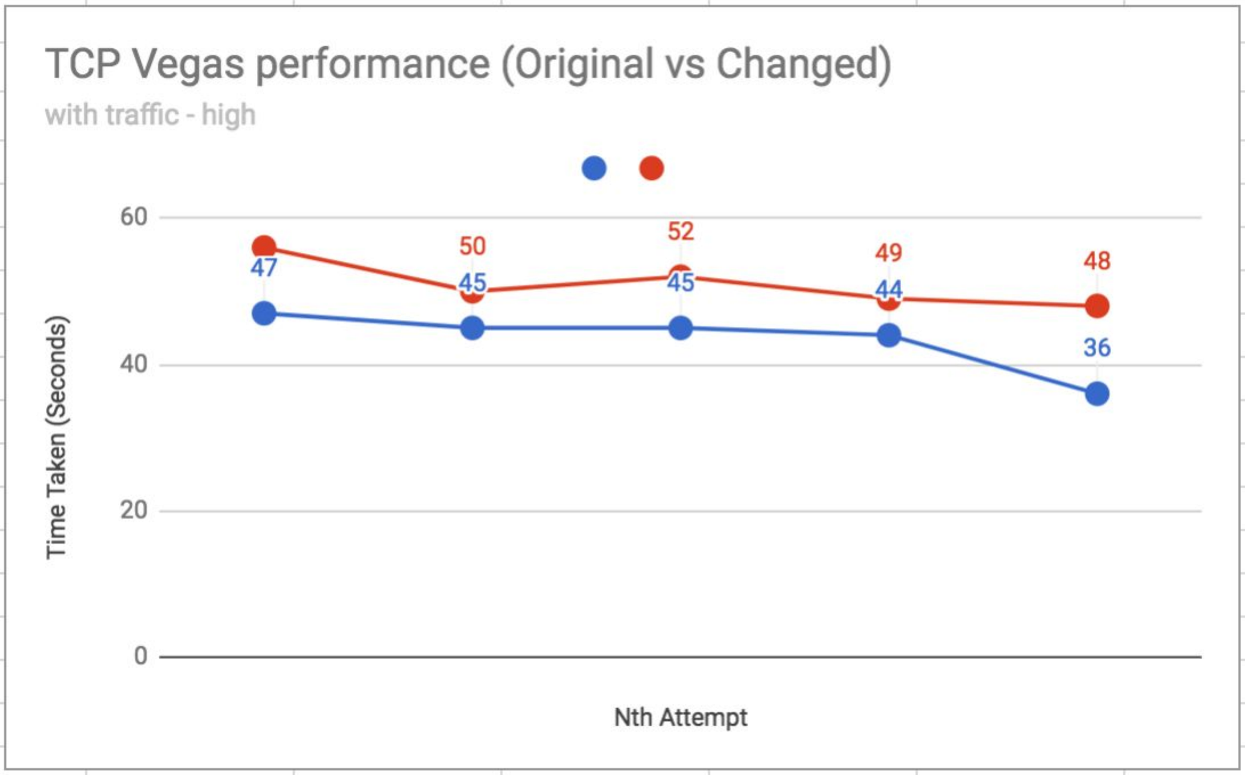
TCP Reno - No. of Retransmitted packets with low traffic



