CHAPTER 5

TCP-Eifel

Competing error recovery is the remaining inefficient cross-layer interaction that our solution of flow-adaptive wireless links does not solve. In the preceding chapter, we concluded that this problem has to be solved by making the respective end-to-end protocol more robust. This is the motivation for this chapter. We propose two new mechanisms for reliable end-to-end protocols, the *Eifel*[1] *algorithm* and the *Eifel retransmission timer.* We have implemented both mechanisms for TCP, and refer to that implementation as *TCP-Eifel* that we have made publicly available [Lud99c].

In Section 5.1, we show that spurious fast retransmits and the go-back-N retransmission behavior triggered by spurious timeouts have the same root: the *retransmission ambiguity* [KP87]. That is, a TCP sender is unable to distinguish an ACK for the original transmission of a segment from the ACK for its retransmission. In Section 5.2, we develop the *Eifel algorithm* that uses extra information in the ACKs to resolve the retransmission ambiguity, and thereby eliminates the problems caused by spurious retransmissions. This work is based on the methodology described in Section 3.3 and has been published in [LK00].

Subsequently, we analyze two alternative retransmission timers for TCP. In Section 5.3, we first analyze the Lite-Xmit-Timer (see Section 2.2.2), and reveal a number of problems related to its definition and implementation. This explains why we had suspected in Section 4.2 that this timer is too conservative. In Section 5.4, we then propose an alternative retransmission timer which we call the *Eifel retransmission timer (Eifel-Xmit-Timer).* Although we only focus on TCP, we believe that our conclusions also apply to other reliable end-to-end and link layer protocols. This work is based on the methodology explained in Section 3.4 and has been described in [LS99].

1. *The Eifel* is the name of a beautiful mountain range in Western Germany.

5.1 Problems of TCP-Lite's Error Recovery

In this section, we provide a detailed description of how spurious timeouts and spurious fast retransmits affect TCP's protocol operation.

5.1.1 Spurious Timeouts

A *retransmission timer* is a prediction of the upper limit of the round-trip time (RTT). In common TCP implementations, an adaptive retransmission timer accounts for RTT variations (see Section 2.2). A spurious timeout occurs when the RTT suddenly increases, to the extent that it exceeds the retransmission timer that had been determined a priori. Spurious timeouts can be due to route changes, or rapidly increasing congestion at the bottleneck link. The latter can in turn be caused by routing table updates [Pax97d] or a reliable link layer protocol running over a wireless link on which the radio quality (temporarily) dropped (see Section 4.2.4). Spurious timeouts affect TCP performance in two ways: (1) the TCP sender unnecessarily reduces its load, and (2) the TCP sender is forced into a go-back-N retransmission mode.

The fundamental problem that leads to the go-back-N retransmissions is the retransmission ambiguity [KP87], i.e., a TCP sender's inability to distinguish an ACK for the original trans-

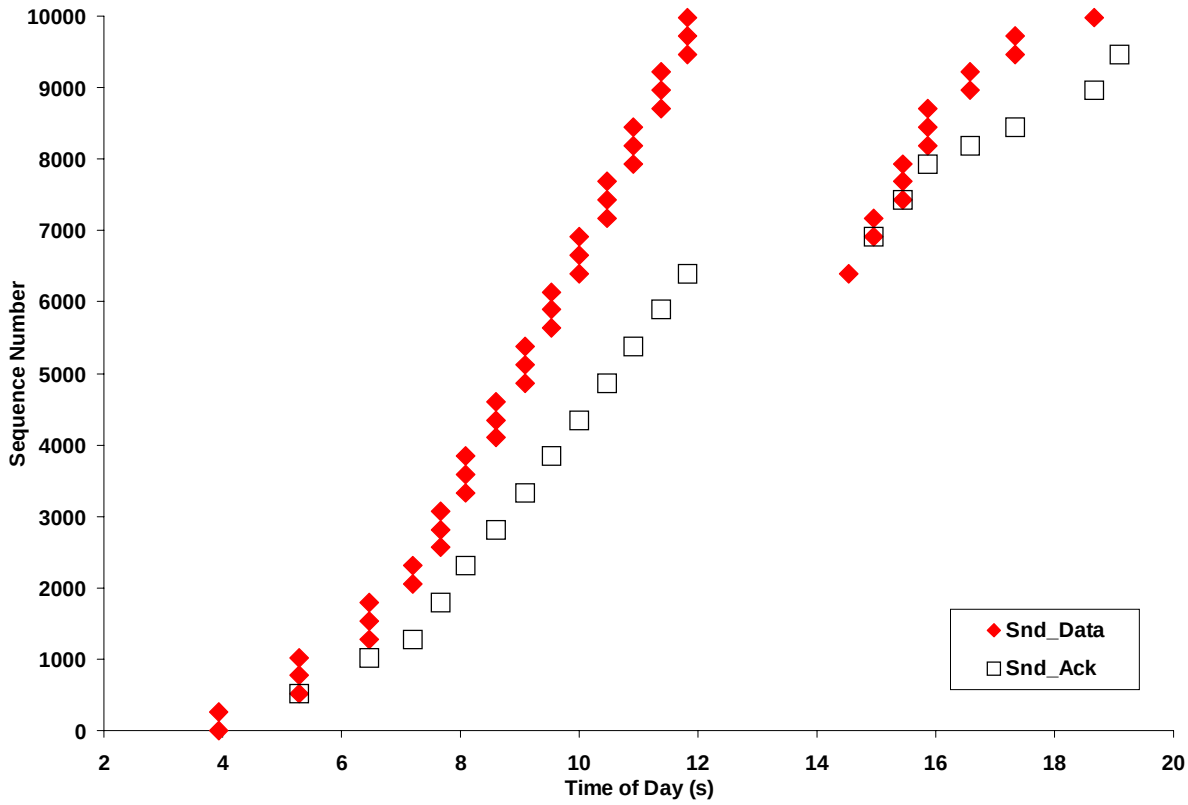

Figure 5-1: Go-back-N after a spurious timeout (sender trace).

mission of a segment from the ACK for its retransmission. This is depicted in Figure 5-1, taken from [Bau97], showing a trace plot of a spurious timeout caused by a reliable link layer protocol over a wireless link. Shortly after the timeout (14.5 s into the connection), the 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 sends (retransmits) the two segments next to the just acknowledged sequence number. These are acknowledged by the second ACK received after the timeout, which really is the ACK for their original transmissions. This continues until the entire window has been retransmitted, i.e., the sender has performed the go-back-N retransmission. Although, we do not have the receiver trace, we can tell that the ACKs returning after the timeout are really the ACKs for the original transmissions: the time between the first retransmission and the first ACK received after the timeout (about 400 ms) is smaller than the minimum possible RTT on the network on which the trace was collected.

To verify this finding, we performed bulk data transfers on the experimental network described in Section 3.3.2 and used `hiccup` to trigger a spurious timeout. This time we also traced the TCP receiver. As explained in Section 3.2.1 both the sender and receiver trace are correlated in time and shown in the same plot (see Figure 5-2).

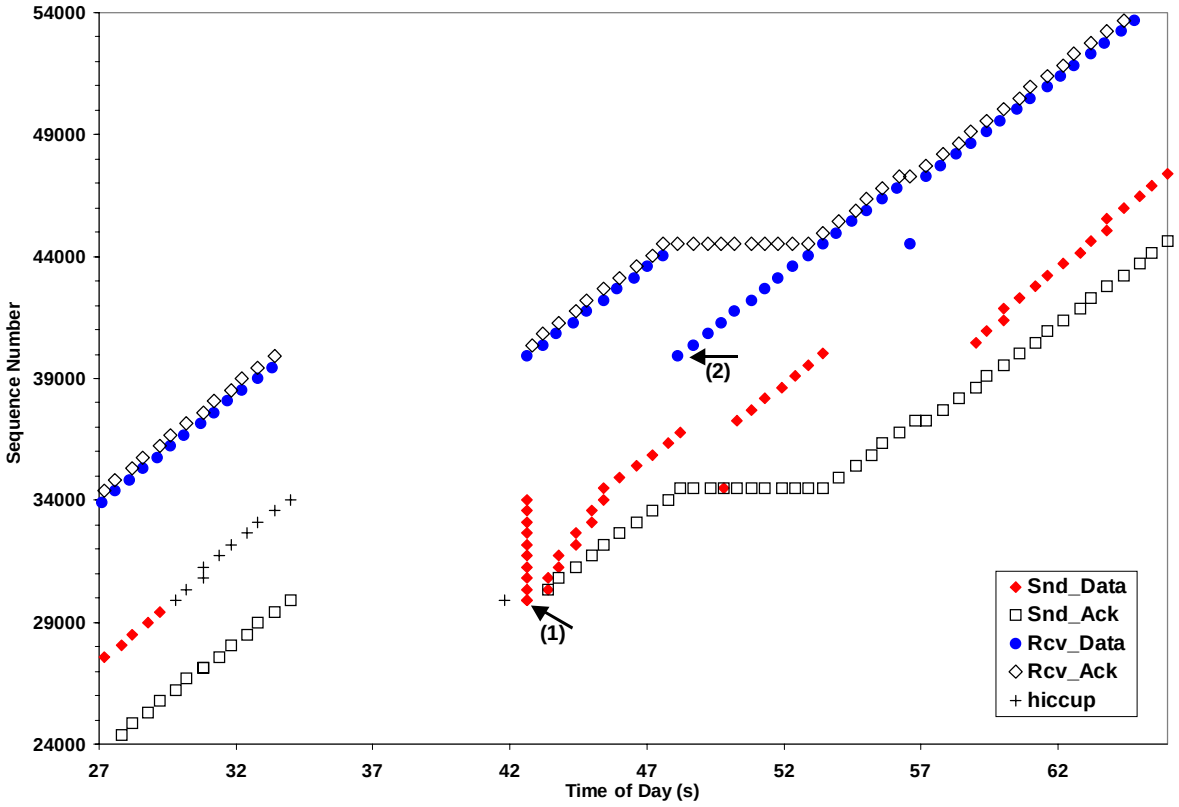

Figure 5-2: Go-back-N after a spurious timeout (sender and receiver traces)

At 29.8 s into the connection we called `hiccup` to intercept and queue outbound packets for 13 s. During this time, all outstanding segments are being acknowledged and each of those ACKs clocks out a new segment (marked as + in Figure 5-2). However, those segments are not logged by the BSD Packet Filter until `hiccup` has terminated in second 42.6, when those segments get placed into the outbound interface buffer all at once. At that time, the sender has already performed one retransmission (marked as + in Figure 5-2). This was also queued by `hiccup` and can therefore only be seen in the receiver trace (see arrow (2) in Figure 5-2). The original transmission and the retransmission of that segment are the same point in the sender trace (see arrow (1) in Figure 5-2). Then the go-back-N retransmission is triggered as described before with respect to Figure 5-1. Returning to Figure 5-2, 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 (why else would the sender send those segments again?). This triggers a spurious fast retransmit, which we describe in the next subsection.

