

# Improving TCP Throughput in 802.11 WLANs with High Delay Variability

Georgios I. Fotiadis and Vasilios A. Siris

Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science  
P.O. Box 1385, GR 711 10 Heraklion, Crete, Greece  
{fotiadis,vsiris}@ics.forth.gr

**Abstract**—Reduced throughput of TCP over wireless networks is mainly due to misinterpretation of wireless losses as congestion signals and high delay variability because of fluctuating radio channel quality. High delay variability often leads to spurious TCP timeouts which result in severe throughput degradation. In this paper, we show the negative impact of large delay variations in a mixed wired and 802.11 network scenario with bursty errors, and propose an adaptive algorithm that adds delay to packets in order to prevent spurious timeouts. Our algorithm can adjust to specific network characteristics such as propagation delays, loss rate, and number of mobile nodes, and is applied at the access point of a wireless LAN.

## I. INTRODUCTION

The Transmission Control Protocol (TCP) is the most widely used protocol for reliable data transfer over the Internet. TCP was initially designed for wired networks but the growing popularity of mobile applications has established its deployment in wireless environments as well. The main goal of TCP is to probe the available bandwidth of an end-to-end connection and efficiently adapt the sender's transmission rate so as to prevent network congestion. Packet losses are the most convenient way to detect congestion in wireline networks where transmission errors are very rare and most losses are due to buffer overflow at the intermediate routers. When a loss is detected the sender reduces its transmission rate, so that the overloaded buffers drain. However, network congestion is usually not the reason for packet losses in wireless networks. Errorprone transmissions over the wireless channel and high delay variability are misinterpreted as signals of congestion from the TCP sender that unnecessarily throttles the transmission rate by reducing the congestion window (*cwnd*). Robust local error recovery mechanisms mitigate the problem of wireless losses but impose additional delay to packets being locally retransmitted. These additional delays can have serious impact on the performance of TCP leading to severe throughput reduction.

In this paper we focus on the impact of high delay variability, often referred to as delay spike, on TCP performance. Delay spikes mostly appear in environments with bursty losses, where multiple link layer frames can be corrupted, and which can be attributed to channel fading, user mobility and hand-offs, or finally link-layer retransmissions. Thus, they result in sudden delay increase of specific TCP segments in contrast to the delay experienced by other segments of the same TCP

session. This sudden increase of the delay of a segment, and therefore of its round trip time, can lead to spurious TCP timeouts, i.e., timeouts that could have been avoided if the retransmission timer (*RTO*) value of the sender was larger. Spurious timeouts have a negative impact on TCP performance since the congestion window (*cwnd*) is unnecessarily reduced to one segment. This negative impact becomes even more serious in emerging high bandwidth wireless networks, where the bandwidth delay product (*BDP*) is fairly large and, as a result, the *cwnd* can obtain large values. Moreover, the lossy and high-delay wireless environment makes it difficult for the TCP sender to quickly recover the optimal *cwnd* value.

Our approach focuses on eliminating wireless losses by applying a robust link layer retransmission mechanism. Persistent local retransmissions avoid packet loss even in a bursty loss environment, which as a result solves the problem of false packet loss misinterpretation. At the same time we encounter the negative effects of high delay variability by injecting artificial delay to specific acknowledgement (ACK) packets at the access point so as to prevent potential spurious TCP timeouts.

The remainder of this paper is organized as follows. Related work on TCP performance in lossy wireless environments is summarized in Section II. Our approach is analyzed in detail in Section III while the results of the application of our method in a mixed wired and wireless network is presented in Section IV. Finally concluding remarks and future directions are given in Section V.

