TCP Westwood with Agile Probing: Dealing with Dynamic, Large, Leaky Pipes

Kenshin Yamada, Ren Wang, M.Y. Sanadidi, and Mario Gerla
Computer Science Department, University of California, Los Angeles
Los Angeles, CA 90095, USA
{kenshin, renwang, medy, gerla}@cs.ucla.edu

Abstract--TCP Westwood (TCPW) has been shown to provide significant performance improvement over high-speed heterogeneous networks. The key idea of TCPW is to use Eligible Rate Estimation (ERE) methods to set the congestion window (**cwnd*****) and slow start threshold (*****ssthresh*****) after a packet loss. ERE is defined as the transmission rate a sender ought to use to achieve high utilization and remain friendly to other TCP variants. This paper presents TCP Westwood with Agile Probing (TCPW-A), a sender-side only enhancement of TCPW. TCPW-A perform well when faced with highly dynamic bandwidth, large propagation time/bandwidth, and random loss in the current and future heterogeneous Internet. TCPW-A achieves its goal by incorporating the following two mechanisms: 1) When a connection initially begins or re-starts after a Timeout, instead of exponentially expanding*** **cwnd** ***to an arbitrary preset ssthresh and then going into linear increase, TCPW-A uses Agile Probing, a mechanism that repeatedly resets*** **ssthresh** ***based on ERE and forces*** **cwnd** ***into an exponential climb each time. The result is fast convergence to a more appropriate*** **ssthresh** ***value. 2) In Congestion Avoidance, TCPW-A invokes Agile Probing upon indication of unused extra bandwidth via a scheme we call Load Gauge (LG). Experimental results, both in Ns-2, and in measurements using FreeBSD implementation, show that TCPW-A can significantly improve link utilization over a wide range of bandwidth, propagation delay and dynamic network loading.***

I. Introduction

TCP has been widely used in the Internet for numerous applications. The success of the congestion control mechanisms introduced in [13] and their succeeding enhancements has been remarkable. The current implementation of TCP Reno/New-Reno runs in two phases: Slow Start and Congestion Avoi-dance. In Slow Start, upon receiving an acknowledgment, the sender increases the congestion window (*cwnd*) exponentially, doubling *cwnd* every Round-trip Time (RTT), until it reaches the Slow-start Threshold (*ssthresh*). Then, the connection switches to Congestion Avoidance, where *cwnd* grows more conservatively, by 1 packet every RTT (linearly). Then upon a packet loss, the sender reduces *cwnd* to half.

It is well known that the current TCP throughput deteriorates in high-speed heterogeneous networks, where many of the packet losses are due to noise and external interference over wireless links. When the Bandwidth-Delay Product (BDP) increases, another problem TCP faces is initial *ssthresh* setting. In many cases, initial *ssthresh* is set to an arbi-trary value, ranging from 4k bytes to arbitrarily high (e.g., maximum possible value), depending on implementation under various operating systems. By setting the initial *ssthresh* to an arbitrary value, TCP performance may suffer two potential problems: (1) if *ssthresh* is set too high relative to the network Bandwidth Delay Product (BDP), the exponential increase of *cwnd* generates too many packets too fast, causing multiple losses at the bottleneck router and coarse timeouts, with significant reduction of the connection throughput. (2) If the initial *ssthresh* is set too low, the connection exits Slow Start and switches to linear *cwnd* increase prematurely, resulting in poor startup utilization especially when BDP is large.

This research was supported by NSF under grant ANI-0221528.

Dynamic bandwidth presents yet another challenge to TCP performance. In today's heterogeneous Internet, bandwidth available to a TCP connection varies often due to many reasons, including multiplexing, access control, and mobility [9]. Standard TCP versions can handle bandwidth decrease fairly well by its *cwnd* and *ssthresh* "multiplicative decrease" upon a congestion loss. However, if a large amount of bandwidth becomes available for reasons such as wide-bandwidth-consuming flows leaving the network, TCP may be slow in catching up (as will be shown below), particularly in Congestion Avoidance with its "additive-increase" mechanism, increasing *cwnd* only by 1 packet per RTT. As a result, link utilization can be lacking, particularly in large propagation times and highly dynamic large bandwidth, or what we might call "large dynamic pipes".

