Responding to Spurious Timeouts in TCP

Andrei Gurtov
University of Helsinki
Finland

Reiner Ludwig
Ericsson Research
Herzogenrath, Germany

Abstract - **Delays on Internet paths, especially including wireless links, can be highly variable. On the other hand, a current trend for modern TCPs is to deploy a fine-grain retransmission timer with a lower minimum timeout value than 1 s suggested by RFC2988. Spurious TCP timeouts cause unnecessary retransmissions and congestion control back-off. The Eifel algorithm detects spurious TCP timeouts and recovers by restoring the connection state saved before the timeout. This paper presents an enhanced version of the Eifel response to spurious timeouts and illustrates its performance benefits on paths with a high delay-bandwidth product. The refinements concern the following issues (1) an efficient operation in presence of packet losses (2) appropriate restoration of congestion control, and (3) adapting the retransmit timer to avoid further spurious timeouts. In our simulations the Eifel algorithm on paths with a high delay-bandwidth product can increase throughput by up to 250% and at the same decrease the load on the network by 3%. The proposed response also shows adequate performance on heavily congested paths.**

I. Introduction

Recent measurement studies [11] show that TCP [41] maintains its position as the dominant transport protocol in the Internet. TCP is a stable, mature, and probably the most thoroughly tested protocol of its kind. Nevertheless, there are some corner cases where TCP could still be improved. The problem of *spurious timeouts*, i.e., timeouts that would not have occurred had the sender waited “long enough”, is an example of the above mentioned corner cases.

Internet measurements show highly variable delays on some paths resulting for example from route flipping [2][3][9]. A measurement study of dial-up connections reports occasional delay jitter of several seconds due to link-layer error recovery by a modem [32]. Especially on wireless links mechanisms providing error recovery, mobility, on-demand resource allocation and priority scheduling can cause high delay variation [20]. For example, we measured delay spikes of several seconds occurring frequently in a wireless cellular network due to cell changes [31][21]. An independent study reports abrupt changes in link RTT resulting from on-demand allocation of a high-speed radio channel [49].

Mobile users start utilizing heterogeneous overlay networks for Internet access. Currently inter-network handovers are extremely challenging in terms of delay jitter and data loss, and it seems unlikely that such disruptions can be completely avoided in the near future [47].

A sudden delay increase can cause a spurious TCP timeout which presents two problems [33]. First, outstanding segments are retransmitted unnecessarily. Second, the congestion control [26][1] is falsely triggered. The Eifel algorithm suggested in [33] uses the TCP timestamp option [27] to reliably detect spurious timeouts and eliminates these two problems. The algorithm can be also used to detect spurious fast retransmits due to packet reordering. However, in this paper we only address response to spurious timeouts.

The importance of recovering from spurious timeouts is increasing as modern TCPs, for instance Linux 2.4, start using a finer-grain timer (10 ms) and a low minimum retransmit timeout value (200 ms) [43]. The recent stable Linux kernel release 2.4 includes the Eifel algorithm. Eifel is advancing through the standardization process in the IETF [35] [36]. Therefore, it is important to assure that the algorithm is efficient and safe for wide deployment in the Internet. The goal of this paper is to refine the response part of the algorithm and to demonstrate its utility in the environment with a high delay-bandwidth product. We show that Eifel can potentially have significant performance benefits for TCP that justifies efforts and additional complexity in its development in order to produce an efficient and robust solution to the problem of spurious timeouts. All required modifications are located at the TCP sender.

The rest of the paper is organized as follows. In Section II we review the problem of spurious timeouts, the Eifel detection algorithm and related work. In Section III we motivate and explain modifications to the Eifel response algorithm. Section IV describes our modelling approach and evaluates the new response for paths with a high bandwidth-delay product. Finally, Section V presents conclusions and plans for the future work.

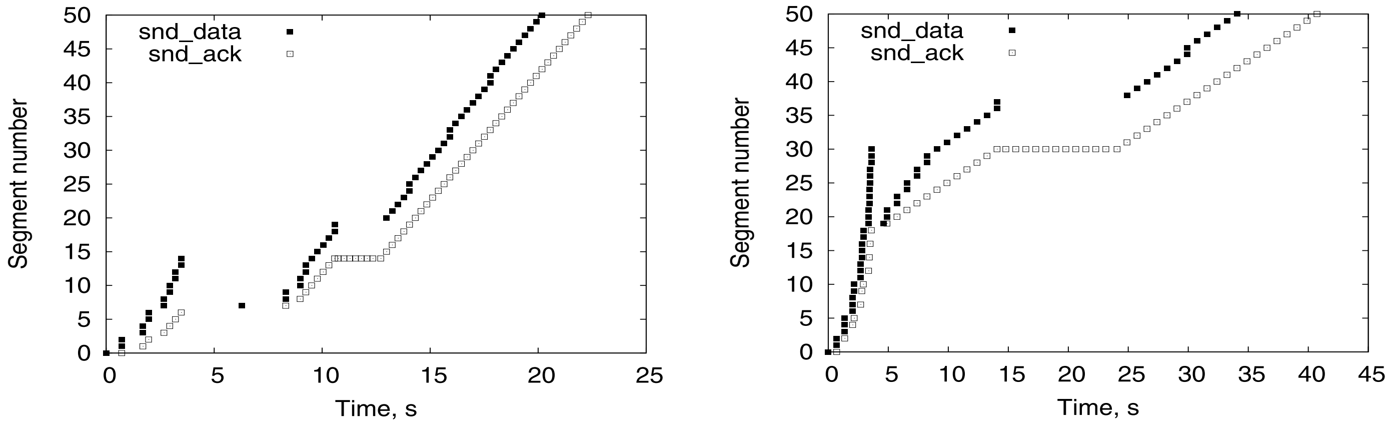
II. Spurious Timeouts in TCP

In this section, we provide a description of how spurious timeouts affect TCP’s protocol operation. We assume that the reader is familiar with the basics of TCP [44].

A. Definition of a Spurious Timeout

A *retransmission timer* is a prediction of the upper limit of the RTT. In common TCP implementations, an adaptive retransmission timer accounts for RTT variations [26]. A spurious timeout occurs when the RTT suddenly increases, to the extent that it exceeds the retransmission timer that had been determined a priori.

0-7803-7753-2/03/$17.00 (C) 2003 IEEE



(a) Triggered by a delay spike.

(b) Triggered by a bandwidth change.

Figure 1. Spurious retransmission timeouts in TCP (Reno-SACK using the timestamp option).

RTT can quickly return back to normal for example in case of a handover-triggered delay spike. RTT stays high when available bandwidth of the bottleneck link has suddenly decreased.

On a spurious timeout TCP assumes that all outstanding segments are lost and retransmits them unnecessarily as shown in Figure 1 (a)[1]. It was shown that the go-back-N retransmission behaviour triggered by spurious timeouts has a root: the retransmission ambiguity [29], i.e., a TCP sender's inability to distinguish an ACK for the original transmission of a segment from the ACK for its retransmission. Shortly after the timeout ACKs for the *original* transmissions return to the TCP sender. On receipt of the first ACK after the timeout, the sender must interpret this ACK as acknowledging the retransmission, and must assume that all other outstanding segments have also been lost. Thus, the sender enters the slow start phase, and retransmits all outstanding segments in this fashion. The go-back-N retransmission triggers the next problem: the receiver generates a DUPACK for every segment received more than once. The receiver has to do that because it must assume that its original ACKs had been lost. This may trigger a spurious fast retransmit at the sender.

