

An Independent H-TCP Implementation under FreeBSD 7.0 – Description and Observed Behaviour

Grenville Armitage,
Lawrence Stewart
Swinburne University of
Technology
Melbourne, Australia
garmitage@swin.edu.au
lastewart@swin.edu.au

Michael Welzl
University of Innsbruck
Austria
michael.welzl@uibk.ac.at

James Healy
Swinburne University of
Technology
Melbourne, Australia
jhealy@swin.edu.au

ABSTRACT

A key requirement for IETF recognition of new TCP algorithms is having an independent, interoperable implementation. This paper describes our BSD-licensed implementation of H-TCP within FreeBSD 7.0, publicly available as a dynamically loadable kernel module. Based on our implementation experience we provide a summary description of the H-TCP algorithm to assist other groups build further interoperable implementations. Using data from our live testbed we demonstrate that our version exhibits expected H-TCP behavior, and describe a number of implementation-specific issues that influence H-TCP's dynamic behavior. Finally, we illustrate the actual collateral impact on path latency of using H-TCP instead of NewReno. In particular we illustrate how, compared to NewReno, H-TCP's *cwnd* growth strategy can cause faster fluctuations in queue sizes at, yet lower median latency through, congestion points. We believe these insights will prove valuable predictors of H-TCP's potential impact if deployed in consumer end-hosts in addition to specialist, high-performance network environments.

Categories and Subject Descriptors

C.2.1 [Computer-Communication Networks]: Network Architecture and Design—*Network communications*; C.2.2 [Computer-Communication Networks]: Network Protocols—*Protocol verification*; C.4 [Computer Systems Organization]: Performance of Systems—*Measurement techniques*

General Terms

Experimentation, Validation, Measurement, Performance

Keywords

H-TCP, TCP, FreeBSD, Congestion control

1. INTRODUCTION

From the perspective of basic connectivity, the key innovation underlying today's internet is the connectionless, destination-based forwarding of packets. The internet protocol (IP) layer sits at the thin waist of most people's hour-glass view of internetworking, providing an abstract layer for common routing and addressing that spans a multitude of different link layer technologies.

However, credit for the internet's wide-spread utility over the past 25+ years must also be given to the transmission control protocol (TCP) [1]. The relatively modern *NewReno* variant of TCP [2] balances two key goals: Provide reliable transfer of byte-streams across the IP layer's unpredictable packet-based service, and minimise congestion inside end hosts and the underlying IP network(s) while maximising performance [3]. TCP is the dominant transport protocol for internet-based applications [4], so the latter goal has been an active and challenging area for academic and industry research into congestion control (CC) techniques [5].

In recent years a number of new CC algorithms have been proposed, simulated and implemented (such as *HS-TCP* [6], *H-TCP* [7] [8], *CUBIC* [9], *CompoundTCP* [10], and *FAST* [11]). Each provide different approaches to inferring the level of, and appropriate dynamic response to, congestion within an IP network. Some have reportedly [12] been deployed operationally prior to full, independent evaluation by the IETF (internet engineering task force) or IRTF (internet research task force).

IETF standards-track progression typically requires multiple independent and interoperable implementations be successfully created from a published specification. In July 2007 the IETF's Transport Area has also requested the assistance of the IRTF's internet congestion control research group (IC-CRG [13]) to evaluate the dynamic behavior of new TCP CC proposals [14]. The IRTF's transport modeling research group (TMRG [15]) is developing metrics and test scenarios by which new CC schemes may be compared in a consistent and independent manner.

To hopefully assist IETF and IRTF efforts, this paper reports on our development of a publicly-available, BSD-licensed implementation of H-TCP within FreeBSD 7.0 [16]. H-TCP was initially designed by the Hamilton Institute (Ireland), and implemented in ns-2 [17] and the Linux kernel. Our implementation derives solely from our reading of the published literature on H-TCP and limited email exchanges with the H-TCP authors. We did not look at any of the Linux or ns-2 source code for H-TCP during development of our FreeBSD implementation.

Section 2 begins with an over-view of issues surrounding TCP CC research. Based on our implementation experience section 3 summarises the H-TCP algorithm to assist other groups build further interoperable implementations. Section 5 describes our testbed, instrumented to allow precise tracking of various TCP parameters before, during and af-

ter congestion events. In section 6 we discuss a number of lessons and trade-offs, and illustrate the H-TCP behaviour shown by our FreeBSD implementation.

In section 7 we illustrate the interesting collateral impact on RTT of using H-TCP instead of NewReno. In particular we show how unrelated UDP flows sharing a congestion point with a long-lived TCP flow may experience lower net latency, but higher levels of latency variation, when H-TCP is used instead of NewReno.

We believe these insights will prove valuable predictors of H-TCP's potential impact if deployed in consumer end-hosts (rather than simply being limited to specialist, high-performance network environments).

2. BACKGROUND

This paper's main focus is on providing a useful description of our H-TCP implementation that is shown to behave properly under FreeBSD 7.0. In this section we shall first summarise the broader context surrounding our work.

2.1 Congestion control

In the early 1980s TCP flow control aimed to send packets only as fast as the receiving host could buffer and consume them (RFC896 [18]). By the late 1980s TCP had been augmented with "slow start" and "congestion avoidance" behaviors to also protect the underlying network itself [19]. TCP congestion control today is a balancing act between the need to get data across the network as promptly as possible while causing minimal disruption to other traffic flows sharing points of potential congestion. TCP needs to be aggressive, yet not too aggressive, in a community where few people entirely agree on what 'too aggressive' really means. Furthermore, TCP must rely on indirect and imprecise indications of congestion within the underlying IP network (such as packet loss, or packet marking by routers along the path) without knowing what any other TCP flow is doing at any given instant in time.

Over the past decade NewReno (the closest we currently come to a 'standard' TCP) has supported an increasingly diverse mix of end-to-end applications running over extremely heterogeneous network paths. Limitations in NewReno's dynamic behavior have given rise to multiple newer CC algorithms, each optimised for particular real-world scenarios considered important by their developers.

Naturally, there is room for debate between proponents of newer CC algorithms. One might insist that a new CC algorithm exhibit good utilization of the bottleneck link in a high bandwidth \times delay product environment, be reactive to congestion and fair to other flows of its own kind as well as (reasonably) fair towards standard TCP ("TCP-friendly"). However, the behavior exhibited by any particular CC algorithm can depend crucially on the chosen test case and metrics.

The TMRG are developing a set of public baseline test scenarios and metrics with which to compare new CC algorithms. Their first "round-table" meeting on TCP evaluation at Caltech on November 8 - 9, 2007 resulted in an interesting discussion paper at PFLDnet 2008 [20]. This work is on-going, and can be tracked on the TMRG website [15].

2.2 Simulation and implementation

Published studies of TCP behaviour in the past decade have been dominated by simulation studies, often using the

ns-2 network simulator [17]. Simulation is attractive because researchers can explore basic CC ideas without having to gain access to, or understanding of, the internals of real operating systems. Simulations are also an attractive way to obtain finely-grained, *microscopic* level details about a CC algorithm's dynamic behavior.

