TCP Westwood: Congestion Window Control Using Bandwidth Estimation

Mario Gerla, M. Y. Sanadidi, Ren Wang, and Andrea Zanella
UCLA Computer Science Department

Claudio Casetti
Politecnico Di Torino

Saverio Mascolo
Politecnico Di Bari

Abstract – **We study the performance of TCP Westwood (TCPW), a new TCP protocol with a sender-side modification of the window congestion control scheme. TCP Westwood controls the window using end-to-end rate estimation in a way that is totally transparent to routers and to the destination. Thus, it is compatible with any network and TCP implementation. The key innovative idea is to continuously estimate, at the TCP sender, the packet rate of the connection by monitoring the ACK reception rate. The estimated connection rate is then used to compute congestion window and slow start threshold settings after a congestion episode. Resetting the window to match available bandwidth makes TCPW more robust to sporadic losses due to wireless channel problems. These often cause conventional TCP to overreact, leading to unnecessary window reduction. Experimental studies of TCPW show significant improvements in throughput performance over Reno and SACK, particularly in mixed wired/wireless networks over high-speed links. The contributions of this paper include a model for fair and friendly sharing of the bottleneck link and a Markov Chain performance model in presence of link errors/loss. TCPW performance is compared to that of TCP Reno, and analytic results are validated against simulation results. Internet and laboratory measurements using a Linux TCPW implementation are also reported, providing further evidence of the gains achievable via TCPW.**

I. INTRODUCTION AND RELATED WORK

The Transmission Control Protocol (TCP) protocol provides end-to-end, reliable, congestion controlled connections over the Internet [1][2]. The congestion control method includes two phases: slow-start and congestion avoidance [3]. Enhanced recovery from sporadic errors is provided by Fast Retransmission and Fast recovery [4]. Nowadays, TCP is called upon to provide reliable and efficient data transfer over communication paths with ever increasing bandwidth-delay product, and over a variety of link technologies including wired (i.e., cable and fiber optic), ground radio, and satellite links. More losses due to link failures, independent as well as correlated, are expected over these wireless links. The new high speed wired/wireless environment is expanding the domain for which TCP was initially developed, tested and tuned. As a consequence, active research is in progress to extend the domain of effective TCP operability [5][6][7][8][9].

TCP Westwood [10] design adheres to the end-to-end transparency guidelines set forth in [11]. Namely, a simple modification of the TCP source protocol stack which allows the source to estimate the available bandwidth, and to use the bandwidth estimation to recover faster, thus achieving higher throughput. TCP Westwood exploits two basic concepts: the end-to-end estimation of the available bandwidth, and the use of such estimate to set the slow start threshold and the congestion window. It is worth underscoring that TCPW does not require any intervention from network layer or proxy agents.

TCPW source continuously estimates the packet rate of the connection by properly averaging the rate of returning ACKs. The estimate is used to compute the "permissible" congestion window and slow start threshold to be used after a congestion episode is detected, that is, after three duplicate acknowledgments or after a timeout. The rationale of this strategy is simple: in contrast with TCP Reno, which simply halves the congestion window after three duplicate ACKs, TCP Westwood (TCPW) attempts to make a more "informed" decision. It selects a slow start threshold and a congestion window that are consistent with the effective connection rate at the time congestion is experienced. We call such mechanism *faster recovery*.

The use of bottleneck bandwidth and connection rate estimation has been proposed before in the TCP literature. The best-known examples are Packet Pair [12] and TCP Vegas [13]. Both of these schemes use the bandwidth computation to estimate the bottleneck backlog. The larger the backlog, the larger the congestion. The backlog feedback is used in both cases to control the send window. The PP scheme achieves perfect bottleneck fair sharing. Unfortunately, it requires per flow queuing and Round Robin scheduling – a feature not available in commercial routers.

Also related to this work is the probing of "available bandwidth" on a path. Allman and Paxson in [14] report and compare techniques for probing the available bandwidth in order to properly initialize the *ssthresh* before a TCP connection is started. An excessively large *ssthresh* can lead to premature timeout, slow start and efficiency loss. Lai and Baker in [15] describe an improved measurement technique (packet tailgating) to probe available bandwidth that is less intrusive (i.e., consumes less bandwidth) than previous techniques. The above techniques share with our scheme the notion of measuring the inter–packet delay gaps and deriving bandwidth information from it. They do, however, probe the available bandwidth *before* the connection is started. This is a very different (and potentially much more difficult) problem than measuring the actual rate that a connection is achieving *during the date transfer*.

The "key innovation" of TCPW is to use the bandwidth estimate "directly" to drive the window, instead of using it to compute the backlog. The rationale is that if a connection is currently achieving a given rate, then it can safely use the window corresponding to that rate without causing congestion in the network. The careful reader will notice that this "aggressive" behavior may still lead to unfairness. In a later section we show that fairness is in fact guaranteed.

