A Comparison of TCP Behaviour at High Speeds Using ns-2 and Linux

Martin Bateman, Saleem Bhatti, Greg Bigwood, Devan Rehunathan, Colin Allison, Tristan Henderson
School of Computer Science, University of St Andrews, St Andrews, UK
{mb, saleem, gjb, dr, colin, tristan}@cs.st-andrews.ac.uk
and
Dimitrios Miras
Department of Computer Science, University College London, London, UK
d.miras@cs.ucl.ac.uk

Keywords: ns-2, testbed, TCP, high-speed, fairness

Abstract

There is a growing interest in the use of variants of the Transmission Control Protocol (TCP) in high-speed networks. ns-2 has implementations of many of these high-speed TCP variants, as does Linux. ns-2, through an extension, permits the incorporation of Linux TCP code within ns-2 simulations. As these TCP variants become more widely used, users are concerned about how these different variants of TCP might interact in a real network environment – how *fair* are these protocol variants to each other (in their use of the available capacity) when sharing the same network. Typically, the answer to this question might be sought through simulation and/or by use of an experimental testbed. So, we compare with TCP NewReno the *fairness* of the congestion control algorithms for 5 high-speed TCP variants – BIC, Cubic, Scalable, High-Speed and Hamilton – on both ns-2 and on an experimental testbed running Linux. In both cases, we use the same TCP code from Linux. We observe some differences between the behaviour of these TCP variants when comparing the testbed results to the results from ns-2, but also note that there is generally good agreement.

1. INTRODUCTION

ns-2 [1] is widely used in the research community to evaluate protocols in a simulation environment. Although other simulators are widely available, ns-2 remains popular as it is freely available, the source is available for inspection and modification, and there is a large user community that provides contributions in the form of extensions that allow new protocols and systems to be simulated. That same community has expressed concerns over the process of evolution of ns-2 and the impact this might have had on how well ns-2 compares to use of protocols in the 'real world' [4]. However, ns-2 remains popular and development on ns-2 continues in parallel with the major development under ns-3[1]. Meanwhile, work by Wei and Cao [10] allows real TCP code to be used in ns-2 simulations.

[1] `http://www.nsnam.org/`

It is important to understand in any specific case, just how well, or not, a simulation (any simulation, not just one based on ns-2) may relate to the real world scenario which it claims to simulate. Arguably, this is of particular importance when the scenario being simulated presents results for existing protocols (i.e. *post hoc*), rather than presenting results that concern, say, a new or experimental protocol that is still in development and is not yet widely deployed. This is because such results could affect deployment decisions and use of real world systems based on those results.

So, our aim in this paper is to examine how well the behaviour of TCP compares across ns-2 and a simple testbed, using the same TCP code in each case, by the use of simple experiments: measurements of throughput and evaluation for the *fairness* of throughput between flows.

1.1. Structure of this paper

We start by presenting our scenario for evaluation and define the *fairness* metric we will use in Section 2, and then go on to describe our configuration for the testbed and for ns-2, and a description of our experiments in Section 3. We then present our results in Section 4. We conclude in Section 5 with some comments on comparisons between testbed and simulation studies.

Our key contributions in this paper are:

- A critical comparison of ns-2 against a testbed for a specific case of TCP high-speed performance tests which use the same Linux TCP code.
- An evaluation of throughput *fairness* between TCP flows operating at high-speed in simulation and on a testbed.

2. SCENARIO

Our study concerns high-speed TCP variants, specifically those that are implemented and widely available in the Linux kernel. There is growing interest in more widespread use of these TCP variants as parts of the user community begin to use more demanding applications (e.g. data-intensive Grid applications). Users and especially administrators, however, are concerned with the impact these variants would have on

the network. This is because they adopt different congestion control algorithms to normal (NewReno) TCP; algorithms that are potentially more 'aggressive' in their transmission behaviour. This is by design: the goal is to make use of available network capacity that NewReno cannot utilise effectively due to its relatively conservative congestion control behaviour. More on the nature of this behaviour follow in the next section. The use of congestion control in TCP has been the key to the Internet's stability, so any change to this behaviour merits its investigation. In order to investigate this behaviour, a researcher may naturally turn to simulation with ns-2 in order to generate a set of experiments that are easily manageable, scalable, configurable and reproducible for their specific scenarios of interest, as reproducing and measuring those scenarios at scale in a testbed may not be feasible.