However, simulations inherently involve simplifications. We cannot truly understand how new CC algorithms will behave unless implemented within real-world operating systems and evaluated on real networks. Only then can we begin to understand the impact of new CC algorithms on the network and other traffic.

A number of studies and testbeds are attempting to provide real-world insights (cf. [21, 22]). However, instrumentation is often limited to providing a relatively *macroscopic* view over time of the CC algorithms under test.

The Linux-based TCP research community often utilises *Web 100* [23, 24] to instrument their end hosts and track TCP state changes. However, Web 100 relies on regular polling of kernel-resident TCP state (such as the congestion window, *cwnd*). This imposes a trade-off between measurement granularity and system performance. (It is possible to poll in such a way that little information is lost, but the adverse effect on machine speed may be significant). In contrast, simulations entirely avoid the performance issues associated with high-speed, finely-grained tracing of TCP state variables in real operating systems.

2.3 Instrumenting FreeBSD 7.0

Our independent H-TCP implementation leverages additions to FreeBSD 7.0 created by two of the authors (Stewart and Healy).

First, we utilised SIFTR (statistical information for TCP research) [25], developed and released to the public during 2007. SIFTR is an event-driven (rather than polled) system for logging changes in TCP state variables within the FreeBSD 7.0 kernel. SIFTR performs well under realistic network load, and provided us with a precise view of how variables such as *cwnd* changed (or didn't change) with every packet emitted or received by our TCP end points.

Second, we implemented (and publicly released at the end of 2007) a kernel patch for FreeBSD 7.0 that allows new TCP congestion control algorithms to be added in a modular fashion [26].

Our modular CC framework abstracts FreeBSD's congestion control behavior into a discrete set of functions called from within the TCP stack. FreeBSD's default CC algorithm (NewReno) was extracted into a module and left as the default. However, we now had the ability to add new modules dynamically and switch between them on a per connection basis.

(We anticipate that the framework will be available in the FreeBSD CVS tree in the coming months, and in FreeBSD 8.0 in approximately 18 months time. Development work can be tracked in our FreeBSD Perforce branch [27].)

3. REVIEW OF THE H-TCP ALGORITHM

The core of H-TCP is currently (as of January 2008) specified in an Internet draft [7]. A number of related papers by Doug Leith and his colleagues at the Hamilton Institute¹

¹In the rest of this section we will use 'Hamilton' when referring to the group who developed H-TCP.

describe further refinements to H-TCP [28, 29, 30]. This section summarises our understanding of the attributes that define the core of H-TCP, and the refinements proposed by Hamilton.

3.1 Core H-TCP Algorithm

It is common practice in TCP research to generalise Standard TCP² by specifying *cwnd* growth and contraction in terms of two constants, “alpha” (α) and “beta” (β)³. Given SMSS (the sender’s maximum segment size) we can more precisely state that during congestion avoidance phase *cwnd* grows by at most $\alpha \times SMSS$ each round trip time (RTT), and after a congestion event *cwnd* is reduced to $\beta \times cwnd$.

During congestion avoidance a common approximation based on [3] is to grow *cwnd* according to Equation 1 every time an acknowledgment arrives.

$$cwnd = \alpha \times \frac{(SMSS \times SMSS)}{cwnd} \quad (1)$$

Standard TCP uses fixed values of $\alpha = 1$ and $\beta = 0.5$.

In version -04 of Hamilton’s internet draft the defining characteristic of H-TCP is the novel way in which α is varied while in congestion avoidance. In particular, H-TCP increases α based on the time (δ) since the last congestion event.

H-TCP’s initial response to a congestion event is to behave like NewReno, setting $\alpha = 1$ until δ exceeds a configurable threshold δ_l . Once δ exceeds δ_l , α is given by the following quadratic:

$$\alpha = 1 + 10 \times (\delta - \delta_l) + ((\delta - \delta_l)/2)^2 \quad (2)$$

δ_l represents the time that must elapse after a congestion event before H-TCP’s own α increase function comes into effect. Once δ_l is exceeded, *cwnd* growth is scaled by α calculated according to Equation 2.

Hamilton currently recommends that $\delta_l = 1$ second. Thus for one second after every congestion event, H-TCP mimics NewReno-style growth of *cwnd*. Beyond one second H-TCP’s *cwnd* grows more aggressively the longer we go without another congestion event.

Hamilton’s own Linux implementation of H-TCP includes extensions beyond the core algorithm described above. They include *RTT scaling* and *Adaptive backoff*, to address various issues likely to influence H-TCP flows. To differentiate from the *core* H-TCP definition we refer to the augmented Linux implementation as *defacto* H-TCP. Both extensions are briefly discussed next.

3.2 RTT scaling

Standard TCP exhibits *cwnd* growth that is inversely proportional to RTT due to faster acknowledgment clocking. It is therefore inherently unfair between competing flows having different RTTs yet traversing a common bottleneck. This sort of unfairness in the context of H-TCP is discussed in [28] and [29].

Hamilton’s RTT scaling attempts to make the growth of *cwnd* largely invariant to path RTT.

²By *Standard TCP* here we refer to NewReno [2], possibly coupled with refinements, such as SACK [31], which do not significantly modify NewReno’s ‘additive increase multiplicative decrease’ (AIMD) CC behavior.

³ α can be considered TCP’s *additive increase* parameter, while β is the *multiplicative decrease* parameter.

During congestion avoidance the core H-TCP algorithm’s α from Equation 2 (now termed α_{raw}) is scaled before being used in the *cwnd* update calculation.

The scale factor ρ is calculated as:

$$\rho = RTT_{flow}/RTT_{ref} \quad (3)$$

where RTT_{flow} is the current estimate of RTT between sender and receiver, and RTT_{ref} is a common reference RTT.

α is then calculated as follows:

$$\alpha_{raw} = 1 + 10 \times (\delta - \delta_l) + ((\delta - \delta_l)/2)^2 \quad (4)$$

$$\alpha = \max(1, (\rho \times \alpha_{raw})) \quad (5)$$

Equation 5 scales α_{raw} (the α that would be used by core H-TCP), and the *max()* function ensures *cwnd* is always able to grow even when $\rho < 1$

Selection of RTT_{ref} balances fairness and responsiveness. Increasing RTT_{ref} penalises low RTT flows more (thereby increasing fairness), but results in longer congestion epoch duration and thus reduced responsiveness.

Left by itself, Equation 3 would result in RTT scaling being unduly harsh on paths with extremely low RTTs (such as LANs) and unduly generous on paths with extremely high RTTs (such as intercontinental/space WANs). Consequently, Hamilton currently suggest setting $RTT_{ref} = 100ms$ and bounding ρ to the range [0.1, 2]. This limits the penalty or gain for flows exhibiting $RTT \leq 10ms$ or $RTT \geq 200ms$ respectively (in turn bounding the achievable fairness and responsiveness).

