Evaluating the Eifel Algorithm for TCP in a GPRS Network

Andrei Gurtov
University of Helsinki – Finland
e-mail: Andrei.Gurtov@cs.Helsinki.FI

Reiner Ludwig
Ericsson Research – Germany
e-mail: Reiner.Ludwig@Ericsson.com

ABSTRACT

Large and sudden variations in packet transmission delays are often unavoidable in GPRS. This may cause spurious timeouts in TCP. 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 Eifel algorithm avoids these consequences. We evaluate the performance of the Eifel algorithm for TCP Reno, NewReno and SACK in a simulated GPRS network. We use throughput and goodput as equally important performance metrics. In all our simulations, we find that the Eifel algorithm improves goodput; in some cases by up to 20 percent. When complemented with an efficient loss recovery scheme (SACK or NewReno), we find that the Eifel algorithm also improves bulk data download times in all our simulations; in some cases by up to 12 percent.

1. INTRODUCTION

An increasing number of mobile users access the Internet via data links provided by cellular wide area wireless networks such as the General Packet Radio Service (GPRS) [1]. GPRS is a packet-switched extension of the Global System for Mobile communications (GSM). While it is being actively deployed globally at the time of writing, GPRS is already operational in many countries. GPRS incorporates many physical and link layer techniques including different forward error correction schemes, Automatic Repeat reQuest (ARQ), power control, and frequency hopping that typically ensure a "smooth" data transmission. Nevertheless, large and sudden variations in packet transmission delays are often unavoidable. This often creates a problem for end-to-end protocols. In particular, the Transmission Control Protocol (TCP) [11] is not sufficiently robust to cope with such delay variations.

Our previous work discusses possible sources of delay spikes in the GPRS network [5]. Possible events that may cause suspension of a TCP connection on the order of seconds are link outages, handovers and radio resource preemption. Link outages can result from a transient loss of radio coverage, e.g., while driving through a tunnel or when using an elevator. During a handover the mobile terminal may have to perform time-consuming operations before data can be transmitted in the new cell. Blocking by high-priority traffic may occur when an arriving voice call or higher priority data user temporally preempts the radio channel.

Such events do not necessarily cause packet losses since GPRS implements a rather persistent link layer retransmission scheme. However, the sudden delay spikes can cause TCP to timeout prematurely, and perform unnecessary retransmissions. This paper presents a quantitative evaluation of the Eifel algorithm for TCP [4] within the context of GPRS. The Eifel algorithm is a mechanism to detect and respond to spurious timeouts and spurious fast retransmits in TCP. We simulate bulk data connections of TCP Reno, NewReno [8] and SACK [2] while generating delay spikes that are typical for those caused by cell reselections in a GPRS network. Although the study could have been done in a live GPRS network, it would have been difficult to reproduce exactly the same sequences of cell reselections, and thus difficult to estimate the effect of Eifel. Additionally, it is difficult to find a flawless TCP implementation. We therefore used the NS2 [3] simulator. For that, we implemented a module that simulates a GPRS link and is able to replay traces of delay variations based on measurements taken in a live GPRS network.

An important observation is that for mobile users and operators the battery power consumption and radio resource usage are often as important as the download time over the wireless link. This suggests that the amount of data sent over the wireless link should be minimized. Thus, in our evaluating of the Eifel algorithm in GPRS, we use the download time and the goodput, i.e., the ratio of useful over total data transmitted, as equally important performance metrics.

The rest of the paper is organized as follows. In Section 2, we describe the cell reselection mechanism in GPRS. In Section 3, we explain the effect of delay spikes on TCP, and how the Eifel algorithm makes the TCP sender robust against the potentially resulting spurious timeouts. In Section 4, we describe the methodology and assumptions underlying our analysis. Our results are presented in Section 5. Section 6 concludes the paper and outlines our plans for future work.

2. CELL RESELECTION IN GPRS

In GPRS, the mobile terminal selects the serving cell. This is different from the basic circuit-switched GSM data service where the network controls the transfer of on-going data calls between cells [6][7]. The cell reselection process causes a delay, and sometimes packet losses in active data flows. The total delay

consists of the radio channel access delay in the Base Station Subsystem (BSS), and the delay caused by mobility management procedures in the core network; more precisely the Serving GPRS Support Node (SGSN).