Therefore, we evaluate the efficacy of the ns-2 simulations for a very specific set of cases: we test the interaction between TCP NewReno and 5 variants of high-speed TCP when operating over network paths and end-to-end capacity of 1Gb/s and with various round-trip times (RTTs) - 50ms, 100ms, 200ms, 300ms and 400ms. On the end-to-end path, the flows in question share a bottleneck link, where congestion occurs, invoking the congestion control behaviour of the respective TCP variant. That is, the TCP flow is operating over a network path with a high bandwidth-delay product (BDP) and competing for resources at a bottleneck on that path. Our objective is to measure how the TCP variants compete with each other and how *fair* they are in their behaviour. We compare fairness by measuring the end-to-end throughput of the flow using a modified version of the well known tool, *iperf*[2], to generate the flows and to report throughput values.

We compare the results from ns-2 to those from a testbed using the same Linux TCP code in both cases. We use the ns-2 Linux enhancements [10] in order to import the TCP congestion control implementations from the Linux kernel version 2.6.22.6 into ns-2. We configure a physical testbed and ns-2 with the same network topology and using the same Linux kernel for sending and receiving nodes. This means that in each of the two cases – simulation and testbed – the same code will be used for the control of TCP flows generated.

So, we can change easily the congestion control algorithms which are in use for direct comparison between the testbed and ns-2 results. In this way, variations in implementation of the congestion control algorithm are removed and only the differences from the simulation and the testbed platform remain.

2.1. Fairness

Our key evaluation criteria is *TCP-Friendliness*: the fairness that a new high-speed TCP variant exhibits towards 'standard' TCP (NewReno) flows.

[2] `http://dast.nlanr.net/Projects/Iperf/`

We evaluate fairness for both the simulated and testbed environments. Key to our evaluation is a definition of the *fairness* metric. We use the well-known Jain's Fairness Index (JFI) [2] to evaluate overall system fairness:

$$F(x) = \frac{(\sum x_n)^2}{N(\sum x_n^2)} \quad (1)$$

where x_n is the measured throughput for flow n, for N flows in the system. In our experiments, we compare the pairwise throughput of two flows, and evaluate fairness between NewReno and each of the 5 other TCP variants ($N = 2$).

We evaluate these metrics by measuring the end-to-end throughput of a flow as reported by the simulation and the test-tools for our testbed, respectively. So, our evaluation mechanism and our metrics are fairly simple but we believe they give sufficient visibility of the observed behaviour in both cases – simulation and testbed. We recognise that other fairness metrics have been proposed for TCP (such as *proportional fairness* and *RTT fairness*), and that other measures of congestion control behaviour, such as time to convergence of flows (e.g. using *epsilon fairness*) might also be considered. We chose to use throughput fairness with a shared bottleneck as it is well-understood, it is easy to measure throughput and it is sufficient for our study.

2.2. High-speed TCP variants

In our work we use an experimental testbed that attempts to control as many of the salient end-system factors as possible. The specific TCP variants to evaluate were chosen using the following criteria:

- protocols that have implementations that are readily available for deployment at the time of writing.
- protocols that are being used or being considered for use across the Internet as well in private IP networks.
- protocols that have attracted interest by the research community.
- protocols that have been used in previous studies in order than we can validate the configuration, calibration and behaviour of our test-bed and simulation functions.

We have chosen BIC [11], Cubic [12], Hamilton TCP [9], High-speed TCP [3] and Scalable-TCP [7]. These *high-speed TCP variants* have attracted considerable interest from the research community to evaluate their performance, identify the need for modifications, and establish whether they are suitable for wider use on the Internet.

We provide in APPENDIX A brief definitions of the congestion control algorithms of each of the high-speed protocols evaluated in this work.

3. METHODOLOGY

We have taken great care to ensure that the ns-2 and testbed configurations are as similar as possible and so we claim that we are testing the same scenarios.

3.1. Testbed

Our testbed set-up is the well-known dumbbell arrangement as depicted in Fig. 1 and used also in previous similar studies [8, 5]. Whilst there are various criticisms that can be levelled at this arrangement, our aim was not to show a production network scenario but to provide a set-up that would allow us to test the behaviour of the TCP variants when used on real hardware in a way that is easy to compare with ns-2. Indeed, this very simple testbed helps us to reduce the factors of error or unknown behaviour that may affect the results. We are also aware of the work within the IRTF Transport Modelling Research Group[3], who are also looking to define mechanisms and metrics for evaluation of transport protocols, including fairness. The ongoing work within the TMRG includes an Internet Draft which suggests that a dumbbell topology is one scenario that can be used to evaluate transport protocol performance.