Another major problem is distortion of congestion control. On one hand, the congestion window and slow start threshold are reduced after a spurious timeout. The reduction is unnecessary as no data loss has yet been detected that would indicate congestion in the network. On the other hand, TCP makes an assumption that all outstanding segments were lost and left the network. In fact, they are likely still located in the bottleneck queue. Therefore, go-back-N retransmissions performed in slow start lead to aggressive sender behaviour. That is, while the original transmissions are draining from the queue, the retransmissions get sent at twice the link rate (assuming the receiver generates an ACK for each segment). This behaviour violates the 'packet conservation' principle [26] and can cause *real* packet losses due to congestion [33]. After a spurious timeout TCP should not cause short-term congestion and should underutilize the link in the long run.

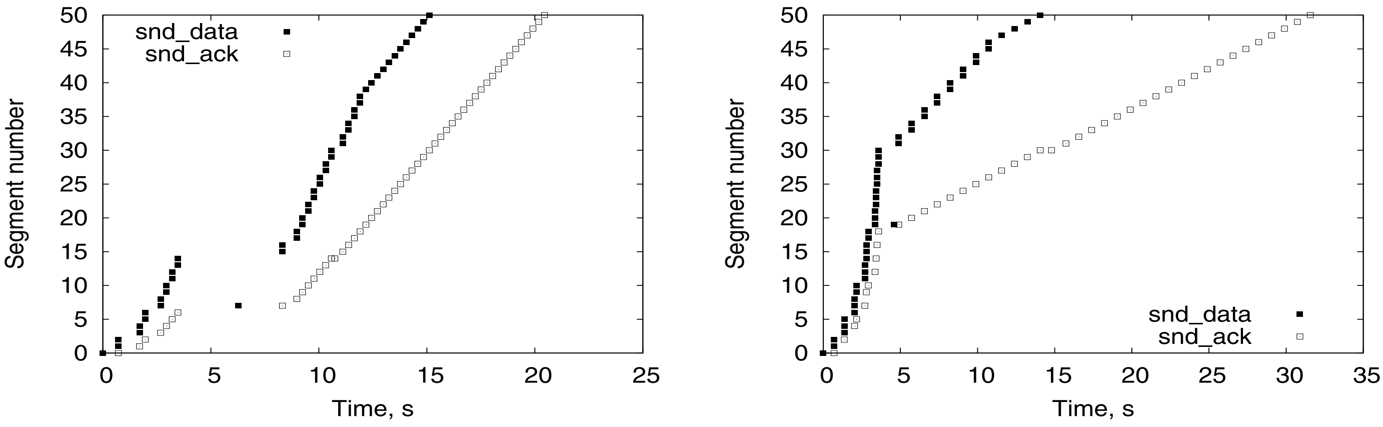
Figure 1 (b) shows a spurious timeout resulting from a bandwidth change. The available bandwidth of a bottleneck link is reduced from 300 kbps down to 10 kbps. Such rapid bandwidth changes can occur due to on-demand allocation and release of a high speed radio channel [30]. The link RTT is increased by several times which causes a spurious TCP timeout. The sender's response is largely the same as in the case of a delay spike.

B. The Eifel Detection Algorithm

Eliminating the retransmission ambiguity requires extra information in ACKs that the sender can use to unambiguously distinguish an ACK for the original transmission of a segment from that of a retransmission. The TCP timestamp option provides exactly what we need. When using the timestamp option the TCP sender writes the current value of a "timestamp clock" into the header of each outgoing segment. The receiver then echos those timestamps in the corresponding ACKs according to the rules defined in [27]. Eliminating the retransmission ambiguity is then implemented as follows. The sender always stores the timestamp of the *first* retransmission triggered by an expiration of the retransmission timer. Then, when the first ACK that acknowledges the retransmission arrives, the sender compares the timestamp of that ACK with the stored value. If it is smaller than the stored value, this indicates that the retransmission was spurious.

A case when a timeout occurs due to lost ACKs has been a subject of some discussion. When receiving a duplicate segment below the cumulative ACK some TCPs update a cached timestamp [10], and some do not [27]. If a TCP sender receives the timestamp from the original segment after a timeout, it deduces that the timeout was spurious. Therefore, if the receiver echoes the original timestamp in response to duplicate segments as the current standard defines [27], then a timeout due to lost ACKs is considered spurious. Restoring the congestion control state in this situation is partly justified; there is no loss and therefore no congestion in the forward path. Ideally, TCP should implement

[1] The trace is recorded in NS2 over a 30 kbps link.



(a) Triggered by a delay spike.

(b) Triggered by a bandwidth change.

Figure 2. TCP sender response to a spurious timeout with the Eifel algorithm.

some mechanism to reduce the amount of generated ACKs to alleviate congestion in the reverse path [5].

Including the 12 bytes TCP timestamp option field in every segment and ACK might seems heavy weight. The advantage of using the timestamp option is that this scheme is already a proposed standard and that it is widely deployed [3]. Existing TCP/IP header compression schemes [12] [28] do not support compression of TCP options, but there is ongoing work to enable it [25]. Furthermore, there is some evidence that timestamps are useful in general on bandwidth-limited paths [19].

C. Related Work

In [18] we evaluated performance of the original Eifel response [33] for a mobile user in a General Packet Radio Service (GPRS) wide-area cellular network [46]. We used simulation to obtain a controllable environment and reference TCP implementations. We also confirmed the results with smaller scale tests in a live GPRS network [20]. In the lossless scenario, the Eifel algorithm improved the performance for all TCP flavours under varying frequency of delay spikes. It reduced download times by up to 12 percent, and increased goodput by up to 20 percent. Even such moderate gains are valuable on a slow GPRS link. In the GPRS case improvement comes from eliminating duplicate data delivery. Unnecessary reduction of the congestion window is not important because of a small bandwidth-delay product of GPRS links. In this paper, we evaluate Eifel on high capacity links where unnecessary reduction of the congestion window has greater impact. Another result of [18] was that in a scenario with congestion losses Eifel can suffer from genuine timeouts, but using advanced loss recovery algorithms such as Reno-SACK [38][8] and Limited Transmit [4] improves the situation.

