The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions

Reiner Ludwig
Ericsson Research
Herzogenrath, Germany

Randy H. Katz
Computer Science Division
University of California at Berkeley

Abstract

We propose an enhancement to TCP's error recovery scheme, which we call the *Eifel algorithm*. It eliminates the retransmission ambiguity, thereby solving the problems caused by spurious timeouts and spurious fast retransmits. It can be incrementally deployed as it is backwards compatible and does not change TCP's congestion control semantics. In environments where spurious retransmissions occur frequently, the algorithm can improve the end-to-end throughput by several tens of percent. An exact quantification is, however, highly dependent on the path characteristics over time. The Eifel algorithm finally makes TCP truly wireless-capable without the need for proxies between the end points. Another key novelty is that the Eifel algorithm provides for the implementation of a more optimistic retransmission timer because it reduces the penalty of a spurious timeout to a single (in the common case) spurious retransmission.

1. Introduction

Internet traffic today is still largely generated by applications that use the Transmission Control Protocol (TCP) [23] as the underlying communications protocol. A great deal of research and engineering went into TCP over the past two decades which has largely contributed to the Internet's success. TCP is a stable, mature, and probably the most thoroughly tested protocol of its kind. Nevertheless, there are some corner cases where TCP could still be improved.

A TCP receiver sends two types of (cumulative) acknowledgements. It sends positive acknowledgements (ACKs) for segments that are received correctly and in-order, and it sends duplicate acknowledgements (DUPACKs) for segments that are received correctly but out-of-order. A DUPACK acknowledges the same sequence number that the last sent ACK acknowledged. Thus, a DUPACK does not convey *which* segment was received correctly (unless Selective Acknowledgement Options [18] are used).

The TCP sender uses two different error recovery strategies: (1) timeout-based retransmission, and (2) DUPACK-based retransmission. In the latter case a retransmission - a so-called *fast retransmit* - is triggered when three[1] successive DUPACKs for the same sequence number have been received, i.e., without waiting for the retransmission timer to expire [9]. Both error recovery strategies are coupled with TCP's congestion control scheme [1], [8], [9] in the following way. After a timeout-based retransmission, the TCP sender decreases its *load*, i.e., the maximum number of unacknowledged segments it may send into the network per *Round-Trip Time (RTT)* [25], to one segment. It then enters the *slow-start* phase during which it increases the load exponentially until the load reaches one half of what its value was before the timeout occurred. The TCP sender then enters the *congestion avoidance* phase, where it increases the load linearly. After a DUPACK-based retransmission, the TCP sender halves its load, and immediately enters the congestion avoidance phase. This behavior is justified because a packet loss usually indicates congestion somewhere along the path and a timeout indicates more severe congestion. Congestion control in TCP is implemented through the use of two state variables, *slow start threshold* and *congestion window*, with which we expect the reader to be familiar (see [25]).

The problem of *spurious timeouts*, i.e., timeouts that would not have occurred had the sender waited "long enough", is an example of the above mentioned corner cases. Spurious timeouts have not generally been a concern in the past. They are rare over all wireline paths [21], as well as on path's that include reliable wireless links that do not lose connectivity [14]. This is due to TCP's conservative retransmission timer [2], [15]. However, we believe that the problem will occur more frequently with the increasing number of hosts accessing the Internet via wide-area packet-radio networks. Frequent disconnections - on the order of seconds - without losing data are not only common in these networks, but are explicitly accounted for in their design. Over such links spurious timeouts in TCP are likely to be more frequent.

The problem of *spurious fast retransmits* is another corner case. These occur when packets get re-ordered beyond the DUPACK-Threshold in the network before reaching the receiver. It is difficult to evaluate how serious this is in the Internet today. Some studies [21] conclude that this event occurs rarely, while others [4] find this problem to be more serious. Clearly, this depends on the paths underlying such studies, e.g., whenever routers are inter-connected via multiple links/paths (for fault tolerance) and load balancing is performed across those links/paths on the aggregate traffic, packet re-orderings will occur more frequently.

