TCP Westwood: congestion control with faster recovery

UCLA CSD TR #200017
S. Mascolo, C. Casetti, M. Gerla, S. S. Lee, M. Sanadidi
Computer Science Department - UCLA
Los Angeles, CA, 90024

Abstract

In this paper we propose a new version of the TCP protocol, called TCP Westwood, which enhances the performance of TCP window congestion control by using as feedback the end-to-end measurement of the bandwidth available along a TCP connection. The available bandwidth is estimated at the TCP source by measuring and low-pass filtering the returning rate of acknowledgments. The estimated bandwidth is then used to properly set the congestion window and the slow start threshold after a congestion episode, that is after a timeout or 3 duplicate acknowledgments. The rationale of this strategy is simple: TCP Westwood sets a slow start threshold and a congestion window which are consistent with the network capacity measured at the time congestion is experienced. In particular, TCP Westwood introduces a mechanism of faster recovery to avoid overly conservative reduction of the congestion window after a congestion episode by taking into account the end-to-end estimation of available bandwidth. The advantage of the proposed mechanism is that the TCP sender recovers faster after losses especially over connections with large round trip times, or running over wireless links where sporadic losses are due to unreliable links rather than congestion. The proposed modifications follow the end-to-end design principle of TCP. They require only slight modifications at the sender side and are backward-compatible. Simulation results show a considerable throughput increment in comparison with TCP Reno and TCP SACK over wired networks and even more over wireless networks.

I. INTRODUCTION

The Transmission Control Protocol (TCP) is designed to offer an end-to-end connection-oriented packet switching service using IP over heterogeneous networks. It was originally designed to provide reliable data delivery over conventional (wired) networks for a limited range of transmission rates and propagation delays. One of TCP strengths lies in its congestion control mechanism proposed in the cornerstone work by Van Jacobson [1]. Nowadays, data transfers over communication paths with ever-larger bandwidth/delay products, quality of service (QoS) requirements for interactive traffic and communication over wireless links, are shifting the domain for which TCP was originally engineered. As a consequence, active research is in progress to extend the domain of TCP operability [2], [3], [4], [5], [6].

The congestion control algorithm used in the TCP/IP protocol is a sliding window mechanism that uses packet loss to detect congestion. In particular, the end-systems probe the network state by gradually increasing the window of packets that are outstanding in the network until the network becomes congested and drops packets. Initially, the increase is exponential during the Slow Start phase. This phase is intended to quickly grab the available bandwidth. When the window size reaches a slow start threshold (*ssthresh*), the increase becomes linear, thus allowing for a gentler probing of the available capacity. Clearly, it is desirable to set the threshold to a value that approximates the connection's "fair share". The optimal value for the slow start threshold is the one that corresponds to the segments in flight in a pipe when TCP rate equal to the available bandwidth [8], i.e. when its transmission window is equal to the available bandwidth–delay product.

When a loss is detected either through duplicate acknowledgements, or through a coarse timeout expiration, the connection backs off by shrinking its congestion window. If the loss is indicated by duplicate ACKs, TCP Reno attempts to perform a "fast recovery" by retransmitting the lost segments and halving the congestion window. If the loss is followed by a coarse timeout expiration, the congestion window is reset to 1. In either case, after the congestion window is reset, the connection needs several round-trip times

before the window-based probing is restored to near-capacity. This problem is exacerbated when random or sporadic losses occur. Random losses are here defined as losses not caused by congestion at the bottleneck link. They are common in the presence of wireless channels. In this case, a burst of lost segments is erroneously interpreted by a TCP source as an indication of congestion, and dealt with by shrinking the sender's window. Such action, clearly, does not alleviate the random loss condition and it merely results in reduced throughput. The larger the bandwidth-delay product, the larger the degradation caused by such action.

A similar situation occurs in presence of bursty sources that may be responsible for small, sporadic losses due to a flurry of UDP packets shortly congesting intermediate routers. Although a smaller transmission window can help lowering the congestion in the short run, it will affect the source's ability to regain speed in the long run. Random or sporadic losses (or a combination of the two) cannot be efficiently handled by conventional TCP algorithms that use packet drop (rather than bandwidth availability) information to set their congestion window.

This paper builds on the guidelines set forward in [7]. It proposes a simple scheme for the TCP source to estimate the available bandwidth and use the bandwidth estimation to recover faster, thus achieving higher throughput. The proposed scheme exploits two basic concepts: the end-to-end estimation of the available bandwidth, and the way such estimate is used to set the slow start threshold and the congestion window. It is worth underscoring that our scheme is in tune with the fundamental TCP design principle that the feedback is merely end-to-end and does not rely upon explicit information from intermediate nodes at the network level.

The paper is organized as follows: Section II details the bandwidth measurement process included in TCP Westwood; in Section III, we discuss the guidelines of the new algorithm and present implementation details in Section IV; experimental test results and comparisons with other versions of the TCP protocol are shown in Section V; Section VI concludes the paper.