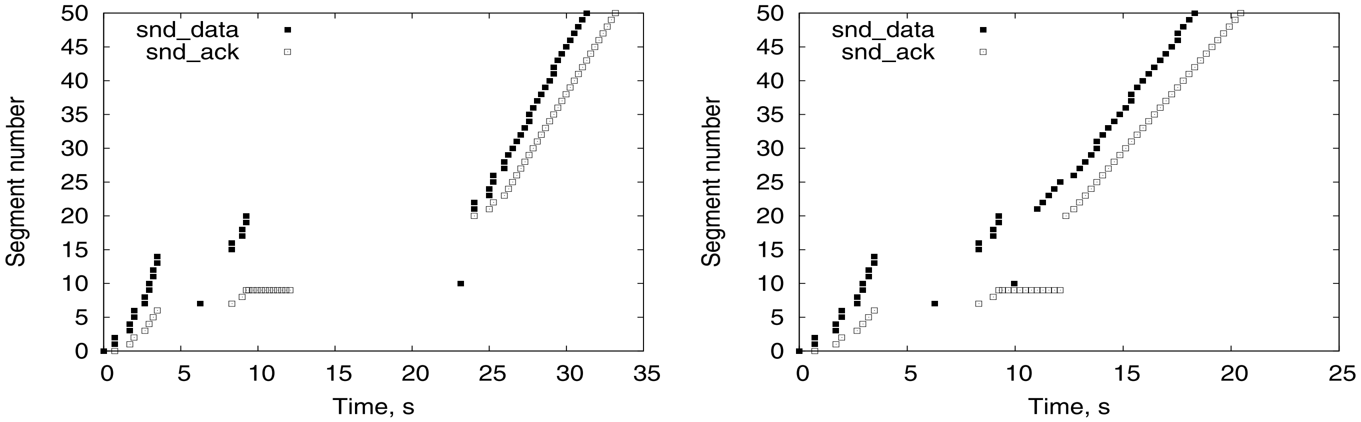
A study of cdma-2000 reports a possibility of "bandwidth oscillation" with certain configuration options from the standard IS-2000 Rel.A [30]. Due to switching of a high-speed radio channel between several users link RTT increases above the estimate of the TCP retransmission timer triggering spurious TCP timeouts. A further study reports that increasing the TCP window helps to decrease the number of spurious timeouts [49]. It is achieved with larger network buffers and a larger TCP receiver window.

Using the timestamp option is not the only possible way to detect spurious timeouts. For instance, a following heuristic was suggested in [2]. Whenever a TCP retransmits due to a timeout, it measures T, the time from the retransmission until the next ACK arrives. If T is less than the minimum RTT measured so far, then arguably the ACK was already in transit when the retransmission occurred, and the timeout was spurious. If the ACK only comes later than the minimum RTT, then likely the timeout was genuine. The response algorithm described in this paper can be applied also with other detection algorithms.

Waiting for the receiver to signal in DUPACKs that is has correctly received duplicate segments, as proposed in [15], would be too late to prevent the unnecessary retransmissions during the go-back-N behaviour. However, this information can be used for restoring congestion control state afterwards and for adapting the retransmit timer.

A "Forward RTO Recovery" (F-RTO) algorithm [42] for recovering from TCP timeouts is a TCP sender only algorithm that does not require any TCP options to operate. After retransmitting the first unacknowledged segment triggered by a timeout, the F-RTO algorithm at a TCP sender monitors the incoming acknowledgements to determine whether the timeout was spurious and to decide whether to send new segments or retransmit unacknowledged segments. The algorithm starts by transmitting new segments after a timeout and reverts to standard go-back-N behaviour only if a DUPACK is received. Otherwise, the timeout is considered spurious and the sender continues transmitting new data. F-RTO cannot properly classify timeout under packet reordering [6] or when no new data is available for transmission. In such cases it uses the standard TCP behaviour.

The Eifel algorithm does not concern with spurious timeouts that occur during loss recovery following a fast retransmit. In [19][22] we discuss restarting the timer on DUPACKs and using a method for recovering lost retransmissions as protection against spurious timeouts during a DUPACK series.



(a) Fast retransmit is blocked [14].

(b) Fast retransmit is allowed.

Figure 3. Response of TCP Reno with Eifel to a spurious timeout. A single delayed segment is lost.

III. The Sender's Response

When a timeout occurs, the Eifel algorithm at the sender stores the current values of the slow start threshold and the congestion window. Upon detecting a spurious timeout, the sender can restore them and resume transmission with the next unsent segment as shown in Figure 2. This section outlines enhancements to this basic response algorithm proposed in [33].

A. Efficient Recovery from Packet Losses

The original Eifel proposal simply specified that the transmission after detecting a spurious timeout always resumes with the next unsent segment [33]. This works fine when none of the delayed segments are lost. In reality, delay spikes are often coupled with data losses, for instance during a handover [21]. In the extreme case, all but the oldest outstanding segment are lost. Simply transmitting new data in this case leads to a second genuine timeout. In such a case recovery using standard go-back-N retransmissions would be faster[2]. However, it is difficult to select a transmit policy on a first ACK after a timeout since there is no information available on the amount of lost data. Therefore, we still believe in resuming transmission with the next unsent segment while relying on efficient loss recovery algorithms to cope with data losses.

It is well recognized that TCP Reno usually experiences a timeout when multiple segments are lost from the same window [13]. Reno with Eifel is not an exception; when several of delayed packets are lost, the timeout is inevitable. In [18] we show that allowing fast retransmits below the recovery point [14], using Limited Transmit [4] and Reno-SACK [38][8] largely solves the problem of poor performance with packet losses. In this section we suggest even more robust recovery methods.

A single lost segment. We begin with this simple case illustrating the response when one of delayed segments is lost. Understanding of it is important also for more advanced recovery schemes discussed in the rest of this section; these schemes still need three DUPACKs to enter the loss recovery phase.

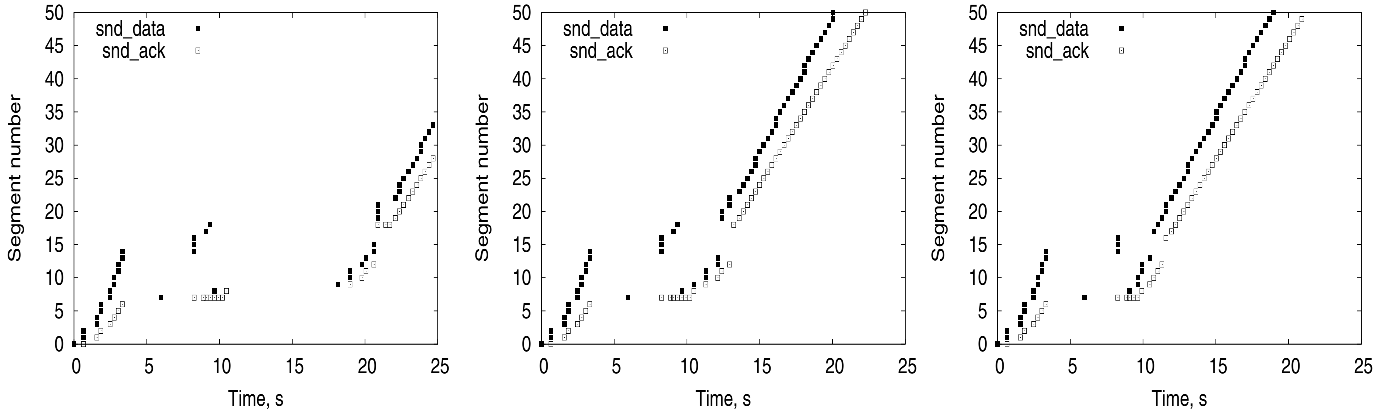
Figure 3 (a) illustrates TCP Reno that blocks fast retransmit until the recovery point[3] is acknowledged. The TCP sender has to wait for a second genuine timeout to recover this lost segment. The RTO value is large as it is calculated from delayed segments (and even may still be backed-off). The reason to block fast retransmits in conventional TCP is a possibility of a DUPACK series from unnecessarily retransmitted segments during go-back-N [14]. However, the transmission is resumed by the Eifel algorithm with the next unsent segment. There are no unnecessary retransmissions and thus a DUPACK series can only indicate a lost segment. There is no reason to block fast retransmit in such a case. As Figure 3 (b) shows, Reno successfully recovers from a lost segment with fast retransmit when allowed to do so.