Yet another problem is that the go-back-N retransmissions are performed in slow-start, leading to an aggressive sender behavior. That is, while the original transmissions are draining from the queue, the retransmissions get sent at twice the line speed (assuming the receiver generates one ACK for each segment). This aggravates the situation and can lead to *real* packet losses due to congestion as shown in Figure 5-3. To show this effect we set the interface buffer size to

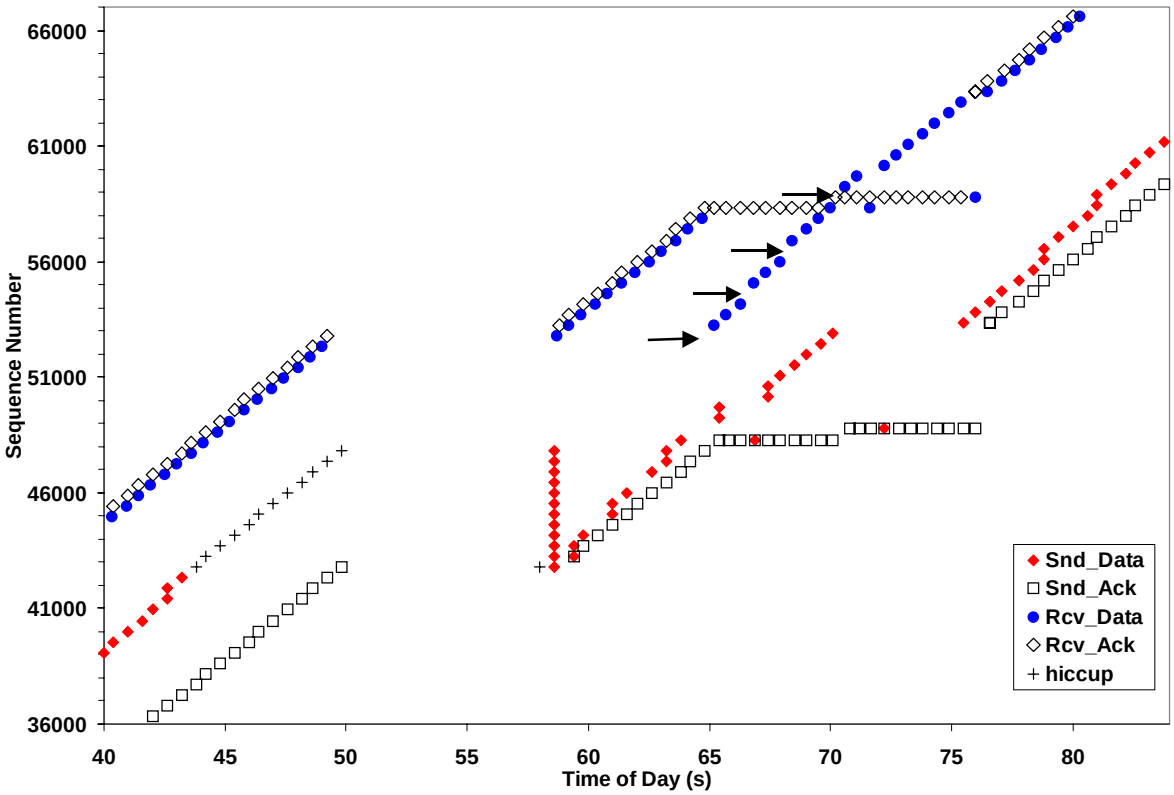

Figure 5-3: Go-back-N after a spurious timeout causing a *real* packet drop.

12 packets and repeated the measurement. During the go-back-N retransmission phase, packets enter the queue faster than they can drain. This eventually causes the interface buffer to overflow four times causing three spurious retransmissions and one original transmission to be dropped (see arrows in Figure 5-3).

5.1.2 Spurious Fast Retransmits

Packet re-orderings with a re-ordering length greater than or equal to the DUPACK-Threshold interfere with TCP's DUPACK-based error recovery, causing a spurious fast retransmits as explained in Section 2.5.4. Spurious fast retransmits affect TCP performance in that the TCP sender unnecessarily reduces its load while also performing a spurious retransmission.

To illustrate a spurious fast retransmit, we performed bulk data transfers on the experimental network described in Section 3.3.2 and used `hiccup` to cause a packet re-ordering event with a re-ordering length of six. This is depicted in Figure 5-4. The packet that was supposed to be sent at second 37.7 (marked as + in Figure 5-4) is queued by `hiccup` while the succeeding six packets are let through. Then `hiccup` sends the single queued packet (see arrow in Figure 5-4) back-to-back with the next packet (the 7th packet sent after `hiccup` was called). This leads to six DUPACKs generated by the receiver, which then trigger the spurious fast retransmit in second 44.7. The error recovery procedure has finished in second 46.9 when the

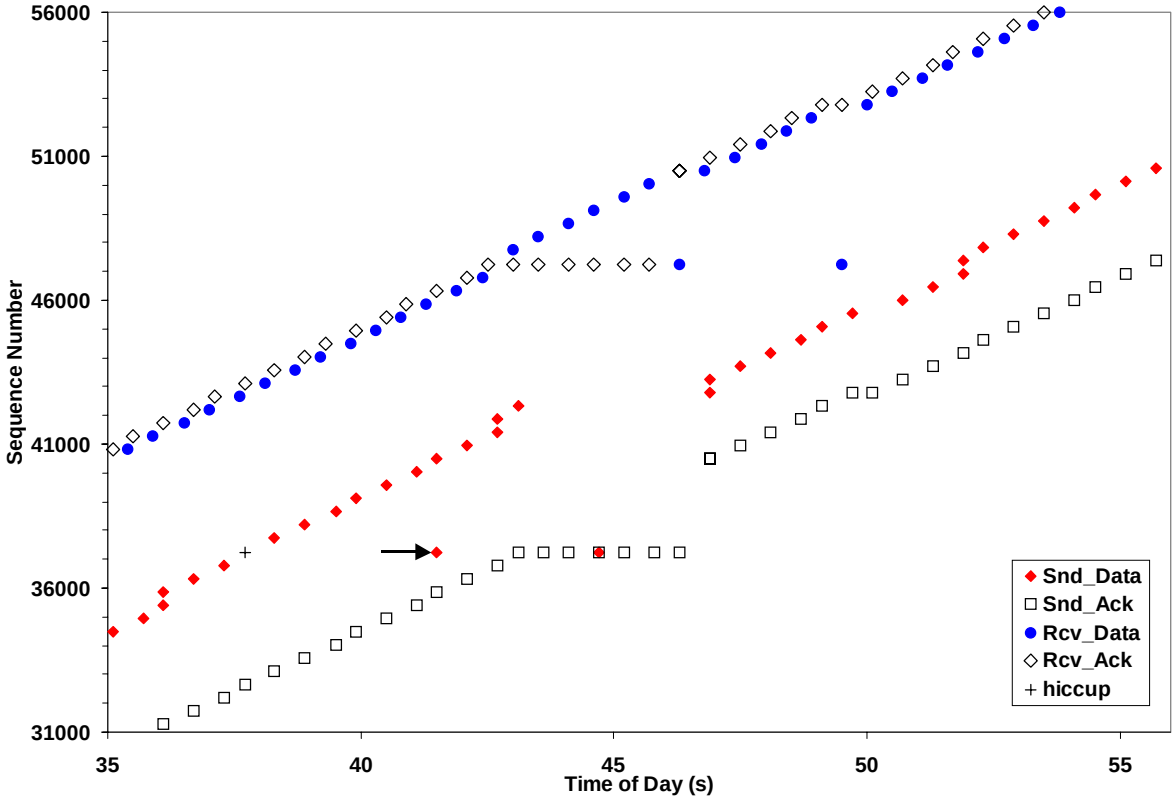

Figure 5-4: The effect of packet re-ordering.

first ACK after the series of DUPACKs arrives at the sender[1]. At that point the sender halves its load.

The retransmission ambiguity is again the core of the problem: on receipt of the first ACK arriving after the series of DUPACKs, the sender must interpret this ACK as having been triggered by the retransmission when in fact it was triggered by the (re-ordered) original transmission of that segment. Hence, the sender unnecessarily halves its load by entering the congestion avoidance phase.

5.2 The Eifel Algorithm

Once we understand the problem explained in the preceding section, the solution is straightforward: first, resolve the retransmission ambiguity and then, restore the load and resume transmission with the next unsent segment. We call this the Eifel algorithm. We have implemented and tested it under FreeBSD, and have made it publicly available [Lud99c]. The current implementation is based on the use of the TCP timestamp option [RFC1323] and adds less than 20 new lines of code to the TCP sender. It does *not* require changes to the TCP receiver code *nor* to the protocol itself. Given this backwards compatibility and the fact that it does not change TCP's congestion control semantics, the new algorithm can be incrementally deployed.

5.2.1 Resolving the Retransmission Ambiguity

Resolving the retransmission ambiguity requires extra information in the ACKs that the sender can use to unambiguously distinguish an ACK for the original transmission of a segment from that of a retransmission. This in turn requires that *every* segment *and* the corresponding ACK carry the extra information to allow the sender to avoid the go-back-N retransmissions described in Section 5.1.1. Waiting for the receiver to signal in DUPACKs that is has correctly received duplicate segments, as proposed in [FMMPR99], would be too late (see Figure 5-2), and is thus not an alternative.

The TCP timestamp option (see Section 2.2.2) provides exactly what we need. Resolving the retransmission ambiguity is then implemented as follows. The sender always stores the timestamp of the *first* retransmission independent of whether that was triggered by an expiration of

1. The 3rd DUPACK sets the slow start threshold to one half of the congestion window (adjusted to a multiple of MSS). On receipt of the first ACK arriving after the series of DUPACKs the sender sets its congestion window to the slow start threshold (which still counts as slow start phase) and then adds one MSS for the new ACK. Thus, the first ACK arriving after the series of DUPACKs always clocks out at least two new segments. In practice, we have noticed that this ACK can also clock out a burst of segments. This happens when the sender has filled the window advertised by the receiver while it continues to grow its congestion window by one MSS for each DUPACK. Figure 4-6 shows such an example.

the retransmission timer or by the receipt of three consecutive DUPACKs[1]. In our implementation, we call that timestamp *ts_first_rexmit*. Then, when the first ACK that acknowledges the retransmission arrives, the sender compares the timestamp of that ACK with ts_first_rexmit. If it is smaller than ts_first_rexmit, this indicates that the retransmission was spurious. The comparison operator “smaller than” leads to the right conclusion in most cases. However, in theory, when the “timestamp clock” is slow or the network is fast, ts_first_rexmit could (at most) also be equal to the timestamp of the first ACK that acknowledges the retransmission. Thus, with using “smaller than”, we are conservative and assume that in those unlikely cases the retransmission was not spurious.

Using the TCP timestamp option to resolve the retransmission ambiguity is one implementation alternative. Which mechanism is implemented for that purpose does not make a difference to the Eifel algorithm. However, including the 12 bytes TCP timestamp option field in every segment and the corresponding ACKs seems heavyweight[2]. Ideally, the Eifel algorithm was based on a single bit in the TCP header (each way) to mark the original transmission of a segment differently from its retransmission(s). That would be similar to the subsequence field proposed in the transport protocol TP 4 [ISO8073]. However, using 2 bits from the 4 remaining reserved bits in the TCP header - as usual - raises deployment concerns. The advantage of using the timestamp option is that this scheme is already a proposed standard and that it is widely deployed. To reduce overhead it may then be an alternative to add timestamp compression to the existing TCP/IP header compression schemes [RFC1144], [RFC2507].

5.2.2 The Sender’s Response

Together with ts_first_rexmit the sender stores the current values of the slow start threshold and the congestion window. When a spurious retransmission has been detected that had led to a *single* retransmission of the oldest outstanding segment, the sender simply restores the slow start threshold and the congestion window to the stored values. After a spurious fast retransmit, this leads to an undesirable packet burst (see Figure 5-6). However, various reasons can cause a TCP sender to send packet bursts as shown in Figure 4-6. That is an orthogonal problem. Ideally, the Eifel algorithm should be complemented with some form of a “burst pacer”.

If, however, more than one retransmission of the oldest outstanding segment has occurred (e.g., a second timeout, or a timeout that occurs after the fast retransmit) the slow start threshold remains halved which was done anyway when the spurious retransmission occurred. If two retransmissions have occurred, the congestion window is also halved (set to the slow start threshold). If more than two retransmissions have occurred, it is set to one segment. Thus, the

1. Unlike in [WS95] the Eifel algorithm counts a fast retransmit as a “regular” retransmission (t_rxtshift++).
2. Another viable alternative has been proposed by Keith Sklower at UC Berkeley which is to use timestamps *only* for retransmissions and their corresponding ACKs.

more spurious retransmissions have occurred, the more conservative the sender gets. Either way, the sender resumes transmission with the next unsent segment. In the case of the fast retransmit algorithm this is done anyway, but in the case of a spurious timeout this prevents the go-back-N retransmissions.

