# On the Properties of an Adaptive TCP *Minimum* RTO

Ioannis Psaras *, Vassilis Tsaoussidis

*Democritus University of Thrace*

*Dept. of Electrical and Computer Engineering*

*12 Vas. Sofias Str., 67100 Xanthi, Greece*

**Abstract**

The *Internet Engineering Task Force* (IETF) specified in RFC 2988 the *Minimum TCP-RTO* and recommended that the TCP Retransmission Timeout (RTO) should not be smaller than 1 second. According to RFC 2988, there are two main limitations that call for a lower bound to protect TCP from spurious timeouts: i) the OS clock granularity (500$ms$ for most OSs at the time of RFC 2988 publication) and ii) the Delayed Acknowledgments (usually set to 200$ms$).

We evaluate the correctness of the suggested policy and investigate the impact of the conservative 1-second Minimum TCP-RTO under modern networking conditions. We define a *Cost Function* to capture the impact of the extra waiting time, due to the conservative IETF specification. Our experimental analysis reveals that the OS clock granularity should not be a matter of concern for modern OSs; we carefully design a mechanism to deal with Delayed ACKs. We call the proposed mechanism *Adaptive MINRTO* (AdMINRTO) to reflect its operational properties. In particular, the mechanism identifies the packets whose ACKs are (possibly) going to be delayed and applies extended Minimum RTO to these packets only, in order

to avoid spurious timeout expirations; otherwise, the Minimum RTO is adjusted to smaller values in order to avoid extensive idle periods.

We show, through simulations, that the Adaptive MINRTO improves TCP performance significantly, especially in case of next generation's high-speed, wireless-access networks. The impact of the proposed mechanism, performance-wise, depends on several network conditions. For example, the impact of the Adaptive MINRTO increases i) with the Packet Error Rate (PER), ii) when the path Round Trip Time is short and iii) for fast transmission links (i.e., high-speeds).

*Key words:* TCP, Congestion Control, Retransmission Timeout, Flow Contention, Packet Scheduling

## 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.

* Corresponding Author.
*Email addresses:* ipsaras@ee.duth.gr (Ioannis Psaras), vtsaousi@ee.duth.gr (Vassilis Tsaoussidis).

Currently, there exists no explicit official instruction to address the setting of the Minimum RTO value for TCP. Based on the analysis in [1], RFC 2988 [11] concludes that the TCP Retransmission Timeout should not be smaller than 1 second. 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. According to RFC 2988 [11], there are two main limitations that call for a lower bound to protect TCP from spurious timeouts:

(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 $200ms$) [3]: in case an ACK is delayed for more than the current TCP-RTO, 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* to capture 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, in modern OSs. In Section 3, we investigate the limitation of the TCP Delayed ACK mechanism on the Minimum TCP-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. The proposed algorithm is called *Adaptive MINRTO* (AdMINRTO). We present our performance evaluation plan in Section 4. Section 5 includes our extensive performance evaluation, which is divided in three main parts. More precisely, in Section 5.1, we observe that several OSs implement different values for the lower bound of the TCP-RTO, leading to communication inconsistencies. In Section 5.2, we simulate last-mile wireless users, where losses happen due to fading channels; we present simulation results for web flows (Section 5.2.1), short FTP flows (e.g., small file transfers, Section 5.2.2) and long FTP flows (Section 5.2.3). Finally, in Section 5.3, we investigate the impact of the Minimum RTO value on Goodput, when losses happen due to buffer overflow (i.e., congestion losses). Simulation results reveal that the proposed algorithm improves significantly TCP performance especially in case of wireless losses. In Section 6 we discuss deployability issues; we conclude the paper in Section 7.

## 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 performance, by forcing the TCP sender to wait the Minimum RTO timer expiration, before 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 (Fig. 6(b)), 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) 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).

