Experimental Evaluation of Latency Induced in Real-Time Traffic by TCP Congestion Control Algorithms

Alana Huebner[1]
Centre for Advanced Internet Architectures, Technical Report 080818A
Swinburne University of Technology
Melbourne, Australia
4087127@student.swin.edu.au

***Abstract*—This report contains an experimental evaluation of the impact that congestion control algorithms NewReno, HTCP and CUBIC have on the latency of VoIP traffic. It was found that when a TCP flow and VoIP flow share a bottleneck link the induced delay in the VoIP flow is related to the growth of the congestion window. CUBIC induced the highest average latency in the VoIP flow and HTCP the least.**

***Index Terms*—TCP, NewReno, CUBIC, HTCP, VoIP**

I. Introduction

The short comings of NewReno under particular traffic conditions has lead to the development of new TCP congestion control (CC) algorithms. The NewTCP project [1] aims to provide independent real-world evaluation of these competing CC algorithms.

The performance of a CC algorithm is commonly evaluated by its interaction with other TCP flows with metrics such as fairness, responsiveness and convergence. CC algorithms may also be evaluated by their interaction with unrelated and non-reactive traffic such as real-time traffic.

This report will investigate the latency that a TCP flow induces in a VoIP flow. This report's work extends previous empirical evaluation of latency induced in VoIP traffic by NewReno and HTCP under FreeBSD [2]. NewReno and HTCP under FreeBSD will be further investigated and the evaluation will be extended to include three congestion control algorithms under Linux, NewReno, HTCP and CUBIC, enabling a comparison between operating systems.

[1]The author is currently an engineering student at Swinburne University of Technology. This report was written during the author's winter internship at CAIA in 2008.

H-TCP and CUBIC are algorithms designed to address NewReno's poor performance over high bandwidth and long distance paths, as a result this report is mainly concerned with induced latency over large $bandwidth \times delay$ product (BDP) paths.

The dynamic nature of CC algorithms impacts on the occupancy of network queues. This report will show that queue occupancy is related to TCP's congestion window, and that the induced delay experienced by VoIP traffic is proportional to the queue occupancy.

Section 2 will begin with an brief discussion of congestion control algorithms and an overview of NewReno, HTCP and CUBIC.

Section 3 will cover the tools that were essential to the experimental method.

The experimental set up will be detailed in Section 4 with a description of the of the NewTCP testbed, hardware and software configuration and TCP tuning.

Section 5 describes the experimental method and how statistics were collected.

An analysis of the observed behaviour of each algorithm will be performed in Section 6. This section will also discuss differences between CC algorithms implemented under FreeBSD and Linux.

Section 7 will investigate the latency each algorithm induces in VoIP traffic.

Section 8 will conclude and identify areas for further work.

II. Congestion Control Algorithms

TCP has four types of congestion control algorithm: Slow Start, Fast Retransmit, Fast Recovery and Congestion Avoidance. This report is primarily concerned with congestion avoidance algorithms.

A congestion avoidance algorithm controls the value of TCP's congestion window variable $cwnd$ adaptively to network conditions. The actual window size used will be the minimum of the sender's buffer size, the receiver's window size and $cwnd$. Traditionally congestion avoidance follows an Additive Increase Multiplicative Decrease (AIMD) scheme.

The additive increase phase is generally specified in terms of the constant "alpha", α. During additive increase $cwnd$ grows incrementally, gently discovering the network's capacity to support the TCP flow. A common approximation described in [3] is to grow $cwnd$ according to Equation 1 every time a non-duplicate acknowledgement arrives.

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

The multiplicative decrease phase is generally specified in terms of the constant "beta", β, also referred to as the *backoff factor*. $cwnd$ continues to grow until congestion is detected, typically inferred by packet loss. Multiplicative decrease reduces $cwnd$ on packet loss to lessen the flow's impact on the congested network. The reduction of $cwnd$ is given by Equation 2.

$$cwnd = \beta \times cwnd \quad (2)$$

The AIMD scheme results in $cwnd$ cycling around the capacity of the path. Examples of this behaviour for each algorithm can be seen in Section VI.

A. NewReno

NewReno [4] is the most common variant of TCP. NewReno modified the Fast Recovery algorithm of standard TCP to include partial acknowledgements. Partial acknowledgements improve the performance of Fast Recovery when Selective Acknowledgements (SACK) are unavailable. Partial acknowledgements and SACK are just two of the extensions included in the evolution of TCP up to NewReno. TCP extensions are well documented in RFC4614 [5].

The AIMD scheme of standard TCP is characterised by $\alpha = 1$ and $\beta = 0.5$. $cwnd$ increases linearly by one Sender's Maximum Segment Size (SMSS) every RTT and halves on congestion.

B. H-TCP

H-TCP [6] is a congestion avoidance algorithm developed at the Hamilton Institute in Ireland to improve TCP performance over high-speed and long distance networks. Unlike NewReno, α and β are no longer constants.

H-TCP has high speed and low speed modes of operation for backwards compatibility with standard TCP. During low speed mode $\alpha = 1$ causing $cwnd$ to increase linearly in the same way as NewReno. After one second has elapsed since congestion H-TCP enters high speed mode. α becomes a parabolic function of time since the last congestion event and $cwnd$ grows more aggressively. When the duration of a congestion epoch is less than one second, as for low BDP networks, H-TCP never exits low speed mode and behaves like NewReno.

