WB-RTO: A Window-Based Retransmission Timeout for TCP

Ioannis Psaras and Vassilis Tsaoussidis
Dept. of Electrical and Computer Engineering
Demokritos University of Thrace
Xanthi, Greece
Email: {ipsaras, vtsaousi}@ee.duth.gr

Abstract— **We present a new timeout algorithm for TCP, based on the observation that TCP-RTO should not be solely based on RTT estimations. We argue that the design principles of the current timeout algorithm may lead to flow synchronization, unnecessary retransmission effort and unfair resource allocation. WB-RTO exhibits two major properties: (i) it cancels retransmission synchronization which dominates when resource demand exceeds by far resource supply and (ii) reschedules flows on the basis of their contribution to congestion.**

I. Introduction

The retransmission timeout policy of standard TCP [14] is governed by the rules of RFC 2988 [13]. The algorithm is based solely on RTT measurements, trying to capture dynamic network conditions by measuring the variation of the RTT samples. In particular, the Retransmission Timeout is calculated upon each ACK arrival after smoothing out the measured samples, and weighting the recent history. More precisely, upon each ACK arrival, the sender:

- calculates the RTT Variation:

$$RTTVAR = 3/4 \times RTTVAR + 1/4 \times |SRTT-$$

$$RTT_{SAMPLE}(1)$$

- updates the expected RTT prior to calculating the timeout:

$$SRTT = 7/8 \times SRTT + 1/8 \times RTT_{SAMPLE} \qquad (2)$$

- and finally, calculates the Retransmission Timeout value:

$$RTO = SRTT + max(G, 4 \times RTTVAR) \qquad (3)$$

where, RTTVAR holds the RTT variation and SRTT the smoothed RTT. In Equation (3), G denotes the timer granularity, which is incorporated in order to assign a lower bound to the retransmission timeout to protect it against spurious timeouts (i.e. when RTT equals the timer granularity).

Although the design of the timeout algorithm has been studied extensively in the past (e.g. [5], [8], [9], [11], [12], [13], [17], [19]), the association with its inherent scheduling properties has not really been evaluated adequately. Instead, much attention has been paid on its ability to reflect network delay accurately (e.g. [1], [3]), allowing for speedy retransmission when conditions permit and avoiding double submission due to early expiration. However, network delay as it is captured by measuring the RTT alone, cannot always capture the level of flow contention [16]. In this context, flows with large windows, a common situation when flow contention is low, do capture network delay with better precision: for example, last packets of the same window will experience more delays. The situation appears to be very diverse when each flow is represented with a single-packet window, for example.

However, there are 3 points to make: 1) this diversity can only be reflected on the timeout when flows are not synchronized, 2) the responsiveness of the algorithm depends on the weighting of the measured samples and 3) all packets entering a full buffer experience the same delays, leading all flows to adjust to the same timeout value. Hence, it is possible for the timeout to shrink when contention grows, at least for those flows that experience less delays in the queue [16].

The problem of scheduling as it is associated with timeout has another dimension as well. When flows are synchronized, or becoming synchronized due to a congestion event, the timeout is adjusted accordingly for all participating flows. The existing policy to exclude the retransmitted (dropped) packets from measurements leaves little space for timeout differentiation among the participating flows, thus leading to possibly synchronized retransmissions. Therefore, fairness cannot be guaranteed since flows are not randomly scheduled, but instead, are possibly partitioned into two groups: the one consisting of flows that utilize the link and the other consisting of flows that unsuccessfully attempt to enter at times when the link is utilized.

The equations that form the timeout algorithm (Equations (1), (2) and (3)) do not allow for differentiation among flows that experience the same queuing delay. For example, flows that enter a system simultaneously will be ordered in the queue and possibly follow the same order throughout the upcoming transmission rounds. Furthermore, flows that enter the system when the buffer is fully utilized, may also be excluded in the next rounds, for the same reason. Current timeout scheduling becomes deterministic, allowing only a particular set of participating flows to utilize the link.