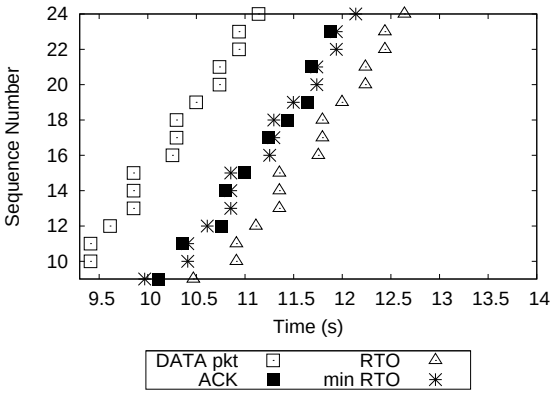



Fig. 1. Granularity = $500ms$, Round-Trip Propagation Delay = $500ms$

We reduce the round-trip propagation delay to $6ms$ and repeat the previous experiment (see Fig. 2). Again, we observe that $C(f) < 1$. We conclude that in case of coarse-grained clocks the Minimum RTO does not have negative 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, presently; we observe that the clock granularity is always set to a value below

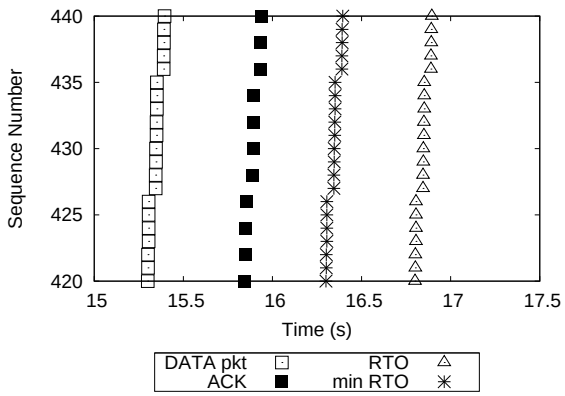



Fig. 2. Granularity = $500ms$, Round-Trip Propagation Delay = $6ms$

(or equal to) $25ms$. We run 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 3 uncovers the significant difference between the TCP-RTO values and the Minimum RTO limitation. In contradiction to the coarser-grained clocks, simulated previously, we observe that the $C(f)$ is now far above 1, obviously leading to severe performance degradation, in case of packet losses.

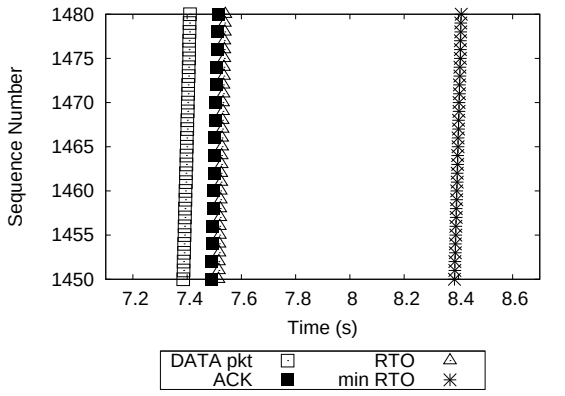



Fig. 3. Granularity = $10ms$, Round-Trip Propagation Delay = $6ms$

For the sake of simplicity, we assume the time interval between the ACK arrival and the RTO value, in Fig. 3, to be negligible and we modify the *Cost Function* (Equation 2) accordingly:

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

where $T(ACK\ Arr)$ holds the ACK Arrival Time, $RCG$ the *Receiver's Clock Granularity*, $RTPD$ the *Round-Trip Propagation Delay*, $TD$ the *Transmission 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. On the contrary, faster transmission links (i.e., high-speeds) will reduce the *Transmission Delay*, leading to Cost Function increase. We conclude that: i) the clock granularity should not be a matter of concern for setting the Minimum RTO, and ii) the conservative 1-second Minimum RTO will have major (negative) impact on TCP's performance, in case of packet loss.

## 3 Dealing with Delayed ACKs

The Delayed ACK mechanism [3] is quite popular among the vast majority of the 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 (piggybacking). 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, called *Adaptive MINRTO* (AdMINRTO), to identify the packets whose ACKs are (possibly) going to be delayed; 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 if it does have, we do not know after how long.

– 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 do not apply any Minimum RTO limitation 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 forward channel 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 Adaptive MINRTO 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 is applied (State 1). The flow-diagram of the proposed mechanism is presented in Fig. 4.

– 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, keep on to 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, keep on to 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.

