View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by MURAL - Maynooth University Research Archive Library

Evaluating the Performance of TCP Stacks for High-Speed Networks

B.Even, Y.Li, D.J.Leith
Hamilton Institute, Ireland

***Abstract*— In this paper we present experimental results evaluating the performance of the Scalable-TCP, HS-TCP, BIC-TCP, FAST-TCP and H-TCP proposals for changes to the TCP congestion control algorithm to improve performance in high-speed network paths.**

***Index Terms*— TCP Congestion control; Evaluation of TCP protocols; High-speed networks.**

I. Introduction

The TCP congestion control algorithm has been remarkably successful in making the current internet function efficiently. However, in recent years it has become clear that it can perform very poorly in networks with high bandwidth-delay product (BDP) paths. In response to this poor performance, numerous proposals have been made in recent years for changes to the TCP congestion control algorithm. These include the HS-TCP proposal of Floyd[6], the Scalable-TCP proposal of Kelly[10] and the FAST-TCP proposal of Low *et al*[7]; more recent proposals include BIC-TCP[16] and H-TCP[12]. These proposals have all been the subject of considerable interest and experimentation in recent years.

Due in no small part to the volume of work that has been carried out in this area, a real need has developed for systematic screening of proposals to identify suitable candidates for more detailed evaluation. Evaluating the performance of new TCP proposals is not easy. One principal difficulty arises form the lack of an agreed set of performance measures. As a result of the latter, different studies typically employ performance tests that highlight particular aspects of TCP performance while casting little light on other, equally important, properties of proposed protocols. Most existing work also fails to control for variations in performance associated with differences in network stack implementation that are unrelated to the congestion control algorithm (see below). This is an important practical aspect that is frequently ignored in academic studies on the topic. In view of these facts it is not surprising that concrete conclusions relating to the merits of competing proposals have been difficult to make based on currently available published results.

Our aim in this paper is to compare the performance of competing TCP proposals in a fair and consistent manner. We present experimental measurements of the performance of the HS-TCP, Scalable-TCP, FAST-TCP, BIC-TCP and H-TCP[1] proposals.

These tests highlight a number of specific deficiencies of the protocols studied, and suggest future research directions to render these suitable for deployment in real networks. In summary, we find that both Scalable-TCP and FAST-TCP consistently exhibit substantial unfairness, even when competing flows share identical network path characteristics. Scalable-TCP, HS-TCP and BIC-TCP all exhibit slow convergence and sustained unfairness following changes in network conditions such as the start-up of a new flow. FAST-TCP exhibits complex convergence behaviour.

The paper is structured as follows. In Section II we discuss some issues that limit the utility of previous evaluation studies and motivate the present work. In Section III network stack implementation issues affecting tests in high-speed networks are discussed. In Section IV we present our experimental measurements. The implications of these results are discussed in more detail in Section V and the conclusions are summarised in Section VII.

II. Some Pitfalls

Comparing the performance of TCP proposals is not always easy and many pitfalls exist. Examples include the following.

Different network stack implementations. In almost all recent studies on high-speed networks, publicly available Linux patches provided by the authors of TCP proposals are used. The performance of these patches are then compared directly. However, patches may relate to different operating system versions. More seriously, performance issues relating to the inefficiency of the network stack implementation, particularly in relation to SACK processing, are known to have a significant impact on performance (e.g. see [11]). As a result, most patches implementing proposed changes to the TCP congestion control algorithm also implement numerous changes to the network stack that are unrelated to the congestion control algorithm. Consequently, direct performance comparisons of these patches risk revealing more about the efficiency of the network stack implementation than about the performance of the congestion control algorithm. In this paper, we use a common network stack implementation with all of the congestion control algorithms studied in order to focus solely on the latter's performance. This implementation is discussed in more detail in the next section.

Congestion control action not exercised. It is important to design experiments that exercise the TCP congestion control

[1] We note that H-TCP is developed by some of the authors of this paper. We emphasise therefore that all of the protocols studied are put through identical tests yielding quantitative and repeatable measurements. While space restrictions prevent us from including all of our experimental measurements in this paper, the measurements are available at www.hamilton.ie/net/eval/.

algorithm rather than other elements of the network stack. For example, it is essential that the bandwidth of the network is lower than that of the server network interface card (NIC), i.e. that the network bottleneck lies external to the server being tested. Otherwise, it is often the case that the transport layer congestion control algorithm is effectively inactive (packet drops are virtual) and performance measurements merely evaluate the efficiency of the NIC driver.

Performance measures too narrow. We argue that it is not sufficient to focus solely on the throughput performance of a single flow. Fairness, responsiveness, *etc* between competing TCP flows should also be evaluated.

Range of network conditions. Frequently results are presented from a single test run only and/or for a specific network condition or small range of network conditions. A huge variety of conditions exist in modern networks. We argue that it is essential, as a minimum, to characterise TCP performance across a broad range of bandwidths (not just on high-speed links), propagation delays (not just trans-continental links) and router buffer sizes (not just very large or very small buffers).

Such issues limit the utility of previous evaluation studies and motivate the approach taken in the present paper. We do not claim that our tests in this paper are exhaustive. We do, however, seek to demonstrate their utility and discriminating power and to initiate wider debate on this topic in the grid community.

