TCP Dynamics in 802.11 Wireless Local Area Networks

Sumathi Gopal*
WINLAB, Rutgers University
671, Rt. 1 South,
North Brunswick, NJ 08902
sumathi@winlab.rutgers.edu

Sanjoy Paul
WINLAB, Rutgers University
671, Rt. 1 South,
North Brunswick, NJ 08902
sanjoy@winlab.rutgers.edu

***Abstract*— In 802.11 wireless links with disabled MAC retries, data and ACK packets within a TCP session collide resulting in packet losses. We show in this paper that in this situation, certain popular optimizations of TCP (*fast-recovery)* worsen the performance by causing deadlocks that terminate with timeouts. We compare the performance of optimized TCP versions (Reno and NewReno) with an earlier version of TCP (Tahoe) and demonstrate the degradation during *fast-recovery*. TCP-Tahoe gains 80% in throughput over TCP-Reno and more moderately over TCP-NewReno. A key contribution of this paper is the visualization of TCP dynamics to capture MAC layer collisions between DATA and ACK packets of a TCP session, and the differences in the behavior of protocols in that situation. This case of poor TCP performance due to self-inflicted losses, makes a sound case for decoupling error and flow control algorithms for transport over 802.11 wireless networks.**

Keywords-TCP, 802.11 wireless LANs, TCP-802.11 interaction, interference

I. INTRODUCTION

It is now known that TCP does not perform well over 802.11 wireless networks [1][11]. In this paper we follow TCP dynamics to explore the reasons for poor performance in *noise-free* 802.11 wireless LAN environments. We show that the commonly used TCP-Reno version does not perform well because of several timeouts in the course of a flow. In fact the earlier less-optimized version - TCP-Tahoe outperforms TCP-Reno. NewReno which enhances Reno's congestion control also demonstrates poor performance and Tahoe performs better for some parameter settings.

Poor TCP performance in noise-free single-hop 802.11 links was identified in earlier papers [1][2]. In 802.11 wireless networks, access to the wireless medium is shared between all nodes transmitting within hearing range of each other. For a TCP flow, this causes bandwidth sharing and collisions of TCP data and ACKs, even if they are of the same flow. In the rest of this paper, we shall refer to this phenomenon as *self-interference*. The earlier papers primarily investigated bulk TCP thoughput and proposed skipping TCP ACKs [1] or limit congestion window size [11]. But the *exact reasons* for poor protocol performance were not understood.

Our detailed analysis of TCP dynamics result in the following insights:

**Supported by a student fellowship grant by Corporate Research, Thomson Inc., Princeton NJ.*

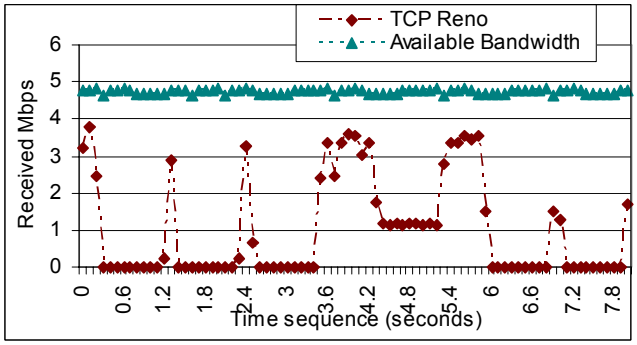

Figure 1: Instantaneous goodput of TCP-Reno during a 1MB file transfer in an error-free 802.11 wireless LAN

- *Poor Reno performance:* Losses due to self-interference often occur in quick succession (before the first loss is perceived by TCP sender). This results in multiple losses in a TCP congestion window. Reno's congestion-control is known to deadlock in such situations [8]. In Figure 1, the **six** "flat goodput" intervals result from deadlocks that end in timeouts.
- *Poor NewReno performance:* NewReno deadlocks when retransmissions are lost. Self-interference increases this likelihood, even during *fast-recovery*.
- *Tahoe outperforms Reno, and in some cases, NewReno:* TCP Tahoe gains in *throughput* over Reno and NewReno because of fewer deadlocks (that end in timeouts) in the self-interference scenarios.

We explain these insights with figures depicting TCP-MAC interaction, dynamics of various TCP flavors during congestion control and the effect of some TCP parameters on overall performance.

To maximize bandwidth utilization despite losses from self-interference, we suggest use of a transport protocol that separates error and flow control algorithms [10]. An example is the new Cross Layer Aware transport Protocol (CLAP) that achieves considerable gains over TCP in various wireless scenarios (Section VI and [3]).