TCP Westwood (TCPW) has been proposed in [5, 19] and shown to provide significant performance improvement, better scalability and stability [6] over high-speed, heterogeneous networks. After a packet loss, instead of simply cutting *cwnd* by half as in standard TCP, TCPW-A resets *cwnd* along with *ssthresh* according to the TCPW sender's Eligible Rate Estimate (ERE), thus maintain a reasonable window size in case of random losses, and preventing over-reaction when transmission speed is high. A brief overview of TCPW and ERE is given in Section II and detailed description can be found in [19].

In TCPW, ERE is only used to set *sshthresh* and *cwnd* after a packet loss. We realize that we can take advantage of ERE further when linear increase is too slow to ramp up *cwnd*. In this paper, we propose TCP Westwood with Agile Probing (TCPW-A), a sender-side only enhancement of TCPW, that deals well with highly dynamic bandwidth, large propagation times and bandwidth, and random loss in the current and future heterogeneous Internet. TCPW-A achieves this goal by incorporating the following two mechanisms into basic TCPW algorithm: The first mechanism is Agile Probing, which is invoked at connection start-up (including after a time-out), and after extra available bandwidth is detected. Agile Probing adaptively and repeatedly resets *ssthresh* based on ERE. Each time the *ssthresh* is reset to a value higher than the current one, *cwnd* climbs exponentially to the new value. This way, the sender is able to grow *cwnd* efficiently (but conservatively) to the maximum value allowed by current conditions without overflowing the bottleneck buffer with multiple losses – a problem that often affects traditional TCP. Agile Probing in TCP Startup phase is similar to Adaptive Start (Astart) that we have presented and evaluated in [18].

The second mechanism, proposed in this paper, concerns how to monitor extra unused bandwidth. We realized that if a TCP sender identifies the newly materialized extra bandwidth and invokes Agile Probing properly, the connection can converge to the desired window faster than usual linear increase. This also applies in the case when a random error occurs during start-up, causing a connection to exit Slow Start prematurely and switch to Congestion Avoidance. In this paper, we propose a Load Gauge mechanism, which identifies the availability of persistent extra bandwidth in Congestion Avoidance, and invokes Agile Probing accordingly. Experimental results, both in ns-2 simulation and lab measurements, show that TCPW-A can significantly improve link utilization under a wide range of system parameters.

The remainder of the paper is organized as follows. In Section II, we give an overview of TCP Westwood and Agile Probing. In Section III, We introduce the Load Gauge (LD) mechanism. Simulation Results are provided in Section IV, and lab measurement results in Section V. Section VI discusses related work. Finally, section VII discusses future work and concludes the paper.

II. TCP Westwood Overview

In TCP Westwood (TCPW), a sender continuously monitors ACKs from the receiver and computes its current Eligible Rate Estimate (ERE) [19]. ERE relies on an adaptive estimation technique applied to the ACK stream. The goal of ERE is to estimate the connection eligible sending rate with the goal of achieving high utilization, without starving other connections. Research on active network estimation [8] reveals that samples obtained by "packet pair" is more likely to reflect link capacity, while samples obtained by "packet train" give short-time throughput. In TCPW, the sender adaptively computes T_k, an interval over which the ERE sample is calculated. A ERE sample is computed by the amount of data in bytes that were successfully delivered in T_k. T_k depends on the congestion level, the latter measured by the difference between 'expected rate' and 'achieved rate' as in TCP Vegas. That is T_k depends on the network congestion level as follows:

$$T_k = RTT \times \frac{cwin/RTT_{\min} - RE}{cwin/RTT_{\min}}, \quad (1)$$

where $\mathrm{RTT_{min}}$ is the minimum RTT value of all acknowledged packets in a connection, and RTT is the smoothed RTT measurement. The expected rate of the connection when there is no congestion is given by *cwnd*/$\mathrm{RTT_{min}}$, while RE is the achieved rate computed based on the amount of data acknowledged during the latest RTT, and exponentially averaged over time using a low-pass filter. When there is no congestion, and therefore no queuing time, *cwnd*/$\mathrm{RTT_{min}}$ is almost the same as RE, producing small T_k. In this case, ERE becomes close to a packet pair measurement. On the other hand, under congestion conditions, RE will be much smaller than *cwnd*/$\mathrm{RTT_{min}}$ due to longer queuing delays. As a result, T_k will be larger and ERE closer to a packet train measurement. After computing the ERE samples, a discrete version of a continuous first order low-pass filter using the Tustin approximation [4] is applied to obtain smoothed ERE.

