# TCP In Conflict-Based Wireless Links

*Gagandeep Singh Brar[1], G.N. Singh[2], Anupam Bhatia[3]*
*[1] Department of Computer Science,*
*D.A.V. Colege, Sector 10, Chandigarh, Punjab, India*
*Email: gagandeep_brar@yahoo.com*
*[2] Department of Computer Science, SD College, Lalgaon, Rewa (MP), India*
*gnsingh_ph_2001@rediffmail.com*
*[3] Department of Computer Science*
*Kurukshetra University Regional Centre, Jind (Haryana), India*
*bhatianupam@gmail.com*

***Abstract :*** *TCP's bidirectional traffic causes self-interference in contention-based wireless links as in IEEE 802.11 wireless LANs and results in the loss of TCP data segments and ACKs. The fast-recovery algorithm is the basis for congestion control in most TCP enhancements proposed to address wireless network characteristics. However we show in this paper, that fast-recovery worsens performance during self-interference by causing deadlock situations that only terminate with a timeout. Both Reno and New Reno are evaluated in comparison to the lesser-optimized TCP-Tahoe to demonstrate the degradation in performance during fast-recovery. The less-optimized TCP-Tahoe that forgets all outstanding packets soon after fast-retransmit, outperforms TCP-Reno with an 80% gain in throughput. For the minrto_ parameter set to 1 second, Tahoe outperforms New Reno by a significant margin. A key contribution in this paper is the visualization of TCP dynamics that capture MAC layer collisions due to self-interference, and the protocols' behavior during congestion control. The paper demonstrates the disadvantages of combined error and flow control and makes a sound case for cross-layer awareness in transport protocols over wireless networks.*

## I. INTRODUCTION

IEEE 802.11 based wireless networks are commonplace in homes, offices and even entire cities now. These networks operate in the Distributed Coordination Function

(DCF) mode of MAC, where nodes have to contend for channel access on a per-packet basis with equal priority for transmission. A TCP flow comprises of data and acknowledgement (ACK) packets traversing in opposite directions. In 802.11-DCF and other contention-based wireless links nodes compete for channel access to transmit them, resulting in *self-interference*. This reduces the net bandwidth available for data packets, and even causes packet losses due to MAC-layer collisions. The losses are forced to be handled by TCP when link-layer retransmissions are disabled. TCP-Reno is the most popular flavor of TCP used on the Internet. Its congestion-control uses *fast-retransmit* and *fast-recovery* algorithms that expedite detection and recovery of lost data segments in wired networks. TCP-NewReno enhances fast-recovery to tackle multiple losses within a congestion window. Due to their success on wired networks, all enhancements proposed for TCP over wireless networks extend the Reno/NewReno flavors.

With elaborate depiction of TCP dynamics in various scenarios, we show in this paper that the *fast-recovery* algorithm fails to perform efficiently following packet loss due to self-interference. The instantaneous *goodput* of TCP Reno during a 1MB file transfer over a *noise-free*, *cross-interference free* 802.11-DCF. There are *six* sizeable durations of low bandwidth utilization in the course of the flow that takes over 8 seconds to complete. Most of the inactivity durations result from *deadlocks* in the protocol after *fast-recovery*. Additionally, observe that the peak instantaneous throughput of TCP-Reno is 30% lower than the available bandwidth at that instant.

In this paper we present insights that explain these aspects of TCP behavior during self-interference:

(1) TCP's window behavior increases self-interference during peak TCP operation (large *congestion window*). This increases the likelihood of multiple losses in a *cwnd* in contention-based wireless links.

(2) Reno experiences a timeout in the event of multiple losses in a *cwnd* [8]. Hence there is a high likelihood of timeouts during Reno operation over contention-based wireless links.

(3) Timeouts in NewReno [5] occur due retransmission losses. They occur with increased likelihood due to "*pipe filling*" during *fast-recovery*, again an artifact of self-interference.

(4) Due to multiple timeouts, the setting for *minimum retransmission timeout (minrto_)* [6] significantly affects the net throughput of TCP Reno and NewReno.

(4) Without *fast-recovery*, TCP-Tahoe achieves goodput gains during timeouts of Reno and NewReno. It outperforms Reno by a significant margin for any *minrto_*, and NewReno for *minrto_* set to the default 1 second [6].