TCPW offers a number of features that are not available in TCP Reno and SACK. For example, the knowledge of the available bandwidth can be used to adjust the rate of a variable rate source (assuming such source is controlled by TCP). In this paper we focus on the behavior of TCPW in random packet loss, caused by link error or wireless interference instead of by buffer overflow. Like Reno, TCPW cannot distinguish between buffer overflow loss and random loss. However, in presence of random loss, TCP Reno overreacts and reduces the window by half. TCPW, on the other hand, after packet loss and retransmission timeout, resumes with the previous window as long as the bottleneck is not yet saturated (i.e., no buffer overflow).

To prevent the unnecessary window reduction of TCP Reno in case of random packet loss, and more precisely the loss caused by wireless links, several schemes have been proposed [16]. Probably, the most popular is the Snoop protocol [17]. Snoop carries out locally the retransmissions of packets lost on wireless links. In order to do so, it monitors TCP segments, stores copies, and retransmits them when it detects that they were lost. In the reverse direction, it suppresses TCP duplicate ACKs on their way to TCP senders in order to avoid retransmission from the TCP source. Other approaches to the wireless/wired problem include Split Connections, and "end-to-end" TCP modifications such as Explicit Loss Notification (ELN) [18][19]. All the above schemes have their pros and cons. They all share, however, a common feature. They require the cooperation of intermediate routers/proxies. TCPW does not require any specific intervention/support from intermediate route, thus preserving the original "end to end design" principle.

In order to study the behavior of TCPW in presence of random errors, an analytic model was developed using Markov chain techniques. This model is an important contribution in that it provides further insight in TCPW operation. Moreover, it allows us to cross validate simulation and measurements.

The paper is organized as follows. In Section II, we provide an overview of the TCPW algorithm. In Section III, an informal proof of TCPW convergence to fair share is presented. In Section IV, we report analytic and experimental performance results. Section presents the analytic model and results comparing TCPW and Reno, while Section VI includes a discussion of the Internet measurements results. Section VII concludes the paper.

II. An Overview of TCP Westwood

In TCP Westwood the sender continuously computes the connection BandWidth Estimate (BWE) that is defined as the share of bottleneck bandwidth used by the connection. Thus, BWE is equal to the rate at which data is delivered to the TCP receiver. The estimate is based on the rate at which ACKs are received and on their payload. After a packet loss indication, (i.e., reception of 3 duplicate ACKs, or timeout expiration), the sender resets the congestion window and the slow start threshold based on BWE. More precisely, cwin=BWE x RTT.

To understand the rationale of TCPW, note that BWE varies from flow to flow sharing the same bottleneck; it corresponds to the rate actually achieved by each INDIVIDUAL flow. Thus, it is a FEASIBLE (i.e., achievable) rate by definition. Consequently, the collection of all the BWE rates, as estimated by the connections sharing the same bottleneck, is a FEASIBLE set. When the bottleneck becomes saturated and packets are dropped, TCPW selects a set of congestion windows that correspond exactly to the measured BWE rates and thus reproduce the current individual throughputs. The solution is feasible, but it is not guaranteed per se to be "fair share." An additional property of this scheme, described in Section III, drives the rates to the same equilibrium point and makes it "fair share" under uniform propagation delays and single bottleneck assumptions.

Another important element of this procedure is the RTT estimation. RTT is required to compute the window that supports the estimated rate BWE. Ideally, the RTT should be measured when the bottleneck is empty. In practice, it is set equal to the overall minimum round trip delay (RTTmin) measured so far on that connection (based on continuous monitoring of ACK RTTs).

A. Setting cwin and ssthresh in TCPW

Further details regarding bandwidth estimation are provided in following sections. For now, let us assume that a sender has determined the connection bandwidth estimate (BWE), and let us describe in this section how BWE is used to properly set cwin and ssthresh after a packet loss indication.

First, we note that in TCPW, congestion window increments during slow start and congestion avoidance remain the same as in Reno, , that is they are exponential and linear, respectively. A packet loss is indicated by (a) the reception of 3 DUPACKs, or (b) a coarse timeout expiration. In case the loss indication is 3 DUPACKs, TCPW sets cwin and ssthresh as follows:

```
if (3 DUPACKs are received)
    ssthresh = (BWE * RTTmin) / seg_size;
    if (cwin > ssthresh) /* congestion avoid. */
        cwin = ssthresh;
    endif
endif
```