3.3 Adaptive backoff

Adaptive backoff seeks to maintain high utilisation of a network path by ensuring the network buffers are never fully empty. Hamilton’s approach modifies the adjustment of α and β after acknowledgment and congestion events. The mathematical proof underpinning their adaptive backoff algorithm can be found in [30] and [28].

Adaptive backoff requires that minimum and maximum RTT estimates (RTT_{max} and RTT_{min}) be maintained during the flow’s lifetime, providing an indirect approximation to the path’s propagation and queuing delays.

On receipt of an acknowledgment:

$$if RTT < RTT_{min}, RTT_{min} = RTT \quad (6)$$

$$if RTT > RTT_{max}, RTT_{max} = RTT \quad (7)$$

$$\beta = RTT_{min}/RTT_{max} \quad (8)$$

$$\alpha = \max(1, (2 \times (1 - \beta) \times \alpha_{raw})) \quad (9)$$

where α_{raw} is derived from Equation 4 (or Equation 2 if you’re implementing adaptive backoff but not RTT scaling).

On congestion:

$$RTT_{max} = RTT_{min} + (7/8)(RTT_{max} - RTT_{min}) \quad (10)$$

$$cwnd = \beta \times cwnd \quad (11)$$

Equation 10 provides a smoothed fading of RTT_{max} as congestion events occur. Equations 8 and 11 ensure *cwnd* is only lightly reduced on congestion if the link appears lightly loaded (i.e. if RTT_{max} is not much higher than RTT_{min}).

Hamilton suggests bounding β to the range [0.5, 0.8], with $\beta = 0.5$ providing Standard TCP behavior. H-TCP does not allow RTT_{min} to ever rise over time (so, for example, RTT_{min} would not react to a topology change that raises the minimum path RTT during a flow’s lifetime).

Hamilton have also proposed a further refinement called *adaptive reset*, which we have not yet evaluated. Adaptive reset overrides adaptive backoff if recent bandwidth estimates indicate the network requires more responsive reaction to congestion events.

4. IMPLEMENTATION EXPERIENCE

In this section we note a number of design choices and issues, and open issues that have transpired during our implementation effort. (Section 6 evaluates our implementation’s *cwnd* growth after congestion events.)

4.1 Interaction with Fast Recovery

Available H-TCP literature did not clearly spell out the relationship between H-TCP and fast recovery (FR) during slow start. Our working assumption is that H-TCP utilises Standard TCP FR behavior when FR is required.

Standard TCP *slowstart* rules govern *cwnd* growth when *cwnd* is below the slow start threshold, *ssthresh*. Traditionally *ssthresh* is set to $0.5 \times cwnd$ on congestion, prior to entering FR. Although not clearly stated in the published H-TCP documentation, an email exchange with Hamilton clarified that for H-TCP, *ssthresh* should be set to $\beta \times cwnd$ on entry to FR (which is $0.5 \times cwnd$ if implementing only core H-TCP).

We do not change the Standard TCP behavior that *cwnd* is set to *flightsize* + *SMSS* if the amount of inflight data is less than *ssthresh* when exiting FR.

Regardless of how α is being calculated (core H-TCP, RTT scaling, adaptive backoff or some combination) we utilise Standard TCP’s slow start algorithm when *cwnd* < *ssthresh*, even if $\alpha > 1$.

4.2 Algorithm-specific CC data

Our H-TCP module implements additional per-flow state in the TCP control block. New variables α and δ are required for *core* H-TCP, and β , RTT_{min} and RTT_{max} are required for *defacto* H-TCP extensions.

δ_l is set to one as a global constant in our H-TCP module.

An additional variable, *prev_cwnd*, is used to record *cwnd* before entering FR, and is used to restore *cwnd* once we exit FR. (FreeBSD’s existing FR code modifies *cwnd* in the control block for its own purposes during the FR phase.)

4.3 Initialising and recalculating H-TCP state

Algorithm-specific variables are initialised as shown in Listing 1. This ensures our H-TCP is functionally equivalent to Standard TCP for the first δ_l seconds.

Listing 1 Initialisation of H-TCP specific CC variables

```

 $\alpha = 1$ 
 $\beta = 0.5$ 
 $\delta = current\_time$ 
 $RTT_{min} = 0$  (until 8 RTT updates have occurred)
 $RTT_{max} = 0$ 

```

We piggyback updates onto the acknowledgment processing data path, so RTT_{min} , RTT_{max} , β and α are all recalculated ready for use after each ACK received.

RTT_{min} is handled as a special case at the beginning of a new TCP connection. FreeBSD’s standard RTT estimator appeared to require a couple of packet exchanges before the current RTT estimate became reasonable. Consequently, we wait for 8 RTT updates to occur before setting $RTT_{min} = RTT$ (i.e. non-zero). Equation 6 applies after that point.

(Our code allows RTT scaling and/or adaptive backoff to be turned off, which modifies or eliminates processing of RTT_{min} , RTT_{max} and β as appropriate.)

4.4 Fixed-point calculation of alpha

The absence of floating-point operations makes Equation 2 non-trivial to perform within the FreeBSD kernel. Listing 2 shows our C macro that implements a reasonable approximation to α using fixed-point arithmetic. Call this $\alpha_{fixedpoint}$

Listing 2 Calculating α with fixed-point arithmetic

```

#define ALPHA_SHIFT 4
#define HTCP_CALC_ALPHA(diff) \
( ( (16) + \
  ((160 * (diff)) / hz) + \
  (((diff) / hz) * \
  (((diff) << ALPHA_SHIFT) / (4 * hz))) \
) >> ALPHA_SHIFT )

```

The macro’s parameter *diff* is the term $\delta - \delta_l$ from Equation 2. Because we count time δ in terms of FreeBSD kernel ‘ticks’ the macro utilises *hz* (the kernel-wide constant indicating the kernel’s tick rate per second) to convert back to real time. Using *hz* also ensures the macro works regardless of what kernel-wide tick rate may have been selected at boot time.

Shifting all values by ALPHA_SHIFT reduces the truncation of intermediate results due to integer division. The initial “16” value is the “1” term in Equation 2 shifted up by ALPHA_SHIFT bits (i.e. multiplied by 16, given that ALPHA_SHIFT is currently defined as 4). Likewise, the “160” value is the “10” multiplier in Equation 2 itself multiplied by 16. The result is down-shifted by ALPHA_SHIFT to bring it back into the correct range.

(Pre-computing the scaled forms of Equation 2’s “1” and “10” terms speeds up the macro. However, these terms must be re-calculated if you change ALPHA_SHIFT.)

Our macro cannot take an arbitrary value for *diff*, lest the middle term $((160 * diff) / hz)$ overflows. We constrain *diff* to be less than the max size of an unsigned long divided by the constant 160 figure, i.e.

$$diff < [(2 \sim (\text{sizeof}(\text{u_long}) * 8)) - 1] / 160$$

With 32-bit unsigned longs we can support *diff* greater than 24 hours, which we consider to be more than sufficient.

