The Eifel Retransmission Timer

Reiner Ludwig
Ericsson Research
Herzogenrath, Germany

Keith Sklower
Computer Science Division
University of California at Berkeley

Abstract

We analyze two alternative retransmission timers for the Transmission Control Protocol (TCP). We first study the retransmission timer of TCP-Lite which is considered to be the current de facto standard for TCP implementations. After revealing four major problems of TCP-Lite's retransmission timer, we propose a new timer, named the *Eifel retransmission timer*, that eliminates these. The strength of our work lies in its hybrid analysis methodology. We develop models of both retransmission timers for the class of network-limited TCP bulk data transfers in steady state. Using those models, we predict the problems of TCP-Lite's retransmission timer and develop the Eifel retransmission timer. We then validate our model-based analysis through measurements in a real network that yield the same results.

1. Introduction

The *retransmission timeout value (RTO)* is the time that elapses after a packet has been sent until the sender considers it lost and therefore retransmits it. This event is called a *timeout*. The RTO is a prediction of the upper limit of the *round-trip time (RTT)*, i.e., the time that elapses after a packet left the sender until the sender receives a positive acknowledgment (ACK) for that packet. In the following, we speak of "the RTT" when referring to the RTT of the last segment for which the sender received the ACK, independent of whether the sender had timed that segment to derive the RTT. Especially on an end-to-end path through the Internet, the RTT may vary considerably for various reasons. The time that remains until the timeout for a packet occurs is maintained by the *retransmission timer state (REXMT)*. Thus, the RTO is the REXMT's initial value. We use the term *retransmission timer* to refer to the combination of REXMT and RTO.

The retransmission timer is a key feature of a reliable link or transport layer protocol. It can greatly influence peer-to-peer performance. A too optimistic retransmission timer often expires prematurely. Such an event is called a *spurious timeout*. It causes unnecessary traffic, so-called *spurious retransmissions*, reducing a connection's effective throughput. In TCP [20], timeouts also trigger congestion control [1], [6], [8], [21] that may additionally reduce the end-to-end throughput. A retransmission timer that is too conservative may cause long idle times before the lost packet is retransmitted. This can also degrade performance. This is obvious for interactive request/response-style connections. But it also affects bulk data transfers whenever the sender has exhausted the window limiting the number of outstanding packets before the retransmission timer expires.

In this paper, we analyze two alternative retransmission timers for TCP. Although, we only focus on TCP, we believe that our conclusions also apply to other reliable end-to-end and link layer protocols. We first study the RTO proposed in [6], and the implementation of that RTO and its corresponding REXMT as documented in [24]. We refer to that implementation of TCP as *TCP-Lite* since it is part of the 4.4BSD-Lite distribution of the BSD (Berkeley Software Distribution) operating system. The BSD networking stack has been ported to various operating systems running on hundreds of thousands of servers and clients on the Internet.

In the following, we refer to TCP-Lite's retransmission timer as the *Lite-Xmit-Timer*. After revealing a number of problems of the Lite-Xmit-Timer, we propose an alternative retransmission timer which we call the *Eifel retransmission timer*, and refer to it as the *Eifel-Xmit-Timer*[1]. In the following, we use the indices *L (Lite)* and *E (Eifel)* as qualifiers for a metric when referring to its definition or implementation. We omit those qualifiers when discussing a particular metric in general. The following set of equations define RTO_L. In its implementation, RTO_L is updated every time the sender completes a new RTT measurement, denoted as RTT_{Sample}.

$$DELTA_L = RTT_{Sample} - SRTT_L$$

$$SRTT_L = SRTT_L + \frac{1}{8} \times DELTA_L$$

$$RTTVAR_L = RTTVAR_L + \frac{1}{4} \times (|DELTA_L| - RTTVAR_L)$$

$$RTO_L = \mathrm{MAX}(SRTT_L + 4 \times RTTVAR_L, 2 \times ticks)$$

SRTT is the so-called *smoothed RTT estimator*. $SRTT_L$ is a low-pass filter that memorizes a connection's RTT history with a fixed weighing factor of 7/8.

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

$DELTA_L$ is the difference between the latest RTT_{Sample} and the current $SRTT_L$. RTTVAR is the so-called *smoothed RTT deviation estimator*. Through RTTVAR, the RTO accounts for variations in RTT. $RTTVAR_L$ is a low-pass filter that keeps a memory of a connection's RTT deviation history with a fixed weighing factor of 3/4. We refer to the constants 1/4 and 1/8 as the *estimator gains* and to the constant 4 as the *variation weight*. Little motivation other than implementation efficiency is provided in [6] for this particular set of constants.

REXMT and RTO are maintained in multiples of *ticks*, i.e., some fraction of a second that is operating system dependent. This is also referred to as the *timer granularity*. Because of the heartbeat timer (explained in Section 3.4) implemented in TCP-Lite, a minimum of 2 ticks is required for RTO_L.