In the pseudo-code, seg_size identifies the length of a TCP segment in bits. Note that the reception of n DUPACKs is followed by the retransmission of the missing segment, as in the standard Fast Retransmit implemented by TCP Reno. Also, the window growth after the cwin is reset to ssthresh follows the rules established in the Fast Retransmit algorithm (i.e., cwin grows by one for each further ACK, and is reset to ssthresh after the first ACK acknowledging new data). During the congestion avoidance phase we are probing for extra available bandwidth. Therefore, when n DUPACKs are received, it means that we have hit the network capacity (or that, in the case of wireless links, one or more segments were dropped due to sporadic losses). Thus, the slow start threshold is set equal to the window capable of producing the measured rate BWE when the bottleneck buffer is empty (namely, BWE*RTTmin). The congestion window is set equal to the ssthresh and the congestion avoidance phase is entered again to gently probe for new available bandwidth.. Note that after ssthresh has been set, the congestion window is set equal to the slow start threshold only if cwin > ssthresh. It is possible that the current cwin may be below threshold. This occurs after time-out for example, when the window is dropped to 1 as discussed in the following section. During slow start, cwin still features an exponential increase as in the current implementation of TCP Reno.

In case a packet loss is indicated by timeout expiration, cwin and ssthresh are set as follows:

```
if (coarse timeout expires)
    cwin = 1;
    ssthresh = (BWE * RTTmin) / seg_size;
    if (ssthresh < 2)
        ssthresh = 2;
    endif;
endif
```

The rationale of the algorithm above is that after a timeout, *cwin* and the *ssthresh* are set equal to 1 and BWE, respectively. Thus, the basic Reno behavior is still captured, while a speedy recovery is ensured by setting *ssthresh* to the value of BWE.

B. Bandwidth Estimation

The TCPW sender uses ACKs to estimate BWE. More precisely, the sender uses the following information: (1) the ACK arrival times and, (2) the increment of data delivered to the destination. Let assume that an ACK is received at the source at time t_k, notifying that d_k bytes have been received at the TCP receiver. We can measure the sample bandwidth used by that connection as $b_k=d_k/(t_k-t_{k-1})$, where t_k-1 is the time the previous ACK was received. Letting $\Delta t_k=t_k-t_{k-1}$, then $b_k=d_k/\Delta t_k$.

Since congestion occurs whenever the low-frequency input traffic rate exceeds the link capacity [20], we employ a low-pass filter to average sampled measurements and to obtain the low-frequency components of the available bandwidth. More precisely, we use the following discrete approximation of the low pass filter due to Tustin [21].

Let b_k be the bandwidth sample, and $\hat{b}_k$ the filtered continuous first order low-pass filter using the Tustin estimate of the bandwidth at time t_k. Let α_k be the time-varying exponential filter coefficient at t_k. The TCPW filter is then given by

$$\hat{b}_k = \alpha_k \hat{b}_{k-1} + (1-\alpha_k)\left(\frac{b_k + b_{k-1}}{2}\right), \qquad (1)$$

where

$$\alpha_k = \frac{2\tau - \Delta t_k}{2\tau + \Delta t_k}, \qquad (2)$$

and $1/\tau$ is the filter cut-off frequency.

Notice the coefficients α_k depend on Δt_k to properly reflect the variable inter-arrival times.

A number of considerations must be taken into account while interpreting the information that a returning ACK carries regarding delivery of segments to the destination. Two of these considerations are:

1. An ACK i received by the source implies that a transmitted packet was delivered to the destination. A DUPACK also implies that a transmitted packet was delivered, triggering the transmission by the receiver of the DUPACK. Thus, DUPACKs are considered in estimating bandwidth.
2. TCP ACKS can be "delayed," i.e., receivers wait for a second packet before sending an ACK, until 200 ms elapse in which case an ACK is sent without waiting. Delayed ACKs are also accounted for by our scheme.

These items are included in our implementation of TCPW under Linux.

III. TCPW Fairness and Friendliness

An important goal of any TCP implementation is for every connection to get its "fair share" of the bottleneck. Another important issue is friendliness to existing TCP versions. We will use an informal argument similar to that used for Reno in [30] to show that TCPW achieves the fair share. We use the same approach to evaluate friendliness towards TCP Reno. Consider the case of two connections with the same RTTs. Suppose, for the sake of example that the RTT is X packet transmission times, and the bottleneck has X buffers. One connection, say A, starts first. Its window "cycles" between X and 2X (as per the TCPW algorithm just discussed in Section 2), each cycle terminating when buffer overflow. Later, connection B starts, first in slow start mode, and then in congestion avoidance mode. In congestion avoidance, during each cycle the A and B windows grow approximately at the same rate, i.e. one segment per RTT. Eventually, the bottleneck buffer overflows, terminating the cycle. One can show [internal report] that the window at overflow is:

$W_i = R_i$ (b/C +RTT), for i = A,B

Where, R is the achieved rate (i.e. BWE), b is the bottleneck buffer size, and C is the bottleneck link capacity. This is a general property true for all TCP protocols, and in particular TCPW. After overflow, TCPW reduces the windows to new values W_i' as follows:

$W_i' = R_i$ (RTT) for i = A, B

Thus, the ratios of the windows of connections A and B are preserved after overflow. Yet, the ratio W_B/W_A keeps increasing during congestion avoidance. Consequently, the B window and throughput ratchet up at each cycle. Equilibrium is reached when the two connections have the same windows and the same fair share of the bandwidth. The Fig. 1 graphically illustrates the convergence to the fix point $W_A=W_B$.