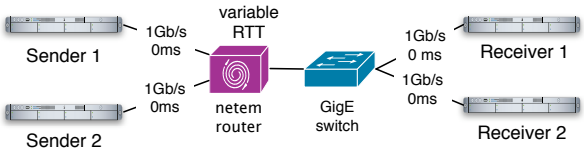


Figure 1. Testbed confguration

The dumbbell configuration, shown in Fig. 1, consists of two senders, two receivers and a router to provide the network delay. All the nodes were Intel Xeon Dual Core 1.6GHz with 2GB of DDR2 (667Mhz) memory and a PCI-X (64-bit, 133Mhz) bus. All network connections were 1Gb/s full-duplex Ethernet. The senders and receivers were running Linux kernel version 2.6.22.6, while the router was running Linux kernel version 2.6.18. The package *netem*[4] was used at the router to introduce delay to the packet flows. On the *netem* router, the network delay was split equally between the forward and reverse paths. The two flows are sent from Sender 1 to Receiver 1 and from Sender 2 to Receiver 2, respectively.

The senders and receivers were configured for high-speed TCP operation by setting the parameters shown in Table 1.

Units for all the parameter values above are in bytes, except for *txqueuelen* and *net.core.netdev_max_backlog* which are in packets. The parameters *net.ipv4.tcp_rmem* and *net.ipv4.tcp_rmem* each have three values. These are, respectively, the minimum, default and maximum values for the

[3] http://www.icir.org/tmrg/

[4] http://www.linux-foundation.org/en/Net:Netem

Linux kernel parameters	values
net.core.rmen_max	524288000
net.core.wmem_max	524288000
net.ipv4.tcp_rmem	4096 104857600 524288000
net.ipv4.tcp_wmem	104857600 524288000
txqueuelen	100000
net.core.netdev_max_backlog	100000
net.ipv4.tcp_timestamps *net.ipv4.tcp_window_scaling* *net.ipv4.tcp_sack* *net.ipv4.tcp_no_metrics_save* *net.ipv4.rfc1337*	enabled

Table 1. Linux kernel parameters: TCP senders / receivers

TCP receive and TCP transmission window sizes. With the TCP *window scaling* option, the initial window size is set to the default value of 104857600 bytes (∼100MB) but it is able to expand up to the maximum window size of 524288000 (∼500MB). In our test the maximum bandwidth delay product (BDP) value was 400ms×1Gb/s ∼50MB.

These settings ensure that there is sufficient buffer space in the end-system protocol stacks so that the end-system does not drop packets and packet drops are due only to the end-to-end capacity constraint imposed in the testbed. Also, the settings ensure that TCP is not constrained by default transmit (flow control) window size (64KB), which would make it perform poorly where the BDP is large (greater than 64KB). The MTU size was 1500 bytes, the default value.

3.2. Router configuration

The router was configured using Linux kernel version 2.6.18 with *netem*. This was used to control the RTT on the end-to-end path. In order to ensure that the *netem* router had enough buffer space to handle the large window sizes of the end-system TCP stacks, the kernel was setup as shown in Table 2. The kernel parameter *jiffies* was set to 1000Hz allowing the delay to be tuned to within 1ms.

Parameter	value
jiffies	1000
net.core.rmen_max	524288000
net.core.wmem_max	524288000
txqueuelen	100000
net.core.netdev_max_backlog	2500

Table 2. Linux kernel parameters - *netem* router

The units of the parameters in Table 2 are the number of buffers allocated by the kernel. The *netem* queues were configured with a limit of 420000 packets.

3.3. Testbed calibration and validation

The RTTs emulated were 50ms, 100ms, 200ms, 300ms and 400ms. For each RTT value, the system was validated:

- by testing the RTT by the use of *ping* from sender to receiver.
- by the use of *iperf* using UDP flows to ensure that a packet-level throughput of 1Gb/s was possible.

In order to check the behaviour of our testbed set-up, we generated single TCP flows and compared our observations to similar studies conducted independently by Leith et al [8] in 2003 and Ha et al [5] in 2006. We find that we have good agreement with Ha et al [5] and some agreement with Leith et al [8], though our tests are at higher data rates than these previous studies. Note, however, that we use different equipment and Linux kernel versions, and in the meantime some changes will have been made by the developers to the TCP code for the variants. So the best comparison is with the more recent study [5].