## II. RELATED WORK

The effects of high delay variability on some TCP implementations are examined in [1]. The authors recommend timing every TCP segment to provide a more conservative value of the retransmission timer (*RTO*) of the TCP sender and discuss how a series of duplicate acknowledgements should be treated. Moreover, in [2] the authors identify the presence of long sudden delays in a GPRS network and study the impact of spurious TCP timeouts on end-to-end performance. A plurality of methods to improve TCP throughput, in the presence of high delay variability, have been proposed over the recent years. In [3], the authors examined how injecting static or random delay over static or random intervals to specific packets of a TCP session can improve throughput. The performance of

the proposed approach depends on the values of the static parameters, which should be different for different networks characteristics such as propagation delays, bit error rates, and number of mobile nodes. Moreover, changes to the TCP senders are required. The results presented in this work suggest throughput improvements up to 8% in the scenarios examined. In [4], [5] the authors propose the Eifel algorithm which detects spurious timeouts with the aid of the TCP timestamp, and boosts the TCP congestion window if it was unnecessarily throttled because of a spurious timeout. The main disadvantage of this approach are the modifications that are required to the TCP senders. The authors of [6] propose a new mechanism that is applied to the wireless part of a TCP session. The proposed mechanism uses the TCP timestamp to exclude from the TCP sender’s RTO calculation the time spent on wireless link-layer retransmissions; this approach requires changes only to the access point, but adds significant complexity to it. Moreover, a large buffer capacity is required in case of multiuser networks. Finally in [7] the authors propose a *RTO* calculation method and evaluate the adaptation of a go-back-N retransmission policy at the TCP sender; the proposed methods yield a throughput improvement of 13.7 and 12% respectively.

Our method focuses on eliminating spurious timeouts by injecting artificial delay to packets at the access point. The proposed algorithm involves deciding which packets to delay and how much to delay them. Furthermore, the algorithm can adjust to the specific network characteristics, and is implemented solely at the access point. The results suggest significant improvement in bursty loss wireless environments by up to 63%.

### III. IMPROVING TCP PERFORMANCE IN PRESENCE OF BURSTY LOSSES

In this paper we focus on infrastructure-based wireless LANs where wired nodes are connected to the wireless ones through an access point. Data flow from the mobile to the fixed nodes and acknowledgements in the opposite direction. Our method addresses the problem of reduced throughput using two basic ideas: (a) it eliminates wireless losses using local error recovery mechanisms at the link layer, (b) it absorbs high delay variabilities, due to the local retransmissions, by injecting delay to acknowledgement (ACK) packets at the access point.

#### A. Suppressing Wireless Losses

As discussed in the introduction, TCP in wireless environments suffers from the misinterpretation of packet losses at the radio path as congestion signals. We decouple the sender from this dilemma by adapting a robust link layer retransmission mechanism. The link retransmission mechanism of 802.11 protocol is governed by two parameters: the *ShortRetryLimit* and *LongRetryLimit*. The *ShortRetryLimit* defines the number of retransmissions for packets with size less than the *RTSThreshold*<sup>1</sup> value while the *LongRetryLimit*

<sup>1</sup>This parameter governs the RTS/CTS mechanism. The standard defines it equal to 2347.

defines the number of retransmissions for packets larger than *RTSThreshold* bytes. In our experiments we disable the *RTS/CTS* mechanism, by assigning a very large value to *RTSThreshold*, so only the *ShortRetryLimit* parameter makes sense. We will refer to it as *RetryLimit*.

We eliminate wireless losses by assigning a very large value to the *RetryLimit* parameter at the mobile nodes. Note that when a mobile host has a large retransmission limit it won’t affect the performance of other nodes in the network. Moreover, if a flow of a node fails to transmit a packet then no other flow of the same node will be able to transmit. On the other hand, the negative effect of persistent local retransmissions is that the packets being locally retransmitted can experience significantly larger delay than the packets that did not undergo retransmission. In bursty loss environments these delay variabilities are expected to be fairly large, hence they can lead to spurious TCP timeouts. We will refer to this modification of the 802.11 protocol as infinite-*RetryLimit* (inf-*RL*) enhancement.