To achieve fairness between flows of differing RTTs through a common bottleneck, HTCP can utilise RTT scaling [7] to make $cwnd$ growth effectively invariant with RTT.

H-TCP has an adaptive backoff factor which ranges from 0.5 to 0.8. There are two algorithms which control the backoff factor, together they try to reach a compromise between bandwidth utilisation and responsiveness.

Bandwidth utilisation is maximised by reducing the idle time of the link caused by the reduction of $cwnd$ after congestion. The backoff factor is calculated such that the queue has just enough time to drain before $cwnd$ increases again. The ratio of RTT$_{min}$/RTT$_{max}$ is used in the calculation, where RTT$_{min}$ is an estimate of the propagation delay and RTT$_{max}$ is an estimate of the maximum additional queuing delay. This algorithm improves throughput on links with small queues but at the cost of responsiveness.

To achieve better responsiveness H-TCP can utilise an adaptive reset algorithm. The maximum value of $cwnd$ that was achieved just before a congestion event is monitored over time. If $cwnd_{max}$ increases H-TCP detects that more bandwidth is available and increases the backoff factor. If $cwnd_{max}$ decreases H-TCP detects that less bandwidth is available and the backoff factor is decreased, releasing bandwidth to any new flows.

C. CUBIC

The CUBIC [8] congestion avoidance algorithm was developed by the Networking Research Lab at North Carolina State University. Like H-TCP it is also designed for high-speed and long distance networks. CUBIC is a revision of the BIC algorithm which was found to be too aggressive towards standard TCP [9]. CUBIC has been the default CC algorithm in the Linux kernel since version 2.6.19.

The $cwnd$ growth is a cubic function of time since the last congestion event. CUBIC adaptively controls the cubic function through selection of the inflection point. The maximum value of $cwnd$ recorded just before congestion becomes the value of the inflection point for the next growth cycle. During the next cycle the $cwnd$ rapidly increases until it forms a plateau at the inflection point, thus maximising the time spent at the point where throughput is highest.

CUBIC's backoff factor is 0.8. CUBIC's authors acknowledge that this choice was made with the knowledge that convergence would be slower than standard TCP, and that they will possibly address the issue with adaptive backoff in a future version of CUBIC. Current versions of CUBIC somewhat mitigate the problem with a fast convergence algorithm. Fast convergence monitors the maximum value of $cwnd$ just before congestion events, if $cwnd_{max}$ decreases CUBIC will place the inflection point at $0.9 \times cwnd_{max}$ to release bandwidth to new flows.

III. Experimental Tools

A number of specially developed tools and patches were used to conduct the experiment described in this report. They can be downloaded from CAIA's NewTCP project web page [1]. Further mention of these tools in this report refer to the modified versions discussed below.

A. Siftr and Web100

SIFTR [10] is a FreeBSD kernel module that operates between the IPv4 and TCP layers of the network stack. When a TCP/IP packet is processed by the stack the state of TCP variables are logged to a file. SIFTR allows TCP statistics to be collected with very fine granularity making it ideal for TCP research.

Web100 [11] is a suite of TCP instruments for Linux created for the purpose of optimizing TCP performance. The instruments are implemented in a kernel module and accessed from userland.

Although each module is different in how and what TCP statistics they collect, the provide comparable data after some post processing of the Web100 output described in Section V. The most important statistic these tools allow us to collect is the instantaneous value of $cwnd$.

B. Dummynet

Dummynet [12] is a FreeBSD utility for traffic shaping and simulating network conditions. It is controlled through FreeBSD's IP firewall (ipfw). Packets are channelled into a dummynet pipe object with a firewall rule. The pipe acts on the traffic to simulate the effects of bandwidth, delay, packet loss and queue size. A patch developed at CAIA logs statistics about each pipe such as the occupancy of the pipe's queue and when packet loss occurs.

C. Iperf

The Iperf [13] utility creates TCP flows between an Iperf client and server. As of version 2.0.4 Iperf can be used to specify the congestion control algorithm used for a TCP flow. A patch developed at CAIA also allows send and receive buffer sizes to be configured for a flow.

D. FreeBSD Modular Congestion Control and H-TCP Module

As part of the NewTCP project a light-weight modularised congestion control framework has been developed for FreeBSD 7.0 [14]. Modular congestion control provides a means for dynamically loading congestion control algorithms into the TCP/IP stack. There were two CC modules available for FreeBSD at the time of writing, NewReno and HTCP. The HTCP module was also developed at CAIA [15].

IV. Experimental Setup

A. The Testbed

Experiments were conducted over a testbed configured in the dumbbell topology shown in Figure 1. The testbed emulates a bottleneck link between two Gigabit Ethernet networks.

The router between the networks runs FreeBSD 7.0 on an Intel Celeron at 2.80GHz. Dummynet was used to simulate a bottleneck link with configurable characteristics. All traffic between the networks passed through two dummynet pipes in either direction. The first pipe was configured with a bandwidth and drop tail queue to create the congestion point. The second pipe introduced the propagation delay of the link.

Host A and Host C acted as the sender and receiver of a TCP flow across the link. The hosts are dual-boot Linux 2.6.25 and FreeBSD 7.0-RELEASE. Linux was instrumented with Web100. FreeBSD was instrumented with the CAIA modular congestion control patch and the H-TCP module. The specifications of these hosts are listed in Table I.