The rest of the paper is organized as follows. Simulation setup is described in Section II. In Section III, we present TCP-dynamics during self-interference and evaluate the throughput cost incurred due to TCP-ACKs. In Section IV, we compare

Overhead	Duration (µs)
DIFS	50
Average duration of random backoff for min. MAC contention window	310
Physical layer: short Preamble(144bits/2Mbps) + PLCP header (48bits/2Mbps)	96
MAC header + FCS duration (8*34bytes/11Mbps)	24.73
LLC + IP headers (8*(8+8)bytes/11Mbps)	11.64
Time taken by Additional SIFS + MAC-ACK, after successful delivery of the Layer-4 packet	10 + 304 = 314
40-byte TCP Header duration (40*8 bits/11Mbps) = 40 byte TCP-ACK duration	29.1
Total Overhead time for each Layer-4 packet (T_{ACK})	**835.47**
Duration of 1000-byte TCP data segment (T_{DATA}) (1000*8 bits/11 Mbps)	**727.28**

Table 1: 802.11 overheads incurred by a TCP packet

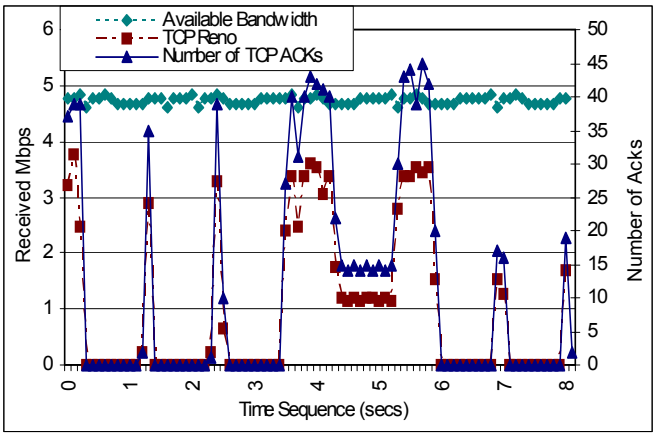

Figure 2: Number of TCP-ACKs received compared to the instantaneous data bytes

TCP-dynamics during congestion control in various TCP flavors and explain reasons for poor Reno/NewReno performance. In Section V we evaluate the effect of the minimum retransmission timeout parameter (*minrto_*) on TCP performance. In Section VI, we present the need for a novel approach to transport over 802.11, followed by Conclusion in Section VII.

II. Simulation Setup

Results presented in this paper were obtained with NS2 simulations (version 2.1b9a). A wireless LAN topology was used, comprising of an Access Point, a wireless node as TCP sender, and a wired node as TCP receiver. A single duplex-link with 10Mbps bandwidth and 2ms delay connected the wired receiver node and the Access Point (AP).

The wireless LAN was operated in 802.11b DCF mode at 11Mbps channel rate (the net data rate at the transport layer is about 5Mbps due to various overheads given in Table 1). The basic rate was set at 1Mbps (Long Phy Preamble). Strictly standard 802.11 MAC was used, disabling MAC retries and RTS/CTS. While these enhancements are default in NS2, they are suggestions in the 802.11 standard [9]. When MAC retries are enabled, a lone TCP session in a noise-free link seldom experiences lost packets. However there is still bandwidth sharing with ACKs and possible delay-variance.

Default TCP parameter settings of NS-2.1b9a were used in all the simulations, except for changes in the minimum retransmission timeout (*minrto_*) setting. The instantaneous link bandwidth was obtained by using a saturating UDP flow between the same pair of nodes in the same wireless scenario. The channel was error free, and there was no other interfering traffic. Figures depicting dynamics of Reno and NewReno over wireless LAN (Figures 3, 4, 5, 7) were drawn from simulation traces. Figure 6 depicting TCP-Tahoe dynamics was drawn anticipating possible Tahoe behavior in that situation.

III. The Cost of TCP Acknowledgements

A. TCP-ACKs are expensive

In the *shared medium* of 802.11 wireless links, signals from multiple transmitters could result in a collision at a receiver. To minimize collisions, the MAC layer implements the CSMA/CA protocol and nodes contend for channel access to send each packet [9]. A packet is sent only when the channel is found to be idle and after a random backoff.

A TCP-ACK comprises of 40 bytes of TCP header. However various time overheads in MAC and Physical layers causes it to consume a significant portion of channel time. Table 1 specifies the various overheads for 11 Mbps channel rate (802.11b). Following is the average time consumed transmitting a TCP data/ACK (assuming previous transmission was successful):

$$T_{PACKET} = T_{DIFS} + T_{PHY\_PREAMBLE,\ HDRS} + T_{MAC\_BACKOFF,\ HDRS} + T_{SIFS,\ MAC\text{-}ACK} + T_{IP\_HDR} + T_{TCP\_HDR} + T_{DATA} \quad (1)$$

$$= T_{ACK} + T_{DATA}$$

where T_{ACK} is the total time taken to transmit a TCP-ACK, given the same header size in TCP data and ACK packets. DIFS and SIFS are the Distributed and Slot Inter-Frame-Spaces respectively introduced by the MAC layer.