We show that spurious fast retransmits and the go-back-N retransmission behavior triggered by spurious tim-

1. Note, that most implementations define a DUPACK-Threshold. However, that threshold is commonly set to three.

eouts have the same root: the *retransmission ambiguity* [13], i.e., a TCP sender's inability to distinguish an ACK for the original transmission of a segment from the ACK for its retransmission. Throughout this paper we use the term *spurious retransmission* when it does not make a difference whether it was triggered by a spurious timeout or a spurious fast retransmit. Our *Eifel*[2] *algorithm* uses extra information in the ACKs to eliminate the retransmission ambiguity, thereby solving the problems caused by spurious retransmissions. This algorithm is the main contribution and the subject of this paper.

The rest of the paper is organized as follows. In Section 2, we describe the measurement platform and methodology we used for our study. Section 3 provides a detailed description of how spurious timeouts and spurious fast retransmits affect TCP's protocol operation. In Section 4, we develop and explain the Eifel algorithm. In Section 5, we argue why it is impossible to come up with a reasonable performance evaluation that quantifies potential throughput improvements that the Eifel algorithm could achieve. Section 6 summarizes our conclusions and outlines our current and future research.

2. Methodology

We are interested in studying specific effects in TCP's operation. We therefore setup a "clean" environment in which measurements are not blurred by uncontrolled effects like delay variations, or packet losses commonly found in the Internet. Nevertheless, these effects are found in real networks, but are hard to "catch". Interpreting the resulting trace plots is usually more difficult because of the mentioned effects.

We used a single hop network for our experiments consisting of two hosts (BSDi 3.0) inter-connected via a direct cable connection running the Point-to-Point Protocol [24] at 9.6 Kb/s with a maximum transmission unit [25] of 512 bytes. In all measurements the TCP timestamp option [11] was enabled. Thus, although the *Maximum Segment Size (MSS)* [25] was 472 bytes (40 bytes for the TCP/IP header), only 460 bytes of user data was sent per segment (leaving 12 bytes of space required for the timestamp option). The TCP receiver advertised a window of 8496 bytes (18 x MSS).

We developed a tool called `hiccup`[3] to trigger spurious timeouts and/or spurious fast retransmits. Depending on the parameters specified by a user-level process, `hiccup` operates on a given interface in the inbound, outbound, or both directions, and generates transient delays by queueing packets, or re-orders packets. The location of `hiccup` in the protocol stack is important to understand the trace plots in this paper (e.g., see Figure 3). Outbound packets queued by `hiccup` are logged as a single burst by the BSD Packet Filter [10], [20] although they have *not* been sent as a burst by the TCP sender. Those packets are clocked out separately by the TCP sender each time an ACK arrives (marked as + in the trace plots), but then get queued by `hiccup`. At that point those packets are not logged by the packet filter. That is done after the transient delay is over, and `hiccup` flushes the queue of packets into the outbound interface buffer. The packets are then spread out in time due to the transmission delay on the outgoing link before they are received by the TCP receiver.

2. *The Eifel* is the name of a beautiful mountain range in Western Germany.
3. `hiccup` was implemented by Keith Sklower at U.C. Berkeley.

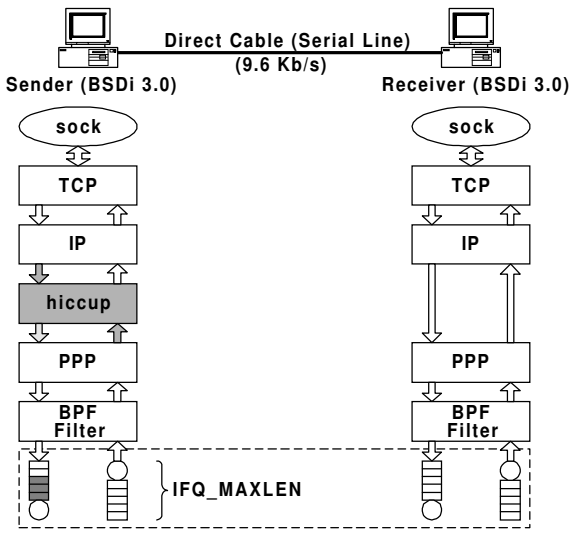

Figure 1: Measurement Setup.

