2009 International Conference on Advanced Information Networking and Applications

An Experimental Evaluation of Sender-Side TCP Enhancements for Wired-to-Wireless Paths: A Real-World Home WLAN Case Study

Ritesh Taank
School of Engineering & Applied Science
Aston University
Birmingham, UK
taankr@aston.ac.uk

Xiao-Hong Peng
School of Engineering & Applied Science
Aston University
Birmingham, UK
x-h.peng@aston.ac.uk

***Abstract*—In this paper we evaluate the performances of recently proposed sender-side TCP enhancements designed to alleviate TCP's deficiencies over wired-to-wireless paths. Our goal is to compare the performances of the legacy TCP RENO, CUBIC, HYBLA, VENO, and WESTWOOD+, which are available to researchers via pluggable congestion control algorithms in the recent Linux v2.6 kernels. We conducted experiments over a custom-built wired-to-wireless testbed consisting of a real-world infrastructure WLAN within a home environment, and a Linux server with real implementations of the various TCP algorithms. Our focus is on the achieved end-to-end TCP throughputs and transfer times for small, medium, and large data transfers from the fixed TCP server to a last-hop wireless client. We also present insights of the average TCP RTO timer values for each TCP algorithm during the experiments.**

Keywords-TCP; wired-to-wireless networks; testbed.

I. INTRODUCTION

The legacy TCP's congestion control algorithm [1] has been very successful in making the current Internet function efficiently, providing a reliable connection-oriented end-to-end data delivery service to end-users across the globe. In recent years, the advent of the IEEE 802.11 WLAN [2] technology has allowed end-users to be mobile and location independent from within their homes and office environments. It is therefore a common scenario today for TCP segments originating from a server somewhere in the Internet to traverse a wireless link in the last-hop portion of the end-to-end journey.

The problems with having a wireless medium for the transmission of data are fairly self-explanatory; i) higher bit and frame error rates (BER and FER) from the natural effects of the radio transmission medium [3], ii) the prevalence of over-the-air collisions of frames when there are multiple wireless devices in the WLAN, and iii) highly variable transmission delays of frames over the WLAN, due to client contention at the access point (AP), as well as sudden queuing delays when the AP's 802.11 stop-and-wait ARQ mechanism kicks in.

The issues with TCP stem from the fact that the legacy (or standard) TCP [4] is unable to distinguish between segment losses that occur over the wired network and from those that occur over the wireless portion. Hence it reacts to both events identically by invoking its original *Additive Increase Multiplicative Decrease* (AIMD) congestion control algorithm and reducing its congestion window (*cwnd*) size [3]. At the same time it is unable to interpret why there is sudden increase in the time taken for an acknowledgement segment (ACK) to return from the recipient, causing a retransmission timeout (RTO) event. This could force TCP to retransmit the unacknowledged segment/s and cut back its sending rate, leading to poor throughputs for wireless end-users and poor utilisation of the wireless channel bandwidth in the last-hop.

Research into the performance issues of the legacy TCP over wired-to-wireless paths has therefore become an avenue of major research in recent years [5] [6]. In this work, we present a detailed analysis and evaluations of some of the recently proposed TCP enhancements for such conditions in an attempt to observe how they weigh up against the standard TCP protocol, TCP Reno. The idea of this paper is to make enhancements to the protocol at the sending side in the fixed network, which can be more easily rolled out across servers in the Internet.

In this paper we focus our attention to some of the popular and recently proposed enhancements to the standard TCP's AIMD congestion control algorithm for wired-to-wireless paths, which have also made their way into the Linux v2.6 kernel's TCP stack. The Linux v2.6 kernel allows TCP AIMD algorithms to be selected at runtime as a 'pluggable' option, without the need for recompilation [7]. Our research within the area suggests that TCP Hybla [8], TCP Veno [9], and TCP Westwood+ [10] appear to be the most popular and most recent candidates for enhanced performance over wired-to-wireless conditions. To increase diversity, we also put TCP Cubic [11] under test, as it is used as the default algorithm in Linux kernels since v2.6.19. The rationale here is that Linux web-servers are widely deployed across the Internet, and server administrators may not make changes to the default TCP congestion algorithm, implying that last-hop wireless end-users around the world are more likely to be served by TCP Cubic. Finally, we also put the legacy TCP Reno [12] under test, acting as a base-line for comparisons and evaluations of the enhanced AIMD algorithms.

1550-445X/09 $25.00 © 2009 IEEE
DOI 10.1109/AINA.2009.51

Evaluating the performance of TCP over wireless paths is not a straight forward task, and many studies opt for simulation or emulation in order to study TCP behaviour in a non-complex manner. Several studies try to add realism through the use of an experimental testbed, consisting of a real-world last-hop wireless network. However, we found very little literature on a systematic evaluation of real-world TCP AIMD implementations in real-world typical conditions, reflecting a true wired-to-wireless path for TCP. Therefore, 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 systematically compare the performances of TCP AIMD enhancement proposals for tackling the issues related to wired-to-wireless conditions. Specifically, we perform a set of benchmark tests over real-world conditions reflecting a typical home WLAN environment connected to a wired backbone, with a Linux v2.6.20 kernel machine acting as our TCP server. We present experimental results of the performances of TCP Reno, TCP Cubic, TCP Hybla, TCP Veno, and TCP Westwood+, over a range of scenarios involving multiple wireless devices in the last-hop WLAN. TCP New Reno was not available as a pluggable sender-side algorithm within the Linux kernel at the time of investigation.