Separation of error and flow control could avoid the various durations of low bandwidth utilization in TCP that happen during timeouts despite high bandwidth availability. This separation enables the incorporation of more efficient flow and error control algorithms for wireless networks. We recently proposed *CLAP - Cross Layer Aware transport Protocol* incorporating these design considerations [3]. It uses an *aggregate NACK* (Negative ACKnowledgements) approach for error control that significantly alleviates self-interference. The flow control algorithm is *rate-based* (rather than window-based like in TCP) and is supplemented by regular link-layer feedback of available link rate. In the wireless scenario under consideration, CLAP outperforms TCP (any flavor) by at least 30%.

The rest of the paper is organized as follows. Given the *insight-filled* nature of this paper, details of the simulation setup are presented *up-front* in Section II. In Section III, the throughput cost incurred due to self-interfering TCP-ACKs is evaluated while delving into TCP-dynamics *during* self-interference. In the following section (Section IV), we depict TCP-dynamics during congestion control in various TCP flavors and explain why Tahoe outperforms fast-recovery in the self-interference situation. We demonstrate the dependence of TCP performance on the value of *minrto_* in Section V. Having thus examined the core reasons for TCP under-performance in wireless links, a case for *cross-layer awareness* is presented in Section VI and Conclusion in Section VII.

## II. Simulation Setup

The insights in this paper are based on results obtained with simulations conducted in NS-2 simulator, version 2.1b9a. The network topology was that of a wireless LAN, 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 connects the wired node and the Access point.

802.11b at 11Mbps channel rate was used in the wireless LAN. The basic rate was set at 1Mbps (Long Preamble) .The MAC layer retransmissions were *disabled* in order to observe the effect of self-interference in TCP's congestion-control. *With* MAC retries and a single flow packets are seldom lost due to self-interference.

Default TCP parameters of NS-2 were used in all the simulations, unless otherwise mentioned (such as the *minrto_* value).

There was no other background traffic and there were no channel errors in the wireless link.

A handle on the instantaneous available link bandwidth was obtained by using a saturating UDP flow between the same pair of nodes in the same topology scenario. The TCP-MAC sequence diagrams are primarily drawn from simulation traces.

## III. The Cost of TCP Acknowledgements

### *A. TCP-ACKs are expensive*

In 802.11 wireless LANs, simultaneous traversal of two or more packets results in a collision. Nodes hence follow the CSMA/CA protocol [11] and contend with each other for channel access. 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 it consumes a significant portion of channel time. Each packet carries sizable overheads due to Phy/MAC headers, MAC random backoff, MAC-ACK and various inter-frame spaces. Following is the time taken for the transmission of a *single* TCP DATA segment over a contention-based wireless link:

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

where $T_{ACK}$ is the total time taken to transmit a TCP-ACK packet, DIFS and SIFS are the Distributed and Slot Inter-Frame-Spaces respectively.

Since the header size of a TCP data packet is the same as that of the ACK packet, and all other overheads remain the same, $T_{ACK}$ also represents the channel time consumed by the *total overhead* incurred by the TCP *payload*.

Table 1 supplies values of the various durations for packet transmission at 11 Mbps channel rate, some of which are available in other papers[9][10]. Some overheads such as those in the physical layer, DIFS/SIFS, and the average MAC random backoff, are independent of the channel rate of transmission of the packet (for example, at 11 Mbps) while the remaining overheads such as due to IP/TCP headers are a factor of the channel rate. In the time taken to transmit the 40 byte TCP-ACK, 99.7% is due to overheads. Each TCP packet (data/ACK) incurs a net overhead of 806.37ms.

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:

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

Lost bandwidth $= (k*1000*8) / i$ bits/sec (3)

The frequency of TCP-ACKs in each unit interval in the course of the 1MB file transfer is depicted in Figure … . At peak operation, the number of returning ACKs is also at its peak - an average of 40 ACKs in a 0.1 second interval. Substituting various values in equations (2) and (3), **the average lost bandwidth due to returning ACKs during peak TCP operation amounts to 1.47 Mbps**. In Figure .. *this number matches the difference between TCP's peak instantaneous throughput and the available bandwidth* and thus corroborates our insight.

This *lost bandwidth* for data packets during peak TCP operation is an artifact of *self-interference* in contention-based wireless links. A more expensive artifact though, is packet loss that triggers congestion control in TCP.