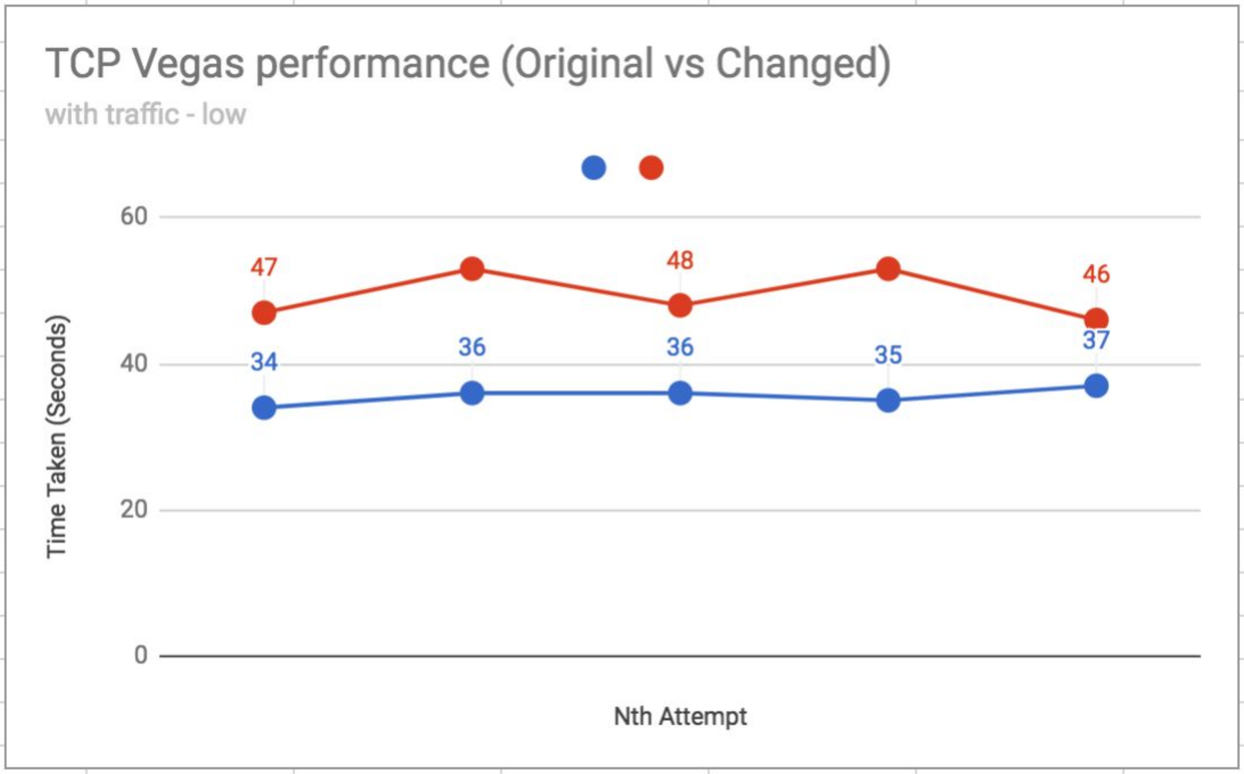
TCP Vegas - Time Taken Analysis without any traffic



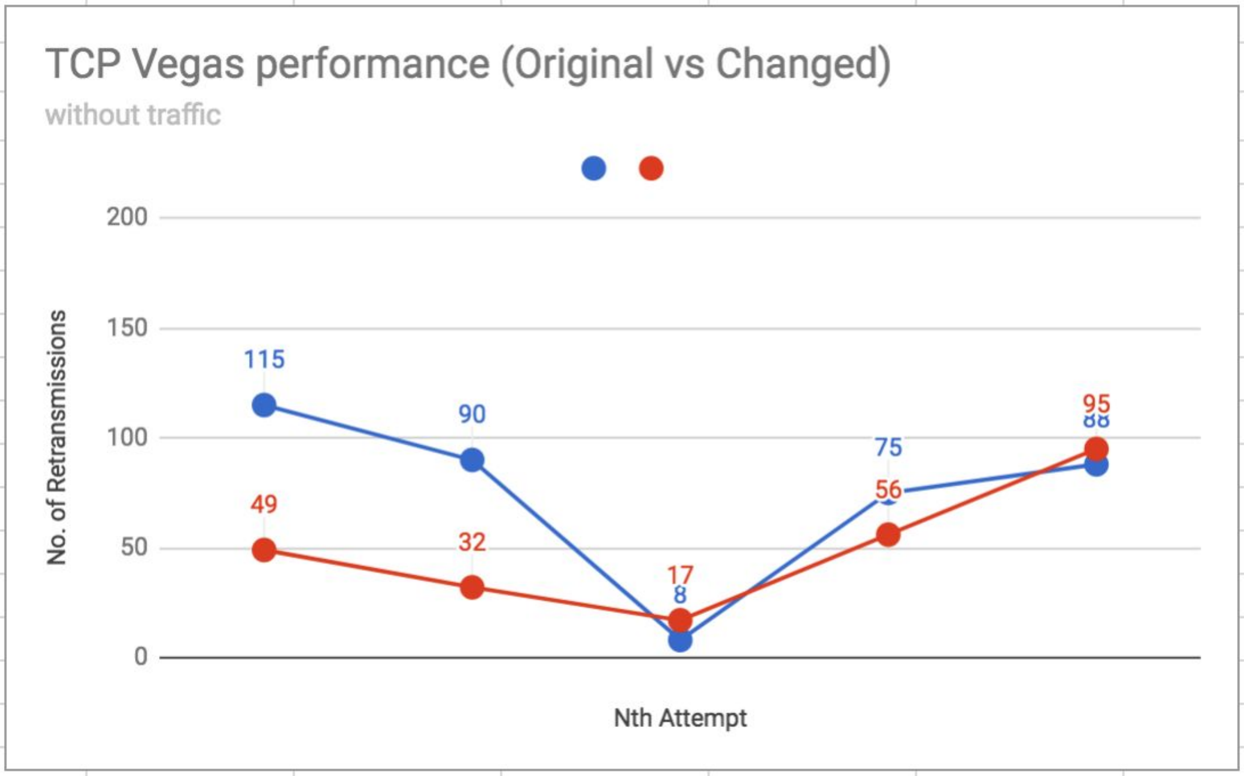
TCP Vegas - Time Taken Analysis with high traffic



