The TCP *Minimum* RTO Revisited

Ioannis Psaras and Vassilis Tsaoussidis

Dept. of Electrical and Computer Engineering
Demokritos University of Thrace, Xanthi, Greece
{ipsaras,vtsaousi}@ee.duth.gr
http://comnet.ee.duth.gr

Abstract. *We re-examine the two reasons for the conservative 1-second Minimum TCP-RTO to protect against spurious timeouts: i) the OS clock granularity and ii) the Delayed ACKs. We find that reason (i) is canceled in modern OSs; we carefully design a mechanism to deal with reason (ii). Simulation results show that in next generation's high-speed, wireless-access networks, TCP-RTO should not be limited by a fixed, conservative lower bound.*

Key words: TCP, Minimum RTO, High Speed Links, Last Mile Wireless

1 Introduction

The Retransmission Timeout policy of standard TCP is governed by the rules defined in RFC 2988 [11]. The TCP-RTO is calculated upon each ACK arrival after smoothing out the measured samples, and weighting the recent RTT-variation history:

$$RTO = SRTT + 4 \times RTTVAR, \tag{1}$$

where $RTTVAR$ holds the RTT variation and $SRTT$ the smoothed RTT. The same RFC also specifies that the TCP-RTO *should not* be smaller than 1 second [11]. This value is known as the *Minimum RTO* and constitutes the subject of interest in the present paper.

Currently, no official instruction exists to address the setting of the Minimum RTO value for TCP. Allman and Paxson in [1] investigated the impact of the Minimum RTO and found that TCP results in lower Throughput performance for Minimum RTO values smaller than 1 second. There were two main limitations that required a (conservative) lower bound for the TCP-RTO to protect it against spurious expirations:

1. the Clock Granularity (500ms for most OSs *at that time*): if the RTT equals the clock granularity, then the timeout may falsely expire before the ACK's arrival at the server.
2. the Delayed Acknowledgments (usually set to 200 ms) [3]: in case an ACK is delayed for more than the current TCP-RTO value, the timer will spuriously expire.

We study each of the above limitations in turn and show that, in fact, there is a lot of space for improvement in the Minimum RTO setting to improve TCP performance. In Section 2, we provide details regarding the clock granularity of modern OSs and find that it is far below the 500ms threshold assumed in [11]. We define a *Cost Function* and show (experimentally) the impact of the Minimum RTO setting on TCP's performance. We conclude that the timer granularity does not constitute a limitation for setting the Minimum RTO, anymore. In Section 3, we investigate the limitation of the TCP Delayed ACK mechanism on the Minimum RTO. We propose a mechanism that makes the TCP server aware of whether the next ACK to be received will, possibly, be delayed or not. Based on that, we assign a Minimum RTO value to each outgoing packet: a longer Minimum RTO to packets whose ACKs may be delayed and no Minimum RTO, otherwise. We present our performance evaluation plan in Section 4. In Section 5.1, we claim that due to limited research studies on the subject of the Minimum RTO, several OSs implement different values for the lower bound of the TCP-RTO, leading to communication inconsistencies. In Sections 5.2 and 5.3, we investigate the impact of a Minimum RTO value on short, web flows and on long FTP flows, respectively; using simulations, we show that the proposed mechanism significantly improves TCP performance, especially in case of wireless losses. We conclude the paper in Section 6.

2 Clock Granularity

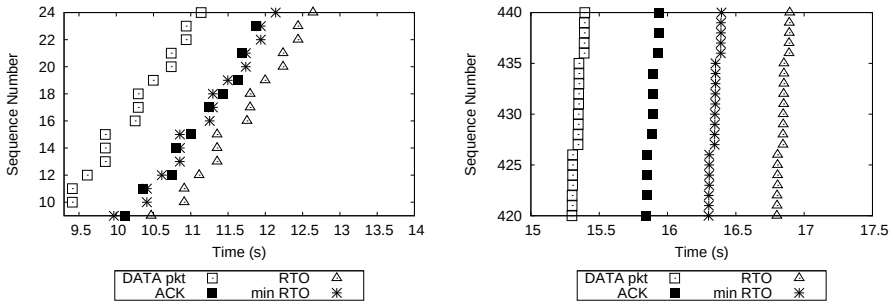
We define a *Cost Function* (Equation 2) to capture the extra time a sender has to wait before retransmitting, due to the conservative Minimum RTO value.

$$C(f) = \frac{RTO_{min}}{RTO_{current}} \tag{2}$$

If $C(f) < 1$, then the Minimum RTO value adds no extra waiting time, in case of packet loss, since the TCP-RTO value is larger than the Minimum RTO. Otherwise, the Minimum RTO value will negatively impact TCP throughput, by forcing the TCP sender to wait for the Minimum RTO timer to expire, prior to retransmitting.