II. EVALUATING TCP ENHANCEMENTS

Before proceeding with the experimental descriptions and methods, we briefly review the basic functionality of each TCP algorithm and assume that the reader is familiar with the basic principles of congestion control in TCP, such as the sender's congestion window and its relationship with the AIMD mechanism. It should also be stated that TCP performs loss detection when a Retransmission Timeout (RTO) timer expires whilst waiting for an ACK from the receiver, or three duplicate ACKs (DUPACKs) for the same unacknowledged data arrive. Because TCP's error detection and recovery is performed end-to-end, it also operates transparently for segment losses occurring over the wireless portion of a connection.

A. TCP Reno

TCP Reno was introduced as an enhancement to the original TCP Tahoe [4] by altering the way it reacted to segment losses due to congestion. It introduced the Fast Recovery algorithm, which is activated immediately after the Fast Retransmit algorithm upon the arrival of three DUPACKs. Although TCP Reno was never designed for usage over wireless paths, it was very widely adopted by the Internet community and is still likely to be in use today for last-hop wireless end-users. Reno was designed with network congestion in mind, where losses were presumed to occur mainly due to buffer overflows along connection paths, and not due to wireless channel errors.

B. TCP Cubic

Cubic is a congestion control algorithm designed to meet the demands of today's high-speed long distance networks and tackle the problem that is exhibited by standard TCP algorithms due to their poor AIMD response times. In essence, Cubic uses a *cubic function* to govern the size of its *cwnd*. A variable W_{max} is maintained as the size of the *cwnd* prior to a reduction event. When the *cwnd* increases again, it grows quickly initially, but decelerates this growth as its size approaches W_{max}, accelerating again as its size continues to grow beyond W_{max}.

C. TCP Hybla

TCP Hybla was proposed with the primary objective of providing better performance in end-to-end paths that possess larger segment round-trip-times (RTTs) such as in a satellite radio link. Briefly, its core enhancements include a modification of the standard TCP AIMD algorithm for higher latency paths, enforcing the use of the SACK policy [13] (to counter multiple losses), segment timestamps (to counter RTO timer exponential back-off issues), and a segment 'pacing' technique. Hybla's modifications to the *cwnd* evolution are a direct result of analytical studies on its behaviour in such conditions, leading to the conclusion that altering the *cwnd*'s sole dependence on the RTT for growth is a viable solution.

D. TCP Westwood+

TCP Westwood+ is a direct modification to the standard TCP AIMD algorithm in an attempt to provide enhanced performance over wired-to-wireless paths. It uses an end-to-end *bandwidth estimator* to alter the size of its *cwnd* and the *ssthresh* value, specifically after a segment loss event. Using a 'low-pass filtering' mechanism, it monitors the rate of returning ACKs from the recipient. The idea behind this is to continuously be aware of the end-to-end bandwidth, so that at the point of a segment loss occurring Westwood+ is adaptively able to set the *ssthresh* and *cwnd* to suitable values, to increase throughput over wireless networks.

E. TCP Veno

TCP Veno is also a direct modification to the standard TCP AIMD algorithm, trying to improve performance over wired-to-wireless paths by heuristically being able to discriminate between segment losses due to congestion in the wired network and random losses due to the wireless radio link. The key feature of Veno is that it monitors the network path for levels of congestion, and then uses this information to determine the cause of loss. If a loss occurs whilst Veno is in the congestive state, it assumes that the loss is due to network congestion, otherwise is random loss. Another feature of Veno is its modified linear portion of the additive increase algorithm of standard TCP, so that when Veno is in the congestive state the *cwnd* growth is less aggressive. This effectively allows any self-induced network congestion to be relieved without segments being dropped in the wired network, helping to maintain a higher overall throughput.

III. TESTBED SETUP AND CONFIGURATION

A. Hardware Components

All tests were conducted over a dedicated wired-to-wireless experimental testbed, as shown in Figure 1. At a high-level, there is a wired path and a wireless path for all end-to-end TCP connections. All wired machines in the testbed are high-end PCs equipped with 1024MB of system memory. The key component in the testbed is the experimental TCP server running the Linux v2.6.20 kernel, which we patched and instrumented with the *Web100* extensions [14]. The Web100 patched kernel allowed us to extract raw TCP data from the live kernel TCP stack whilst running experiments.

The Internet emulator machine consists of two 10/100Mbps Ethernet NIC cards, which were bridged at layer-2 using the Linux Bridge packages. This allowed traffic to flow through it in both directions (full-duplex) as close as possible to the PHY-layer. We also deployed the open-source Linux Advanced Routing and Traffic Control (LARTC) packages on the emulator in order to shape forward and reverse traffic flowing through the bridge interface. Rather than selecting arbitrary values as parameters, we conducted some real-world studies on the general characteristics of the Internet, as documented in [15]. We were primarily interested in those statistics that would impact the end-to-end behaviour of TCP. We used the *pchar* tool to gauge chosen statistics from the live Internet at random intervals over a period of seven days.