Host B is a SmartBits2000 [16] industrial traffic generator and network performance analysis and testing system. It was used to generate a stream of pseudo VoIP traffic equivalent to the G.711 encoding standard: 200 Byte IP/UDP/RTP packets with inter-arrival times of 20ms. The VoIP traffic was sent from Host B to Host D

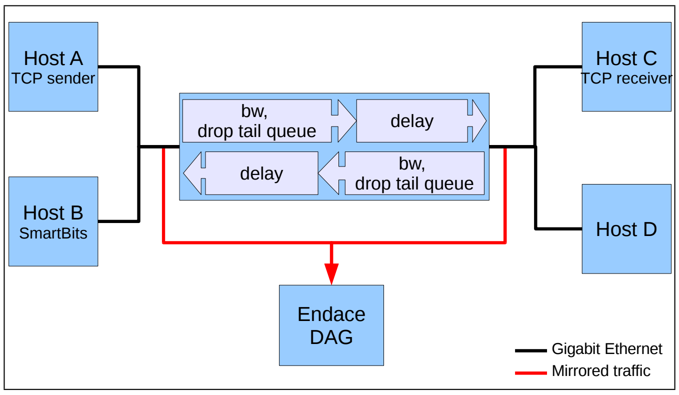

Fig. 1. Logical topology of the TCP Testbed

Motherboard	Intel Desktop Board DG965WH
CPU	Intel Core2 Duo E6320 1.86GHz 4MB L2 Cache
RAM	1GB (1 x 1GB) PC5300 DDR2-667
HDD	Seagate 250GB ST3250410AS SATA II
NIC	Intel 82566DC PCIe gigabit Ethernet (onboard) Intel PRO/1000 GT 82541PI PCI gigabit Ethernet

TABLE I
TESTBED PC SPECIFICATIONS

across the bottleneck link. Host D implemented a firewall to make it behave as a sink, accepting traffic from Host B but generating no reply traffic.

All ingress and egress traffic to and from the router was mirrored to a Endace DAG 3.7GF high performance capture card. The DAG host runs FreeBSD 7.0-RELEASE on a dual core Intel Pentium III at 1266MHz. Configuring the DAG card and capturing traffic with it is documented in [17]. The card was configured to time stamp packets with microsecond resolution.

B. TCP Tuning

As participants of the TCP flow Host A and Host C require a considerable amount of TCP tuning to reveal the growth of $cwnd$ as determined by the congestion control algorithm. TCP extensions, characteristics of the underlying operating system and implementation features all have the potential to obfuscate the behaviour of the algorithm. See [18] for a discussion about TCP tuning for FreeBSD. Configuration options not in Listing 1 were left at their default values.

C. H-TCP Tuning

The default operation of H-TCP varies between Linux and FreeBSD. The features of H-TCP were tuned on each operation system so that the implementations would behave in the same way.

Listing 1 Tuning TCP on FreeBSD

Boot loader tuning

```
vm.kmem_size=536870912
vm.kmem_size_max=536870912
kern.ipc.nmbclusters=32768
kern.maxusers=256
kern.hz=2000
hw.em.rxd=4096
hw.em.txd=4096
net.inet.tcp.reass.maxsegments=75000
htcp_load="YES"
```

sysctl tuning

```
kern.ipc.maxsockbuf=104857600
net.inet.tcp.inflight.enable=0
net.inet.tcp.hostcache.expire=1
net.inet.tcp.hostcache.prune=5
net.inet.tcp.recvbuf_auto=0
net.inet.tcp.sendbuf_auto=0
net.inet.tcp.tso=0
net.inet.ip.fastforwarding=1
net.isr.direct=1
net.inet.tcp.delayed_ack=0
net.inet.tcp.reass.maxqlen=13900
```

Listing 2 Tuning TCP on Linux

```
ethtool -K <experiment interface> tso off
sysctl net.ipv4.tcp_no_metrics_save=1
```

Adaptive backoff for bandwidth utilisation is enabled and not tunable on Linux, on FreeBSD it is tunable through the variable $adaptive\_backoff$ but disabled by default. To enable a comparison between the implementations $adaptive\_backoff$ was enabled on FreeBSD as shown in Listing 4.

Adaptive reset is only implemented on Linux so it was disabled as shown in Listing 3. It is controlled through the variable $use\_bandwidth\_switch$, which is enabled by default. This feature is not required as the tests involved only a single TCP flow across a constant bandwidth link.

RTT scaling is disabled by default on FreeBSD, but on Linux it is enabled by default. RTT Scaling is also unnecessary for tests involving single TCP flows so it was disabled in Linux through the variable $use\_rtt\_scaling$ as shown in Listing 3.

Listing 3 Tuning H-TCP on Linux

```
modprobe -r tcp_htcp
modprobe tcp_htcp use_rtt_scaling=0
use_bandwidth_switch=0
```

Listing 4 Tuning H-TCP on FreeBSD

```
sysctl net.inet.tcp.cc.htcp.adaptive_backoff: 1
```

V. Experimental Method

Each test run consisted of a single TCP flow and single VoIP flow simultaneously sharing the bottleneck link. A test run lasted 3 minutes and was performed at least twice. Additional runs were performed if the tests showed unexpected results as discussed in Section VII-A.

During each test run tcpdump captured all traffic on Hosts A, C and DAG.

A range of path characteristics were tested using all combinations of the parameters listed in Table II. Table III provides a full list of the test variables, the list of BDPs was derived from Table II.

A. Automated Test Scripts

A suite of scripts have been created as part of the NewTCP [1] project to automate the experimental process. The scripts configure the testbed, confirm connectivity, start and terminate logging, start and terminate TCP flows and transfer log files to a storage location. Some steps have been taken towards automating the data processing stage as well.