3.4. Flow generation for the testbed

The TCP flows were generated using a modified version of *iperf* v2.0.2[5], allowing us to switch easily between the TCP variants on a per-flow basis.

Each experiment was run for 600 seconds. The first flow was started at $t = 0s$ and the second flow was started at $t = 60s$ to help avoid initial synchronisation effects. The calculation of JFI used the throughput reported by *iperf* from 120s to 600s at 1s intervals, i.e. the first 119 measurements reported were ignored, to give the flows time to reach steady-state.

Each experiment was run 5 times, with the mean of the 5 values taken as the measured value, and the minimum and maximum values across the 5 runs plotted as error bars to show the variation.

3.5. ns-2 set-up

We used ns-2 version 2.31 with the TCP Linux extension from Wei and Cao [10], running Linux Kernel version 2.6.22.6 (the same as the testbed sender and receiver nodes). ns-2 was set to simulate a dumbbell network topology shown in Fig. 2. The bottleneck bandwidth of 1Gb/s was applied at the same time and on the same link as the variable RTT. The simulations were performed with the TCP protocol settings as documented in Table 3. The sizes are given in TCP 'packets', with a packet size of 1488 bytes this gives a window size of 148,800,000 bytes (∼141 MB), well above the BDP for the paths that will be used. The *queueLimit* parameter was applied to each path in the system, again using a value well above the BDP for the path, and is the same as the *txquelen* parameter fro *netem*. The *windowSize* parameter is the same value as *txqueuelen* on the sender and receiver nodes.

[5] Available, upon request, from the authors.

Parameter	value
windowSize	100000
queueLimit	420000

Table 3. ns-2 TCP Parameters

We have configured two experimental networks, one simulated and one testbed. We have configured in each case buffer space that exceeds the BDP for the paths used in our experiments, thus ensuring that 'network' buffering would not become a constraint on the throughput of any of the TCP flows.

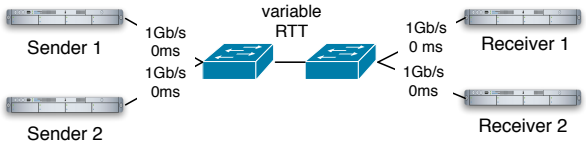


Figure 2. Simulation configuration

4. RESULTS

When we observe typical graphs for the throughputs of the pair-wise flows, we see that both flows reach high maximum throughputs, as shown in Fig. 3. We show the TCP variation under study for simulation and with the testbed. The competing NewReno flow which is omitted for clarity.

The results for Jain's Fairness Index (JFI) in Fig. 4 all show a good match between the flows generated on the simulation and on the testbed: error bars for maximum and minimum values are shown on each point, but may be too small to be visible in all cases. We note, however, that there is a noticeable difference between the mean values of the JFI, as highlighted by the lines joining the points on the graphs, even though the trends of the JFI values are consistent for the simulation and the testbed.

The difference in fairness between the testbed and ns-2 is between 0.04 and 0.21 with a mean difference of 0.087.

5. SUMMARY, CONCLUSION AND COMMENTS

We have performed a simple set of experiments comparing several TCP high-speed variants, both in simulation and in a testbed. We have deliberately kept the complexity of our network configuration low in order to highlight the TCP behaviour rather than explore 'realistic' network scenarios. We have also used a ns-2 extension to include the same TCP code into the simulation as is used by the testbed. We notice that typical behaviour over 5 runs is not significantly different when mean values for Jain's Fairness Index (JFI)

are considered across the experiments. However, it is clear that there is some difference between the mean values of JFI. Whether this can, with further experimentation, be shown to be a statistically-significant difference is another matter (we say more on this below). Comparing our results with previous tests by other groups who have used the same network topology, we can see that our testbed results are largely consistent, although the other tests were conducted at lower data rates and with different Linux kernel versions. So we believe that for these experiments, ns-2 can be used to report behaviour that reflects use of these protocols in operational use.

Our results show that:

- the behaviour of the *individual* TCP high-speed implementations show good agreement when run in ns-2 and when run across our testbed.
- the behaviour of the *pair-wise interaction* between TCP high-speed flows are very similar when run in ns-2 and when run across our testbed.
- code implementing a given algorithm when run in ns-2 can be used, with the extensions from Wei and Cao [10] to produce behaviour which is similar to that same code running in Linux on a simple testbed.
- there is good fairness, as measured using JFI evaluated using end-to-end throughput, between the high-speed TCP variants and NewReno.