A connection's bandwidth-delay product [25] is the sum of two components: (1) the *pipe capacity*, i.e., the minimum number of segments a sender needs to have in transit to fully utilize its share of bandwidth available at the bottleneck link, and (2) the *bottleneck queue*, i.e., the number of segments a connection contributes to the queue at the bottleneck link. In general, a TCP sender has no way to determine the pipe capacity nor the bottleneck queue separately but can only determine the sum. We always measured a single connection at a time, and the pipe capacity was always two segments. The size of the interface buffer (IFQ_MAXLEN [26]) which in BSD-derived systems is maintained in terms of Internet Protocol (IP) [22] packets was used to limit the bottleneck queue of the connection. For example, an interface buffer size of 12 allows 12 packets to be queued before a packet (tail-)drop occurs. We used the interface buffer size to trigger certain effects explained below.

3. The Problem

In this section, we provide a detailed description of how spurious timeouts and spurious fast retransmits affect TCP's protocol operation. For that purpose, we use trace plots that provide an excellent means to visualize a protocol's operation over time correlated with effects occurring in the network, such as (excessive) packet delay or packet re-ordering.

3.1 Spurious Timeouts

A *retransmission timer* is a prediction of the upper limit of the RTT. In common TCP implementations, an adaptive retransmission timer accounts for RTT variations [8]. 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

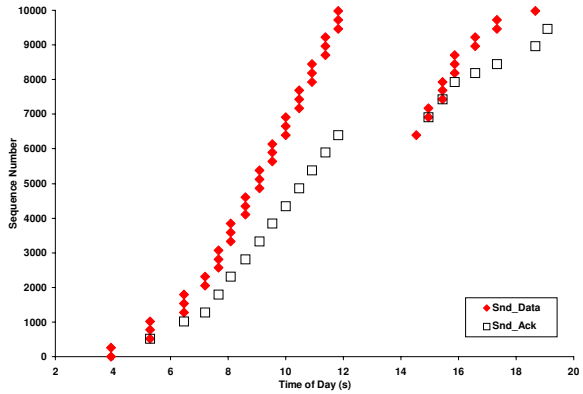

Figure 2: Go-back-N after a spurious timeout. (sender trace)

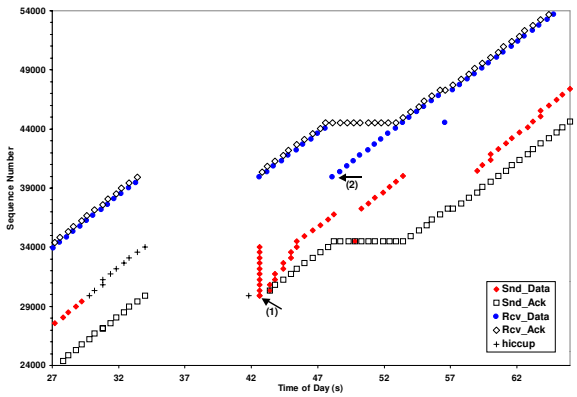

Figure 3: Go-back-N after a spurious timeout (sender and receiver traces)

or rapidly increasing congestion at the bottleneck link. The latter can in turn be caused by routing table updates [21] or a reliable link layer protocol running over a wireless link on which the radio quality (temporarily) dropped [14]. 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 latter is also described in [15].

The fundamental problem which leads to the go-back-N retransmissions is the retransmission ambiguity mentioned in Section 1. This is depicted in Figure 2, taken from [3], showing a trace plot of a spurious timeout which was caused by a reliable link layer protocol running over a wireless link. Shortly after the timeout (14.5 seconds 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 that is received after the timeout (about 400 ms) is smaller than the minimum possible RTT on the network underlying the trace.

To verify this finding, we performed bulk data transfers on the experimental network described in Section 2 and used `hiccup` to trigger a spurious timeout. This time we also traced the TCP receiver. As with all subsequent trace plots in this paper, both the sender and receiver trace are correlated in time and shown in the same plot (see Figure 3). The receiver trace is offset by 10,000 bytes to not overlap with the sender trace. The clocks of the sending and the receiving host were *not* synchronized, because the exact timing of events was not necessary for our study. Instead, we loosely synchronized the sender and the receiver traces by defining as "time zero" the time when the sender sends the connection establishment request (SYN) and when the receiver gets it. Thus, apart from clock drifts on both hosts, the receiver trace is offset by the one-way delay of the initial SYN.

