Performance of high-speed TCP Protocols over NS-2 TCP Linux

Masters Project Final Report

Author: Sumanth Gelle

Email: sgelle@cs.odu.edu

Project Advisor: Dr. Michele Weigle

Email: mweigle@cs.odu.edu

Project Presentation Date: February 11, 2008

Department of Computer Science

Old Dominion University

INDEX

Abstract

TCP, the most widely-used protocol on Internet, has a major problem in that its congestion control algorithm does not allow flows to obtain full available bandwidth on fast long-distance links. To address this issue, many high-speed TCP protocols have been proposed. Currently, there is no consensus on which high-speed TCP protocol should be used, so users might be employing different high-speed TCP protocols to transfer their data on the same shared link. So, there is a need to understand and analyze how the various high-speed TCP protocols share the bandwidth among them. Previous work has used the NS-2 network simulator to analyze the fairness in sharing bandwidth of high-speed TCP protocols. The problem with using current versions of NS-2 to analyze the behavior of high-speed TCP protocols is that they have a TCP implementation which is based on the 1992 BSD kernel. This implementation differs widely from the current implementations of TCP in most widely-used operating systems, like Linux. Since most of the high-speed TCP protocols were first implemented in Linux, we want to study the fairness among these high speed TCP protocols using implementations closer to that in Linux. The purpose of this project is to use NS-2 with TCP-Linux modifications to analyze the fairness among high-speed TCP protocols and compare the results to those obtained using the high-speed TCP implementations based on NS-2's default TCP.

1. Introduction

In the previous study [5, 6] NS-2 was used to analyze the fairness among high-speed TCP protocols in sharing bandwidth. FAST TCP, H-TCP [2], CUBIC [3], BIC-TCP [4], HS-TCP [7], and Scalable TCP [8] were the high-speed TCPs used in the experiment. Two flows using two different high-speed TCP protocols were run, such that the second flow started 50 seconds after the first flow started, thus, the first flow attained full bandwidth by the time the second flow started, as this is the most likely situation in the real world. The same experiment was repeated each time with a different combination of flows.

The motivation for this project is to use NS-2 TCP Linux [1] instead of the standard NS-2 TCP Agent to study the fairness among the protocols. The implementation of TCP modules present in current NS-2 is based on the 1992 BSD kernel's TCP modules, and this implementation differs widely from the TCP implementation present in current operating systems like Linux. Further, all the high-speed TCP protocols were first designed on Linux platform, so the results obtained in the previous study may not be realistic. NS-2 TCP Linux is a new implementation of TCP in NS-2 whose implementation is based on Linux 2.6 TCP. Future versions of NS-2 will include TCP-Linux Agents. In this work, we study BIC-TCP, HS-TCP, Scalable TCP, CUBIC, and H-TCP. FAST TCP is not included as there is no open source version of the protocol available.

2. Background

Here we present an overview of the high-speed TCPs evaluated in this study and a brief explanation of TCP-Linux.

High-speed TCP protocols can be broadly categorized into two categories based on how they sense congestion in the network:

- Loss -Based Protocols
- Delay-Based Protocols

Loss-based protocols use packet loss in the network to detect congestion where as delay-based protocols use queuing delays at the routers, in addition to loss, to detect congestion. FAST TCP is the only high-speed TCP protocol which is delay based, but its implementation is not present in Linux, so all the protocols we have considered in our experiments are loss-based protocols.

2.1 HS-TCP

High-Speed TCP (HS-TCP) [7] uses the value of the previous congestion window to compute its new congestion window value. HS-TCP behaves like standard TCP when the congestion window (cwnd) is below a threshold called Low_Window. Above Low_Window HS-TCP acts more aggressively in attaining bandwidth by increasing its congestion window size aggressively.

On each arrival of a new acknowledgement, HS-TCP increases its congestion window by the following:

cwnd ← cwnd + α / cwnd.

When congestion is detected through packet loss, the congestion window is decremented as follows:

cwnd ← cwnd * (1 – β).

For congestion window values less than Low_Window, the values of α and β are 1 and 0.5 respectively, as with Standard TCP. When the congestion window is above Low_Window, the values of α and β are determined by a lookup table.

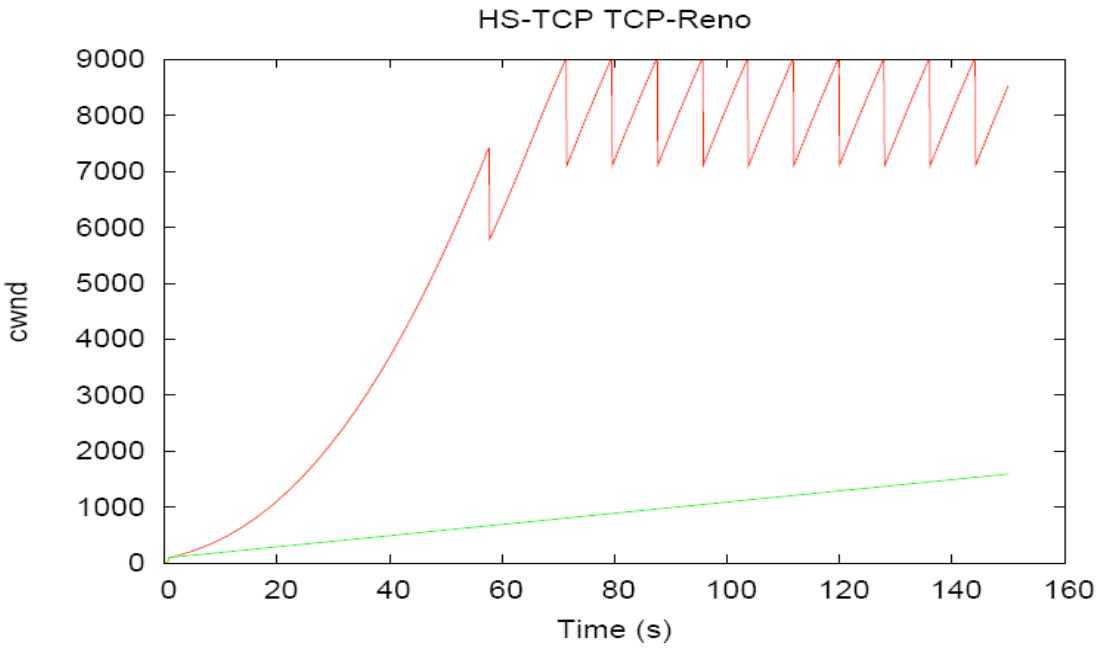

Figure 1. HS-TCP vs. TCP-Reno

Figure 1 shows the congestion window of a single HS-TCP flow and a single TCP Reno (Standard TCP) flow, run separately. The darker (red) line represents HS-TCP, and the lighter (green) line represents TCP Reno. From the graph we can see that HS-TCP is able to increment its congestion window very quickly so that it can attain the whole of available bandwidth.