B. Gathering Latency Data

The traffic capture on the DAG interface records each packet twice, once upon entering the congested link and then again on leaving it. VoIP traffic was found in the dump file by filtering on the MAC addresses of SmartBits and Host D. The two observations of each VoIP packet were found by pairing up identical packet hashes. By subtracting the timestamps of the two observations it can be determined how long the packet spent in the link. This value is referred to as the one way delay (OWD).

C. Recording the Congestion Window

During the FreeBSD tests SIFTR was used to log TCP statistics on hosts A and C, SIFTR was configured to capture statistics for every TCP/IP packet processed by the stack ensuring the finest granularity possible. During the Linux tests TCP statistics were logged with Web100. Web100 was polled every 1 ms to ensure a reasonable granularity even with the lowest BDP test conducted. The Web100 output was logged to a file in the same format as the SIFTR log to allow us to easily process statistics collected under both operating systems.

Bandwidth	1Mbps, 10Mbps, 100Mbps
Queue size	1 BDP, 1/4 BDP
Delay	16ms, 80ms, 160ms

TABLE II
DUMMYNET CONFIGURATION PARAMETERS AND VALUES

Operating Systems	Linux 2.6.25, FreeBSD 7.0-RELEASE
Algorithms	NewReno, H-TCP, CUBIC (Linux only)
BDPs (Bytes)	$2*10^3$, $1*10^4$, $2*10^4$, $1*10^5$, $2*10^5$, $1*10^6$, $2*10^6$

TABLE III
TEST VARIABLES

VI. Algorithm Behaviour Analysis

Observing *cwnd* gives a direct insight into the behaviour of each algorithm. The value of *cwnd* gathered by SIFTR and Web100 is presented here across three typical congestion epochs from the middle of the 3 minute flow.

A congestion epoch is the time between congestion events, where the congestion event is packet loss due to a full queue at the congestion point. During a single epoch we observe TCP in fast recovery and congestion avoidance mode. Due to the static network conditions produced by the testbed the behaviour of *cwnd* during each epoch is approximately identical for the duration of the flow[1].

Figures 2, 3, 4, 5, and 6 show *cwnd* for tests with an 10Mbps link, round trip time of 80ms and queuing capacity equal to the BDP. For bottleneck links with a small BDP, such as the lowest BDP tested ($1Mbps \times 16ms$) there was no discernible difference between the algorithms because *cwnd* could not grow large. Differences began to appear when *cwnd* could grow beyond 25

[1]Except at the start and end of each run, and allowing for small anomalies which inevitably occur when experimenting with real equipment. A TCP flow starts in slow start mode and often ends with the congestion window suddenly opening in order to flush the send buffer. As this behaviour is not determined by the congestion avoidance algorithm we do not explore it further in this report.

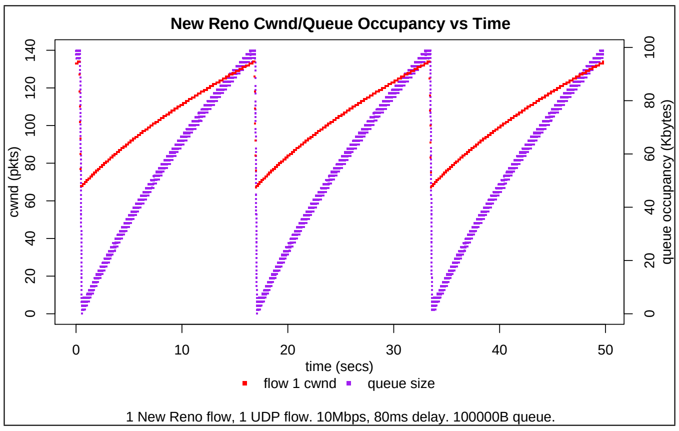

Fig. 2. Linux's NewReno

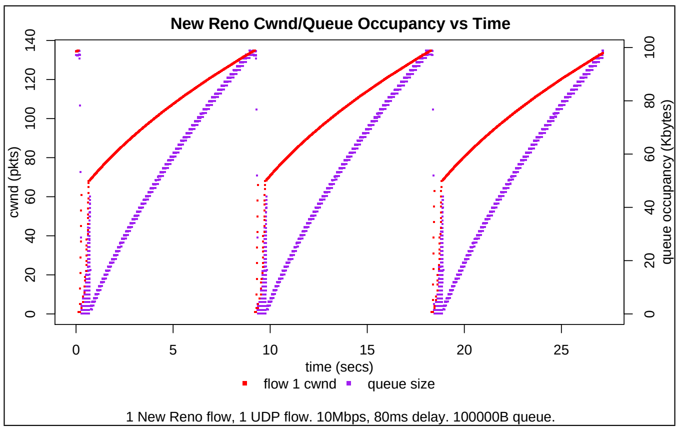

Fig. 3. FreeBSD's NewReno

packets. $10Mbps \times 80ms$ was the lowest BDP tested to show the growth of *cwnd* over an epoch with a fine enough granularity to clearly reveal the differences between the algorithms.

A. Congestion Epoch Duration

On comparing the NewReno and HTCP results on FreeBSD and Linux it is notable that the FreeBSD implementations have a shorter congestion epoch duration than the Linux versions due to faster *cwnd* growth.