At 29.8 seconds into the connection we called `hiccup` to intercept and queue outbound packets for 13 seconds. During this time, all outstanding segments are being acknowledged and each of those ACKs clocks out a new segment (marked as + in Figure 3). However, those segments are not logged by the packet filter until `hiccup` has terminated in second 42.6, when they get placed into the outbound interface buffer all at once. At that time, the sender has already performed one retransmission (marked as + in Figure 3) which was also queued by `hiccup` and can therefore only be seen in the receiver trace (see arrow (2) in Figure 3). The original transmission and the retransmission of that segment are the same point in the sender trace (see arrow (1) in Figure 3). Then the go-back-N retransmission is triggered as described before with respect to Figure 2. Returning to Figure 3, 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 is described in the next section.

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

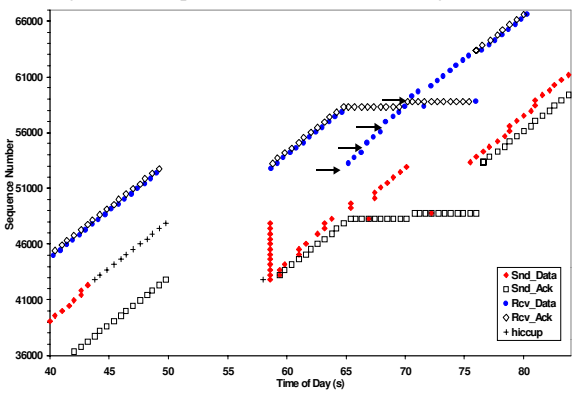

Figure 4: Go-back-N after a spurious timeout causing a *real* packet drop.

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 4. To show this effect we set the interface buffer size to 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 4).

3.2 Spurious Fast Retransmits

The IP protocol is connection-less and as such does not guarantee *in-order delivery* of packets. That is, the sequence of packets as generated by the source does not need to be preserved when the packets are delivered to the destination. That responsibility is left to TCP. In particular, packets belonging to the same connection may take different routes to the destination and in practice sometimes do [21]. The fact that IP does not need to preserve the packet order also allows for reliable link layer protocols to perform *out-of-order delivery* of correctly received IP packets [15]. That provides for more memory-efficient link layer implementations.

We speak of a *packet re-ordering* event when a packet arrives at the receiver after one or more packet(s), that had left the source later, have already arrived. We call the number of packets that had already arrived out-of-order, the *re-ordering length*. For example, if packets 1 - 10 are sent but packet 1 arrives last, then the re-ordering length is 9. 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 the mentioned spurious fast retransmits. Spurious fast retransmits affect TCP performance in that the TCP sender unnecessarily reduces its load and unnecessarily retransmits a segment, i.e., performs a spurious retransmission.

To illustrate a spurious fast retransmit, we performed bulk data transfers on the experimental network described in Section 2 and used `hiccup` to cause a packet re-ordering event with a re-ordering length of six. This is depicted in Figure 5. The packet which was supposed to be sent at second 37.7 (marked as + in Figure 5) is queued by `hiccup` while the succeeding six packets are let through. Then `hiccup` sends

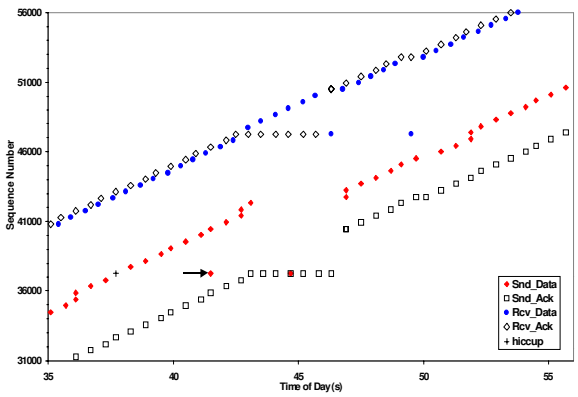

Figure 5: The effect of packet re-ordering.

the single queued packet (see arrow in Figure 5) 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 first ACK after the series of DUPACKs arrives at the sender[4]. 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.

4. The Eifel Algorithm

Once we understand the problem, the solution is straightforward: first, eliminate the retransmission ambiguity and then, restore the load and resume transmission with the next unsent segment. We call this the Eifel algorithm, which we briefly described in [15]. We have implemented and tested it under FreeBSD, and have made it publicly available [17]. The current implementation is based on the use of the TCP timestamp option [11] 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.