III. Network Stack Implementation

A. Performance Issues

In this paper we focus on the Linux 2.4/2.6 network stack implementation as it is widely used in TCP research for high-speed networks. Specifically, measurements were taken using commodity high-end servers, see Table I for details, and a Linux 2.6.6 kernel modified to include instrumentation of network stack operation and timing. Many of the issues discussed are, however, also relevant to other operating system network stacks. The efficiency of the TCP implementation for high-speed networks has received considerable attention with, for example, widespread support within gigabit-speed (and above) network interface cards for hardware offload to reduce the processing burden within the operating system kernel. However, the bulk of this work has focussed on fast path optimisation. While fast path performance is key in situations where packet loss is rare (e.g. server farms), we demonstrate that it is the slow path performance that is the bottleneck in wide-area transfers where the probing action of the AIMD strategy in TCP's congestion control action means that packet loss is an intrinsic feature of normal operation.

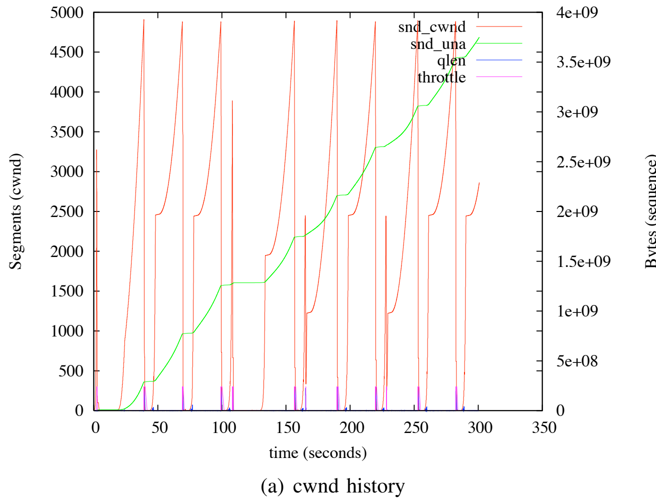
This is illustrated in Figure 1, which shows the $cwnd$ for a TCP flow on a 250Mb/s path with 200ms two-way propagation delay. It can be seen that, following packet loss, the data transfer stalls (the TCP ACK sequence number snd_una does not advance) and the congestion window $cwnd$ remains at a small value for an extended period. Also shown in Figure 1 is the occupancy of the netdev queue that lies between the NIC driver and the TCP stack. Incoming packets are placed in this queue by the NIC driver to await processing by the TCP stack. When the netdev queue becomes full, Linux enters a throttle mode whereby all subsequent incoming packets are dropped until processing of the queued packets completes and the netdev queue empties; time spent in this throttle mode is also marked on the figure. What we see happening is that following packet loss the TCP stack is unable to process the incoming TCP ACK packets quickly enough, resulting in a build up of ACK packets in the netdev queue and entry into throttle mode. Once throttle mode is entered, the resulting sustained loss of ACK packets stalls the TCP data transfer. Eventually this induces a TCP timeout and recovery. To our knowledge, these are the first published stack measurements detailing the performance degradation within the Linux network stack during high-speed operation and describing the mechanism involved.

The underlying problem is that the TCP stack is unable to process incoming ACK packets quickly enough as the network speed rises. Figure 1(b) plots the distribution of ACK processing times. On a 250Mb/s link with 1500 byte data packets, the mean inter-arrival time between packets is $48\mu s$. It can be seen that many ACK packets take much longer than 48 μs to process (specifically, in this example 53.4% of ACK packets take longer than 48 μs to process). On further inspection, we find that long ACK processing times are largely associated with time spent processing SACK information. The SACK information included in a TCP ACK consists of notification by the receiving machine of up to three contiguous blocks of packets that have successfully reached the destination - packet losses can be inferred from the gaps between these blocks. Processing of SACK information imposes a significant book-keeping burden on the sender. The current SACK processing algorithm involves, for each ACK received, multiple walks by the sender of the linked-list consisting of data packets sent but not yet acknowledged (i.e. the packets in flight). Hence, the computational burden of the algorithm scales with $cwnd$ and quickly becomes unacceptable on high bandwidth-delay product paths. We note that the burden is also exacerbated by the switch from delayed acking (whereby an ACK packet is sent for roughly every two data packets received) to quick acking (where an ACK packet is sent for every data packet) during loss recovery, effectively doubling the number of ACK packets that need to be processed.

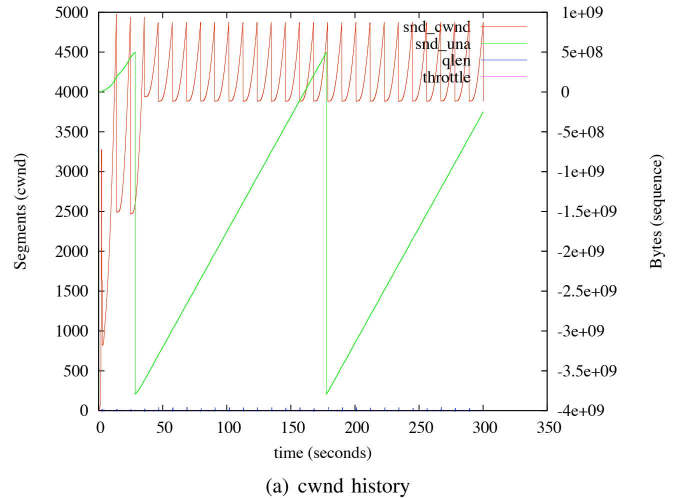
	Description
CPU	Intel Xeon CPU 2.80GHz
Memory	256 Mbytes
Motherboard	Dell PowerEdge 1600SC
Kernel	Linux 2.6.6 altAIMD-0.6
txqueuelen	1,000
max_backlog	300
NIC	Intel 82540EM Gigabit Ethernet Controller
NIC Driver	e1000 5.2.39-k2
TX & RX Descriptors	4096