We call the time that has elapsed since a segment was sent the *age of a segment*. Likewise we refer to the *oldest outstanding segment* as that segment in the sender's send buffer with the highest age. That segment also carries the lowest sequence number of all outstanding segments. It is the segment that gets retransmitted when REXMT expires. TCP-Lite maintains a single REXMT per TCP connection. When a segment is sent and $REXMT_L$ is not active, it is started (initialized with RTO_L). When an ACK arrives that acknowledges the oldest outstanding segment and more segments are still outstanding, $REXMT_L$ is re-initialized with RTO_L.

We briefly summarize related work concerning the Lite-Xmit-Timer. *Karn's algorithm* [11] must be implemented in TCP [3]. It prevents a clamped RTO by ignoring the RTT_{Sample} derived from a retransmission *and* doubling the RTO (*exponential timer backoff*) up to a maximum of two times the maximum segment lifetime [3], i.e., 240 seconds, each time REXMT expires for the same segment. This makes it possible to eventually collect a valid RTT_{Sample} again. Otherwise, the sender might get stuck retransmitting the oldest outstanding segment while the RTO is clamped at too low a value. The authors of [4] remove an inaccuracy in the implementation of RTO_L that made it more conservative than intended in its definition. This has been updated accordingly in later TCP implementations (e.g., in the FreeBSD operating system). Through trace-driven simulation, the Lite-Xmit-Timer and some of its variations are evaluated in [2] against a large set of real measurements. The authors conclude that the RTO minimum (*2 x ticks*) dominates the performance of the Lite-Xmit-Timer and that its performance can be further increased when a timer granularity of 100 ms or less is implemented. However, that study also concludes that the estimator gains and the RTT sampling rate (explained in Section 2.2) have little influence on the Lite-Xmit-Timer's performance. We disagree with those two conclusions. On the contrary, we show in Section 3.2 and Section 5.1 that, in some cases, the RTT sampling rate greatly influences the Lite-Xmit-Timer's performance. Furthermore, we show in Section 3.2 and Section 4.2 that when the RTT sampling rate is high and the TCP sender's load is large, the choice of the estimator gains and the variation weight becomes crucial. Moreover, we argue that the definition of the estimator gains and the variation weight should depend on the RTT sampling rate.

The rest of the paper is organized as follows. Section 2 describes our model- and measurement-based analysis approaches. We use the model to analyze the Lite-Xmit-Timer and explain its problems in Section 3. We further apply the model to develop the Eifel-Xmit-Timer in Section 4. We use measurements in an experimental network to validate our model-based analysis, and also to validate our implementation of the Eifel-Xmit-Timer. This is explained in Section 5. Section 6 summarizes our conclusions and outlines our current and future research.

2. Analysis Methodology

We develop a model of the class of network-limited TCP bulk data transfers in steady state which we describe in Section 2.1 and Section 2.2. In Section 2.3, we describe the measurement setup that was used for validation purposes.

2.1 Choosing a "typical" TCP Connection

TCP's operation and performance is largely determined by the path's metrics such as available bandwidth, end-to-end delay, and packet drop pattern. Ideally, a well-designed retransmission timer should perform well over any possible end-to-end path. In the Internet, however, those path metrics can vary considerably over short and long time scales [18]. Consequently, *the* typical TCP connection does not exist. This makes it particularly difficult to validate the design of a an end-to-end retransmission timer. Our approach is therefore to study *one* common class of TCP connections which is frequently found in the Internet, yet, is simple enough to allow for a model-based analysis.

A TCP sender's *load*, i.e., the number of segments outstanding at a given time, is either limited by the flow control imposed by the receiver or by the congestion control (implicitly or explicitly) imposed by the network[2]. Accordingly, one refers to such connections as being *receiver-* or *network-limited*.

2. In addition, a TCP sender's load may also be limited by the size of the TCP sender's send buffer.

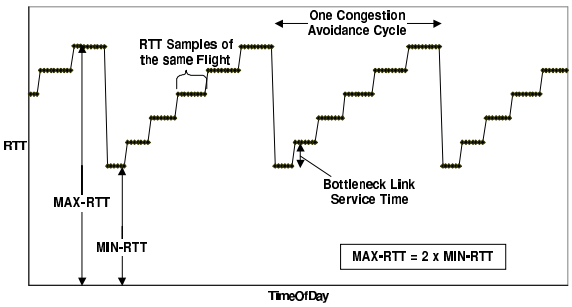

Figure 1: The RTT in steady state.

We study the class of network-limited TCP bulk data transfers in steady state. In this case the sender goes through periodic congestion avoidance cycles during which it linearly increases the load on the network until it receives a congestion signal. It then halves the load which effectively means that it does not send any more segments for one half the RTT. This gives the queue at the bottleneck link time to drain. We further assume a non-shared bottleneck link with a fixed bandwidth and a fixed bottleneck buffer size. The sender always sends fixed size segments. In addition, we assume that the sender fully utilizes the bottleneck link at any point in time. The latter has the effect that whenever the sender increases the load by one segment, that this will increase the queue length at the bottleneck by one. Consequently, the RTT increases by the segment's service time at the bottleneck link. It also yields a maximum RTT that is twice the minimum RTT as illustrated in Figure 1.