Figure 5-5 illustrates the operation of the Eifel algorithm in the event of a spurious timeout. At 43.3 s into the connection we called `hiccup` to intercept and queue outbound packets for 12 s (marked as + in Figure 5-5). At second 55.2 `hiccup` has terminated and the queued packets get placed into the outbound interface buffer all at once. At that time, the sender has already performed one retransmission (marked as + in Figure 5-5) which was also queued by `hiccup`. We have marked the retransmission with arrow (3) in the receiver trace and with arrow (1) in the sender trace. In the sender trace the original transmission and the retransmission of that segment are the same point. The retransmission was sent at second 54.7 and the Eifel algorithm stores that timestamp as ts_first_rexmit. The first ACK that acknowledges the retransmission (see arrow (2) in Figure 5-5) carries a timestamp of 43.3 s, which is when the original transmission of the corresponding segment took place. By comparison with ts_first_rexmit, the Eifel algorithm detects that the timeout was spurious, restores the slow start threshold and the congestion window, and resumes transmission with the next unsent segment.

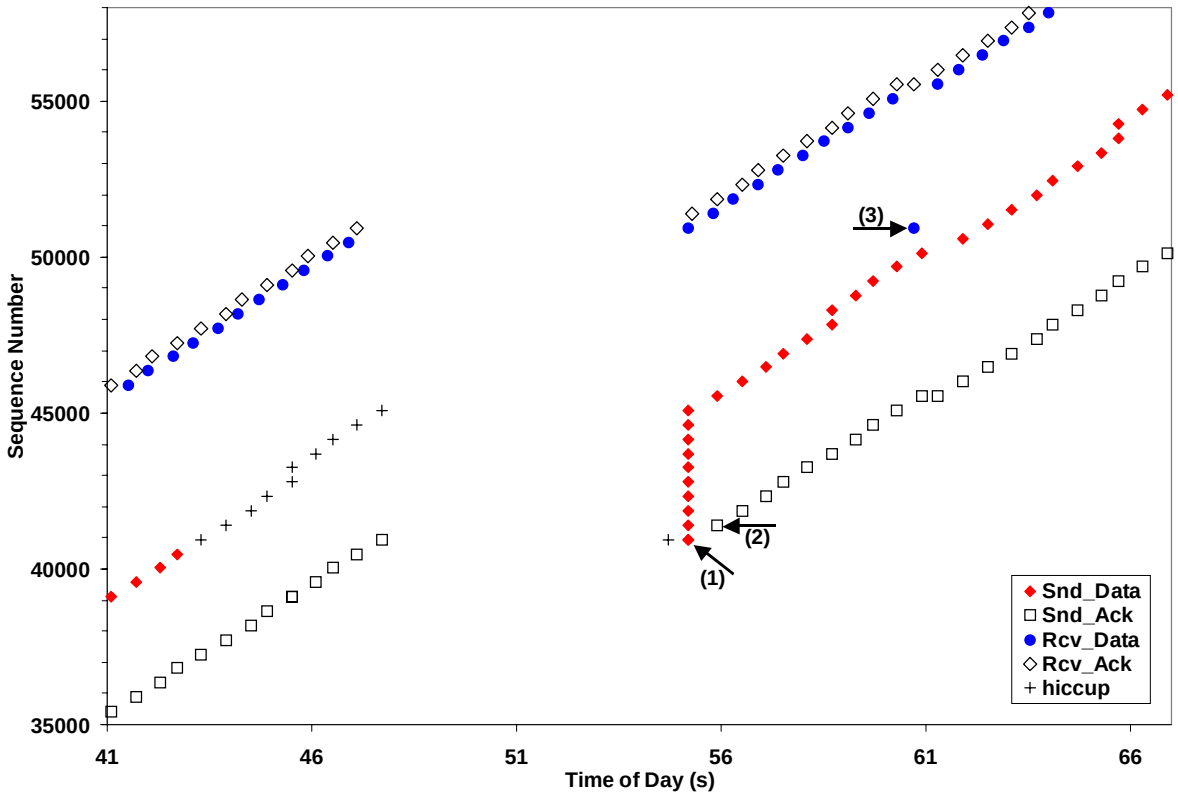

Figure 5-5: Response after spurious timeout.

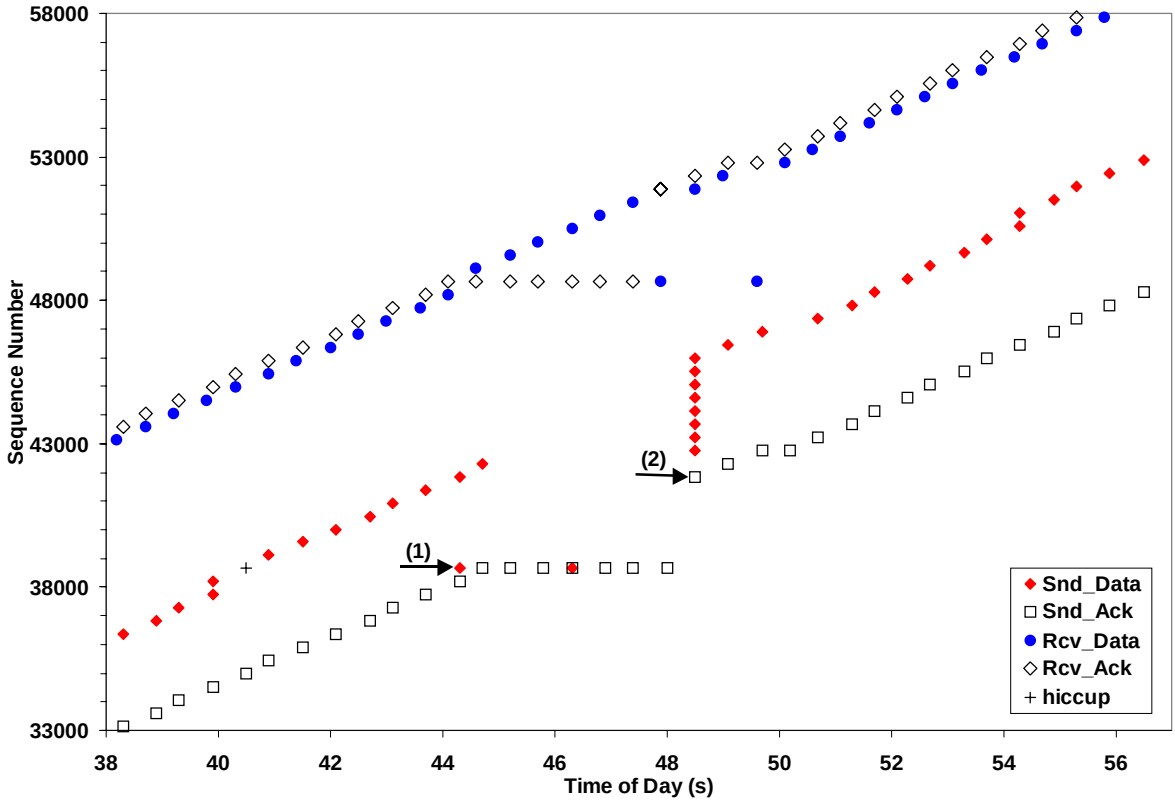

Figure 5-6: Response after a spurious fast retransmit.

Figure 5-6 illustrates the operation of the Eifel algorithm in the event of a spurious timeout. At 40.5 s into the connection we called `hiccup` to cause a packet re-ordering event with a re-ordering length of six (marked as + in Figure 5-6). Then `hiccup` sends the single queued packet (see arrow (1) in Figure 5-6) back-to-back with the next packet. This leads to six DUPACKs generated by the receiver which then trigger the spurious fast retransmit in second 46.3. The Eifel algorithm stores that timestamp as ts_first_rexmit. The first ACK that acknowledges the retransmission (see arrow (2) in Figure 5-6) carries a timestamp of 40.5 s which is when the original transmission of the re-ordered segment took place. By comparison with ts_first_rexmit the Eifel algorithm detects that the fast retransmit was spurious, and restores the slow start threshold and the congestion window. This causes the packet burst mentioned at the beginning of this subsection.

5.2.3 Performance Evaluation

The Eifel algorithm aims to increase TCP's throughput in the face of spurious retransmissions. In this section we argue why it is impossible to perform a definitive performance evaluation for the throughput improvement it can achieve.

First of all, the Eifel algorithm is an improvement for corner cases. If those corner cases never occur during a connection, the Eifel algorithm never gets triggered, and trivially does not affect the connection's throughput. If they occur, the throughput improvement that the Eifel algorithm achieves can be approximated as the avoided performance loss that spurious timeouts and/or spurious fast retransmits would have caused for "standard" TCP. Yet, it is impossible to quantify in general terms the mentioned performance loss. It depends on too many factors, such as traffic type (bulk data or interactive) and the frequency and distribution with which spurious retransmissions occur over the entire lifetime of a connection. Further, assuming bulk data transfers, the performance improvement depends on the ratio of the sender's load to the pipe capacity right before the first spurious retransmission. We explain this using the two graphs shown in Figure 5-7.

A network-limited TCP connection in steady state goes through periodic congestion avoidance cycles as explained in Section 2.3.2. We denote as *W* the load at the end of each cycle in terms of number of segments. We assume a non-shared bottleneck link with a fixed bandwidth and a fixed bottleneck buffer size. This is a common case when the access link becomes the bottle-

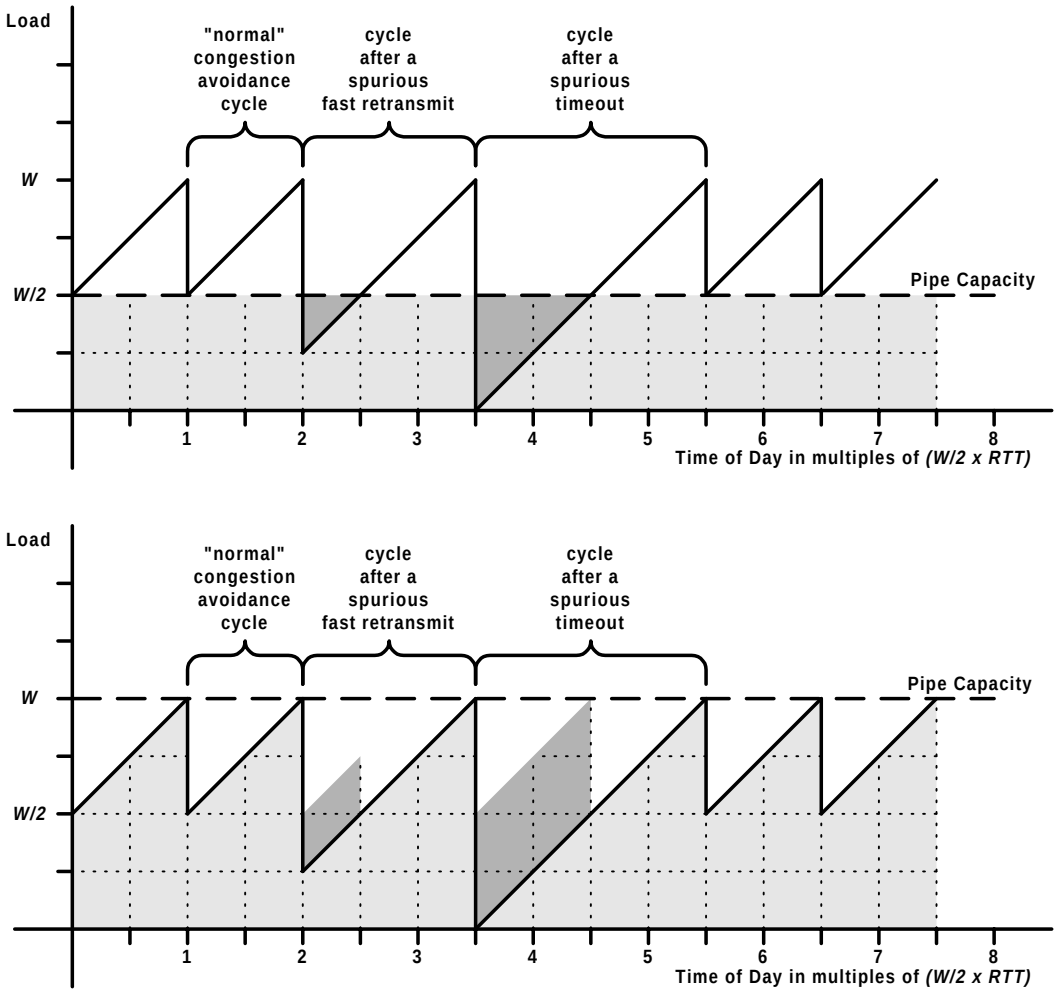

Figure 5-7: Approximating the performance loss.