4.1 Eliminating the Retransmission Ambiguity

Eliminating 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 3.1. Waiting for the receiver to signal in DUPACKs that is has correctly received duplicate segments, as proposed in [6], would be too late (see Figure 3), and is thus not an alternative.

The TCP timestamp option provides exactly what we need. When using the timestamp option the TCP sender writes the current value of a "timestamp clock" into the header of each outgoing segment. The receiver then echos those timestamps in the corresponding ACKs according to the rules defined in [11]. Eliminating the retransmission ambiguity is then implemented as follows. The sender always stores the times-

4. 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.

tamp of the *first* retransmission independent of whether that was triggered by an expiration of the retransmission timer or by the receipt of three consecutive DUPACKs[5]. 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 eliminate the retransmission ambiguity is one implementation alternative. Which mechanism is implemented for that purpose does not make a difference for the Eifel algorithm. However, including the 12 bytes TCP timestamp option field in every segment and the corresponding ACKs seems heavy weight[6]. 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 [7]. 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 [5], [12].

4.2 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 7). However, various reasons can cause a TCP sender to send packet bursts (e.g., see Footnote 4). 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 more spurious retransmissions of the oldest outstanding segment 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.

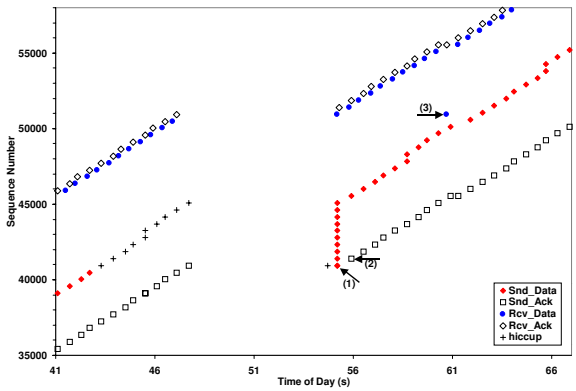

Figure 6: Response after spurious timeout.

Figure 6 illustrates the operation of the Eifel algorithm in the event of a spurious timeout. At 43.3 seconds into the connection we called `hiccup` to intercept and queue outbound packets for 12 seconds (marked as + in Figure 6). 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 6) 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 6) carries a timestamp of 43.3 seconds 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 7 illustrates the operation of the Eifel algorithm in the event of a spurious fast retransmit. At 40.5 seconds into the connection we called `hiccup` to cause a packet re-ordering event with a re-ordering length of six (marked as + in Figure 7). Then `hiccup` sends the single queued packet (see arrow (1) in Figure 7) 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 7) carries a timestamp of 40.5 seconds 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

5. Unlike in [26] the Eifel algorithm counts a fast retransmit as a "regular" retransmission (`t_rxtshift++`).
6. Another viable alternative has been proposed to the authors which is to use timestamps *only* for retransmissions and their corresponding ACKs.

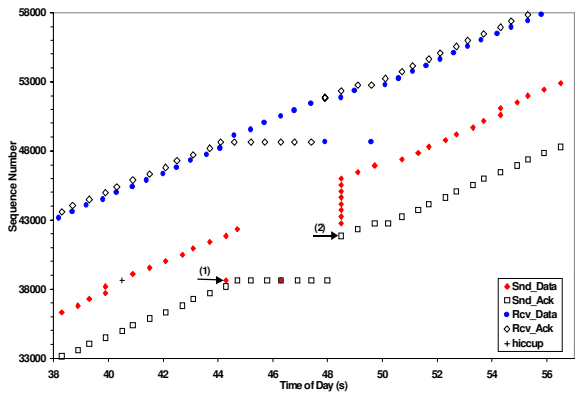

Figure 7: Response after a spurious fast retransmit.

threshold and the congestion window. This causes the packet burst mentioned at the beginning of this subsection.

5. 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 Eifel algorithm's throughput improvement 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 8.