In this paper, we extend our previous study [16] in the sense that we still consider cases, where the timeout algorithm becomes the scheduler for the link. We present a new algorithm which records the congestion window and calculates the average window in order to capture how contention evolves with time. In addition, our algorithm uses the current congestion window to assess the contribution of the flow to congestion. The RTT instead is only used to determine the lower bound of the timeout. We call our algorithm *Window-Based RTO*. We investigate the behavior of the proposed algorithm and find that WB-RTO cancels TCP's inability to administer simultaneous retransmissions and consequently WB-RTO achieves higher goodput, better fairness, and less retransmission overhead.

II. Related Work

Several researchers have reported problems regarding the TCP-RTO [5], [9], [12], [16], [17], [19]. Lixia Zhang, in [19], identifies several drawbacks of the TCP retransmission timer and reports its intrinsic limitations. The paper concludes that mainly external events should trigger retransmissions and timers should be used only as a last notification about packet loss. Although WB-RTO departs from a different point, it also alleviates problems reported in [19], due to the fact that these problems are mainly caused by the exclusive relation of the timers with the RTT.

The Eifel Algorithm [9], [12], [8] focuses on spurious timeouts. The Eifel algorithm uses the TCP timestamp option [10], in order to detect spurious timer expirations. Once a spurious timeout is detected, the sender does not back-off, but instead it restores the transmission rate to the recorded rate prior to the timeout event.

The Forward RTO Algorithm [17] targets the detection of spurious timeouts too. The algorithm instead of using timestamp options, it checks the ACK packets that arrive at the sender after the timer's expiration, observes whether the ACKs advance the sequence number or not and finally concludes on whether the timeout was spurious or not.

Both the above algorithms (Eifel [12] and F-RTO [17]) improve TCP's performance ([2], [18]) significantly. However, none of them really solves the problems stated in [19], due to the fact that they do not modify the retransmission timeout algorithm itself, but instead they only change the *response* of the transport protocol *after a timeout has occurred*. More precisely, both algorithms ([12], [17]) take into consideration outstanding data packets only after the timer expires, while the nature of the problem calls for modifications of the timeout itself. In this context, WB-RTO adjusts the timeout to an appropriate value which efficiently avoids unnecessary retransmissions (up to 500%), without impacting the goodput performance of TCP (see section V).

In [16], we have shown that TCP timers, based solely on RTT estimations, do not always respond according to the level of flow contention. We investigated the behavior of TCP in high contention scenaria and confirmed that it is possible for the timeout to decrease when contention increases. We concluded that this anomaly is due to flow synchronization. Our analysis called for a new design, which should also account for approximation of the level of flow contention, along with a mechanism to cancel synchronization. We exploit these directives in the present work.

III. WB-RTO: The Proposed Algorithm

We propose a new algorithm for the TCP RTO to: (i) practically approximate the level of contention, (ii) estimate the contribution of each flow to congestion and (iii) allow for asynchronous retransmissions, ordered in time in reverse proportion to their contribution to congestion. We note that (i), (ii) and (iii) form a collaborative detect/respond scheme to increase efficiency of the *system* response, i.e. not a particular flow response. Clearly, if all flows calculate a single accurate time for retransmission, the system will fail to provide efficient service, due to flow synchronization. In this context, (iii) allows for better responsiveness to contention.

A. Implementation

During periods of high contention, it is possible for all flows to operate with minimal windows, in which case we induce randomization to guarantee timeout diversity. However, it is also possible for the contending flows to operate with different window sizes. In this case, we attempt to adjust the timeout according to the degree of their contribution to congestion, which is captured in parameter c. In particular[1], we initially classify the flow depending on its current transmission window and charge it with an appropriate penalty (parameter c), according to the following observations: the current congestion window ($cwnd\_$) is compared with the maximum congestion window ($max\_cwnd\_$) that the flow has ever reached, since the connection establishment. In case the *cwnd_* is smaller than half of the maximum congestion window, the flow is marked with the minimal charge ($c = 1$). If the flow's *cwnd_* belongs to the interval $[max\_cwnd\_/2,\ (3/4)\times max\_cwnd\_)$, the penalty is higher ($c = 1,5$). Finally, if the current *cwnd_* lies in the remaining area ($[(3/4)\times max\_cwnd\_,\ max\_cwnd\_]$), the flow is given a major penalty ($c = 2$). The justification for the above policy is as follows: the penalty charged to the flow will influence the retransmission timeout value. The higher the penalty, the longer the RTO. This way, we attempt to punish flows operating with large congestion windows upon a timeout event, since the timer's expiration is considered here as a congestion signal. On the contrary, a flow operating with small congestion window (compared to the maximum congestion window) has not contributed much to the network load and hence there is no need to be punished. The above statements are grafted in Equation (4) below.