This informal proof is validated by actual simulation results. It can be generalized to many simultaneous connections (all with same RTT). It can also be applied to the case when the bottleneck is affected by random errors equally hitting all connections.

The same method can also be used to evaluate reciprocal "friendliness" of TCPW and TCP Reno. If two connections - TCPW and Reno - are sharing the bottleneck, and the buffer size (X in our previous example) is exactly equal to the optimal window size to "fill the pipe", then the two connections spilt the bottleneck fairly. In fact, at equilibrium, each has window = X when buffer overflows. After overflow, the TCPW connection gets window = C*RTT/2 = X/2; TCP Reno simply halves the current window, to X/2. Thus, friendliness is preserved (our simulation results confirm this property). Note that sizing the buffer to match the "pipe size" is a common and intuitively acceptable design choice. If the buffer is much smaller than pipe size, TCPW returns a larger CWIN than Reno, and thus tends to capture the channel. If, on the other hand, the buffer is several times larger than pipe size, Reno tends to prevail over TCPW.

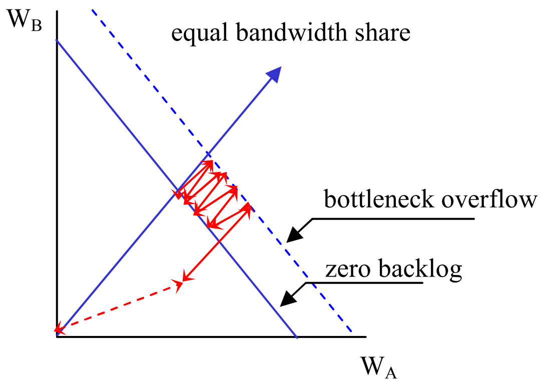

Fig. 1 Convergence toward the fair bandwidth sharing.

IV. Performance Results

All simulations presented in this paper were run using the LBL network simulator, 'ns' ver.2 [18]. Existing modules for TCP Reno and TCP Sack were used. New modules for TCP Westwood were written and they are available at [19]. A TCPW testbed based on Linux was implemented, see Fig. 2. Testbed measurements are also reported.

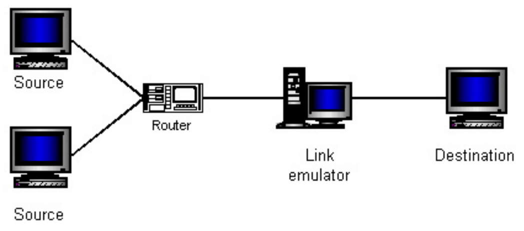

Fig. 2. Experimental testbed layout

We begin with simulation results illustrating the accuracy of TCPW bandwidth estimation scheme. Fig. 3 shows a single TCP connection sharing the bottleneck link with two background UDP ON/OFF sources of varying data rates with NO flow control. TCP packets are 1400 bytes in length including TCP/IP headers. TCP and UDP packets are assigned the same priority. The 5 Mbps bottleneck link has a round trip propagation time of 70 ms. Each UDP connection transmits at a constant bit rate of 1 Mbps while ON.

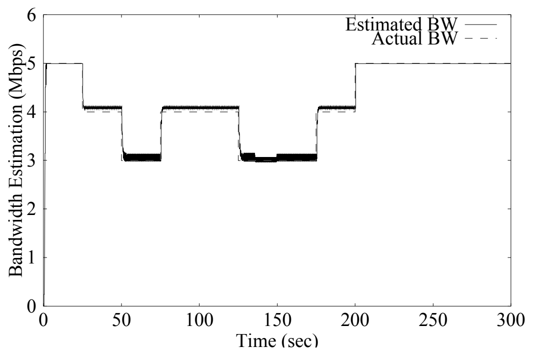

Fig. 3 TCPW with concurrent UDP traffic - bandwidth estimation

Both UDP connections start in the OFF state; after 25s, the first UDP connection is turned ON, joined by the second one at 50s; the second connection follows an OFF-ON-OFF pattern at times 75s, 125s and 175s; at time 200s the first UDP connection is turned off as well.