Loss of all but one segment. A worst-case spurious timeout occurs when all outstanding segments are lost except for the oldest segment that is delayed. It is an example of a case when Reno-SACK often cannot recover without a genuine timeout. Figure 4 (a) illustrates the case when segments 9 to 15 are deliberately dropped. In summary, Reno-SACK cannot often avoid the genuine timeout when there are large 'holes' in the receiver window or a few ACKs were lost. The Reno-SACK scheme is conservative because it considers segments reported missing by the receiver to be still outstanding in the network. The cost is that the sender cannot retransmit segments as it is limited by the congestion window.

The Forward Acknowledgment algorithm [37], on the other hand, assumes that missing packets left the network. This often allows for a faster recovery than with Reno-SACK. In Figure 4 (c) TCP with the FACK algorithm recovers efficiently from packet losses. FACK is not standardized by IETF due to concerns with operation in presence of packet re-ordering. Linux

[2] It is also more aggressive and breaks the principle of packet conservation.

[3] snd_max, the highest sequence number transmitted before the timeout



(a) Reno-SACK.

(b) NewReno-SACK.

(c) FACK.

Figure 4. Response of TCPs with Eifel to a spurious timeout. All delayed segment but one are lost.

TCP uses FACK by default but disables it when packet reordering is detected [43].

A question is whether genuine timeouts for Reno-SACK could be avoided with a simple modification retaining the principles of conservative recovery. We found that a major part of genuine timeouts of Reno-SACK is due to lack of retransmissions on partial ACKs. NewReno [24] retransmits a packet on every partial ACK and is widely used in the Internet [39]. We combined the NewReno and SACK so that the sender always retransmits at least one segment on a partial ACK[4]. These retransmissions are accounted into the pipe estimate and therefore in the long run NewReno-SACK should be fair to other TCP flows. Although we believe it is a safe modification for general use, as an extra precaution it is possible to enable it only after a spurious timeout and disable it when the recovery point is acknowledged.

In Figure 4 (b) NewReno-SACK recovers lost packets without experiencing a genuine timeout. On a first partial ACK which arrives at 11 s NewReno-SACK retransmits a segment. Reno-SACK in Figure 4 (a) has to remain silent on this partial ACK because the estimated number of segments in flight is larger than the congestion window. As a result, the go-back-N recovery has to be used at 18 s.

B. *Restoring the Congestion Control State*

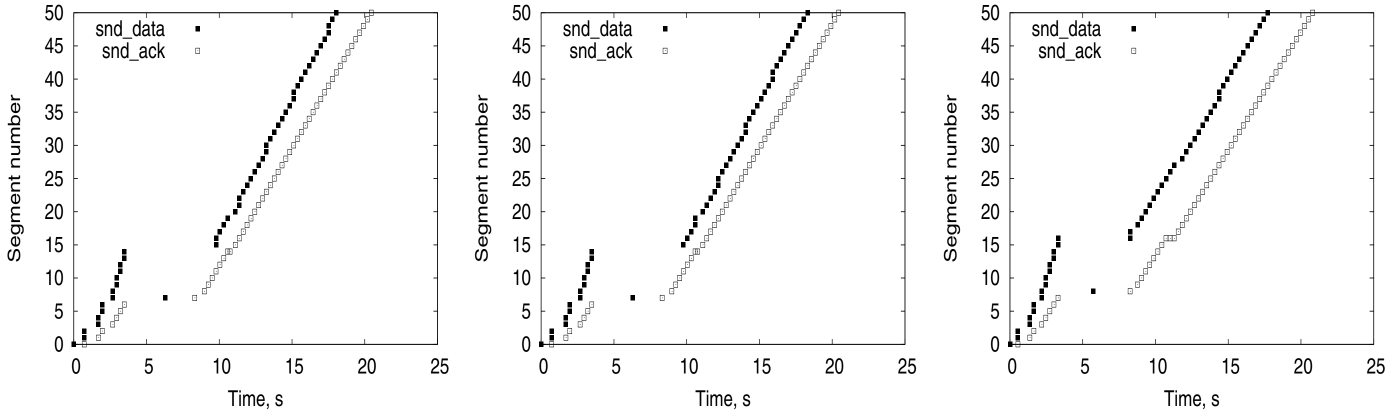
In Section II.A we described problems with congestion control experienced by conventional TCP after a spurious timeout. We explain how these problems can be resolved.

The original paper [33] proposed the following options for restoring the congestion control state:

1. ssthresh=ssthresh_old, cwnd=cwnd_old
2. ssthresh=cwnd_old/2, cwnd=ssthresh
3. ssthresh=cwnd_old/2, cwnd=1

The first option, complete restoration, is to set the slow start threshold and the congestion window to values stored before the timeout. The second option, partial restoration, is to set the slow start threshold to the half of the old congestion window (as done normally by TCP). However, instead of leaving the congestion

[4] A TCP combining NewReno and SACK is also mentioned in [5].



(a) ssthresh=cwnd_old/2, cwnd=ssthresh.

(b) ssthresh=cwnd_old/2, cwnd=1.

(c) ssthresh=cwnd_old, cwnd=ssthresh.

Figure 5. Different options of restoring the congestion control state.

window at one segment after the timeout, it is set to the new value of the slow start threshold. The third option is not to restore the congestion control state, i.e. set the slow start threshold to the half of the old congestion window and the congestion window to one segment.

The first option was used only after a single spurious timeout. The second option was used after two subsequent timeouts, and the third option was used after three or more timeouts. So far we have not found any practical evidence that the length of a delay reflects the amount of change in network characteristics. Therefore, we do not make our response algorithm depend on the number of subsequent timeouts.

Figure 2 (a) shows using option 1 after a spurious timeout. A question was raised whether restoring the congestion control state after a spurious timeout can cause undesirable bursty TCP behaviour. This is not the case[5] because the Eifel algorithm resumes transmission with the next unsent segment (snd_nxt=snd_max) which also restores the estimate of the flight size [1][48]. As TCP only transmits data when the congestion window is greater than the flight size, no burst is produced when both parameters are restored from the equilibrium state. However, we emphasize that TCPs such as Linux 2.4 [43] which determine the flight size in a different way than BSD TCP must explicitly restore it after a spurious timeout.

In practice, options 2 and 3 do not perform well. If the congestion window is not restored fully, the sender cannot transmit on original ACKs after a spurious timeout as shown in Figure 5 (a) and (b) because the flight size estimate is larger than the congestion window. Additionally, lack of restoring of the slow start threshold can cause significant unnecessary underutilization of a link if a spurious timeout occurs in an early phase of slow start. Experiments in Section IV indicate that with options 2 and 3 the TCP sender is prone to genuine timeouts which decreases throughput. Furthermore, the load on the network is actually increased due to a greater number of unnecessary retransmissions. Given this fact and that full restoring of the congestion control state does not cause bursts, applying the option 1 seems to be an attractive choice.