Next, we further classify the flow according to its (recent) congestion window history (average window, denoted as $awnd\_$). We define four different thresholds[2] (*Threshold 1 to 4*) and classify the flow according to its *awnd_* value, which corresponds to the current level of flow contention (see Equation (5) below). *Threshold*$_1$ corresponds to very high contention, while *Threshold*$_4$ refers to the situation where

[1]The following proccess is triggered every time the RTO is calculated for a specific TCP flow.

[2]Note that both the thresholds and the parameters discussed below are determined based on experiments and they constitute subject of further investigation (see section V.B).

congestion events happen sparsely. Throughout the experiments, *Threshold*$_1$ (high contention) was set to 5 packets, while *Threshold*$_4$ (low contention) was set 50 packets.

We assign four multipliers (a_1 to a_4) corresponding to the four predetermined intervals ($(0, Threshold_1)$ to $(Threshold_3, Threshold_4)$). More precisely, a_i (where $a_i < a_{i-1}$) multiplies c in order to set the penalty and consequently the RTO based on both the window and its history (Equations (6) and (7). *Summarizing, the first classification (i.e. parameter c in Equation 4) captures the contribution of the flow to congestion, while the second classification (Threshold 1 to 4 in Equation (5)) gives an estimation of the level of flow contention.*

Finally, we randomly select a value from the interval (rtt, $a_i \times c$), where the lower bound, rtt, guarantees that the timeout will not expire prior to the RTT, preventing the algorithm from becoming too aggressive (Equations (6) and (7). The pseudocode is presented below:

- compare *cwnd_* with *max_cwnd_* and assign a penalty accordingly
$$c = f(cwnd\_,\ max\_cwnd\_) \tag{4}$$
- classify the flow to the appropriate *Threshold*, according to its *awnd_*
$$a_i = g(awnd\_,\ Threshold_i) \tag{5}$$
- finally, calculate the Window-Based RTO
$$WB - RTO = random(rtt,\ c\ \times\ a_i) \tag{6}$$
or
$$WB - RTO = random(rtt,\ f(cwnd\_,\ max\_cwnd\_)$$
$$\times\ g(awnd\_,\ Threshold_i))(7)$$

where $Thresholds$ 1 to 4 are set to 5, 10, 30 and 50, respectively and the corresponding parameters (a_1, a_2, a_3, a_4) are set to 10, 5, 3 and 2, respectively.

B. Expected Behavior

The proposed algorithm attempts to i) to incorporate the level of flow contention (Equation (5)), ii) penalize flows according to their contribution to contention (Equation (4)) and iii) schedule retransmission in a manner that synchronization is avoided (Equation (7)).

In Figure 1, we present the behavior of WB-RTO for a wide range of average transmission windows. Three plots are presented in this Figure. Each line plot represents the response of WB-RTO relevant with the average transmission window for the three possible penalties. Three points are salient: i) the highest values (in average) for the retransmission timeout correspond to the highest penalties, ii) it is possible for a flow to calculate a small RTO even when it operates with large windows, iii) timeout settles to smaller values as the average window increases. Note that in all cases,

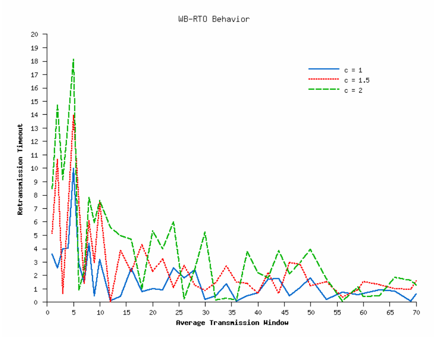


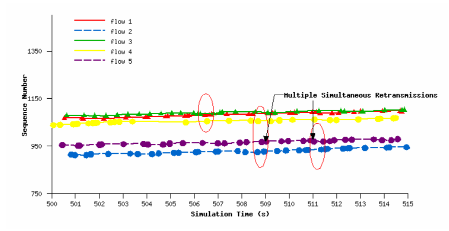
Fig. 1. WB-RTO vs awnd_

we prevent the algorithm from calculating a timeout value smaller than the round trip time, since this would result in an undesirably aggressive behavior, which would negatively impact the system performance.