Changing between cells that belong to the same base station controller can be typically done within the BSS without involving the SGSN, which reduces the delay. Some events in the network may cause cell reselection to be aborted and later restarted which significantly increases the delay. According to the GPRS specifications, a cell reselection should be completed within a few seconds. However, in a live GPRS network we observed that it can take any time from a few to a few tens of seconds.

The frequency of cell reselections is to a large extent determined by the speed of a user's movement and the size of cells. For example, driving in an urban area may cause frequent cell reselections. A typical interval between cell reselections in such a case is around a minute, but can be as small as few tens of seconds in densely populated environment.

To show that a delay spike caused by a cell reselection can indeed trigger a spurious timeout in TCP, we have performed a simple test. We took a laptop running Linux (RedHat version 6.2) connected via a Motorola Timeport GPRS phone to a live GPRS network. By forcing cell reselections from the phone and recording the TCP behavior using `tcpdump` [12], we have obtained the TCP trace plot shown in Figure 1. More details on reading TCP trace plots can be found in [4]. The first cell reselection occurs at 510 s when the TCP connection is in the slow start phase and takes 7 seconds. The second one occurs at 550 s during the congestion avoidance phase and takes 8 seconds. In both cases, TCP experiences a spurious timeout and performs unnecessary retransmissions.

3. THE EIFEL ALGORITHM

The Eifel algorithm proposed in [4] makes the TCP sender robust against spurious timeouts and packet reordering. In this section, we only explain the Eifel algorithm in the context of spurious timeouts. When a delay spike exceeds the current value of TCP's retransmission timer, a timeout occurs, and the TCP sender retransmits the oldest outstanding segment. If that segment or the corresponding ACK is only delayed but not lost, that retransmission was unnecessary and the timeout is said to be spurious. Figure 2 shows a spurious timeout for Reno TCP produced using the NS2 simulator. The receiver trace is offset by 25 segments to prevent an overlap with the sender trace. We enhanced the `hiccup` tool [9] to generate the delays in this test. The first retransmission is sent at second 6, and is also delayed. The sender interprets the ACK generated by the receiver in response to the original segment as corresponding to the retransmission. This happens because TCP ACKs bear no information that would allow the TCP sender to distinguish an ACK for the original segment from that for the retransmission. Likewise the TCP sender misinterprets the following original ACKs, and retransmits all outstanding segments using the slow start algorithm. Also, a number of new segments allowed by the congestion window are transmitted in this phase.

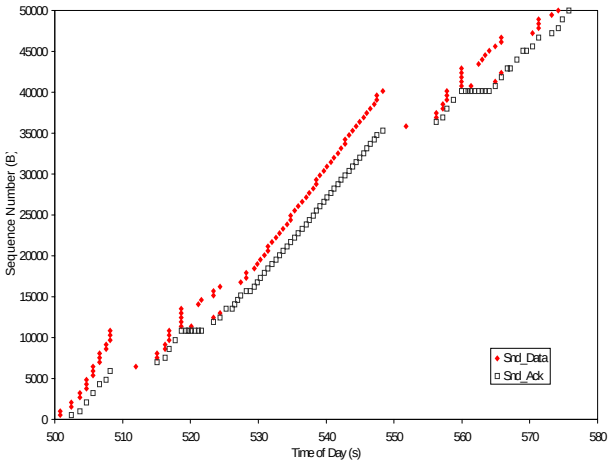

Figure 1. A TCP trace shows spurious retransmissions caused by two cell reselections in a live GPRS network.

At second 8 the retransmitted segments arrive at the TCP receiver and generate DUPACKs [13]. When the threshold of three DUPACKs is reached at the sender, a spurious fast retransmit is triggered since the TCP sender does not implement the careful version of the fast retransmit algorithm [8].

Figure 3 illustrates the operation of the Eifel algorithm in the event of a spurious timeout. The Eifel algorithm stores the timestamp of the first retransmission occuring at second 6. The first ACK that acknowledges the retransmission at second 8 carries a timestamp of 3 s which is when the original transmission of the corresponding segment took place. By comparison with the timestamp stored for the retransmission (6 s) the Eifel algorithm detects that the timeout was spurious.