The results in Fig. 3 confirm the effectiveness of the TCPW feedback control when subjected to "step" and "impulse" interfering traffic: the TCPW bandwidth estimation perfectly tracks the UDP fluctuations, adjusting throughput accordingly.

Next we address fairness (within TCPW flows) and friendliness (towards Reno). To test fairness and friendliness we use our Linux testbed. We first measure the throughput for a total of 5 connections with a variable Reno/TCPW mix in an errorless channel (Fig. 4). Next, we further probe the friendliness of TCPW on the bottleneck link with 1 % packet error rate (Fig. 5). Errors and delays are introduced by a Linux emulator. Note that TCP West shines in presence of line errors (as it will be shown in the next section), so, friendliness in the error situation is even more difficult to establish. RTT in both cases was 100 ms, with bottleneck buffer to match pipe size.

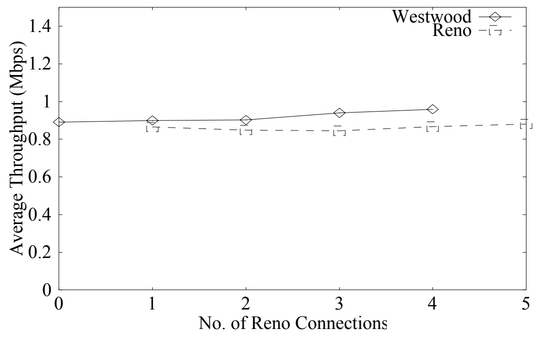

Fig. 4 Measured throughput (error-free link)

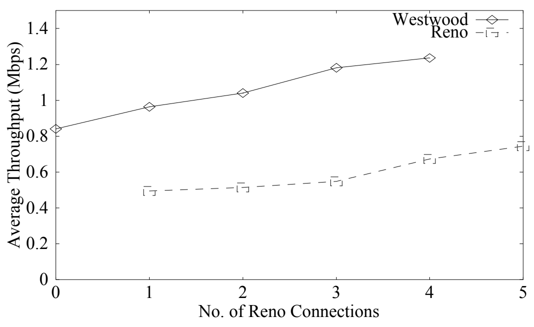

Fig. 5 Measured throughputs (lossy link)

In Fig. 4 and Fig. 5, we show the average TCPW and Reno throughput per connection as a function of number of Reno connections in the mix. For example, at the point marked 3 on the horizontal line, the experiment consists of 3 Reno connections and 2 TCPW connections. The results in Fig. 4 confirm the fairness and friendliness of TCPW already predicted in Sect 3. First, each TCPW connection added to the set in Fig. 4 achieves the same throughput as the previous ones (fairness). Incidentally, Reno is also fair. Second, Reno is minimally affected by TCPW. The average Reno throughput is the same with or without TCPW (friendliness). Fig. 5 illustrates two important points. First, the superior performance of TCPW in a high error rate environment is observed. Namely, 5 TCPW connections get 10 % more throughput than 5 TCP Reno connections. This behavior will be confirmed by theory and experiments in Section 5. Secondly, friendliness is preserved. Even though TCPW has an advantage over Reno in the error-prone environment, Reno connections are not starved. In fact, the improvement shown by TCPW is due more to its ability to deal with wireless losses efficiently than to the "stealing" of bandwidth from Reno.

V. TCPW PERFORMANCE IN PRESENCE OF LINK ERRORS

As mentioned in Section I, we expect TCPW to perform better than Reno and SACK in presence of errors, since TCPW reinstates the previous congestion window (if the system is not congested) rather than reducing it by half. We begin with a set of simulation experiments. Fig. 6 presents simulation results comparing the throughput of Reno and TCPW as a function of error rates. The bottleneck bandwidth is set to 45Mbps, and the two-way propagation time is 70ms. With no errors, the performance of TCPW and Reno is virtually identical. As error rate increases, TCPW outperforms Reno. At 1 % error rates, appropriate for wireless links, the throughput improvement is 615 %. As the error rate increases further, say above 10%, even TCPW collapses, as expected.

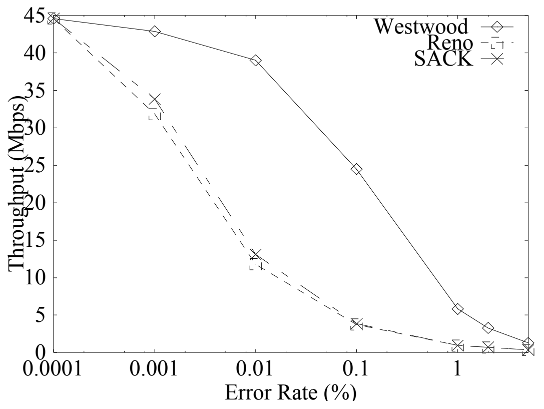

Fig. 6 Impact of error rates (Simulation results)

We also investigated the impact of channel speed and propagation time on throughput in a lossy link situation. As expected, TCPW relative performance with respect to TCP Reno improves with larger speed and larger delay (i.e., larger operational window). The interested reader can find detailed results in [22].