TABLE I

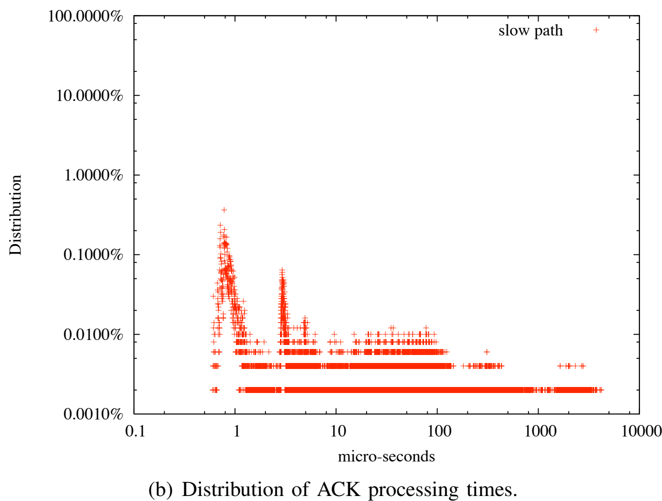
Hardware and Software Configuration.



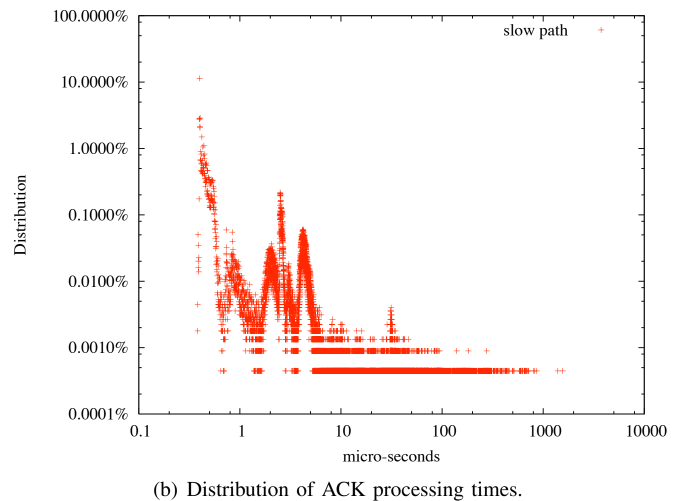
(a) cwnd history



(a) cwnd history



(b) Distribution of ACK processing times.



(b) Distribution of ACK processing times.

Fig. 1. Performance of a single TCP transfer on a path with 250Mb/s bottleneck bandwidth and 200ms two-way propagation delay; Linux 2.6.6 kernel, H-TCP congestion control algorithm (standard TCP algorithm is too sluggish).

Fig. 2. Performance of a single TCP transfer on a path with 250Mb/s bottleneck bandwidth and 200ms two-way propagation delay; *altAIMD* modified Linux 2.6.6 kernel, H-TCP congestion control algorithm.

B. Improving Efficiency

We consider the following changes to the network stack to improve efficiency.

- *O(loss) SACK processing.* As noted above, the current implementation of SACK processing in the Linux kernel requires a processing time which is $O(cwnd)$. We have implemented a modified SACK processing algorithm that only requires a walk of the unsacked "holes" in the linked-list of packets in flight rather than a walk of the entire list. This algorithm effectively scales with $O(loss)$ rather than $O(cwnd)$ and so can yield significant performance gains in high bandwidth-delay product networks.
- *SACK block reordering.* Sorting SACK blocks by sequence number allows a single pass through the packets in flight link-list (in the standard Linux implementation a separate pass is made for each SACK block i.e. typically three passes for every ACK packet received).
- *ACK queueing.* Significant efficiency gains are possible by coalescing the SACK information from multiple ACK packets - in effect, by using a form of delayed acking inside the SACK processing code.
- *Throttle Disabled.* A build-up of ACK packets at the sender can cause an overflow in the netdev queue which invokes a throttle action that causes all packets to be dropped. We have modified this behaviour so that the netdev queue operates a pure drop-tail discipline.

Patches implementing these changes are available online at www.hamilton.ie/net/. The impact of these changes can be seen in Figure 2. It can be seen that in this example the netdev queue now never fills (and so no ACK packets are now dropped within the network stack) and stalling of the data transfer no longer occurs. The impact on ACK processing time is shown in Figure 2(b), where it can be seen that the distribution of processing times has been pushed to the left so that now almost all ACKs are processed in less than 48 μs. We note that the distribution of ACK processing times does include a tail beyond 48 μs. The source of this tail is currently unclear (it seems to be associated with occasional extended memory access times) but affects only a very small proportion of ACKs packets - specifically, only in this example 0.11% of ACK packets take longer than 48 μs to process.

C. altAIMD kernel

In the rest of this paper, we make use of a Linux 2.6.6 kernel modified as discussed above. In addition, we have included in this kernel implementations of the Scalable-TCP, HS-TCP, FAST-TCP, BIC-TCP and H-TCP congestion control algorithms together with Appropriate Byte Sizing (RFC3465)[1][2]. A single `sysctl` is used to switch between congestion control algorithms on-the-fly, without the requirement of rebooting. This kernel is referred to as the *altAIMD* kernel and is available online at www.hamilton.ie/net/. The congestion control algorithm implementations are based on publicly available patches[3] . However, these patches are often for different versions of Linux and, as noted previously, typically also make extensive changes to the network stack that are not directly related to the congestion control algorithm. Use of a common network stack implementation therefore serves to provide consistency, and control against the influence of differences in implementation as opposed to differences in the congestion control algorithm itself.