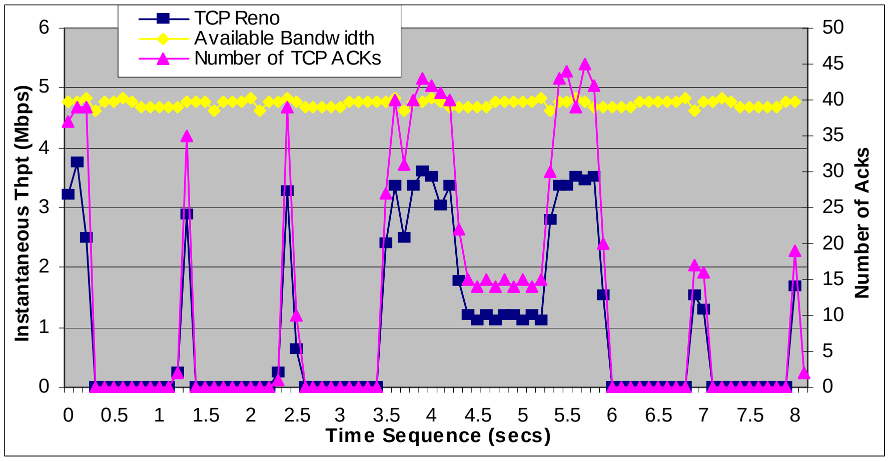
| **Overhead** | **Duration (ms)** |
|---|---|
| DIFS | 50 |
| Average duration of random backoff for min. MAC contention window | 310 |
| Physical layer: short Preamble(144bits/2Mbps) + PLCP header (48bits/2Mbps) | 96 |
| MAC header + FCS duration (8*34bytes/11Mbps) | 24.73 |
| LLC + IP headers (8*(8+8)bytes/11Mbps) | 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*

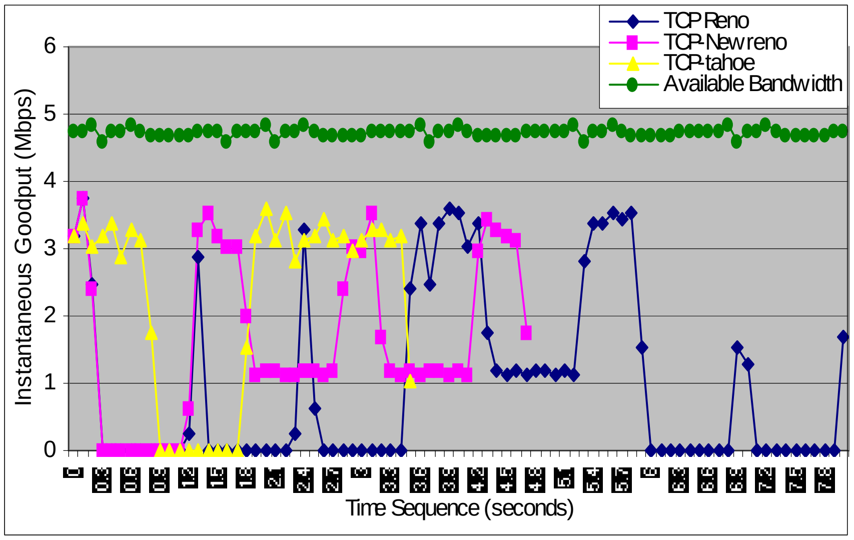
## B. Packet Loss due to Self-interference

Figure … demonstrates that when TCP operates at peak instantaneous goodput, the number of acknowledgements is also at its peak. The state of the *transmission pipe* between the sender and receiver during a large *cwnd* is depicted in Figure … . The sequence diagram captures the operational states of the two TCP peers and their corresponding 802.11 MAC layers (transmitting respective TCP data and ACK packets). The trace was obtained during an NS-2 simulation. The sequence number in data (ACK) packets sent by *Sender-TCP* (*Rcvr-TCP*) are represented by the outer labels.