Each TCP packet (data/ACK) incurs an average overhead of 806.37µs and constitute **99.7%** of the channel time consumed by a TCP-ACK packet.

The bandwidth consumed by *n* ACKs in a unit interval *i*, at the expense of data packets may be calculated as follows:

Number of data packets that could have been sent:

$$k = n * T_{ACK} / (T_{DATA} + T_{ACK}) \quad (2)$$

$$\text{Lost bandwidth} = (k*1000*8) / i \text{ bits/sec} \quad (3)$$

The frequency of TCP-ACKs in each unit interval in the course of the 1MB file transfer is depicted in Figure 2. At peak operation, the number of returning ACKs is also at its peak - an average of 40 ACKs in a 0.1 second interval. At this time, the lost bandwidth from equations (2) and (3) is **1.47 Mbps**. In Figure 2 this number matches the difference between TCP's

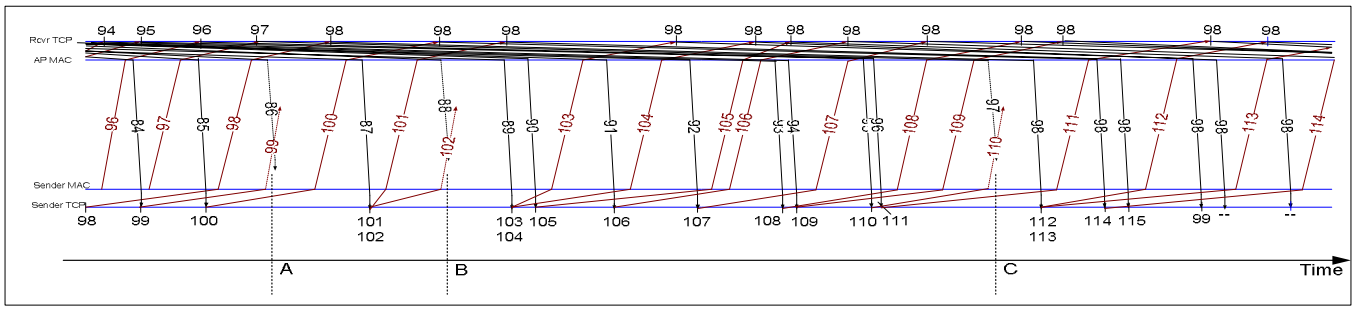

Figure 3: Self-interference during peak TCP operation when TCP congestion window = 15 segments

peak instantaneous throughput and the available bandwidth, corroborating the ill-effects of self-interference on TCP throughput. A more expensive effect however is packet loss that triggers congestion control in TCP.

B. Packet loss due to self-interference

In Figure 2, when the throughput is at its peak, the number of returning ACKs is also at its peak. Figure 3 captures the dynamics of TCP and 802.11 MAC during this time. It demonstrates the situations when self-interference results in three MAC collisions in quick succession of each other (instants A, B and C), before TCP-sender detects the first loss. Hence all the losses occur within a single congestion window (15 segments at this time).

A vertical cross section between a TCP and MAC process (in Figure 3) shows how many packets are waiting in the MAC queue (an indirect conclusion on the AP side). Clearly with a consistent supply of TCP packets, the MAC operates in saturation. From earlier papers, the collision likelihood in this MAC situation is 3% [1].

TCP sender detects a loss (the first one) only when three duplicate ACKs arrive. It then scales down the congestion window (and hence the sending rate) despite a high bandwidth availability in the wireless link at that instant. We show in the next section that the situation of multiple losses in a congestion window, often leads to timeouts in Reno and NewReno, that are a lot more expensive.

IV. TAHOE OUTPERFORMS RENO IN 802.11 WIRELESS LINKS

Tahoe, Reno and NewReno operate identically in the normal operation mode (no losses), but differ in their congestion control algorithms. Figures 4, 5 and 6 depict congestion control algorithms of Reno, NewReno and Tahoe respectively, triggered in response to multiple losses of Figure 3. All three versions implement fast-retransmit where the segment is retransmitted upon three duplicate ACKs. (They also implement limited-transmit (RFC 3042) [7], where the first and second duplicate ACKs trigger transmission of up to two data segments over the congestion window.). They differ in how they adjust the congestion window after fast-retransmit.