The idea behind our experimental TCP server and the Internet emulator is simply to create a lower-level of abstraction of a typical server in the Internet, one that isn't in a geographically distant location, and one that we can actually control and whose TCP behaviour we can actually observe and measure. It can be appreciated that in the real-world Internet it is not practical to instrument or measure the TCP behaviour of a web-server. Our view is that the bulk of TCP data sending is done by web-servers in the Internet, and hence it is more beneficial to study TCP settings and behaviour at the server in the wired domain.

The IEEE 802.11g AP is an off-the-shelf Linksys WRT54G device whose settings were their factory defaults. The last-hop home WLAN consists of a maximum of five 802.11g devices, configured using DCF in an infrastructure configuration. The details of their assigned names and specifications are provided in Table 1. A fast Netgear SR2016 Gigabit switch was used to interconnect both wired domains to the AP. All wired connections between the experimental server, the live Internet, the Internet emulator, the switch, and the AP are using full-duplex 10/100Mbps Ethernet cables.

B. Software Configurations

On the experimental TCP server we enabled the following options in the TCP stack: SACK, window scaling, delayed-ACKs, the Nagle algorithm, and path

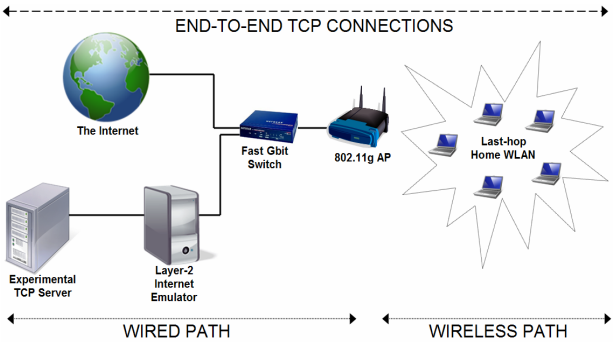

Figure 1. The Wired-to-Wireless Testbed

Table 1. 802.11g Device Specifications

Client Name	OS	Memory	CPU
SEAMOSS	Win XP SP2	512MB	1.8GHz
HANALENI	Linux	512MB	1.9GHz
TEMPLETRE	iPhone OS	128MB	412MH
CEMPAKA	Win XP SP2	256MB	1.5GHz
DAVANA	Win XP SP2	512MB	1.5GHz

MTU discovery. We feel such settings are typical of most TCP implementations today [16]. The TCP segment size (MSS) was 1448 Bytes.

We used *iperf* to transfer fixed amounts of TCP data from the server to a designated 802.11g client (SEAMOSS) in the last-hop WLAN. To capture live data from TCP variables in the kernel we wrote a *PythonC* script specifically for the Web100 extensions. The script was executed each time an iperf transfer was initiated at the server, and was designed to periodically read selected live TCP variables and dump their values to a file for later processing. We also used *tcpdump* at the TCP layer on the experimental server and on SEAMOSS to capture all sent and received segments per experimental trial. The dump files were then analysed using the popular *tcptrace*. All 802.11g devices were running default TCP/IP networking settings according to operating system installation in order to maintain realism throughout.

C. Home WLAN Scenarios

The infrastructure WLAN component of the testbed was setup and configured to represent a modern-day home WLAN environment, consisting of five wireless end-users (802.11g devices) connected to the Internet via a local gateway, the 802.11g AP. In this context we assume typical UK family household consisting of two adults and three youths. Each family member presumably owns an 802.11g compatible device, and is connected to the AP with unlimited access to the Internet. We allocate five different physical locations within an actual home where wireless users typically could be situated, independently downloading and uploading data to and from the Internet via the AP. At each of the chosen locations we assign a particular 802.11g device from Table 1. The layout of the

home is illustrated in Figure 2 (not to scale), showing the positions of the five chosen locations (marked 1 to 5), as well as the location of the AP. The dimensions of the home are 12.3m by 10.4m, with internal room wall partitions made from two layers plasterboard.

To create as much realism as possible we use a range of scenarios within the home WLAN in order to observe how the TCP enhancements in the wired domain perform over a variety of conditions in the last-hop connection. In total we consider five scenarios, ranging from A to E. In scenario A there is just a single client in the WLAN, positioned at location 1. We refer to the device at location 1 as SEAMOSS. In Table 2 we describe each of the scenarios in detail, with information on where the devices are positioned, and the actions that were performed by each. Each scenario from A to E increases the number of 802.11g devices in the WLAN, from 1 to 5 respectively. Note that for each scenario, and in order to maintain consistency in our results, SEAMOSS at location 1 is always the TCP client for all our tests. All other devices were communicating with the live Internet as per Table 2. Note that in each scenario the WLAN was saturated, i.e. all devices were downloading/ uploading data for the full duration of the TCP data download by SEAMOSS.

D. *Slected Experiments and Measurements*

In this paper we consider *small*, *medium*, and *large* TCP data flows to cover a diverse range of testing styles. The *small* flows transfer 1MB of data (representative of web-page downloads), the *medium* flows 10MB (representative of hi-res image files), and the *large* flows 50MB (representative of an open-source application file via FTP). All data flows are from the experimental TCP server to SEAMOSS in the last-hop WLAN. All TCP segments (including TCP ACKs returning from SEAMOSS) passed through the Internet emulator machine. The *small* transfers test the initial start-up behaviour of the various TCP congestion control algorithms, where it is mainly the *slow start* mechanism. In contrast, the *medium* and *large* transfers test the steady-state behaviour of the various algorithms, which rely mainly on the *congestion avoidance* mechanism [17] for the bulk of the data transfers.