The response to a spurious timeout in the original study [4] resumes transmission with the next unsent segment. How the congestion control state is reversed depends on the number of subsequent spurious timeouts. After the first timeout, the sender restores the slow start threshold and the congestion window to the values before the timeout. Figure 3 shows this situation when a delay spike occurs in the beginning of the connection in the slow start phase. After detecting the spurious timeout at second 8, the slow start phase continues. The behavior after two subsequent timeouts is shown later in Figure 7. In this case, the slow start threshold is set to the previous value of congestion window, which itself is left halved. In that case, a TCP sender ignores some of the original ACKs after a spurious timeout until the congestion window has sufficiently increased. After three and more subsequent spurious timeouts the congestion control state is not reversed at all.

In this paper, we evaluate the Eifel algorithm as proposed in [4] leaving the study of various enchancments to the response part for future work.

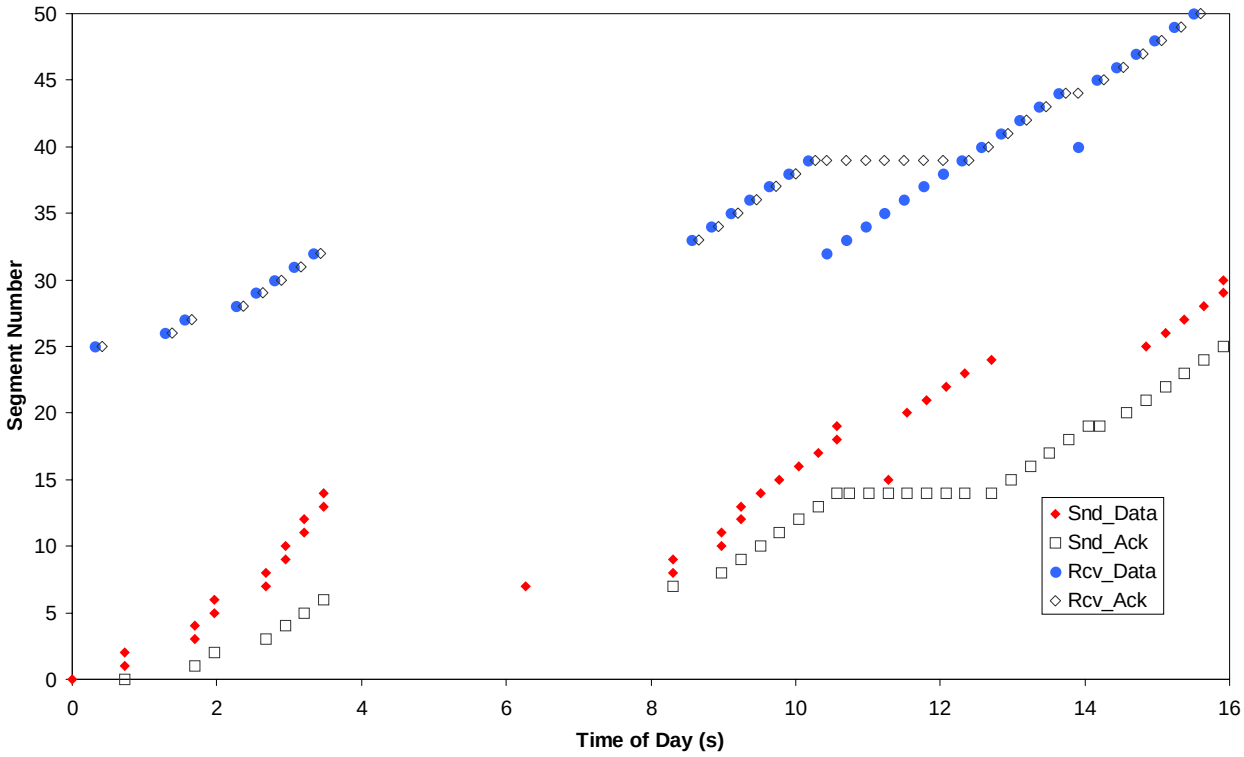

Figure 2. Spurious timeout of TCP Reno due to a 5 s delay spike.