2.2 Scalable TCP

Scalable TCP [8] is an Additive Increase Multiplicative Decrease (AIMD) protocol, i.e. it increases its congestion window linearly and decreases it congestion window multiplicatively. Scalable TCP is similar to HS-TCP, but it has fixed values for α and β.

When a new acknowledgement is received, Scalable TCP increments the cwnd as follows:

cwnd ← cwnd + 0.01

When congestion is detected, the congestion window is decremented as follows:

cwnd ← cwnd * 0.875

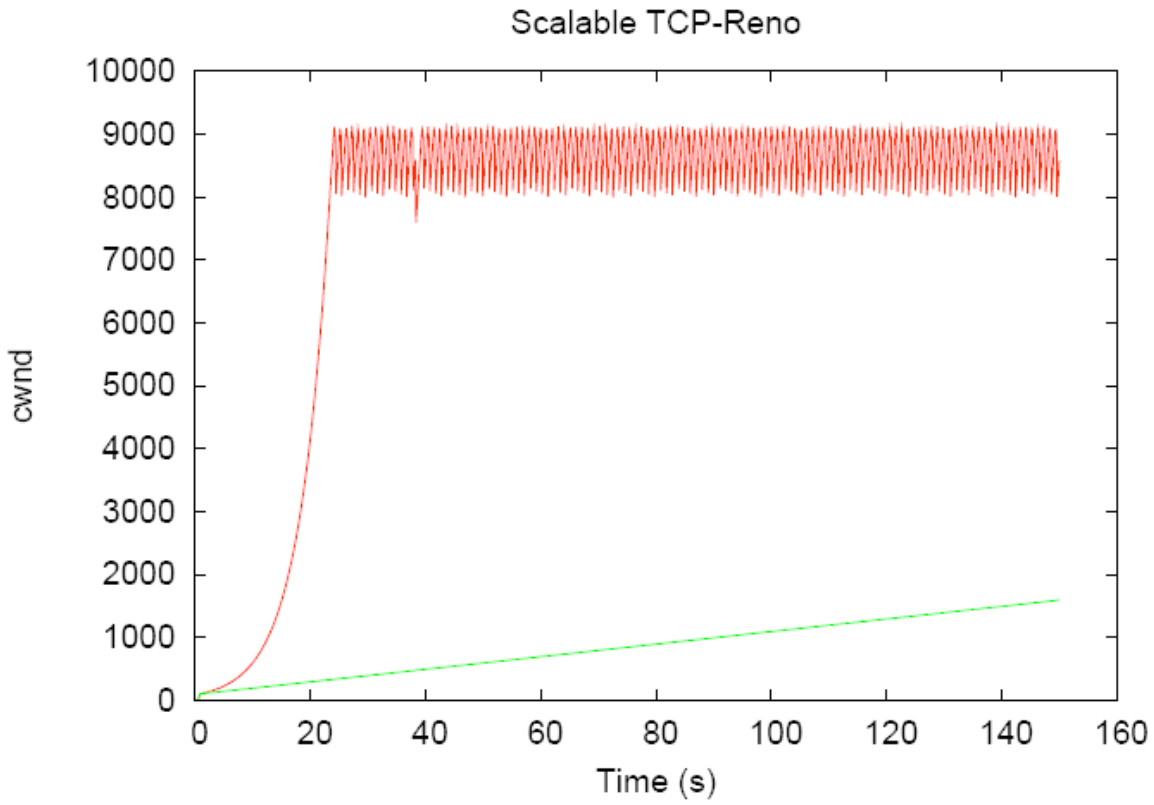

Figure 2. Scalable-TCP vs. TCP-Reno

Figure 2, we clearly see that Scalable-TCP increases its congestion window to a maximum size with which it can utilize the whole bandwidth of the network. We can also see that Scalable-TCP is more aggressive than HS-TCP.

2.3 BIC-TCP

BIC-TCP [4] makes an estimate of the available network bandwidth and sets a target window size, called maximum window. It maintains the current congestion window size in a variable called minimum window. BIC-TCP employs a binary search function to increment the minimum window to the midpoint of minimum window and

maximum window when a new acknowledgement is received. But initially when the difference between the minimum window and the maximum window is large, BIC-TCP employs additive increase. When the difference is below a threshold value, BIC-TCP increments the congestion window size using binary search. Fast convergence, i.e. binary search increase combined with a multiplicative decrease, is used to converge quickly and achieve a fair share with other protocols. Figure 3 shows the performance of BIC-TCP in comparison with TCP-Reno. We can see that BIC-TCP represented by the darker (red) color was able to utilize the full bandwidth by increasing the congestion window accordingly.