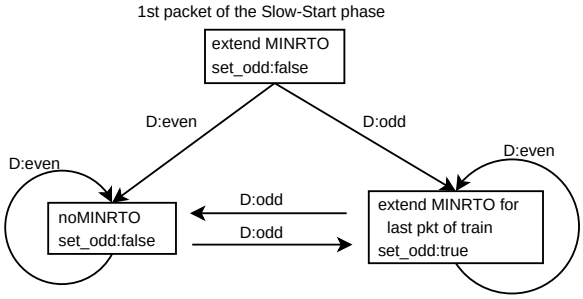



Fig. 4. The Proposed Mechanism

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

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

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. 5. 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 $3^{rd}$ packet's Minimum RTO and sets `set_odd` to true (State 2).

From that point onwards, 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).

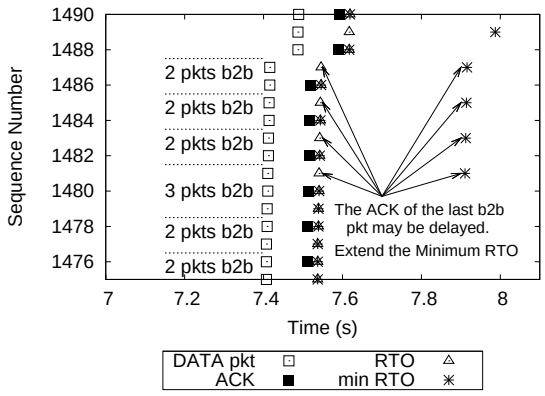



Fig. 5. Modeling the ACKs Arrival

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

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

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.

We note that the proposed mechanism does not apply for packets sent during the Fast Retransmit phase. During fast retransmit, the Minimum RTO is set to $R\ ms$; afterwards, the mechanism resumes from Step 6. Furthermore, the sender keeps one extra variable to account for Delayed ACKs that are, finally, sent to the TCP sender. Although the packet interarrival gap, which equals the

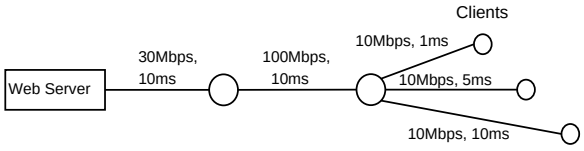
packet inter-departure gap plus potential sudden increases in network queuing delays, is typically smaller than $200ms$ (i.e., the Delayed ACK interval) for a regular FTP application [1], the proposed algorithm needs to deal with such exceptions as well. Otherwise, the algorithm may operate in a wrong state (see Figure 4). Furthermore, interactive applications, such as SIP [16] transactions over TCP, piggyback ACKs together with data packets from the client towards the direction of the server. In that case, it is possible that a Delayed ACK is piggybacked to a data packet in the TCP data channel. Again, the triggered Delayed ACK will enter the algorithm in the wrong state.

More precisely, arrival of a Delayed ACK is similar to `D` (Equation 4) becoming odd, which (according to Figure 4) causes State-change. Inline with the proposed algorithm's main functionality, which is to identify whether the next ACK expected is going to be delayed or not, we introduce a variable called `next_ack_expected`, which is TRUE when the ACK is, potentially, going to be delayed and FALSE, otherwise. Therefore, upon an ACK arrival and in case the `next_ack_expected = 1` the algorithm changes State, accordingly. This way, the proposed algorithm effectively avoids operation in the wrong state in case of triggered Delayed ACKs or piggybacked packets.

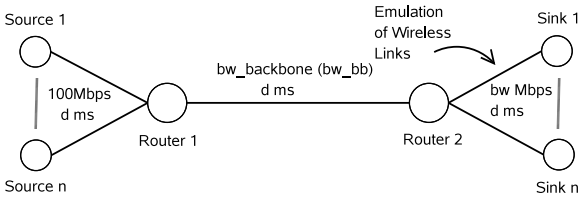
[1] We observed, through simulations, that the probability of a triggered Delayed ACK increases with the error rate, due to extensive idle periods caused by timeout events. However, we do not present such results here, since the behavior of Delayed ACKs is outside the scope of the current study.

## 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], [18]. That is, most OSs use the SACK version of TCP [7] with the timestamps option enabled [6] and the response against spurious timeouts [5], [17] 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 Figure 6; buffer sizes are set according to the $Bandwidth - Delay\ Product$ (BDP) of the outgoing links. We use the RED [4] queuing policy whenever $BDP > 20$ packets and Drop Tail otherwise, since we consider it unrealistic to set the RED minimum threshold to less than 5 packets.