Arrival of a segment at either TCP peers triggers a packet (data/ACK) transmission. The random transmission times from the MAC layers are due to random backoff independently selected for each packet. Thus the time when a packet *actually leaves the wireless node* is a function of



***Instantaneous goodputs of Tahoe, Reno and NewReno during 1MB file transfers***



**The number of acknowledgements received by the TCP-sender are determined by the TCP goodput at the receiver. The number of ACKs is maximum when TCP operates at peak instantaneous throughput**

Arrival of a segment at either TCP peers triggers a packet (data/ACK) transmission. The random transmission times from the MAC layers are due to random backoff independently selected for each packet. Thus the time when a packet *actually leaves the wireless node* is a function of queuing and transmissions delays. The region between *Sender-TCP* and *Sender-MAC* reveals that *at any instant* there is *at least one* packet in the interface queue waiting to be sent. Since the number of data and ACK packets are proportionate (Figure ..), similar will be the situation at the *AP-MAC*. This corroborates the insight that during peak TCP operation, the 802.11 MAC is operated in saturation when it consistently contends for channel access [1][2]. The likelihood of a MAC collision when two nodes consistently contend for channel access is 3% [1].

In the depicted example of self-interference in Figure .. the high likelihood results in 3 collisions in a congestion window, before the TCP sender perceives packet losses.

TCP notices packet losses only with the arrival of duplicate ACKs. At that time it cuts down the sending rate (*cwnd*) while the wireless link has high bandwidth available.

## IV. Tahoe Outperforms Fast-Recovery in Contention-Based Wireless Links

In the normal operation mode when no loss is perceived, Tahoe, Reno and NewReno behave identically. They also implement *limited-transmit* (RFC 3042) [7], where first and second duplicate ACKs trigger transmission of up to two data segments over *cwnd*. The three flavors differ in their congestion control algorithms. All three implement *fast-retransmit* where the segment is retransmitted upon three duplicate ACKs. Subsequently Tahoe scales down *cwnd* to 1 segment, forgetting everything about all segments that were sent after the lost one. Reno and NewReno on the other hand scale down *cwnd* to half its value and enter *fast-recovery* where each incoming duplicate ACK is used to grow *cwnd* and send new data segments. At the end of *fast-recovery, cwnd* is restored to its post-fast-retransmit value and normal operation in slow-start/congestion-avoidance is resumed.

Reno and NewReno differ in *when* they exit *fast-recovery*. Reno exits when the successful receipt of the retransmitted segment is confirmed (i.e. with the arrival of the first non-duplicate ACK during congestion-control). NewReno instead exits *fast-recovery* only with the successful receipt of all *cwnd* segments before congestion-control was triggered. Multiple segments could be retransmitted during *fast-recovery.*

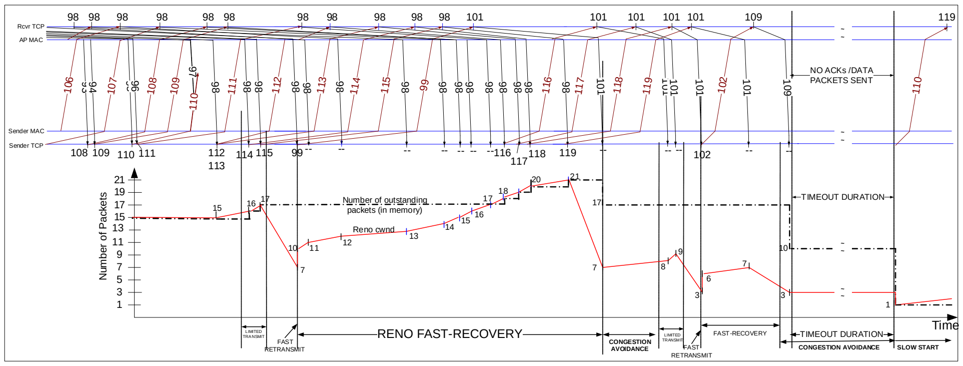
The comparison of instantaneous *goodputs* of Tahoe, Reno and NewReno in the course of the 1MB file transfer in the said wireless scenario is depicted in Figure .. . Several observations may be drawn:

(a) Tahoe completes the file transfer in a significantly shorter duration than Reno and NewReno, despite being the least-optimized version of them all.