We set (both the client's and the server's) clock granularity to 500ms and simulate one flow over a 500ms round-trip propagation delay path, to observe: i) the rationale behind the conservative 1-second Minimum RTO setting [11], [1] and ii) the impact of the Minimum RTO value relatively with the actual TCP-RTO value. We find (see Fig. 1(a)) that: i) the TCP-RTO algorithm adjusts to values higher than 1 second, hence, $C(f) < 1$ and ii) the Minimum RTO value is only needed as a security setting against spurious retransmissions (i.e., in case the round-trip propagation delay *or* the client's clock granularity equals the server's clock granularity *and* at the same time, the TCP-RTO adjusts to a smaller value, the sender will spuriously timeout).

We reduce the round-trip propagation delay to 6ms and repeat the previous experiment (see Fig. 1(b)). Again, we observe that $C(f) < 1$. We conclude that in case of coarse-grained clocks the Minimum RTO does not have negative



(a) Granularity = 500ms, Round-Trip Propagation Delay = 500ms

(b) Granularity = 500ms, Round-Trip Propagation Delay = 6ms

Fig. 1. 500ms Clock Granularity

impact on TCP Throughput, since the TCP-RTO adjusts to values higher than the Minimum RTO. The Minimum RTO, instead, is only needed as a security setting against spurious timeouts.

In Table 1 we present details regarding some of the most popular OSs, currently; we observe that the clock granularity is always set to a value below (or equal to) 25ms. We repeat the above experiment using, this time, a finer-grained clock of 10ms.

Table 1. Details on Modern OSs

OS	Clock Granularity	Delayed ACK
Windows	15-16ms	200ms
Solaris	10ms	50-100ms
Linux	$\leq$ 25ms	Dynamically Set

Figure 2 uncovers the significant difference between the TCP-RTO values and the Minimum RTO limitation. In contradiction to coarser-grained clocks, simulated previously, we observe that $C(f)$ is now far above 1, obviously leading to severe performance degradation, in case of packet losses.

For the sake of simplicity, we assume the time interval between the ACK arrival and the RTO value, in Fig. 2, to be negligible and we modify the *Cost Function* (Equation 2) accordingly:

$$C(f) \approx \frac{RTO_{min}}{T(ACK\ Arr)} \leq \frac{RTO_{min}}{RCG + RTPD + QD}, \tag{3}$$

where $T(ACK\ Arr)$ holds the *ACK Arrival Time*, RCG the *Receiver's Clock Granularity*, $RTPD$ the *Round-Trip Propagation Delay* and QD the *Queuing Delay*. Since we simulate only one flow, we also consider the Queuing Delay to be insignificant. Hence, from Equation 3 we derive that $C(f) \approx 62.5$. Of course, the cost of extra waiting time due to a high Minimum RTO value will decrease as the *Round-Trip Propagation* and *Queuing Delay* increase. We conclude that:

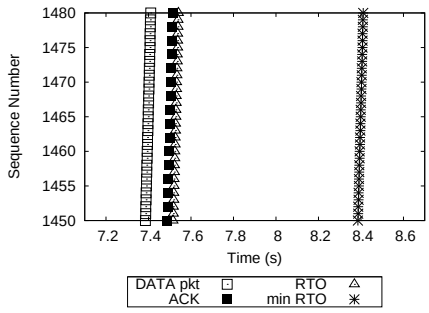


Fig. 2. Granularity = 10ms, Round-Trip Propagation Delay = 6ms

i) the clock granularity should not be a matter of concern for the setting of the Minimum RTO, and ii) the conservative 1-second Minimum RTO will have major impact on TCP's performance, in case of packet losses.

3 Dealing with Delayed ACKs

The Delayed ACK mechanism [3] is quite popular among the vast majority of OSs, currently. According to that mechanism, the TCP client will delay sending an ACK for an incoming packet, for as long as the Delayed ACK timer suggests (see Table 1), unless another packet needs to be sent on that connection (piggypacking). In other words, if a stream of packets arrive at the TCP client, the latter will generate *one* ACK for every *other* packet. Otherwise, if one packet arrives at the TCP client, without being followed by any subsequent packet, then an ACK will be generated only after the Delayed ACK timer expiration. The Minimum RTO will prevent spurious RTO expiration in the latter case.