We did not find much difference between the option 2 and 3 for links with a moderate delay-bandwidth product. With the option 2 the sender continues in congestion avoidance incrementing the congestion window only by a single segment per window. With the option 3, the sender is in slow start and increments the congestion window by one segment per ACK. Therefore, it reaches the same size as used by the option 2 quickly.

We suggest a forth option to restore the congestion control state

4. ssthresh=cwnd_old, cwnd=ssthresh,

where the slow start threshold is set to the old value of the congestion window, and the congestion window is fully restored. This allows the sender to immediately resume transmission on ACKs as shown in Figure 5 (c). However, the sender is forced to continue in congestion avoidance which may lead to underutilization on high-delay bandwidth paths. A variation of this approach would restore the slow start threshold but only when no loss has yet been detected during the connection.

Different other options for restoring the congestion control state are possible. For example, the slow start threshold could be set to the old value of the congestion window. We have not found any of such options to be particularly good and therefore do not include them in this paper.

So far we have discussed the situation when a spurious timeout occurs during a slow start. The response when a spurious timeout occurs in congestion avoidance is similar.

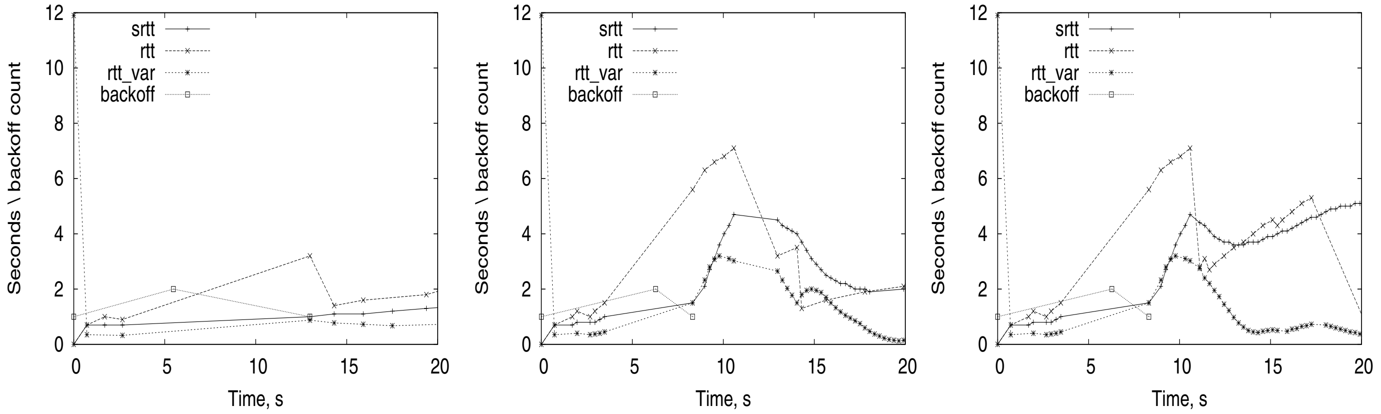
C. *Adapting the Retransmit Timer*

With traditional TCP, a sender that uses a too aggressive retransmit timer has to pay the price (i.e. slow down) after a spurious timeout. Presumably this discourages developing too aggressive retransmission timers and preserves the network from duplicate retransmissions that do no useful work. Therefore, some modification to the retransmit timer that makes it more conservative after a spurious timeout is needed. This section discusses various approaches to adapting the retransmit timer after a spurious timeout. Note that in order to increase conservativeness of the retransmit timer, the TCP sender must be robust to packet losses. Otherwise, the sender will suffer excessively from waiting for genuine timeouts. TCP FACK and NewReno-SACK suggested in Section III.A seem to be sufficiently robust to packet losses.

Figure 6 shows RTO parameters of TCP Reno, Reno with timestamps, and Reno with timestamps and with the Eifel algorithm after a spurious timeout. In Figure 6 (a) the RTO parameters of Reno without timestamps remain nearly at the same level as before the timeout. In other words, TCP does not learn much from a delay spike. This situation is explained by the Karn's algorithm as collecting of RTT samples from retransmitted segments is denied due to the retransmission ambiguity problem [29]. Therefore, during the go-back-N behaviour no RTT samples can be collected, but the RTO is kept backed off. A spurious fast retransmit present in some TCPs after go-back-N can even further delay obtaining a valid RTT sample. Once a new RTT sample is collected, SRTT and RTTVAR are recalculated from the new sample and the back off counter is reset. The RTO value basically returns at the level before the delay spike.

Figure 6 (b) shows behaviour of RTO parameters for Reno with timestamps. It is less aggressive than RTO computed without timestamps due to using the delayed segments for RTT sampling. Immediately after a timeout when original ACKs are arriving, the RTO becomes very high. Lack of updates in the graph between 10-13 s is due to arriving DUPACKs which cannot be used for RTT sampling [27]. The RTO stabilizes at the new level approximately 10 s after the spurious timeout. This is likely a too quick decrease to protect the sender from spurious timeouts in the future. Making SRTT and RTTVAR weights

[5] With exception of bursts due to ACK losses and compression which are intrinsic to TCP.



(a) Reno.

(b) Reno with timestamps.

(c) Reno with timestamps and Eifel.

Figure 6. RTO dynamics after a spurious timeout.

adaptive to the frequency of RTT sampling as suggested in [34] can solve this problem.

Figure 6 (c) shows the RTO dynamics of Reno with Eifel. The retransmit timer naturally uses timestamps since they are required for the Eifel detection algorithm. Already this fact makes TCP with Eifel more conservative than widely used Reno without timestamps. Furthermore, the idle period in RTO updates due to DUPACKs such as in Figure 6 (b) is not present here. The RTO with Eifel does not decrease as quickly after the timeout. This is because Eifel restores the congestion control state and gets more data outstanding in the network. Higher RTT in such a case makes the timer less prone to spurious timeouts [30].

In summary, using a conservative RTO such as suggested in [40][6] with timestamps provides a sufficient protection against excessive spurious timeouts in many cases.

Further adapting the timer may include the following options:

1. Re-seed the RTO after a spurious timeout
2. Reset the back-off counter only on a genuine timeout
3. Increase the minimum RTO

The first option is to use an RTT sample obtained with timestamps from delayed segments to re-initialize SRTT and RTTVAR variables and restart the timer. The second option is to keep the back-off counter at the level set during a spurious timeout and reset it only on a genuine timeout. The third option is to perform additive increase of the minimum RTO value on each spurious timeout and reset it to the default value on a genuine timeout.

Further possible option could be to increase the minimum RTO value based on a length of a delay or its experientially smoothed average. We have not evaluated such options; a study on adapting of the DUPACK threshold shows that they do not work particularly well [7]. It is also possible to exploit different options for reducing the minimum RTO value, such as halve it on every genuine timeout instead of simply resetting it to the default value.

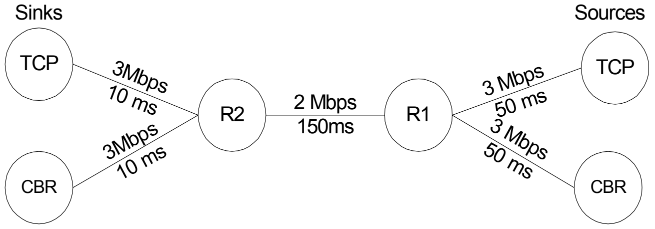


Figure 7. Measurement setup in NS2.