Tahoe scales it down to 1 segment, forgetting all about higher sequence number segments that were already sent (at instant A in Figure 6). No more packets are sent until an ACK arrives confirming the retransmitted packet. Normal operation subsequently resumes in slow-start mode.

Reno and NewReno cut down congestion window in half after *fast-retransmit* (at instant A in Figures 4 and 5) and "fill the pipe" with new data packets until a new ACK arrives. This is the *fast-recovery* mode of operation where the congestion window is incremented by one segment for each duplicate ACK. New packets are sent when congestion window exceeds the number of already outstanding packets.

Reno exits *fast-recovery* and resumes normal operation when the first non-duplicate ACK arrives recovering from the first loss. In case of multiple losses (before instant A), Reno's fast-recovery ends when there are more outstanding packets than the congestion window (at instant B in Figure 4). The congestion window is cut down further when recovering from the subsequent losses. In Figure 4, Reno enters congestion control to recover the 2nd loss at instant C, and exits at instant D. Since the congestion window is small, no new packets are sent between instants B and D. Subsequently there are no more ACKs and the deadlock situation results at instant D. It ends in a timeout.

To overcome the deadlock, NewReno [5] continues in fast-recovery until all losses (that occurred before instant A in Figure 5) are recovered. The segment is retransmitted when the first ACK indicating it arrives (at instant B in Figure 5). The congestion window is halved after each retransmission. NewReno recovers all losses if and only if all retransmissions are successful. A timeout occurs otherwise. In Figure 5 depicts a deadlock situation that occurs in NewReno when a packet retransmitted during fast-recovery is lost. (packet #102 at instant C in Figure 5). Subsequent duplicate ACKs grow the congestion window, but this not sufficient to send new data packets. With no more data packets, no ACKs are triggered and a deadlock situation occurs (at instant D in Figure 5).

The comparison of instantaneous goodputs of Tahoe, Reno and NewReno in the course of the 1MB file transfer in the said wireless scenario is depicted in Figure 8. Despite being the least optimized version of TCP, Tahoe completes the file transfer in a significantly shorter time compared to Reno and NewReno. TCP dynamics reveal the following insights:

- *Reno suffers multiple timeouts*: With the frequent occurrence of multiple losses in a TCP congestion window, Reno *deadlocks* several times that end in timeouts. The 1-second duration of each interval is due to the *minimum retransmission timeout* setting (*minrto_*).