To gain better understanding of the performance with errors, we developed an analytic model of TCPW congestion control mechanism. The model takes into account the filter operation, and includes the following system parameters: the bottleneck link transmission speed, the round trip time, and the link error rate. Here, due to space constraint, we simply outline the method adopted to derive the model, referring to [technical report] for detailed coverage.

Following [23] and [24], we consider a single TCP source with infinite fixed–size packet backlog. The sender releases packets into a limited FIFO buffer that can hold up to B packets (packets arriving to a full buffer are discarded). The packets are then sent over a single bottleneck link with a speed of μ packets per second and a two-way delay of d seconds. Here d is deterministic and reflects propagation time and any other transmission and processing delays along the path excluding the transmission and queuing time at the bottleneck. Let $T=d+1/\mu$ denote the time between the start of a packet transmission and the reception of an ACK for the same packet, excluding queuing delays in the buffer. Thus, the pipe capacity (i.e., packets required to fill the pipe) is given by $C=\mu T$, while the maximum window size a connection can achieve before overflow is $W_{max}=B+C$.

The throughput achieved by the system can be derived by studying the time evolution of the congestion window, $W(t)$. The evolution of this parameter has a cyclic structure, in which $W(t)$ grows until a packet loss is detected. At this point, the missed packet is retransmitted. Further packet transmission is halted until the cumulative ACK for the last window of packets is received. Then, the cycle starts again with the congestion window set to the value derived from the bandwidth estimated at that instant.

During a cycle, packet transmissions are initially bursty. A full window is transmitted each round trip time T, and acknowledgments arrive in *bursts*. The congestion window is increased by one at the end of each burst. If W_0 is the initial congestion window, the first burst contains exactly W_0 packets, the second W_0+1 and so on. ACKs in the same burst arrive $1/\mu$ seconds apart while consecutive bursts arrive T second apart, until the pipe capacity, C, is reached. After this point, ACKs arrive continuously at rate μ.

The bandwidth estimate is adjusted after each ACK, following the algorithm described in the previous sections, while the congestion window is increased by one at the end of each burst.

The window W(m) at the end of a generic cycle m is uniquely identified by the initial congestion window, $W_0^{(m)}$, and the sequence number $N_{drop}^{(m)}$ of the missing packet that causes the cycle to terminate. The packet loss may arise from a link error or a buffer overflow. Assuming that packet errors occur independently with probability ν, the probability that the cycle ends at packet n is given by

$$p_{i,n} = \Pr\left[N_{drop} = n \middle| W_0 = i\right] = \begin{cases} \nu(1-\nu)^{n-1}, & n \le n_{\max}(i) \\ (1-\nu)^{n-1}, & n > n_{\max}(i) \end{cases} \qquad (3)$$

where $n_{max}(W_0)$ is the maximum number of packets that may be sent and acknowledged in a cycle, given by

$$n_{\max}(W_0) = 1/2\left(\mathrm{W}_{\max} - \mathrm{W}_0 + 1\right)\left(\mathrm{W}_{\max} + \mathrm{W}_0\right) + \mathrm{W}_{\max}. \qquad (4)$$

The average throughput achievable in the cycle is, then, given by

$$\eta(W_0) = \sum_{n=1}^{n_{\max}(W_0)+1} \frac{n-1}{t(W_0,n)} p_{W_0,n} \qquad (5)$$

where $t(W_0,n)$ is the cycle duration, and can be expressed in terms of W_0 and n [technical report].

At the end of the cycle, the congestion window is set to the current bandwidth estimate. The bandwidth estimate at the end of a cycle is uniquely determined by the start cycle estimate and by the ACK-arrival process during the cycle. Thus, he evolution of $W_0^{(m)}$ can be described as an embedded Markov process . The transition probability $P_{i,j}$ from $W_0^{(m-1)}=i$ to $W_0^{(m)}=j$ can be computed as

$$P_{i,j} = \sum_{n \in I_j(i)} p_{i,n} \; ; \quad i=2,\ldots,C \qquad (6)$$

where $p_{i,n}$ is given by equation (3), while $I_j(i)$ is the set of packet sequence numbers n for which, starting with a window = i at the beginning of the cycle, the estimated bandwidth after n packet transmissions corresponds to a window size = j.

The average throughput achieved by the system can be expressed as

$$\bar{\eta} = \sum_{W_0=2}^{C} \pi_{W_0} \eta(W_0), \qquad (7)$$

where π_W denotes the asymptotic probability of the window size W at the start of a cycle and $\eta(W_0)$ is given by (5).