II. End-to-end bandwidth measurement

A basic assumption of TCP design is that the network is a "black box". As a consequence, a TCP source cannot receive any explicit congestion feedback from the network and has to rely only on implicit feedback such as timeouts, duplicate acknowledgments, round trip measurements. It is thus said that TCP must perform an "end-to-end" control. In this work, we introduce a new implicit feedback to be used for congestion avoidance. We propose that a source perform an end-to-end estimate of the bandwidth available along a TCP connection by *measuring the rate of returning acknowledgments*. For such an estimate to be meaningful, the source must be able to infer the amount of data delivered to the receiver over time. The TCP protocol provides for the receiver to notify the sender of the reception of a segment by means of an acknowledgement (ACK), carrying an indication as to what segment was received. When an ACK is received by the source, it conveys the information that an amount of data corresponding to a specific transmitted packet was delivered to the destination. If the transmission process is not affected by losses, simply averaging the delivered data count over time yields a fair estimation of the bandwidth currently used by the source.

When duplicate ACKs (DUPACKs), indicating an out-of-sequence reception, reach the source, they should also count toward the bandwidth estimate, and a new estimate should be computed right after their reception. However, the source is in no position to tell for sure which segment triggered the DUPACK transmission, and it is thus unable to update the data count by the size of that segment. An average of the segment size sent thus far in the ongoing connection should therefore be used, allowing for corrections when the next cumulative ACK is received. For the sake of simplicity, though, in this paper we will assume all TCP segments as having the same size. Following this assumption, we will further assume that sequence numbers are incremented by one per segment sent, although the actual TCP implementation keeps track of the number of bytes instead: the two notations are interchangeable if segments have all the same size.

It is important to notice that, immediately after a congestion episode, followed either by a timeout or n duplicate ACKs, the bandwidth used by the connection is exactly equal to the maximum bandwidth available to that connection. This is confirmed by the fact that packets have been dropped, a clear indication that

buffers are at (or near) saturation. Before a congestion episode, the used bandwidth is less or equal to the available bandwidth because the TCP source is still probing the network capacity. It is important to employ a low-pass filter to obtain the low-frequency components of the available bandwidth. In fact it is known [8] that congestion occurs whenever the low-frequency input traffic rate exceeds the link capacity. Therefore it is useful to track only low-frequency components of the available bandwidth. In our scheme, the bandwidth estimation is performed using a low-pass filter, as described by the following pseudocode:

```
if (ACK is received)
  sample_BWE[k] = (acked*pkt_size*8)/
                  (now - lastacktime);
  BWE[k] = (19/21)*BWE[k-1] + (1/21)*
           (sample_BWE[k]+ sample_BWE[k-1]);
endif
```

where `acked` indicates the number of segments acknowledged by the latest ACK, `pkt_size` indicates the segment size in bytes, `now` indicates the current time, `lastacktime` the time the previous ACK was received, `k` and `(k-1)` indicate the current and the previous value of the variable, and `BWE` is the low-pass filtered measurement of the available bandwidth. The filter is obtained by discretizing a first-order low-pass filter using the trapezoidal rule [1] and by assuming a time constant to sampling-time ratio [2] equal to 10.

The estimated bandwidth is eventually translated into the appropriate windows size as $cwin = BWE * RTT_{min}$, where RTT_{min} is the smallest round-trip time routinely computed by the TCP source (and used to set the coarse timeout).

As a final remark, in this paper we focus on estimating the available bandwidth at the TCP source in order to minimize and localize modifications of TCP at the sender side. It is clear that the bandwidth available along a TCP connection can be evaluated at the receiver side using the same filtering procedure. Then, this feedback could be delivered back to the source via ACKs by setting the $AdvertisedWindow$ field equal to $min(AdvertisedWindow, RTT_{min} * BWE))$. On the one hand, this choice has the major advantage of robust bandwidth estimation with respect to losses of ACKs along the returning path. Indeed, losses of ACKs, i.e., along asymmetric TCP connections, could negatively affect the bandwidth estimation at the source. On the other hand, it would require modifications of the TCP receiver, whereas our choice of placing the bandwidth estimation at the sender favors a sender-side-only implementation of the new protocol.

A. On the effects of delayed and cumulative ACKs on BWE

As previously stated, DUPACKs should count toward the bandwidth estimation, since their arrival indicates a successfully received segment, albeit in the wrong order. As a consequence, a cumulative ACK should only count as one segment's worth of data since duplicate ACKs ought to have already been taken into account. However, the matter is further complicated by the issue of *delayed ACKs*. The standard TCP implementation provides for an ACK being sent back once every other segment received, or if a 200-ms timeout expires after the reception of a single segment [9].

The combination of delayed and cumulative ACKs can potentially disrupt the bandwidth estimation process, as pointed out by the following example. Suppose a connection has successfully delivered every segment up to no. 99, and no. 100 through 109 are lost due to sudden congestion. If the transmitter window at that point is sufficiently large to send 20 more segments, and the congestion has been relieved, segments ranging from no. 110 to 129 will be successfully delivered and will elicit a flurry of 20 duplicate acknowledgements. According to our bandwidth estimation algorithm, each DUPACK received should trigger a BWE update. On receiving three DUPACKs, the TCP will enter the 'Faster Retransmit' phase, and will resend packets from no. 100 onwards. Assuming no losses occur, the receiver will counter these resent segments by issuing 5 delayed ACKs (one for each pair of segments from 100 through 109) and 1 cumulative

[1] The trapezoidal rule is also known as bilinear transformation or Tustin rule

[2] In a further research, we are going to explore filters that take into account the fact that the sampling time, i.e. the interarrival time of ACKs, is not constant

ACK (acknowledging segments from 110 through 129, which it originally received and stored). Clearly, if the above pseudocode is applied verbatim, the bandwidth estimation would surge upward as the source receives back-to-back ACKs, one of which single-handedly acknowledges 20 segments. The value of `acked` in the pseudocode must therefore be carefully chosen. This example stresses two important aspects of the bandwidth estimation process:

- the source must keep track of the number of DUPACKs it has received before new data is acknowledged;
- the source should be able to detect delayed ACKs and act accordingly.