We are encouraged that using 'real code' within ns-2 reflects behaviour that is consistent with a simple testbed. We believe, therefore, that there is great value in encouraging capability within ns-2 that allows the inclusion of real code taken from implementations of protocols.

For a general scenario, it is not clear that we can make, by performing more experiments, through the usual experimental techniques, a statistical inference about the comparison of ns-2 and the testbed. In this case, the variation of results from the testbed are such that we can consider the ns-2 results and testbed results to be equivalent. However, with a larger, more complex testbed, additional factors (e.g. CPU scheduling, memory management, hardware variations, etc.) may result in wider variations of results; it may not be possible to capture easily and accurately these factors within the simulation. Additionally, for a network designer or network operator, knowing the possible variation in capability within a given scenario may be just as valuable as knowing what the typical behaviour should be. So, it is not clear that we can generalise easily to conclude that it is possible to simulate larger testbeds with the same degree of accuracy as we have in our simple scenario.

We observe that if we consider the throughput and fairness values in these pair-wise experiments, both simulation and testbed, there does not appear to be a significant difference between TCP NewReno and the high-speed variants. However, the fairness measurements, made across the duration of the test, masks the overall higher throughput of the high-speed variants compared to NewReno. We do observe, however, that at higher RTT (higher BDP), we see greater instability between some of the flows in pair-wise tests, seen has higher variations (error bars) in the JFI values of Fig. 4. Additionally, our other observations (not reported in this paper) show that NewReno needs much tuning of the end-system stack in order to operate effectively at high-speeds, and even then its throughput starts to drop when the RTT (and so the BDP) increase (see the low throughput at 400ms for the testbed flow in Fig. 3(a)).

5.1. Ongoing and future work

We have also conducted pair-wise evaluations between the various high-speed variants, not just comparisons to NewReno, which we will report in future work. We are also currently looking at evaluating fairness for SCTP and DCCP in comparison to TCP, and also the use of other metrics, apart from JFI, for evaluating fairness.

ACKNOWLEDGEMENTS

The authors would like to express their great appreciation for all those who have dedicated their time and energy to creating, maintaining and furthering the development of ns-2, and those who have undertaken the task of creating ns-3.

REFERENCES

[1] ns-2 http://nsnam.isi.edu/nsnam/.

[2] D. Chiu and R. Jain. Analysis of the increase/decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN*, 17(1):1–24, June 1989.

[3] S. Floyd. Highspeed TCP for large congestion windows. RFC 3649, Dec 2003.

[4] S. Floyd and V. Paxson. Difficulties in simulating the internet. *IEEE/ACM Transactions on Networking*, 9:392–403, 2001.

[5] S. Ha, Y. Kim, L. Le, I. Rhee, and L. Xu. A step toward realistic performance evaluation of high-speed TCP variants. In *Fourth International Workshop on Protocols for Fast Long-Distance Networks (PFLDNet06)*, Feb 2006.

[6] V. Jacobson and M. Karels. Congestion avoidance and control. In *ACM SIGCOMM*, Aug 1988.

[7] T. Kelly. Scalable TCP: Improving performance in high-speed wide area networks. *SIGCOMM Computer Communication Review*, 33(2):83–91, 2003.

[8] Y.-T. Li, D. Leith, and R. Shorten. Experimental evaluation of TCP protocols for high-speed networks. Hamilton Institute. To appear in Transactions on Networking.

[9] R. N. Shorten and D. J. Leith. H-TCP: TCP for high-speed and long-distance networks. In *Fourth International Workshop on Protocols for Fast Long-Distance Networks (PFLDNet06)*, Feb 2006.

[10] David X. Wei and Pei Cao. NS-2 TCP-Linux: an NS-2 TCP implementation with congestion control algorithms from Linux. In *WNS2 '06: Proceeding from the 2006 workshop on ns-2: the IP network simulator*, page 9, New York, NY, USA, 2006. ACM Press.

[11] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control for fast long-distance networks. In *IEEE INFOCOM*, Mar 2004.

[12] L. Xu and I. Rhee. CUBIC: A new TCP-Friendly high-speed TCP variant. In *Third International Workshop on Protocols for Fast Long-Distance Networks (PFLDNet06)*, Feb 2005.

APPENDIX A

We assume the reader is familiar with basic congestion control in TCP, using a *congestion window (cwnd)*, Additive Increase Multiplicative Decrease (AIMD) behaviour [6]:

$$\begin{aligned} OnACK : cwnd &\leftarrow cwnd + \alpha \\ OnLoss : cwnd &\leftarrow \beta.cwnd \end{aligned}$$

with $\alpha = 1$ and $\beta = 0.5$.