We refer to the segments a sender sends per RTT as a flight of segments or simply *flight*. For network-limited connections, a flight comprises those segments that are sent at a given load, i.e., the segments sent between load increases. Given our assumptions, the RTT of a given flight within one congestion avoidance cycle is the sum of the RTT of the preceding flight and a segment's service time at the bottleneck link (see Figure 1 where each dot in the graph denotes one RTT sample).

TCP connections that fulfill these assumptions can, e.g., be found in situations where the access link (e.g., low bandwidth dial-up or wide-area wireless) becomes the bottleneck link, and only a single application creates traffic. The analysis of a receiver-limited connection in such a situation is trivial as the RTT is constant in that case.

2.2 Model-based Analysis

Given an RTT that evolves in a deterministic and recurrent manner as outlined in Section 2.1, the RTO does also, as it is a function of RTT. Thus, we have chosen to model the RTT, the RTO, and all other relevant sender-side connection state variables on a spread sheet [14]. We make the following additional assumptions:

• *RTT sampling rate*
We refer to the RTT sampling rate as the number of RTT samples the TCP sender captures per RTT divided by the sender's load. In our model, we assume that every segment is timed to measure the RTT. In this case, the RTT sampling rate is 1 if the receiver acknowledges every segment, and it is 1/2 if the receiver uses delayed ACKs [3]. In "standard" TCP implementations only one segment per flight is timed, i.e., the RTT sampling rate is the reciprocal of the sender's load. The closer the RTT sampling rate is to 1 the more accurately the RTT is measured. Timing every segment is commonly implemented using the TCP timestamp option [10].

• *Explicit congestion signal*
We assume that congestion is signalled explicitly [21] at the end of each congestion avoidance cycle instead of through a dropped packet [1], [6], [8]. This simplifies the model-based analysis without limiting it.

• *Timer granularity*
To make our model independent of the impact of the timer granularity (discussed in Section 3.4) we model "time" in terms of ticks which can be arbitrarily defined.

On our spread sheet, columns correspond to a specific connection state variable (e.g., the RTT or the RTO) and rows correspond to the arrival of a new ACK, i.e., a new RTT sample. Thus, the "Time of Day" progresses from one row to the next by the bottleneck link's service time. The spread sheet has a number of parameters including the segment size, the bottleneck link's bandwidth and buffer size, and the end-to-end latency. Those are used to instantiate the spread sheet to reflect a specific connection, i.e., a specific evolution of RTT. In the following we refer to such an instantiation of the spread sheet as "the model". The mentioned parameters itself are less important for our analysis. What matters is the sender's load at the end of each congestion avoidance cycle. This is discussed in Section 3.2.

2.3 Measurement-based Analysis

Our goal is to reproduce a connection with characteristics as close as possible to a connection we can model using the technique and the assumptions described in Section 2.1 and Section 2.2. For that purpose, we used a single hop network consisting of two hosts running the BSD/386 Version 3.0 operating system that are inter-connected by a direct serial cable (see Figure 2). We used the the BSD Packet Filter [7], [17] to collect packet traces.

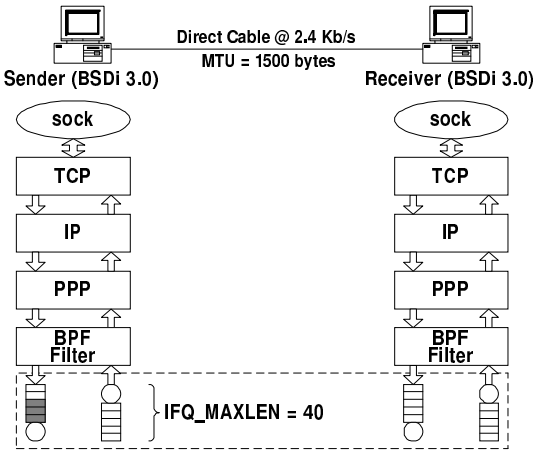

Figure 2: Measurement Setup.

In addition, we deliberately configured this setup to yield RTTs that are several multiples of the timer granularity implemented in BSD/386 (500 ms) to study the RTO at a sufficient resolution. This can be achieved by using large packets and a low bottleneck link speed to create large transmission delays, and by allowing large queueing buffers to create large queueing delays. In particular, we chose a link speed of 2.4 Kb/s, configured the maximum receive unit of the Point-to-Point Protocol [22] to 1500 bytes, and set the size of the interface buffer (IFQ_MAXLEN [24]) to 40 packets. With these settings, the RTT at the end of a congestion avoidance cycle is about 250 seconds[3] or 500 ticks (!). Although, we do not believe that such settings are commonly found, our conclusions are not materially affected by them. We could have obtained similar results by choosing a higher link speed and a smaller queueing buffer, but that would have required a lower timer granularity.