The approach we have chosen to take care of these two issues is detailed by the `AckedCount` procedure, detailed below, showing the set of actions to be undertaken upon the reception of an ACK, for a correct determination of `acked`. The key variable is `accounted_for`, which keeps track of the received DUPACKs. When an ACK is received, the number of segments it acknowledges is first determined (`cumul_ack`). If `cumul_ack` is equal to 0, then the received ACK is clearly a DUPACK and counts as 1 segment towards the BWE; the DUPACK count is also updated. If `cumul_ack` is larger than 1, the received ACK is either a delayed ACK or a cumulative ACK following a retransmission event; in that case, the number of ACKed segments is to be checked against the segments already accounted for (`accounted_for`). If the received ACK acknowledges fewer or the same number of segments than expected, it means that the "missing" segments were already accounted for when DUPACKs were received, and they should not be counted twice. If the received ACK acknowledges more segments than expected, it means that although part of them were already accounted for by way of DUPACKs, the rest are cumulatively acknowledged by the current ACK; therefore, the current ACK should only count as the cumulatively acknowledged segments. It should be noted that the last condition correctly estimates the delayed ACKs (`cumul_ack` = 2 and `accounted_for` = 0).

```
PROCEDURE AckedCount

cumul_ack = current_ack_seqno - last_ack_seqno;

if (cumul_ack = 0)
  accounted_for=accounted_for+1;
  cumul_ack=1;
endif

if (cumul_ack > 1)
  if (accounted_for >= cumul_ack)
    accounted_for=accounted_for-cumul_ack;
    cumul_ack=1;
  else if (accounted_for < cumul_ack)
         cumul_ack=cumul_ack-accounted_for;
         accounted_for=0;
       endif
  endif
endif

last_ack_seqno=current_ack_seqno;
acked=cumul_ack;

return(acked);

END PROCEDURE
```

III. TCP WESTWOOD: ALGORITHM GUIDELINES

In this section we describe how the bandwidth estimation can be used by the congestion control algorithm executed at the sender side of a TCP connection in order to accomplish a faster recovery after a congestion event. First, we outline the algorithm in the most general form. Then, we describe the specific form we

have implemented. As will be explained, the congestion window dynamics during slow start and congestion avoidance are unchanged, that is they increase exponentially and linearly, respectively, as in current TCP Reno.

The general idea is to use the estimated bandwidth BWE to set the congestion window (*cwin*) and the slow start threshold (*ssthresh*) after a congestion episode. Recall that the basic role played by *cwin* and *ssthresh* in TCP congestion control is that *cwin* is increased and decreased to track the available bandwidth–delay product that should be represented by *ssthresh*.

A major, additional advantage of using BWE as an implicit feedback to set *cwin* and *ssthresh* comes from the fact that network routers can easily enforce fair queueing on FIFO queues by implementing simple queueing policies such as RED, WRED or FRED. In the past, several researchers [10], [11] have proposed droppers to allocate available bandwidth to different flows according to specific queueing policies. While TCP Westwood *does not* rely on information from intermediate nodes, it can nonetheless exploit these queueing strategies, if present, thanks to the resulting accurate flow-by-flow bandwidth allocation. Overall, TCP Westwood performance improves if some form of fair sharing is implemented in the network, although this aspect will be discussed in a different work.

We start by describing the general algorithm behavior after n duplicate ACKs and after a coarse timeout expiration.

A. Algorithm after n duplicate ACKS

The pseudocode of the algorithm is the following:

```
if (n DUPACKs are received)
  if (cwin>ssthresh) /* congestion avoid. */
     ssthresh = f1(BWE*RTTmin);
     cwin = ssthresh;
  endif
  if (cwin<ssthresh) /*slow start */
    ssthresh= f2(BWE*RTTmin)
    if (cwin > ssthresh)
      cwin = ssthresh
    endif
  endif
endif
```

The rationale of the algorithm is simple. 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 of more segments were dropped due to sporadic losses). Thus, the slow start threshold is set equal to the available pipe size, which is $BWE * RTTmin$, the congestion window is set equal to the *ssthresh* and the congestion avoidance phase is entered again to gently probe for new available bandwidth. Function `f1` introduces one degree of freedom that can be used to tune the algorithm. In this paper we have chosen an identity function for `f1`, i.e. $\texttt{f1}(\cdot) = (\cdot)$. During the slow start phase we are still probing for the available bandwidth. Therefore the BWE we obtain after n duplicate ACKs is used to set the slow start threshold. After *ssthresh* has been set, the congestion window is set equal to the slow start threshold only if `cwin>ssthresh`. In other words, during slow start, *cwin* still features an exponential increase as in the current implementation of TCP Reno. Function `f2` introduces one more degree of freedom that we can use to tune the algorithm.