We ran several *ns* simulation experiments to validate the model. Fig. 7 shows a comparison of analysis and simulation results for both TCP Reno and TCPW for different packet error probabilities.

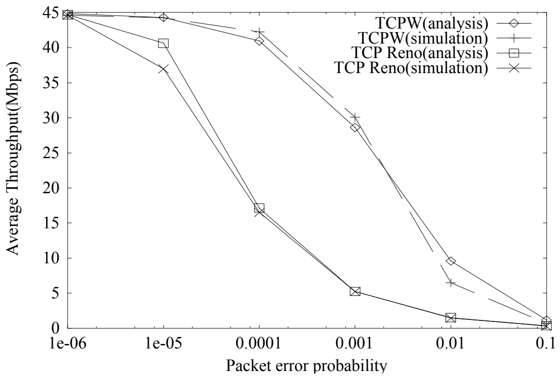

Fig. 7 Throughput comparison (bottleneck capacity: 45Mbps)

VI. Internet Measurements

Using our Linux testbed, we have tested TCPW also in the global Internet environment (Fig. 8). The sources are at UCLA, while the destinations are chosen in three different continents (Europe, South America, and Asia). The destination Hosts are, of course, unaware whether the source Host runs TCPW or Reno.

Tests were scheduled during normal working hours at the destination sites. Experiments included either single or multiple file transfers. Throughput results were obtained by averaging repeated single file transfers.. A rather large file size was used (10 Mbytes) to capture only steady state behavior. We used a standard FTP client (ncftp-3.0.2) as testing software with additional code for obtaining detailed logging at 1-second intervals. We measured application throughput in terms of user data/second as reported by ncftp. The average throughput achieved by Reno and TCPW on the three intercontinental connections is shown in Tab.1. Tests were repeated about 200 times throughout the day. The results show that TCPW performs marginally better that Reno on the Italy and Taiwan connections. It performs significantly better on the Brazil connection.

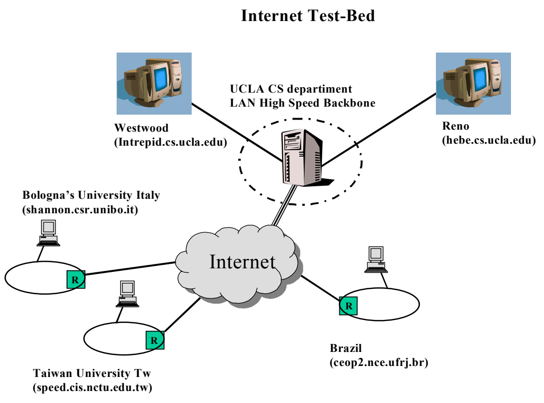

Fig. 8 Internet measurement scenario

TABLE 1

INTERNET THROUGHPUT MEASUREMENTS

Destination RTT	Italy 170 ms		Taiwan 250 ms		Brazil 450 ms	
Protocol	TCPW	Reno	TCPW	Reno	TCPW	Reno
Throughput (KB/s)	78.66	73.93	167.38	152	22.16	15.4

We further investigated this discrepancy using the `traceroute` Linux tool. We found that Italy and Taiwan are connected using standard wired technology with minimal link errors. As expected, TCPW does not introduce much improvement over Reno. On the other hand, the Brazil path has a "lossy" satellite link provided by Teleglobe. The satellite link is only in the direction to destination, while the opposite direction uses terrestrial links. The lossy satellite link accounts for the TCPW improved performance.

VII. Conclusions

We have introduced a new TCP scheme (TCP Westwood) that requires modifications only in the TCP source stack and is thus compatible with TCP Reno and Tahoe destinations. TCP Westwood (or TCPW for short) differs from Reno in that it adjusts the *cwin* (congestion window) after a loss detection by setting it to the *measured rate* currently experienced by the connection, rather than using the conventional multiplicative decrease scheme (i.e., divide the current window by half). We have shown with qualitative arguments and with experimental results that the new control scheme converges to "fair share." at steady state under uniform path conditions Moreover, TCPW has several pleasing properties with respect to

Reno. Most important, it can handle losses caused by link errors or wireless channel conditions more efficiently than TCP Reno. One general concern with new TCP versions is friendliness towards current implementations. TCPW exhibits some "aggressiveness" due to its unique window adjustment. However, if there is adequate buffering at the bottleneck, TCPW and Reno share the channel fairly. Moreover, if TCPW and Reno coexist on a bottleneck with error induced losses (e.g., wireless link), TCPW outperforms Reno mainly because it can make better use of the channel, while "stealing" only a modest fraction of throughput from Reno. TCPW has been implemented in LINUX and has been tested extensively in a local testbed at UCLA as well as on Internet cross-continental links. The performance measured in the testbeds and in the Internet confirms the simulation results. In particular, the throughput performance over a lossy route to Brazil (satellite link) exhibited a 100 % improvement over Reno.