(a) Topology 1




(b) Topology 2

Fig. 6. Simulation Topologies

We use two traditional performance metrics:

(1) the Task Completion Time (TCT) to measure the time required for a file

transfer, 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.

Whenever deemed appropriate, we also present the retransmission effort of the transport protocol. Fairness properties are not considered here, since the proposed algorithm neither hurts nor improves the protocol's Fairness performance.

## 5 Results

We divide the Results Section in three main subsections. Firstly (Section 5.1), we show that due to limited standardization efforts on the subject of the Minimum RTO setting, communication problems may arise when different OSs attempt to "talk" to each other. Next, we present the impact of the Minimum RTO setting on: i) web flows (Section 5.2.1), ii) short FTP flows (Section 5.2.2) and iii) long FTP flows (Section 5.2.3). Throughout Section 5.2, we emphasize on next generation, broadband wireless access networks, where flow-contention is low and losses occur, mainly, due to wireless errors. Finally, in Section 5.3, we simulate flows that experience congestion losses.

We find that the impact of the proposed mechanism, performance-wise, de-

pends on several network conditions. For example, the impact of the Adaptive MINRTO increases for file transfers that take place over short RTT paths or high speed links and losses happen mainly due to wireless fading channels. On the contrary, losses due to buffer overflow minimize the impact of the Adaptive MINRTO.

### *5.1 The Need for a Standard Mechanism*

We have already shown that there exist different implementation settings both for the Delayed ACK timer and for 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 and `D = 1` (e.g., after *every* timeout expiration).

$$Server's\ Minimum\ RTO < RTPD\ +\ TD\ +\ QD +\ Client's\ Delayed\ ACK\ Timer \tag{6}$$

$$Minimum\ RTO > RTO_{cur} \tag{7}$$

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. 6(a)).

Indeed, we see in Fig. 7 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. 8). In the present work,

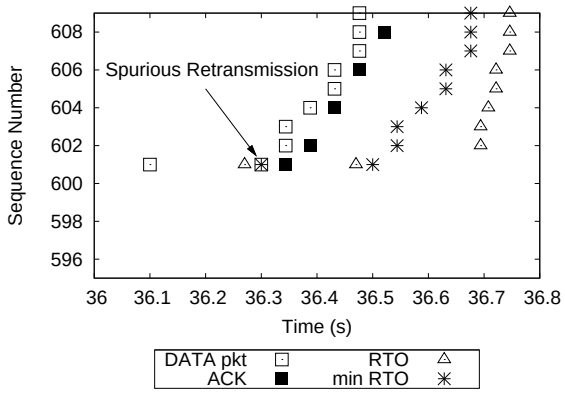



Fig. 7. Linux Server - 200$ms$ Delayed ACK Client (e.g., Windows client)

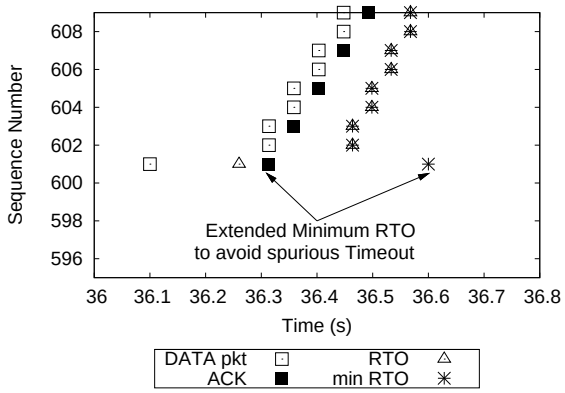



Fig. 8. Modified Linux Server - 200$ms$ Delayed ACK Client (e.g., Windows client)

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 $RTPD + TD + QD \leq 300ms$ (see Equation 6), since we have not found any implementation, where the Delayed ACK interval is greater than $200ms$.

### *5.2 Wireless Losses*

#### *5.2.1 Web Flows*

We use the topology shown in Fig. 6(a), where three 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 IETF, Solaris and Linux TCP implementations, comparatively with the Adaptive MINRTO, after 20 successfully completed tasks.