B. Algorithm after coarse timeout expiration

The pseudocode of the algorithm is

```
if (coarse timeout expires)
  if (cwin>ssthresh) /* congestion avoid. */
```

```
    ssthresh = f3(BWE*RTTmin);
    if (ssthresh < 2)
      ssthresh = 2;
      cwin = 1;
    else
      cwin = f4(BWE*RTTmin);
    endif
  endif
  if (cwin<ssthresh) /* slow start */
    ssthresh = f5(BWE*RTTmin)
    if (ssthresh < 2) ssthresh = 2;
       cwin = 1;
    else
       cwin = f6(BWE*RTTmin)
    endif
  endif
endif
```

The rationale of the algorithm is again simple. After a timeout the *cwin* and the *ssthresh* are set according to one of the functions `fi`, `i`=3,6 depending on the phase the algorithm is in when a timeout is experienced. Notice that using the general functions `fi`, `i`=1,6 provides six degree of freedom to tune the algorithm. In the next Sections, we will investigate and simulate a sample selection of these functions, and provide default values.

IV. TCP WESTWOOD: ALGORITHM IMPLEMENTATION

In this section, we illustrate an implementation obtained choosing simple `fi` functions.

A. Algorithm after 3 duplicate ACKS

The pseudocode of the algorithm is as follows:

```
if (3 DUPACKs are received)

  if (cwin<ssthresh) /* slow start */
    a = a + 0.25;
    if (a > 4)
      a = 4;
    endif
  endif

  if (cwin>ssthresh) /* congestion avoid. */
    a = 1;
  endif

  ssthresh = (BWE*RTTmin)/(pkt_size*8*a);

  /* reset cwin to ssthresh, if larger */

  if (cwin>ssthresh)
    cwin = ssthresh;
  endif

endif
```

By inspecting the code, it can be seen that during congestion avoidance, `f1` is simply chosen as an identity function, i.e. $\texttt{f1}(x) = x$. Conversely, during slow start, `f2` is chosen as $\texttt{f2}(x) = x/a$. Notice that a increases from 1 to 4, in steps of 0.25, every time 3 DUPACKs are received in slow start, while a is set to 1 when 3 DUPACK are received in congestion avoidance. At connection setup, a is initialized as 1.

The purpose of the *threshold reduction factor* a is to dampen a possible overestimation of the available bandwidth, an occurrence which is rather common during prolonged periods of congestion. Indeed, the more frequently a triple DUPACK is received during slow start (an indication that `ssthresh` was set too high), the bigger the reduction factor becomes. Following the same line of reasoning, a is restored to 1 if congestion is detected in congestion avoidance: clearly, *ssthresh* was set correctly and there is no need to reduce the impact of BWE.

B. Algorithm after coarse timeout expiration

Following a coarse timeout expiration, a set of actions similar to the triple DUPACK case is triggered:

```
if (coarse timeout expires)

  if (cwin<ssthresh) /* slow start */
    a = a + 1;
    if (a > 4)
      a = 4;
    endif
  endif

  if (cwin>ssthresh) /* congestion avoid. */
    a = 1;
  endif

  ssthresh = (BWE*RTTmin)/(pkt_size*8*a);

  if (ssthresh > 2)
    ssthresh = 2;
    cwin = 1;
  endif

endif
```

In this case, function `f3`, which is used to set *ssthresh* when a timeout occurs during congestion avoidance, is chosen as $\texttt{f3}(x) = x$. Function `f4` is chosen equal to one, i.e. $\texttt{f4}(x) = 1$. Function `f5`, which is used to set *ssthresh* when a timeout happens during slow start phase, is chosen as $\texttt{f5}(x) = x/a$ where a increases from 1 to 4, in steps of 1 (as opposed to 0.25 in the triple DUPACK case) every time a timeout happens, and a is set to 1 when a timeout occurs in congestion avoidance. Also `f6` is set equal to 1.

Notice the congestion window is reset to 1 after a timeout, as is done by TCP Reno. This choice is conservative because it does not take full advantage of the BWE information to avoid the shrinking of the congestion window to 1 in the presence of sporadic losses due to wireless links interference rather than to congestion. There is a reason for this choice: fairness. We think that with drop-tail FIFO queuing it is important to preserve the cyclic behavior of TCP, allowing traffic load fluctuations on each TCP connection. Indeed, this behavior ensures the fair sharing of bandwidth resources between different connections bottlenecked at the same FIFO queue without affecting the stability of the algorithm. Different settings may be proposed for *cwin* and *ssthresh* after a timeout when network nodes implement RED or WRED, but this issue is left for further research. Also, new mechanisms can be devised to switch between congestion avoidance and slow start, i.e., a strategy to increase *ssthresh*, by using a filter designed for bandwidth estimation during underutilization of the network.