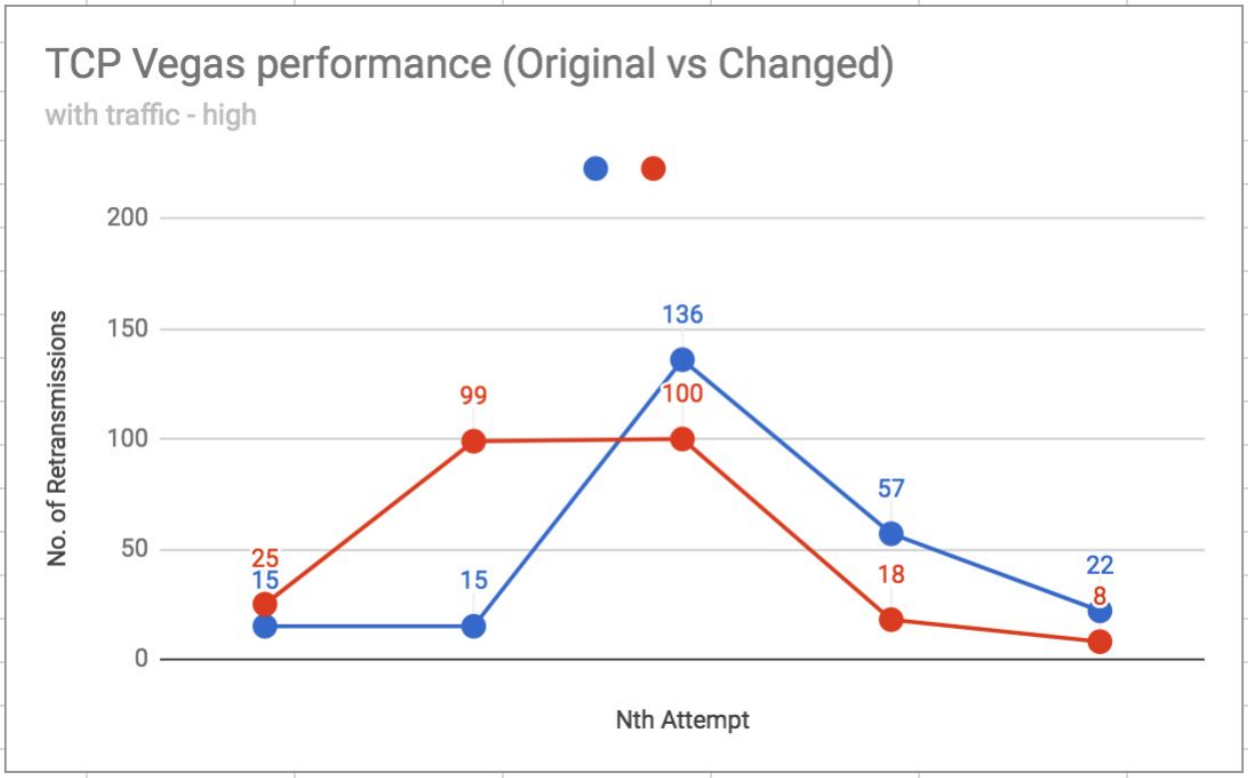
TCP Vegas - Time Taken Analysis with low traffic