neck link (e.g., low bandwidth dial-up or wide-area wireless), and only a single application creates traffic. In fact, this scenario is emulated by the setup described in Section 3.3.2. In such a situation, *W* and the pipe capacity are constant. Increasing the load increases the connection's throughput as long the load stays below the pipe capacity. We have indicated that with the lightly shaded area. The cycle length in terms of multiples of the RTT is *W/2* (e.g., see [MSMO97]). The pipe capacity is assumed to be equal to *W/2* in the upper graph of Figure 5-7, and equal to *W* in the lower graph. For both cases, we then approximate the performance loss that a spurious timeout and a spurious fast retransmit cause. As a simplification we assume that the load increase during the slow start phase is linear (in fact it is exponential). On the other hand, we also ignore the performance loss due to spurious retransmissions, i.e., we only evaluate the performance loss caused by the load decrease. We further make the worst-case assumption that the spurious retransmission is triggered at the end of a cycle when the load has already been halved, and that no additional spurious retransmission is triggered before the load as reached *W* again.

Given these assumptions, the performance loss that a spurious fast retransmit causes can be approximated as 9 percent for the upper graph and 13 percent for the lower. We arrive at these numbers by "counting squares" indicated by the dotted lines in Figure 5-7. In the upper graph the spurious fast retransmit creates a cycle of length 1.5 (from 2.0 to 3.5). During that time the sender could have sent "6 squares" when it really only transmitted "5.5 squares". This translates into a performance loss of 9 percent. Likewise, we approximate the performance loss that a spurious timeout causes as 33 percent for the upper graph and 50 percent for the lower.

The general conclusion drawn from this simple analysis is that the lower the ratio of the sender's load to the pipe capacity right before the first spurious retransmission, the higher the performance loss. This is intuitive, but the simple analysis also shows that it is impossible to quantify the loss. It could be anything from nothing to several tens of percent. For example, in our measurements the loss due to the load decrease, i.e., not accounting for the spurious retransmissions itself, is minimal (see Figure 5-2 and Figure 5-4). This can be seen from the receiver trace as there are almost no idle times during which no data is received. The reason is that the bottleneck link in our measurements was massively overbuffered, i.e., the ratio of the sender's load to the pipe capacity was always extremely high.

5.3 Problems of TCP-Lite's Retransmission Timer

In this section, we explain four major problems of the Lite-Xmit-Timer. The first two are fundamental flaws in the definition of RTO_L while the latter two concern the implementation of $REXMT_L$. While the first, third, and fourth problems make the Lite-Xmit-Timer more conser-

vative, the second problem makes it more aggressive. However, the latter is usually outweighed by the other three factors.

5.3.1 Prediction Flaw when the RTT Drops

$RTTVAR_L$ is calculated using the absolute value of $DELTA_L$. Although this is the mathematically correct definition of the mean deviation, it is not motivated in [Jac88] whether using the mean deviation in this strict manner is an appropriate design choice. The undesirable behavior this causes is that the predictor "goes up" when the signal "goes down". More precisely, it causes the RTO to initially increase after the connection's RTT has dropped to the extent that it falls below SRTT, i.e., when DELTA becomes negative.

In those cases, the effect on RTO is the same as if RTT had increased by the same amount. This leads to an RTO that largely over-predicts the RTT, and it takes some time until the RTO has decayed to a reasonable level. We illustrate this in Figure 5-8 generated from the model described in Section 3.4.2. The model was configured to a maximum of 10 for the flow's load and a timer granularity of 1 ms. As in all following figures we use the notation RTT(i) to denote the i-th RTT_{Sample} for which the corresponding RTO, RTO(i-1), was determined from the previous, the (i-1)-th, RTT_{Sample}.

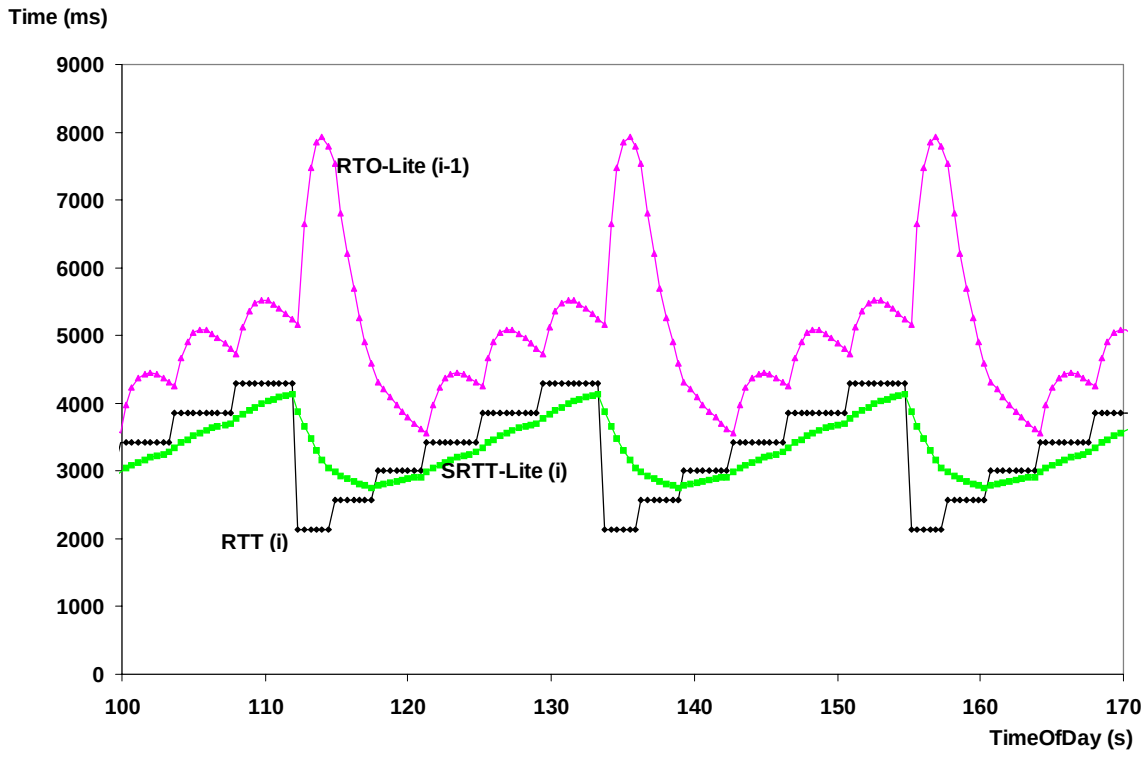

Figure 5-8: Prediction Flaw in RTO_L.

5.3.2 Failure of the “Magic Numbers”

The Lite-Xmit-Timer has been defined under the assumption that only one segment per flight was timed. The estimator gains (1/8 and 1/4) and the variation weight (4) have been tuned to

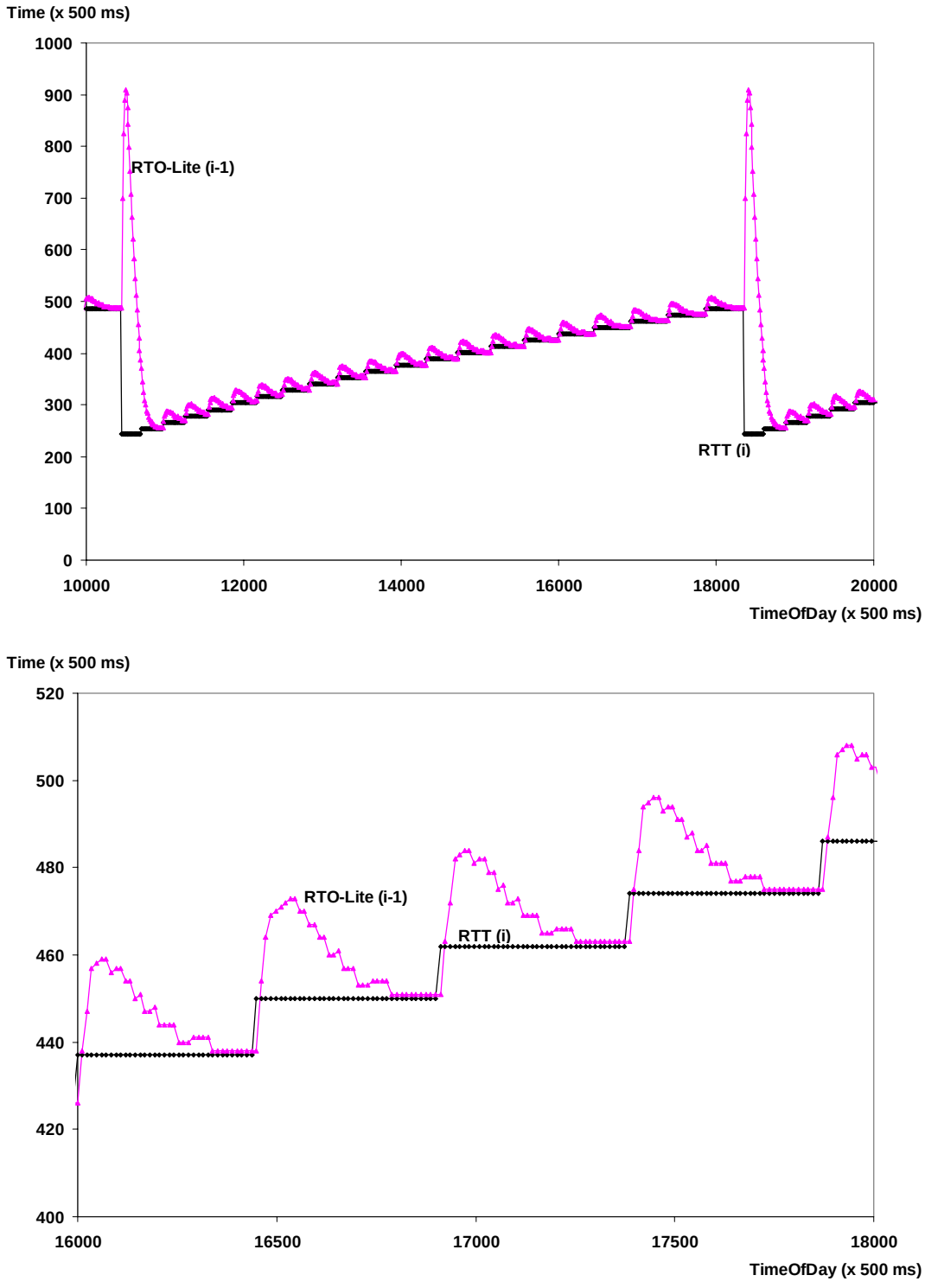

Figure 5-9: A Collapsed RTO_L (model).

that case. However, if the RTT sampling rate is higher *and* the flow's load is large, the fixed estimator gains and the fixed variation weight (the "magic numbers") fail. The problem in that case is that the Lite-Xmit-Timer's variation weight is too low to raise the RTO to a sufficient level, while its estimator gains are too high. This causes $SRTT_L$ and $RTTVAR_L$ to decay too quickly. Thus, RTO_L collapses into the RTT, i.e., RTO_L becomes too aggressive. We illustrate this in Figure 5-9 where the lower graph is a "zoom" of the upper one. The graphs are based on the model configured to a maximum of 40 for the flow's load and a timer granularity of 500 ms. In theory, the aggressive RTO_L should lead to many spurious retransmissions. In practice, this is not the case as we explain in Section 5.3.3 and Section 5.3.4.