We always measured a single TCP connection at a time with the TCP timestamp option enabled. The transmission delay for a segment in this setup is too high to trigger delayed ACKs. Consequently, we always measured with an RTT sampling rate of one. The only difference to the model of this connection is that the TCP sender in the measurements had to rely on a dropped packet and the corresponding three duplicate ACKs [8] as the congestion signal. The minor impact of this difference is discussed in Section 5.

3. Identified Problems of the Lite-Xmit-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 conservative, the second problem makes it more aggressive. However, the latter is usually out-weighed by the other three factors.

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 [6] whether using the mean deviation in this strict manner is an appropriate design choice. The undesirable behavior this causes is that the predictor (RTO_L) "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.

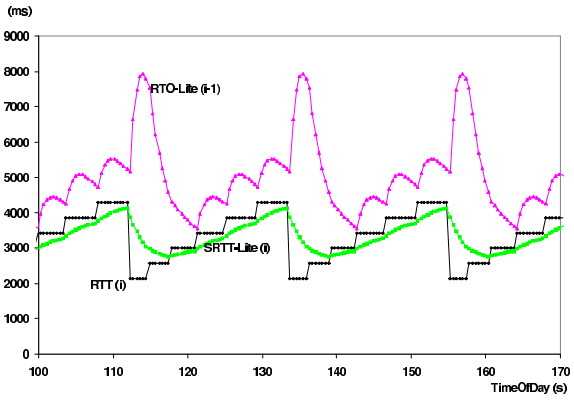

Figure 3: Prediction Flaw in RTO_L.

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 3 generated from the model described in Section 2.2. The model was configured to a sender's maximum load of 10 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}.

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 that case. However, if the RTT sampling rate is higher *and* the sender'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

3. 40 packets of 1500 bytes draining from the interface buffer at 240 bytes/s.

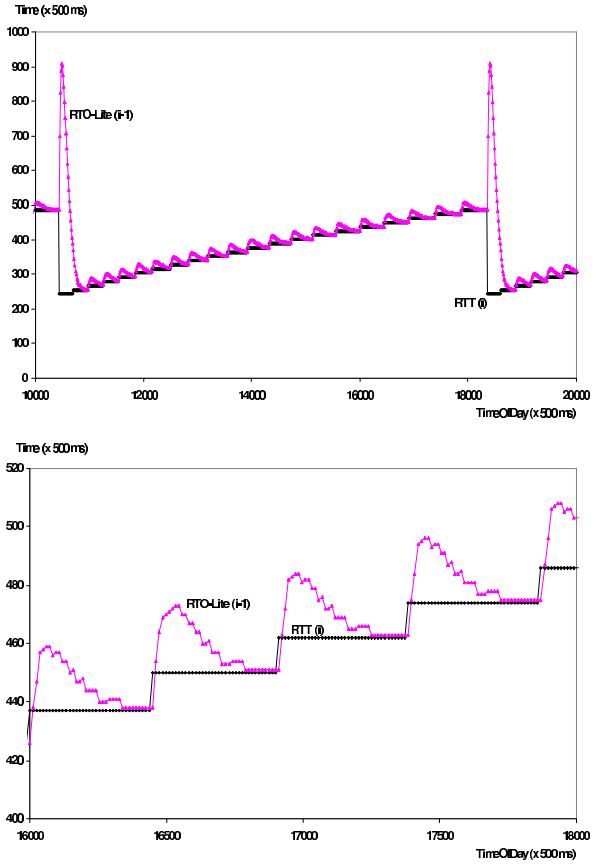

Figure 4: A Collapsed RTO_L (model).

$RTTVAR_L$ to decay too quickly. Thus, RTO_L collapses into the RTT, i.e., RTO_L becomes too aggressive. We illustrate this in Figure 4 where the lower graph is a "zoom" of the upper one. The graphs are based on the model configured to a sender's maximum load of 40 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 for the reasons explained in Section 3.3 and Section 3.4.

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, 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). This makes $REXMT_L$ significantly conservative. We have described this problem in [12].

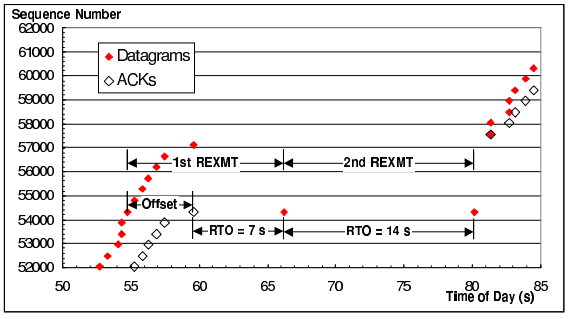

Figure 5: The "REXMT-Restart Bug".