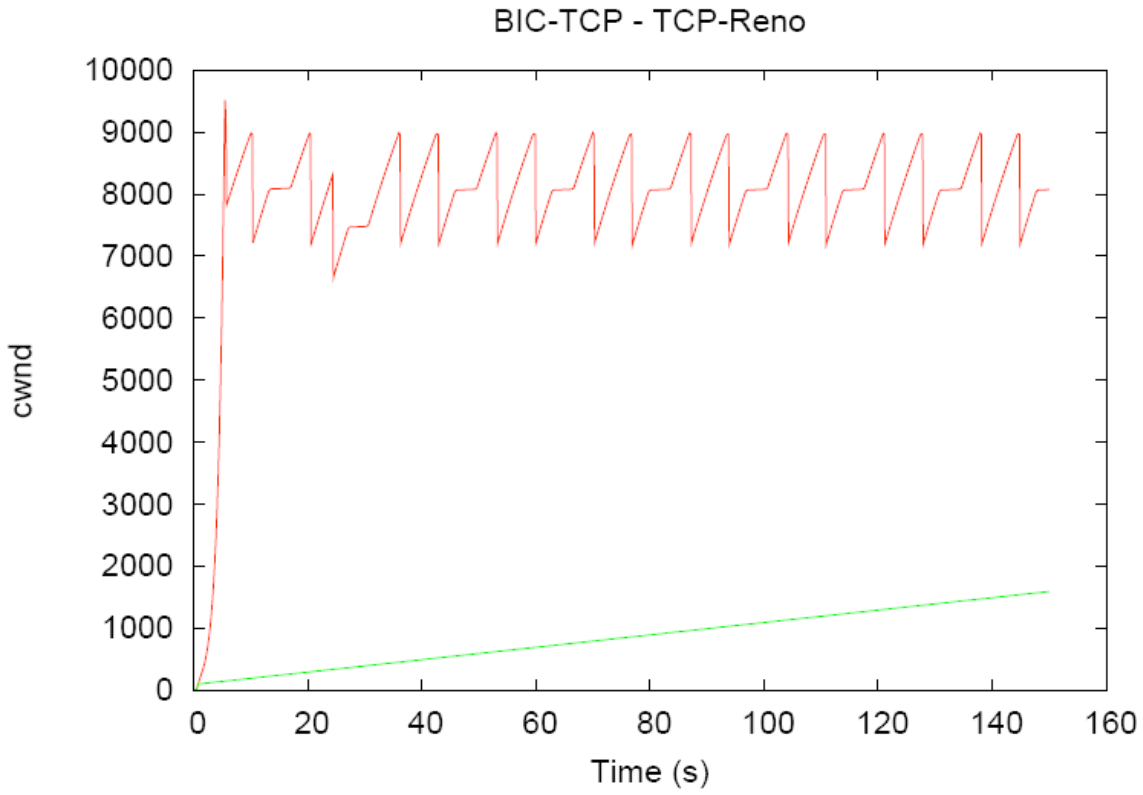

Figure 3. BIC-TCP vs. TCP-Reno

2.4 CUBIC

The main design goal of CUBIC [3] was to improve upon BIC-TCP by making it less aggressive. CUBIC uses the time delay between packet drops to adjust its congestion window value. It uses a cubic function to increment its congestion window. Figure 4

shows the performance of CUBIC in comparison to TCP-Reno. We can see the cubic increment of the congestion window.

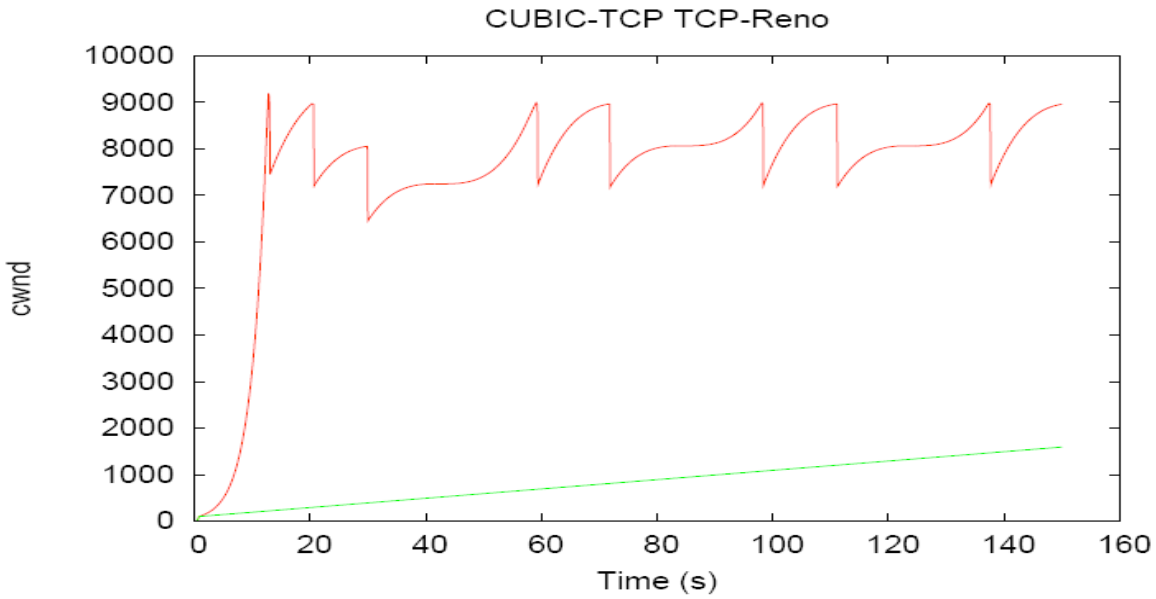

Figure 4. CUBIC-TCP vs. TCP-Reno

2.5 H-TCP

Hamilton TCP (H-TCP) [2] uses the time between packet drops to adjust the congestion window. It uses the following function to decrement its congestion window:

$$cwnd \leftarrow cwnd * \beta.$$

where β is such that $0.5 < \beta < 0.8$.

H-TCP adopts an adaptive back off strategy to decrement the congestion window when it detects congestion. It uses the ratio of the minimum-observed RTT to the maximum-observed RTT to compute the new congestion window value. Figure 5 shows the performance of H-TCP versus TCP-Reno.