In current TCPW implementation, upon packet loss (indicated by 3 DUPACKs or a timeout) the sender sets *cwnd* and *ssthresh* based on its current ERE. TCPW sets its *ssthresh* to ERE×$\mathrm{RTT_{min}}$, and *cwnd* to *ssthresh* (*cwnd* to 1 in case of timeout). For more details on TCPW and ERE, and its performance evaluation in high-speed, error-prone environments, please refer to [5] and [19].

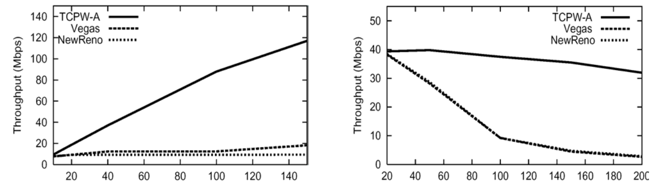
A. Agile Probing

In this Section, we describe Agile Probing scheme that improves TCP performance during start-up, and over large dynamic pipe with the help of Load Gauge, which we will introduce in this paper.

Agile Probing uses ERE to adaptively and repeatedly reset *ssthresh*. During Agile Probing, when the current *ssthresh* is lower than ERE, the sender resets *ssthresh* higher accordingly, and increases *cwnd* exponentially. Otherwise, *cwnd* increases linearly to avoid overflow. In this way, Agile Probing probes the available network bandwidth for this connection, and allows the connection to eventually exit Slow-start close to an ideal window corresponding to its share of path bandwidth.

By repeating cycles of linear increase and exponential increase, *cwnd* adaptively converges to the desired window in a timely manner, enhancing link utilization in Slow Start.

We highlight the performance comparison of Agile Probing with other TCP favors in Fig. 1. All simulation results in this paper are obtained using the ns-2 simulator [16]. The initial *ssthresh* for NewReno is set to be 32 Kbytes, a common default value among many implementations.



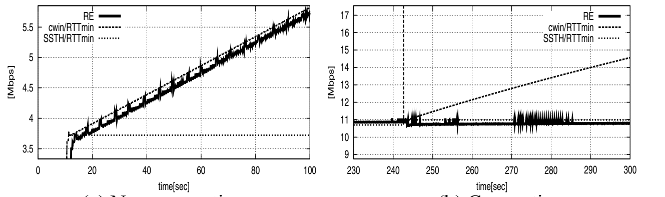
(a) Throughput vs. bottleneck capacity (b) Throughput vs. propogation time
Fig 1. Throughput comparison of TCPW-A, Vegas and NewReno (first 20 sec, rtt=100msec)

Fig. 1 examines the throughput of Agile Probing, NewReno, and Vegas under varied bottleneck bandwidth and propagation delay. The results show that Agile Probing achieves much higher throughput, and scales well with bandwidth or propagation delay increase. For more detail on Agile Probing, please refer to [18].

III. Load Gauge

In this section, we present a Load Gauge (LG) mechanism that aims at monitoring extra available bandwidth and invoking Agile Probing accordingly. In Congestion Avoidance, a connection monitors the congestion level constantly. If a TCP sender detects light load conditions, which indicates that the connection may be eligible for more bandwidth, the connection invokes Agile Probing to capture such bandwidth and improve utilization.