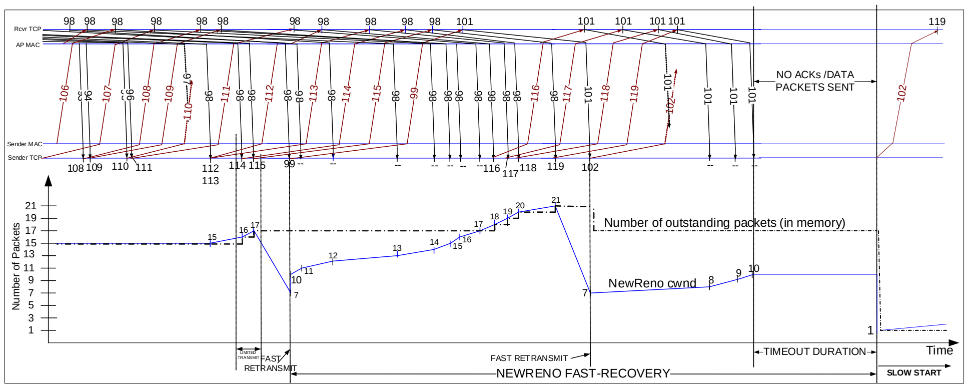
(b) There are several 1-second intervals of low-bandwidth utilization. Reno has six such intervals while NewReno and Tahoe have three and one respectively.

In the rest of this section, we present detailed explanation of the reasons behind this behavior of TCP in contention-based wireless links.

*Insight I : Reno suffers multiple timeouts*

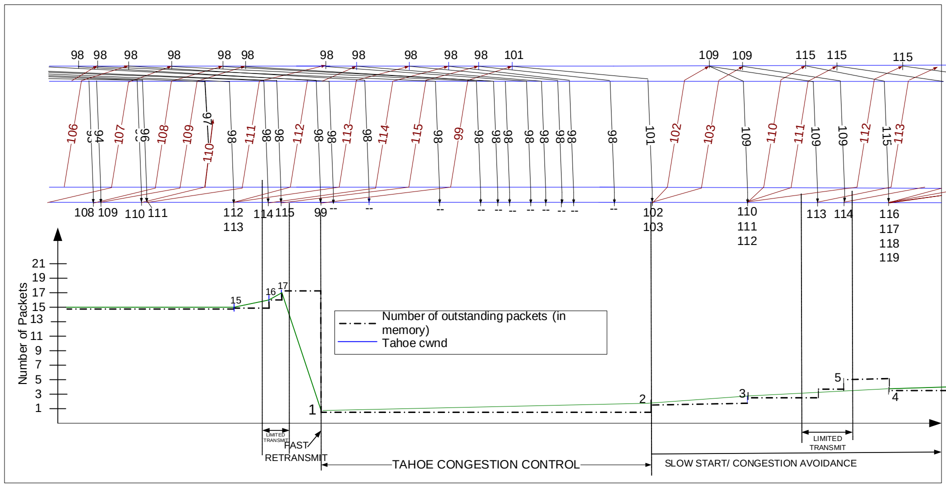


***TCP-Reno congestion control following multiple losses in a congestion window***



***Timeout during congestion control in NewReno after multiple losses in a congestion window***

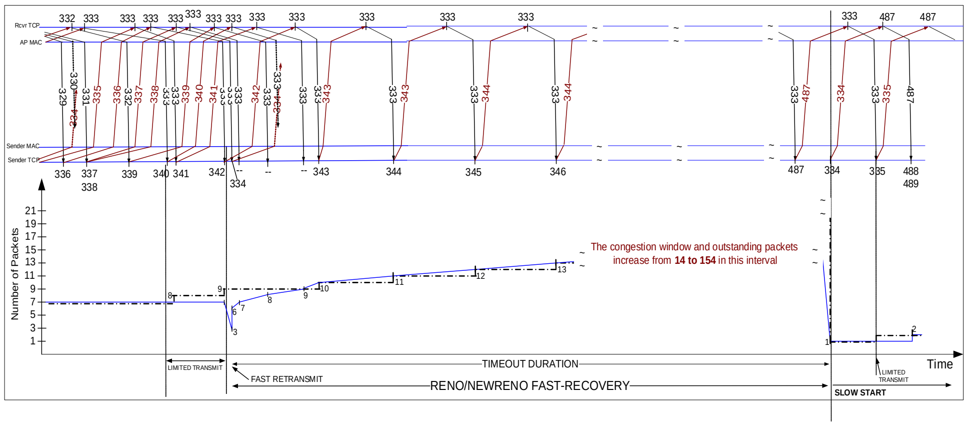
We showed in Section II that multiple losses in a *cwnd* are commonplace in contention-based wireless links due to self-interference. *Fast-recovery* in Reno is known to fail in this situation [8]. The dynamics of Reno's congestion control following one such situation is depicted in Figure .. Reno enters congestion-control *three* times in quick succession to recover each of the three lost packets. *Cwnd* is cut in half each time, rendering it too small to send any new segments, particularly after the *first fast-recovery*. ACKs thus cease to arrive, hampering *cwnd* growth. A *deadlock* situation results, that is released *only* with a timeout. In Figure .. this deadlock results in five 1-second intervals with zero goodput in Reno. The duration of each interval is due to the *minimum retransmission timeout* setting (*minrto_* ) of 1-second.