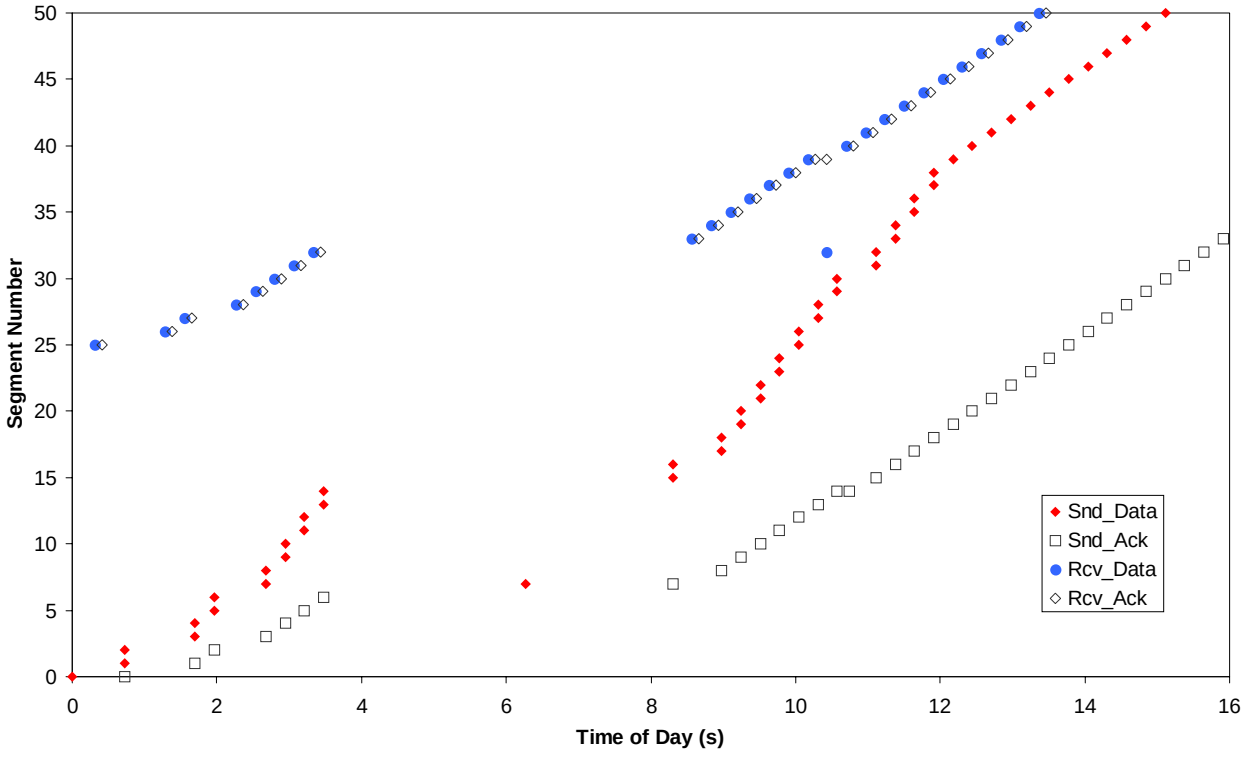

Figure 3. Spurious timeout of TCP Reno with Eifel due to a 5 s delay spike.

.

4. SETUP OF EXPERIMENTS

We evaluate the Eifel algorithm in three scenarios: easy, mediocre, and difficult. For each scenario, we assume different intervals between cell reselections where the intervals are drawn from a uniform distribution between a minimum and a maximum interval as shown in Table 1. The interval times have been chosen in a way that for a typical download time of 120 s on average one, two and three delay spikes occur per connection for the easy, mediocre and difficult scenario, respectively. We further assume that the time to complete a cell reselection is uniformly distributed between 3 and 15 seconds.

Table 1. Interval between Cell Reselections for three Scenarios.

	Min (s)	Max (s)
Easy	80	140
Mediocre	40	80
Difficult	20	40

For our experiments, we used the one-way models of Reno, NewReno and SACK TCP with delayed acknowledgments in NS2, and a slightly adapted version of the Eifel implementation from [9]. The test configuration is shown in Figure 4. It contains two nodes and a link with a drop-tail queue. The full-duplex link has a rate of 30 kbps and a one-way latency of 300 ms. The `hiccup` tool generates delays on the link. We have improved `hiccup` to suspend data flow in *both* directions and *after* the queue.