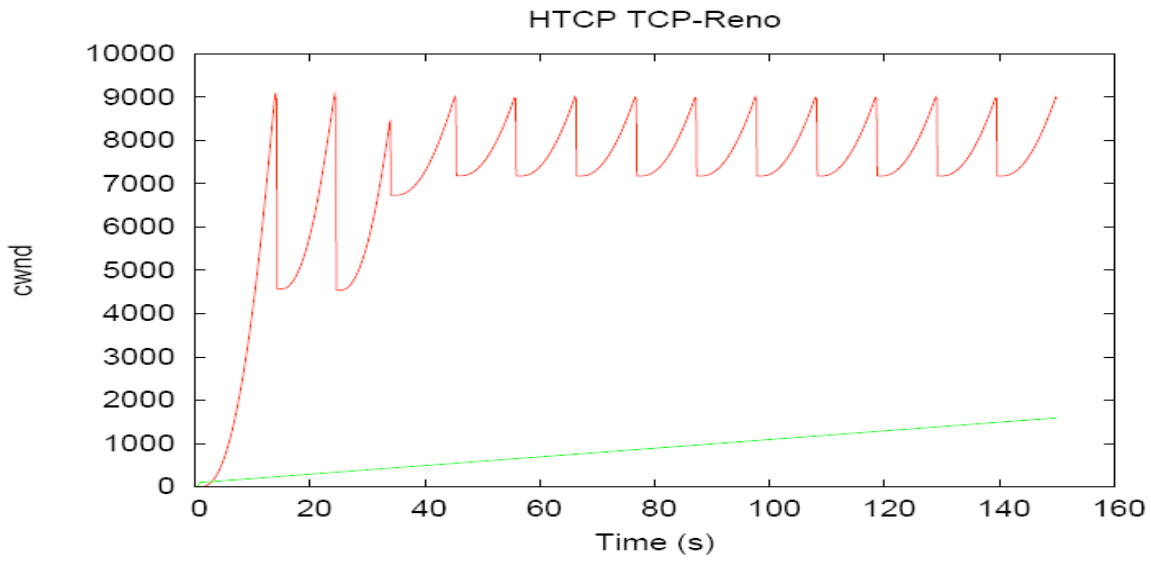

Figure 5. HTCP vs. TCP-Reno

In Figure 5 we can clearly see the adoptive back off strategy employed by HTCP.

2.6 NS-2 TCP Linux

Recently, a module, NS-2 TCP Linux [1], was developed for NS-2 to provide Linux-based TCP Agents. Linux TCP has several differences from the TCP implementation present in current NS-2. Linux-TCP's SACK processing can recover even from the loss of retransmitted packet where as NS-2's default TCP times out. Linux-TCP does not use delayed acknowledgements initially, so that the congestion window can grow faster. Linux-TCP uses the D-SACK information to rearrange out of order packets. Linux-TCP uses an improved packet loss detection algorithm called score-boarding. All these improvements in Linux-TCP might cause the results obtained by using NS-2 TCP Linux to differ from those obtained using NS-2's default TCP.

3. Methodology

All the experiments (except those involving FAST TCP) used to study the fairness among high-speed TCP protocols used in the previous study [5, 6] were rerun using NS-2 TCP Linux.

The topology shown in Figure 6 was used to study the inter-protocol fairness.

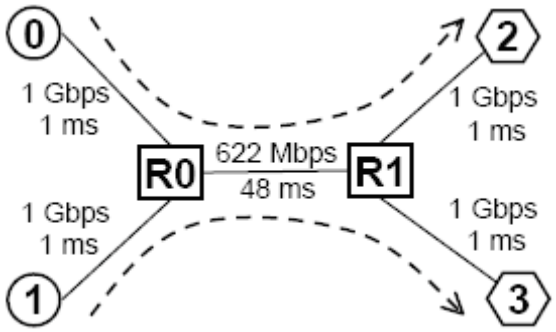

Figure 6. Network Topology (from [5])

It consists of two senders and receivers with two routers in between connected by a 622 Mbps channel and with a propagation delay of 48 ms. The senders and the receivers are connected to routers through a 1 Gbps channel with 1 ms propagation delay. The two flows run from node 0 to node 2 and from node 1 to node 3. The channel provides the senders with a Round Trip Time (RTT) of 100 ms.

The experiment is run once for each combination of H-TCP, CUBIC, BIC-TCP, HS-TCP, and Scalable TCP. As in the previous study [5, 6] flow 2 is started 50 seconds after flow 1. The experiments are run for 500 seconds and the average throughput for each flow is measured between the interval [250, 500] seconds. The results obtained are compared to the earlier results.

The asymmetry metric is defined as the ratio of difference of average throughputs of the flows and the sum of the average throughputs of the flows. The asymmetry metric is $(x_1 - x_2) / (x_1 + x_2)$ where x_i is the average throughput obtained for flow i. The asymmetry metric is used as a measure of fairness among protocols. An asymmetry value closer to 0 implies more fairness, an asymmetry value closer to 1 implies that flow 1 dominates, and an asymmetry value closer to -1 implies that flow 2 dominates.

4. Results

The asymmetry value for each combination of high-speed TCP protocols is computed and then compared to the asymmetry value for the same experiment when run on NS-2's default TCP. Table 1 shows the comparison between asymmetry values obtained when using NS-2's default TCP and when using NS-2 TCP Linux:

Experiment(500s)	Asymmetry(Default)	Asymmetry (Linux)	Difference
BIC-BIC	0.28	0.001	0.279
BIC-CUBIC	0.76	0.815	-0.055
BIC-HS	0.492	0.307	0.185
BIC-HTCP	0.693	0.371	0.322
BIC-Scalable	-0.896	-0.996	0.1
CUBIC-BIC	-0.633	-0.765	0.132
CUBIC-CUBIC	0.08	0.051	0.029
CUBIC-HS	-0.481	-0.436	-0.045
CUBIC-HTCP	-0.34	-0.322	-0.018
CUBIC-Scalable	-0.978	-0.996	0018
HS-BIC	0.425	-0.065	0.49
HS-CUBIC	0.599	0.596	0.003
HS-HS	0.074	0.082	-0.008
HS-HTCP	0.247	0.249	-0.002
HS-Scalable	-0.996	-0.997	0.001
HTCP-BIC	0.06	-0.222	0.282
HTCP-CUBIC	0.729	0.801	-0.072
HTCP-HS	-0.234	-0.221	-0.013
HTCP-HTCP	0	0.001	-0.001
HTCP-Scalable	-0.994	-0.997	0.003
Scalable-BIC	0.882	0.998	-0.116
Scalable-CUBIC	0.987	0.995	-0.0008
Scalable-HS	0.996	0.997	-0.001
Scalable-HTCP	0.993	0.997	-0.004
Scalable-Scalable	0.975	0.878	0.097

Table 1. Comparison of Asymmetry values of NS-2's default TCP and NS-2 TCP Linux

In Table 1, the first column represents the experiment run, the second column represents the asymmetry value obtained for the experiment when NS-2's default TCP was used, and the third column represents the asymmetry value obtained when NS-2 TCP Linux was used, the fourth column shows the difference between the two asymmetry values. All the rows with a significant difference in asymmetry values are highlighted. In Figures 7 – 10 we show throughput for the two flows for selected experiments.