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 than 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. That 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 (e.g., Solaris) have been implemented with a finer timer granularity. 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 $REXMT_L$ is that it is based on a so-called *heartbeat timer* provided by the BSD operating system. It expires *every* 500 ms, triggering a specific interrupt routine that updates the $REXMT_L$ (decrements it by one tick) of each active TCP connection. It does so independent of whether one of those $REXMT_L$ would actually go to zero or not. Simply increasing the frequency of the heartbeat timer would therefore 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. The heartbeat timer is also the reason for the minimum defined for RTO_L because a $REXMT_L$ of 1 tick can expire anywhere between 0 - 1 tick.

4. The Eifel-Xmit-Timer

Our motivation for developing the Eifel-Xmit-Timer is to eliminate the problems of the Lite-Xmit-Timer explained in Section 3. The RTO_E is defined by the following equations which we explain in the following sub-sections.

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

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

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

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

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

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

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

4.1 Predicting a Decreasing RTT

To avoid the problem described in Section 3.1, 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 6 using the same parameters chosen for the model discussed with respect to Figure 3.

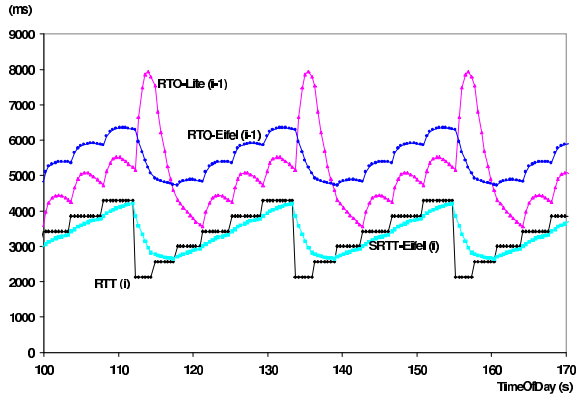

Figure 6: Fixing the Prediction Flaw with $RTTVAR_E$.

With this subtle change in the definition of RTTVAR, RTO_E does not exhibit the spikes seen with RTO_L when the RTT drops. Also, note that the graph of $REXMT_L$ (not shown in Figure 6 to not overload the plot) lies roughly one RTT "above" the graph of RTO_L because of the problem described in Section 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 4.5.

4.2 Scaling the Gains and the Variation Weight

To avoid the problem described in Section 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 sender'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 sender'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 sender'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 sender'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 $DELTA_E$[4].

Various alternatives exist to define $FLIGHT_E$. It is only important that it corresponds to the sender's load. In fact, one could define $FLIGHT_E$ as the actual 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 sender's load. The slow start threshold [6] (*SSTRESH*) is an appropriate candidate for that. In the common case the slow start threshold equals half the congestion window (*CWND*) [6] 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

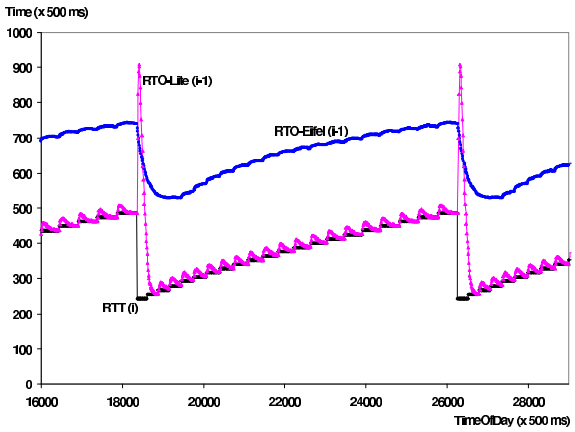

Figure 7: RTO_E scales the sender's load (model).

4. In those situations the minimum defined for RTO_E (see Section 4.4) would become effective. Thus, to be more conservative, one might also define the variation weight as $m/GAIN_E$ with m = 2, 3, 4, ...

lower bound of the sender's load. We add the constant 1 in the definition of $FLIGHT_E$ because the number of segments in the first flight 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 7).

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 6 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 sender'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 sender's load. We therefore multiply $GAIN_E$ 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 6).

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 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.

4.5 Implementing REXMT Precisely

Eliminating the problem described in Section 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 algorithm [8], i.e., when error recovery has to rely on the retransmission timer, $REXMT_E$ can greatly improve the end-to-end performance compared to $REXMT_L$.

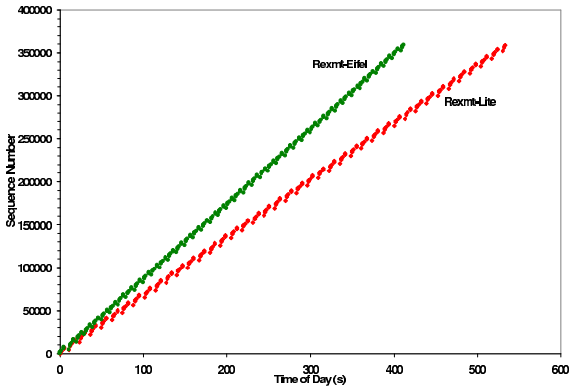

Figure 8: Restarting $REXMT_E$ precisely.