On Linux delayed acknowledgement was enabled which may account for the difference. Delayed Acknowledgement defined in RFC1122 [19] increases TCP efficiency by allowing a single ACK to acknowledge up to two full sized data segments. The drawback of delayed acknowledgement is that less ACKs received by the sender results in *cwnd* being updated less often. Linux implements Appropriate Byte Counting defined

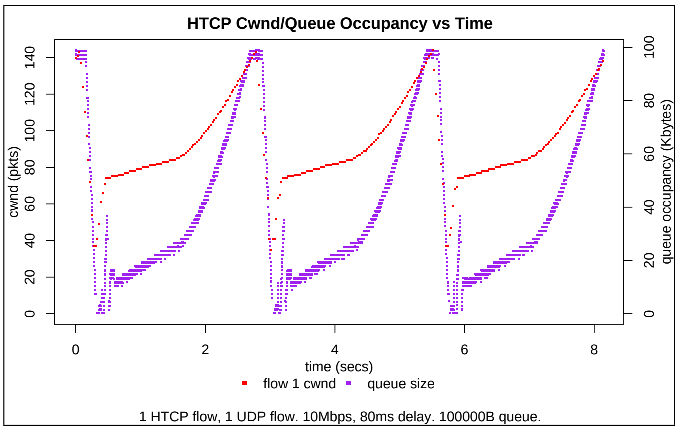

Fig. 4. Linux's H-TCP

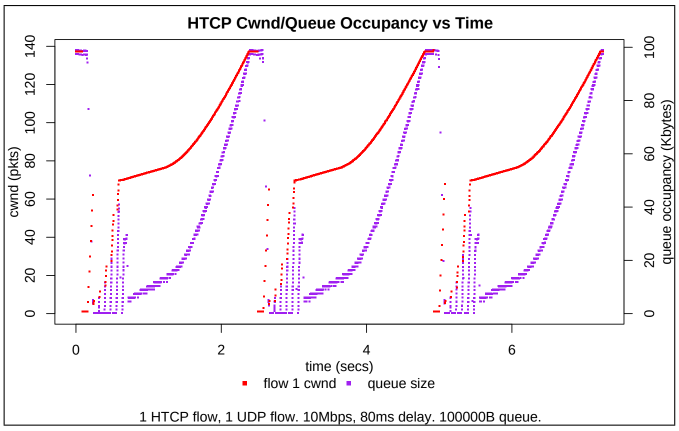

Fig. 5. FreeBSD's H-TCP

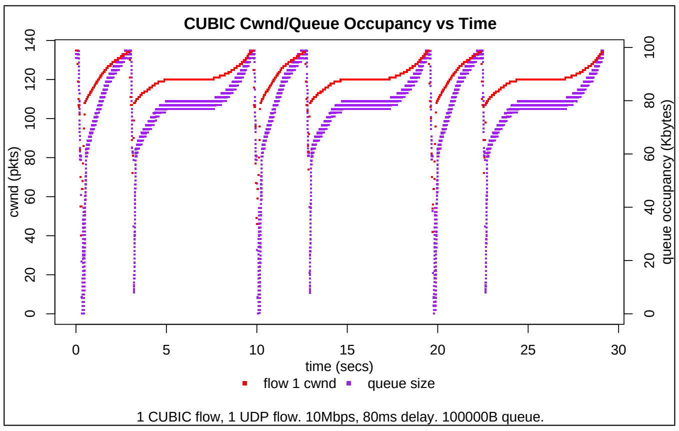

Fig. 6. Linux's CUBIC

in [20] which was designed to mitigate the problem by increasing $cwnd$ based on the number of bytes being acknowledged.

Despite Appropriate Byte Counting there is still a clear difference in the epoch duration between the FreeBSD and Linux NewReno tests, our results showed Linux's NewReno epoch duration to be approximately 1.85 times the epoch duration of FreeBSD's NewReno.

HTCP was less affected, Linux's HTCP epoch duration was approximately 1.14 times that of FreeBSD's. This may be because HTCP's high speed mode defines α as a function of time. Only the low speed NewReno mode of HTCP experienced noticeably slower growth.

B. Fast Recovery

FreeBSD and Linux implement different fast recovery algorithms which can be most clearly seen in Figures 4 and 5 near $t = 3$ seconds. FreeBSD implements the standard fast recovery algorithm defined in RFC2581 [3] and Linux implements an algorithm called Rate-Halving [21].

Theoretically $cwnd$ is set to $\beta \times cwnd$ after fast recovery but in implementation this can cause micro-bursts, a large burst of packets which may saturate the queue. Linux and FreeBSD solve this problem in a similar way, by incrementally yet rapidly increasing $cwnd$ up to the backoff value. The queue experiences a series of small bursts during this process. Burst mitigation is discussed in [22].

C. CUBIC's fast convergence mode

An interesting behaviour of CUBIC is the two-part $cwnd$ cycle shown in Figure 6. In our tests this was observed whenever the queuing capacity of the path was 1 BDP and for all queue capacities of our largest BDP tests $(100Mbps \times 160ms)$.

The short epoch is the expected behaviour of CUBIC during congestion avoidance. We observe the concave tail of a cubic function with the inflection point set at $cwnd_{max}$ of the last epoch.

During the longer section of the cycle CUBIC appears to have entered fast convergence mode as described in [8]. CUBIC has detected a decrease in $cwnd_{max}$ and lowered the inflection point to release bandwidth to new flows. The actual decrease experienced in the test was no more than a few packets but the Linux 2.6.25 implementation of CUBIC takes any decrease in $cwnd_{max}$ to trigger fast convergence. For this traffic scenario CUBIC's fast convergence algorithm incorrectly assumes that even the slightest decrease in $cwnd_{max}$ means new flows have entered the link.