We propose a mechanism to identify the packets whose ACKs are (possibly) going to be delayed[1]; the Minimum RTO is extended accordingly, for those packets *only*, to prevent spurious TCP-RTO expirations. Our mechanism is based on the following observations:

- TCP's *Sending Window Management and ACK Processing* [2] specifies that the TCP server should send `D` *back-to-back* packets, upon each new-ACK arrival (*ACK-clocking*), according to Equation 4:

$$\texttt{D = snd.una + min(cwnd, rwnd) - snd.nxt}, \tag{4}$$

  where `snd.una` holds the oldest unacknowledged sequence number, `cwnd` and `rwnd` the congestion and advertised window, respectively and `snd.nxt` the next sequence number to be sent.
- At the time when `D` back-to-back packets are generated, TCP does *not* know if the application has more data to send, and even if it does have, we do *not* know after how long.

[1] We leave interactive applications as a subject of future work.

- Since the `D` packets "travel" back-to-back, only the ACK of the last packet of the "train" of packets *may* be delayed, *iff* the server's application stops generating new data.
- Every 2^{nd} packet will always be ACKed.

Consider that at time t_0 all previously transmitted packets are already ACKed and `D` $= 4$ (or, generally, `D` is even). The TCP client will sent ACKs for the 2^{nd} and 4^{th} packets. In this case, the client will not delay ACKing any packets and consequently, there is no need for an extended Minimum RTO. Hence, we apply no Minimum RTO and leave the TCP-RTO deal with the outgoing packets' timeout value. Now, consider that at time t_0, `D` $= 3$ (or, generally, `D` is odd). The TCP client will immediately ACK the 2^{nd} packet and will trigger the Delayed ACK timer for the 3^{rd} packet. If the server's application does not generate any other packet (within the Delayed ACK's timer interval minus the one-way propagation delay), then the 3^{rd} packet will experience delayed ACK response. In this case, we need to extend the Minimum RTO, for the 3^{rd} packet *only*, to prevent spurious timeout expiration.

We extend the above considerations to cover all possible back-to-back sending patterns; we use one variable, which we call `set_odd` and is initially set to false. The proposed mechanism operates in one of the following States:

- State 1: "noMINRTO". Do *not* apply extended Minimum RTO to any outgoing packet (i.e., the receiver will always ACK the last packet of the back-to-back train of packets); set `set_odd` to false.
- State 2: "extended MINRTO". Apply extended Minimum RTO to the last packet of the next train of back-to-back packets; set `set_odd` to true.

According to the following steps, the proposed mechanism applies an extended Minimum RTO value *only if needed* (State 2). Otherwise, the TCP-RTO algorithm deals with the timeout value (State 1). The flow-diagram of the proposed mechanism is presented in Fig. 3.

- Step 1: Extend the Minimum RTO for the first packet sent in the Slow-Start phase and proceed to step 2 or 3, depending on the value of `D`.
- Step 2: If (and for as long as) `D` is even *and* `set_odd` is false, remain in State 1.
- Step 3: Once `D` becomes odd, go to State 2.
- Step 4: If (and for as long as) `D` is even *and* `set_odd` is true, remain in State 2.
- Step 5: When `D` becomes odd *again*, go to State 1 (i.e., the sum of two odd numbers is always even and hence, the ACK for the last packet of the next train will not be delayed).
- Step 6: Proceed to step 2, if `D` is even, or to step 3, otherwise.

Summarizing, the Minimum RTO is set according to the following equation:

$$RTO_{min} = \begin{cases} R\ ms, & \text{for the last pkt } \textit{if } \texttt{set\_odd} = 1, \\ RTO_{cur}, & \text{otherwise,} \end{cases}$$

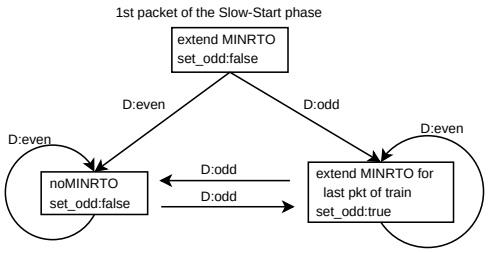


Fig. 3. State Diagram of the Proposed Algorithm

where R is a fixed, extended value for the Minimum RTO. We discuss the setting of this value in Section 5.1.