BIC TCP

Binary Increase Congestion control TCP – *BIC* TCP [11] – uses a binary search algorithm between the window size just before a reduction (W_{max}) and the window size after the reduction (W_{min}). If w_1 is the midpoint between W_{min} and W_{max}, then the window is rapidly increased when it is less than a specified distance S_{max} from w_1 and grows more slowly when it is near w_1. If the distance between the minimum window and the midpoint is more then S_{max}, the window is increased by S_{max}, following a linear increase. BIC reduces *cwnd* by a multiplicative factor β. If no loss occurs, the new window size becomes the current minimum, otherwise, the window size becomes the new maximum. If the window grows beyond the current maximum, an exponential and then linear increase is used to probe for the new equilibrium window size.

Cubic TCP

Cubic TCP [12] uses a cubic function to control its congestion window growth. If W_{max} is the congestion window before a loss event, then after a window reduction, the window grows fast and then slows down as it approaches W_{max}. Around W_{max}, the window grows slowly, again accelerating as it moves away from W_{max}. The following formula determines the congestion window size (*cwnd*):

$$cwnd = C(T-K)^3 + W_{max}$$

where C is a scaling constant, T is the time since the last loss event and $K = \sqrt[3]{W_{max}\frac{\beta}{C}}$, where β is the multiplicative decrease factor after a loss event. C and β are set to 0.4 and 0.2 respectively. To increase fairness and stability, the window is clamped to grow linearly when it is far from W_{max}.

Hamilton TCP (HTCP)

Hamilton-TCP (HTCP) [9] modulates the normal AIMD parameters as a function of the elapsed time Δ since the last congestion event.

$$\begin{aligned} OnACK : cwnd &\leftarrow cwnd + \frac{2(1-\beta)f_\alpha(\Delta)}{cwnd} \\ OnLoss : cwnd &\leftarrow g_B(B).cwnd \end{aligned}$$

where:

$$f_\alpha(\Delta) = \begin{cases} 1 & \Delta \le \Delta_L \\ max(\bar{f}(\Delta)T_{min}, 1) & \Delta > \Delta_L \end{cases}$$

$$g_B(B) = \begin{cases} 0.5 & |\frac{B(k+1)-B(k)}{B(k)}| > \Delta_B \\ min(\frac{T_{min}}{T_{max}}, 0.8) & otherwise \end{cases}$$

Δ_L is a threshold such that the standard TCP algorithm is applied if $\Delta \le \Delta_L$, and a quadratic function f_α is used: $\bar{f}_\alpha = 1 + 10(\Delta - \Delta_L) + 0.25(\Delta - \Delta_L)^2$. T_{min} and T_{max} are, respectively, the minimum and maximum round-trip times seen by the flow, and $B(k+1)$ is the maximum bandwidth measurement during the last congestion epoch.

Highspeed-TCP

Highspeed-TCP [3] modulates its increase and decrease parameters as a function of the current value of *cwnd*. The higher the current value of *cwnd*, the higher its additive increase for a lossless RTT time interval. It also uses a smaller multiplicative decrease coefficient in response to a loss event.

$$\begin{aligned} OnACK : cwnd &\leftarrow cwnd + a(cwnd)/cwnd \\ OnLoss : cwnd &\leftarrow b(cwnd).cwnd \end{aligned}$$

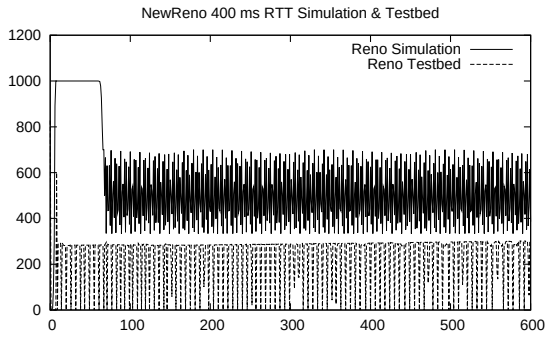
Highspeed-TCP employs an increasing logarithmic function for $a(cwnd)$ and a decreasing logarithmic function for $b(cwnd)$. The AIMD parameters revert to standard TCP values (1 and 0.5 respectively) if *cwnd* is below a low threshold.