Figure 1 plots both $\alpha_{fixedpoint}$ and Equation 2’s ‘theoretical’ floating-point α for *diff* up to 5 seconds (that is, $\delta > 6$ given that δ_l is 1). Given the rounding down in our macro, $\alpha_{fixedpoint}$ usually slightly *underestimates* the theoretical value for α . Thus our H-TCP is never more aggressive than theoretical H-TCP behavior.

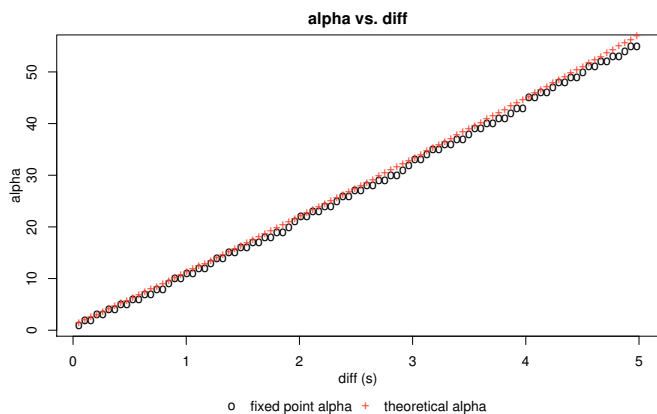


Figure 1: Fixed-point and theoretical alpha vs $(\delta - \delta_t)$

4.5 Open issues

A number of questions remain around our current design choices, and their impact on processing load or H-TCP behavior performance.

Are we updating RTT_{max} , RTT_{min} , β and α more frequently than necessary (by doing it on receipt of every ACK)?

We currently rely on the FreeBSD kernel’s existing smoothed RTT estimator, based on RFC1323 and RFC 2988 [32, 33]. Future work is required to properly characterise the range of errors this might introduce into H-TCP’s RTT scaling and adaptive backoff.

We currently calculate δ prior to entering FR, as this is the time at which we know congestion occurred. However, the time spent in FR thus accrues as time since congestion. For large RTT paths this could potentially lead to interesting side effects where we exit fast recovery with $\alpha > 1$, and immediately begin aggressive $cwnd$ growth.

We treat RTO firing as an indication of likely congestion. However, in case the first firing was a false alarm we set δ on the second firing of RTO. It may be necessary to revisit this decision.

Finally, our work showed up some inconsistencies between the various published works relating to H-TCP. Beneficially, we have provided feedback to Hamilton and demonstrated that functional H-TCP implementation can indeed be created using the public material. However, errors may have also crept into our own understanding.

5. EXPERIMENTAL METHODOLOGY

Although hardly representative of normal network complexity, the traditional *dumbbell* topology in Figure 2 is quite suitable for this paper’s goals. First, we need to show that our H-TCP implementation’s dynamic behavior in the face of congestion equals that of H-TCP behavior published elsewhere. Second, we aim to measure and compare the changes in latency through a congested node when using H-TCP rather than NewReno.

Hosts A, B, C and D run FreeBSD 7.0-RC1 on a 1.86GHz Intel Core2 Duo E6320 (4MB L2 Cache) with 1GB PC5300 DDR2 RAM and Intel PRO/1000 GT 82541PI PCI gigabit Ethernet interfaces. These hosts are instrumented with SIFTR [25], enabling precise tracking of $cwnd$ over the lifetimes of active TCP sessions.

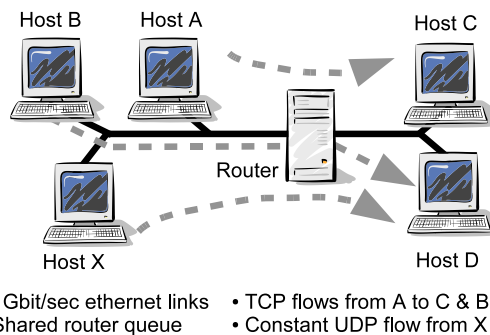


Figure 2: Simple dumbbell topology for FreeBSD H-TCP validation and latency measurements across a congested router

The router runs FreeBSD 7.0-RC1 on a 2.80GHz Intel Celeron D (256K L2 Cache), 512MB PC3200 DDR-400 RAM, with two Intel PRO/1000 GT 82541PI PCI gigabit Ethernet cards as forwarding interfaces.

We utilise *dummynet* [34] to provide configurable latency and bandwidth limits through the router. The router’s kernel ticks 2000 times per second ($kern.hz = 2000$) for fine-grained responsiveness and latency resolution of 0.5ms. In addition, we patched *dummynet* to log changes in internal queue size on a per-packet basis.

An Endace 3.7GF gigabit ethernet traffic capture card (not shown in Figure 2) timestamps packet arrivals and departures on both sides of the router. This allows us to calculate one way delay (OWD) through the router for the trials in section 7.

Host X is a SmartBits 2000 traffic generator, sending 200-byte UDP/IP packets to Host D at 20ms intervals during the latency trials in section 7. By monitoring the UDP packets from host X to host D, OWD across the router is thus sampled approximately every 20ms, regardless of the TCP traffic flowing at any given instant.

6. VERIFYING CORE H-TCP BEHAVIOR

A key step in verifying our FreeBSD H-TCP implementation is showing that $cwnd$ growth in *core* H-TCP mode behaves as expected.

Figure 3 compares the growth of $cwnd$ for 2.5 seconds after a congestion event when using core H-TCP, defacto H-TCP (RTT scaling and adaptive backoff) and regular NewReno over our testbed in Figure 2. A single flow ran from host A to host C, with 80ms configured as the RTT (40ms each way through the router).

As expected, $cwnd$ grows more aggressively with the core H-TCP algorithm than defacto H-TCP, and NewReno is less aggressive than either form of H-TCP. The behavior of $cwnd$ exhibited by our FreeBSD H-TCP is also consistent with previously published analyses of H-TCP.