We present part of the above process in Fig. 4. Initially (i.e., until packet 1478) `set_odd` is false and `D` = 2, in which case there's no need for an extended Minimum RTO (State 1). Next, `D` = 3 and hence the proposed mechanism extends the Minimum RTO of the 3^{rd} packet and sets `set_odd` to true (State 2). From that point onwands, since `set_odd` is true and `D` is *not odd*, the proposed mechanism will extend the Minimum RTO of the last (i.e., 2^{nd}) packet of the back-to-back train of packets (State 2).

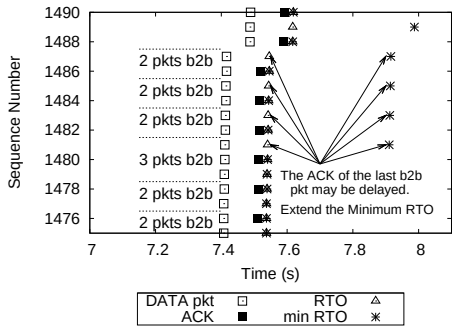


Fig. 4. Modeling ACKs Arrival

We note that the proposed mechanism does not apply for packets sent during Fast Retransmit (FR). During FR the Minimum RTO is set to $R\ ms$; the mechanism resumes from Step 6 after FR or timeout expiration.

According to the above, we re-write Equation 2, for the proposed mechanism as follows:

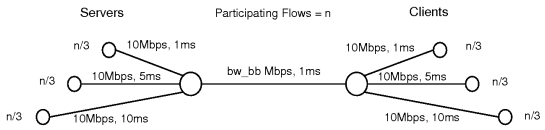
$$C(f) = \begin{cases} \frac{R\ ms}{RTO_{cur}}, & \text{for the last pkt } \textit{if } \texttt{set\_odd} = 1, \\ 1, & \text{otherwise.} \end{cases}$$

Obviously, the cost of extra waiting time, due to the conservative Minimum RTO setting, is now significantly decreased; at the same time, the risk of running

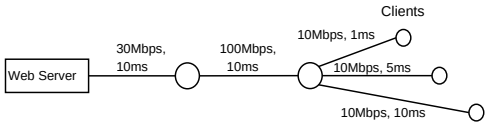
into spurious timeouts, due to delayed ACK response from the TCP client, is effectively avoided.

4 Performance Evaluation Plan

We evaluate the performance of the proposed mechanism using ns-2 [10]. We use realistic protocol settings to reflect the behavior of Internet servers [9], [8], [14]. That is, most OSs use the SACK [7] version of TCP with the timestamps option enabled [6] and the response against spurious timeouts [5], [13] in place. We set the Delayed ACK timer to 200ms and the clock granularity to 10ms; we compare the proposed mechanism with three different Minimum RTO implementations: i) 200ms implemented in Linux TCP, ii) 400ms implemented in Solaris TCP and iii) 1 second as proposed by IETF (and probably implemented in Windows TCP). We use the network topologies shown in Fig. 5, where all buffers use the RED [4] queuing policy. The buffer sizes are set according to the $Delay - Bandwidth\ Product$ of the outgoing links.



(a) Topology 1



(b) Topology 2

Fig. 5. Simulation Topologies

We use two traditional performance metrics:

1. the Average Task Completion Time (ATCT) in case of short, web-applications, and
2. the System Goodput, in case of FTP applications:

$$Goodput = \frac{Original\_Data}{Connection\_time}, \tag{5}$$

where $Original\_Data$ is the number of Bytes delivered to the high-level protocol at the receiver (i.e., excluding the retransmitted packets and the TCP header overhead) and $Connection\_time$ is the amount of time required for the data delivery.

5 Results

We divide the Results Section in three main subsections. Initially (Section 5.1), we show that due to limited standardization efforts on the subject of the Minimum RTO setting, communication problems may arise when the communicating ends are supported by different OSs. Next, we present the impact of the Minimum RTO setting on: i) short, web-like flows (Section 5.2) and ii) long FTP flows (Section 5.3). We emphasize on next generation, broadband wireless access networks, where flow-contention is low and losses occur mainly due to wireless errors.