In all of these figures flow 1 is represented using darker (red) color and flow 2 is represented using lighter (green) color. The following are some of the cases where there is not much difference in asymmetry (A) values.

HS-Scalable

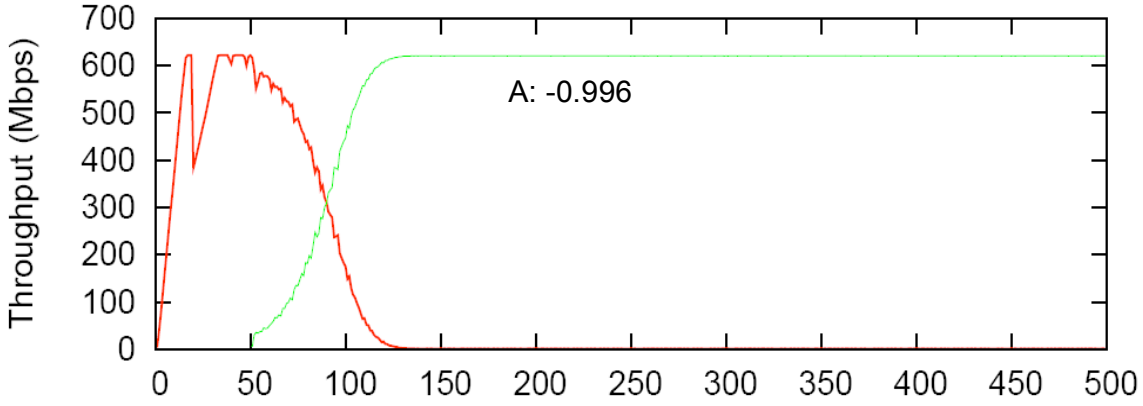

Figure 7a. NS-2 Default TCP

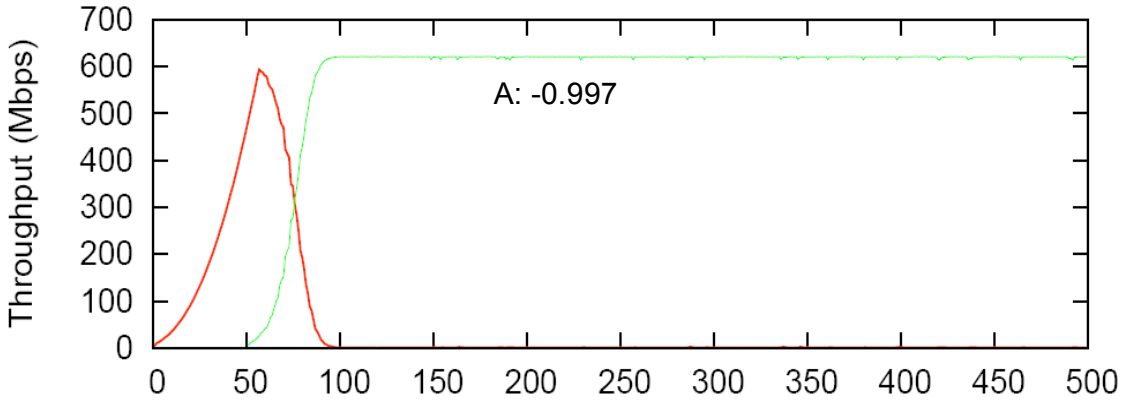

Figure 7b. NS-2 TCP Linux

CUBIC-Scalable

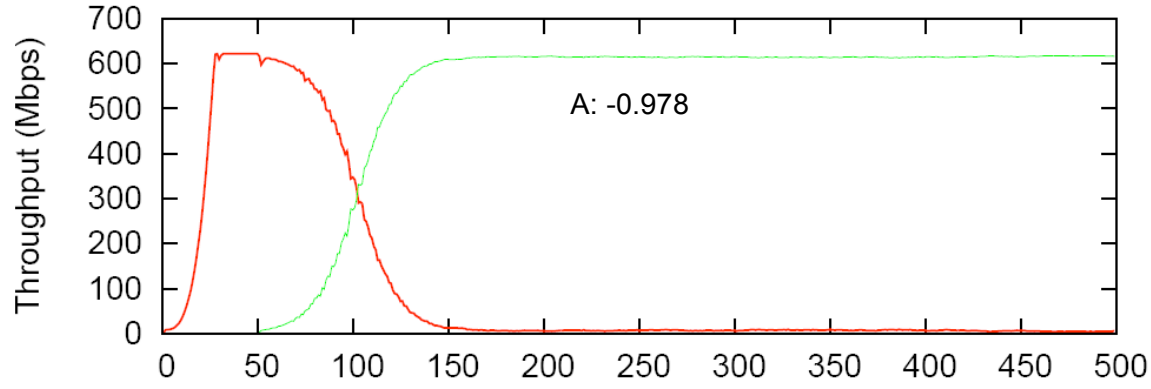

Figure 8a. NS-2 Default TCP

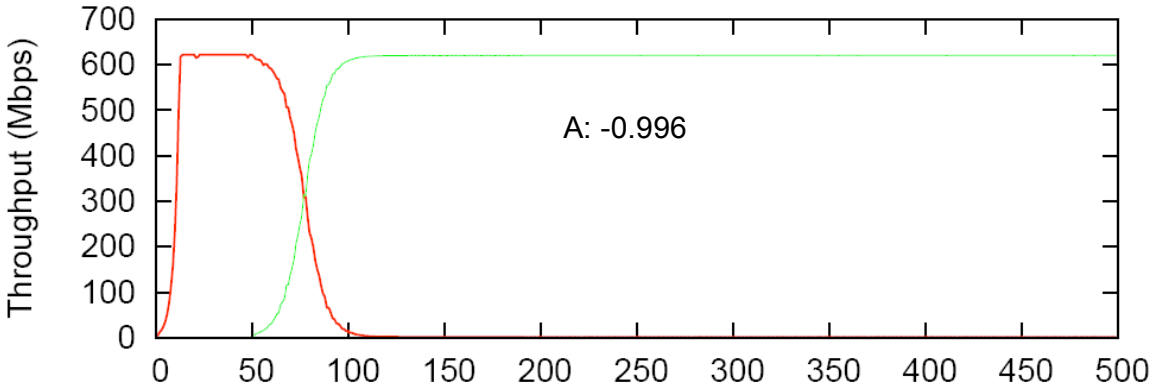

Figure 8b. NS-2 TCP Linux

From the Figures 7- 8 we can observe that all the experiments which did not involve BIC-TCP have similar asymmetry values which justifies the NS-2‘s implementation of these high-speed protocols.

The following are the graphs of some experiments for which there is significant difference in asymmetry values.