The key performance measurements in our experiments were a) the *time taken to transfer* the entire amount of data per flow, and b) the *average throughput achieved* during the lifetime of a flow. There are many additional indicators of TCP performance; however the idea in this paper is simply to evaluate how TCP performs in a real-world context, where wireless end-users are expectant of high performance from the Internet, i.e. quick downloads and response times. As a supplement we have also included a section that presents the *average retransmission timeout* (*RTO*) value. The duration of time TCP waits for an ACK is known as the RTO timer value, which when it expires triggers the immediate retransmission of the first unacknowledged segment.

Table 2. Scenarios A to E

	Client Name	Client Action	Location
A	SEAMOSS	Downloading iPerf TCP Data	1
B	SEAMOSS	Downloading iPerf TCP Data	1
	HANALENI	Downloading FTP Data	2
C	SEAMOSS	Downloading iPerf TCP Data	1
	HANALENI	Downloading FTP Data	2
	TEMPLETREE	Streaming Video from BBC iPlayer	3
D	SEAMOSS	Downloading iPerf TCP Data	1
	HANALENI	Downloading FTP Data	2
	TEMPLETREE	Streaming Video from BBC iPlayer	3
	CEMPAKA	Uploading FTP Data	4
E	SEAMOSS	Downloading iPerf TCP Data	1
	HANALENI	Downloading FTP Data	2
	TEMPLETREE	Streaming Video from BBC iPlayer	3
	CEMPAKA	Uploading FTP Data	4
	DAVANA	Streaming Video from BBC iPlayer	5

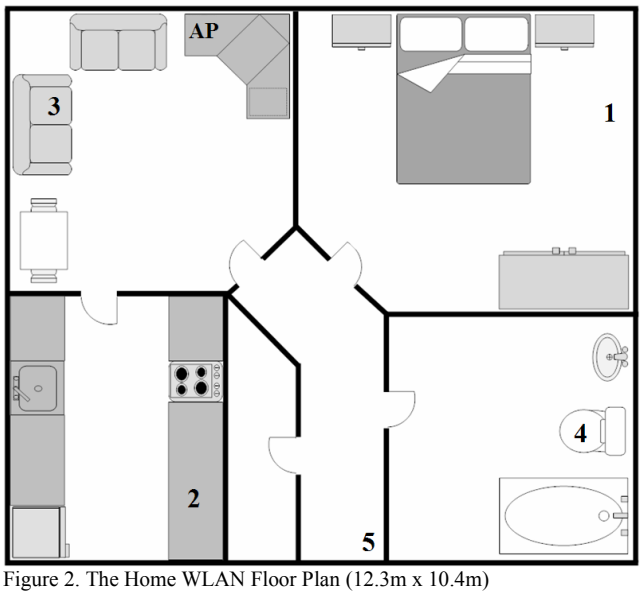

Figure 2. The Home WLAN Floor Plan (12.3m x 10.4m)

It has recently been derived that the traditional computation of TCP's RTO timer may not be suited to wired-to-wireless paths [18], which are typical of non-congestion related segment losses. The RTO timer has been shown to expire over wired-to-wireless connections whilst the AP is still retransmitting erroneous 802.11 frames locally. However, observing the average RTO timer value of

TCP congestion control algorithms can be insightful, because it is dependent on TCP's estimate of the RTT. Since each of the TCP algorithms utilises different methods for obtaining and updating its RTT measurement, this could have a direct impact on values computed for the RTO timer.

IV. RESULTS AND EVALUATIONS

In this section we present our results obtained from performing end-to-end TCP data transfers to SEAMOSS in the last-hop WLAN. We report on the effectiveness of TCP Reno, TCP Cubic, TCP Hybla, TCP VEeno, and TCP Westwood+ over a wired-to-wireless testbed, and for differing scenarios over the WLAN.

A. Small TCP Transfers

For 1MB transfers between the experimental TCP server and SEAMOSS, our results incorporate data transfers for scenario A (with SEAMOSS only) to scenario E (with five 802.11g clients in the WLAN including SEAMOSS). We present averages of three independent runs per TCP congestion control variant, per scenario.

Figure 3 is a plot of the average time taken to transfer 1MB of data between the TCP server and SEAMOSS in the last-hop WLAN. It can be seen that TCP Hybla supersedes all of the other TCP congestion control algorithms on test for all scenarios. TCP Reno performed the transfer in less time than TCP Veno, TCP Westwood+, and TCP Cubic in scenarios A, B, and C, suggesting that for short transfers and fewer 802.11g devices in the WLAN, its performance may not suffer as prominently as may originally have been thought. However, when we increased the number of devices in the WLAN, TCP Reno's performance quickly degraded the most. Interestingly, TCP Veno performed only slightly better than TCP Reno, as it is modifications of the AIMD algorithm based on Reno's.