IV. TCP Congestion Control Performance

It is important to emphasise that our goal in this paper is not to achieve exhaustive testing, but rather to perform initial screening of proposals for changes to the TCP congestion control algorithm. We therefore seek to employ benchmark tests that can be consistently applied and that exercise the core functionality of TCP. The performance problems of standard TCP over high bandwidth-delay product paths are largely associated with bulk data transfers. It is therefore natural to take this as our starting point in testing new TCP proposals. In addition to restricting our attention to long-lived flows, we also confine consideration to drop-tail queues, since this is the prevalent queueing discipline in current networks, and to a single shared bottleneck link. Short-lived TCP flows, and indeed non-TCP flows, constitute a large proportion of traffic in real networks. Similarly, not all routers operate drop-tail queueing disciplines. However, as we shall see, restricting our attention to long-lived TCP flows operating in drop-tail environments is already sufficient to highlight important features of new TCP proposals.

A. Test Setup

Before proceeding, we consider some issues common to all of our proposed tests.

We define our tests on the dumbbell topology shown in Figure 3. We recognize that this topology is a limited one, but the behaviour of standard TCP on this topology is well studied and so it provides a natural starting point. All tests are conducted on an experimental testbed with commodity high-end PCs connected to gigabit switches to form the branches

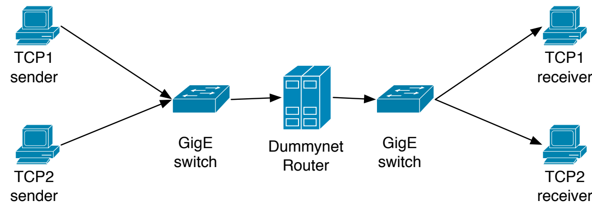


Fig. 3. Experimental set-up.

of the dumbbell topology. All sender and receiver machines used in the tests have identical hardware and software configurations as shown in Tables I and II and are connected to the switches at 1Gb/sec. The sender and receiver machines run the *altAIMD* kernel. The router, running the FreeBSD dummynet software, can be configured with various bottleneck queue-sizes, capacities and round trip propagation delays to emulate a range network conditions.

We consider round-trip propagation delays in the range 16ms-320ms and bandwidths ranging from 1Mb/s-250Mb/s. We do not consider these values to be definitive – the upper value of bandwidth considered can, in particular, be expected to be subject to upwards pressure. We do, however, argue that these values are sufficient to capture an interesting range of network conditions that characterises current communication networks. In all of our tests we consider delay values of 16ms, 40ms, 80ms, 160ms, 320ms and bandwidths of 1Mb/s, 10Mb/s, 100Mb/s and 250Mb/s. This defines a grid of measurement points where, for each value of delay, performance is measured for each of the values of bandwidth.

In order to minimise the effects of local hosts queues and flow interactions, we only ran one flow per PC. Flows are injected using `iperf` into the testbed. Each individual test was run at least ten minutes each. In the case of tests involving Standard TCP, we ran individual tests for up to an hour as the congestion epoch duration becomes very long on large bandwidth-delay products paths. In order to obtain a good representation of the run-to-run variability in performance metrics, all individual tests were repeated at least 5 times and the arithmetic mean taken. An error on the measurement was taken as the standard error from this mean.

B. Response Function Test

As our starting point we evaluate the impact of packet loss on efficiency, similarly to Padhye *et al* [14] and Floyd[5], by measuring the average throughput of a single TCP flow as the level of random packet losses is varied. As discussed previously, these measurements are carried out for a range of propagation delays and link bandwidths. Measurements of the response functions obtained are shown in Figure 4.

C. Fairness Test

We next evaluate fairness by considering two TCP flows with shared bottleneck link. Owing to space restrictions, we cannot include the results of all our tests here. We therefore present results for a subset of network conditions that are

[2]The counting of ACK's by the number of bytes acknowledged rather than the number of ACKS's received to counter the problems of slow $cwnd$ growth when using delayed acking.

[3]We note that the implementation of BIC-TCP included in the standard Linux 2.6.6 kernel distribution is known[13] to be incorrect (this has subsequently been corrected). In our tests we use a corrected implementation based upon the original Linux patch developed by the BIC-TCP authors.