BIC-BIC

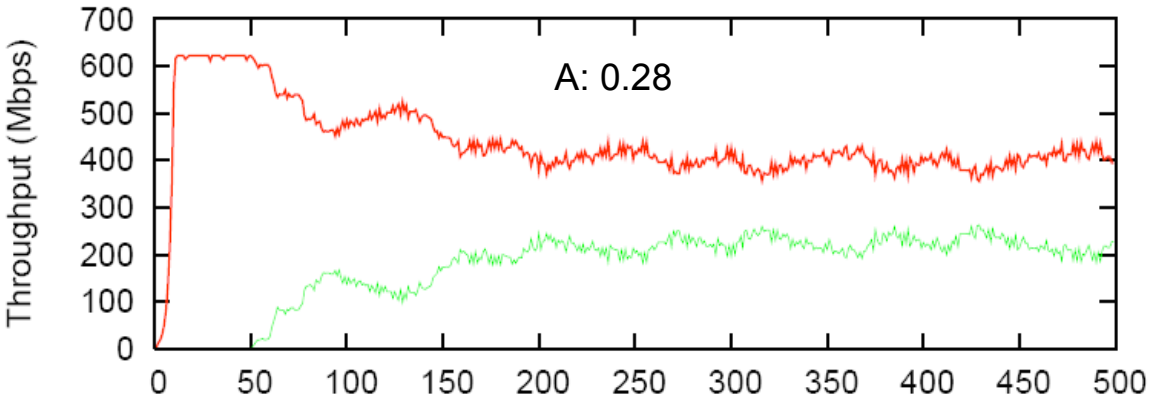

Figure 9a. NS-2 Default TCP

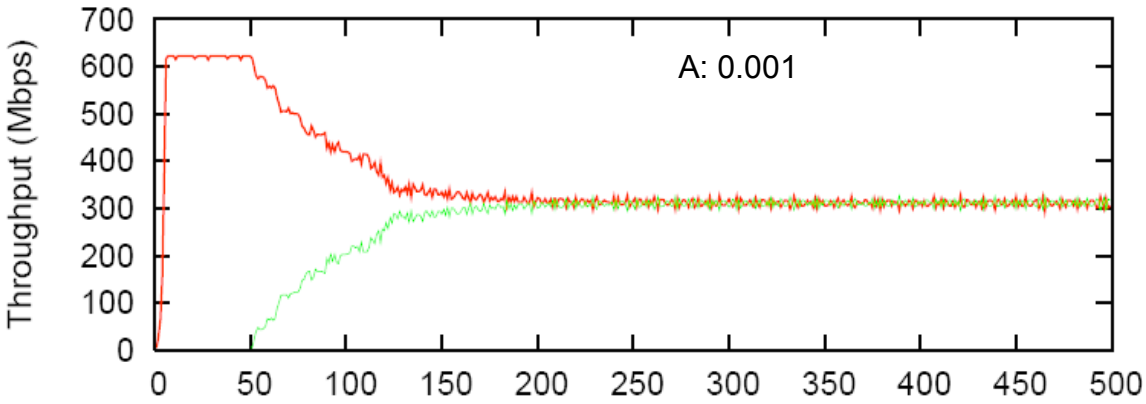

Figure 9b. NS-2 TCP Linux

HS-BIC

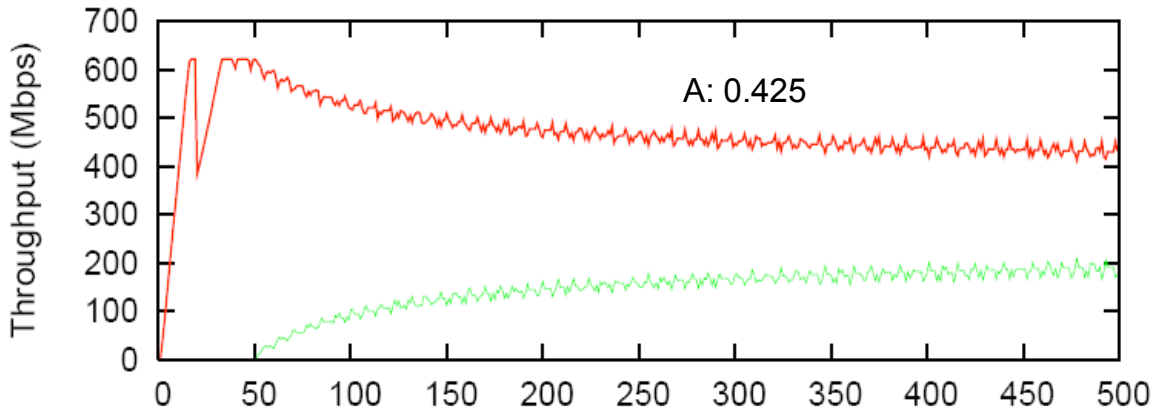

Figure 10a. NS-2 Default TCP

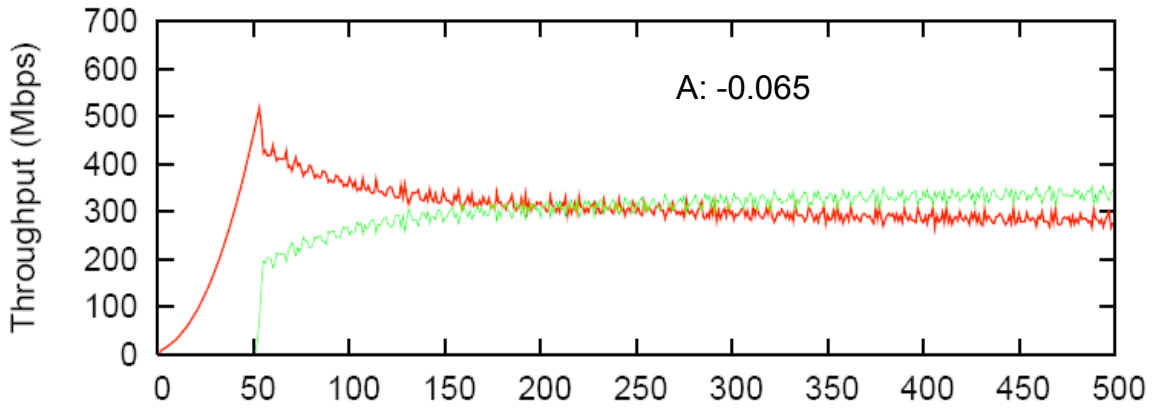

Figure 10b. NS-2 TCP Linux

From the Figures 9- 10 we can observe that all the experiments in which BIC-TCP was involved showed significant difference in the asymmetry value more over NS-2 TCP Linux's BIC-TCP was much fair in sharing the bandwidth compared to NS-2 Default TCP .

5. Conclusions

All the experiments used to study the fairness among high-speed TCP protocols done previously [5, 6] were re-run on NS-2 TCP Linux and the asymmetry values are computed and compared to the ones obtained previously. From the results obtained though these experiments we could conclude that the NS-2's implementation of all the

high-speed TCP protocols except BIC is valid. There was significant improvement in fairness for NS-2 TCP Linux's BIC-TCP over NS-2 default TCP's implementation. The future work for this project includes analyzing the reasons which caused the improvement in fairness for BIC protocol.