As described in section II, Rate Estimate (RE) is an estimate of the rate achieved by a connection. If the network is not congested and extra bandwidth is available, RE will increase as *cwnd* increases. On the other hand, if the network is congested, RE flattens despite of the *cwnd* increase. Fig. 2(a) illustrates the Expected Rate, which is equal to *cwnd*/$\text{RTT}_{\min}$, and RE in non-congested path; while Fig. 2(b) shows Expected Rate and RE under congestion. Also shown in these figures, are the *ssthresh/RTTmin* plots. Such values correspond to the "initial" expected rate. That is the expected rate when Congestion Avoidance was entered, or after a packet loss. From Fig. 2(a), we see that RE follows *cwnd*/$\text{RTT}_{\min}$ continuously in non-congestion case. On the other hand, Fig. 2(b) shows that RE does not grow and remains equal to *ssthresh*/$\text{RTT}_{\min}$ under congestion. In this case, RTT increases by an amount equal to the queuing time at the bottleneck. Then, this RTT's growth cancels out the *cwnd* increase, and keeps RE constant, as it should.



(a) Non congestion (b) Congestion

Fig 2. RE, *cwnd*, *ssthresh* dynamics

As mentioned before, *cwnd*/$\text{RTT}_{\min}$ indicates Expected Rate in no congestion and RE is the achieved rate. To be more precise, RE is the Achieved Rate corresponding to the Expected Rate 1.5 times RTT earlier[1]. Thus, we must use in a comparison, the corresponding Expected Rate, that is (*cwnd* - 1.5)/$\text{RTT}_{\min}$. RE tracks the Expected Rate in non-congestion conditions, but flattens, remaining close to the initial Expected Rate (*ssthresh* / $\text{RTT}_{\min}$) under congestion. We define the Congestion Boundary as

$$CongestionBoundary = \beta \cdot ExpectedRate + (1 - \beta) \cdot InitialExpectedRate \qquad 0 < \beta < 1 \qquad (2)$$

Fig. 3 illustrates the relation among Congestion Boundary, Expected Rate and Initial Expected Rate with $\beta=0.5$.

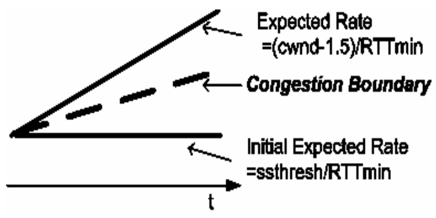


Fig 3. Congestion Boundary, Expected Rate and Initial Expected Rate with β=0.5

RE may fluctuate crossing above and below the Congestion Boundary. To declare light load condition (indicated by persistent non-congestion), we use a (*non-congestion*) counter, which increases by one every time RE is above the Congestion Boundary and decreases by one if RE is below the Congestion Boundary. A pseudo code of the LG algorithm is as follows:

```
if ( in Congestion Avoidance except for the initial two RTT){
  if ( RE >Congestion Boundary){
    no_congestion_counter++;
  else if (no congestion_counter > 0){
    no_congestion_counter--;
  if (no_congstion_counter > cwnd){
        re-start Agile Probing;
}else{
  no_congestion_counter = 0;
  }
```

If the parameter β is greater than 0.5, the Congestion Boundary line gets closer to Expected Rate. We can make this algorithm more conservative by setting $\beta > 0.5$.

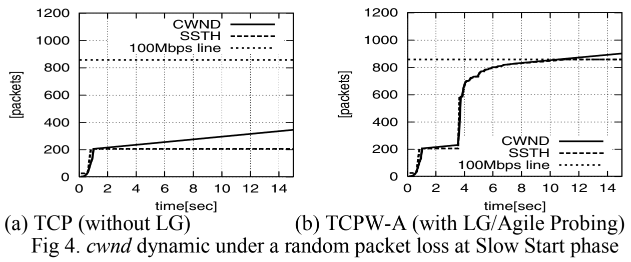
Even if the LG algorithm can accurately indicate light load condition, there is always the possibility that the network becomes congested immediately after the connection switches to Agile Probing phase. One such scenario is after a buffer overflow at the bottleneck router. Many of the TCP connections may decrease their *cwnd* after a buffer overflow, and congestion is relieved in a short time period. The LG mechanism in some connection may observe light load and invoke Agile Probing. However, the erroneous indication is not a serious problem. Unlike exponential *cwnd* increase in Slow Start phase of NewReno, the TCP connection adaptively seeks the fair share estimate in Agile Probing mode. Thus, if the network has already been congested when a new Agile Probing begins, the "Agile Probing" connection will not increase *cwnd* much, and will go back to linear probing soon enough.