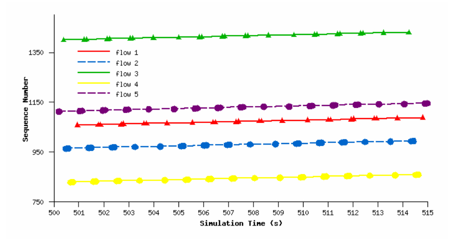
C. Proof of Concept

We provide a more detailed analysis based on a simulated scenario using the Dumbbell network topology (Figure 4). Our scenario involves 5 high-demanding sessions over a low $Delay \times Bandwidth$ product link ($D \times B = 10pkts$), where buffers are set in accordance, to hold up to 10 packets. Due to limited resource supply, the demand should be adjusted to an average of 2 packets per window per flow. In Figure 2 we use the sequence number to capture the progress in time. We highlight the interval between 500 and 515 seconds, out of a simulation run that lasts for 1500 seconds and note that the sequence number progress is similar throughout the whole simulation. We observe that TCP-RTO scheduling results in multiple simultaneous transmissions/retransmissions (for example packets in the circles). Figure 3(a), which depicts the measured RTT for the same time interval, justifies the noticeable difference observed in Figure 2. We observe that no RTT variation is present (Figure 3(a)) among different flows, resulting in identical RTO settings (Figure 3(b)) for the participating flows, and in turn, to flow synchronization.

On the contrary, WB-RTO schedules flows more efficiently as shown in Figure 2(b). A closer look verifies that no packets are transmitted simultaneously (at least not more than 2 at a time). The Goodput performance and the retransmission effort of the two protocols, is shown in Table I. TCP-RTO retransmits approximately 300 packets per flow more than WB-RTO, resulting in a sum of 1500 more retransmissions in a total of 1500 seconds of simulation time. The fairness of the two protocols is similar, with a slightly fairer behavior for WB-RTO. Although we can not present the sequence number progress for the complete experiment due to space limitations, we observed that, in case of standard TCP and in the last 300 seconds of the experiment, the third flow did not advance its sequence number, resulting in degraded Goodput performance and hence in unfair system behavior. This is further explained by Figure 3(b), where we see that flow 2 makes wrong RTO estimation which results in an extraordinary long timeout wait

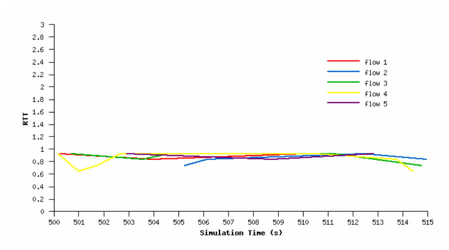


(a) TCP-RTO

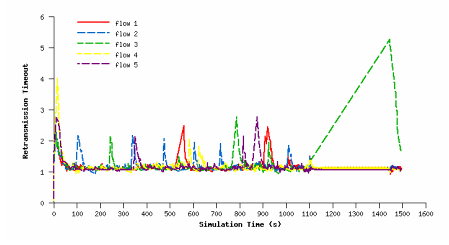


(b) WB-RTO

Fig. 2. Sequence Number Progress



(a) Round Trip Time (in seconds)



(b) Retransmission Timeout (in seconds)

Fig. 3. Standard TCP Behavior

in the last 300 seconds of the simulation.

IV. Evaluation Methodology

We have implemented our evaluation plan on the ns-2 network simulator. The network topology, used as a test-bed, is the single-bottleneck dumbbell shown in Figure 4 and the simulation time was fixed at 1500 seconds.