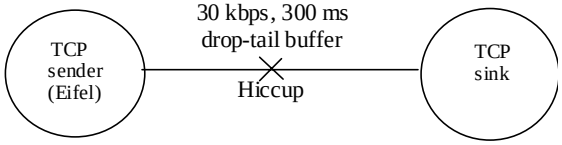

Figure 4. Test configuration in NS2.

A single test is based on a TCP connection transferring 300 KB of bulk data. Each test has been repeated a hundred times to ensure sound statistics. We experimented with a bottleneck buffer size of 10 KB that causes congestion losses, and a size of 100 KB that is large enough to fit the receiver window of data, and thus avoids any losses. We used the default settings for all parameters in the simulator, except for disabling the "bug fix" [8] and enabling the TCP Timestamps option [15].

5. RESULTS OF EXPERIMENTS

Figure 5 and Table 2 depict the average download times for the various bulk data transfers, while Figure 6 and Table 3 show the goodput results. As goodput we defined the ratio of the minimum number of segment required for completing the data transfer to the actual number of segments transmitted. Results are based on TCP Reno (R), NewReno (N), and SACK (S) with and without the Eifel algorithm. As expected, the download time increases and goodput decreases from the easy to the difficult scenario with growing frequency of delay spikes.

In an environment without congestion losses (100 KB buffer), Eifel reduces the download time and the number of unnecessarily retransmitted segments in all scenarios and for all three TCP flavors. Interestingly, NewReno without Eifel suffers from a large number of unnecessary retransmissions and increased download time compared to TCP Reno and SACK [5]. The goodput for all three TCP flavors with Eifel is close to 100 percent in all scenarios. This is a significant improvement compared to 87 percent of Reno and SACK or 80 percent of NewReno in the difficult scenario. The download time reduction ranges from 4 percent in the easy scenario to 12 percent in the difficult scenario.

In case of the small bottleneck buffer size (10 KB), Eifel improves the goodput for all three TCP flavors in all scenarios, and reduces the download time of NewReno and SACK. The goodput of NewReno and SACK with Eifel is close to 100 percent in all scenarios, compared to about 90 percent without Eifel in the difficult scenario. Eifel reduces the download time for NewReno and SACK by up to 8 percent. Although Reno with Eifel shows an improvement in goodput of several percent compared to pure Reno, the download time is notably increased in all scenarios. To explain this unexpected result we have closely examined packet traces of such connections. Apparently, Reno with Eifel suffers from lengthy non-spurious timeouts caused by packet losses.

Figure 7 shows an example of the poor performance of Reno with Eifel when packet losses occur. The first timeout at second 30 is caused by a delay and is spurious. The Eifel algorithm successfully detects the spurious timeout, and resumes transmission with the next unsent segment at second 36. In this case, some of the original segments were lost due to a buffer overflow. DUPACKs for the first lost segment start to arrive at second 37 *below* the highest outstanding segment. The "bug fix" [8] is disabled, and thus the fast retransmit is triggered at second 38. However, due to multiple losses the sender experiences a second timeout at second 61. That timeout is not spurious. However, the RTO at that time is huge, as it is calculated from the timestamps in the delayed segments. Additionally, the RTO may still be backed-off after the first timeout (It is not quite clear what the requirement level is in [10] for resetting the back-off counter once a new RTT sample is collected. We have instrumented TCP to reset the counter). When the retransmit timer finally expires at second 61 lost segments are recovered and normal transmission resumes. Without Eifel, Reno often avoids the second non-spurious timeout as it retransmits all outstanding segments in go-back-N, including the lost ones. Disabling the "bug fix" (as we have done in our tests) helps to recover without a non-spurious timeout when one, and sometimes two, segments are lost. However, in order to provide good TCP performance in environments with delay spikes and high loss rate, the Eifel algorithm should be coupled with efficient loss recovery schemes like SACK and Limited Transmit [14].