IV. Simulation Results

In this section, we evaluate the performance of our TCP Westwood with Agile Probing (TCPW-A) algorithms in terms of throughput, friendliness, and window dynamics. The results show that TCPW-A exhibits significantly improved performance, yet remains friendly toward TCP NewReno, the de facto TCP standard over the Internet.

A. *Premature Exit from Slow Start*

Fig. 4 shows *cwnd* dynamics under random packet loss during Slow Start. The bottleneck link bandwidth is 100Mbps, two-way propagation delay is 100msec, and the bottleneck buffer is equal to the BDP. When *cwnd/RTTmin* reaches 2Mbps, a packet is dropped (assumed to be random loss). The connection exits Slow Start phase and enters Congestion Avoidance. Without LG, *cwnd* increases slowly, one packet every RTT, requiring more than 60 seconds for *cwnd* to reach BDP. With the help of LG, the TCPW-A connection detects light load condition within a few seconds, and then starts a new Agile Probing again to utilize the bandwidth efficiently.



(a) TCP (without LG) (b) TCPW-A (with LG/Agile Probing)

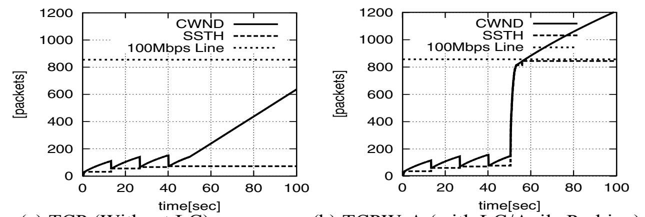
Fig 4. *cwnd* dynamic under a random packet loss at Slow Start phase

B. *Dynamic bandwidth*

To illustrate how TCPW-A behaves under dynamic bandwidth, Fig. 5 shows *cwnd* dynamics when non-responsive UDP flows are gone from the path, causing extra bandwidth to

[1] The oldest acknowledged packet used for RE calculation is received one RTT before. Packets are traveled back and forth for RTT time period. Thus, the oldest ACK used for ERE calculation is sent two RTT before, and the newest ACK is sent one RTT before. On average, the ACK packets used for RE calculation are sent 1.5 RTT before. Then, the '*cwnd* at 1.5 RTT before' becomes (*cwnd* – 1.5).

become available. The bottleneck link bandwidth is 100Mbps, two-way propagation delay is 100msec, and the bottleneck buffer is set to BDP.



(a) TCP (Without LG) (b) TCPW-A (with LG/Agile Probing)

Fig 5. *cwnd* dynamic when dominant flows are gone from the bottlneck router

The non-responsive UDP flows have disappeared from the path around 50 seconds, and the remaining flow is eligible to use the newly materialized bandwidth. Without Load Gauge, the connection needs 60 seconds to reach BDP. On the other hand, Load Gauge indicates the unused bandwidth within a few seconds, and a new Agile Probing phase makes instant use of this unused bandwidth possible! Note that dynamic bandwidth due to other reasons stated in Introduction will induce similar behavior that helps to improve utilization.

C. *Throughput Comparison Under Dynamic load*

We evaluated the performance of TCPW-A under highly dynamic load conditions. In 20 minutes simulation time, we ran 100 connections, each with a lifetime of 30 seconds. The starting time of the connections are uniformly distributed over the simulation time. Thus, the set of connections is randomly spread out over 20 minutes simulation time, making the bandwidth available to a connection quite oscillating. The initial *ssthresh* of TCP NewReno is set to default value, 32k bytes.

Fig. 6 shows Total throughput vs. bottleneck bandwidth. The total throughput is computed as the sum of throughputs of all connections. Two-way propagation delays are 70ms, and the bottleneck buffer size is set equal to the pipe size (BDP). As the bottleneck link capacity increases, TCPW-A exhibits much better scalability than NewReno. At 150 Mbps, TCPW-A achieves at least twice as much throughput as NewReno does.