#### B. Adaptive Algorithm to Absorb High Delay Variability

Our goal is to develop an algorithm that dynamically delays packets, by a value that depends on the network characteristics, in order to absorb the high delay variability caused by the local retransmissions. Indeed, the size of the delay spike reflects the packet’s delay due to local retransmissions. This information, considering the burstiness of errors in the wireless channel, provides the intuition of how much delay we should inject in order to avoid future spurious timeouts.

For the purposes of the algorithm, we define packet delay (*pdelay*) as the interval from the time the mobile TCP sender transmits a packet until the time the corresponding TCP acknowledgement reaches the access point. This interval is calculated from the TCP acknowledgement’s timestamp minus the local time at the access point<sup>2</sup>, and includes propagation delays, queuing delays, and local retransmission delays. Also, let average delay (*avdelay*) be the exponentially smoothed average of *pdelay*. When there are no link-layer retransmissions, i.e. the wireless radio path is in a good state, *avdelay* will include only the propagation and queuing delays. Hence, the difference of *pdelay* minus *avdelay* gives a good estimate of the time spent on local retransmissions, and therefore suggests the delay we need to inject in order to avoid spurious timeouts.

In order to identify delay spikes that can lead to spurious timeouts, we apply a formula similar to the TCP RTO calculation formula, as suggested in [8], on the variable *pdelay* at the access point. So, for the *k* – *th* incoming TCP acknowledgement at the access point, it is:

$$S[k] = (1 - a) * S[k - 1] + a * pdelay[k] \quad (1)$$

$$V[k] = (1 - b) * V[k - 1] + b * (|pdelay[k] - S[k]|) \quad (2)$$

$$dthresh = S[k] + c * V[k] \quad (3)$$

<sup>2</sup>This assumes that clocks are synchronized. If this is not the case, the access point can estimate the time lag, and consider it in the calculation.

This formula calculates the  $dthresh$  variable as the sum of the smoothing average of  $pdelay$  ( $S$ ) plus the its variance ( $V$ ). Typical values for  $a$ ,  $b$  and  $c$  are  $\frac{1}{8}$ ,  $\frac{1}{4}$  and 4 respectively, as suggested in [8]. Delay spike identification is crucial for our algorithm since it indicates when delay should be injected. The  $dthresh$  is then compared to  $pdelay$  for every incoming acknowledgement at the access point in order to detect a delay spike. Once a delay spike is detected, the delay  $D$  to inject is proportional to the size of the delay spike detected, i.e. the difference of  $pdelay$  minus  $avdelay$ ; the delay  $D$  also includes a factor  $delay\_factor$ . The algorithm includes a linear reduction parameter ( $subthresh$ )<sup>3</sup> of the injected delay to avoid needlessly delaying packets when delay spikes are infrequent. A high level description of the proposed algorithm, which runs for every ACK packet received by the access point and travelling towards the mobile nodes, is shown below:

---

**Algorithm 1** Delay Injection

---

```

1: if  $pdelay > dthresh$  then
2:    $D = \frac{(pdelay - avdelay)}{delay\_factor}$ 
3: else
4:   if not delayed then
5:     inject_delay( $D$ )
6:   else
7:     delay by up to  $D$ 
8:   end if
9:    $D = D - subthresh$ 
10: end if
11: update( $dthresh$ )

```

---

The  $inject\_delay()$  function adds delay  $D$  to the first ACK packet after the delay spike is detected. All subsequent ACK packets that arrive within time  $D$  of the delayed ACK are transmitted after it (i.e., no additional delay is added). ACKs that arrive after time  $D$  of the first delayed ACK are again delayed by time  $D$ . The value of  $D$  decreases linearly with each ACK that does not trigger a delay spike detection. Finally the  $update()$  function calculates the  $dthresh$  value according to equations (1), (2) and (3).