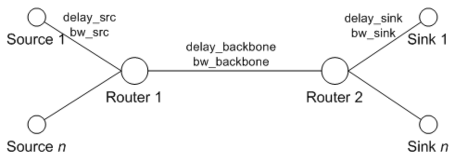


Fig. 4. Simulation Topology

TABLE I
Protocol Performance in the Small Scenario

	Goodput (KB/s)		Retransmitted Packets	
	TCP-RTO	WB-RTO	TCP-RTO	WB-RTO
1st Flow	1.95	2.25	600	310
2nd Flow	2.2	2.15	620	300
3rd Flow	1.8	2.35	550	315
4th Flow	2.0	2.05	600	315
5th Flow	2.05	2.2	620	305
Total	10.0	11.0	2990	1545

In the current work we test the performance of the proposed algorithm under the Drop Tail queuing policy[3], where the buffers' capacity is set to 50 packets. The *Window-Based Retransmission Timeout* is implemented in TCP-Reno. We note that with the exception of Tahoe, which lacks the Fast Recovery mechanism, all other TCP versions perform similarly and hence, such implementations are excluded from the current work.

V. Simulation Results

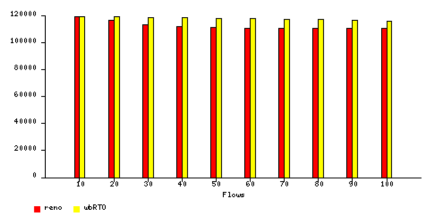
Our simulation experiments focus mainly on cases with high contention, where the timeout plays a dominant role in scheduling. We describe the results in two subsections. In the first subsection we describe the behavior of WB-RTO over a Dumbbell network topology (Figure 4). In the second subsection we discuss parameter adjustments and exploit some interesting performance tradeoffs.

A. Performance Evaluation

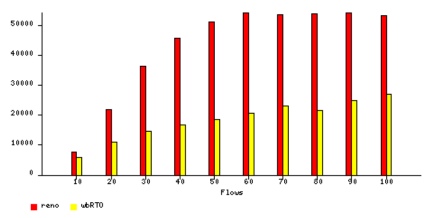
The Delay-Bandwidth Product ($D \times B$) of the backbone link in the dumbbell network topology is 100 packets and the router's Drop Tail buffer can hold up to 50 packets. The simulation is repeated 10 times, increasing each time the number of participating flows (10, 20,..., 100). We have intentionally designed the simulated conditions so that resource supply does not suffice for what users demand. The purpose is to give the timeout algorithm the ultimate role of the transmission scheduler for the link. In our case, the main target of the retransmission timer is to distribute flows in time and permit *all* flows to utilize the network resources, instead of rejecting some flows for the benefit of the rest. Consequently, we expect improvement in terms of fairness [4].

Figure 5 summarizes the performance of the two protocols. WB-RTO outperforms TCP-RTO in all cases, in terms of Goodput (Figure 5(a)). TCP-RTO results in 50% more retransmissions (25MB in 1500 secs) than WB-RTO does, as shown in Figure 5(b). In particular, TCP-RTO retransmits more than 50.000 packets (almost 50MB), while WB-RTO retransmits less than 20.000 packets, when 60 flows utilize the link (Figure 5(b)). Note that even with that huge number of retransmissions, standard TCP can not achieve better Goodput performance than WB-RTO. Concluding, we argue that this

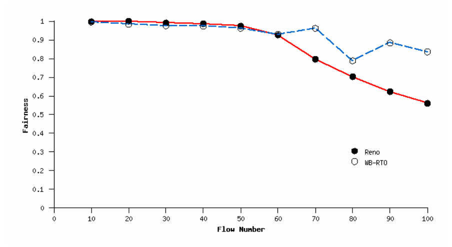
[3]Further performance evaluation, including Active Queue Management schemes can be found in [15].



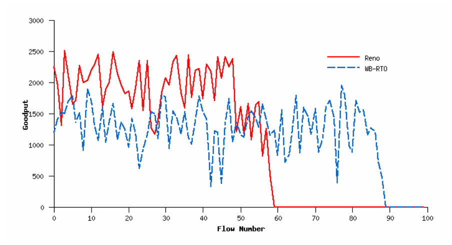
(a) Goodput (in Bytes/sec)



(b) Retransmitted Packets



(c) Fairness



(d) Goodput per Flow

Fig. 5. Protocol Performance in the Dumbbell Topology

behavior owes to the unnecessarily forcefull retransmission policy of standard TCP.