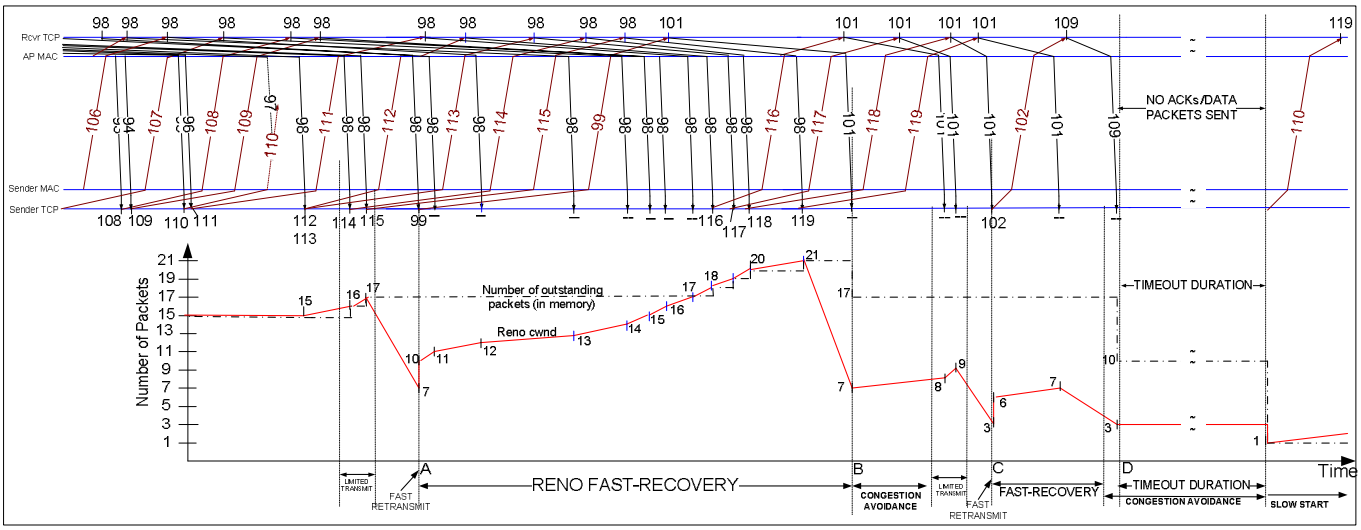

Figure 4: Congestion Control in TCP-Reno following multiple losses of Figure 3

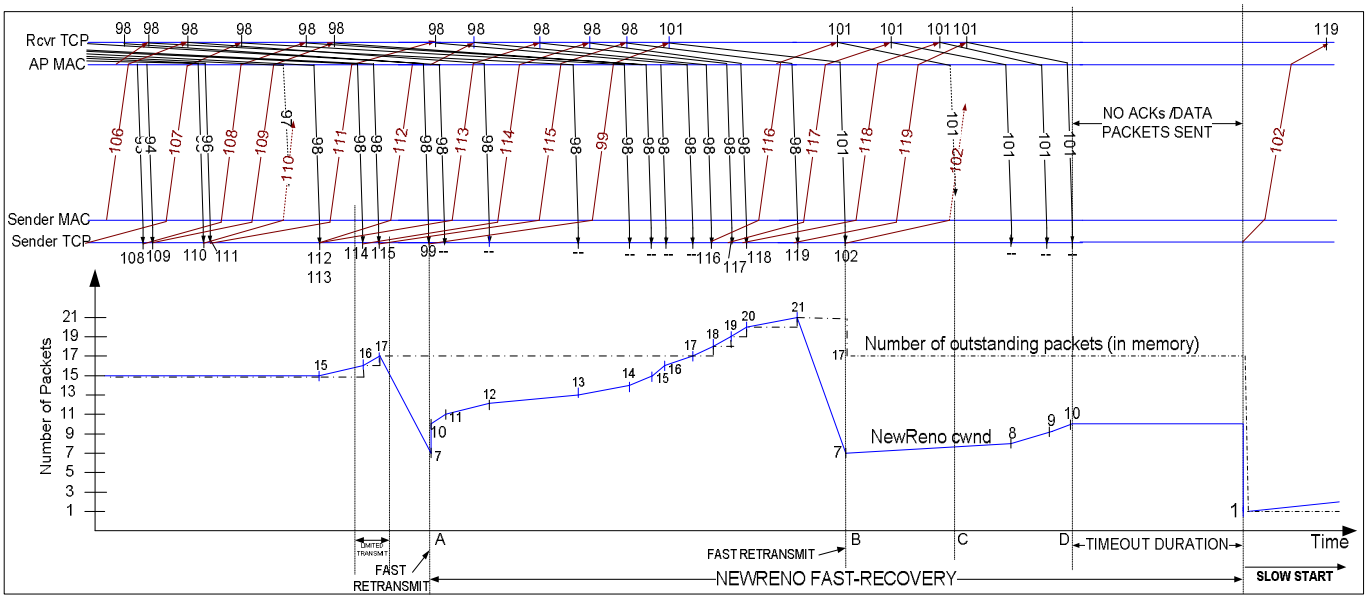

Figure 5:Congestion control in NewReno following multiple losses of Figure 3

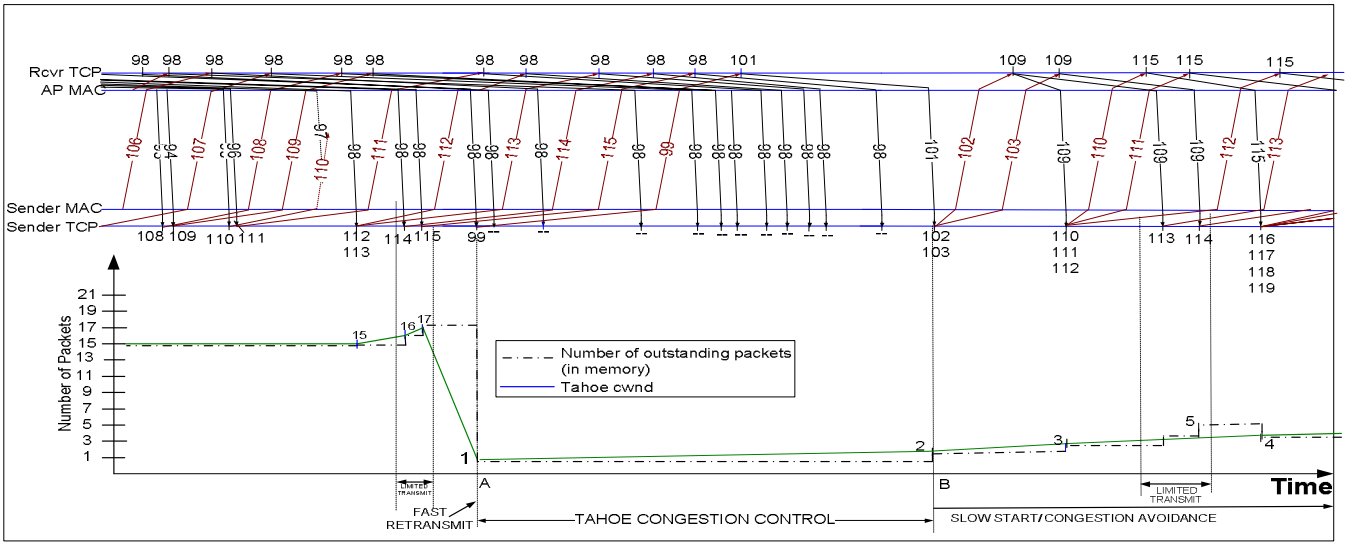

Figure 6:Tahoe's congestion control after multiple losses of Figure 3

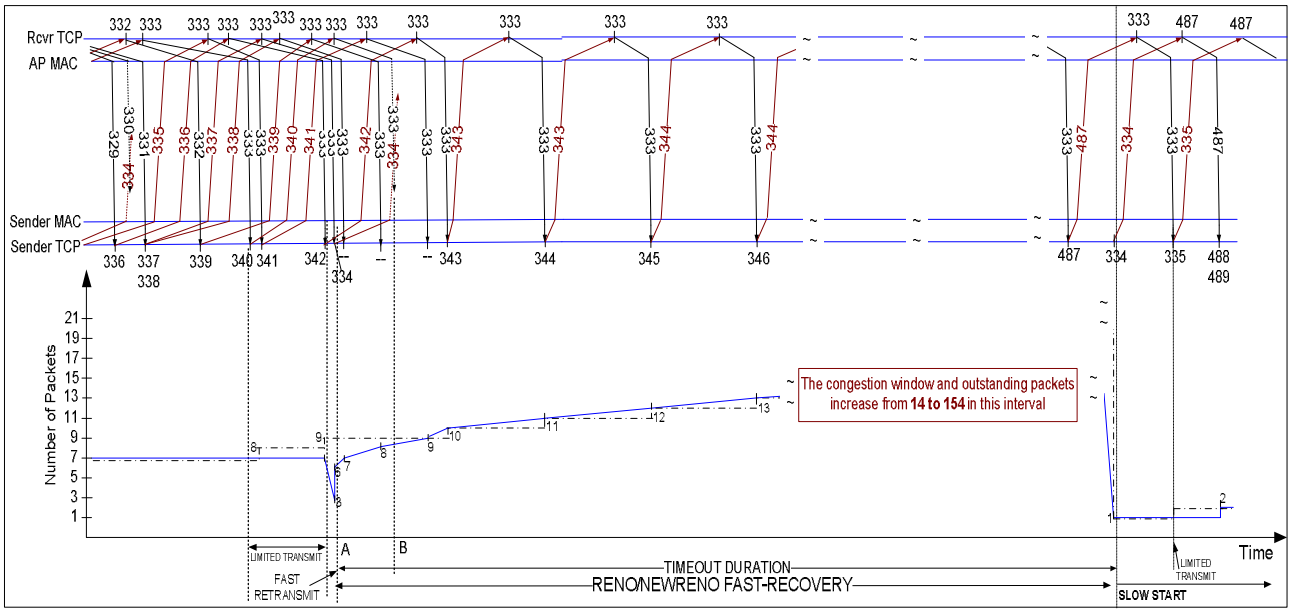

Figure 7:Reno/NewReno congestion control when the first retransmission is lost following self-interference

▪ *NewReno also suffers multiple timeouts:* During fast-recovery, NewReno sustains a "full transmission pipe" by sending new packets. Thus the probability of self-interference is still large during fast-recovery. In the example of Figure 8, deadlock situations occur three times in the course of the 1MB file transfer, and end in timeouts (after minrto_ of 1 sec). They all occur due to loss of the first retransmission, or of one of the packets sent during fast recovery.

▪ *A low throughput is sustained in Reno/NewReno in some timeout intervals*: Figure 7 captures the dynamics of TCP NewReno/Reno the scenario when the retransmission at the start of congestion control is lost (at instant B in Figure 7). Despite this, duplicate ACKs #333 sustain the growth of the congestion window, and consistently trigger new data segments. In the example of Figure 7, congestion window and the number of outstanding packets reach *154 segments* at the end of fast-recovery. With a single packet in transit at a time, the likelihood of self-interference diminishes to *zero*. But with stop-and-wait approach during this interval, the goodput drops to a minimum. In Figure 8, Reno starting at 4.4 seconds, and NewReno at 1.9 and 3.3 seconds experience this situation.