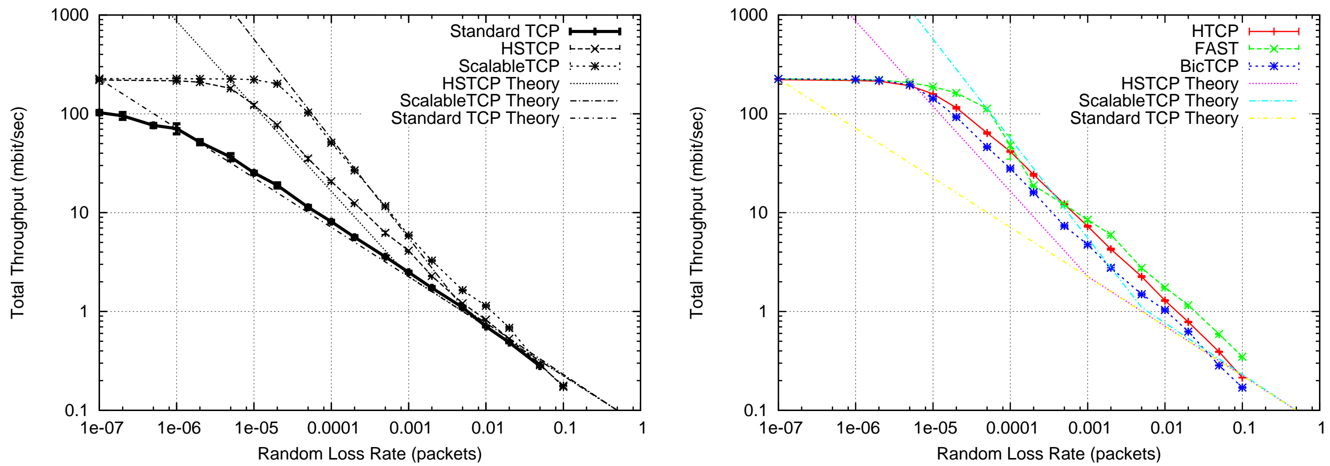


Fig. 4. Measured response functions with 250Mbit/sec bottleneck link and 162ms RTT.

Fig. 5. Ratio of throughputs of two flows under symmetric conditions (same propagation delay, shared bottleneck link, same congestion control algorithm) as path propagation delay is varied. Results are shown for 10Mbit/sec and 250Mbit/sec bottleneck bandwidths. The bottleneck queue size is 20% BDP. Observe that while standard TCP and H-TCP are essentially fair (the competing flows achieve, to within 5%, the same average throughput) under these conditions, Scalable-TCP and FAST-TCP are notably unfair. HS-TCP and BIC-TCP can also be seen to exhibit significant unfairness, albeit to a lesser degree than Scalable-TCP and FAST-TCP.

representative of the full test results obtained. Figure 5 plots the ratio of measured throughputs for two flows with the same propagation delay sharing a common bottleneck link as the path propagation delay is varied. Tests are of 10 minutes duration. Results are shown both for a bottleneck link bandwidth of 10 Mb/s and 250Mb/s, roughly corresponding to low and high-speed network conditions. Results are shown when the queue is sized at 20% BDP but similar results are also obtained when the queue is 100% BDP.

V. Discussion

Perhaps the most striking results are obtained from the fairness test where the throughputs of competing flows with the same propagation delay are compared, see Figure 5. Under these conditions, the standard TCP congestion control algorithm consistently ensures that each flow achieves the same (to within less than 5%) average throughput. However, the measurements shown in Figure 5 indicate that many of the proposed protocols exhibit substantial unfairness under the same conditions. While both FAST-TCP and Scalable-TCP display very large variations in fairness, BIC-TCP and HS-TCP also display significant levels of unfairness.

In view of the somewhat surprising nature of these results, it is worthwhile investigating this behaviour in more detail. We consider in turn each of the protocols exhibiting greater levels of unfairness than standard TCP.

- *Scalable-TCP.* Figure 6 shows typical examples of measured $cwnd$ time histories. It can seen that the $cwnd$'s either do not converge to fairness or else converge very slowly indeed (not reaching fairness within the 10 minute duration of these tests). Although sometimes expressed as a modified additive increase algorithm, it is easily shown that the Scalable-TCP algorithm is in fact a multiplicative-increase multiplicative-decrease (MIMD) algorithm. It has been known since the late 1980s [3] that in drop-tail networks such algorithms may not converge to fairness.
- *FAST-TCP.* Figure 7 shows typical examples of measured $cwnd$ time histories when using the FAST-TCP algorithm. The upper figure shows measurements taken on a 250Mb/s path with 42ms propagation delay. Rapid variations in $cwnd$ are evident which are somewhat surprising in view of the delay-based rather than loss-based nature of the FAST-TCP algorithm. The middle figure shows the $cwnd$'s measured when the propagation delay on the path is increased to 162ms. The rapid variations in $cwnd$ are no longer present, but the flows now exhibit a number of abrupt changes in $cwnd$ including a sharp increase in unfairness after 500s. It is perhaps worth emphasising that these examples are representative of our measurements across a wide range of network conditions and are not selected as worst case behaviours. Our purpose in this paper is not to analyse or explain the FAST-TCP algorithm or its performance. We do, however, comment that the behaviour in the low latency example appears to be associated with use of an aggressive $cwnd$ increase strategy leading to flooding of the queue and consequently generating many packet losses, while the behaviour in the higher latency example appears to be associated with the adaptive switching of the increase strategy.