To see how RTO_L performed when only a single RTT sample was collected per RTT, we repeated the measurement described in Section 3.4.3 while disabling the timestamp option. The result is shown in Figure 5-10. Although the spikes in the graph of RTO_L still occur for the reason described in Section 5.3.1, at least the estimators gains and the variation weight work. Thus, the problem described in Section 5.3.2 only occurs when the RTT sampling rate is one or close to one.

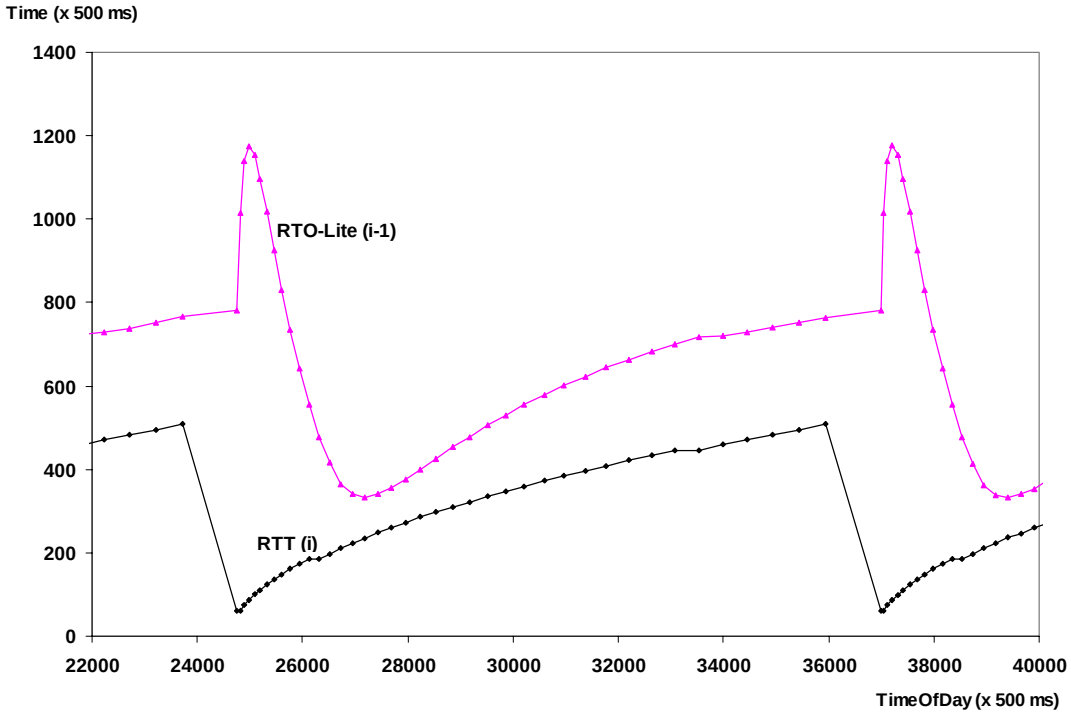

Figure 5-10: RTO_L when timing one segment per RTT.

5.3.3 The "REXMT-Restart Bug"

The problem with the implementation of $REXMT_L$ is that it is re-initialized with RTO_L when an ACK arrives acknowledging the oldest outstanding segment, and more segments are still outstanding. This does not account for the age of the (new) oldest outstanding segment. Thus,

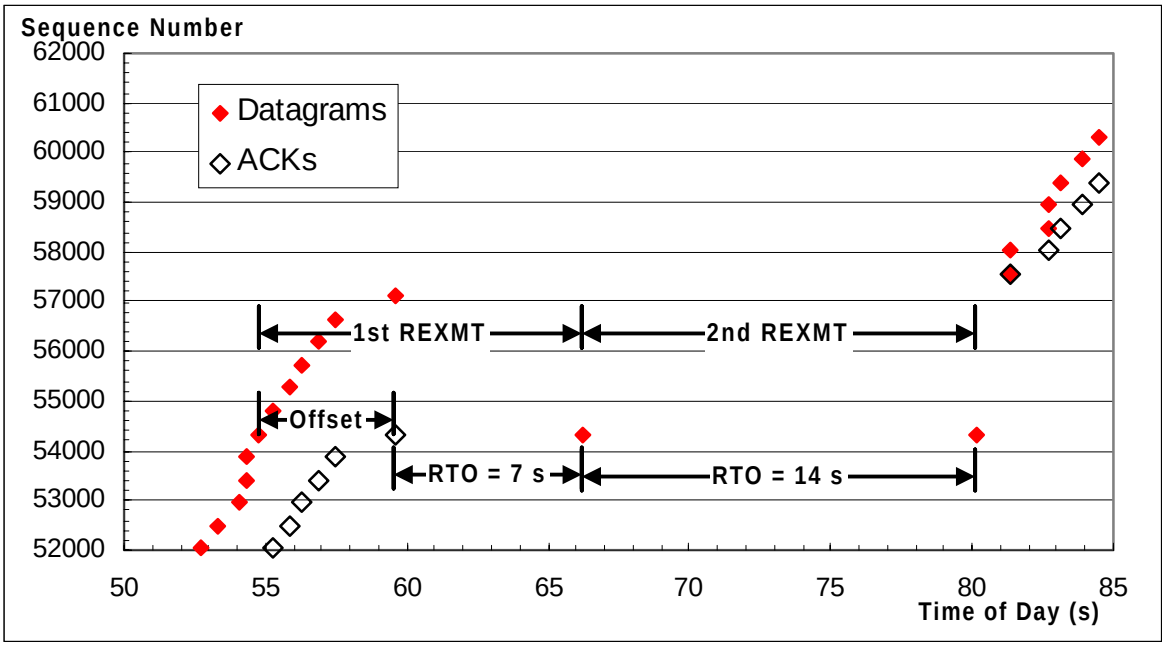

Figure 5-11: The "REXMT-Restart Bug".

before the first timeout occurs, $REXMT_L$ is the sum of RTO_L and the age of the oldest outstanding segment which during bulk data transfer roughly corresponds to the RTT (denoted as "offset" in Figure 5-11). This makes $REXMT_L$ significantly conservative. We have described this problem in [Lud99b].

5.3.4 Timer Granularity

Given that the RTO is a prediction of the upper bound of RTT, the higher the timer granularity, the more imprecise and consequently the more conservative the RTO. Thus, a low timer granularity is desirable. As a rule of thumb we claim without proof that the timer granularity should at least be an order less then the RTT. For example, given that worst-case RTTs commonly found in the wide-area Internet today are on the order of a few 100 ms, the timer granularity should at least be 10 ms or a few multiples of that. Hence, the timer granularity of 500 ms, chosen for TCP-Lite is inadequate. This is one reason why the Lite-Xmit-Timer is so conservative. This issue has been raised many times in the research community. It motivates why other operating systems have been implemented with a timer granularity of 10 ms or less. In addition, a timer granularity of 500 ms obviously defeats the purpose of putting much effort into the formula that determines the RTO when the RTT never grows beyond a few 100 ms.

The problem with the $REXMT_L$ is that it is based on a heartbeat timer (see Section 2.2.2) that expires *every* 500 ms. Simply increasing the frequency of the heartbeat timer would result in a waste of valuable processing power to handle all the "useless" interrupts. That can become a great problem for busy Web servers that might have to handle thousands of TCP connections simultaneously.

5.3.5 Validating the Model

As a validation of the model we decided to reproduce the plots shown in Figure 5-9 which were generated from the model. Thus, we chose the parameter settings for our measurement

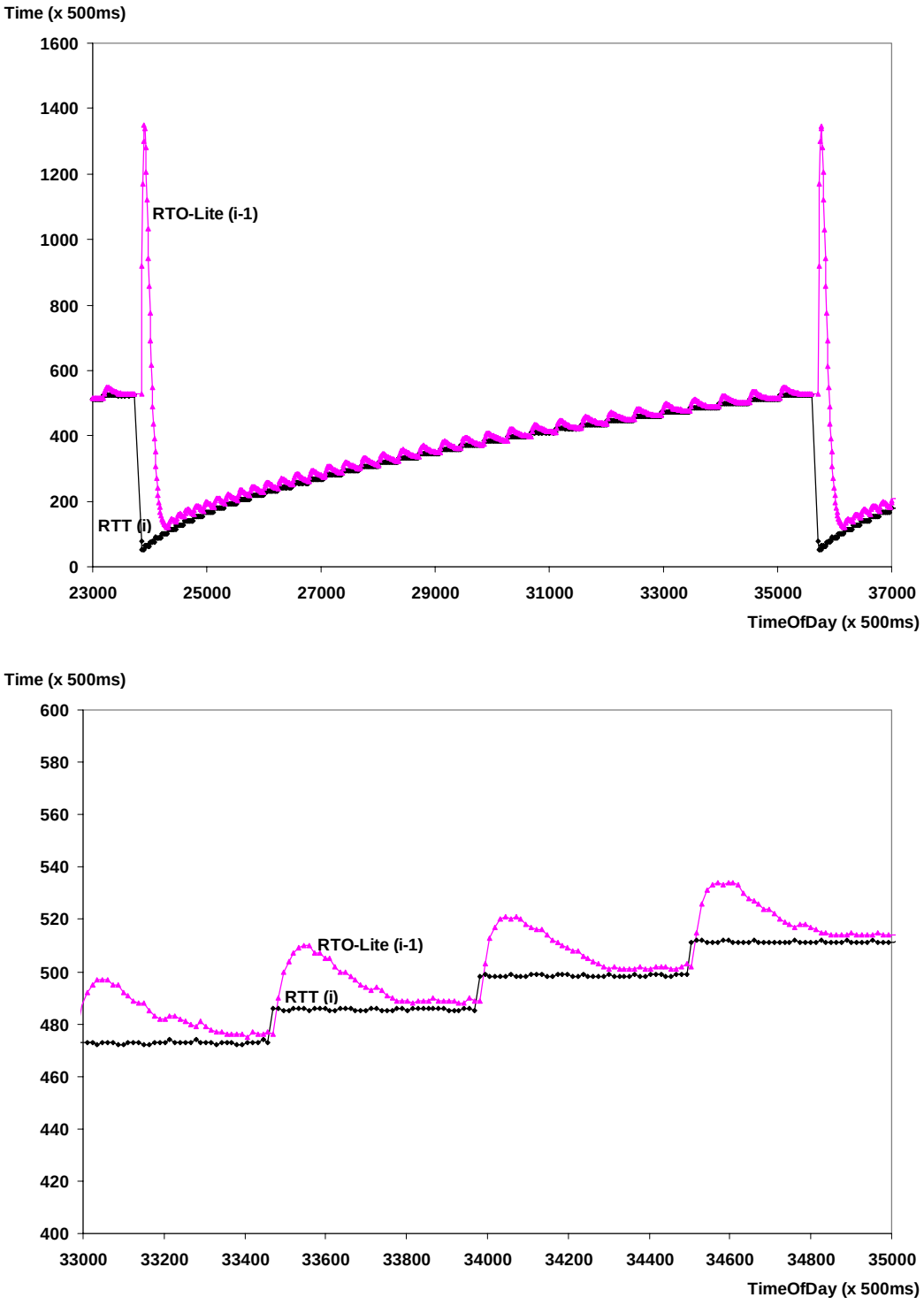

Figure 5-12: A Collapsed RTO_L (measured).

setup as described in Section 3.4.3. Figure 5-12 shows the measurement result. Although we do not get an exact match, it is obvious that the trend of the graphs are identical. This assured us that our model is correct. Hence, we validated in practice what we had already predicted with our model in Section 5.3.2.