Figure 4 compares the growth of $cwnd$ for 2.5 seconds after a congestion event for our implementation of core H-TCP and Hamilton’s ns-2 implementation of H-TCP⁴. We configured ns-2 to simulate the same testbed configuration used for Figure 3. The dynamic growth of $cwnd$ tracks very

⁴We ported Hamilton’s ns-2.26 code (available from <http://www.hamilton.ie/net/research.htm>) to ns-2.31 [35]

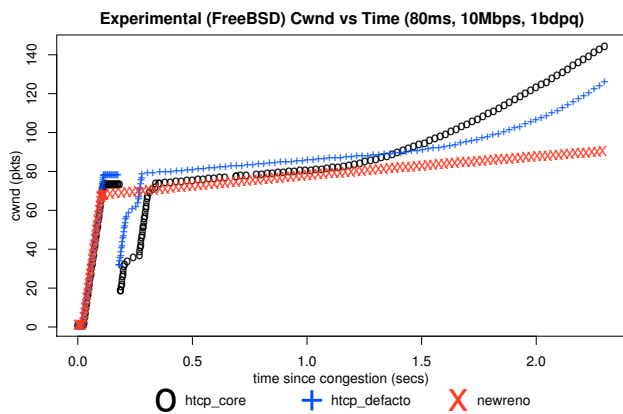


Figure 3: Core H-TCP, H-TCP with RTT scaling and NewReno *cwnd* since last congestion event

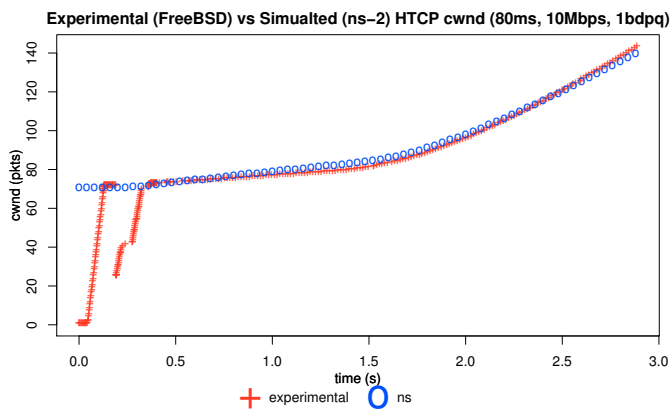


Figure 4: FreeBSD core H-TCP and Hamilton’s ns-2 H-TCP *cwnd* since last congestion event

closely⁵. It seems reasonable to conclude our implementation’s behaviour matches Hamilton’s intentions.

7. IMPACT OF H-TCP ON PATH LATENCY

CC algorithms used by consumer end hosts will increasingly share congestion points with latency-sensitive traffic (such as home ‘broadband routers’ supporting internet-based VoIP and multiplayer game applications). It is not intrinsically noteworthy that TCP flows can create an increase in average queue length at congestion points. However, the *dynamic nature* of the resulting queue fluctuations is interesting as it impacts on unrelated application flows. This is particularly true for CC algorithms that can exhibit ‘faster than NewReno’ growth for *cwnd* (or include path latency in their congestion-estimation feedback loop).

Rather than simply duplicate previous work comparing H-TCP’s performance against various other CC variants [8], here we present some initial observations regarding the impact of H-TCP on the latency experienced by an unrelated

⁵Delayed ACKs were enabled in the FreeBSD H-TCP stack and disabled in the ns-2 simulation. We are still reviewing this minor discrepancy at the time of writing. Delayed ACKs were used in all empirical measurements involving FreeBSD H-TCP.

UDP flow sharing a single congestion point. The impact on ‘innocent bystanders’ (such as VoIP or game traffic sharing a congestion point) is not yet covered by existing TMRG guidelines for evaluating CC algorithms. To provide some context for our results we compare H-TCP with the latency fluctuations that would be caused by New Reno under the same circumstances. We hope that this preliminary observation of H-TCP’s impact will encourage similar empirical analyses involving other non-New Reno CC algorithms in the future.

7.1 Latency measurement testbed

Two trial scenarios were run on the testbed of Figure 2 - *low speed* (router configured for 1Mbit/sec rate limit) and *moderate speed* (router configured for 10Mbit/sec rate limit). In each case the router provided a one way delay (OWD) of 40ms in each direction. (These figures are rough approximations for someone serving data from behind a consumer broadband connection to other sites within a modest country or state region. In much of the western world 1Mbit/sec uplink speeds are unusual. Nevertheless certain countries are trending towards deployment of consumer broadband services with multi-megabit/sec uplinks.)

The bandwidth-delay product (BDP) of the unloaded path was 100000 bytes (80ms × 10Mbit/sec) and 10000 bytes (80ms × 1Mbit/sec) for the moderate and low speed cases respectively. Some trials had the router’s dummynet queue set (in bytes) to one BDP, other trials used 0.25 BDP.

Non-overlapping bulk data transfers of 60 second duration were initiated from Hosts A and B toward C and D using H-TCP and NewReno. The destination hosts (C and D) were configured to advertise a receive window of 3 BDP (300000 and 30000 bytes for the moderate and low speed scenarios respectively). Other details are as stated in section 5.

For clarity the Figures in this section will focus on representative subsets of time from each trial, rather than the full 60 seconds.

7.2 OWD due to H-TCP and NewReno

7.2.1 H-TCP with one flow

Figure 5 shows the variation over time of the router’s internal queue size, and the OWD experienced by UDP packets from Host X to Host D, when a single flow of H-TCP traffic transits the router with a 10Mbit/sec rate limit. The queue was capped at 1 BDP. OWD takes ≈2.5 seconds to grow from 40ms (the configured minimum) to 120ms (40ms plus the peak queuing delay) and collapse back to 40ms.

Figure 6 shows the same trial as Figure 5, but showing the relationship between *cwnd* and router queue size over time. (Under the same test conditions NewReno showed a cyclical queue growth pattern, but taking ≈10 seconds to grow from 40ms to 120ms and then collapse back to 40ms.)

Figure 6 shows a consistent relationship between the growth of *cwnd* and the growth in queue size at the router. This is to be expected - as *cwnd* grows the sender launches packets into the network even more frequently, increasing the nett backlog of packets in the router queue.

One aspect of Figure 6 may seem somewhat surprising at first sight: at the end of the FR phase, a sender should set *cwnd* to *ssthresh*, which should be half of the *cwnd* before entering this phase. However, Figure 6 shows that *cwnd* is

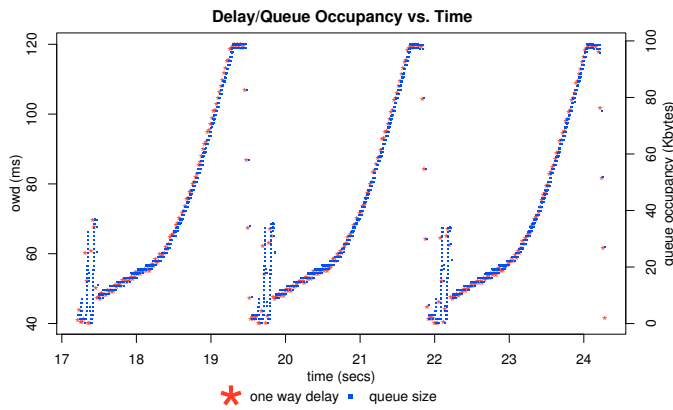


Figure 5: OWD and queue size vs time - 1 H-TCP flow, 1 UDP flow, 1 BDP queue size, 10Mbit/sec