V. SIMULATION ANALYSIS OF TCP WESTWOOD

In this section, we present several simulation experiments that aim both at exemplifying the behavior of TCP Westwood and at comparing it with other well-established versions of TCP, such as Reno and Sack [12]. All simulations results presented in this paper were run using the LBL network simulator, 'ns' ver.2. [13]. New simulation modules for TCP Westwood were written and they are available at [14], while

existing modules for simulations involving TCP Reno and TCP Sack were used. All simulated TCP receivers implement a delayed-ACKs policy, consistent with correct TCP implementations. Each scenario, involving different capacity, RTT or number of concurrent connections, is designed as a single-bottleneck network. The intermediate node buffer is always supposed to hold a number of packets equal to the bandwidth-delay product for that scenario.

A. The transient behavior of TCP Westwood

The first set of simulations aims at gaining a better understanding of the transient behavior of TCP Westwood in changing network conditions. The scenario features a 45 Mb/s bottleneck with a one-way end-to-end delay of 100ms. One TCP connection shares a FIFO bottleneck with two ON/OFF UDP connections, with the same priority as TCP traffic. Each UDP connection transmits at a constant bit rate of 9 Mb/s while ON. Both UDP connections start in the OFF state; after 50 s, the first UDP connection is turned ON, joined by the second one at 100 s; the second connection follows and OFF-ON-OFF pattern at times 150 s, 250 s and 350 s; at time 400 s the first UDP connection is turned off too. They remain silent until the end of the simulation, except for a brief 2-second ON burst around time 460 s. The TCP connection sends data throughout the simulation.

Although hardly realistic, this network setup can provide us with an insight as to the step and pulse responses of the control algorithm implemented by TCP Westwood. These responses will be compared to those of TCP Reno.

Figure 1 shows the behavior of the bandwidth estimation process. A dotted line identifies the theoretical available bandwidth (left over by the UDP connections). It can be seen that the bandwidth estimation keeps track rather easily of downward transitions (e.g., at times 50 s, 100 s and 250 s, as well as at 460 s), although some “overestimate” spikes can be spotted. The upward transitions show a slower convergence to the right estimate. The reason is that when the available bandwidth increases, TCP takes time to reach an input rate equal to the available bandwidth. Therefore the ACK rate is less or equal than the bandwidth estimation. For this reason we have already remarked that BWE is used only after congestion episodes, i.e., downward transitions. The interference between the estimation process and the transmission window mechanism, which forces the TCP source to stop when the window is closed, explains the fluctuations.

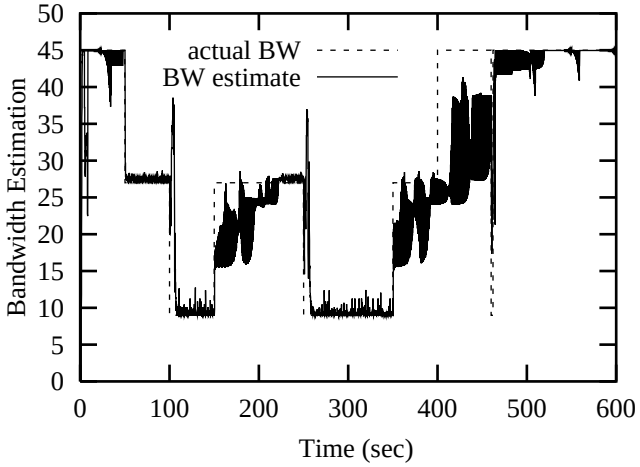

Fig. 1. TCP Westwood with concurrent UDP traffic - bandwidth estimation

Figures 2 and 3 allow a comparison between the congestion windows dynamics of Westwood and Reno: while the latter’s *ssthresh* is forced back further and further after each burst of losses, TCP Westwood estimates the bandwidth to reset the *ssthresh* compatibly with the available bandwidth.

The next three plots depict the transient behavior of Westwood over lossy links. The scenario being the same as described above (including 100 ms end-to-end delay), the bottleneck link is now subject to random

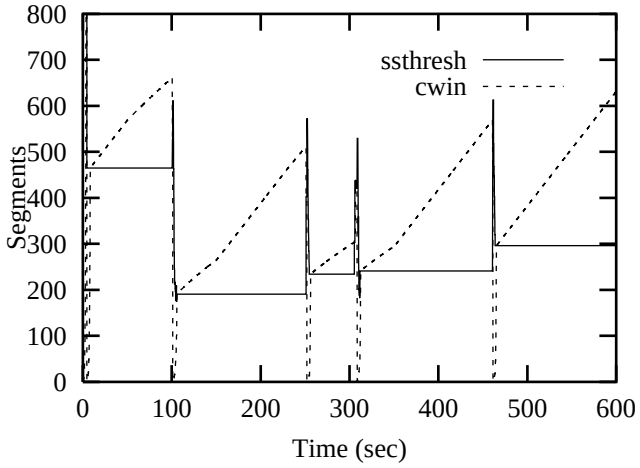

Fig. 2. TCP Westwood with concurrent UDP traffic - congestion window and *ssthresh*