5.4 The Eifel Retransmission Timer

Our motivation for developing the Eifel-Xmit-Timer is to eliminate the problems of the Lite-Xmit-Timer explained in Section 5.3. In the following, we use the indices *E (Eifel)* as qualifier for a metric when referring to its definition or implementation. We omit those qualifiers when discussing a particular metric in general. The RTO_E is defined by the following equations which we explain in the following sub-sections.

$$DELTA_E = RTT_{Sample} - SRTT_E$$

$$FLIGHT_E = \mathrm{MAX}\left(SSTHRESH, \frac{CWND}{2}\right) + 1$$

$$GAIN_E = \begin{cases} \frac{1}{FLIGHT_E}, & \text{if RTT Sampling Rate} = 1 \\ \frac{2}{FLIGHT_E}, & \text{if RTT Sampling Rate} = \frac{1}{2} \\ \frac{1}{3}, & \text{if 1 RTT Sample per RTT} \end{cases}$$

$$\overline{GAIN_E} = \begin{cases} GAIN_E, & \text{if } (DELTA_E - RTTVAR_E) \geq 0 \\ {GAIN_E}^2, & \text{if } (DELTA_E - RTTVAR_E) < 0 \end{cases}$$

$$SRTT_E = SRTT_E + GAIN_E \times DELTA_E$$

$$RTTVAR_E = \begin{cases} RTTVAR_E + \overline{GAIN_E} \times (DELTA_E - RTTVAR_E), & \text{if } DELTA_E \geq 0 \\ RTTVAR_E, & \text{if } DELTA_E < 0 \end{cases}$$

$$RTO_E = \mathrm{MAX}\left(\left(SRTT_E + \frac{1}{GAIN_E} \times RTTVAR_E\right), RTT_{Sample} + (2 \times ticks)\right)$$

5.4.1 Predicting a Decreasing RTT

To avoid the problem described in Section 5.4.1, we define $RTTVAR_E$ to remain constant when $DELTA_E$ is smaller than zero. In that case RTO_E decreases only as fast as $SRTT_E$ decreases. This is illustrated in Figure 5-13 using the same parameters chosen for the model discussed with respect to Figure 5-8.

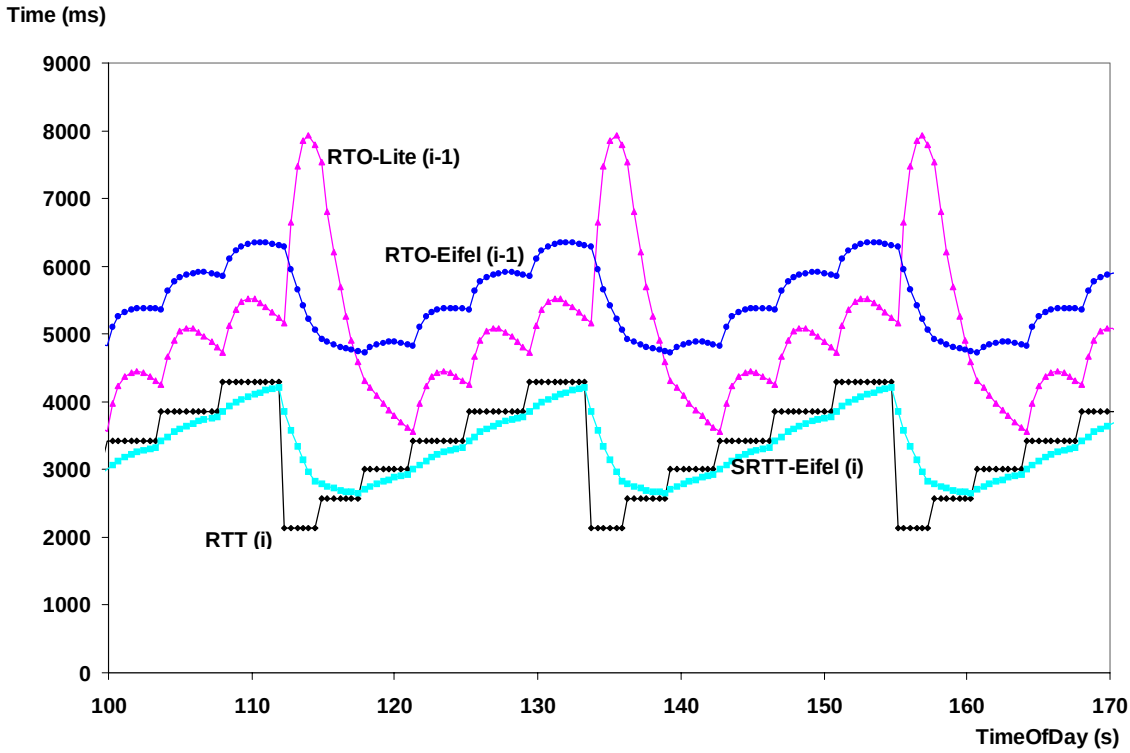

Figure 5-13: Fixing the Prediction Flaw with $RTTVAR_E$.

With this subtle change in the definition of RTTVAR, RTO_E does not exhibit the spikes seen with RTO_L when the RTT drops (see Figure 5-8). Also, note that the graph of $REXMT_L$ (not shown in Figure 5-13 to not overload the plot) lies roughly one RTT "above" the graph of RTO_L because of the problem described in Section 5.3.3. The graph of $REXMT_E$, on the other hand, is identical to the graph of RTO_E for the reason described in Section 5.4.5.

5.4.2 Scaling the Estimator Gains and the Variation Weight

To avoid the problem described in Section 5.3.2, we remove the constant estimator gains. We replace them with a single gain for both $SRTT_E$ and $RTTVAR_E$ that scales with the flow's load and which also depends on the RTT sampling rate. If more than one segment is timed per RTT, the idea is to distribute the entire weight of 1 equally over the number of RTT samples per flight, i.e., to limit the memory of both estimators to one RTT. With an RTT sampling rate of 1 this leads to an estimator gain which is the reciprocal of the flow's load, and it leads to twice that gain when delayed ACKs are used. If only one RTT sample is obtained per RTT, we define our own "magic number" of 1/3 as the estimator gain. We have verified with the model and a broad range of parameter settings (especially with a small maximum for the flow's load) that this constant leads to an RTO_E that is sufficiently safe against spurious timeouts.

Likewise, we define the variation weight as the reciprocal of the estimator gain and thereby also make it scale with the flow's load. In a situation where the RTT has remained constant for a "long time" (i.e., when $RTTVAR_E$ has become zero and $SRTT_E$ has converged to the RTT) and the RTT suddenly increases, this ensures that RTO_E is the sum of $SRTT_E$ and $DELTA_E$[1].

Various alternatives exist to define $FLIGHT_E$. It is only important that it corresponds to the flow's load. In fact, one could define $FLIGHT_E$ as the actual flow's load at any point in time as that can be derived from the sender-side TCP state. However, we found that that can be too noisy, leading to many RTO_E spikes. We have therefore chosen to approximate a lower bound for the flow's load. The slow start threshold [Jac88] (*SSTRESH*) is an appropriate candidate for that. In the common case the slow start threshold equals half the congestion window [Jac88] (*CWND*) but not necessarily, e.g., when the available bandwidth increases. In that case we use half the congestion window to determine the approximation of the lower bound of the flow's load. We add the constant 1 in the definition of $FLIGHT_E$ because a flow's load at the beginning of a congestion avoidance cycle equals (*SSTRESH + 1*) or (*CWND/2 + 1*). In that case both terms are equal. With those changes we arrive at an RTO where the fraction RTO/RTT remains fairly constant (see Figure 5-14).

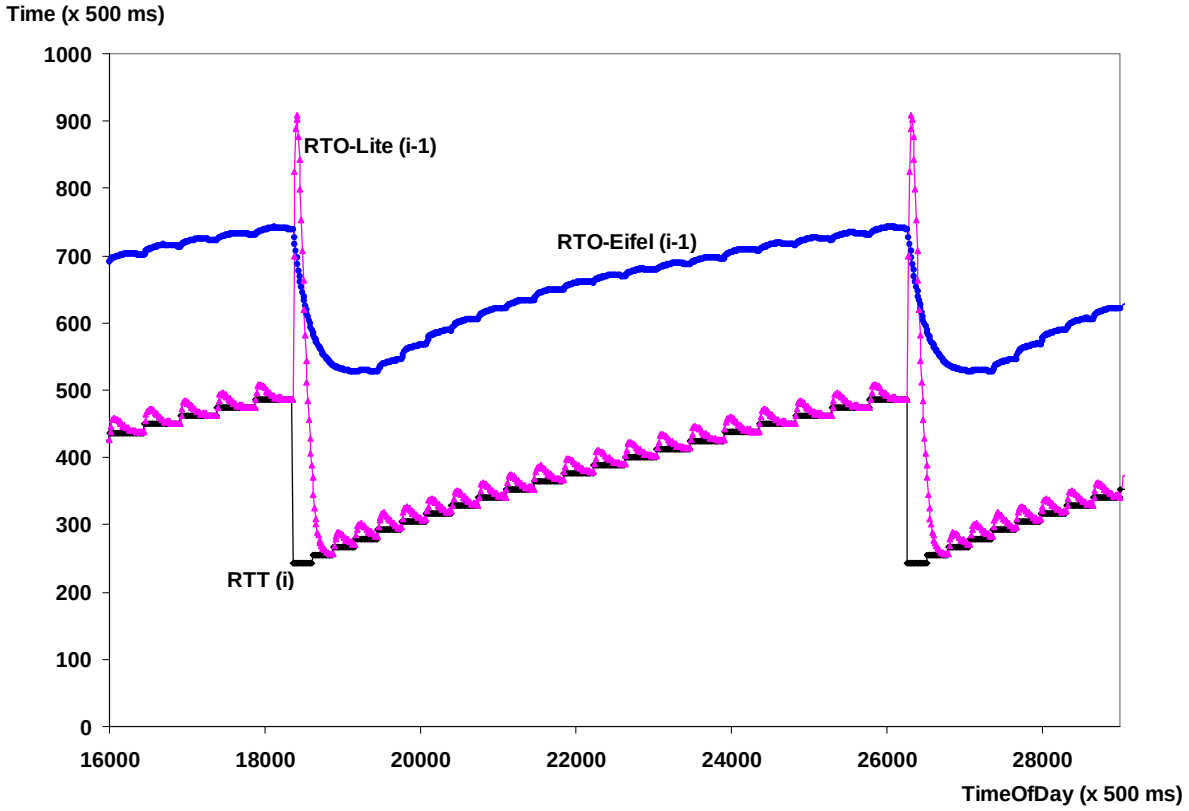

Figure 5-14: RTO_E scales with the flow's load (model).

1. In those situations the minimum defined for RTO_E would become effective. Thus, to be more conservative, one might also define the variation weight as $m/GAIN_E$ with m = 2, 3, 4,....