Since the propagation and transmission delays are the same in all cases, we subtract them in order to capture the delay difference solely due to the proposed algorithm. We present the performance difference ratio for each Minimum RTO setting comparatively i) with the previous larger Minimum RTO setting (first number in parentheses) and ii) with the largest Minimum RTO setting (i.e., IETF standard, second number in parentheses). That is, for example, a Linux server improves the $1^{st}$ flow's performance by 6% against a Solaris server and by 17.5% against an IETF-compliant server.

Table 2

Average Task Completion Time (ATCT)

| | flow 1 | flow 2 | flow 3 |
|---|---|---|---|
| IETF | 2.9s | 3.0s | 3.2s |
| Solaris | 2.55s (12.1%, 12.1%) | 2.75s (8.5%, 8.5%) | 3s (6.35%, 6.35%) |
| Linux | 2.4s (6%, 17.5%) | 2.5s (9.19%, 16.76%) | 2.7s (10.11%, 15.75%) |
| adMINRTO | 2.1s (12.6%, 27.65%) | 2.4s (4.12%, 20.1%) | 2.55s (5.67%, 20.4%) |

We observe, in Table 2, that in all cases (i.e., for all flows) the shorter the Minimum RTO setting, the faster the web-page transfer. Thus, the proposed algorithm is faster than the rest of the OS's Minimum RTO settings. The per-

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

formance difference may reach 12% improvement for Linux servers and 27% for IETF-compliant servers. At the same time, the algorithm presented here, avoids spurious timeout expirations, in contrast to the Linux TCP implementation, as shown in Section 5.1. We note that Mac OS, as well as FreeBSD implementations set the Minimum RTO value to 1.2 seconds. Although we do not present such results here, we noticed that greater values for the Minimum RTO degrade TCP's performance further.

### *5.2.2 Short Flows*

We conduct a series of experiments with regard to the time required to download small files (i.e., the TCT), up to 6MBs[3], considering various network conditions. We use the topology shown in Figure 6(b), the scenario setup details are presented in Table 3, while the corresponding results are presented in Figure 9.

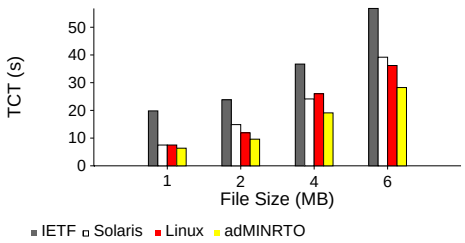
Table 3

Short Flows Experiment Details

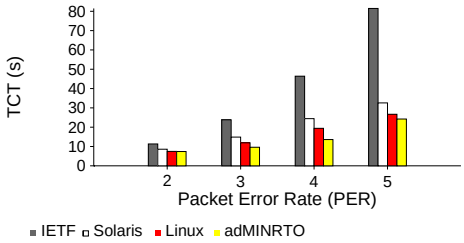
| | File Size | PER | Flows | $bw$ | $d$ |
|---|---|---|---|---|---|
| Fig. 9(a) | see Fig. | 3% | 3 | 5 Mbps | $1ms$ |
| Fig. 9(b) | 2MB | see Fig. | 3 | 5 Mbps | $1ms$ |
| Fig. 9(c) | 2MB | 3% | see Fig. | 5 Mbps | $1ms$ |
| Fig. 9(d) | 2MB | 3% | 3 | see Fig. | $1ms$ |

In all cases, we see that IETF-compliant servers extend significantly the file delivery time. Occasionally, the difference between the IETF specification and the rest of the Minimum RTO settings may reach 100% increase (e.g.,

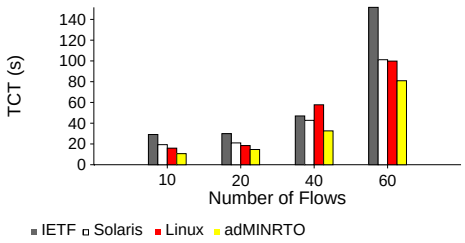
[3] Although the term "short flows" is typically used for web flows (i.e., up to 100KBs) in the related literature, it is very common that users wait for a small file delivery, before resuming other tasks (e.g., small program installation, software updates or heavy presentation files). Therefore, for the purpose of the current study, we use the term "short flows" for such file transfers as well.