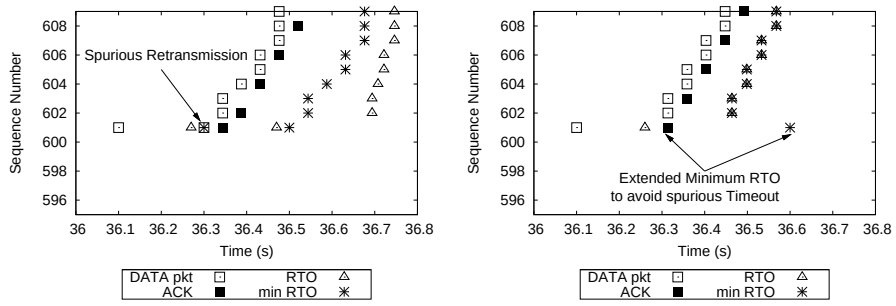
5.1 The Need for a Standard Mechanism

We have already shown that there exist different implementation settings for both the Delayed ACK timer and the Minimum RTO value among different OSs. We report, however, that in case Equations 6 and 7 hold, then the sender will run into spurious timeout expirations *every time* the receiver delays the ACK response.

$$Server's\ Minimum\ RTO < RTPD\ + QD + \ Client's\ DelACK\ Timer \quad (6)$$

$$Minimum\ RTO > RTO_{cur} \quad (7)$$

We verify the above statement experimentally. We simulate a Linux server ($Minimum\ RTO\ =\ 200ms$) and a Windows client ($Delayed\ ACK\ Timer\ =\ 200ms$), over a 42ms Round Trip Propagation Delay path (see Fig. 5(b)).



(a) Linux Server - 200ms Delayed ACK Client (e.g., Windows client)

(b) Modified Linux Server - 200ms Delayed ACK Client (e.g., Windows client)

Fig. 6. The Need for a Standard Mechanism

Indeed, we see in Fig. 6(a) that the Linux server *spuriously* times-out and retransmits packet 601 (i.e., the ACK arrives 42ms later).

On the contrary, the proposed mechanism extends the Minimum RTO long enough to avoid spurious retransmissions (see Fig. 6(b)). In the present work,

whenever deemed necessary, according to the proposed mechanism, we apply $Minimum\ RTO = R = 500ms$. That is, the proposed mechanism will effectively deal with situations where $RTPD + QD \leq 300ms$ (see Equation 6), since we have not found any implementation, where the Delayed ACK interval is greater than 200ms.

5.2 Impact on Short Web Flows

We use the topology shown in Fig. 5(b), where 3 flows, download a content-rich web-page (i.e., 100KBs) every 5 seconds; end-users are connected through wireless, lossy links to router R_2 ($PER = 3\%$). In Table 2, we present the Average Task Completion Time[2] (ATCT) for the Linux TCP implementation, comparatively with the proposed mechanism, after 20 successfully completed tasks.

Table 2. Average Task Completion Time (ATCT)

	flow 1	flow 2	flow 3
Linux	2.4s	2.5s	2.7s
noMINRTO	2.1s	2.4s	2.55s
Difference	$\sim 13.2\%$	$\sim 4.22\%$	$\sim 6\%$

Since the propagation and transmission delay is the same in both cases, we subtract them in order to capture the delay difference solely due to the proposed algorithm; we find that the 200ms Minimum RTO value, implemented in Linux TCP, increases the ATCT by 8% in average. We note that the difference in the ATCT is further increased in case of higher Minimum RTO values (e.g., Solaris, IETF), as well as in case of faster transmission links.

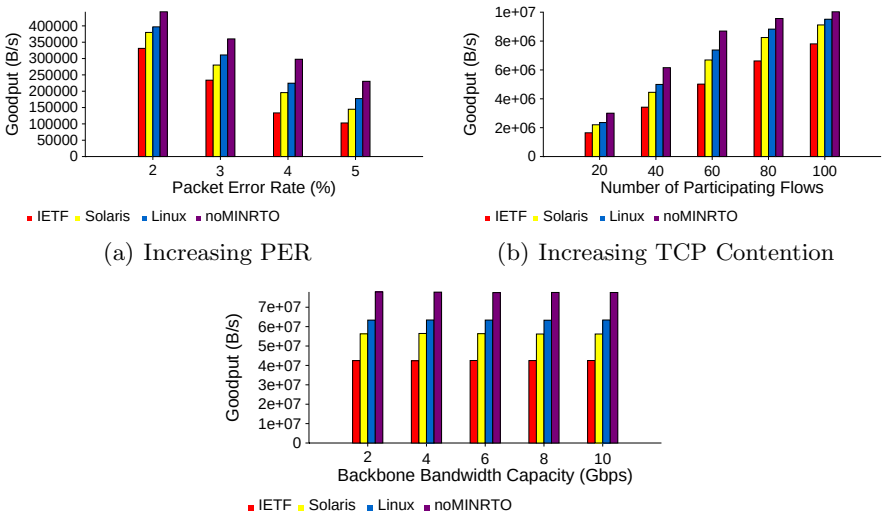
5.3 Impact on Long FTP Flows

We present three different evaluation scenarios considering three network parameters: i) the PER, ii) the Number of Participating flows and iii) the Bandwidth Capacity of the Backbone link. In all cases, we use the topology shown in Fig. 5(a); the simulation setup for each experiment is shown in Table 3, while the corresponding results are presented in Figs. 7(a), 7(b) and 7(c).