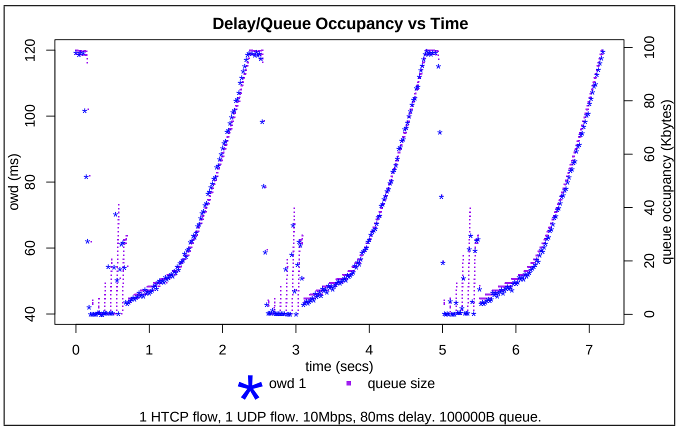

Fig. 7. OWD experienced by VoIP traffic sharing a congested link with Linux's H-TCP

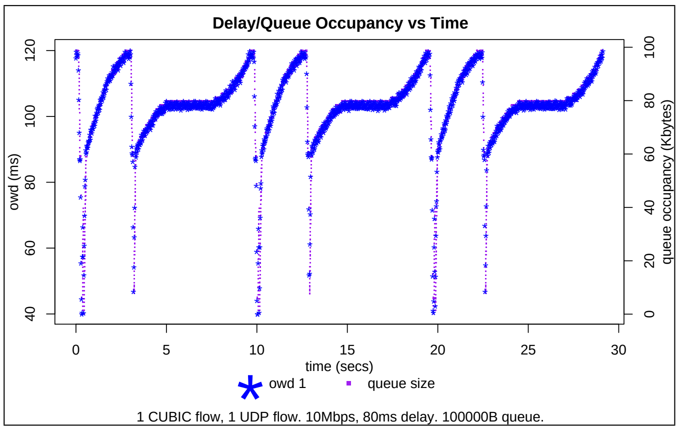

Fig. 8. OWD experienced by VoIP traffic sharing a congested link with Linux's CUBIC

VII. Latency Analysis

Figures 2, 3, 4, 5, and 6 show the occupancy of the queue at the congestion point as $cwnd$ cycles. These figures demonstrate the close relationship that exists between the congestion window and the queue occupancy. As $cwnd$ grows the throughput of the flow increases and the queue fills. When the queue reaches maximum capacity packets are dropped and within a few RTTs TCP reacts by multiplicatively decreasing $cwnd$. With the reduction in throughput the queue has time drain. The occupancy of the queue over time mimics the shape of $cwnd$ growth and has the same epoch frequency. The rate at which each algorithm fills the queue is as different as the algorithms themselves.

FIFO queues introduce delays proportional to the number of packets already in the queue. The one way

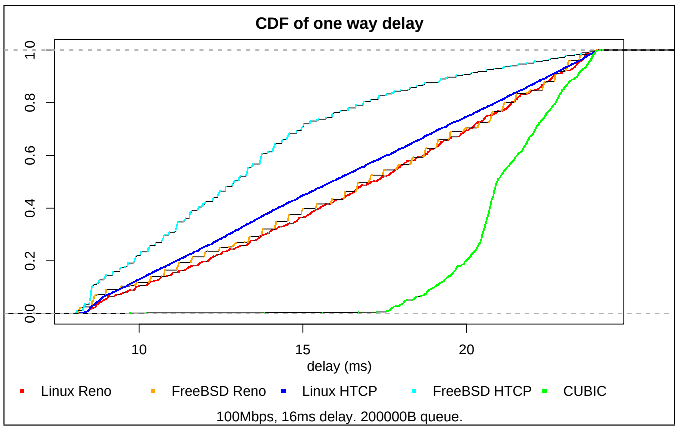

Fig. 9. CDF of delay experienced by VoIP traffic sharing a congested link with a TCP flow

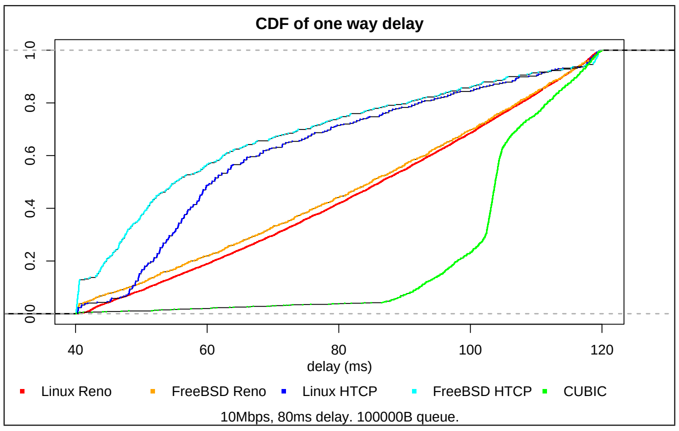

Fig. 10. CDF of delay experienced by VoIP traffic sharing a congested link with a TCP flow

delay experienced by the VoIP traffic against queue occupancy is shown in Figure 7 for H-TCP and in Figure 8 for CUBIC. These figures confirm that delay and queue occupancy are directly proportional. Therefore the delay experienced by VoIP packets over time is also closely related to *cwnd* growth of the TCP flow it shares the bottleneck link with.