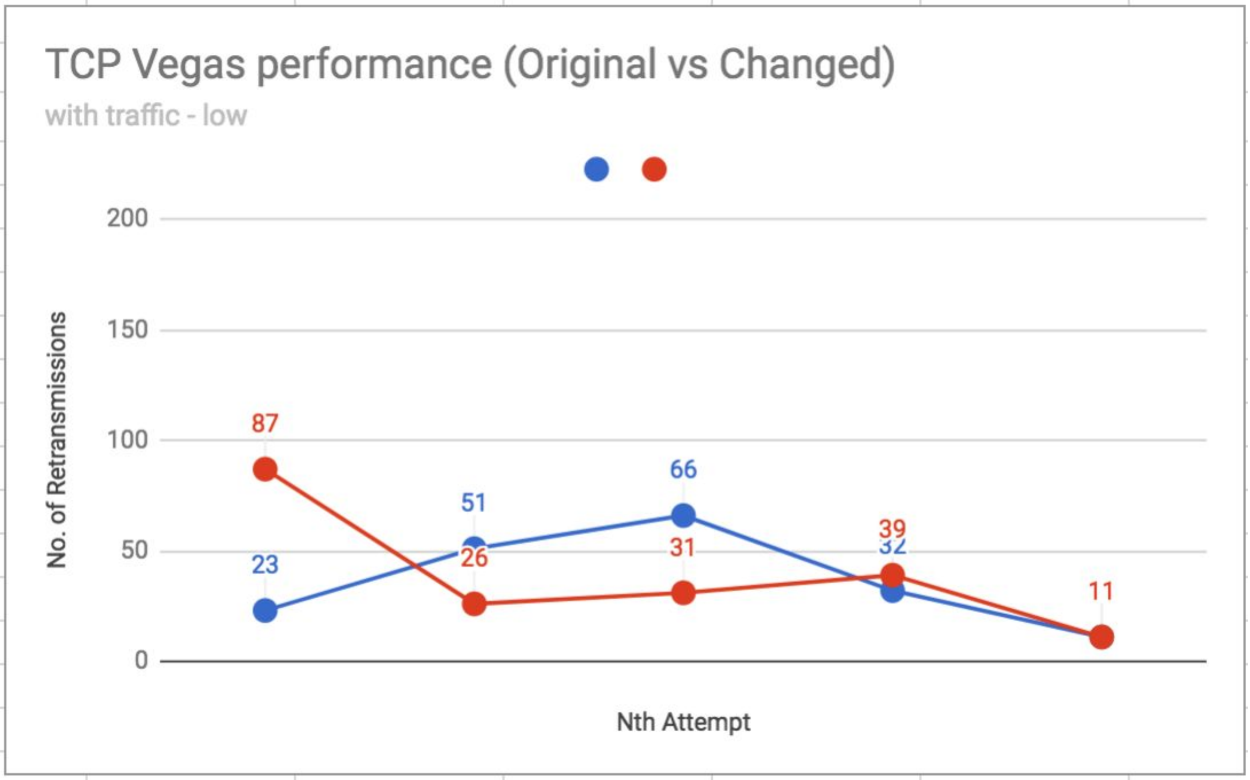
TCP Vegas - No. of Retransmitted packets without traffic



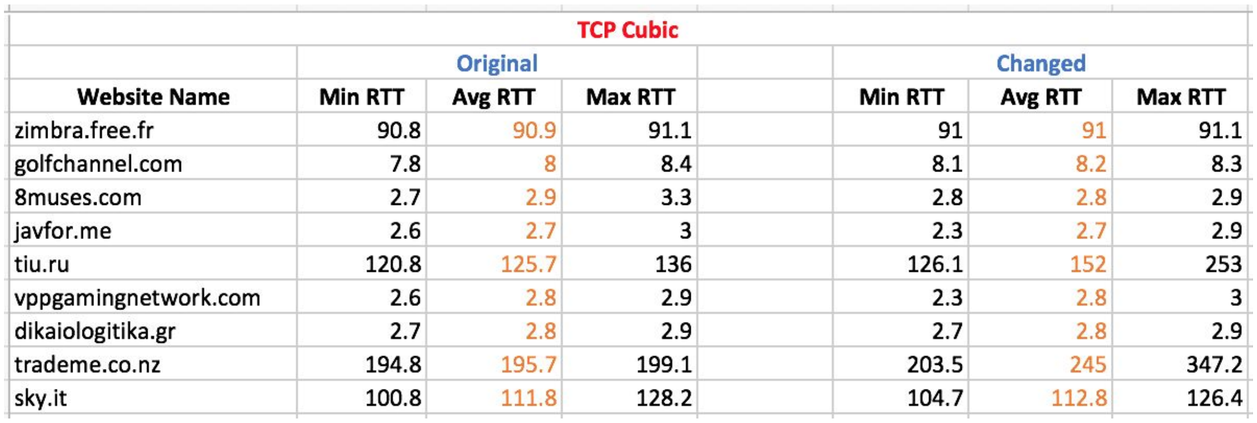
TCP Vegas - No. of Retransmitted packets with high traffic