Figure 4 plots the average throughput achieved during the lifetime of a particular transfer for each of the TCP algorithms on test, which we calculated by using tcptrace on the tcpdump capture files running on SEAMOSS. From this we extracted the number of TCP data segments that were actually received, denoted by N_{TCP}, and then used the time taken to transfer the entire 1MB of data, denoted by T. The TCP throughput (bps), η, was obtained using the following equation (where $MSS = 1448 \times 8 = 11{,}584$ bits):

$$\eta = \frac{N_{TCP} \cdot MSS}{T} \qquad (1)$$

It can be seen from Figure 4 that all TCP congestion control algorithms experienced a drop in average throughput as we increased the number of devices in the last-hop WLAN, from scenario A to scenario D. However, we noticed an interesting phenomenon with TCP Reno, TCP Veno, and TCP Westwood+, in that they all experienced a slight increase in their average throughput in scenario E, i.e. when the number of 802.11g devices was increased from four to five. TCP Cubic and TCP Hybla

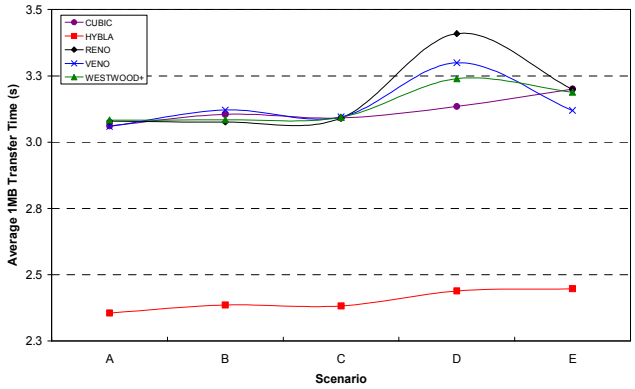

Figure 3. Small Transfers: Average Transfer Times

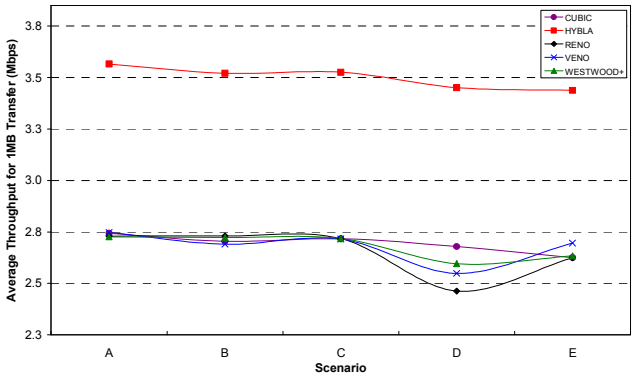

Figure 4. Small Transfers: Average Throughputs

continued to show a decrease, as would be expected due to the higher end-to-end latencies caused by high contention levels at the AP. Since each of the TCP congestion control algorithms on test relies on the RTT for advancing its *cwnd*, one would expect a gradual decrease in end-to-end throughput between the TCP server and SEAMOSS as the number of 802.11g devices increase. Another factor that may have affected the average end-to-end throughput for TCP is the increased frame errors over the WLAN. According to [19], 802.11 frame errors tend to be more prevalent as the number of 802.11g devices and hence TCP traffic levels increase due to the effect of 'self-collisions'. The 802.11 MAC at the AP is able to conceal the majority of errors through its stop-and-wait ARQ mechanism. However, not all 802.11 frames can be recovered because the AP's MAC will only attempt a fixed number of local retransmissions before giving up. This effectively becomes a lost TCP data segment that will not arrive at SEAMOSS, eventually causing it to produce a duplicate TCP ACK, which will cause TCP at the server to perform a *fast retransmit* action reducing its *cwnd* size, and consequently its sending rate, unnecessarily.

The most striking result in Figure 4 is that TCP Hybla achieves the highest average throughput across all scenarios, producing up to a 30% better performance than the next best algorithm, TCP Cubic. In scenario E, however, TCP Veno achieves a higher throughput than TCP Cubic,