Scalable-TCP

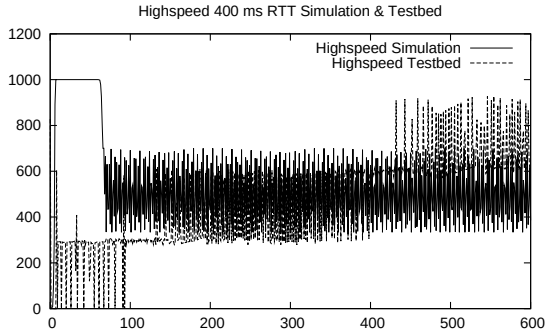
Scalable-TCP [7] tackles the problem of the long recovery time of standard TCP over large BDP links by making the process of updating its congestion window independent of the congestion window value at any time. The generalised Scalable-TCP modifies the congestion window as:

$$\begin{aligned} OnACK : cwnd &\leftarrow cwnd + \alpha \\ OnLoss : cwnd &\leftarrow \beta.cwnd \end{aligned}$$

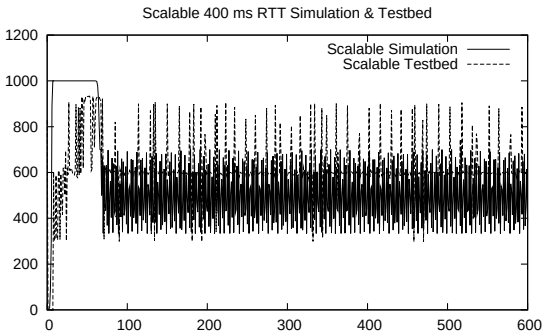
where α and β are constants, with $\alpha, \beta \varepsilon (0,1)$ (the recommended values are $\alpha = 0.01$ and $\beta = 0.875$). Scalable-TCP reverts to the standard TCP congestion window update algorithm when its congestion window is smaller than a certain threshold, the *legacy congestion window* (*lcwnd*).