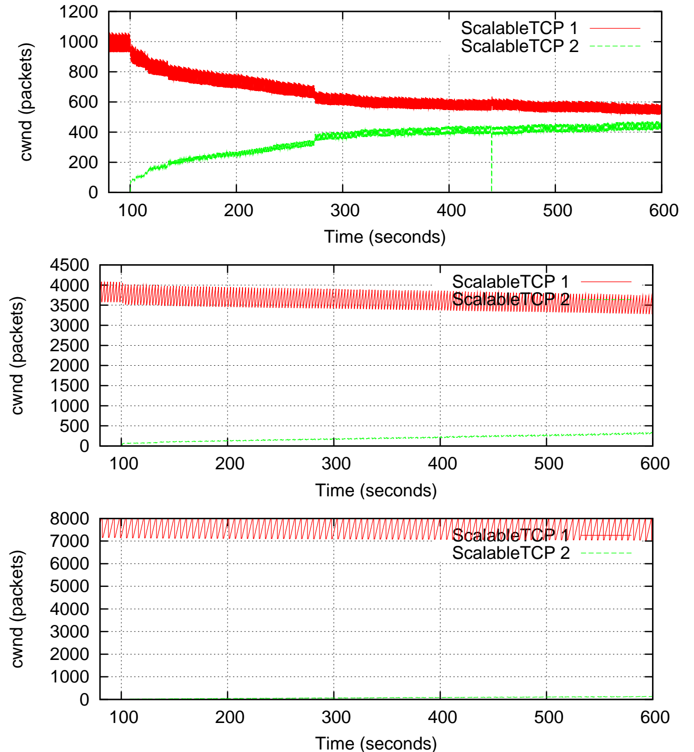


Fig. 6. Scalable-TCP $cwnd$ time histories following startup of a second flow. RTT of both flows is 42ms (top), 162ms (middle) and 324ms (bottom). Bottleneck bandwidth is 250Mbit/sec, queue size 20% BDP.

- *HS-TCP.* Figure 8 shows examples of HS-TCP $cwnd$ time histories for flows with the same round-trip time following startup of a second flow. It can be seen that the flows do converge to fairness, but that the convergence time can be long. This effect becomes more pronounced as the path propagation delay is increased. These experimental measurements are in good agreement with the simulation results previously reported in [15]. Recall that the AIMD increase parameters are functions of $cwnd$ in HS-TCP. The slow convergence appears to originate in the asymmetry that exists in HS-TCP between the AIMD parameters of newly started flows (with small $cwnd$) and existing flows (with large $cwnd$). Existing flows with large $cwnd$ have more aggressive values of increase and decrease parameters than do newly started flows which have small $cwnd$. Hence, sustained unfairness can occur.
- *BIC-TCP.* Figure 9 shows examples of the $cwnd$ time history of BIC-TCP following startup of a second flow. It can seen that as the path propagation delay increases the $cwnd$'s converge increasingly slowly, not reaching fairness within the 10 minute duration of these tests when the path propagation delay is large. This behaviour manifests itself in Figure 5 as a fall in the measured fairness as propagation delay increases.
- *H-TCP.* Figure 10 shows $cwnd$ time histories of H-TCP

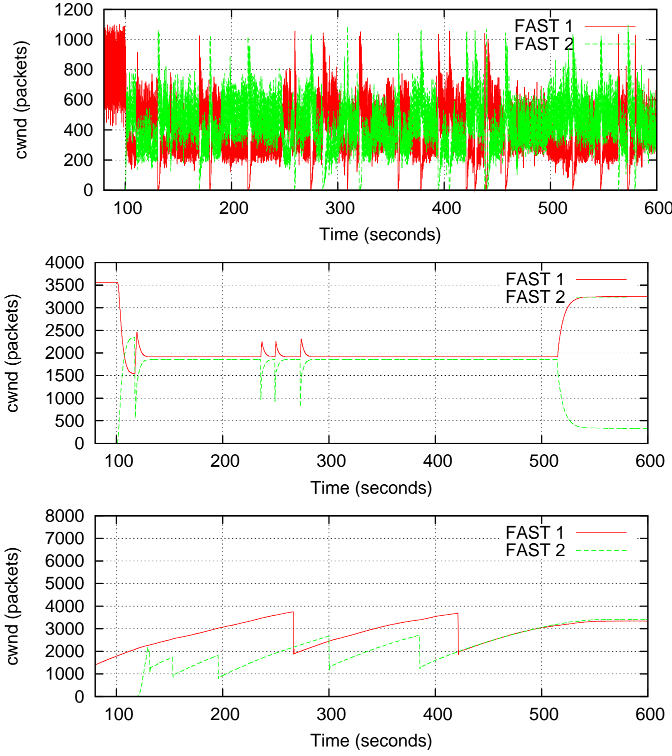


Fig. 7. FAST-TCP $cwnd$ time histories following startup of a second flow. RTT is 42ms (top), 162ms (middle) and 324ms (bottom). Bottleneck bandwidth is 250Mbit/sec, queue size 20% BDP.

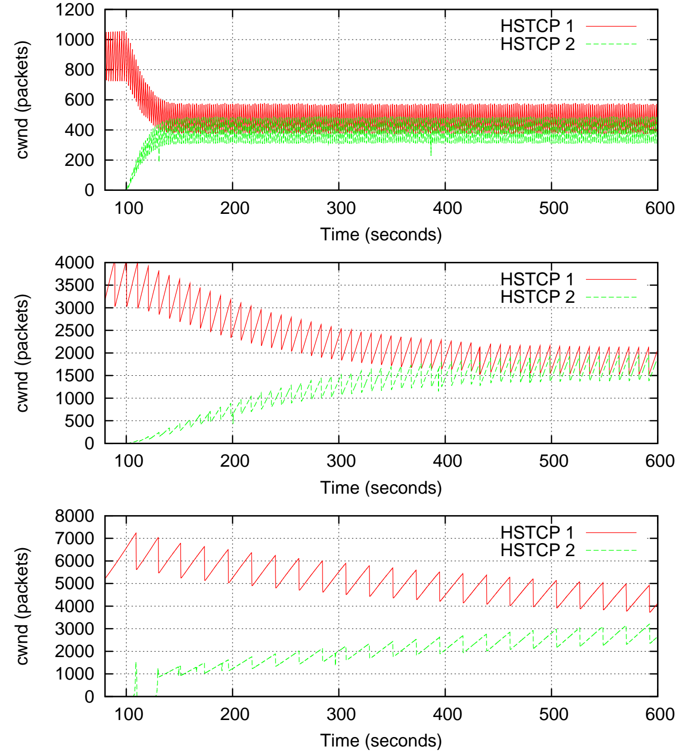


Fig. 8. HS-TCP $cwnd$ time histories following startup of a second flow. RTT is 42ms (top), 162ms(middle) and 324ms (bottom). Bottleneck bandwidth 250Mbit/sec, queue size 20% BDP.

following startup of a second flow. The equal sharing achieved between the two competing flows is evident.