References

[1] David X. Wei, Pei Cao. "NS-2 TCP-Linux: An NS-2 TCP Implementation with Congestion Control Algorithms from Linux". In the proceedings of the workshop on NS-2, Oct, 2006.

[2] D. Leith, R. Shorten. "H-TCP: TCP for high-speed and long-distance networks" In Proceedings of PFLDnet, Feb 2004.

[3] Injong Rhee and Lisong Xu. "CUBIC: A New TCP-Friendly High-Speed TCP Variant". In Proceedings of PFLDnet 2005.

[4] Lisong Xu, Khaled Harfoush, and Injong Rhee1. "Binary Increase Congestion Control for Fast, Long Distance Networks". In the proceedings of INFOCOM 2004.

[5] Michele C.Weigle, Pankaj Sharma, and Jeese R. Freeman IV. "Performance of Competing High-Speed TCP Flows". In the Proceedings of NETWORKING, May 2006.

[6] Pankaj Sharma, Michele C.Weigle. "Performance Analysis of High-Speed Transport Control Protocols". A Thesis Presented to the Graduate School of Clemson University, August 2006

[7] S.Floyd. "High-Speed TCP for Large Congestion Windows". IETF RFC 3649, December 2003.

[8] Tom Kelly. "Scalable TCP: Improving Performance in High-Speed Wide Area Networks". Computer Communication Review, April 2003