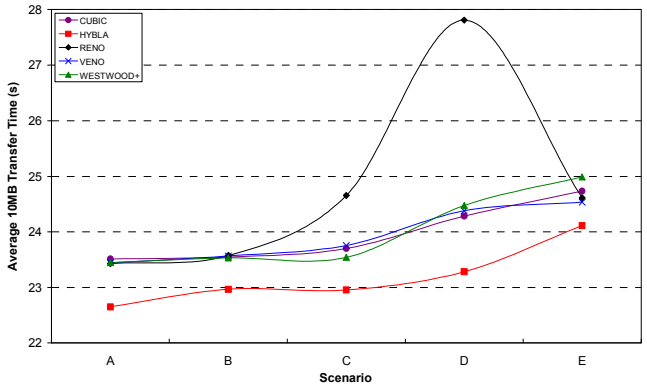

Figure 5. Medium Transfers: Average Transfer Times

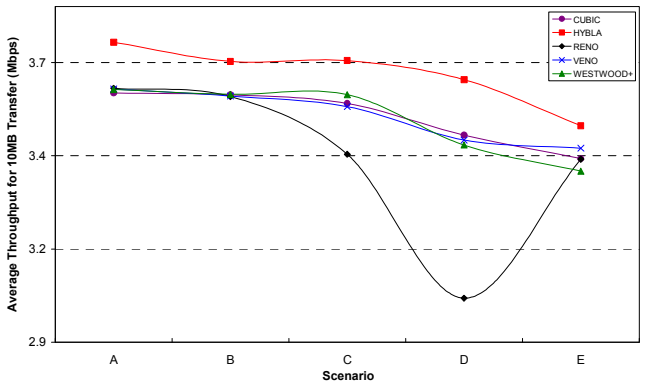

Figure 6. Medium Transfers: Average Throughputs

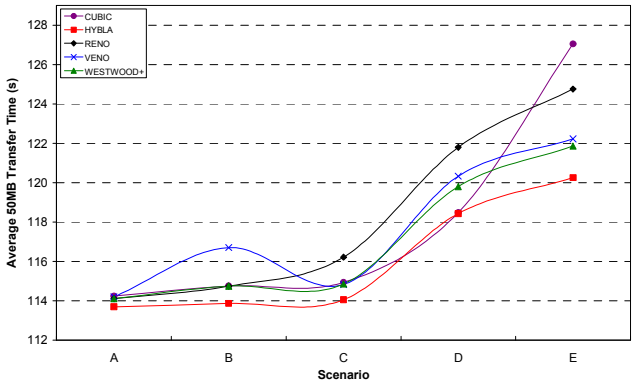

Figure 7. Large Transfers: Average Transfer Times

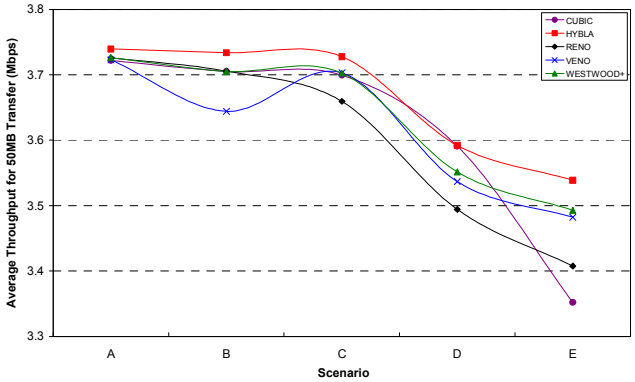

Figure 8. Large Transfers: Average Throughputs

perhaps suggesting its superiority when the number of 802.11g clients increases significantly, which would require further investigation for confirmation.

B. Mudium TCP Transfers

Figure 5 is a plot of the average times taken to transfer 10MB of data for each of the TCP algorithms on test. It can be seen that all TCP algorithms, except TCP Reno, follow a similar trend between scenarios A to E, possessing a gradual increase in the time taken to complete the transfer as the number 802.11g devices increases. TCP Hybla performed the best, by completing the transfer, on average, 1 second quicker in all scenarios. In contrast, TCP Reno exhibited a significant increase its transfer time from scenario B to scenario D inclusive. In scenario C it took nearly 2 seconds (~7%) longer than TCP Hybla to transfer the 10MB of data, and in scenario D it took nearly 5 seconds (~22%) longer than TCP Hybla. However, TCP Reno performed better than TCP Westwood+ and TCP Cubic in scenario E, where there were five 802.11g devices in the WLAN. A possible explanation for this sudden increase in performance could be due to the AP's DCF function at the 802.11g MAC that randomly allocates 'back-off periods' from its *contention window* to clients. Thus, five devices in the WLAN may have altered the DCF's randomness in favour of SEAMOSS acquiring access to the radio medium quicker than the other devices.

In Figure 6 we present a plot of the average throughput achieved over the lifetime of a 10MB data transfer to SEAMOSS. Because the *congestion avoidance* phase was more prevalent in these transfers, all TCP algorithms were able to probe for higher bandwidth in all scenarios. Hence the throughput values in Figure 6 are greater than those seen in Figure 4. We also noticed that up to scenario C (three 802.11g devices in the WLAN), all TCP algorithms experienced only a mild decrease in average throughput. When introducing a fourth (scenario D) and fifth (scenario E) device into the WLAN, the decrease in throughput was much more accelerated. Therefore, it could be said from our results that medium size TCP transfers are little affected by last-hop WLANs containing up to three 802.11g clients, but show a more rapid decline in performance when the number of devices exceeds this number.

C. Large TCP Transfers

Figure 7 is a plot of the average times taken to transfer 50MB of data for each of the TCP algorithms on test, and Figure 8 is a plot of the average throughput achieved over the lifetime of connections. It can be seen that from scenarios A to C, the average throughput (and subsequently transfer times) for each algorithm remains relatively constant, around the 3.7Mbps to 3.75Mbps (114s to 115s) mark. TCP Veno experienced a slight increase in the average transfer time in scenario B. From scenario C onwards, it can be seen that TCP Reno's performance

degradation was the most accelerated. In contrast, TCP Hybla achieved the highest throughput and quickest transfer times across all scenarios.

An interesting observation is that in scenario E (with five 802.11g devices), TCP Cubic was the worst performer, possessing a transfer time in excess of 127s, taking over 2s longer than TCP Reno to complete the transfer, and 7s longer than TCP Hybla, which we feel is quite a significant difference from an end-user's perspective. A possible explanation for this is that TCP Cubic is designed for wired-to-wireless paths, although its performance has been better than TCP Reno's with fewer 802.11g devices in the WLAN. This was more likely to be due to it's the cubic growth of its *cwnd*, initially possessing a 'steady state' behaviour, then with further time elapsing, exhibiting more of a bandwidth probing behaviour. In scenario E, the transfer times are the longest overall, and this may have put TCP Cubic's *cwnd* into the second bandwidth probing phase. Although this may work well for TCP Cubic in a wired network, a sudden accelerated opening of the *cwnd* in wired-to-wireless paths would cause large numbers of TCP segments to arrive at the AP. This can degrade performance of TCP Cubic in scenario E.