▪ *Tahoe outperforms Reno. Outperforms NewReno considerably for minrto_ = 1 second.* Tahoe operation following multiple losses in a congestion window is depicted in Figure 6. By *not* implementing fast-recovery and resuming operation in slow-start soon after fast-retransmit (at instant A in Figure 6), Tahoe reduces the loss likelihood due to self-interference, improving the resilience to loss recovery. Several duplicate transmissions of data segments could ensue. But Tahoe does not deadlock and timeout as long as the retransmission is successful. Hence the likelihood of a deadlock in Tahoe is far lower than in Reno and NewReno. Tahoe's gain in goodput during the additional timeout periods of Reno and NewReno offsets the bandwidth wasted by redundant retransmissions and the lack of "pipe-filling" during loss recovery.

V. EFFECT OF *MINRTO_* ON TCP PERFORMANCE

In the wireless LAN scenario considered here, the round-trip time fluctuated 7 - 30 milliseconds. RFC 2988 [6] stipulates a minimum timeout duration (*minrto_* in NS2) of 1 second. This was to avoid spurious timeouts and retransmissions in TCP in wired nets with large fluctuating round trip times. In our scenario, Reno and NewReno waste several 1-second intervals in a deadlock before the timeout occurs and resumes the sending rate.

More recent TCP implementations set minrto_ to 0.2 seconds. Figure 9 compares the net throughputs of Reno, NewReno and Tahoe for various minrto_ settings. minrto_ = 0 implies that the retransmission timeout duration is completely based on the estimated round-trip time. With these settings, Reno and NewReno still experience the same number of timeouts but gain in throughput because of the shorter time spent in deadlock, and Tahoe's gain over Reno and NewReno diminishes.

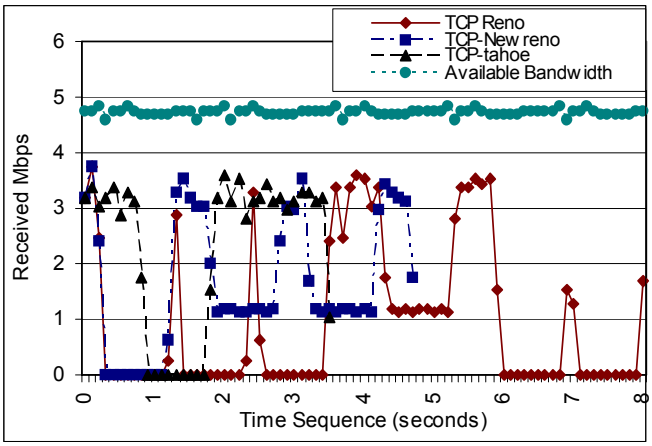

Figure 8:Instantaneous goodputs of Tahoe, Reno and NewReno during 1MB file transfers

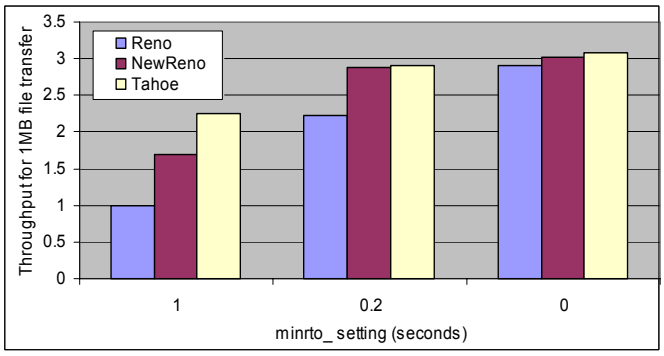

Figure 10: Performance of various TCP flavors for different values of minimum retransmission timeout

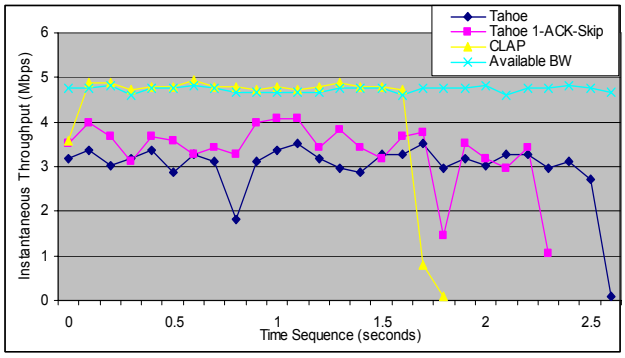

Figure 9: Comparison of CLAP with TCP-Tahoe with/without ACK-skipping for minrto_ = 0

VI. CASE FOR CROSS-LAYER AWARENESS

The self-interference problem is alleviated by separating flow and error control and reducing feedback packets, as demonstrated by the new CLAP protocol It also uses a rate *rate-based* flow control algorithm that incorporates *cross-layer* status information.