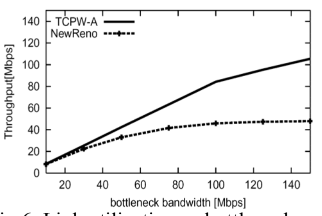


Fig 6. Link utilization vs bottleneck link capacity (RTT=70msec)

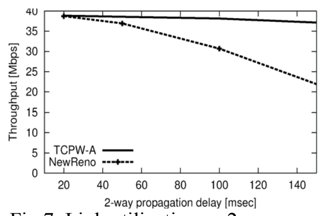


Fig 7. Link utilization vs 2-way propa-gation time

Fig. 7 shows the total throughput vs. two-way propagation delay. The bottleneck bandwidth is 45 Mbps, and the bottleneck buffer size is set equal to the pipe size (BDP). NewReno performance degrades as the propagation delays increase, showing lack of scalability to long propagation time. TCPW-A, on the other hand, scales well with increasing propagation times.

Note that there are two factors that account for TCP NewReno's inability to scale with bandwidth and RTT. One is its slow (linear) *cwnd* increase even when other connections leave the network and more bandwidth becomes available. The other factor is due to its small initial *ssthresh,* causing premature exit from Slow Start. One can argue that increasing the initial *ssthresh* easily solves this problem. However, a very large initial *ssthresh* may risk the connection into multiple packet losses and reduce the utilization even further. As also pointed out in [3], we think it is best to adaptively figure out the *ssthresh* value on the fly.

We also examined how well TCPW-A coexists with NewReno, the results show that TCPW-A connections may benefit initially by quickly reaching cruising speed, but they reach the same *cwnd* after a few congestion losses.

V. LAB MEASUREMENTS RESULTS

To evaluate TCPW-A performance in actual systems, we have implemented TCPW-A algorithms, including Agile Probing and Load Gauge on FreeBSD [22]. Lab measurements confirmed our simulation results, showing that TCPW-A behaves quite well in actual systems.

In our lab measurement experiments, all PCs are running on FreeBSD Release 4.5. We use Dummynet [21] to emulate the bottleneck link. The bottleneck link (from PC router to TCP receiver) speed is set to 10 Mbps. The propagation time is 400msec, and the router buffer is set equal to BDP (500 Kbytes), equivalent to 62 packets. Queue management at the router is Drop-tail. We use Iperf [23] as a traffic generator. The receiver's advertised window is set large enough at 4Mbytes. The initial *ssthresh* for TCP NewReno is set to be 32 Kbytes in our measurements.

Fig. 8 shows measured *cwnd* dynamics in TCPW-A and NewReno connections during Slow Start. NewReno enters Congestion Avoidance phase after *cwnd* reaches the initial *ssthresh* (32k), and *cwnd* increases linearly by one packet per RTT. *cwnd* reaches only 2Mbps, 20 % of the link capacity, for the first 25 seconds. On the other hand, in the TCPW-A connection, *cwnd* quickly converges to the link capacity by Agile Probing, We can also confirm that *cwnd* growth is not exponential throughout Agile Probing. Initially *cwnd* increases rapidly for the first 3 seconds, and then increases more slowly as the connection approaches the link capacity.

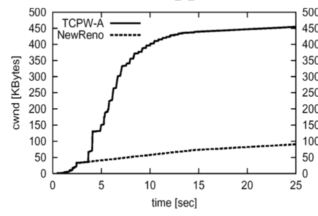


Fig 8. *cwnd* dynamics in TCPW-A and NewReno at Start-up

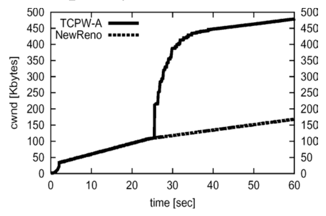


Fig 9. *cwnd* dynamics in TCPW-A and NewReno with prexisting flows in (0,25 sec)