To demonstrate that we configured our experimental network described in Section 2 to a link speed of 9.6 Kb/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 8, we compare $REXMT_L$ with $REXMT_E$ using RTO_L in *both* cases to isolate the improvement that is achieved by restarting

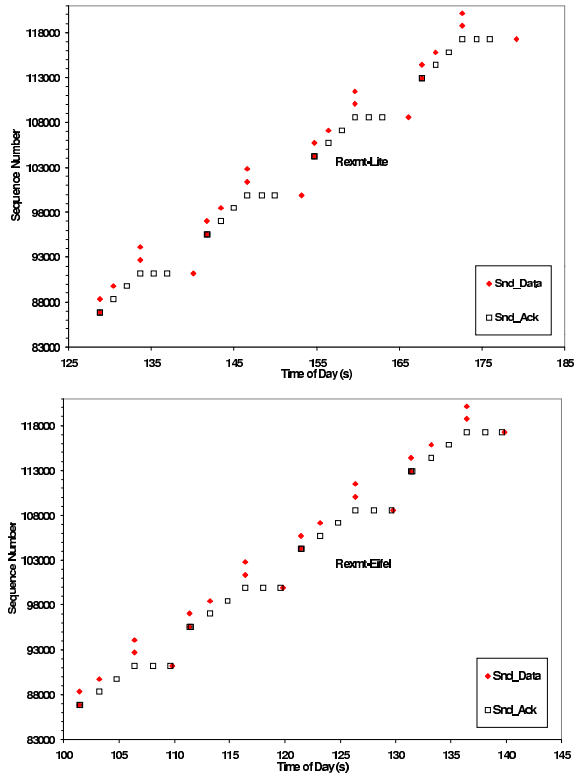

Figure 9: Zoom of the graphs shown in Figure 8.

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 9 shows a detailed view of sections of the two graphs shown in Figure 8. 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 retransmit algorithm, the Eifel algorithm [13] suppresses the fast retransmit, and restores the slow start threshold and the congestion window as if the timeout had not occurred.

4.6 Adapting to Spurious Timeouts

The Eifel algorithm [13] 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 single spurious retransmission. With that in mind and given that in steady state RTO_E/RTT is a fairly constant fraction, we can go beyond the given definition of RTO_E and multiply it with a factor smaller than one. That gets RTO_E closer to RTT. But what should the value of that factor be?

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

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

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

Instead of finding a constant factor, we experiment with the idea of having the factor adapt to the number of spurious timeouts that occur during the lifetime of a connection. We let 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} using an adaptive factor which we call *AGG* (aggressive).

CYCLE is the well known formula (e.g., see [16]) 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 10, based on the model configured to a sender's maximum load of 26 (= *MAXCWND*), 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.

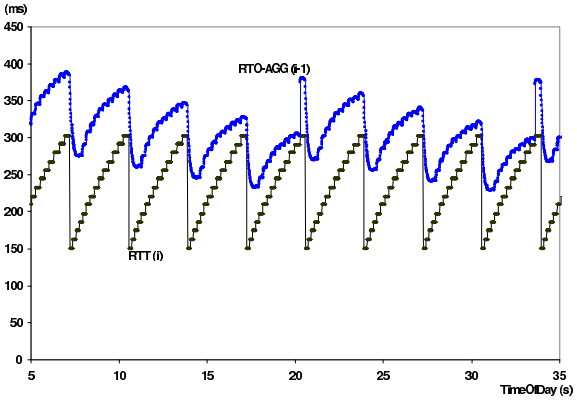

Figure 10: A self-trained RTO.

5. Measurement-based Analysis

To validate that the model described in Section 2.2 and applied in Section 3 and Section 4, we performed the measurements described in Section 2.3. In addition, we performed measurements to study RTO_L and RTO_E in case only one RTT sample is collected per RTT.

5.1 Collecting only a single RTT Sample per RTT

To see how RTO_L performed when only a single RTT sample was collected per RTT, we repeated the measurement described in Section 2.3 while disabling the timestamp option. The result is shown in Figure 11. Although the spikes in the graph of RTO_L still occur for the reason described in Section 3.1, at least the estimators gains and the variation weight work. Thus, the problem described in Section 3.2 only occurs when the RTT sampling rate is one or close to one. Figure 12 shows the same situation for RTO_E. The graph of RTO_E does not look much different from that of RTO_L in Figure 11, except that it does not have those spikes at the end of a congestion avoidance cycle.