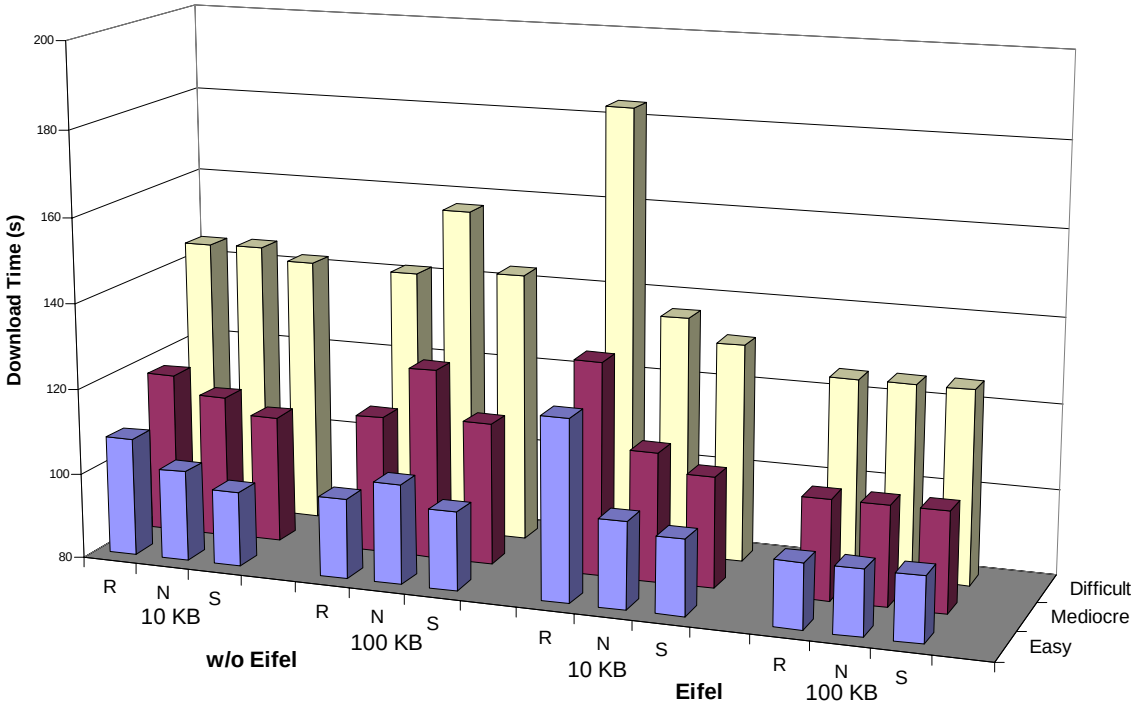

Figure 5. Download time of Reno (R), NewReno (N) and SACK (S).