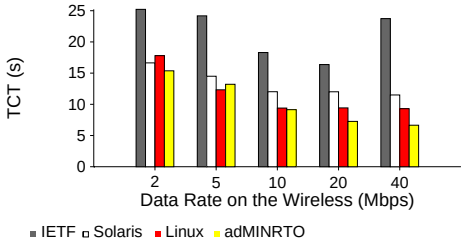

(a) Increasing File Size




(b) Increasing Packet Error Rate




(c) Increasing Contention




(d) Increasing Data Rate on the Wireless

Fig. 9. Short Flows

Fig. 9(b), PER = 5%). Clearly, the conservative 1-second Minimum RTO setting becomes inefficient since in many cases, users have to wait for more than one minute before a small file is downloaded. The rest of the Minimum TCP-RTO implementations achieve much faster Task Completion Times (TCT), with the Adaptive MINRTO being the fastest one. Although one would expect that the protocols' performance difference would decrease as contention increases, small buffer sizes set according to the BDP, are not associated with extensive queuing delays (see Figure 9(c)). The Linux Minimum RTO setting, occasionally, falls in consecutive spurious timeout expirations (see Section 5.1), increasing this way the Task Completion Time (e.g., Fig. 9(a), 4 MB or Fig. 9(c) 40 and 60 flows).

### *5.2.3 Long Flows*

We extend our experimental analysis to investigate the performance of long FTP flows. We use the topology shown in Figure 6(b); the simulation time is fixed to 300 seconds; the scenarios are summarized in Table 4, while the corresponding results are presented in Figure 10, where we measure the Goodput performance of the transport protocols.

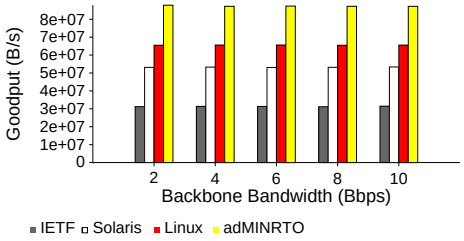
Table 4

Long Flows Experiment Details

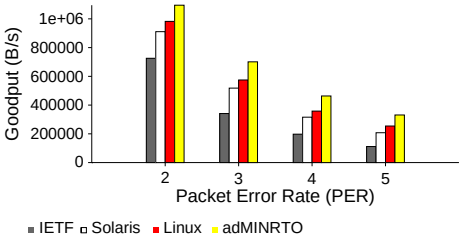
| | *bw_bb* | PER | Flows | *bw* | *d* |
|---|---|---|---|---|---|
| Fig. 10(a) | see Fig. | 3% | 500 | 50 Mbps | $1ms$ |
| Fig. 10(b) | 100 Mbps | see Fig. | 3 | 5 Mbps | $1ms$ |
| Fig. 10(c) | 100 Mbps | 3% | see Fig. | 5 Mbps | $1ms$ |
| Fig. 10(d) | 100 Mbps | 3% | 3 | see Fig. | $1ms$ |
| Fig. 10(e) | 100 Mbps | 3% | 3 | 5 Mbps | see Fig. |

We observe significant performance increase when the Adaptive MINRTO is used. Faster transmission links (i.e., increased bandwidth) allow for faster retransmission attempts and consequently better protocol performance, as we see in Fig. 10(a) and 10(d). Performance increase may be greater than 30% against the Linux Minimum RTO setting, 45% against the Solaris implementation and 100% against the IETF proposal. Results are similar in case of increasing PER (Figure 10(b)). Note that Mac OS and FreeBSD implementations will further reduce TCP's Goodput performance. We report that the performance difference increases even more, in extreme environments, where the packet error rate may be up to 10% or more (e.g., wireless mesh networks).

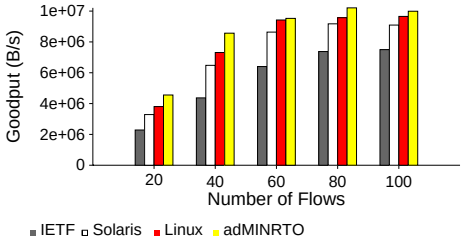
The effect of the proposed algorithm is reduced as the Round Trip Propagation Delay (RTPD) and the Queuing Delay (QD) increase. In particular, the effect of the proposed algorithm decreases as $RTPD + TD + QD \rightsquigarrow RTO_{min}(f)$, where $f$ represents the Linux, Solaris or IETF corresponding setting.