IV. Performance Evaluation

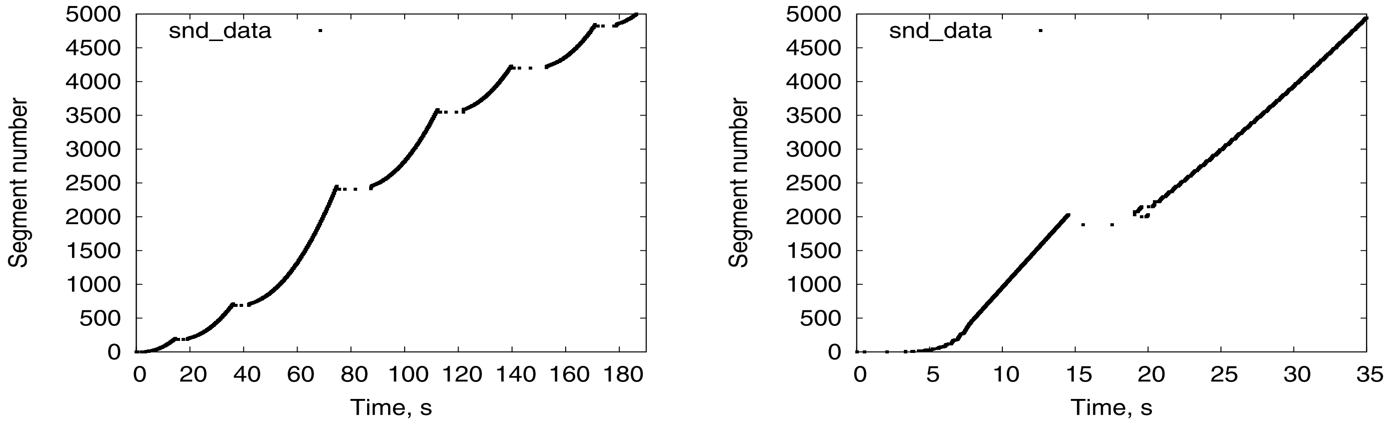
A. *Methodology*

We choose simulation to have a reproducible and controllable environment with reference TCP implementations. A simple 'dumb bell' topology has a bottleneck link with 2 Mbps and with high latency of 150 ms. Such characteristics are typical for satellite links and the third generation wireless wide area networks [46]. In all measurements the TCP timestamp option [27] was enabled and the MSS was 1000 bytes. The receiver advertised a window of 150000 bytes and the bottleneck queue was DropTail with the maximum queue length of 75 packets. We used one-way models of Reno-SACK, NewReno-SACK, and FACK TCP with delayed acknowledgments. Our modified Eifel response algorithm is implemented for NS2.1b9a and is publicly available [23].

We also implemented a tool to trigger delay spikes over a simulated link. It suspends transmission in both directions simulatenously. The length of delay spikes is uniformly distributed between 3 and 15 s; they occur at the interval of 20-40 s. Shorter delay spikes would suffice to trigger spurious timeouts in our tests, but we decided to use the typical values experienced by a cellular network user driving in an urban area [21]. In all tests a conservative timer [40] is applied.

Tests summarized in Table 1 use a single TCP connection transferring 5 MB of data. In Table 2, Table 3, and Table 4 a competing constant bit rate flow is added running at 1 Mbps. It

[6] This timer is restarted on ACKs and uses the minimum RTO value of 1 s.



(a) Reno-SACK.

(b) Reno-SACK with Eifel.

Figure 8. Effect of Eifel on Reno-SACK on a uncongested link.

congests the link especially during delay spikes. Adaptation of the retransmit timer is only performed in Table 4. In Table 1, Table 2 and Table 4 the congestion control state is completely restored (option 1). All values are averaged over 100 repetitions.

B. *Results*

For mobile users and operators the battery power consumption and radio resource preservation are often as important as the throughput across the wireless link. We therefore used the download time and the total number of transmitted segments as equally important performance metrics. We also give the average number of spurious and genuine timeouts for each connection to indicate how susceptible a TCP modification is to timeouts.

In the first test, we use Reno-SACK over a link without other traffic. Table 1 shows results with and without Eifel. Applying the Eifel algorithm gives 254% increase in throughput and at the same time requires 3% less segments to complete the connection. Most of the improvement in throughput comes in this case from restoring of the congestion control state after spurious timeouts. Figure 8 (a) shows that Reno-SACK reduces the congestion window and performs go-back-N retransmissions on every spurious timeout. Enabling the Eifel algorithm in Figure 8 (b) allows the connection to increase the congestion window until a segment loss is detected at 20 s. The number of spurious timeouts is decreased due to shorter connection lifetime.

TABLE 1. EFFECT OF EIFEL ON RENO-SACK ON A UNCONGESTED LINK.

TCP	Eifel	Time, s	Segments sent	Spurious RTOs	Genuine RTOs
Reno-SACK	Off	138	5234	4.68	0.00
Reno-SACK	On	39	5088	1.37	0.03

Table 2 shows results for Reno-SACK over a congested link with the same delay jitter model. It is a challenging scenario for Eifel, as often many segments are lost during a delay spike. Reno-SACK with Eifel has 73% longer download time in this case due to a large number of genuine timeouts. Timeouts typically occur when the TCP sender enters the fast recovery phase but cannot retransmit lost segments due to large ‘holes’ in the receiver window. Figure 9 (a) shows three such timeouts at 100 s, 200 s, and 250 s. NewReno-SACK corrects this problem by recovering at least one segment per RTT and allows Eifel to achieve higher throughput and goodput. In Figure 9 (b) no genuine timeouts are present. FACK with Eifel achieves 43% reduction in download time over Reno-SACK even in such harsh conditions. Figure 9 (c) shows that FACK avoids genuine timeouts and recovers from packet losses faster than NewReno-SACK

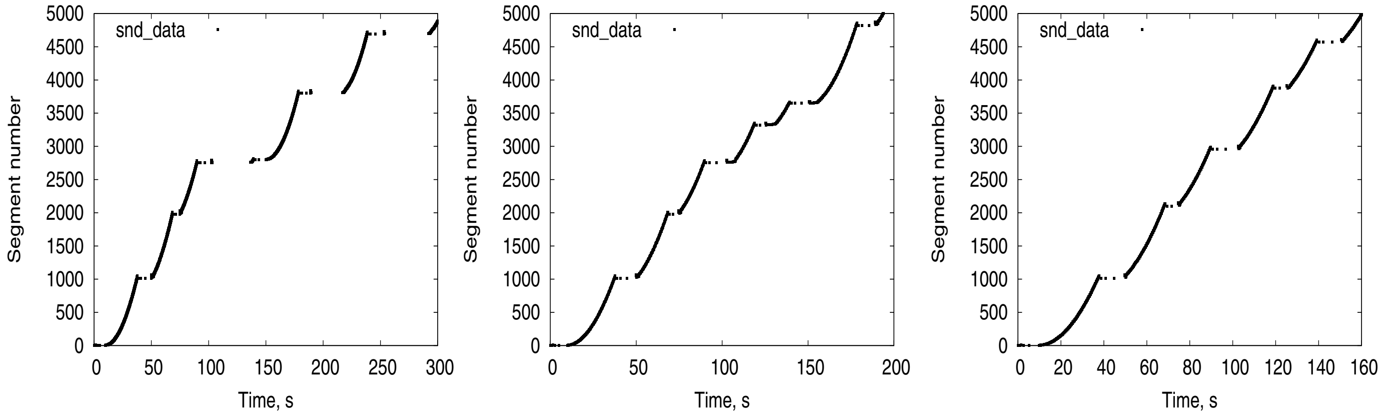
TABLE 2. EFFECT OF EIFEL ON TCPS OVER A CONGESTED LINK.