***Tahoe's congestion control after multiple losses in a congestion window***

## *Insight II: NewReno also suffers multiple timeouts*

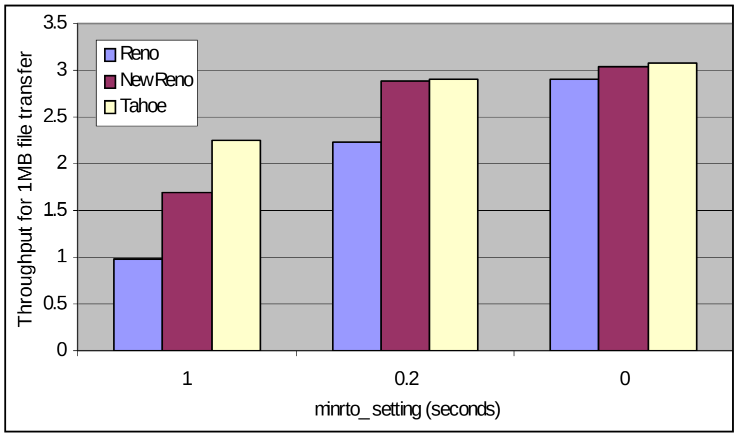
NewReno was proposed as a modification to Reno's *fast-recovery* in order to recover from multiple losses in a *cwnd* [5]. However we observe that NewReno still underperforms when *any one of the retransmitted segments is lost*, a situation that occurs with likelihood during self-interference. In fact, NewReno *deadlocks* when in a multiple loss condition, a retransmission *during fast-recovery* is lost.


Rcvr TCP
AP MAC
Sender MAC
Sender TCP
Number of Packets
Time
The congestion window and outstanding packets increase from 14 to 154 in this interval
LIMITED TRANSMIT
FAST RETRANSMIT
TIMEOUT DURATION
RENO/NEWRENO FAST-RECOVERY
SLOW START

***Reno/NewReno congestion control when the first retransmission is lost due to self-interference (an ACK is also lost)***

One such deadlock situation is depicted in Figure .. Following the recovery of the first loss *cwnd* is restored to the post-fast-retransmit value (half the value before congestion-control). Following loss of the retransmission, the number of outstanding packets remains large. Additional duplicate ACKs increment *cwnd*, but not enough to send a new segment. Subsequently the receiver and sender are *not* triggered to send any more ACK/data segments. This results in the deadlock situation, that is *only* released by a timeout. In the instantaneous goodput of NewReno depicted in Figure .. this deadlock occurs in the 1 second duration starting at 0.3 seconds.

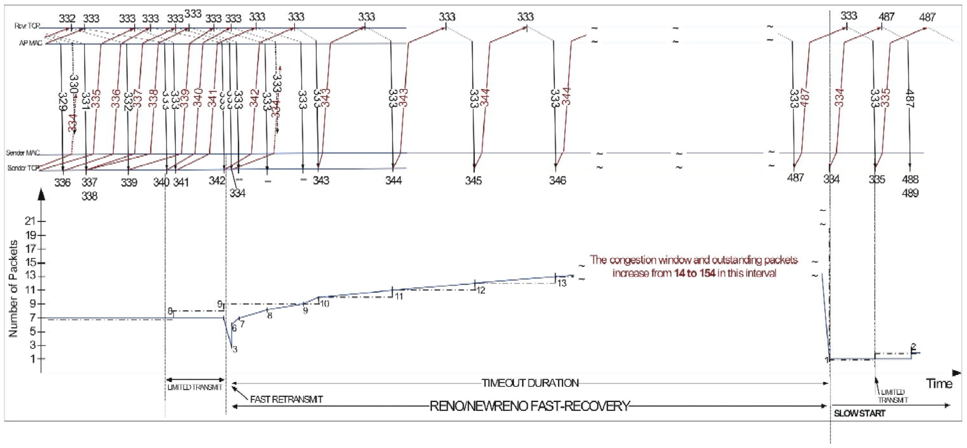
*Fast-recovery* in either of Reno or NewReno does sustain a *non-zero goodput* during the timeout period, *when the first retransmission is lost* - i.e. the segment retransmitted at the start of congestion-control during *fast-retransmit*. Figure .. depicts one such scenario. Duplicate ACKs that arrive indicating the same segment sustain the growth of *cwnd*, and consistently trigger new data segments to be sent. In the given example, the *cwnd* subsequently grows to 154 segments. The likelihood of self-interference diminishes to *zero* here since *only one* segment is in transit at a time. However restricted *cwnd* growth during *fast-recovery* underutilizes the link bandwidth. In the instantaneous goodput depicted in Figure .. Reno undergoes this situation in the 1-second interval beginning at 4.4 seconds and NewReno experiences the same in the intervals beginning at 1.9 and 3.3 seconds.