#### IV. EXPERIMENTAL EVALUATION

To evaluate the proposed method, we use the Network Simulator 2 (Ns-2) [9] in a wired+wireless topology with an access point infrastructure, as shown in Figure 1. The TCP sender lies in the mobile nodes, while the TCP receiver is at the fixed nodes. The wireless protocol is 802.11 at 11 Mbps. The wired links' capacity is 100 Mbps. The TCP implementation used in the experiments was TCP Newreno [10].

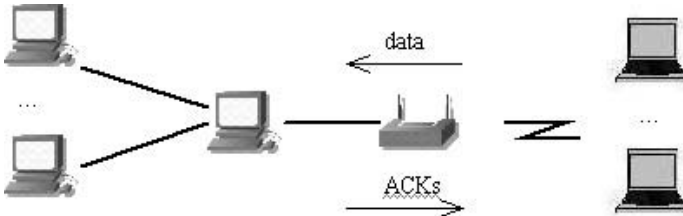


Fig. 1. Network Topology

<sup>3</sup>Detailed explanation of the  $delay\_factor$  and  $subthresh$  parameter selection is included Section IV.

The error model used is a simple loss model that introduces consecutive packet losses, e.g. in a 10% loss scenario 10 out of 100 packets are lost consecutively. Our delay injection algorithm is implemented at the access point and is applied to the TCP acknowledgement ( $ACK$ ) packets travelling from the wired nodes to the mobile ones. In our experiments, we measured the end-to-end throughput in one flow scenario and the end-to-end aggregate throughput in multiple flow scenarios. We compared our method to the standard 802.11 protocol ( $RetryLimit = 4$ ) and the infinite  $RetryLimit$  ( $inf - RL$ ) enhancement to the 802.11 protocol. Recall that our method combines the infinite  $RetryLimit$  enhancement and algorithm 1. We show that injecting delay can improve TCP performance but it introduces the following tradeoff: preventing spurious timeouts against increasing the Round Trip Time (RTT) of the packets. Preventing TCP spurious timeouts is not always beneficial. As we show in some scenarios, the significant increment in RTT of the packets being delayed outweighs the performance gains of spurious timeout avoidance.

#### A. Experiments with Different Propagation Delays

To show that our algorithm is independent of specific networks characteristics, such as the wired propagation delay, we conducted experiments with different propagation delays on the wired links. In Figure 2, we show the TCP throughput as a function of the packet error rate (PER) for one TCP flow, when the wired propagation delay is 15 ms.

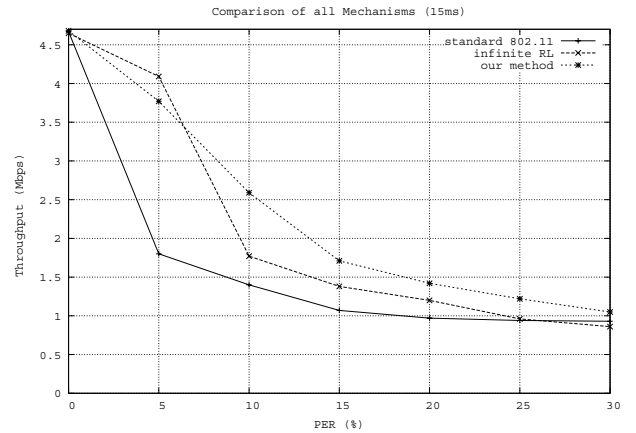


Fig. 2. Results

TCP over standard 802.11 experiences poor throughput due to misinterpretation of packet losses as congestion signals. The infinite  $RetryLimit$  enhancement solves the problem of packet loss misinterpretation, since packets are no longer dropped on the wireless channel, but introduces high delay to the packets being locally retransmitted. High delay variability results in spurious timeouts and therefore severe throughput degradation. The above two problems are addressed by our method, which performs significantly better than the other two methods when  $PER$  is above 10%. For example, our algorithm presents throughput improvements of 46% and 32% in comparison to the standard 802.11 and  $inf-RL$  enhancement respectively, when  $PER$  is 10%. When  $PER$  is 5%,

the improvement of our algorithm is smaller than the *inf-RL* enhancement, though still better than standard 802.11 (improvement of 52%). This is because a low *PER* (5%) does not impose much delay to the locally retransmitted packets so as to generate spurious timeouts. In this case, injecting delay does not have a positive impact on TCP performance.