Figures 9, 10 and 11 show the CDF of delay experienced by the VoIP traffic sharing the link with a TCP flow for three sets of link characteristics. The queuing capacity was equal to the BDP for each link. The data set was taken over three typical congestion epochs of the TCP flow.

Each algorithm has a unique distribution of induced delay which is approximately consistent[2] for the three link scenarios presented here. The distribution of delay induced by NewReno is approximately uniform due to its linear *cwnd* growth. HTCP's slow then fast *cwnd* growth distributes induced delay towards the minimum delay. CUBIC's initially agressive concave *cwnd* growth distributes induced delay towards the maximum delay.

Each algorithm filled the queue to its maximum capacity causing the VoIP traffic to experience the maximum queuing delay. NewReno and HTCP allowed the queue to completely drain before slowly filling it again. CUBIC rarely allows the queue to become empty, keeping the queue over $3/4$ full 90% of the time.

The median induced delay for the three link scenarios is shown in Figure 12. At each BDP H-TCP induced the lowest median delay while CUBIC induced the largest median delay.

[2]Excluding HTCP under Linux, discussed in VII-A

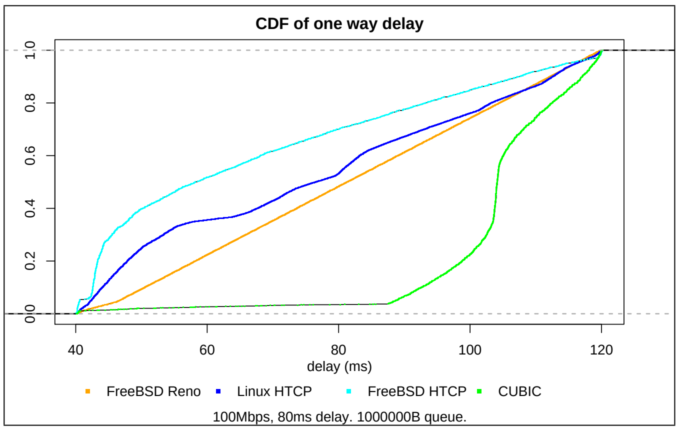

Fig. 11. CDF of delay experienced by VoIP traffic sharing a congested link with a TCP flow

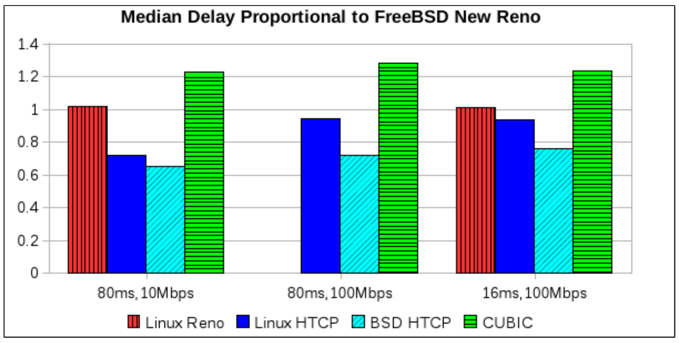

Fig. 12. Median delay experienced by VoIP traffic sharing a congested link with a TCP flow. 1 BDP queues.

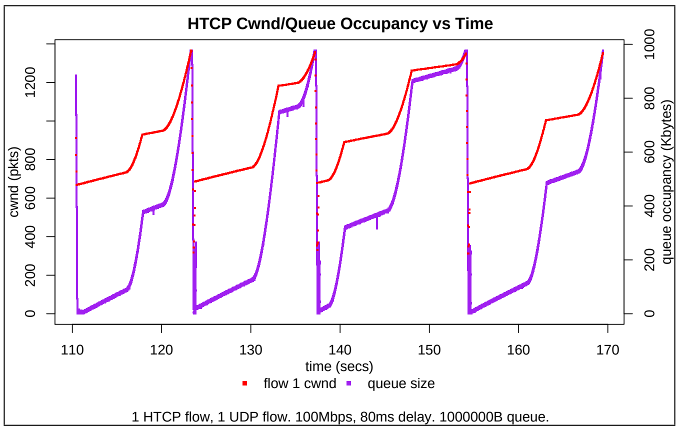

Fig. 13. Unexpected *cwnd* growth of Linux's HTCP

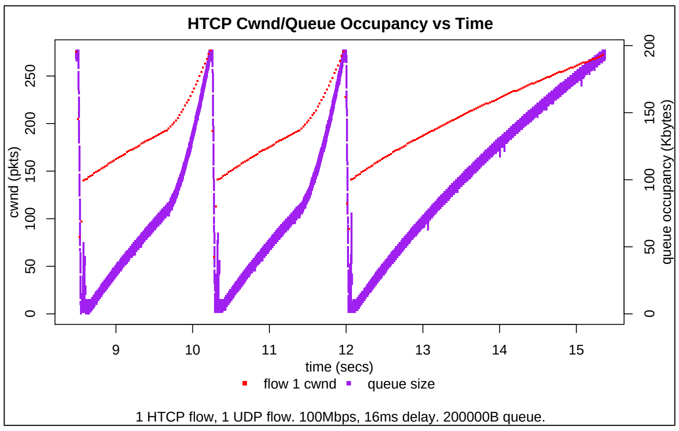

Fig. 14. Unexpected *cwnd* growth of Linux's HTCP

A. *HTCP Irregularities*

For particular path characteristics we observed unexpected results from HTCP under Linux. Unexpected *cwnd* growth was consistent across all test runs for the duration of each test. The cause of this behaviour is left for future investigation.