Earlier papers have found that skipping a single ACK achieves considerable gain in the net throughput due to reduction in self-interference [2]. Figure 10 compares the performance of Tahoe, Tahoe with 1-ACK-skip and CLAP (1MB file transfer). Since there are no out of order packets, minrto_ is prudently set to 0. Hence only a small fraction of a second is spent before a timeout. The gain with 1-ACK-skip, is 12%, while CLAP gains 95%. For more details of CLAP performance in time-varying 802.11 scenarios, refer to [3].

VII. RELATED WORK

All TCP enhancements proposed for wireless networks, are extensions of either TCP-Reno or TCP-NewReno. To the best of our knowledge ours is the first attempt to investigate the dynamics of operation of different TCP versions in an 802.11 scenario.

The enhancement protocols may be clearly categorized into those for cellular networks and those for multi-hop 802.11 networks. Cellular networks however do not operate in a shared medium, and hence do not suffer the MAC problem. Some papers addressing TCP in multi-hop wireless have proposed limiting the congestion window in decreasing proportion to the number of hops, to reduce interference [11]. But this drastically limits TCP throughput in the multi-hop scenario.

VIII. CONCLUSION

Although *fast-recovery* is an efficient congestion-control algorithm widely used in wired nets, it often causes deadlock situations in wireless links that end in a timeout. This paper has examined TCP dynamics during congestion control of Tahoe, Reno and NewReno versions that encounter in different timeout situations. We have showed that less-optimized TCP-Tahoe that forgets all outstanding packets after *fast-retransmit* gains significantly over Reno and NewReno that use *fast-recovery*.

Self-interference in TCP cannot easily be mitigated because of its tightly intertwined error and flow control mechanisms. The window-based flow control is heavily dependent on the pace and quality of returning ACKs. TCP also suffers various other challenges over wireless networks that primarily stem from its layer-independent design. On the other-hand, lower-layers, particularly in wireless nodes are equipped to provide status information that can be used to supplement decision-making processes in the transport protocol. An alternative is to use the new CLAP protocol [3] for data transport in wireless scenarios.

REFERENCES

[1] S. Gopal, S. Paul, D. Raychaudhuri," Investigation of the TCP Simultaneous Send problem in 802.11 Wireless Local Area Networks", IEEE ICC 2005, May 16-20, Seoul, South Korea.

[2] S. Gopal, D. Raychaudhuri, "Experimental Evaluation of the TCP Simultaneous Sent problem in 802.11 Wireless Local Area Networks", ACM SIGCOMM Workshop on Experimental Approaches to Wireless Network Design and Analysis (EWIND-05), August 22nd 2005, Philadelphia.

[3] S. Gopal, S. Paul, D. Raychaudhuri "Leveraging MAC-layer information for single-hop wireless transport in the Cache and Forward Architecture of the Future Internet", WILLOPAN Workshop,)held in conjunction with COMSWARE 2007, Bangalore, INDIA, January 2007

[4] RFC 2581 : "TCP Congestion Control with Fast-Retransmit and Fast-Recovery" http://www.ietf.org/rfc/rfc2581.txt

[5] RFC 2582: "NewReno modification to TCP's Fast Recovery" http://www.ietf.org/rfc/rfc2582.txt , April 1999

[6] RFC 2988: "Computing TCP's Retransmission Timer", http://www.ietf.org/rfc/rfc2988.txt

[7] RFC 3042: "Enhancing TCP's Fast-Recovery Using Limited Transmit", http://www.ietf.org/rfc/rfc3042.txt

[8] J. C. Hoe, "Improving the start-up behavior of a congestion control scheme for TCP", In Proceedings of the ACM SIGCOMM '96, pages 270 - 280, Stanford, CA, August 1996

[9] IEEE 802.11, 1999 Edition (ISO/IEC 8802-11: 1999) Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications

[10] D.D.Clark, M.L. Lambert, L. Zhang, "NETBLT: a High Throughput Transport Protocol", ACM SIGCOMM 1987

[11] Fu, Z.; Zerfos, P.; Luo, H.; Lu, S.; Zhang, L.; Gerla, M., "The impact of multihop wireless channel on TCP throughput and loss" Proc. of INFOCOM 2003. Volume 3, 30 March-3 April 2003 Page(s):1744 - 1753 vol.3