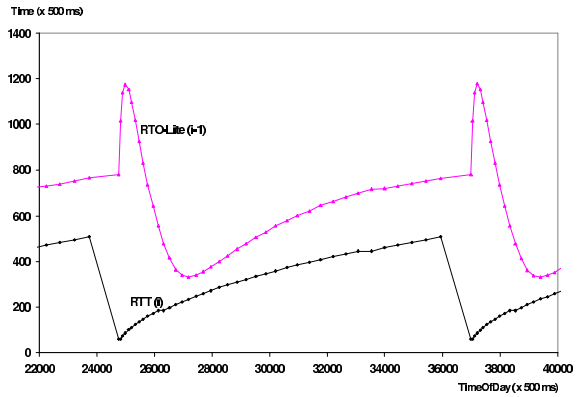

Figure 11: RTO_L when timing one segment per RTT.

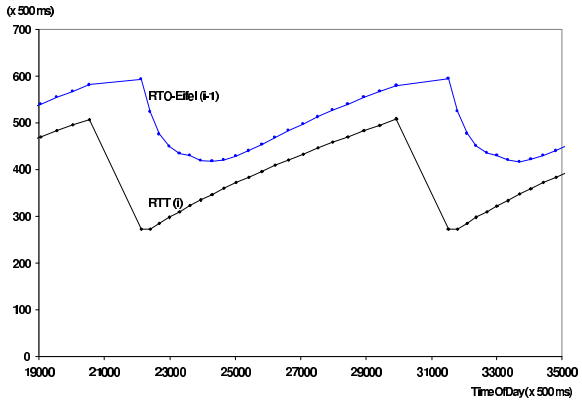

Figure 12: RTO_E when timing one segment per RTT.

Another phenomena can be seen when comparing Figure 11 and Figure 12. Although the maximum RTT is about 250 seconds 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.

5.2 Validating the Model

As a validation of the model we decided to reproduce the plots shown in Figure 4 which were generated from the model. Thus, we chose the parameter settings for our measurement setup as described in Section 2.3. Figure 13 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 3.2.

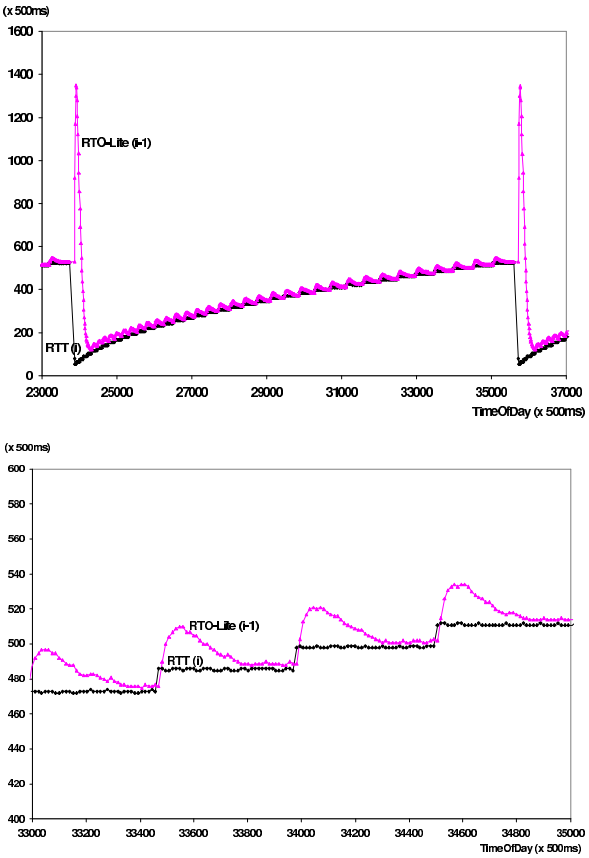

Figure 13: A Collapsed RTO_L (measured).

5.3 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 7 which was generated from the model. Again, we chose the parameter settings for our measurement setup as described in Section 2.3. Figure 14 shows the measurement result. A comparison yields a close match. Given that we know from Section 5.2 that the model is correct, we now have also validated that the implemen-

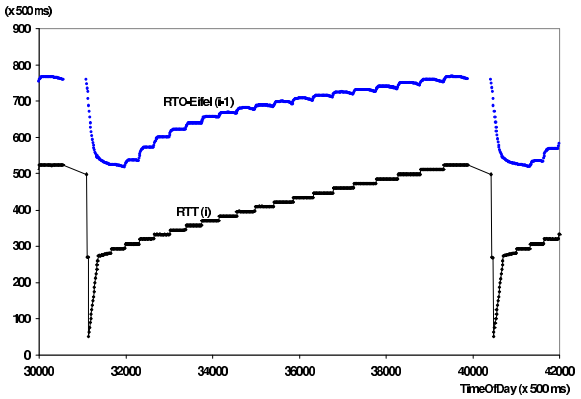

Figure 14: RTO_E scales with sender's load (measured).

tation of RTO_E is correct in the sense that it conforms to the definition of RTO_E provided at the beginning of Section 4.