The wired propagation delay is straightforwardly related to the end-to-end throughput, which is inversely proportional to the RTT of the path. Note, also, that larger propagation delays at the wired part result in a larger bandwidth delay product (*BDP*) and therefore larger values of the congestion window parameter. When the congestion window is large, the impact of a spurious timeout is more significant than in the case of a small congestion window, since in the latter case the window restoration will occur more faster.

In Table I we can see our method in comparison to the standard 802.11 and the infinite *RL* enhancement, when the wired propagation delay is 20 ms, for different *PERs*. Our method improves performance when the *PER* is greater than 15%. A *PER* of 10% is not high enough to produce spurious timeouts, in contrast to the experiment where the wired propagation delay was 15 ms. This happens because the *BDP* is now larger, allowing more packets to be injected to the network and therefore introducing larger inter-packet variances. These variances result in larger values for the sender's *RTO*, hence, there are less timeouts.

TABLE I

IMPROVEMENTS OF PROPOSED DELAY INJECTION ALGORITHM FOR 20 MS WIRED PROPAGATION DELAY

	<i>PER</i>					
	5%	10%	15%	20%	25%	30%
vs st. 802.11	+63%	+48%	+47%	+43%	+38%	+23%
vs <i>inf-RL</i>	-9%	-2%	+29%	+23%	+23%	+19%

As illustrated in Table I, in the presence of spurious timeouts ( $PER \geq 15\%$ ) the improvement of our method is higher compared to the experiment with a wired propagation delay of 15 ms. Recall that when the propagation delay is larger, *cwnd* attains greater values. Hence, the negative impact of spurious timeouts are more crucial and, therefore, avoiding them is more performance critical.

Finally, in Table II we show the results of the same experiment with a smaller wired propagation delay (10 ms).

TABLE II

IMPROVEMENTS OF PROPOSED DELAY INJECTION ALGORITHM FOR 10 MS WIRED PROPAGATION DELAY

	<i>PER</i>					
	5%	10%	15%	20%	25%	30%
vs st. 802.11	+35%	+15%	+18%	+3%	-18%	-36%
vs <i>inf-RL</i>	-5%	+18%	+9%	-2%	-	-10%

In this case our method performs better than both the other two only when *PER* is 10%, 15% and 20%. When *PER* is lower than 10% then, as in the former experiments, no spurious timeouts take place at the TCP sender. So, our algorithm does

not outperform the *inf-RL* enhancement. On the other hand, when *PER* is greater than 25%, standard 802.11 standard outperforms the other two approaches. This is because the small propagation delay bounds the *cwnd* to small values and moreover the low end-to-end delay helps the sender to quickly recover the optimal *cwnd* value. As a result, it is more beneficial to have dropped packets at the wireless part than have them locally retransmitted and then delayed in order to eliminate high delay variabilities.

### B. Multiple User Experiments

Next, we present the experiments that involve multiple mobile nodes. Recall that in a wireless network, where multiple flows from different users are multiplexed, the TCP congestion window of each user is expected to be smaller than in the case of a single user. Also, packet multiplexing in queues results in significant per packet delay variance, which leads to more conservative *RTO* values. Thus, in a multiuser wireless environment spurious timeouts are less frequent.

In Figure 3 we present the results with two participating mobile users. The propagation delay of the wired links is 15 ms. As in the single user experiments, our method outperforms the other two by a significant factor for *PER* values greater than 10%. Indeed, the achieved performance is larger than in a single flow experiment. This can be attributed to the fact that in the case of two flows: (i) spurious timeouts are still plenty and (ii) the *cwnd* restoration phase is very slow (slower than in the single user case) since a user that reduces its transmission rate finds it hard to recover because the links are occupied by the packets of the other flow.