Table 3. Experiment Details

	PER	TCP Flows	*bw_bb*
Fig. 7(a)	see Fig.	3	6 Mbps
Fig. 7(b)	3%	see Fig.	100 Mbps
Fig. 7(c)	3%	500	see Fig.

[2] Each Task is defined as a complete transfer of a web page.



(a) Increasing PER

(b) Increasing TCP Contention

(c) Increasing Backbone Bandwidth Capacity

Fig. 7. Impact on Long FTP Flows

In all three cases, we observe that the proposed mechanism provides significant performance increase against the Minimum RTO settings implemented in Linux TCP, Solaris TCP and the IETF proposal (Windows TCP). When 4% of the transmitted packets are corrupted due to wireless errors (Fig. 7(a)), for example, the proposed mechanism improves TCP Goodput performance by approximately 33% over the Linux/Solaris TCP implementation, while the increase becomes even larger (i.e., 50%) against the IETF setting. Faster transmission links provide further advantage to the proposed algorithm (Fig. 7(c)); both the RTT and the Queuing Delay decrease, making the cost of extra waiting time an even more dominant factor, performance-wise (see Equations 2 and 3). As TCP contention increases (Fig. 7(b)) [12], however, the performance difference decreases, since in that case the queuing delay, reflected in the TCP-RTO value, minifies the impact of extra waiting time due to the Minimum RTO setting (see Equations 2 and 3).

6 Conclusion

We have shown that the conservative 1-second Minimum RTO setting causes severe TCP performance degradation, especially in case of last mile wireless users connected to high-speed backbone links. We argue that such a (security) setting, to protect against spurious TCP timeouts, is not needed, since: i) modern OSs use fine-grained clocks and ii) the Delayed ACK response can be dealt with, using the proposed mechanism. Simulation results show that under conditions (i.e., high-speed backbone links, wireless errors at the last mile wireless link), TCP may

achieve up to 50% higher Goodput performance, when the proposed mechanism is used; at the same time, spurious timeout expirations, due to delayed ACK response from the TCP client, are effectively avoided.

References

1. M. Allman and V. Paxson. On Estimating End-to-End Network Path Properties. In *Proceedings of ACM SIGCOMM*, pages 263–274, September 1999.
2. M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control, RFC 2581, April 1999.
3. R. Braden. Requirements for internet hosts - communication layers, October 1989.
4. S. Floyd and V. Jacobson. Random Early Detection gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.
5. A. Gurtov and R. Ludwig. Responding to Spurious Timeouts in TCP. In *Proceedings of IEEE INFOCOM*, 2003.
6. V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance, RFC 1323, May 1993.
7. M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options, RFC 2018, April 1996.
8. Alberto Medina, Mark Allman, and Sally Floyd. Measuring interactions between transport protocols and middleboxes. In *Proceedings of IMC '04*, pages 336–341, New York, NY, USA, 2004. ACM Press.
9. Alberto Medina, Mark Allman, and Sally Floyd. Measuring the evolution of transport protocols in the internet. *SIGCOMM CCR*, 35(2):37–52, 2005.
10. ns 2. The Network Simulator - ns - 2, http://www.isi.edu/nsnam/ns/.
11. V. Paxson and M. Allman. Computing TCP's Retransmission Timer, RFC 2988, May 2000.
12. Ioannis Psaras and Vassilis Tsaoussidis. Why TCP Timers (still) Don't Work Well. *Elsevier Computer Networks Journal, COMNET, to appear 2007*.
13. P. Sarolahti, M. Kojo, and K. Raatikainen. F-RTO: An Enhanced Recovery Algorithm for TCP Retransmission Timeouts. In *Proceedings of ACM SIGCOMM*, September 2003.
14. P. Sarolahti and A. Kuznetsov. Congestion control in linux TCP. In *Proceedings of USENIX'02*.