One may naively think that the difference in Goodput performance achieved by WB-RTO is negligeable compared to the traditional TCP RTO. However, the *Retransmission* Timeout Algorithm is responsible for the *Retransmission* effort of the protocol, rather than for the actual Goodput performance of the protocol. Hence, we pay more attention to the combination of the *retransmission effort* spent by the protocol in order to achieve the measured *Goodput* performance, rather than on the Goodput performance alone. Furthermore, we focus on the scheduling potential of the TCP timeout algorithm, emphasizing on the fair resource allocation among the competing flows.

In this context, WB-RTO greatly outperforms TCP-RTO in terms of fairness (see Figure 5(c)); when contention increases (e.g. more than 60 flows), the scheduling property of the timeout becomes more dominant. In effect, TCP RTO behaves unfairly to some flows, since it continously fails to provide a time scale suitable for the whole system of flows, which could guarantee efficient link utilization. To strengthen our claims, we analyze the achieved Goodput per flow in Figure 5(d), in case of 100 participating flows. As expected, we see that in case of standard TCP, some flows continously transmit (flows 1-60), while the rest of them (flows 61-100) continously get rejected and achieve zero Goodput performance. On the contrary, WB-RTO successfully provides resources to the most of the participating flows (only 10 flows get rejected). Standard TCP's forcefull retransmissions result in higher Goodput for the first 60 flows (compared to WB-RTO), while WB-RTO's more adaptive retransmission timer allows for transmission opportunities to a larger number of participating flows (Figure 5(d)). WB-RTO exhibits one significant characteristic: it occassionally behaves aggressively and occassionally more conservatively, depending on the level of contention (as this is captured by the window adjustments of the protocol).

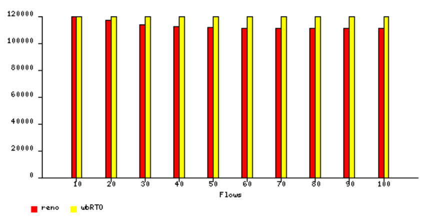
B. *Tradeoffs and Parameter Adjustments*

As already mentioned in section 3, both the *Thresholds* and the corresponding parameters (a_1, a_2, a_3, a_4), of the proposed algorithm are set experimentally. We deemed necessary to exploit when and how far do such settings impact the results and the generality of the algorithm. In this section, we assign different values to parameters a_1 to a_4, simulate the previous scenario and investigate performance tradeoffs regarding the behavior of WB-RTO. More specifically, we increase 10 times the values of a parameters[4], making the algorithm even more conservative in terms of the timeout value adjustments.

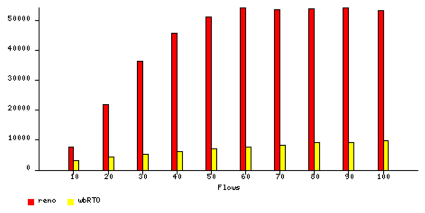
In Figure 6 we plot the performance of the protocols. The difference in the Goodput performance (Figure 6(a)) is even greater than what it was previously (Figure 5(a)). Furthermore, the number of retransmissions for WB-RTO decreased, reaching the outstanding level of 500% less retransmissions (Figure 6(b)). However, fairness degrades the overall performance of WB-RTO (Figure 6(c)) when less than 60 flows compete for network resources. Figure 6(d) reveals the unfair behavior of WB-RTO. In this Figure, we plot the per flow Goodput performance for the experiment of 100 participating flows. Like before, TCP-RTO rejects the last 40 flows from utilizing the link, while this time WB-RTO extents the timeout value long enough and gives transmission opportunities to all flows, resulting in more fair resource utilization (last dots in Figure 6(c)).

The same Figure, holds also the justification for the unfair behavior of WB-RTO, when less than 60 flows participate in the experiment. That is, we notice significant fluctuation in the goodput performance of different flows, which results in unfair resource allocation (Figure 6(c)): a flow enters the link operating with small average window during the start-up phase. Upon packet loss, the timeout is likely to get a rahter high value, since the interval $(rtt,\ c\ \times\ a_i)$ is now larger. After possibly waiting for a long time, the flow attempts to

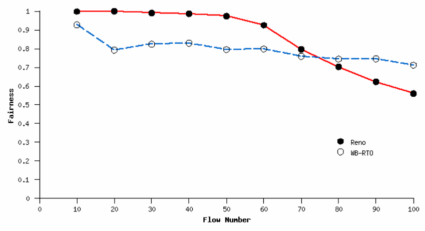
[4]$a_1 = 100$, $a_2 = 50$, $a_3 = 30$, $a_4 = 20$