VI. Related Work

Performance measurements are included in many papers proposing modifications to the TCP congestion control algorithm and we briefly mention here the main studies relevant to the present paper. In [10], Kelly presents an experimental comparison of the aggregate throughput performance of Scalable-TCP and standard TCP. In [9], Low and co-authors present throughput and packet loss measurements from a lab-scale test network for FAST-TCP, HS-TCP, Scalable-TCP, BIC-TCP and TCP-Reno. Only throughput measurements that are an aggregate over all flows are presented, thus preventing the fairness of the TCP algorithms from being evaluated; only a single queue size is used and network convergence time is not considered. In [8], aggregate throughput measurements are presented for FAST-TCP and TCP Reno. In [7], throughput and $cwnd$ time histories of FAST-TCP, HS-TCP, Scalable-TCP and TCP Reno are presented for a lab-scale experimental testbed. Aggregate throughput, throughput fairness (measured via Jain's index) and a number of other measures are presented, but only for an 800Mb/s bottleneck bandwidth setting and 2000 packet queue. In [16], NS simulation results are presented comparing the performance of HS-TCP, Scalable-TCP, BIC-TCP and standard TCP.

We note that the foregoing papers all propose changes to the TCP congestion control algorithm and thus present performance measurements in support of these changes. While the design of congestion control algorithms for high-speed networks has been the subject of considerable interest, the evaluation of competing proposals per se has received far less attention. Notably, [2], [4] present evaluation studies specifically targeted at measuring the performance of TCP proposals. Experimental measurements are presented for Scalable-TCP, HS-TCP, FAST-TCP, H-TCP, BIC-TCP, HSTCP-LP and P-TCP (i.e. 16 parallel standard TCP flows) over network paths within the U.S and between the U.S and Europe. Measurements presented include aggregate throughput and throughput fairness (via Jain's index). Convergence time and the impact of queue provisioning are not considered.

In all of the experimental tests noted above, no attempt is made to control for changes to the Linux network stack implementation that are unrelated to the congestion control algorithm.

VII. Summary and Conclusions

In this paper we present experimental results evaluating the performance of the Scalable-TCP, HS-TCP, BIC-TCP, FAST TCP and H-TCP proposals.

We find that many recent proposals perform surprisingly poorly in even the most simple test, namely achieving fairness between two competing flows in a dumbbell topology with the same round-trip times and shared bottleneck link. Specifically, both Scalable-TCP and FAST TCP exhibit very substantial unfairness in this test.

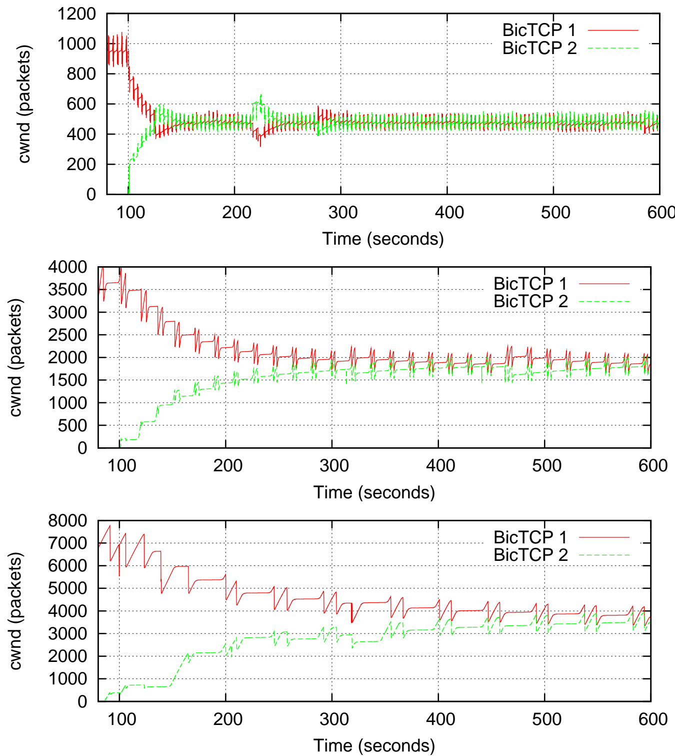


Fig. 9. BIC-TCP $cwnd$ time histories following startup of a second flow. RTT is 42ms (top), 162ms(middle) and 324ms (bottom). Bottleneck bandwidth is 250Mbit/sec, queue size 20% BDP.