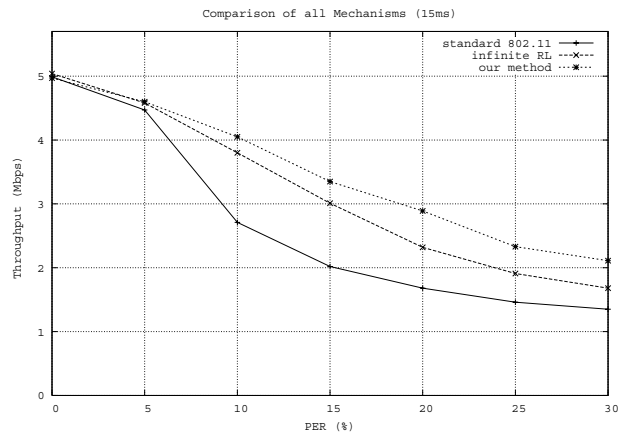


Fig. 3. Results with 2 Mobile Users

Finally, Figure 4 shows the aggregate throughput of 5 mobile users, i.e., 5 different TCP flows. As stated earlier, multiuser network topologies present significant packet delay variance. Due to this, the proposed scheme does not improve performance compared to the *inf-RL* enhancement. Packet multiplexing from the 5 different flows increases delay variance significantly, resulting in large values for the senders' *RTOs*. Hence, spurious timeouts are less likely to happen. Nevertheless, note that the proposed scheme still achieved improved performance compared to standard 802.11.

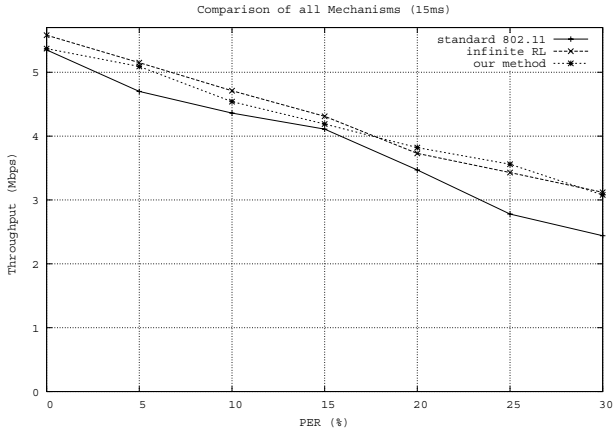


Fig. 4. Results with 5 Mobile Users

### C. Algorithm Parameter Selection

In this section, we discuss the appropriate selection of the two parameters of Algorithm 1 (*delay\_factor* and *subthres*).

1) *Delay Factor (delay\_factor)*: The *delay\_factor* parameter reflects how much delay is to be injected in proportion to the size of the delay spike. The effect of *delay\_factor* on performance is shown in Figure 5.

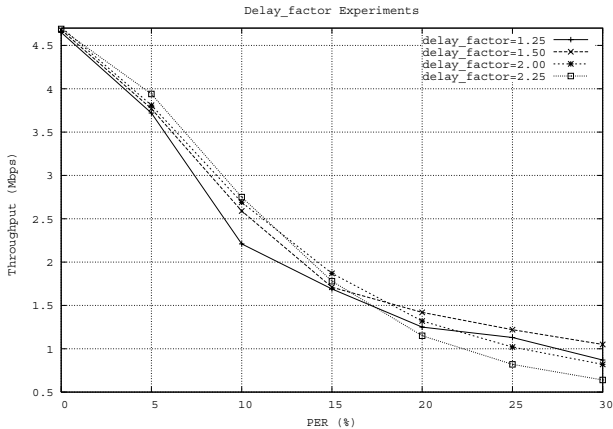


Fig. 5. Results with 2 Mobile Users

A smaller *delay\_factor* results in a larger delay being injected, hence decreases the likelihood of delay spikes. A larger *delay\_factor* value results in more significant throughput improvement in low *PER* scenarios, e.g., when *delay\_factor* = 2.25 and *PER* = 10%, but this is not the case when *PER* is higher. In order to guarantee the algorithm's proper overall operation we can choose a small value (1.5) in order to ensure that less spurious timeouts will occur.