It can be seen from Figure 7 that in scenario E, TCP Hybla, TCP Westwood+, and TCP Veno offer better transfer times than TCP Reno and TCP Cubic. This confirms that the enhancements made in TCP Hybla, TCP Westwood+, and TCP Veno for wired-to-wireless paths do indeed produce better performances than those designed for traditional wired paths only. We have noticed that the differences between these two categories of algorithms are more noticeable in last-hop WLAN conditions consisting of at least three active 802.11g devices.

D. Retransmission Timeout (RTO) Timer

In Figure 9 we present the average RTO timer values for the *small* transfers; with the exception of TCP Reno. It can be seen that the average value of the RTO timer was fairly constant for all TCP algorithms. We calculated the average end-to-end RTT for all TCP algorithms across all five scenarios to be 175.7ms. It shows that TCP Reno's RTO timer value is quite large across all scenarios in relation to the average RTT. In contrast, the other TCP algorithms seem to all possess much lower RTO timer values, implying that their response times would be better over the WLAN. Recall that a large RTO timer value causes TCP to unnecessarily wait for longer time for an ACK to arrive from the WLAN. The likelihood of segments being lost over the WLAN is high, so TCP Reno is waiting on average up to three times longer than the other algorithms. Figure 3 and Figure 4 confirm this theory, which clearly show TCP Reno to be the worst performer for small TCP transfers between scenarios C to E. TCP Reno's over-inflated RTO timer computation may explain its degraded performance.

In Figure 10 we present the average RTO timer values for the *medium* transfers. Here TCP Reno shows great variation in its RTO timer value, experiencing a sudden

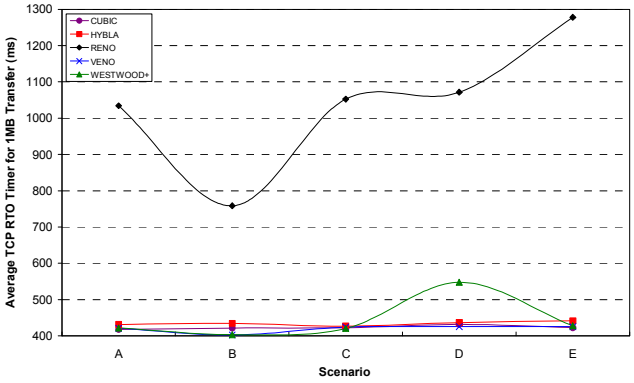

Figure 9. RTO Timer: Small TCP Transfers

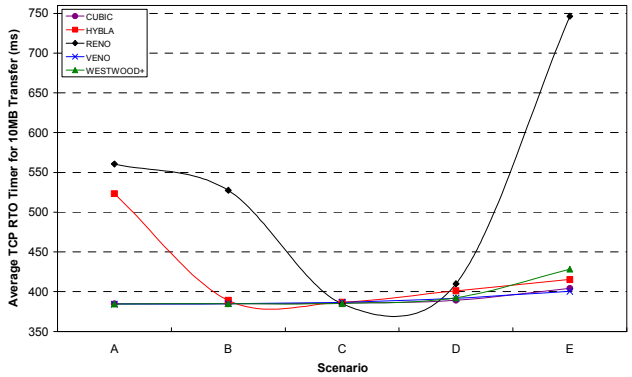

Figure 10. RTO Timer: Medium TCP Transfers

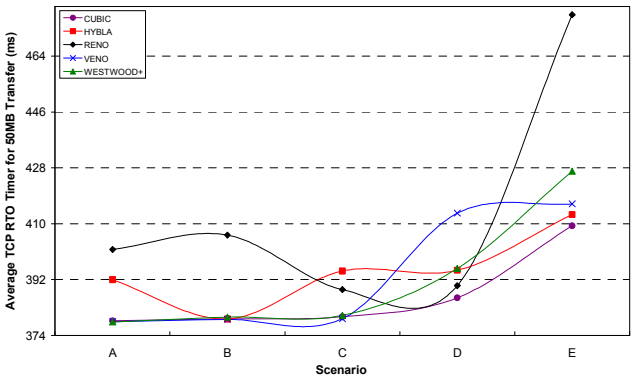

Figure 11. RTO Timer: Large TCP Transfers

increase from scenario D onwards. We calculated the average end-to-end RTT for all TCP algorithms across all five scenarios to be 184.7ms. Comparing all the algorithms, it appears that TCP Reno's RTO timer is highly sensitive to conditions over the last-hop WLAN, which again may explain its poor performance in such conditions, as can be seen in Figure 5 and Figure 6.

In Figure 11 we present the average RTO timer values for the *large* transfers. We calculated the average end-to-end RTT for all TCP algorithms across all five scenarios to be 181.2ms. TCP Reno's RTO timer value appears to exhibit similar behaviour to that seen in the *small* and *medium* TCP transfers, again highlighting its high variability and

sensitivity over such conditions. The other TCP algorithms displayed less variability, but generally all of them showed an increase in the values of their RTO timer as the number of 802.11g devices is increased. Here we feel as though the TCP RTO timer value should remain fairly constant across the scenarios, or even be reduced. Our justification for this is that one would expect channel conditions over the WLAN to be lossier in scenarios D and E. For this reason, it seems logical for TCP to reduce the magnitude of its RTO timer as opposed to increasing it. Further investigations would be needed to assess the impact of reducing the RTO timer with increasing 802.11g client numbers on end-to-end performances of TCP congestion control algorithms.