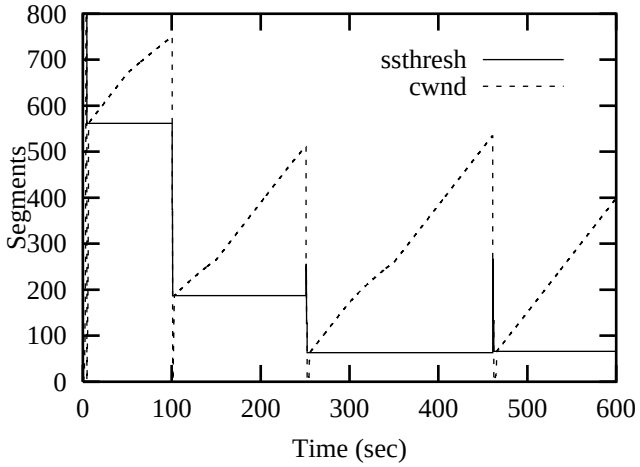

Fig. 3. TCP Reno with concurrent UDP traffic - congestion window and *ssthresh*

losses, mimicking the behavior of a wireless link. Loss events are characterized by dropped segments between the source and the intermediate node. In our simulations, the probability of a 'loss event' is 10^{-4}.

Again, the comparison of *cwin* and *ssthresh* for TCP Westwood and Reno connections (Figures 4 and 5) provides the most telling evidence of Westwood's "faster recovery" features. Indeed, while repeated losses floor Reno's attempts at regaining transmission speed, Westwood exploits its bandwidth estimation capabilities to resume transmission almost unhindered.

Figure 6 summarizes the test results shown so far by comparing throughputs of Westwood and Reno connections over lossy and non-lossy links.

B. Losses caused by bursty traffic

Bursty UDP traffic is often the cause for TCP losses because of its uncontrolled nature. In our tests, we compared the behavior of TCP connections in the presence of an ON/OFF UDP source. ON and OFF periods are 60 seconds each, and the UDP source consumes 90% of the bottleneck during ON periods. The UDP bursty source could represent a high-bandwidth, real-time scientific application (e.g., virtual reality, visualization of scientific experiments, etc.). All tests lasted 600 simulated seconds. Results are reported on graphs showing the average goodput (i.e., the delivered data rate) attained by connections using either TCP Westwood, Reno or Sack.

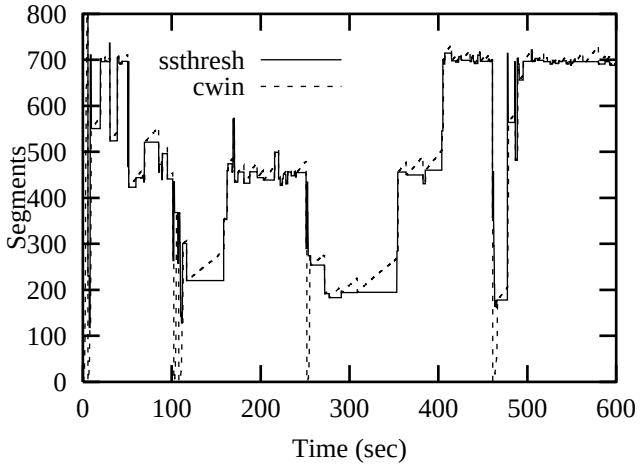

Fig. 4. TCP Westwood with concurrent UDP traffic on lossy link - congestion window and *ssthresh*

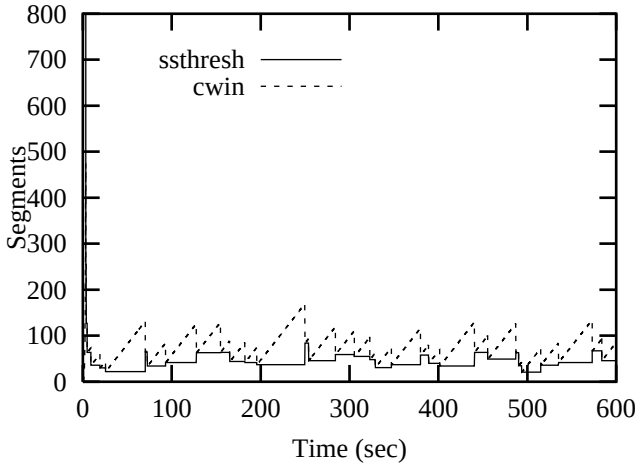

Fig. 5. TCP Reno with concurrent UDP traffic on lossy link - congestion window and *ssthresh*

Figure 7 compares the scalability of each TCP version running over a 45 Mb/s bottleneck with a 500-ms end-to-end delay (e.g., a scenario featuring a satellite hop): results point out the versatility of TCP Westwood, which achieves a very high goodput when few other connections are present, while managing to provide an acceptable performance when more connections share the same bottleneck.

The average goodput of 10 connections is shown in Figure 8 for increasing values of the bottleneck capacity (again the end-to-end delay was 500 ms). At the highest tested capacity (150 Mb/s) TCP Westwood outperforms the standard TCP versions because its aggressiveness pays off especially for long, fat pipes, where it is important to send the largest amount of data (when feasible) in the shortest time.

Westwood performs better than Reno under all tested RTTs, and is only slightly outperformed by Sack if the end-to-end delay is smaller than 0.2 seconds, as depicted in Figure 9 (10 connections, 45 Mb/s link capacity). Indeed, for small RTTs, Sack manages to recover fast enough so as to make up for its inherent lack of aggressiveness (when compared to Westwood). TCP Westwood, on the other hand, suffers from being too aggressive for small RTTs, and the resulting poorly-accurate bandwidth estimation forces it into slow start too often.