Appendix

The following is the modified script which uses the NS-2 TCP Linux with the changes indicated with bold text:

```
set begintime  [clock second]

# simulation end time
set endtime [lindex $argv 0]

# high-speed flow 1 type, flow 2 type
set hstcp_type(0) [lindex $argv 1];   # HS, Scalable, HTCP, BIC, CUBIC
set hstcp_type(1) [lindex $argv 2];   # HS, Scalable, HTCP, BIC, CUBIC
set hstcpflows_num 2
set hstcpflows_type 0
set alpha [lindex $argv 3]

# flow 2 start time
set flow2_start [lindex $argv 4]

# run
set run [lindex $argv 5]

# datadir
set DATADIR [lindex $argv 6]

set bandwidth 622;   # 622 Mbps
set delay 48;        # total of 100 ms RTT
set bf_size 1555;    # qlen is 20% BDP

# parameters for CUBIC
set cubic_tcp_mode 1

set low_window 0
set high_window 83000
set high_p 0.0000001
set high_decrease 0.1
set hstcp_fix 1

remove-all-packet-headers       ; # removes all except common
add-packet-header Flags IP TCP  ; # hdrs reqd for TCP

source utils.tcl

set frep [open $DATADIR/report.txt w]
set urep [open $DATADIR/util.out w]

puts "------------------------------------------------------------"
puts "------------------------------------------------------------"
puts [pwd]

printlegend

set ns [new Simulator]
```

```
#$ns use-scheduler Heap

global defaultRNG
$defaultRNG seed 9999;

for {set i 1} {$i < $run} {incr i} {
   $defaultRNG next-substream
}

set n1 [$ns node]
set n2 [$ns node]

set tcp0 [new Agent/TCP/Linux]

Agent/TCP/Linux set timestamps_ 1
Agent/TCP/Linux set window_ 67000
Agent/TCP/Linux set packetSize_ 1000
Agent/TCP/Linux set overhead_ 0.000008
Agent/TCP/Linux set max_ssthresh_ 100
Agent/TCP/Linux set maxburst_ 2

# BIC
$ns at 0 "$tcp0 set_ca_default_param tcp_bic beta 819"
$ns at 0 "$tcp0 set_ca_default_param tcp_bic max_increment 32"
$ns at 0 "$tcp0 set_ca_default_param tcp_bic fast_convergence 1"
$ns at 0 "$tcp0 set_ca_default_param tcp_bic BICTCP_B 4"
#$ns at 0 "$tcp0 set_ca_default_param tcp_bic initial_ssthresh 100"

# CUBIC
$ns at 0 "$tcp0 set_ca_default_param tcp_cubic max_increment 16"
$ns at 0 "$tcp0 set_ca_default_param tcp_cubic fast_convergence 1"
$ns at 0 "$tcp0 set_ca_default_param tcp_cubic beta 819"
$ns at 0 "$tcp0 set_ca_default_param tcp_cubic bic_scale 410"
$ns at 0 "$tcp0 set_ca_default_param tcp_cubic tcp_friendliness $cubic_tcp_mode"
#$ns at 0 "$tcp0 set_ca_default_param tcp_cubic initial_ssthresh 100"

puts "BufferSize(Packets) $bf_size"
puts $frep "BufferSize(Packets) $bf_size"

$ns duplex-link $n1 $n2 [expr $bandwidth]Mb [expr $delay]ms DropTail
$ns queue-limit $n1 $n2 $bf_size
$ns queue-limit $n2 $n1 $bf_size

set qmon [$ns monitor-queue $n1 $n2 ""]
set qfp [open $DATADIR/queue.out w]
print-queue 0.1 $qfp

set fmon [$ns makeflowmon Fid]
$ns attach-fmon [$ns link $n1 $n2] $fmon

# back path fmon
set fmon2 [$ns makeflowmon Fid]
$ns attach-fmon [$ns link $n2 $n1] $fmon2
```

```
set starttime(0) 0.1
set starttime(1) $flow2_start

for {set i 0} {$i < $hstcpflows_num} {incr i} {
  set hsnode(s$i) [$ns node]
  set hsnode(r$i) [$ns node]

  $ns duplex-link $hsnode(s$i) $n1 1000Mb 1ms DropTail
  $ns duplex-link $hsnode(r$i) $n2 1000Mb 1ms DropTail
  puts "hstcp$i, forwardlink delay=1 ms"
  puts $frep "hstcp$i, forwardlink delay=1 ms"
  puts "hstcp$i, RTT = 100.00 ms"
  puts $frep "hstcp$i, RTT = 100.00 ms"

  $ns queue-limit $hsnode(s$i) $n1 67000
  $ns queue-limit $n1 $hsnode(s$i) 67000

  $ns queue-limit $n2 $hsnode(r$i) 67000
  $ns queue-limit $hsnode(r$i) $n2 67000

   if {$hstcp_type($i) != "FAST"} {
        set hstcp$i [$ns create-connection TCP/Linux $hsnode(s$i) TCPSink/Sack1
$hsnode(r$i) $i]

        if {$hstcp_type($i) == "HS"} {
           $ns at $starttime(0) "[set hstcp$i] select_ca highspeed"
           set hstcpflows_type 8
           set low_wind 38
           $ns at 0 "[set hstcp$i] set_ca_default_param tcp_cubic low_window
$low_wind "
        } elseif {$hstcp_type($i) == "Scalable"} {
           $ns at $starttime(0) "[set hstcp$i] select_ca scalable"
           set hstcpflows_type 9
           set low_wind 16
           $ns at 0 "[set hstcp$i] set_ca_default_param tcp_cubic low_window
$low_wind"
           [set hstcp$i] set scalable_lwnd_ $low_window
        } elseif {$hstcp_type($i) == "BIC"} {
           $ns at $starttime(0) "[set hstcp$i] select_ca bic"
           set hstcpflows_type 12
           set low_wind 14
          $ns at 0 "[set hstcp$i] set_ca_default_param tcp_cubic low_window
$low_wind"

        } elseif {$hstcp_type($i) == "CUBIC"} {
           $ns at $starttime(0) "[set hstcp$i] select_ca cubic"
           set hstcpflows_type 13
        } elseif {$hstcp_type($i) == "HTCP"} {
       $ns at $starttime(0) "[set hstcp$i] select_ca htcp"
           set hstcpflows_type -10
      }

if {$hstcp_type($i) != "HTCP"} {
     [set hstcp$i] set max_ssthresh_ 100
```

```
} else  {
      [set hstcp$i] set max_ssthresh_ 1
}

   } else {
        set hstcp$i [$ns create-connection TCP/Fast $hsnode(s$i) TCPSink/Sack1
$hsnode(r$i) $i]
      [set hstcp$i] set alpha_ $alpha
      [set hstcp$i] set beta_ $alpha
   }

  set hsftp$i [[set hstcp$i] attach-app FTP]

  set lastBytes($i) 0

  $ns at $starttime($i) "[set hsftp$i] start"
  $ns at [expr $endtime+5] "[set hsftp$i] stop"

  # print this connection start time
  puts "hstcp$i starttime: $starttime($i)"
  puts $frep "hstcp$i starttime: $starttime($i)"

  set cfp_hs($i) [open $DATADIR/cwnd_hstcp$i.out w]
  set tput_hs($i) [open $DATADIR/tput_hstcp$i.out w]
  if { $hstcpflows_type == 12 } {
    print-cwnd-bic [set hstcp$i] 0.1 $cfp_hs($i)
  } else {
    print-cwnd [set hstcp$i] 0.1 $cfp_hs($i)
  }

  set lastKBytes($i) 0
  print-th-one $i $fmon 1
}

set halftime [expr $endtime / 2]

puts stderr "halftime: $halftime   endtime: $endtime"

for {set i 0} {$i < $hstcpflows_num} {incr i} {
  $ns at $halftime "print-stat-one-pre $i $fmon"
  $ns at $endtime "print-stat-one $i $fmon"
}

$ns at $halftime "print-stat-all-pre"
$ns at $endtime "print-stat-all"

$ns at $endtime "printlegend"
$ns at $endtime "finish"

$ns at 0 "timeReport 1"
$ns at 0.1 "print-util 0.1 0"

$ns run
```