2) *Subtraction Threshold (subthres)*: In contrast to *delay\_factor*, *subthres* does not directly affect TCP's performance, and its usage is to mitigate the negative impact of delay injection in scarce loss environments. Suppose that in the wireless environment a burst of corrupted packet occurs; our algorithm will detect the delay spike and would inject delay according to Algorithm 1. Without the presence of the *subthres* parameter the access point would continue delaying ACKs despite the fact that there is no other delay

variability. In this case the injected delay would be unnecessary and would result in throughput degradation. The *subthres* is selected as follows: the access point counts the number of packets that are interjected between two delay spikes (*pkt\_count*); *subthres* is then assigned the value  $\frac{D}{5 * pkt\_count}$ . With such a value, if in an interval containing  $5 * pkt\_count$  packets no delay spikes occur, then the value of the injected delay is reduced to zero.

## V. CONCLUSIONS AND FUTURE WORK

In this work, we presented an adaptive algorithm for delay injection at the access point, in order to avoid TCP spurious timeouts. The algorithm adjusts to specific network characteristics and performs significantly better than a standard 802.11 wireless network. Moreover no changes at the end systems are required, in contrast to other methods proposed in the bibliography.

Further extensions of the present work would be to take into consideration downlink traffic as well. Applying an infinite *RetryLimit* at the access point is not appropriate since a continuously transmitting flow would prevent packets of other flows from being transmitted. An alternative is to use a round robin queueing mechanism where each destination mobile node would have its queue at the access point. An unsuccessfully transmitted packet would not be dropped but would be put in the head of its corresponding queue. Another interesting research direction is the application of the proposed algorithm to a multihop wireless network.

## REFERENCES

- [1] A. Gurtov, "Making TCP Robust Against Delay Spikes," *University of Helsinki, Department of Computer Science, Technical Report C-2001-53*, November 2001.
- [2] —, "Effect of Delays on TCP Performance," in *Proceedings of IFIP Personal Wireless Communications*, August 2001.
- [3] T. Klein, K. Leung, R. Parkinson, and L. Samuel, "Avoiding Spurious TCP Timeouts in Wireless Networks by Delay Injection," in *Proceedings of IEEE GLOBECOM*, November 2004.
- [4] R. Ludwig and R. Katz, "The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions," in *ACM Computer Communication Review*, vol. 30, No 1, January 2000.
- [5] A. Gurtov and R. Ludwig, "Responding to Spurious Timeouts in TCP," in *Proceedings of IEEE INFOCOM*, March 2003.
- [6] K. Ratnam and I. Matta, "WTCP: An Efficient Mechanism for Improving TCP Performance over Wireless Links," in *Proceedings of 3rd IEEE Symposium on Computers and Communications (ISCC '98)*, Athens, Greece, 1998.
- [7] K. Leung, T. Klein, C. Mooney, and M. Haner, "Methods to Improve TCP Throughput in Wireless Networks With High Delay Variability," in *Proceedings of IEEE Veh. Tech. Conf.*, 2004.
- [8] V. Paxson and M. Allman, "Computing TCP's Retransmission Timer," *RFC2988*, 2000.
- [9] Network Simulator 2, available at <http://www.isi.edu/nsnam/ns/>.
- [10] S. Floyd and T. Henderson, "The NewReno Modification to TCP's Fast Recovery Algorithm," *RFC 2582*, April 1999.