TCP	Eifel	Time, s	Segments sent	Spurious RTOs	Genuine RTOs
Reno-SACK	Off	191	5251	5.79	0.68
	On	331	5237	6.02	4.98
NR-SACK	Off	191	5251	5.78	0.69
	On	146	5192	4.35	0.57
FACK	Off	191	5251	5.74	0.70
	On	108	5225	3.24	0.38

Table 3 shows performance of FACK with Eifel with different options (described in Section III.B) of restoring the congestion control state. Options 2 and 3 perform poorly in terms of throughput and goodput. Not only the download time is several times higher than for the option 1, but also more unnecessary retransmissions are sent wasting the network capacity. The option 4 achieves close to the same throughput as the option 1. Thus, the option 4 can be used by a careful sender which does not want the more aggressive option 1.

TABLE 3. FACK WITH EIFEL ON A CONGESTED PATH WITH VARYING RESTORATION OF THE CONGESTION CONTROL STATE.

CC restore	Time, s	Segments sent	Spurious RTOs	Genuine RTOs
option 1	108	5225	3.24	0.38
option 2	540	5325	8.48	8.43
option 3	912	5558	11.19	14.68
option 4	109	5226	3.26	0.38



(a) Reno-SACK with Eifel.

(b) NewReno-SACK with Eifel.

(c) FACK with Eifel.

Figure 9. Effect of Eifel on TCPs over a congested link.

Table 4 shows results of experiments with adapting the retransmission timer. Re-seeding the timer with a new sample after a spurious timeout does not have any effect in this scenario. The RTO is already high after a timeout but re-reseeding it does not help to prevent its fast descent. Using the back-off approach reduces the number of spurious timeouts by 40% with only a small decrease in throughput. This might be an attractive option to use. Increasing the minimum RTO is slightly less effective in this scenario than using the back-off counter.

TABLE 4. FACK WITH EIFEL ON A CONGESTED PATH WITH DIFFERENT RTO ADAPTATION TECHNIQUES.

RTO adapt.	CC restore	Time, s	Segments sent	Spurious RTOs	Genuine RTOs
std	option 1	109	5225	3.24	0.38
reseed		109	5225	3.24	0.38
back-off		113	5166	1.92	0.40
min++[a]		114	5168	2.41	0.43

a. the minimum RTO value is incremented by 1 s after each spurious timeout and reset to 1 s on a genuine timeout.

The effect of proposed RTO adaptation methods could be different for other delay scenarios. For example, when the link bandwidth oscillates the delay jitter typically only slightly exceeds the RTO. In the scenario we have studied, RTO is exceeded significantly. Finally, adaptation techniques could be more effective if learnt characteristics of the path would be shared between TCP connections to the same destination [45].

C. *Discussion*

We believe that TCP with the Eifel algorithm is friendly to other TCPs as the basic congestion control mechanisms triggered on a packet loss are not modified. Even if the congestion window and the slow start threshold are fully restored after a spurious timeout, they are reduced again after detecting a packet loss. TCP with the Eifel algorithm gains the capacity underutilized by other TCPs.

The experiments were also executed with setting Adaptive RED [17] with automatic configuration of parameters as the bottleneck queue instead of Drop-Tail. The conclusions made based on the Drop-Tail measurements still hold and Eifel showed equal or better performance. However, in general TCP throughput was from slightly to many times lower than in case of a Drop-Tail queue. We interpret this as an artifact of our test setup with a low degree of statistical multiplexing and presence of a competing constant bit rate flow unresponsive to congestion.

We made typical modelling assumptions that TCP connections are long-lived and there is no congestion in the opposite direction. Determining the extend to which these assumptions hold in the Internet is a hard problem [16]. Wide-scale Internet measurements of TCP with the Eifel algorithm would be useful, but they are difficult to obtain, share and reproduce.

Formally assessing performance gains of applying the Eifel algorithm is difficult as the result depends on many factors [33]. It could be from nothing to several hundred percent depending on the frequency of delay spikes, path characteristics, the retransmission timer and type of workload. The best case for Eifel occurs when one of the segments in the initial window experiences a spurious timeout on a high-delay bandwidth path. In such a case, the TCP connection stays in congestion avoidance and is likely to use only a small fraction of the available bandwidth. A TCP connection with the Eifel algorithm will continue the slow start until a segment loss indicating a real need to slow down.

The Eifel algorithm is robust to packet losses caused by data corruption, but does not perform more aggressively than traditional TCP as it still relies on a segment loss as an indication of congestion.

V. CONCLUSION AND FUTURE WORK

This paper shows that in a broadband environment applying the Eifel algorithm can give up to 250% increase in throughput and at the same time decrease the load on the network by 3%. In a scenario with heavy congestion, TCP with Eifel suffers from genuine timeouts even with Reno-SACK and Limited Transmit.

The original response [33] could be further improved as follows. Eifel performs well in combination with FACK, but it may not be always used due to concerns in presence of packet reordering. Therefore, we suggested combining the NewReno and SACK algorithms in a single TCP. NewReno-SACK avoids retransmission timeouts present for Reno-SACK due to large 'holes' in the receiver window. Eifel with NewReno-SACK works well even under heavy packet losses and is presumably safe to use in the Internet.

We showed that full restoration of the congestion control state does not cause bursty behaviour. Furthermore, partial or lack of restoring of the congestion window reduces the throughput and loads the network with a greater number of unnecessary retransmissions. This is because if only the flight size is restored the sender cannot transmit segments on arriving ACKs. It makes the sender prone to genuine timeouts. We suggested a new option for partially restoring of the congestion control state which seems to perform as well as full restoration but is more conservative.

We studied a number of techniques for adapting the RTO to avoid further spurious timeouts. TCP with the Eifel algorithm uses samples from delayed segments to update RTO. This alone provides a more conservative timer than TCP Reno without timestamps. However, additional methods for learning from a spurious timeout may be desirable. Re-seeding the timer with a new sample is ineffective in the scenario we used. Increasing the exponential back-off counter decreases the number of spurious timeouts by 40% with only a small decrease in throughput. Increasing the minimum RTO works slightly worse than the back-off method. Therefore, it is reasonable to implement one of the latter techniques with the Eifel algorithm. However, either FACK or NewReno-SACK are required at the same time to avoid low throughput due to a large number of genuine timeouts.

In future work we plan to evaluate behaviour of other retransmit timers, such as the Eifel timer [34] in presence of highly variable delays.

Acknowledgments

We thank Mark Allman, Sally Floyd, Pasi Sarolahti, Alexey Kuznetsov, Farid Khafizov and the anonymous reviewers for constructive criticism and helpful suggestions.