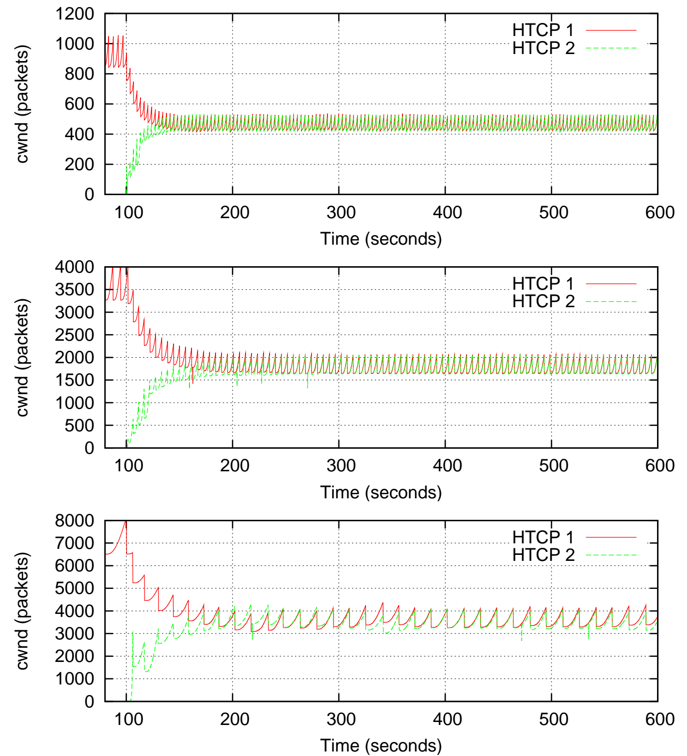


Fig. 10. H-TCP $cwnd$ time histories following startup of a second flow. RTT is 42ms (top), 162ms (middle) and 324ms (bottom). Bottleneck bandwidth is 250Mbit/sec, queue size 20% BDP.

References

[1] M.Allman, TCP Congestion Control with Appropriate Byte Counting (ABC). IETF RFC 3465, February 2003.

[2] H. Bullot, R.L. Cottrell, R. Hughes-Jones, Evaluation of Advanced TCP Stacks on Fast Long Distance Production Networks. J.Grid Comput, 2003.

[3] D.M. Chiu, R. Jain, Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. Computer Networks and ISDN Systems, 1989.

[4] R.L. Cottrell, S. Ansari, P. Khandpur, R. Gupta, R. Hughes-Jones, M. Chen, L. MacIntosh, F. Leers, Characterization and Evaluation of TCP and UDP-Based Transport On Real Networks. . Proc. 3rd Workshop on Protocols for Fast Long-distance Networks, Lyon, France, 2005.

[5] S.Floyd, K.Fall, Promoting the use of end-to-end congestion control in the internet. IEEE/ACM Transactions on Networking, August 1999

[6] S.Floyd, HighSpeed TCP for Large Congestion Windows . Sally Floyd. IETF RFC 3649, Experimental, Dec 2003.

[7] C. Jin, D.X. Wei, S,H. Low, FAST TCP: motivation, architecture, algorithms, performance. Proc IEEE INFOCOM 2004.

[8] C. Jin, D. X. Wei, S. Low, G. Buhrmaster, J. Bunn, D. H. Choe, R. L. A. Cottrell, J. C. Doyle, W. Feng, O. Martin, H. Newman, F. Paganini, S. Ravot, S. Singh, FAST TCP: From Theory to Experiments. IEEE Network, 19(1):4-11, 2005

[9] S. Hegde, D. Lapsley, B. Wydrowski, J. Lindheim, D.Wei, C. Jin, S. Low, H. Newman, FAST TCP in High Speed Networks: An Experimental Study. Proc. GridNets, San Jose, 2004.

[10] T. Kelly, On engineering a stable and scalable TCP variant, Cambridge University Engineering Department Technical Report CUED/F-INFENG/TR.435, June 2002.

[11] D.J.Leith, Linux implementation issues in high-speed networks. Hamilton Institute Technical Report, 2003, www.hamilton.ie/net/LinuxHighSpeed.pdf.

[12] D.J.Leith, R.N.Shorten, H-TCP Protocol for High-Speed Long-Distance Networks. Proc. 2nd Workshop on Protocols for Fast Long Distance Networks. Argonne, Canada, 2004.

[13] Y.T.Li, D.J.Leith, BicTCP implementation in Linux kernels. Hamilton Institute Technical Report, 2004, www.hamilton.ie/net/LinuxBicTCP.pdf.

[14] J. Padhye, V. Firoiu, D.F. Towsley, J.F. Kurose, Modeling TCP Reno performance: a simple model and its empirical validation. Proc. SIGCOMM 1998 (Also IEEE/ACM Transactions on Networking, 2000).

[15] R.N.Shorten, D.J.Leith,J.Foy, R.Kilduff, Analysis and design of congestion control in synchronised communication networks. Automatica, 2004.

[16] L. Xu, K. Harfoush, I. Rhee, Binary Increase Congestion Control for Fast Long-Distance Networks. Proc. INFOCOM 2004

TCP Protocol	Parameters
HS-TCP	High_P=1^{-7}, Low_Window=31 and High_Window=$83,000$
Scalable-TCP	$\alpha = 0.01$, $\beta = 0.875$ and Low_Window=16
H-TCP	$\Delta_L = 1sec$, $\Delta_B = 0.2$
BIC-TCP	$S_{max} = 32$, $B = 4$, smooth_part=20, $\beta = 0.8$ Low_Util=15%, Util_Check=2 and Low_Window=14
FAST-TCP	$\gamma = 50$, m0a=8, m1a=20, m2a=200 m0u=1500, m1l=1250, m1u=15000 and m2l=12500

TABLE II

Default New-TCP Parameters Used In All Tests.