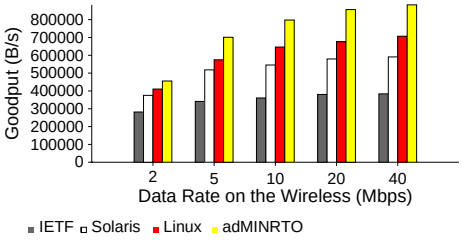

(a) Increasing File Size



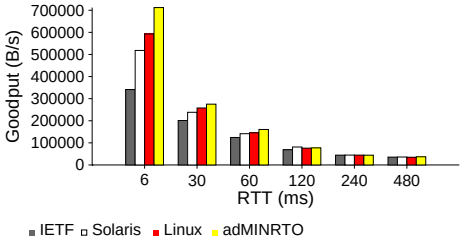

(b) Increasing Packet Error Rate




(c) Increasing Contention




(d) Increasing Data Rate on the Wireless




(e) Increasing RTT

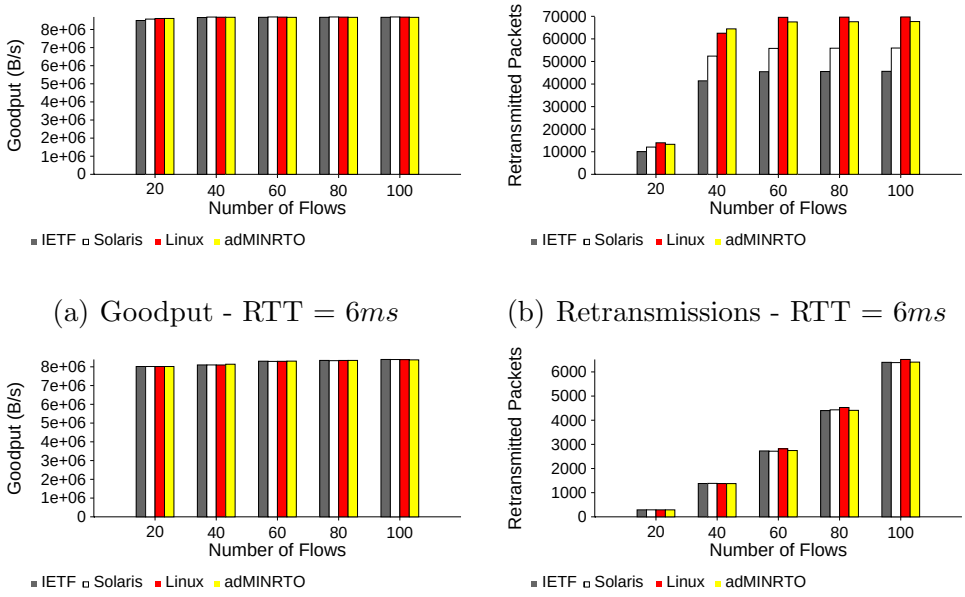
Fig. 10. Long Flows

Indeed, we see in Figure 10(e) that the effect of the proposed algorithm, gradually, fades away as the round trip propagation delay (and consequently the queuing delay) of the end-to-end path increases. Reduction of the performance difference, however, exists even for short round trip time paths. For example, in case $RTT = 60ms$, the performance difference of the proposed algorithm decreases to 10% against the Linux setting and to 22% against the IETF specification. Longer RTT paths (e.g., $120ms$) minimize the effect of the proposed algorithm, due to large buffer sizes, which, in turn, increase the queuing delay. Therefore, the proposed algorithm boosts TCP's performance for file trans-

fers that take place over short paths (e.g., within university campuses, cities or WANs), end-users are wirelessly connected and losses are due to fading channels.