***Performance of various TCP flavors for different values of minimum retransmission timeout (minrto_) setting***

*Insight III: Tahoe outperforms Reno and NewReno*

The possible behavior of Tahoe following the same multiple losses depicted in Figure .. , is plotted in Figure .. Tahoe does *not implement fast-recovery*. Instead it resumes operation in *slow*-start mode soon after *fast-retransmit* and resetting *cwnd* to 1 segment. By doing so, it forgets everything about packets already outstanding. Several duplicate transmissions of data segments could ensue but Tahoe *does not deadlock if the retransmission during fast-retransmit is successful.*



***Reno/NewReno congestion control when the first retransmission is lost due to self-interference (an ACK is also lost)***

*Compare Tahoe with Reno:* With no deadlocks, Tahoe recovers quickly from multiple losses in a *cwnd* despite duplicate retransmissions after multiple losses.

*Compared to NewReno:* Tahoe reduces the likelihood of self-interference after each loss, by scaling down the *cwnd* to 1 segment. The *cwnd* remains small for the subsequent duration when the remaining losses in the *cwnd* are retransmitted. On the other hand, the likelihood of self-interference remains high in NewReno, while it retransmits subsequent lost packets in the *cwnd*. *Tahoe thus has a higher likelihood of recovery from multiple losses than NewReno in a self-interference environment*

During the time wasted by Reno and NewReno during timeout periods Tahoe achieves a non-zero goodput and hence gains overall in net throughput.

## V. Effect of *minrto_* on TCP Performance

RFC 2988 [6] stipulates the minimum timeout duration (*minrto_*) to be set to 1 second. This was in order to avoid spurious timeouts and retransmissions in TCP in wired nets with large fluctuating round trip times.

However since Reno and NewReno undergo multiple timeouts *not caused by network congestion*, a large timeout duration wastes precious link bandwidth.

More recent implementations of TCP set *minrto_* to 0.2 seconds. Using this value, TCP-Reno's net throughput for a 1MB file transfer nearly doubled. Comparison of instantaneous throughputs and congestion window dynamics for the two different values of *minrto_* is shown in Figures … for *minrto_* set to 0.2 seconds. Reno and NewReno still experience the same number of timeouts but gain in throughput because of the shorter time spent in the deadlock situation before each timeout. Tahoe still outperforms both these flavors although by a lesser margin.

In the extreme case when minrto_ is set to zero, all three versions of TCP achieve similar throughputs in the scenario we have considered where there are single wired and wireless hops with no other active flows in the course of the experiment.

## VI. A Case for Cross-Layer Awareness

Self-interference of TCP in contention-based wireless links is an example of complex interaction among layers in the network stack. TCP fails to independently identify the reason for a packet loss, and invariably scales down the flow rate. It is evident from this work and various other, that the following design characteristics are desired in a transport protocol operating over wireless links:

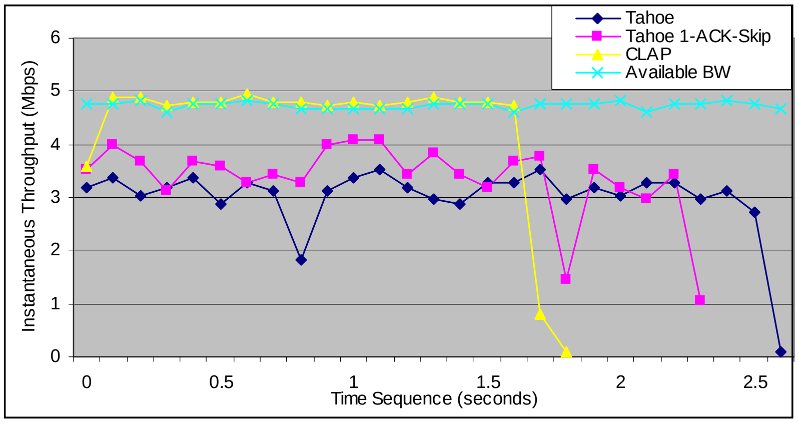
1. ACKs must be minimized to alleviate self-interference.
2. Flow control and error control must be decoupled.

Earlier papers have found that skipping a single ACK achieves considerable gain in the net throughput due to reduction in self-interference [1][2]. They also found that skipping higher ACKs interfered in normal TCP operation due to *cwnd* starvation. The study considered TCP-Reno.

Given our finding that Tahoe performs better than Reno during self-interference, we examined the performance gain in TCP Tahoe with ACK skipping. The result is plotted in Figure .. for *minrto_* = 0 (rto_ is entirely based on estimated rtt_). This is a prudent setting in the considered topology since there are no out-of-order packets and losses due to network congestion. With this setting, little time is wasted in deadlock situations. With 1-ACK-Skip, the net throughput improves from 3 Mbps to 3.36 Mbps achieving a throughput gain of 12%. Tahoe *still* experiences a timeout with 1-ACK-skip. However with lesser self-interference during peak operation, it gains in instantaneous goodput. The Figure also compares the instantaneous goodput of *the CLAP protocol*. CLAP utilizes the full available bandwidth at most operating instants and achieves a net throughput of 4.41 Mbps. The throughput gain is 95% over default Tahoe and 47% over Tahoe with minrto_ = 0. The gains are a *lot higher* when there are link errors and multiple flows [3].

CLAP (Cross Layer Aware transport Protocol) addresses key challenges in wireless networks such as self-interference, time-varying bandwidth, link errors etc. Following are its primary design charactistics:

(1) Decouples error and flow control



***Comparison of CLAP with TCP-Tahoe with/without ACK-skipping for minrto_ = 0***

(2) Uses a *rate-based* flow control approach using link rate information from lower layers.

(3) Uses an *aggregate Negative ACK (NACK)* approach for error control.

For further details of CLAP, and performance in error-prone time-varying wireless links, refer to [3].

## VII. Related Work

TCP enhancements for wireless networks may be clearly categorized into those for cellular networks [Indirect-TCP], [Snoop-TCP], those for contention-based multi-hop wireless networks (majority of the papers) [TCP-BEAD], [TCP-ELFN], [TCP-Westwood], and those for single-hop wireless links [1]. With per-node channel scheduling in cellular networks, link-layer collisions due to self-interference do *not* occur. However self-interference still occurs with ACK packets consuming expensive channel time at the cost of data packets.

All TCP enhancements for wireless networks, are extensions of TCP-Reno or TCP-NewReno. To the best of our knowledge ours is the first attempt to investigate fast-recovery in contention-based wireless links.

Ours is also the first-attempt to depict transport and MAC layer behavior in a single easy-to-read graph.

## 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 various TCP versions that result 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 be easily mitigated because of its tight coupling of error and flow control mechanisms. The window-based flow control is heavily dependent upon 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, wireless nodes are indeed equipped to provide status information that can be used to supplement decision-making processes in the transport protocol. CLAP [3] that decouples error and flow control and utilizes cross-layer status information achieves significant throughput gains over that of TCP in this and other 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, “CLAP: A Cross Layer aware Transport Protocol for Time-Varying Wireless Links”, *submitted to* INFOCOM 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] S. Gopal, K. Ramaswamy, C. Wang, “On Video Multicast over Wireless LANs”, IEEE Conference on Multimedia and Expo (ICME), Taipei, Taiwan, April 2004.

[10] J.Jun, P. Peddabachagari, M. Sichitiu. “Theoretical Maximum Throughput of IEEE 802.11 and its Applications”, Second International Conference on Network Computing and Applications, April 2003.

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