Fig. 9 compares *cwnd* dynamics in TCPW-A to that of NewReno as obtained from lab measurements. When a connection starts, the path is already filled with non-responsive UDP flows. NewReno enters Congestion Avoidance after *cwnd* reaches the initial *ssthresh* (32k). TCPW-A connection does not increase *cwnd* much since the bottleneck link has been already largely occupied. The dominant flows are finished after 25 seconds, and the entire link capacity becomes available to the remaining connection. The TCPW-A connection tracks the ligh load condition, and invokes Agile Probing to quickly capture the unused bandwidth. On the other hand, NewReno stays in Congestion Avoidance and linearly (and slowly) increases its *cwnd*

VI. RELATED WORK

A great deal of research effort has been made to enhance TCP performance for dynamic, large, leaky pipes. There are several other approaches to improving TCP scalability to large pipes that require only sender-side modification, including Scalable TCP [15], HighSpeed TCP [10], and Vegas-based FAST TCP [7]. In these schemes, as in traditional TCP, packet losses are exclusively treated as congestion signals. Compared to the previous schemes, TCPW-A is equipped with the ability to better handle random errors in high-speed heterogeneous networks. EXplicit Control Protocol (XCP) [14] is a well-designed congestion control scheme for high-speed, long delay networks. However, it requires cooperation from routers and receivers, making it difficult to deploy.

With the increase of short-lived web traffic [11], researchers realize that start-up performance is important, especially over large-pipes. The Agile Probing scheme in TCPW-A provides a realistic means to figure out the right *ssthresh* on the fly. A variety of other methods have been recently suggested in the literature to avoid multiple losses and to achieve higher utilization during Slow Start. A larger initial *cwnd*, roughly 4Kbytes, is proposed in [2]. This could greatly speed up transfers with only a few packets. However, the improvement is still inadequate when BDP is very large, and the file to transfer is bigger than just a few packets [20]. Fast start [17] uses *cwnd* and *ssthresh* cached from recent connections to reduce the transfer latency. The cached parameters may be too aggressive or too conservative when network conditions change. In [12], Hoe proposes to set the initial *ssthresh* to the BDP estimated using packet pair measurements. This method can be too aggressive when the bottleneck buffer is not big enough, or many flows are co-existing. In [20], SPAND (Shared Passive Network Discovery) has been proposed to derive the optimal initial values for TCP parameters. SPAND needs leaky bucket pacing for outgoing packets, which can be costly and problematic in practice [1].

To deal with dynamic bandwidth, TCP-EBN (Early Bandwidth Notification) is proposed in [9]. In TCP-EBN, a TCP sender increases or decreases its *cwnd* according to a bandwidth estimate that is sent to it from routers. Thus, this scheme relies on router cooperation and is therefore not an "end-to-end" approach to the problem at hand. Also, the router has to either keep per-flow state to get an accurate estimate, with the resultant scalability problem; or assumes that flows share bandwidth fairly, which is not true often. Comparing to this scheme, and XCP (which takes advantage of router feedback to swiftly adjust *cwnd* thus can handle dynamic bandwidth), TCPW-A only requires sender-side modification, thus much easier to deploy.

VII. CONCLUSION AND FUTURE WORK

In this paper we introduced TCP Westwood with Agile Probing (TCPW-A), a sender side only modification of TCP that addresses the issues of highly dynamic bandwidth, large delays, and random loss. Besides basic TCPW schemes presented in [5,19], TCPW-A incorporates two new mechanisms: Agile Probing and Load Gauge. Agile Probing enhances probing during Slow Start and whenever persistent light load conditions are observed. Agile Probing converges to more appropriate *ssthresh* values thereby making better utilization of large pipes, and reaching "cruising speeds" faster, without causing multiple packet losses. Another contribution of this work is the introduction of the Load Gauge (LG) mechanism. LG is shown to be effective in detecting light load conditions, upon which TCPW-A invokes Agile Probing. The combination ensures that during Congestion Avoidance, TCPW-A can make quick use of bandwidth that materializes because of dynamic loads among other causes.

The results presented above were obtained using both simulation and lab measurements with actual implementation under FreeBSD operating systems. The results show that TCPW-A works as well in actual system as it does in simulation experiments.

In the future, we will evaluate TCPW-A further in terms of friendliness, more complex topologies, and interaction with active queue management schemes. We also plan to move our measurements from the lab to the Internet and on satellite links.