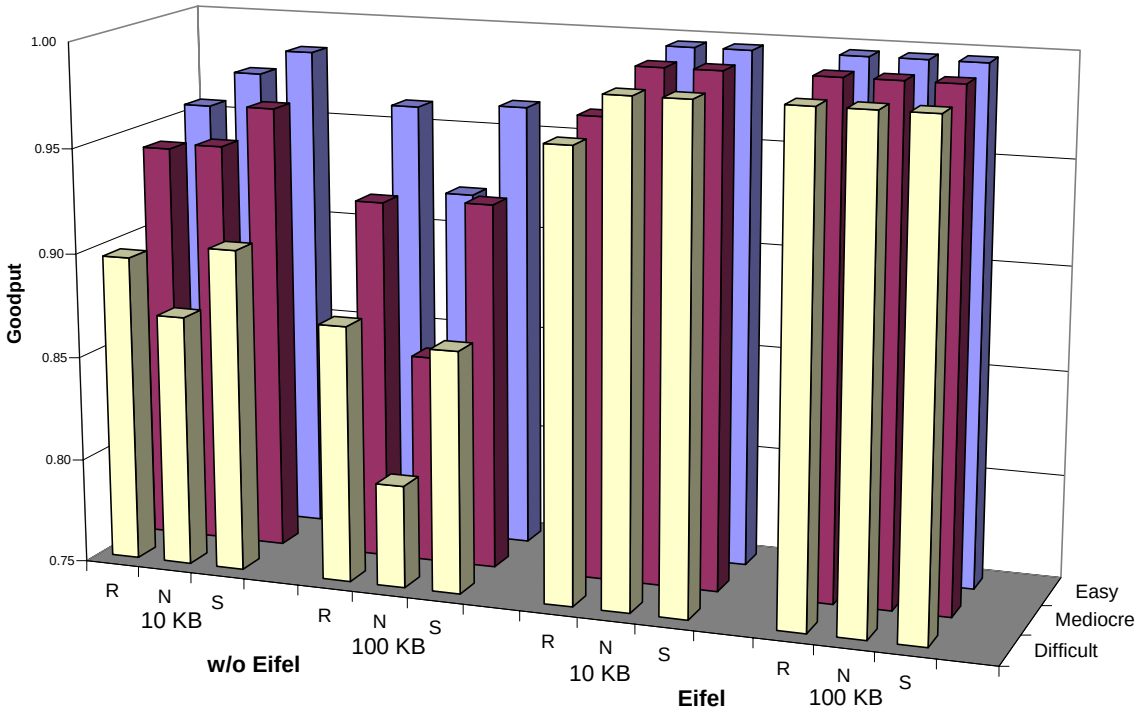

Figure 6. Goodput of Reno (R), NewReno (N) and SACK (S).

Table 2. Download time (s) of Reno (R), NewReno (N) and SACK (S).

Eifel	NO						YES					
Buffer	10KB			100KB			10KB			100KB		
TCP	R	N	S	R	N	S	R	N	S	R	N	S
Easy	108	101	98	99	103	98	122	100	98	95	95	95
Mediocre	118	113	110	112	125	113	130	110	106	103	103	103
Difficult	145	145	142	142	157	143	184	136	131	125	125	125

Table 3. Goodput of Reno (R), NewReno (N) and SACK (S).

Eifel	NO						YES					
Buffer	10KB			100KB			10KB			100KB		
TCP	R	N	S	R	N	S	R	N	S	R	N	S
Easy	0.96	0.97	0.98	0.96	0.92	0.96	0.97	1.00	1.00	1.00	1.00	1.00
Mediocre	0.94	0.94	0.96	0.92	0.85	0.93	0.97	0.99	0.99	0.99	0.99	0.99
Difficult	0.90	0.87	0.90	0.87	0.80	0.87	0.96	0.99	0.99	0.99	0.99	0.99