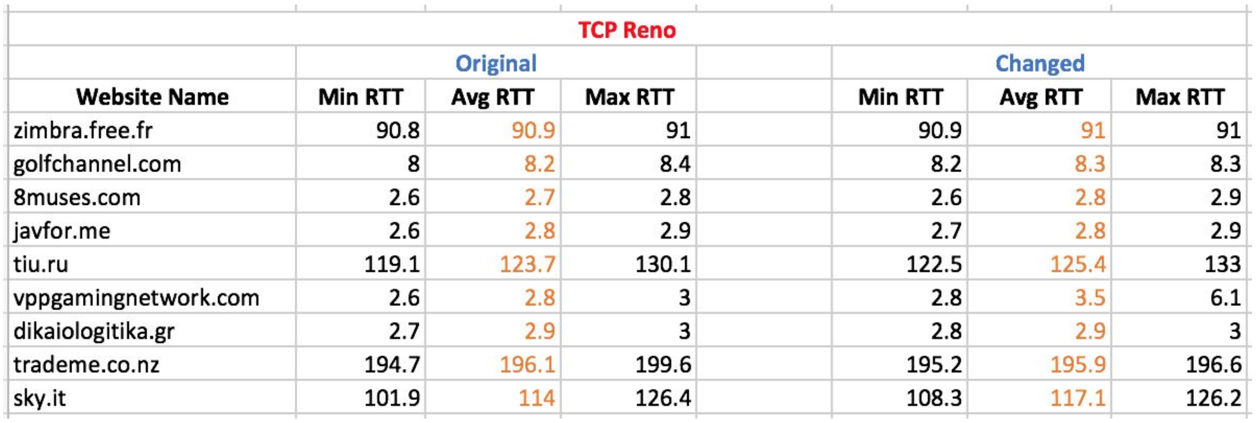
TCP Vegas - No. of Retransmitted packets with low traffic



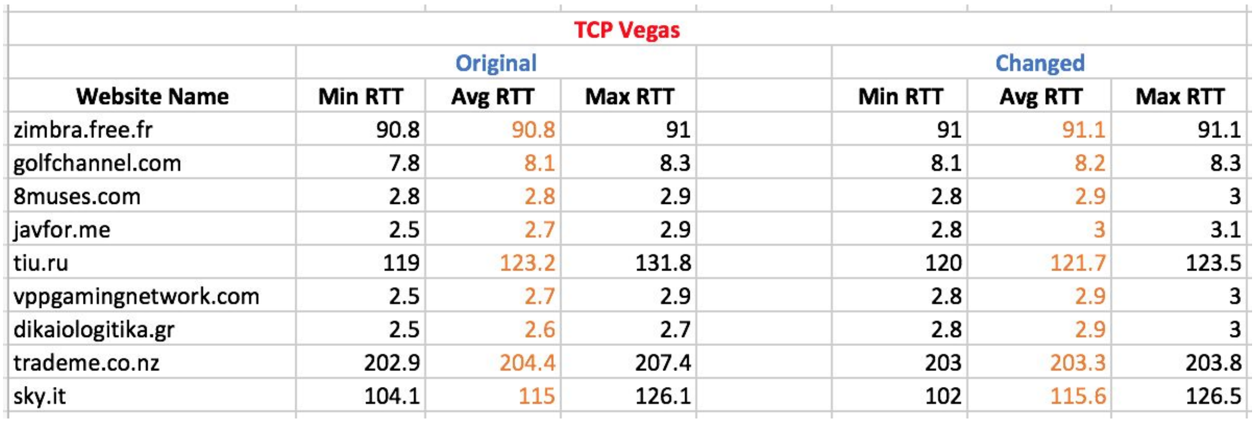
Sanity Check: TCP ping (hping3) is used to hit the following websites numerous times and the Min RTT, Avg RTT, Max RTT are calculated for the original and changed versions of 3 TCP variants with a help of a script and they are tabulated below. Here, we observe that changed TCP versions does not perform drastically and this acts as a sanity test.