References

[1] M. Allman, V. Paxson, W. Stevens, *TCP Congestion Control*, RFC 2581, April 1999.

[2] M. Allman, V. Paxson, On Estimating End-to-End Network Path Properties, *ACM SIGCOMM*, September 1999.

[3] M. Allman, A Web Server's View of the Transport Layer. *ACM Computer Communication Review*, vol. 30(5), October 2000.

[4] M. Allman, H. Balakrishnan, and S. Floyd, *Enhancing TCP's Loss Recovery Using Limited Transmit*, RFC 3042, January 2001.

[5] H. Balakrishnan, Challenges to Reliable Data Transport over Heterogeneous Wireless Networks, PhD Thesis, University of California at Berkeley, 1998.

[6] J.C.R. Bennett, C. Partridge, N. Shectman, Packet Reordering is Not Pathological Network Behavior, *IEEE/ACM Transactions on Networking*, December 1999.

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

[8] E. Blanton, M. Allman, K. Fall, L.Wang, *A Conservative SACK-based Loss Recovery Algorithm for TCP*, draft-allman-tcp-sack-13.txt, work in progress, October 2002.

[9] J. C. Bolot, Characterizing End-to-End Packet Delay and Loss in the Internet, *Journal of High Speed Networks*, vol. 2(3), September 1993.

[10] R. Braden, *TCP Extensions for High Performance: An Update*, unpublished, http://www.kohala.com/start/tcplw-extensions.txt, June 1993.

[11] CAIDA, *Traffic Workload Overview*, http://www.caida.org/outreach/resources/learn/trafficworkload/tcpudp.xml, July 2002.

[12] M. Degermark, B. Nordgren, S. Pink, *IP Header Compression*, RFC 2507, February 1999.

[13] K. Fall, S. Floyd, Simulation-based Comparisons of Tahoe, Reno, and SACK TCP, *ACM Computer Communication Review*, vol. 26(3), July 1996.

[14] S. Floyd, T. Henderson, *The NewReno Modification to TCP's Fast Recovery Algorithm*, RFC 2582, April 1999.

[15] S. Floyd, J. Mahdavi, M. Mathis, M. Podolsky, A. Romanow, *An Extension to the Selective Acknowledgement (SACK) Option for TCP*, RFC 2883, July 2000

[16] S. Floyd, V. Paxson, Difficulties in Simulating the Internet, *IEEE/ACM Transactions on Networking*, vol 9(4), August 2001.

[17] S. Floyd, R. Gummadi, S. Shenker, Adaptive RED: An Algorithm for Increasing the Robustness of RED's Active Queue Management, unpublished, August 2001.

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

[19] A. Gurtov, *Making TCP Robust Against Delay Spikes*, University of Helsinki, Department of Computer Science, Technical Report C-2001-53, November 2001.

[20] A. Gurtov, Effect of Delays on TCP Performance, In Proceedings of *IFIP Personal Wireless Communications*, August 2001.

[21] A. Gurtov, M. Passoja, O. Aalto, M. Raitola, Multilayer Protocol Tracing in a GPRS Network, In Proceedings of the *IEEE Vehicular Technology Conference*, September 2002.

[22] A. Gurtov, On Treating DUPACKs in TCP, draft-gurtov-tsvwg-tcp-delay-spikes-01.txt, work in progress.

[23] A. Gurtov, *Implementation of the Eifel algorithm for NS2*, http://www.cs.helsinki.fi/u/gurtov/ns, December 2002.

[24] J. Hoe, Improving the Start-up Behavior of a Congestion Control Scheme for TCP, *ACM SIGCOMM*, August 1996

[25] IETF, *Robust Header Compression*, http://www.ietf.org/html.charters/rohc-charter.html, July 2002.

[26] V. Jacobson, Congestion Avoidance and Control, *ACM SIGCOMM*, August 1988.

[27] V. Jacobson, R. Braden, D. Borman, *TCP Extensions for High Performance*, RFC 1323, May 1992.

[28] V. Jacobson, *Compressing TCP/IP Headers for Low-Speed Serial Links*, RFC 1144, February 1990.

[29] P. Karn, C. Partridge, Improving Round-Trip Time Estimates in Reliable Transport Protocols, *ACM SIGCOMM*, August 1987.

[30] F. Khafizov, M. Yavuz, Running TCP over IS-2000, In Proceedings of the *IEEE Conference on Communications*, April 2002.

[31] J. Korhonen, O. Aalto, A. Gurtov, H. Laamanen, Measured Performance of GSM HSCSD and GPRS, In Proceedings of the *IEEE Conference on Communications*, June 2001.

[32] D. Loguinov and H. Radha, Measurement Study of Low-bitrate Internet Video Streaming, In Proceedings of the *ACM SIGCOMM Internet Measurement Workshop*, November 2001.

[33] R. Ludwig, and R. H. Katz, The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions, *ACM Computer Communication Review*, vol. 30 (1), January 2000.

[34] R. Ludwig, K. Sklower, The Eifel Retransmission Timer, *ACM Computer Communication Review*, vol. 30(1), July 2000.

[35] R. Ludwig, M. Meyer, *The Eifel Algorithm for TCP*, draft-ietf-tsvwg-tcp-eifel-alg-07.txt, work in progress.

[36] R. Ludwig, A. Gurtov, *The Eifel Response Algorithm for TCP*, draft-ietf-tsvwg-tcp-eifel-response-02.txt, December 2002, work in progress.

[37] M. Mathis, J. Mahdavi, Forward Acknowledgment: Refining TCP Congestion Control, *ACM SIGCOMM,* August 1996.

[38] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, *TCP Selective Acknowledgement Option*s, RFC 2018, October 1996.

[39] J. Padhye, S. Floyd, On Inferring TCP Behavior, *ACM SIGCOMM*, August 2001.

[40] V. Paxson, M. Allman, *Computing TCP's Retransmission Timer*, RFC 2988, November 2000.

[41] J. Postel, *Transmission Control Protocol*, RFC793, September 1981.

[42] P. Sarolahti, M. Kojo, and K. Raatikainen, *F-RTO: A New Recovery Algorithm for TCP Retransmission Timeouts*, University of Helsinki, Department of Computer Science, Technical Report C-2002-07, February 2002.

[43] P. Sarolahti, A. Kuznetsov, Congestion Control in Linux TCP, In Proceedings of the *USENIX Annual Technical Conference*, June 2002.

[44] W. R. Stevens, *TCP/IP Illustrated, Volume 1 (The Protocols)*, Addison Wesley, November 1994.

[45] J. Touch, *TCP Control Block Interdependence*, RFC 2140, April, 1997.

[46] B. Walke, Mobile Radio Networks, Networking and Protocols (2. Ed.), Wiley & Sons, Chichester 2001.

[47] H. J. Wang, R. H. Katz, J. Giese, Policy-Enabled Handoffs Across Heterogeneous Wireless Networks, In Proceedings of the *2nd IEEE Workshop on Mobile Computing Systems and Applications*, February 1999.

[48] G. R. Wright, W. R. Stevens, *TCP/IP Illustrated, Volume 2 (The Implementation)*, Addison Wesley, January 1995.

[49] M. Yavuz, F. Khafizov, TCP over Wireless Links with Variable Bandwidth, In Proceedings of the *IEEE Vehicular Technology Conference*, September 2002.