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 triggered the fast retransmit and fast recovery algorithm. No valid RTT 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.

6. Conclusion and Future Work

In this paper, we analyzed two alternative retransmission timers for TCP. We first studied the Lite-Xmit-Timer which is the retransmission timer found in most implementations of TCP today. After revealing four major problems with the Lite-Xmit-Timer, we propose an alternative retransmission timer we call the Eifel-Xmit-Timer. It eliminates these problems.

However, we do not claim that the Eifel-Xmit-Timer is a sufficiently mature solution at this stage. More testing and verification under various network conditions, e.g., following the approach suggested in [2], is certainly required. We encourage further research in this area, and have therefore made our model [14], and our implementation [15] of the Eifel-Xmit-Timer publicly available. Still, our analysis allowed us to draw a number of conclusions that apply in general to any future end-to-end retransmission timer:

- RTT samples that fall below the smoothed RTT estimator (SRTT) should not be used to update the smoothed RTT deviation estimator (RTTVAR).
- The estimator gains and the variation weight need to be dependent on the RTT sampling rate.
- If every segment is timed to measure the RTT, e.g., by using the timestamp option [10], the estimator gains and the variation weight need to be scaled with the sender's load.

Apart from correcting the problems of the Lite-Xmit-Timer, we have proposed a new retransmission timer feature. The idea is to let the RTO become increasingly aggressive, i.e., let it converge to RTT, while adapting it to the number of spurious timeouts that occur during the lifetime of a connection. This feature relies on the use of the Eifel algorithm [13]. The Eifel algorithm allows to detect whether a timeout was spurious, and minimizes the number of spurious retransmissions. Especially interactive request/response-style applications will benefit from the quicker loss recovery provided by a more aggressive RTO. However, more research is needed to find the right level of aggressiveness of such an RTO.

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

In our future research we plan enhance our model to work off an arbitrary RTT evolution. We will then further verify the Eifel-Xmit-Timer by gathering "real world" RTT traces which we then analyze with the enhanced model.

Acknowledgments

Many thanks to Sally Floyd, Vern Paxson, Ramesh Govindan, and the CCR reviewers for comments on earlier versions of this work.

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*, In Proceedings of ACM SIGCOMM 99.

[3] R. Braden, *Requirements for Internet Hosts - Communication Layers*, RFC 1122, October 1989.

[4] L. S. Brakmo, L. L. Peterson, *Performance Problems in BSD4.4 TCP*, ACM Computer Communication Review, 25(5), October 1995.

[5] L. S. Brakmo, L. L. Peterson, *TCP Vegas: End to End Congestion Avoidance on a Global Internet*. IEEE Journal of Selected Areas in Communication, Vol. 13, No. 8, October 1995.

[6] V. Jacobson, M. J. Karels, *Congestion Avoidance and Control*, Revised version of a paper that appeared in Proceedings of ACM SIGCOMM 88, available at http://ee.lbl.gov/, 1992.

[7] V. Jacobson, C. Leres, S. McCanne, `tcpdump`, available at http://ee.lbl.gov/.

[8] V. Jacobson, *Modified TCP Congestion Avoidance Algorithm*, Email to the end2end-interest mailing list, April 30, 1990, available at ftp://ftp.ee.lbl.gov/email/vanj.90apr30.txt.

[9] V. Jacobson, C. Leres, S. McCanne, `tcpdump`, available at http://ee.lbl.gov/.

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

[11] P. Karn, C. Partridge, *Improving Round-Trip Time Estimates in Reliable Transport Protocols*, In Proceedings of ACM SIGCOMM 87.

[12] R. Ludwig, *A Case for Flow-Adaptive Wireless Links*, Technical Report UCB//CSD-99-1053, University of California at Berkeley, May 1999.

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

[14] R. Ludwig, *Model of the TCP Sender Connection State in Equilibrium*, available at http://iceberg.cs.berkeley.edu, January 1999.

[15] R. Ludwig, *TCP-Eifel*, Patches for FreeBSD, available at http://iceberg.cs.berkeley.edu, October 1999.

[16] M. Mathis, J. Semke, J. Mahdavi, T. Ott, *The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm*, ACM Computer Communications Review, 27(3), July 1997.

[17] S. McCanne, V. Jacobson, *The BSD Packet Filter: A New Architecture for User-Level Packet Capture*, In Proceedings of the 1993 Winter USENIX Conference.

[18] V. Paxson, *Measurements and Analysis of End-to-End Internet Dynamics*, Ph. D. dissertation, University of California, Berkeley, April 1997.

[19] J. Postel, *Internet Protocol*, RFC 791, September 1981.

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

[21] K. K. Ramakrishnan, S. Floyd, *A Proposal to add Explicit Congestion Notification (ECN) to IP*, RFC 2481, January 1999.

[22] W. Simpson, *The Point-to-Point Protocol*, RFC 1661, July 1994.

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

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