			TCP Cubic				
		Original				Changed	
Website Name	**Min RTT**	**Avg RTT**	**Max RTT**		**Min RTT**	**Avg RTT**	**Max RTT**
zimbra.free.fr	90.8	90.9	91.1		91	91	91.1
golfchannel.com	7.8	8	8.4		8.1	8.2	8.3
8muses.com	2.7	2.9	3.3		2.8	2.8	2.9
javfor.me	2.6	2.7	3		2.3	2.7	2.9
tiu.ru	120.8	125.7	136		126.1	152	253
vppgamingnetwork.com	2.6	2.8	2.9		2.3	2.8	3
dikaiologitika.gr	2.7	2.8	2.9		2.7	2.8	2.9
trademe.co.nz	194.8	195.7	199.1		203.5	245	347.2
sky.it	100.8	111.8	128.2		104.7	112.8	126.4



			TCP Reno				
		Original				Changed	
Website Name	**Min RTT**	**Avg RTT**	**Max RTT**		**Min RTT**	**Avg RTT**	**Max RTT**
zimbra.free.fr	90.8	90.9	91		90.9	91	91
golfchannel.com	8	8.2	8.4		8.2	8.3	8.3
8muses.com	2.6	2.7	2.8		2.6	2.8	2.9
javfor.me	2.6	2.8	2.9		2.7	2.8	2.9
tiu.ru	119.1	123.7	130.1		122.5	125.4	133
vppgamingnetwork.com	2.6	2.8	3		2.8	3.5	6.1
dikaiologitika.gr	2.7	2.9	3		2.8	2.9	3
trademe.co.nz	194.7	196.1	199.6		195.2	195.9	196.6
sky.it	101.9	114	126.4		108.3	117.1	126.2



			TCP Vegas				
		Original				Changed	
Website Name	**Min RTT**	**Avg RTT**	**Max RTT**		**Min RTT**	**Avg RTT**	**Max RTT**
zimbra.free.fr	90.8	90.8	91		91	91.1	91.1
golfchannel.com	7.8	8.1	8.3		8.1	8.2	8.3
8muses.com	2.8	2.8	2.9		2.8	2.9	3
javfor.me	2.5	2.7	2.9		2.8	3	3.1
tiu.ru	119	123.2	131.8		120	121.7	123.5
vppgamingnetwork.com	2.5	2.7	2.9		2.8	2.9	3
dikaiologitika.gr	2.5	2.6	2.7		2.8	2.9	3
trademe.co.nz	202.9	204.4	207.4		203	203.3	203.8
sky.it	104.1	115	126.1		102	115.6	126.5

Conclusions

Thus from the pseudo code mentioned in the solution section, we can conclude that the RTT estimator gives more weight to the recent RTT’s than the older RTT’s and also helps in increasing or decreasing the RTT fastly unlike the original TCP version. Also, from the graphs we can see that the modified versions retransmits packets more when compared to the original version and this is because of the tighter bound on the RTO. Theoretically, the modified versions perform better in lossy conditions as the sender does not wait for too long to retransmit in case of packet loss. However changing only the rtt might not improve the performance, changing the operation of cwin of TCP might also be needed according to RTT estimator changes for it to show improvements practically.

Work done for the extra 30%: All the results are presented in the form of graphs and tables depicting meaningful performance analysis between different version of TCP. Apart from the normal sanity results, we performed an exhaustive analysis comparing the versions by artificially inducing traffic and congestion into the network and also digged deep to improve the TCP performance by introducing rtt_error along with prediction to reduce the number of retransmissions.

Future Work

Tight estimation of RTT might alone not help the TCP to perform better, changing the operation of how TCP retransmits in correspondence with RTT might improve the TCP throughput.

Code Repository

https://github.com/swetheendra84/CSE534Project

References

https://en.wikipedia.org/wiki/Transmission_Control_Protocol
https://en.wikipedia.org/wiki/TCP_Vegas
https://en.wikipedia.org/wiki/CUBIC_TCP
https://en.wikipedia.org/wiki/Talk:TCP_congestion-avoidance_algorithm
https://help.ubuntu.com/community/Kernel/Compile
https://www.tcpdump.org/tcpdump_man.html
https://www.wireshark.org/docs/man-pages/tshark.html
https://www.wireshark.org/docs/man-pages/
https://elixir.bootlin.com/linux/latest/source
https://linux.die.net/man/8/hping3
https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6006098