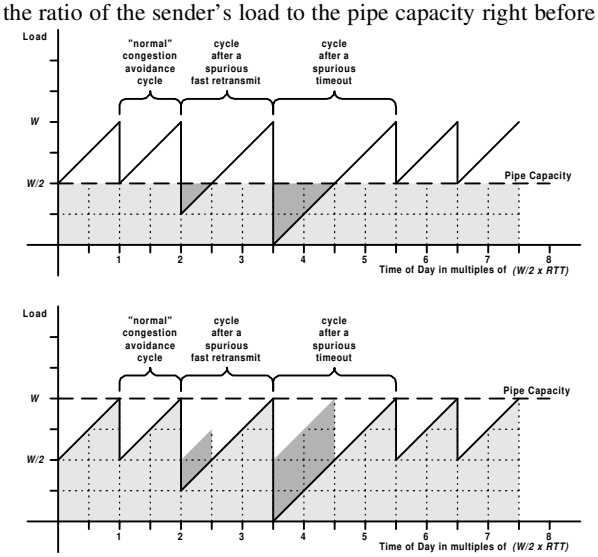

Figure 8: Approximating the performance loss.

A network-limited TCP connection in steady state goes through periodic congestion avoidance cycles. 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 (e.g., low bandwidth dial-up or wide-area wireless) becomes the bottleneck link, and only a single application creates traffic. In fact, this scenario is emulated by the setup described in Section 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 [19]). The pipe capacity is assumed to be equal to *W/2* in the upper graph of Figure 8, 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 8. 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 3 and Figure 5). 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.

6. Conclusion and Future Work

In this paper, we propose 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 retransmission

ambiguity, and uses that to solve the problems caused by spurious timeouts and spurious fast retransmits. 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.

We see the major benefit of the Eifel algorithm in wireless environments. In [16] we showed that the end-to-end performance of reliable flows, such as those based on TCP, can only be optimized by running persistent link layer error recovery. That assumes that delay-sensitive flows are queued and transmitted separately. 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. Thus, with the Eifel algorithm implemented in TCP and an appropriately designed link layer, the well-known problem of "TCP over lossy links" is eliminated.

A general key advantage of the Eifel algorithm is that it opens 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. In our current and future work we therefore study alternative implementations of TCP's retransmission timer. One of our ideas is to make the retransmission timer adaptive to the measured fraction of spurious timeouts. In preparation for this work, we have included in the Eifel algorithm a feature we have not discussed in this paper. 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.

Acknowledgments

Many thanks to Keith Sklower for implementing `hiccup` and the many private "BSD kernel tutorials". Many thanks to Michael Meyer, Stephan Baucke, Sally Floyd, and Vern Paxson for comments on earlier versions of this paper.

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] S. Baucke, *Leistungsbewertung und Optimierung von TCP für den Einsatz im Mobilfunknetz GSM*, Diploma Thesis, CS-Dept. 4, Aachen University of Technology, Germany, April 1997.

[4] J.C.R. Bennett, C. Partridge, N. Shectman, *Packet Reordering is Not Pathological Network Behavior*, Appears in IEEE/ACM Transactions on Networking, December '99.

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

[6] S. Floyd, J. Mahdavi, M. Mathis, M. Podolsky, A. Romanow, *An Extension to the Selective Acknowledgement (SACK) Option for TCP*, work in progress, August 1999.

[7] ISO/IEC, *Information processing systems - Open Systems Interconnection - Connection oriented transport protocol specification*, International Standard ISO/IEC 8073, December 1988.

[8] V. Jacobson, *Congestion Avoidance and Control*, In Proceedings of ACM SIGCOMM 88.

[9] V. Jacobson, *Modified TCP Congestion Avoidance Algorithm*, end2end-interest mailing list (ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail), April 30, 1990.

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

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

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

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

[14] R. Ludwig, B. Rathonyi, A. Konrad, K. Oden, A. Joseph, *Multi-Layer Tracing of TCP over a Reliable Wireless Link*, In Proceedings of ACM SIGMETRICS 99.

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

[16] R. Ludwig, A. Konrad, A. D. Joseph, R. H. Katz, *Optimizing the End-to-End Performance of Reliable Flows over Wireless Links*, Appears in Wireless Networks (Special issue: Selected papers from ACM/IEEE MOBICOM 99).

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

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

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

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

[21] V. Paxson, *End-to-End Routing Behavior in the Internet*, IEEE/ACM Transactions on Networking, Vol.5, No.5, pp. 601-615, October 1997.

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

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

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

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

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