### *5.3 Congestion Losses*

We evaluate the performance of the proposed algorithm when losses happen due to buffer overflow, rather than due to wireless errors. For that purpose, we repeat the simulation presented in Figure 10(c) but in the current setup, wireless $PER = 0\%$. The corresponding results are presented in Figure 11; in Fig. 11(a) and 11(b) $RTT = 6ms$, while in Fig. 11(c) and 11(d) $RTT = 120ms$.




(a) Goodput - RTT = $6ms$

(b) Retransmissions - RTT = $6ms$

(c) Goodput - RTT = $120ms$

(d) Retransmissions - RTT = $120ms$

Fig. 11. Congestion Losses

In both cases, we see that there is no difference in the Goodput performance of the different TCP implementations (Fig. 11(a) and 11(c)). Similar results were presented in a recent, trace-driven, study [15]. As for the retransmission

effort, we observe that in all cases, excluding the Linux TCP implementation, the shortest the Minimum RTO setting, the more the retransmission effort of the transport protocol, when the data transfer takes place over short RTT paths (Fig. 11(b)). Noteworthy, we observe that the Linux Minimum RTO implementation, which, in conjunction with the $200ms$ Delayed ACK timer, falls in spurious timeout expirations quite frequently (see Section 5.1), increasing this way the number of retransmitted packets (Fig. 11(b) and 11(d)).

As we have already shown in Fig. 10(e), the effect of small Minimum RTO values decrease as the RTT increases. This expectation is further validated in Fig. 11(d), where we see that smaller Minimum RTO settings do not result in increased retransmission effort. Higher levels of flow contention result in inefficient retransmission timeout (Equation 1) calculations [14], [12], [13] and, hence, such scenarios do not correspond to the scope of the current study.

## 6 Deployability

Extensive performance evaluation revealed that the proposed algorithm improves significantly TCP's performance in case of data transfers over short (up to approximately $100ms$) RTT paths, where the last-mile wireless link induces random errors. In case of short RTT paths and minimal wireless losses, short Minimum RTO settings increase TCP's retransmission effort. Otherwise, in case of long RTT paths and losses due to congestion, the proposed algorithm neither improves nor degrades TCP's performance. Therefore, the proposed algorithm presents high potential for deployability, since it does not impact (by increased retransmission overhead) the stability and convergence of long, transatlantic Internet flows. Instead, (incremental) deployment of the

proposed algorithm will improve TCP performance for flows within university campuses, cities or regional urban areas.

Furthermore, current TCP implementations, due to different Minimum RTO settings, may occasionally become unfair. For example, the Linux implementation is much more aggressive than the IETF specification. Therefore, users that download data from a Linux server may steal bandwidth resources from users connected to an IETF-compliant server. A universal setting for the Minimum TCP-RTO can cancel this unfairness.

## 7 Conclusions

We have shown that the conservative 1-second Minimum RTO setting causes severe TCP performance degradation, especially in case of wireless losses. We argued that such a conservative 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 Adaptive MIN-RTO algorithm. Simulation results show that the proposed algorithm achieves significantly higher Goodput performance, when flows traverse wireless fading channels, while the performance difference decreases in case of congestion losses.

## 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. WB-RTO: A Window-Based Retransmission Timeout. In *Proceedings of Globecom 2006, San Francisco, CA, USA*, November 2006.

[13] Ioannis Psaras and Vassilis Tsaoussidis. Why TCP Timers (still) Don't Work Well. *Computer Networks Journal (COMNET), Elsevier Science*, 51:2033–2048, 2007.

[14] Ioannis Psaras, Vassilis Tsaoussidis, and Lefteris Mamatas. CA-RTO: A Contention-Adaptive Retransmission Timeout. In *Proceedings of ICCCN 2005, San Diego, CA, USA*, October 2005.

[15] S. Rewaskar, J. Kaur, and F. Donelson Smith. A Performance Study of Loss Detection/Recovery in Real-world TCP Implementations. In *Proceedings of ICNP 2007*.

[16] J. Rosenberg, H. Schulzrine, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol, RFC 3261, June 2002.

[17] P. Sarolahti, M. Kojo, and K. Raatikainen. F-RTO: An Enhanced Recovery Algorithm for TCP Retransmission Timeouts. In *Proceedings of ACM SIGCOMM*, September 2003.

[18] P. Sarolahti and A. Kuznetsov. Congestion control in linux TCP. In *Proceedings of USENIX'02*.