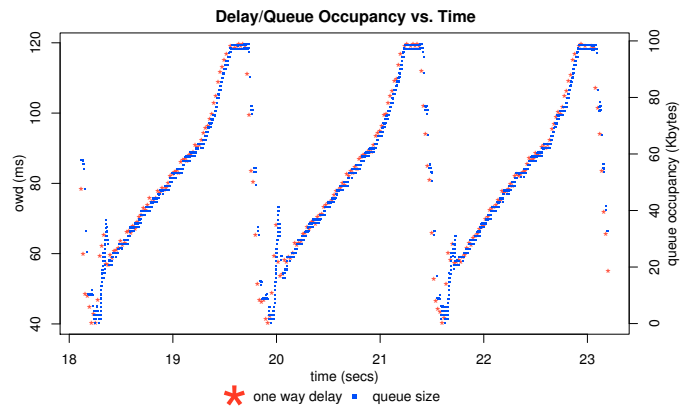


Figure 7: OWD and queue size vs time - 4 H-TCP flows, 1 UDP flow, 1 BDP queue size, 10Mbit/sec

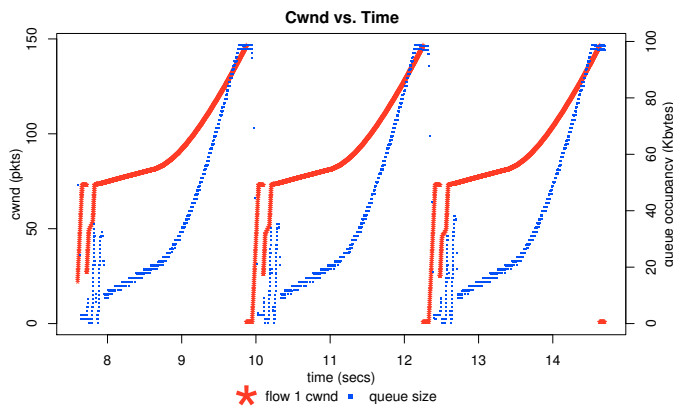


Figure 6: *cwnd* and queue size vs time - 1 H-TCP flow, 1 UDP flow, 1 BDP queue size, 10Mbit/sec

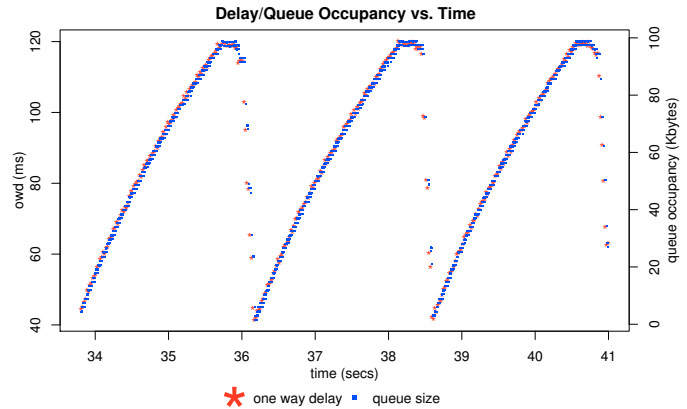


Figure 8: OWD and queue size vs time - 4 NewReno flows, 1 UDP flow, 1 BDP queue size, 10Mbit/sec

set to a very small value and quickly increased up to half the previous value of *cwnd* (*ssthresh*) after the FR phase.

This is caused by a mechanism for preventing so-called “microbursts” in the FreeBSD TCP implementation. To prevent the queue from drastically growing for a short period by setting *cwnd* directly to *ssthresh*, *cwnd* is updated in a smoother manner. Several algorithms for doing this were proposed in [36]. In FreeBSD, a (sort of) slow start behavior is implemented, the only difference to RTO-initiated slow start being that there are not zero but *ssthresh* packets in flight when slow start begins.

This speeds things up significantly. For instance, starting with *cwnd* = one packet, it takes an RTT for *cwnd* to grow to 2, another RTT for it to grow to 4 and so on. If, however, slow start begins with *ssthresh* packets in flight, it takes just one RTT for *cwnd* to reach *ssthresh*. In comparison with setting *cwnd* directly to *ssthresh*, this behavior is less bursty on a very short timescale.

7.2.2 H-TCP and NewReno with four flows

Figure 7 shows the OWD variation when four independent H-TCP flows running concurrently through the router with a 10Mbit/sec rate limit and 1 BDP queue size. Two flows from A to C, two flows from B to D. Queue growth is more rapid

(driven by the higher aggregate number of packets in flight), shortening the time between congestion events. OWD now takes ≈ 1.6 seconds to vary from 40ms to 120ms. (Inspection of the SIFTR logs revealed that cyclic *cwnd* growth and decrease for all four flows was synchronised during this trial.)

For comparison, Figure 8 shows the same system’s behaviour when the end hosts use four NewReno flows rather than H-TCP. The cyclic change in OWD takes ≈ 2.4 seconds, slower than that caused by four H-TCP flows in Figure 7.

7.2.3 H-TCP with 0.25 BDP queue size

Figure 9 shows OWD and queue variation over time for H-TCP with a 10Mbit/sec rate limit, but this time the router’s queue size is reduced to 0.25 BDP. Congestion events occur with a sufficiently small number of packets in flight that our H-TCP implementation struggles to ‘regain its feet’ after exiting fast recovery. Almost 1.5 seconds elapse before *cwnd* growth (and hence queue growth) leaps upward, almost immediately triggering another congestion event. (This isn’t specific to H-TCP. With the queue capped at 0.25 BDP NewReno exhibited a similar inability to grow the queue until roughly 2.5 seconds after a previous congestion event.)

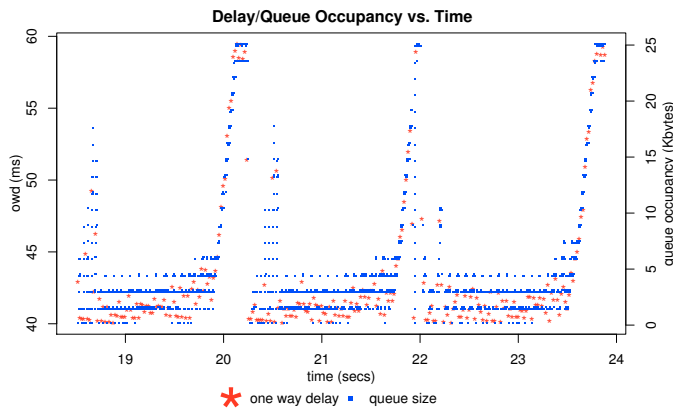


Figure 9: OWD and queue size vs time - 1 H-TCP flow, 1 UDP flow, 0.25 BDP queue size, 10Mbit/sec