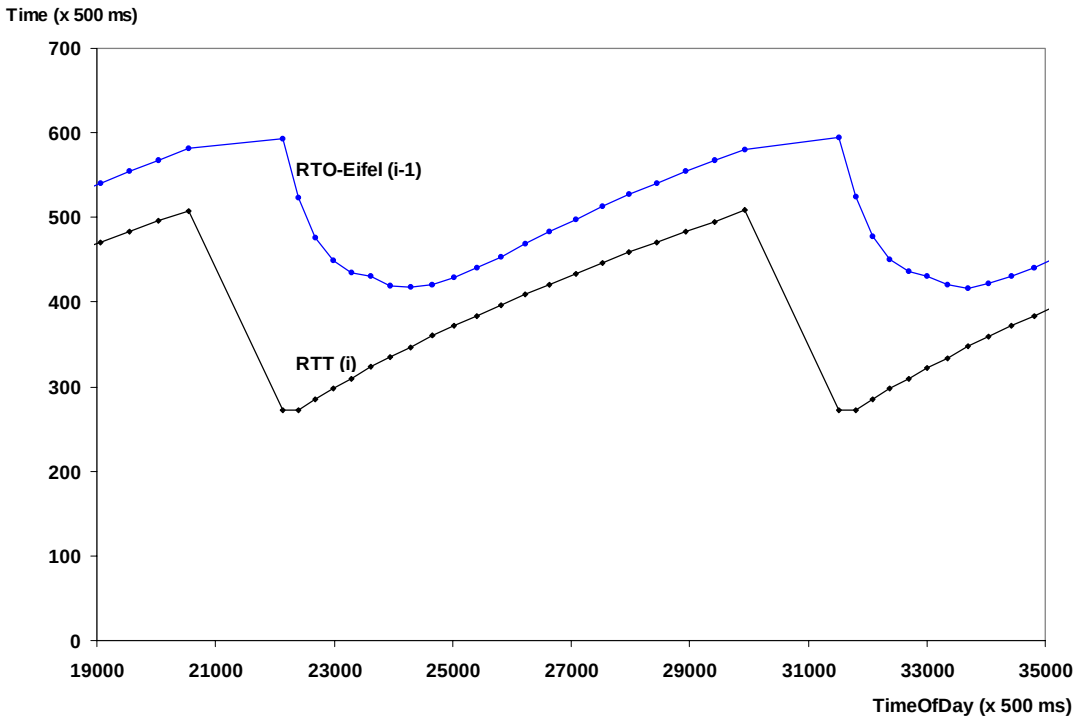

Figure 5-15: RTO_E when timing one segment per RTT.

To see how RTO_E performed when only a single RTT sample was collected per RTT, we repeated the measurement described in Section 3.4.3 while disabling the timestamp option. The result is shown in Figure 5-15. The graph of RTO_E does not look much different from that of RTO_L in Figure 5-10, except that it does not have those spikes at the end of a congestion avoidance cycle.

Another phenomenon can be seen when comparing Figure 5-10 and Figure 5-15. Although the maximum RTT is about 250 s in both cases, the minimum RTT is quite different. This is due to the TCP sender's "choice" about which segments get timed to collect an RTT sample. If a segment gets timed just before the end of a congestion avoidance cycle, the RTT is high, and it will take the duration of that RTT until the next segment is timed. However, during this phase of the connection the queue at the bottleneck has drained *and* already begun to build up again. Thus, during that time the RTT had dropped and slowly increased again. This had gone unnoticed by the TCP sender that was still waiting to collect the (high) RTT sample. On the other hand, if the timing of a segment ends shortly after the end of a congestion avoidance cycle, the following low RTTs get sampled, too.

5.4.3 Shock Absorbers

In our initial definition of RTO_E we were seeing the same effect that can, e.g., be seen in Figure 5-13 with respect to RTO_L. There the RTO_L increases when RTT increases. However,

the increase phase of RTO_L ends half way through each flight. Then the RTO_L decreases rapidly during the second half of each flight. This can become problematic when the flow's maximum load is small. At the end of a each flight, the RTO_L might get too close to the RTT. To avoid that, we defined the gain for $RTTVAR_E$ to be the square of $GAIN_E$ whenever $RTTVAR_E$ is decreasing. We call this the "shock absorber effect": the variation goes up quickly but comes down slowly. As with the estimator gains, no constant would have worked to slow the decrease of $RTTVAR_E$. We therefore, again, chose to make that inverse proportional to the flow's load by multiplying $GAIN_E$ with $1/FLIGHT_E$. This has the effect that RTO_E stays roughly at the same level during the second half of each flight (see the graph of RTO_E in Figure 5-13).

5.4.4 The RTO Minimum

The RTO minimum should be seen as necessary to protect against spurious timeouts in situations where the RTT is close to or even below the timer granularity. In all other cases, the minimum should have no effect. If it *does*, then this clearly shows that the RTO has failed as a predictor of an appropriate upper bound for the RTT. When using a heartbeat timer, the RTO minimum must at least be 2 ticks as discussed in Section 5.3.4. In addition, it seems reasonable to have the RTO not drop below the latest RTT sample. This had already been implemented in the FreeBSD operating system. This motivates our definition of the minimum for RTO_E.

5.4.5 Implementing REXMT Precisely

Eliminating the problem described in Section 5.3.3 is straightforward. In our implementation of the Eifel-Xmit-Timer, we simply store the timestamp of when each segment is sent in a dynamic data structure. That way we always know the age of the oldest outstanding segment and can implement $REXMT_E$ according to the following definition.

$$REXMT_E = RTO_E - \text{'Age of oldest outstanding segment'}$$

In situations where a connection does not have enough segments in flight to trigger the fast retransmit/recovery algorithm [Jac90a], i.e., when error recovery has to rely on the retransmission timer, $REXMT_E$ can greatly improve the end-to-end performance compared to $REXMT_L$.

To demonstrate that we configured our experimental network described in Section 3.3.2 to a link speed of 9.6 Kbit/s and set the interface buffer to a size of one packet. This meant that no more than three segments were in flight at any point in time, effectively disabling the fast retransmit algorithm. In Figure 5-16, we compare $REXMT_L$ with $REXMT_E$ using RTO_L in *both* cases to isolate the improvement that is achieved by restarting REXMT precisely. In this case, $REXMT_E$ improves the end-to-end throughput by almost 30 percent due to the quicker recovery of the periodically dropped segments. Figure 5-17 shows a detailed view of sections of the

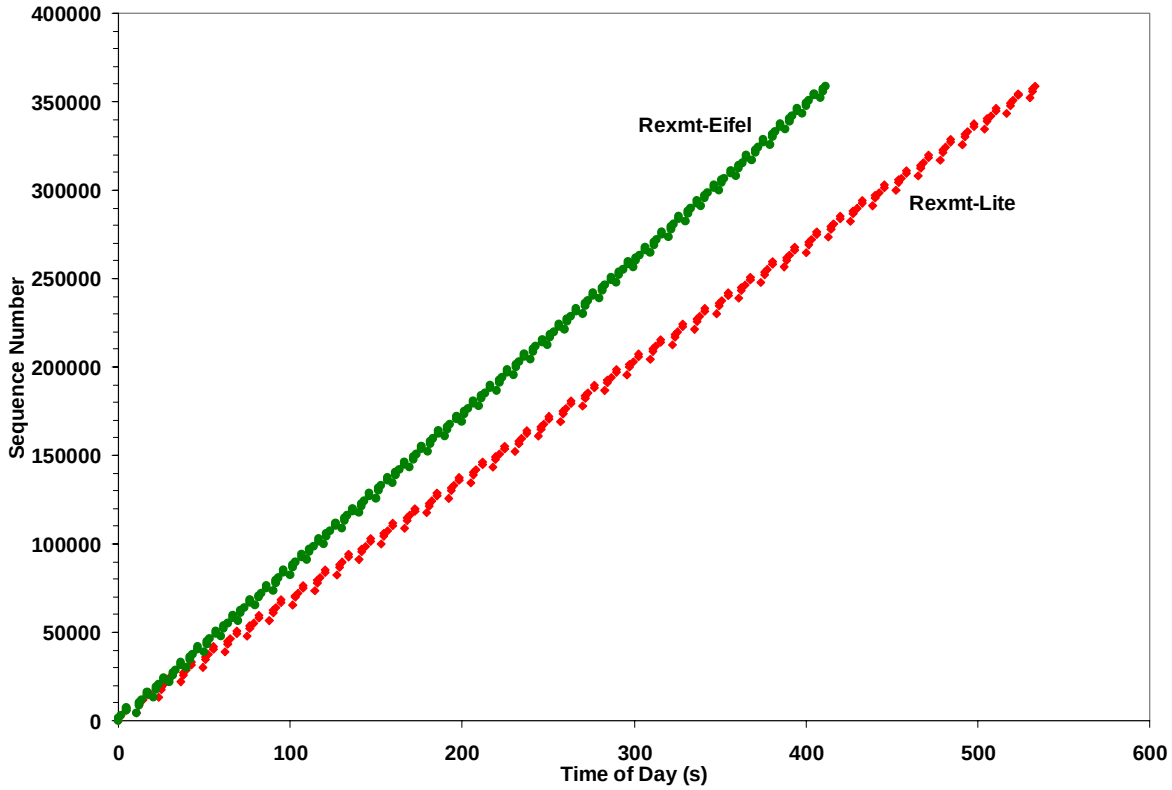

Figure 5-16: Restarting $REXMT_E$ precisely.

two graphs shown in Figure 5-16. For $REXMT_E$ one can see that the timeout occurs before a third duplicate ACK would have been received by the sender, had the receiver sent that ACK. To avoid the resulting competition between timeout-based error recovery and the fast recovery algorithm, the Eifel algorithm suppresses the fast retransmit, and restores the slow start threshold and the congestion window as if the timeout had not occurred.

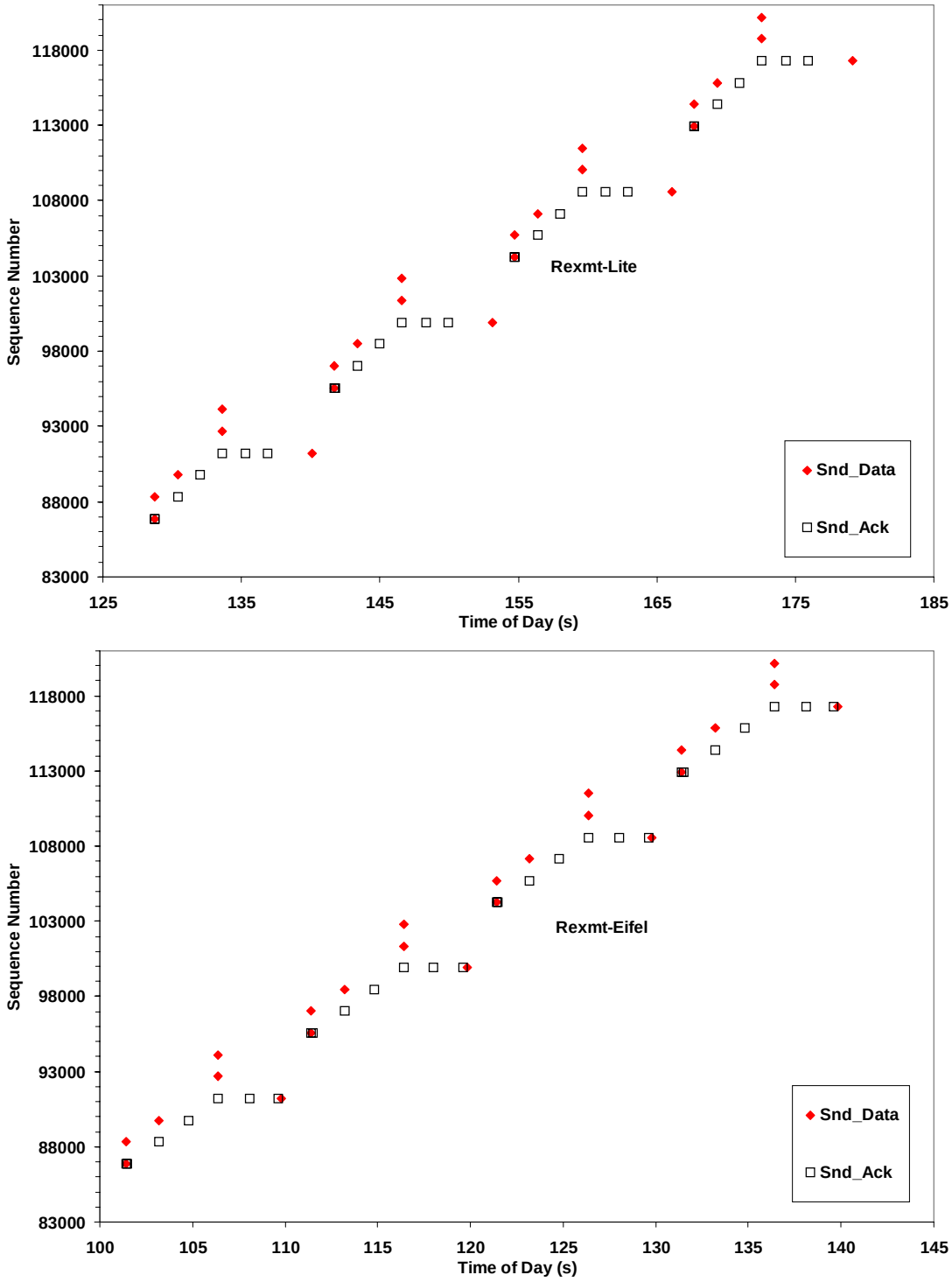

Figure 5-17: Zoom of the graphs shown in Figure 5-16.