In Figure 13 the growth of *cwnd* can be seen fluctuating between low speed mode and high speed mode within a single congestion epoch. This behaviour was observed when the link was characterised by the two highest BDPs tested ($100Mbps \times 80ms$ and $100Mbps \times 160ms$). The congestion epoch duration was of the order of 10s of seconds which may be why this behaviour is not observed for lower BDP tests. The effect of this fluctuation on the distribution of delay induced by HTCP can be observed in Figure 11.

In Figure 14 *cwnd* growth is seen to behave as HTCP for the first two epochs and as NewReno for last epoch. Consistently observed during the $100Mbps \times 16ms$ tests, this behaviour continues across the entire length of the test run, with HTCP-like epochs randomly dispersed among NewReno-like epochs. As a result the distribution of delay induced by HTCP in Figure 9 is similar to NewReno.

VIII. Conclusion and Further Work

This report describes real-world experimentation with TCP CC algorithms to investigate how they impact on the latency of real-time traffic. Tests consisted of a TCP flow and a VoIP flow traversing a bottleneck link. The algorithms investigated were NewReno, HTCP under FreeBSD and NewReno, HTCP and CUBIC under Linux. A range of link characteristics were tested, with most interesting results occurring at high BDPs.

The operation of algorithms during congestion avoidance mode was observed through the growth of *cwnd*. At the same time the queue occupancy was shown to rise and fall in the same cyclical way as *cwnd*. It was found that the delay experienced by the VoIP flow was directly proportional to the queue occupancy. CUBIC induced the most average delay in the VoIP flow and HTCP the least.

On comparing the implementations of NewReno and HTCP under FreeBSD and Linux it was found that each operating system used a different fast recovery technique but that the CC algorithms were generally the same. The Linux algorithms had longer congestion epochs which was possibly due to delayed acknowledgements being enabled. In future experiments better comparisons could be made with further TCP tuning under Linux.

It would be useful to extend this analysis to link characteristics which emulate a particular real-word scenario. For example a consumer ADSL link, to investigate the impact of latency fluctuations on a user who is simultaneously holding a VoIP conversation and executing a bulk TCP data transfer.

Furthermore our results show that the rate at which latency cycles is different for each algorithm, it is left for future work to discover the impact of congestion epoch frequency on real-time traffic.

Acknowledgements

I would like to thank Lawrence Stewart for guidance and assistance throughout this project, Grenville Armitage for his support and CAIA for the opportunity to undertake an internship. This report has been made possible in part by a grant from the Cisco University Research Program Fund at Community Foundation Silicon Valley.

References

[1] "The NewTCP Project," August 2008, Accessed 8 Aug 2008. [Online]. Available: http://caia.swin.edu.au/urp/newtcp

[2] G. Armitage, L. Stewart, M. Welzl, and J. Healy, "An independent H-TCP implementation under FreeBSD 7.0 - description and observed behaviour," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, 2008.

[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] 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

[5] M. Duke, R. Braden, W. Eddy, and E. Blanton, "A Roadmap for Transmission Control Protocol (TCP) Specification Documents," RFC 4614 (Proposed Standard), Sep. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4614.txt

[6] 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

[7] 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

[8] 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

[9] "Bic and cubic," Accessed 7 Aug 2008. [Online]. Available: http://netsrv.csc.ncsu.edu/twiki/bin/view/Main/BIC

[10] 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

[11] "The Web100 Project," November 2007, Accessed 19 Nov 2007. [Online]. Available: http://web100.org/

[12] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols," *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 1, pp. 31–41, 1997.

[13] , "Iperf - The TCP/UDP Bandwidth Measurement Tool," May 2005, Accessed 19 Nov 2007. [Online]. Available: http://dast.nlanr.net/Projects/Iperf/

[14] 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

[15] 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

[16] "Spirent smartbits - trusted industry standard for router and switch testing," Accessed 11 Aug 2008. [Online]. Available: http://www.spirent.com/analysis/technology.cfm?media=7&ws=325&ss=110&stype=15&a=1

[17] A. Heyde, L. Stewart, "Using the Endace DAG 3.7GF Card With FreeBSD 7.0," CAIA, Tech. Rep. 080507A, May 2008. [Online]. Available: http://caia.swin.edu.au/reports/080507A/CAIA-TR-080507A.pdf

[18] L. Stewart, J. Healy, "Tuning and Testing the FreeBSD 6 TCP Stack," CAIA, Tech. Rep. 070717B, July 2007. [Online]. Available: http://caia.swin.edu.au/reports/070717B/CAIA-TR-070717B.pdf

[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] M. Allman, "TCP Congestion Control with Appropriate Byte Counting (ABC)," RFC 3465 (Experimental), Feb. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3465.txt

[21] Matt Mathis, Jeff Semke, Jamshid Mahdavi, Kevin Lahey, "The Rate-Halving Algorithm for TCP Congestion Control," Internet Engineering Task Force, Tech. Rep., June 1999. [Online]. Available: http://www.tools.ietf.org/html/draft-mathis-tcp-ratehalving-00.txt

[22] M. Allman and E. Blanton, "Notes on burst mitigation for transport protocols," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 2, pp. 53–60, 2005.