V. SUMMARY AND CONCLUSIONS

In this paper we present experimental results evaluating the performances of TCP Reno, TCP Cubic, TCP Hybla, TCP Veno, and TCP Westwood+ congestion control algorithms over wired-to-wireless connections. A purpose-built experimental testbed was used, which consisted of real-world implementations of the TCP algorithms, an emulated fixed network domain, and a real-world operational last-hop 802.11g home WLAN with up to five devices. Our evaluations based on the experimental results achieved are summarised below:

1. We discovered that, with the notable exception of TCP Reno, each of the recent TCP proposals performed on comparable terms, with TCP Hybla consistently being the best performer in all of our testing scenarios.
2. We observed from all our experiments that there was a clear turning point in the data transfer performances for all TCP algorithms when the number of 802.11g devices in the last-hop WLAN exceeded three, resulting in reduced performances in *small*, *medium*, and *large* sized data transfers from the wired domain.
3. With regards to the legacy TCP Reno, we noticed that it was actually the worst performer in our tests, fully supporting the many claims researchers have made about its poor performance over wired-to-wireless conditions.
4. From further observations into the retransmission timeout (RTO) timer values for each algorithm we tested, TCP Reno revealed some interesting insights that may explain its weak performance in wired-to-wireless paths, which may have been overlooked in previous work. We learnt that TCP Reno's RTO timer value was much more sensitive to changes in the last-hop WLAN.
5. Interestingly, TCP Cubic, although designed primarily for wired networks, was a strong performer in all of our tests. This technique seems to have worked well over our testbed scenarios, as the bandwidth over the last-hop WLAN was potentially higher than that of the wired path (shaped according to the Internet emulator).
6. Finally, we found that the performances of TCP Veno and TCP Westwood+ were on very close terms throughout our tests, exhibiting similar performances across all scenarios and data transfer sizes.

In this paper we have demonstrated a consistent method for performing standardised testing for TCP congestion control algorithms over wired-to-wireless paths, with our results yielding useful insights. Such insights can be a step towards the development and/or refinement of a TCP congestion control algorithm that is able to operate with maximum efficiency over an increasingly heterogeneous Internet.

REFERENCES

[1] M. Allman, V. Paxson, and W. Stevens, "TCP congestion control," RFC 2581, 1999.

[2] IEEE-802.11, "Part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications," IEEE Working Group for WLAN Standards, Edition 1999.

[3] J. Tourrilhes, "PiggyData: reducing CSMA/CA collisions for multimedia and TCP connections," Vehicular Technology Conference, 1999, pp. 1675-1679, vol.3.

[4] J.B. Postel, "Transmission control protocol," RFC 793, 1981.

[5] A.C.H. Ng, D. Malone, and D.J. Leith, "Experimental evaluation of TCP performance and fairness in an 802.11e test-bed," ACM SIGCOMM '05, pp. 17 - 22.

[6] C. Parsa and J.J. Garcia-Luna-Aceves, "Improving TCP performance over wireless networks at the link layer," Mobile Networks and Applications, vol. 5, 2000, pp. 57 - 71.

[7] P. Sarolahti and A. Kuznetsov, "Congestion control in Linux TCP," Monterey, CA, USA: 2002.

[8] C. Caini and R. Firrincieli, "TCP Hybla: a TCP enhancement for heterogeneous networks," Int. Journal of Satellite Communications and Networking, 2004, pp. 547-566.

[9] C.P. Fu and S.C. Liew, "TCP Veno: TCP enhancement for transmission over wireless access networks," IEEE Journal of Selected Areas in Communications (JSAC), vol. 21, 2003.

[10] Cui Lin et al., "Enhanced Westwood+ TCP for wireless/heterogeneous networks," Asia-Pacific Conference on Communications, 2006, pp. 1-5.

[11] L. Xu and I. Rhee, "CUBIC: A new TCP-friendly high-speed TCP variant," 3rd Int. Workshop on Protocols for Fast Long-Distance Networks, 2005.

[12] J. Padhye et al., "Modeling TCP Reno performance: a simple model and its empirical validation," IEEE/ACM Transactions on Networking, vol. 8, 2000, pp. 133-145.

[13] M. Mathis et al., "TCP Selective Acknowledgement Options," RFC 2018, 1996.

[14] NSF, "Web100 Project," http://web100.org/.

[15] R. Taank and Xiao-Hong Peng, "An experimental testbed for evaluating end-to-end TCP performance over wired-to-wireless paths," IEEE 5th Consumer Communications and Networking Conference, 2008, pp. 523-527.

[16] M. Allman and A. Falk, "On the effective evaluation of TCP," ACM CCR '99, vol. 5, 1999.

[17] V. Jacobson and M.J. Karels, "Congestion avoidance and control," ACM CCR, vol. 18, 1988, pp. 314 - 329.

[18] P. Sarolahti, M. Kojo, and K. Raatikainen, FRTO: A new recovery algorithm for TCP retransmission timeouts, University of Helsinki, 2002.

[19] S. Gopal, and D. Raychaudhuri, "Experimental evaluation of the TCP simultaneous-send problem in 802.11 wireless local area networks," ACM SIGCOMM Workshop on Experimental Approaches to Wireless Network Design & Analysis, 2005, pp.23-28.