5.4.6 Adapting to Spurious Timeouts

The Eifel algorithm allows a more optimistic retransmission timer because it ensures that the penalty for underestimating the RTT is minimal. In the common case, the only penalty is a sin-

gle spurious retransmission. With that in mind and given that in steady state RTO_E/RTT is a fairly constant fraction, we present an alternative to the given definition of RTO_E that makes it become increasingly aggressive. For that purpose, we multiply RTO_E with a factor smaller than one, we call *AGG (Aggressive)*, and let *AGG* decay over time. In preparation for this work, we have included in the Eifel algorithm a feature we have not explained in Section 5.2. It avoids competition between timeout-based and DUPACK-based error recovery in case the third DUPACK for a segment arrives *after* a timeout has already occurred. In that case, the fast retransmit is suppressed, and the slow start threshold and the congestion window are restored as if the timeout had not occurred, i.e., the TCP sender goes into congestion avoidance.

$$CYCLE = \frac{3}{8} \times MAXCWND^2$$

$$AGG = \begin{cases} AGG \times \left(1 - \frac{k}{CYCLE}\right), & \text{for each valid } RTT_{Sample} \\ \text{MIN}\left(AGG + \frac{1}{2} \times (1 - AGG), 1\right), & \text{for each spurious timeout} \end{cases}$$

$$RTO_{AGG} = AGG \times RTO_E$$

We define *AGG* so that it adapts to the number of spurious retransmissions that occur during the lifetime of a connection. This lets the RTO become increasingly aggressive, i.e., let it converge to RTT, until a spurious timeout occurs, and then back it off to a more conservative level before it becomes more aggressive again. We propose an alternative definition provided above for the RTO which we call RTO_{AGG}.

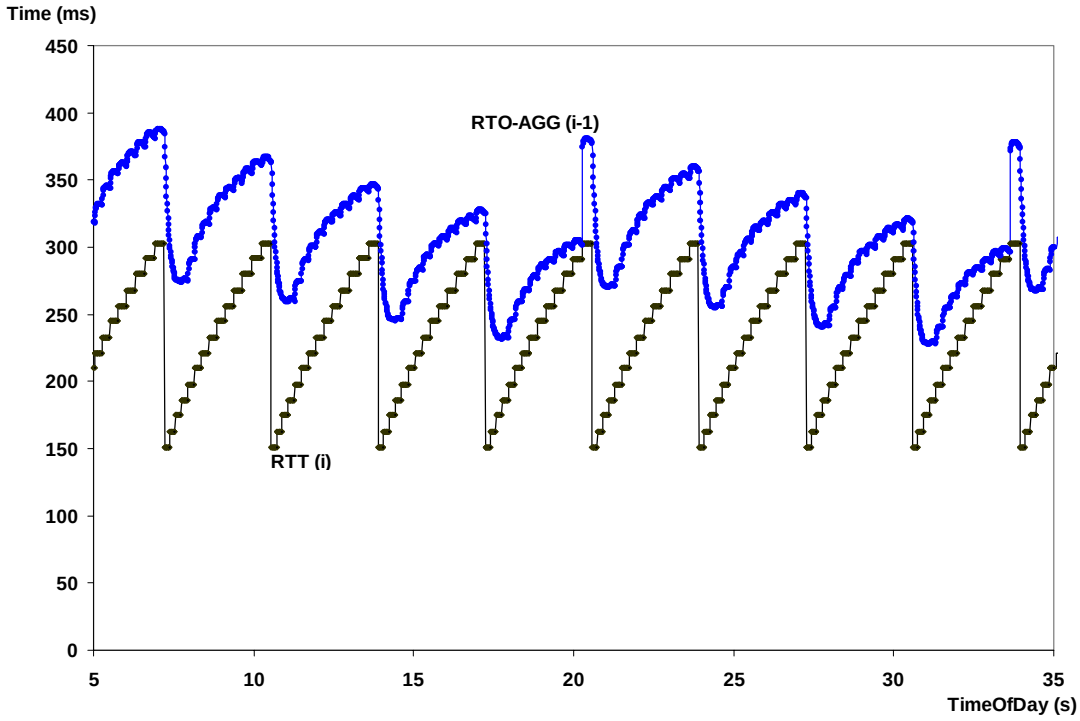

Figure 5-18: A self-trained RTO.

CYCLE is the formula (e.g., see [MSMO97]) that determines the number of segments sent within the last congestion avoidance cycle which ended with a congestion window of *MAXCWND* (in multiples of the segment size). The factor *k* (*0 < k < 1*) determines how quickly RTO_{AGG} converges to RTT. For example, k = 0.1 reduces *AGG* (*0 < AGG < 1*) by roughly 10 percent per congestion avoidance cycle. We illustrate this in Figure 5-18, based on the model configured to a maximum of 26 (= *MAXCWND*) for the flow's load, a timer granularity of 1 ms, and a factor *k* of 0.05. Clearly, more research is required to determine a reasonable value for *k*.

5.4.7 Validating the Implementation of RTO-Eifel

As a validation of our implementation of RTO_E we decided to reproduce the graph of RTO_E shown in Figure 5-14 that was generated from the model. Again, we chose the parameter settings for our measurement setup as described in Section 3.4.3. Figure 5-19 shows the measurement result. A comparison yields a close match. Given that we know from Section 5.3.5 that the model is correct, we now have also validated that the implementation of RTO_E is correct in the sense that it conforms to the definition of RTO_E provided at the beginning of Section 5.2.

We have deliberately plotted the graph of RTO_E without connecting lines to highlight the gap after each congestion avoidance cycle. During that time the TCP sender received a series of duplicate ACKs that tiggered the fast retransmit and fast recovery algorithm. No valid RTT

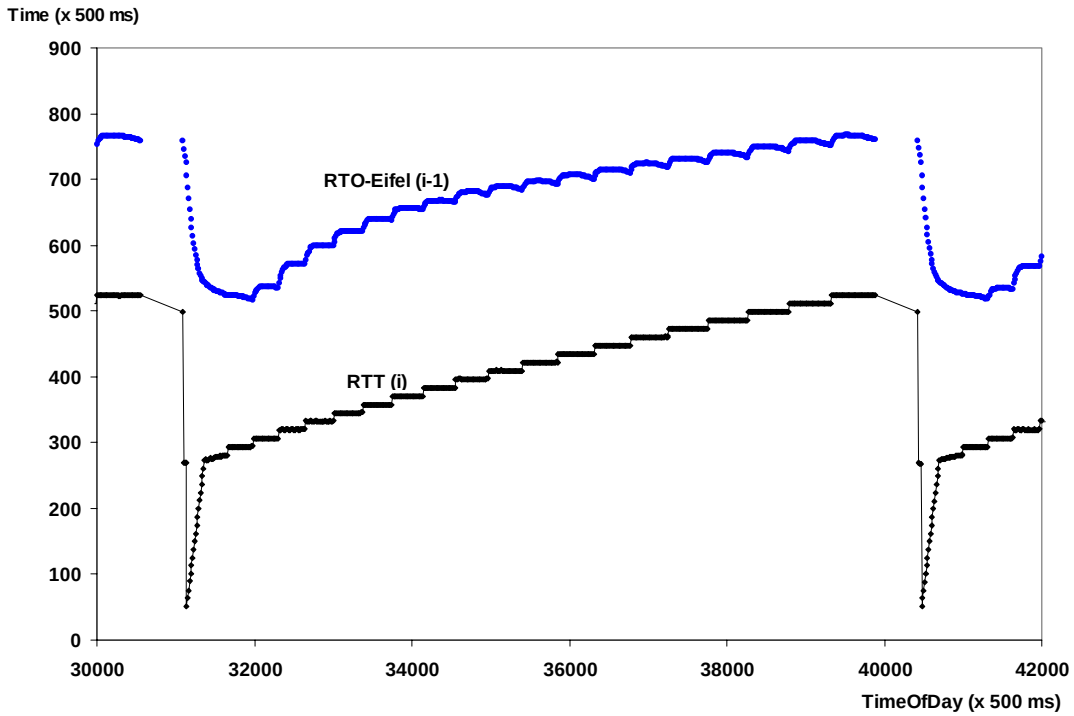

Figure 5-19: RTO_E scales with the flow's load (measured).

samples are derived from those duplicate ACKs which causes the gaps in the graph. This is different in our model for which we have modeled explicit congestion notification.

5.5 Summary

We proposed an enhancement to TCP's error recovery scheme, which we call the *Eifel algorithm*. It uses extra information in the TCP header to eliminate the problems caused by competing error recovery. Our current implementation is based on the TCP timestamp option, and only requires changes to the TCP sender implementation. It does not require changes to the TCP receiver code nor to the protocol itself. Thus, given this backwards compatibility and the fact that it does not change TCP's congestion control semantics, the new algorithm can be incrementally deployed.

In Chapter 4, we showed that the end-to-end performance of fully-reliable flows, such as those based on TCP, can only be optimized by running highly persistent link layer error recovery. The one missing piece, however, was a solution for situations where the wireless connectivity is intermittent, i.e., situations where spurious timeouts are likely to occur. Frequent disconnections - on the order of seconds - without losing data are common in packet-radio networks. In such environments, the algorithm can improve the end-to-end throughput by several tens of percent, although we show that an exact quantification is highly dependent on the path characteristics. Thus, with the Eifel algorithm implemented in TCP and the implementation of a flow-adaptive wireless link, the long standing problem of "TCP over lossy links" is eliminated.

In addition, we have proposed a new retransmission timer for TCP, which we call the *Eifel retransmission timer*, that can also be incrementally deployed. It eliminates four major problems of TCP-Lite's retransmission timer which have revealed in our work. We demonstrated that the Eifel retransmission timer is a more precise predictor of an upper bound for the path's RTT while reacting quicker to packet losses. We showed that this can increase the end-to-end throughput by more than 30 percent. As another alternative, we proposed an advanced version that becomes increasingly optimistic while adapting to the measured fraction of spurious timeouts. This requires the Eifel algorithm that opened the door to the development of a more optimistic retransmission timer because the Eifel algorithm ensures that the penalty for underestimating the RTT is minimal. In the common case, the only penalty is a single spurious retransmission. Although we studied retransmission timers in the context of TCP, we believe that the design principles we proposed are applicable to other reliable end-to-end, and link layer protocols.

The strength of our work related to end-to-end retransmission timers lies in its hybrid analysis methodology explained in Section 3.4. We developed models of each retransmission timer for the class of network-limited TCP bulk data transfers in steady state. With that model we were able to predict the problems of TCP-Lite's definition of the RTO. We also used that model to develop a new RTO for the Eifel retransmission timer. We then validated the correctness our model-based analysis through measurements in a real network that yielded the same results.