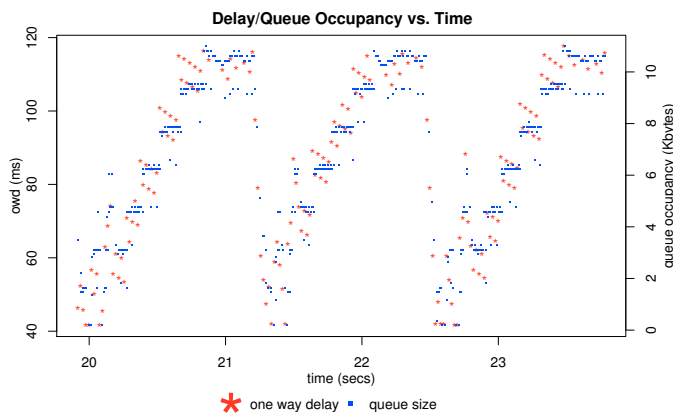


Figure 10: OWD and queue size vs time - 1 H-TCP flow, 1 UDP flow, 1 BDP queue size, 1Mbit/sec

7.2.4 H-TCP with 1Mbit/sec rate limit

Figure 10 shows OWD and queue variation over time for H-TCP through a 1 BDP queue, but this time the router's rate limit is reduced to 1Mbit/sec. Because of the relatively low rate limit, a 1 BDP queue is only a handful of packets long. Given the 80ms base RTT over the path, H-TCP congests the queue in just over a one second, essentially while still in its NewReno phase. The equivalent graph for NewReno at 1Mbit/sec and a 1 BDP queue is not shown because, not surprisingly, it looked virtually identical to Figure 10.

7.2.5 Median OWD versus number of flows

Figure 11 illustrates the impact of queue size and number of flows on the median OWD caused by H-TCP and NewReno.

The difference between H-TCP and NewReno is most notable for 10Mbit/sec rate limit and 1BDP queue size. In the single flow case, the median OWD added by H-TCP is ≈ 21 ms, whereas the median OWD added by NewReno is almost 50ms. This is likely due to H-TCP's more aggressive *cwnd* growth when congestion events occur more than one second apart. Because H-TCP then spends shorter periods of time with the queue close to full, unrelated traffic shar-

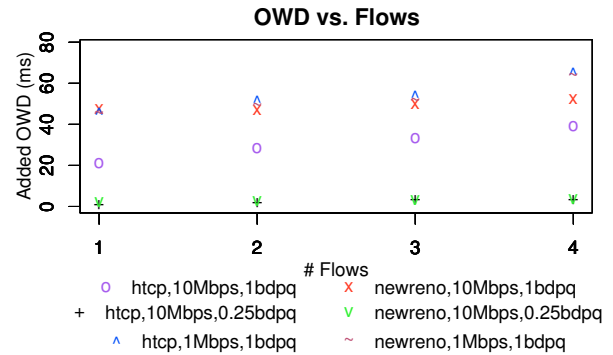


Figure 11: Median OWD versus number H-TCP or NewReno flows, 1 and 0.25 BDP queue size

ing the queue sees less frequent instances of high additional latency. NewReno, by comparison, leaves the queue in a relatively full state for longer as it climbs towards generating a congestion event.

As the number of flows increases, the absolute OWDs added by H-TCP and NewReno at 10Mbit/sec both increase slightly, but the difference between them shrinks. More flows result in more frequent congestion events, and less time spent by H-TCP in its aggressive *cwnd* growth phase.

Trials at 1Mbit/sec and 1 BDP queue size show virtually no difference between the OWD added by H-TCP or NewReno. They both contribute just under 50ms in the one flow case, growing to just over 60ms in the four flow case. As seen in Figure 10, H-TCP spends virtually no time in its more aggressive *cwnd* growth mode, essentially behaving just like NewReno.

Trials at 10Mbit/sec with 0.25 BDP queue size also show virtually no difference between H-TCP and NewReno. This is largely because both H-TCP (as seen in Figure 9) and NewReno spend most of their time between congestion events struggling to grow *cwnd* at all. (And as the queue stays largely unfilled between congestion events, the median OWD added in either case is quite low.)

In other words, if available bandwidth, RTT and a congestion point's queue size together allows H-TCP to spend a lot of time in its aggressive *cwnd* growth phase, H-TCP will cause faster fluctuations in latency through the congestion point than NewReno. But the median increase in latency is likely to be lower than that imposed by NewReno.

8. CONCLUSION AND FURTHER WORK

This paper is our contribution to the community process for evaluation and independent implementation of new congestion control algorithms for TCP. Based on our experience developing the first publicly-available, BSD-licensed implementation of H-TCP within FreeBSD 7.0 [16], we provide a description of H-TCP and associated parameter settings that should assist other implementors. Our independent implementation replicates H-TCP's *cwnd* growth behavior as seen in papers previously published by the original authors of H-TCP.

In addition, we make a preliminary exploration of the impact of H-TCP's modified CC algorithm on latency measured through a congested point in the network. TCP (re-

ardless of CC algorithm) shares the best-effort internet with increasingly popular non-reactive, latency-sensitive applications such as VoIP and online games. Our experiments confirmed that H-TCP can (compared to NewReno) induce faster cycling of queue length in (and hence latency through) a congested router, whilst causing slightly lower increase in median latency. We believe there is clearly room for more research into the impact of other CC algorithms on the dynamic network conditions experienced by unrelated IP-based traffic.

9. ACKNOWLEDGMENTS

This work has been made possible in part by a grant from the Cisco University Research Program Fund at Community Foundation Silicon Valley.