REFERENCES

[1] A. Aggarwal, S. Savage, T.E. Anderson, "Understanding the Performance of TCP Pacing," In Proceedings IEEE INFOCOM 2000, Tel Aviv, Israel, March 2000.

[2] M. Allman, S. Floyd and C. Patridge, "Increasing TCP's initial Window", INTERNET DRAFT, April 1998.

[3] M. Allman, end2end-interest discussion group, July, 2003. http://www.postel.org/pipermail/end2end-interest/2003-July.txt.

[4] K. J. Astrom, and B. Wittenmark, "Computer controlled systems," Prentice Hall, Englewood Cliffs, N. J., 1997.

[5] C. Casetti, M. Gerla, S. Mascolo, M. Y. Sanadidi, and R. Wang, "TCP Westwood: bandwidth estimation for enhanced transport over wireless links," In Proceedings of Mobicom 2001, Rome, Italy, Jul. 2001.

[6] J. Chen, F. Paganini, R. Wang, M. Y. Sanadidi, M. Gerla " Fluid-flow Analysis of TCP Westwood with RED ", Proceedings of Globecom 2003, San Francisco, Ca., November 2003.

[7] D.H. Choe, and S.H. Low, "Stabilized Vegas", In Proc. of IEEE/INFOCOM 2002, San Francisco, USA, April, 2002.

[8] C. Dovrolis, P.Ramanathan and D. Moore, "What Do Packet Dispersion Techniques Measure?," In Proceedings of Infocom 2001, Anchorage AK, April 2001.

[9] D. Dutta and Yongguang Zhang, "An Early Bandwidth Notification (EBN) Architecture for Dynamic Bandwidth Environments," Proceedings of IEEE International Conference on Communications (ICC'02), Apr 2002.

[10] S. Floyd, "HighSpeed TCP for Large Congestion Windows", Internet draft draft-ietf-tsvwg-highspeed-01.txt, work in progress, August 2003.

[11] L. Guo and I. Matta. The War between Mice and Elephants. In Proceedings of ICNP'2001: The 9th IEEE International Conference on Network Protocols, Riverside, CA, November 2001.

[12] J. C. Hoe, "Improving the Start-up Behavior of A Congestion Control Scheme for TCP", Proc. ACM SIGCOMM '96, pp. 270-280.

[13] V. Jacobson, "Congestion avoidance and control," ACM Computer Communications Review, 18(4) : 314 - 329, Aug. 1988.

[14] D. Katabi, M. Handley, and C. Rohrs, "Internet Congestion Control for Future High Bandwidth-Delay Product Environments." In Proceedings of Sigcomm 2002.

[15] T. Kelly, "Scalable TCP: Improving Performance in Highspeed Wide Area Networks", Submitted for publication, December 2002.

[16] NS-2 Network Simularor (ver.2.) LBL, URL: http://www.mash.cs.berkley.edu/ns/

[17] V.N. Padmamabhan and R.H. Katz, "TCP Fast Start: A Technique for Speeding Up Web Transfers", Proceedings of IEEE globecom'98, Sydney, Australia, Nov. 1998.

[18] R. Wang, G. Pau, K. Yamada, M. Sanadidi and M.Gerla, "TCP Start up Performance in Large Bandwidth Delay Networks", IEEE INFOCOM 2004, Hong Kong, March, 2004.

[19] R. Wang, M. Valla, M.Y. Sanadidi and M. Gerla, "Using Adaptive Bandwidth Estimation to provide enhanced and robust transport over heterogeneous networks", 10th IEEE International Conference on Network Protocols (ICNP 2002), Paris, France, Nov. 2002.

[20] Y. Zhang, L. Qiu and S. Keshav, "Optimizing TCP Start-up Performance", Cornell CSD Technical Report, February, 1999.

[21] Luigi Rizzo, "Dummynet: a simple approach to the evaluation of network protocols", ACM Computer Communication Review, 1997

[22] FreeBSD Project, URL: http://www.freebsd.org/

[23] Iperf Version 1.7.0, URL: http://dast.nlanr.net/Projects/Iperf/