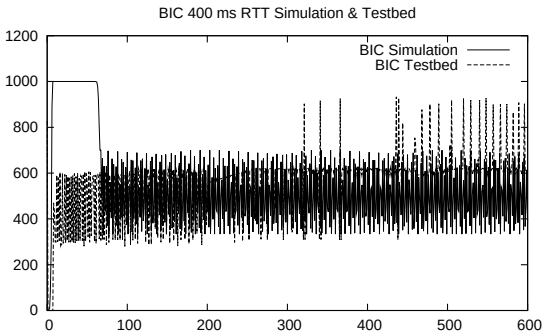
(a) NewReno



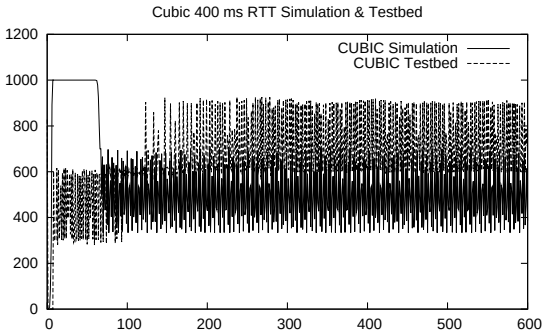
(b) Highspeed



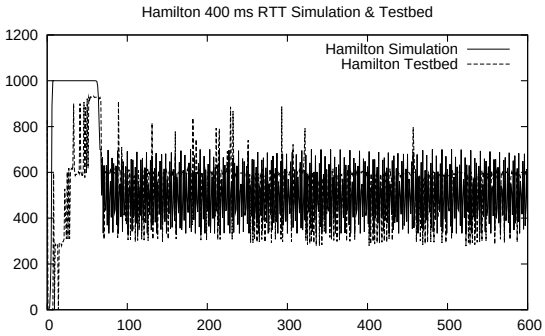
(c) Scalable



(d) BIC

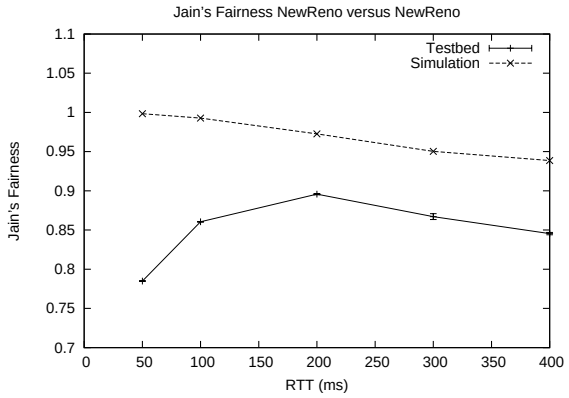


(e) Cubic

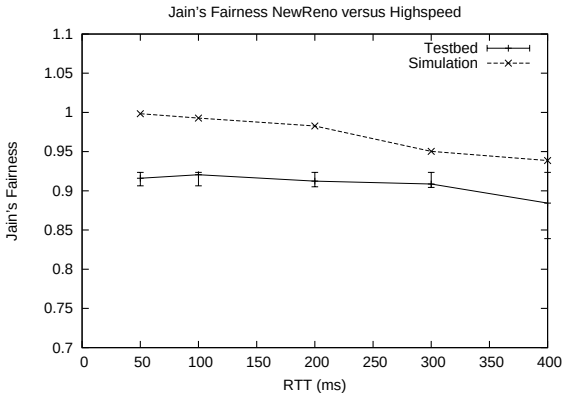


(f) Hamilton

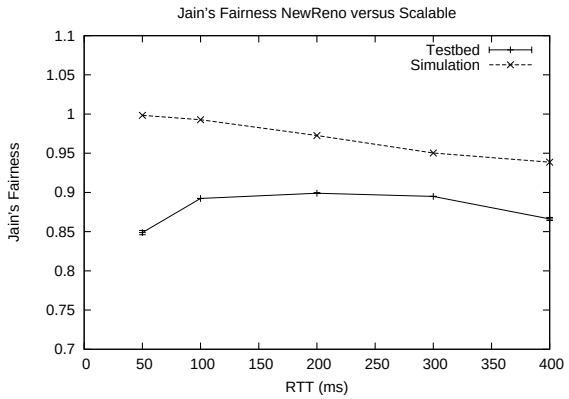
Figure 3. Throughput



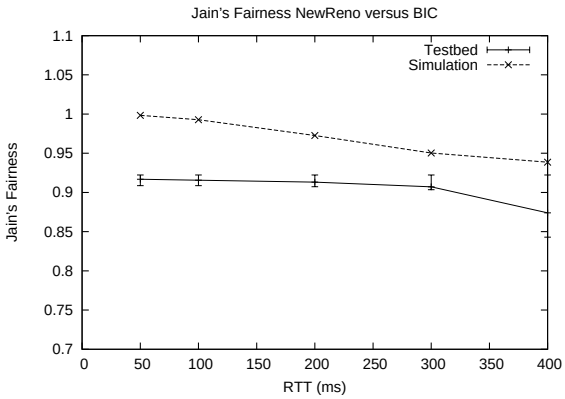
(a) NewReno vs NewReno



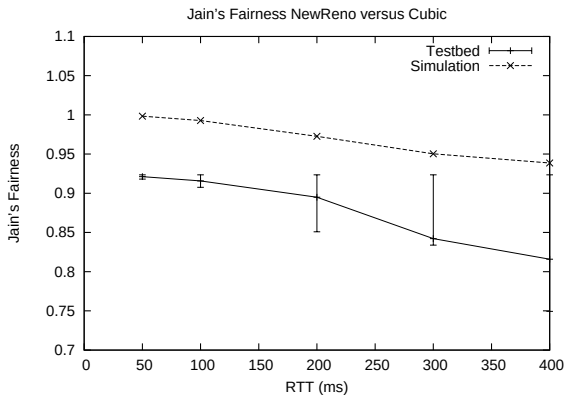
(b) NewReno vs Highspeed



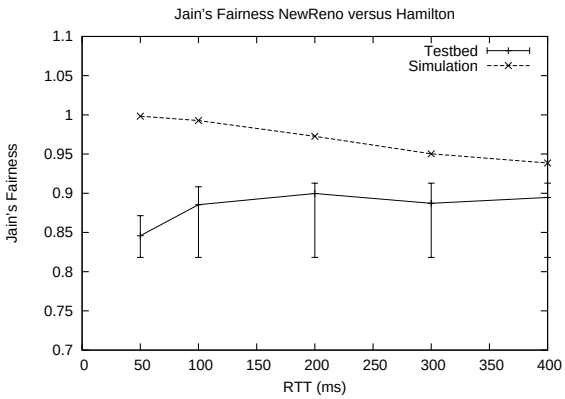
(c) NewReno vs Scalable



(d) NewReno vs BIC



(e) NewReno vs Cubic



(f) NewReno vs Hamilton

Figure 4. Jain's Fairness Index