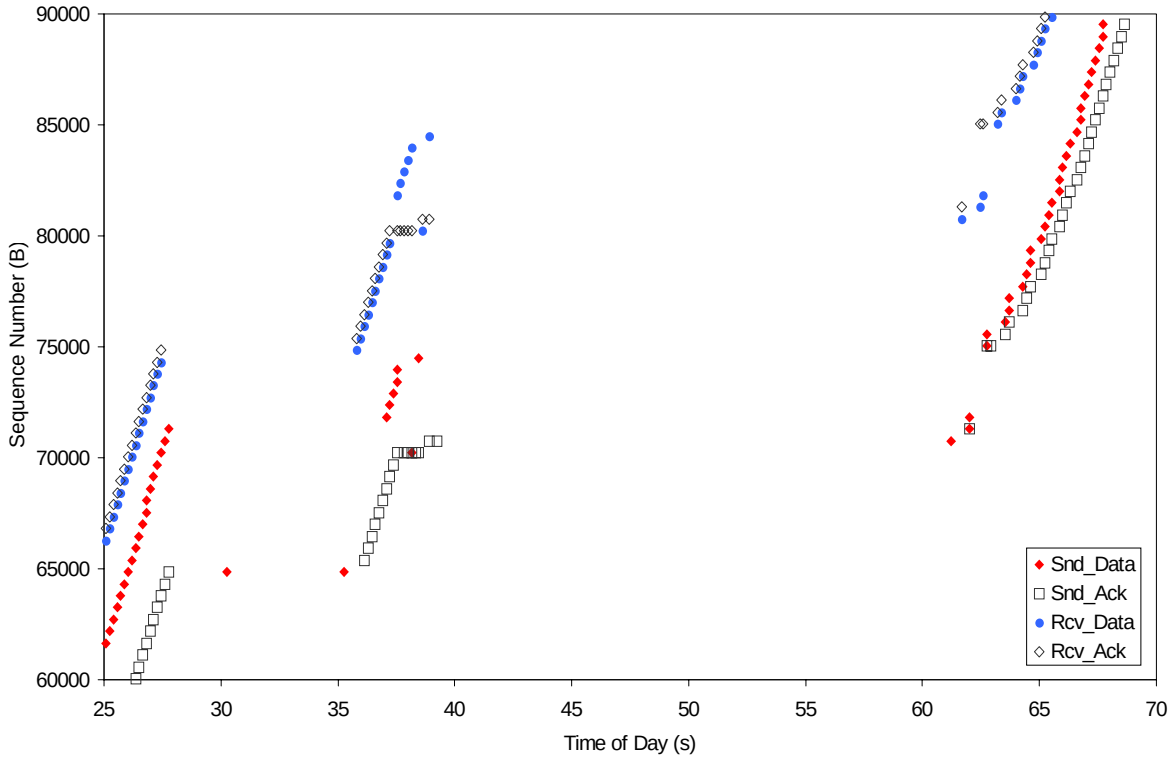

Figure 7. Performance problem of Reno with Eifel when packet losses are present.

6. CONCLUSIONS

We have studied the Eifel algorithm as defined in [4] for TCP Reno, NewReno and SACK in a simulated GPRS network. Large and sudden variations in packet transmission delays are often unavoidable in GPRS potentially causing spurious timeouts in TCP. For example, cell reselections may cause such delay variations. We simulated different scenarios with on average between one and three cell reselections taking place over the course of a two minutes bulk data transfer.

For mobile users and operators the battery power consumption and radio resource usage are often as important as the throughput across the wireless link. We therefore used throughput (download times) and goodput as equally important performance metrics. The bottleneck queue was assumed to be within the GPRS network. In bandwidth-dominated systems such as GPRS, the size of the bottleneck queue can greatly impact TCP's performance. We used two different sizes of the simulated drop-tail queue to capture this impact.

In case of a bottleneck queue that is sufficiently large to accommodate the maximum receiver window of a TCP connection, the Eifel algorithm improves the performance for all TCP flavors in all three scenarios. It reduces download times by up to 12 percent, and increases goodput by up to 20 percent. Bottleneck queues of such a size are often found in real GPRS networks.

In case of a smaller bottleneck queues, congestion losses may occur, and hence the TCP connection becomes network-limited. In that case, the Eifel algorithm still improves goodput by up to 10 percent for all TCP flavors in all three scenarios. For SACK and NewReno it also improves download times by up to 8 percent in all three scenarios.

Unexpectedly, TCP Reno yielded a considerable increase in download times when the Eifel algorithm was enabled and the bottleneck queue was small. We found that the reason for that were non-spurious timeouts with huge RTOs that typically follow a spurious timeout when packets from the outstanding flight were in fact lost due to congestion. From that we conclude that the Eifel algorithm is ideally complemented with an efficient SACK- or NewReno-based loss recovery scheme.

Our future work will focus on studying various modifications to the response part of the Eifel algorithm including how to reverse congestion control state, and how to adapt the round-trip time estimators.

REFERENCES

[1] G. Brasche and B. Walke. Concepts, services and protocols of the new GSM phase 2+ general packet radio service. IEEE Communications Magazine, pages 94--104, August 1997.

[2] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An extension to the selective acknowledgment (SACK) option for TCP. IETF RFC 2883, July 2000.

[3] ISI at University of South California. Network simulator 2. Available at: http://www.isi.edu/nsnam/ns/.

[4] R. Ludwig and R. H. Katz. The Eifel algorithm: Making TCP robust against spurious retransmissions. ACM Computer Communication Review, 30(1), January 2000.

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

[6] 3GPP TS 05.08 Radio subsystem link control, 2001.

[7] ETSI GSM 04.08, Mobile radio interface; Layer 3 specification.

[8] S. Floyd and T. Henderson. The NewReno modification to TCP's fast recovery algorithm. RFC 2582, April 1999.

[9] M. Schläger, NS TCP Eifel Page, http://www-tkn.ee.tu-berlin.de/~morten/eifel/ns-eifel.html

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

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

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

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

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

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