REFERENCES

[1] V. C. Cerf and R. E. Kahn, "A Protocol for packet Network Interconnections," IEEE Transactions on Communications, vol. COM-22, no. 5, pp. 637-648, May 1974.

[2] V. Jacobson, "Congestion Avoidance and Control," ACM Computer Communications Review, 18(4) : 314 - 329, August 1988.

[3] V. Jacobson, "Berkeley TCP evolution from 4.3-Tahoe to 4.3 Reno," Proceedings of the 18th Internet Engineering Task Force, University of British Colombia, Vancouver, BC, Sept. 1990.

[4] T. Bonald, "Comparison of TCP Reno and TCP Vegas: Efficiency and Fairness," In Proceedings of PERFORMANCE'99, Istanbul, Turkey, October 1999.

[5] U. Hengartner, J. Bolliger, and T. Gross, "TCP Vegas Revisited," In Proceedings of IEEE INFOCOM'2000, Tel Aviv, Israel, March 2000.

[6] M. Gerla, R. Lo Cigno, S. Mascolo, and W. Weng, "Generalized Window Advertising for TCP Congestion Control," CSD-TR 990012, UCLA, CA, USA, February 1999.

[7] L. Kalampoukas, A. Varma, and K. K. Ramakrishnan, "Explicit Window Adaptation: A Method to Enhance TCP Performance," In Proceedings of IEEE INFOCOM'98, San Francisco, Ca, USA, March/April 1998.

[8] T. Goff, J. Moronski, D. S. Phatak, and V. Gupta, "Freeze-TCP: a True End-to-end TCP Enhancement Mechanism for Mobile Environments," In Proceedings of IEEE INFOCOM'2000, Tel Aviv, Israel, March 2000.

[9] D. Clark, "The design philosophy of the DARPA Internet protocols," In Proceedings of Sigcomm'88 in ACM Computer Communication Review, vol. 18, no. 4, pp. 106 - 114, 1988.

[10] C. Casetti, M. Gerla, S. Lee, S. Mascolo, and M. Sanadidi, "TCP with Faster Recovery," MILCOM 2000, Los Angeles, CA, October 2000.

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

[12] M. Allman and Vern Paxson, "On Estimating End-to-End Network Path Properties", Sigcomm 1999.

[13] Kevin Lai and Mary Baker, "Measuring Link Bandiwdths Using a Deterministic Model of Packet Delay", Sigcomm 2000

[14] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz, "A Comparison of Mechanisms for Improving TCP Performance over Wireless Links," IEEE/ACM Transactions on Networking, December 1997.

[15] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz, "Improving TCP/IP Performance Over Wireless Networks," MOBICOM'95, Berkeley, CA, USA, November 1995

[16] Hari Balakrishnan and Randy H. Katz, "Explicit Loss Notification and Wireless Web Performance," In Proceedings of IEEE GLOBECOM'98 Internet Mini-Conference, Sydney, Australia, November 1998.

[17] S. Q. Li and C. Hwang, "Link Capacity Allocation and Network Control by Filtered Input Rate in High speed Networks," IEEE/ACM Transactions on Networking, vol. 3, no. 1, pp. 10 - 25, Feb. 1995.

[18] K. J., B. Wittenmark, "Computer controlled systems," Prentice Hall, Englewood Cliffs, N. J., 1997.19 ns-2 network simulator (ver 2). LBL, URL: http://www-mash.cs.berkeley.edu/ns.

[19] J. Kurose and K. Ross, "Computer Networking: A Top-Down Approach Featuring the Internet", Addison Wesley, 2000.

[20] ns-2 network simulator (ver.2). LBL, URL: http://www.mash.cs.berkley.edu/ns.

[21] TCP Westwood modules for ns-2. URL: http://www1.tlc.polito.it/casetti/tcp-westwood.

[22] A. Zanella, G. Procissi, M. Gerla, M.Y.Sanadidi, "TCP Westwood: analytic model and performance evaluation", Technical Report,URL:http://www.cs.ucla.edu/csd/pubs/pubs.html

[23] L. Zhang, S. shenker, and D. D. Clark, "Observations on the Dynamics of A Congestion Control Algorithm: the Effects of Two-way traffic", Proc. ACM SIGCOMM '91, pp.133-147.

[24] A. A. Abouzeid, S. Roy, and M. Azizoglu, "Stochastic Modeling of TCP over Lossy link," INFOCOM 2000, Tel Aviv, Israel, March 2000.