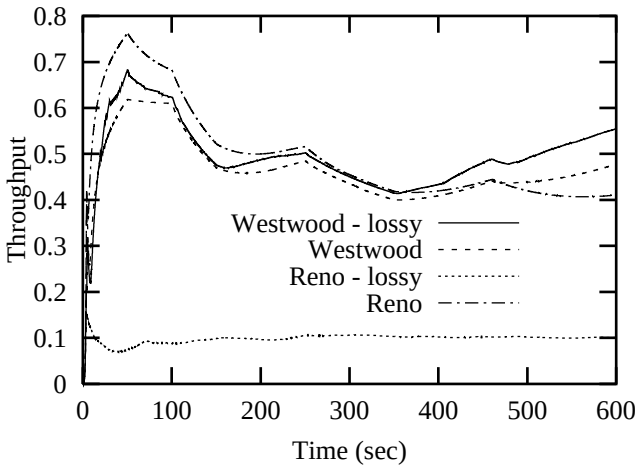

Fig. 6. TCP Westwood and Reno throughput comparison over non-lossy and lossy link

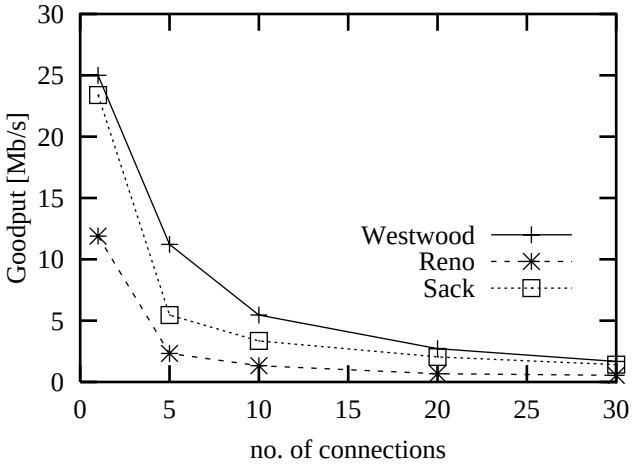

Fig. 7. Goodput as a function of number of concurrent connections

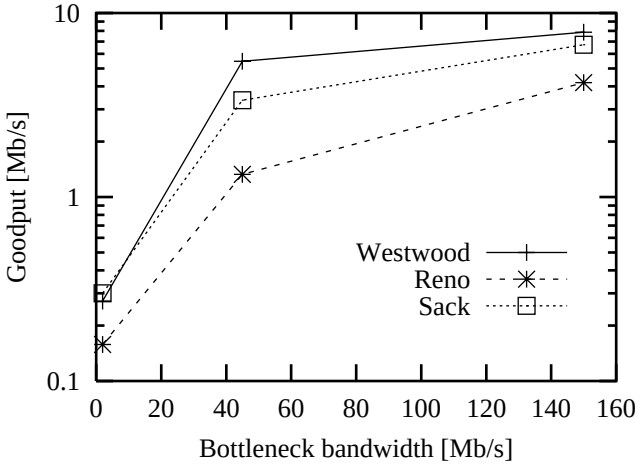

Fig. 8. Goodput as a function of bottleneck bandwidth

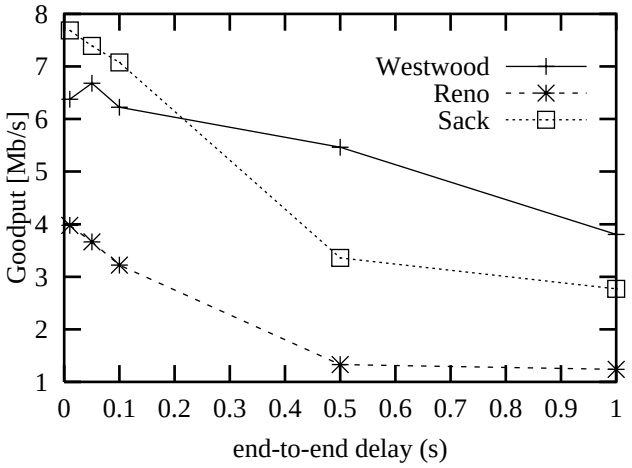

Fig. 9. Goodput as a function of one-way end-to-end delay

C. Performance over lossy links

Bursty traffic experiments have shown that TCP Westwood constantly outperforms Reno, and has a clear competitive edge over Sack in almost every test. However, it is the performance over lossy links, for which TCP Westwood was originally designed, that shows the most promising results.

In Subsection V-A, we already showed the superiority of TCP Westwood in a dynamically changing traffic environment. In this Section, we extend the experiments by considering the impact of the number of connections, bandwidth and end-to-end delay. The scenarios are identical to the previous Section (including the interfering On/Off UDP connection), but the link is "lossy" with the same probability as in Subsection V-A.

As in previous experiments, we ran 600-second tests under several settings: Figure 10 compares results for different number of connections (1 through 30); Figure 11 for different bottleneck bandwidths (2 through 150 Mb/s) and Figure 12 for different end-to-end delays (10ms through 1 sec). The results underscore that TCP Westwood is particularly suited for lossy links, under almost all scenarios, as predicted.