10. REFERENCES

- [1] J. Postel, "Transmission Control Protocol," RFC 793 (Standard), Sep. 1981, updated by RFC 3168. [Online]. Available: <http://www.ietf.org/rfc/rfc793.txt>
- [2] S. Floyd, T. Henderson, and A. Gurtov, "The NewReno Modification to TCP's Fast Recovery Algorithm," RFC 3782 (Proposed Standard), Apr. 2004. [Online]. Available: <http://www.ietf.org/rfc/rfc3782.txt>
- [3] M. Allman, V. Paxson, and W. Stevens, "TCP Congestion Control," RFC 2581 (Proposed Standard), Apr. 1999, updated by RFC 3390. [Online]. Available: <http://www.ietf.org/rfc/rfc2581.txt>
- [4] M. Fomenkov, K. Keys, D. Moore, K. Claffy, "Longitudinal study of Internet traffic in 1998-2003," in *Winter International Symposium on Information and Communication Technologies (WISICT)*, Cancun, Mexico, January 2004. [Online]. Available: http://www.caida.org/publications/papers/2003/nlanr/nlanr_overview.pdf
- [5] S. Floyd, "Congestion Control Principles," RFC 2914 (Best Current Practice), Sep. 2000. [Online]. Available: <http://www.ietf.org/rfc/rfc2914.txt>
- [6] S. Floyd, "HighSpeed TCP for Large Congestion Windows," RFC 3649 (Experimental), Dec. 2003. [Online]. Available: <http://www.ietf.org/rfc/rfc3649.txt>
- [7] D. Leith, R. Shorten, "H-TCP: TCP Congestion Control for High Bandwidth-Delay Product Paths," Hamilton Institute, Tech. Rep., July 2007. [Online]. Available: <http://tools.ietf.org/draft/draft-leith-tcp-htcp/draft-leith-tcp-htcp-04.txt>
- [8] Y.-T. Li, D. Leith, and R. N. Shorten, "Experimental evaluation of TCP protocols for high-speed networks," *IEEE/ACM Trans. Netw.*, vol. 15, no. 5, pp. 1109–1122, 2007.
- [9] I. Rhee, L. Xu and S. Ha, "CUBIC for Fast Long-Distance Networks," North Carolina State University, Tech. Rep., August 2007. [Online]. Available: <http://tools.ietf.org/id/draft-rhee-tcpm-cubic-00.txt>
- [10] M. Sridharan, K. Tan, D. Bansal and D. Thaler, "Compound TCP: A New TCP Congestion Control for High-Speed and Long Distance Networks," Microsoft, Tech. Rep., October 2007. [Online]. Available: <http://www.ietf.org/internet-drafts/draft-sridharan-tcpm-ctcp-01.txt>
- [11] D. W. C. Jin and S. Low, "FAST TCP for High-Speed Long-Distance Networks," Caltech, Tech. Rep., June 2003. [Online]. Available: <http://tools.ietf.org/id/draft-jin-wei-low-tcp-fast-01.txt>
- [12] "[e2e] Are we doing sliding window in the Internet?" January 2008. [Online]. Available: <http://mailman.postel.org/pipermail/end2end-interest/2008-January/007032.html>
- [13] Internet Research Task Force, "Internet Congestion Control Research Group," Accessed 8 Jan 2007. [Online]. Available: <http://www.irtf.org/charter?gtype=rg&group=icrg>
- [14] Internet Engineering Task Force, "Experimental Specification of New Congestion Control," July 2007, Accessed 3 Mar 2008. [Online]. Available: <http://www.ietf.org/IESG/content/ions/ion-tsv-alt-cc.txt>
- [15] Internet Research Task Force, "Transport Modeling Research Group," Accessed 8 Jan 2007. [Online]. Available: <http://www.irtf.org/charter?gtype=rg&group=tmrg>
- [16] J. Healy, L. Stewart, "H-TCP Congestion Control Algorithm for FreeBSD," December 2007. [Online]. Available: <http://caia.swin.edu.au/urp/newtcp/tools/htcp-readme-0.9.txt>
- [17] "The Network Simulator - ns-2," Accessed 19 Nov 2007. [Online]. Available: <http://www.isi.edu/nsnam/ns/>
- [18] J. Nagle, "Congestion control in IP/TCP internetworks," RFC 896, Jan. 1984. [Online]. Available: <http://www.ietf.org/rfc/rfc896.txt>
- [19] R. Braden, "Requirements for Internet Hosts - Communication Layers," RFC 1122 (Standard), Oct. 1989, updated by RFC 1349. [Online]. Available: <http://www.ietf.org/rfc/rfc1122.txt>
- [20] L. Andrew, C. Marcondes, S. Floyd, L. Dunn, R. Guillier, W. Gang, L. Eggert, S. Ha, and I. Rhee, "Towards a Common TCP Evaluation Suite," in *Sixth International Workshop on Protocols for Fast Long-Distance Networks*, Manchester, GB, March 2008. [Online]. Available: http://www.hep.man.ac.uk/PFLDnet2008/paper/08_Lachlan_pfldnet2008.pdf
- [21] 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*, Nara, Japan, March 2006. [Online]. Available: <http://netsrv.csc.ncsu.edu/export/realistic-evaluation.pdf>
- [22] G. S. Lee, L. L. H. Andrew, A. Tang, and S. H. Low, "WAN-in-Lab: Motivation, Deployment and Experiments," in *Fifth International Workshop on Protocols for Fast Long-Distance Networks*, Marina Del Rey, CA USA, February 2007, pp. 85–90. [Online]. Available: <http://wil.cs.caltech.edu/pubs/PFLDnet07.pdf>
- [23] M. Mathis, J. Heffner, and R. Reddy, "Web100: extended tcp instrumentation for research, education and diagnosis," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 3, pp. 69–79, 2003.

- [24] “The Web100 Project,” November 2007, Accessed 19 Nov 2007. [Online]. Available: {<http://web100.org/>}
- [25] L. Stewart, J. Healy, “Characterising the Behaviour and Performance of SIFTR v1.1.0.” CAIA, Tech. Rep. 070824A, August 2007. [Online]. Available: {<http://caia.swin.edu.au/reports/070824A/CAIA-TR-070824A.pdf>}
- [26] L. Stewart, J. Healy, “Light-Weight Modular TCP Congestion Control for FreeBSD 7,” CAIA, Tech. Rep. 071218A, December 2007. [Online]. Available: {<http://caia.swin.edu.au/reports/070717B/CAIA-TR-070717B.pdf>}
- [27] J. Healy, L. Stewart, “P4DB: Branch jhealy_tcpcc,” Accessed 3 Mar 2008. [Online]. Available: {<http://perforce.freebsd.org/branchView.cgi?BRANCH=jhealy%5ftcpcc>}
- [28] D. J. Leith, R. N. Shorten, Y. Lee, “H-TCP: A framework for congestion control in high-speed and long-distance networks,” Hamilton Institute, Tech. Rep., August 2005. [Online]. Available: {<http://www.hamilton.ie/net/htcp2005.pdf>}
- [29] D. J. Leith, R. N. Shorten, “On RTT Scaling in H-TCP,” Hamilton Institute, Tech. Rep., September 2005. [Online]. Available: {<http://www.hamilton.ie/net/rtt.pdf>}
- [30] D. Leith, R. Shorten, “H-TCP: TCP for high-speed and long-distance networks,” in *Second International Workshop on Protocols for Fast Long-Distance Networks*, Argonne, Illinois USA, February 2004. [Online]. Available: {<http://www.hamilton.ie/net/htcp3.pdf>}
- [31] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, “TCP Selective Acknowledgement Options,” RFC 2018 (Proposed Standard), Oct. 1996. [Online]. Available: <http://www.ietf.org/rfc/rfc2018.txt>
- [32] V. Jacobson, R. Braden, and D. Borman, “TCP Extensions for High Performance,” RFC 1323 (Proposed Standard), May 1992. [Online]. Available: <http://www.ietf.org/rfc/rfc1323.txt>
- [33] V. Paxson and M. Allman, “Computing TCP’s Retransmission Timer,” RFC 2988 (Proposed Standard), Nov. 2000. [Online]. Available: <http://www.ietf.org/rfc/rfc2988.txt>
- [34] L. Rizzo, “Dummysnet: a simple approach to the evaluation of network protocols,” *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 1, pp. 31–41, 1997.
- [35] M. Welzl, “Port of h-tcp from ns-2.26 to ns-2.31,” Accessed 22 May 2008. [Online]. Available: http://caia.swin.edu.au/urp/newtcp/tools/htcp_ns-2.31.tar.gz
- [36] M. Allman and E. Blanton, “Notes on burst mitigation for transport protocols,” *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 2, pp. 53–60, 2005.