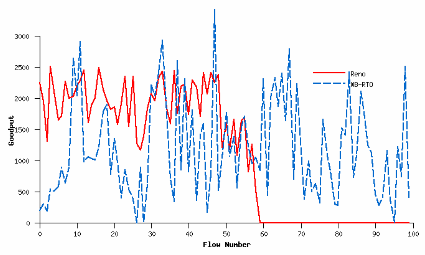
(a) Goodput (in Bytes/sec)



(b) Retransmitted Packets



(c) Fairness



(d) Goodput per Flow

Fig. 6. Protocol Performance in the Dumbbell Topology (Conservative Adjustments)

enter the link, but the *awnd_* is still small resulting in another possible long wait, in case of timer expiration. The situation remains, and the flow awaits for long time periods, until it can retransmit, something that inevitably degrades Goodput performance[5]. It is worth mentioning that RED gateways cancel this drawback of the proposed algorithm [15]. Inline to their design property, RED [6] gateways drop packets in proportion to the transmission rate.

The above analysis exploits an interesting tradeoff: high values for the Window-Based RTO reduce retransmission overhead at the cost of fairness. More sophisticated parameter adjustments constitute subject of future work.

We have also investigated the performance of WB-RTO in scenaria with different time-demanding applications (i.e. short together with long flows [7]). As a side effect, we observed that long, standard-TCP flows synchronize their transmission attempts with the short web-like flows. Given that, the queuing delay increases periodically affecting the performance of the time-demanding web applications. On the contrary WB-RTO schedules short flows independently of the long ones, resulting in more efficient link and buffer utilization [15].

[5]The situation is similar to the Ethernet capture effect.

VI. Conclusions

We have shown that TCP timers do not always adjust to the level of flow contention. Several flows may calculate the same timeout, leading to congestion events due to simultaneous retransmissions. We proposed a solution based on three mechanisms: i) approximation of the current level of network contention, ii) estimation of the contribution of the flow to congestion, and iii) allowance for asynchronous retransmissions when timeout happens. Our results match our design goal. However, during simulations we exploited interesting performance tradeoffs, which call for further optimizations.

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] E. Blanton and M. Allman. On Making TCP More Robust to Packet Reordering. *ACM Computer Communication Review*, 32(1), January 2002.

[4] D.-M. Chiu and R. Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems*, 17(1):1–14, 1989.

[5] Hannes Ekstrom and Reiner Ludwig. The Peak-Hopper: A New End-to-End Retransmission Timer for Reliable Unicast Transport. In *Proceedings of IEEE INFOCOM*, 2004.

[6] S. Floyd and V. Jacobson. Random Early Detection gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[7] Liang Guo and Ibrahim Matta. The war between mice and elephants. In *Proceedings of ICNP 2001*, October 2001.

[8] A. Gurtov and R. Ludwig. Evaluating the Eifel Algorithm for TCP in a GPRS Network. In *Proceedings of European Wireless*, 2002.

[9] A. Gurtov and R. Ludwig. Responding to Spurious Timeouts in TCP. In *Proceedings of IEEE INFOCOM*, 2003.

[10] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance, RFC 1323, May 1993.

[11] P. Karn and C. Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. In *Proceedings of ACM SIGCOMM*, September 1987.

[12] R. Ludwig and H. Katz. The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions. *ACM Computer Communication Review*, January 2000.

[13] V. Paxson and M. Allman. Computing TCP's Retransmission Timer, RFC 2988, May 2000.

[14] J. Postel. Transmission Control Protocol, RFC 793, September 1981.

[15] I. Psaras and V. Tsaoussidis. On the Scheduling Properties of TCP Timeout, http://utopia.duth.gr/ ipsaras/rtoproperties.pdf. *Technical Report*, February 2006.

[16] I. Psaras, V. Tsaoussidis, and L.Mamatas. CA-RTO: A Contention-Adaptive Retransmission Timeout. In *Proceedings of ICCCN*, October 2005.

[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] W. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, RFC 2001, January 1997.

[19] L. Zhang. Why TCP Timers Don't Work Well. In *Proceedings of ACM SIGCOMM*, pages 397–405, September 1986.