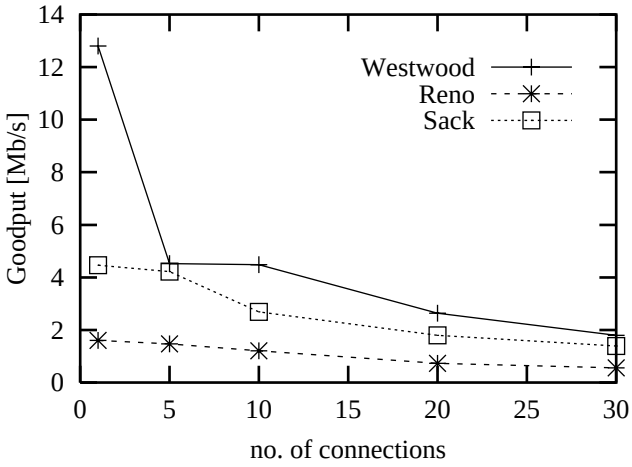

Fig. 10. Goodput as a function of number of concurrent connections

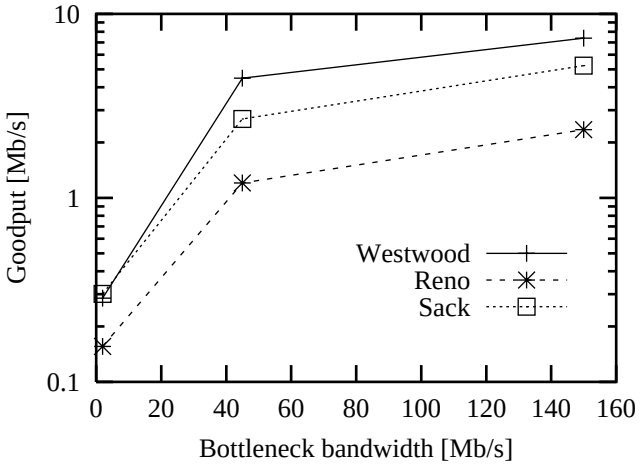

Fig. 11. Goodput as a function of bottleneck bandwidth

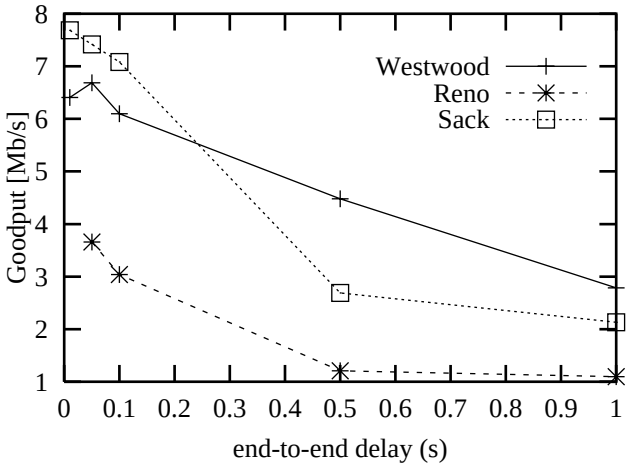

Fig. 12. Goodput as a function of one-way RTT

VI. Conclusions

In this paper we have proposed a new version of the TCP protocol, aimed at improving its performance under random or sporadic losses. The new version has been tested through simulation, showing considerable gain in terms of goodput in almost all scenarios.

Our modifications can be viewed as a further step in the evolution from TCP Tahoe to TCP Reno. TCP Tahoe was modified to TCP Reno by introducing fast recovery, which is a way to shrink the congestion window after 3 duplicate ACKs. TCP Tahoe resets *cwin* to one after a loss, whereas TCP RENO halves *cwin* after three duplicate ACKs. Now, TCP Westwood introduces ”faster” recovery to avoid over-shrinking *cwin* after three duplicate ACKs by taking into account the end-to-end estimation of the bandwidth available to TCP. Therefore, modifications required to implement TCP Westwood are comparable to the ones implemented in the transition from TCP Tahoe to TCP Reno.

Further work is in progress, especially in regard to the friendliness toward other connections employing TCP Tahoe or Reno. Also, further refinements of the bandwidth estimation process as well as of various algorithm tuning parameters are under study.

REFERENCES

[1] V. Jacobson. Congestion Avoidance and Control. *ACM Computer Communications Review*, 18(4):314–329, August 1988.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] C. Casetti, M. Gerla, S.S. Lee, S. Mascolo, and M. Sanadidi. TCP with Faster Recovery. To appear at *IEEE MILCOM'2000*, Los Angeles, CA, USA, September 2000.

[8] J.C. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *Proceedings of ACM SIGCOMM'96*, Stanford, CA, USA, August 1996.

[9] W.R. Stevens. *TCP/IP Illustrated, vol. 1*. Addison Wesley, Reading, MA, USA, 1994.

[10] I. Stoica, S. Shenker, and H. Zhang. Core-Stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High Speed Networks. In *Proceedings of the ACM SIGCOMM'98*, Vancouver, Canada, September 1998.

[11] D. Lin and R. Morris. Dynamics of Random Early Detection. In *Proceedings of the ACM SIGCOMM'97*, Cannes, France, September 1997.

[12] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options. RFC 2018, April 1996.

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

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