Eliminating Inefficient Cross-Layer Interactions in Wireless Networking

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften
der Rheinisch-Westfälischen Technischen Hochschule Aachen
zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften
genehmigte Dissertation

Vorgelegt von
Diplom-Informatiker
Reiner Eric Ludwig
aus Bishop, California (USA)

Berichter:
Prof. Dr. rer. nat. Otto Spaniol
Prof. Randy H. Katz Ph.D. (University of California at Berkeley)

Tag der mündlichen Prüfung: 04.04.2000

D 82 (Diss. RWTH Aachen)

Acknowledgements

Foremost, I want to thank my wife, Monika Ludwig, for her patience, faith, and support over the past five years.

I would like to thank Prof. O. Spaniol and Prof. R. H. Katz for their advice.

Many thanks to Almudena Konrad, Kimberly Oden, Bela Rathonyi, and Keith Sklower who contributed to the development of the measurement tools and the collection of many hours of traces.

This work greatly benefited from disscussions with Badri Badrinath, Hari Balakrishnan, Stephan Bauke, Jean Bolot, Mikael Degermark, David Eckhardt, Andreas Fasbender, Sally Floyd, Tom Henderson, Anthony Joseph, Roger Kalden, Phil Karn, Markku Kojo, Michael Meyer, Venkata Padmanabhan, and Vern Paxson.

Last but not least I want to thank Norbert Niebert, Frank Reichert, Olle Viktorsson, and Fiona Williams for supporting my work within Ericsson Research.

Table of Contents

CHAPTER 1

Introduction and Outline

The Internet has evolved into *the* communication medium of the future. It will not be long before virtually all people-to-people, people-to-machine, and machine-to-machine communication are carried end-to-end in *Internet Protocol (IP)* [RFC791], [RFC2460] packets. The recent tremendous growth of the Internet in terms of connected hosts is only matched by the similar growth rate of cellular telephone subscribers. While most hosts on today's Internet are still wired, the next *big* wave of hosts has yet to hit the Internet. We believe that the predominant Internet access of the future will be wireless. Not only every cellular phone, but the majority of general communication devices will have: (1) an IP protocol stack and (2) a wireless network interface.

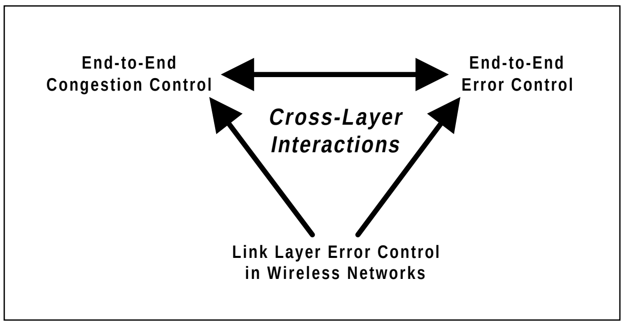

End-to-End Congestion Control
End-to-End Error Control
Cross-Layer Interactions
Link Layer Error Control in Wireless Networks

Figure 1-1: Cross-Layer Interactions in Wireless Networking.

Wireless networking and more specifically, the problems related to protocol performance of "IP over Wireless", are high priority research topics in both academia and industry. Yet, a number of difficult problems remain unsolved. The root of these lies in inefficient cross-layer interactions, caused by events occurring at the link layer that affect the performance of end-to-end error and congestion control schemes (see Figure 1-1). Those events are packet losses due to transmission error and increased packet transmission delays caused by link layer error control.

The goal of this dissertation is to study inefficient cross-layer interactions in wireless networking and to develop and evaluate a solution that eliminates these inefficiencies.

Internet traffic today is still largely generated by applications that use the *Transmission Control Protocol (TCP)* [RFC793] as the underlying communications protocol. The most popular Internet applications, namely World Wide Web (WWW), e-mail, and file transfer, all rely on TCP. In the Internet backbone studied in [TMW97], the authors find that TCP averages about 80 percent of the flows. Related traffic analysis studies find similar numbers. This motivates why TCP has been the focus when it comes to studying inefficient cross-layer interactions in wireless networking.

The solution we develop in this dissertation is, however, *not* "TCP-specific", but is independent from any specific protocol. Nevertheless, to demonstrate the feasibility of our approach, most of our measurements, analysis, and implementation work is based on TCP. Our solution comprises new end-to-end *and* link layer mechanisms that make the protocol implementations on those layers more "intelligent" and robust. For flows that are loss responsive and fully-reliable, in particular those based on TCP, our solution eliminates all known inefficient cross-layer interactions. It provides optimal end-to-end performance over a wide range of conditions of the wireless link while efficiently utilizing radio resources. Beyond solving the problems for such flows, our solution provides a framework that accommodates loss responsive real-time flows, e.g., semi-reliable, or error-resilient flows.

Related work suggests solutions that are either pure end-to-end (e.g., [SF98]), leaving unexploited the potential of link layer error control schemes to optimize end-to-end performance, or only suggest link layer mechanisms (e.g., [BBKT97], [ES98], [Kar93]), none of which is capable of solving all inefficient interactions with end-to-end control schemes. Most prior work (e.g., [BB95], [BK98], [BS97], [DMT96], [HK99], [KRLKA97]), however, suggests *Performance Enhancing Proxies (PEPs)* that couple link layer and end-to-end control schemes to solve inefficient cross-layer interactions. These solutions violate *the* fundamental design principle in data communications, *protocol layering*, by requiring access to transport layer headers by the PEPs. Our approach is fundamentally different in that we do not require - in fact, we argue against - such cross-layer couplings. The key advantages of our solution over PEP-based approaches are (A) its independence from transport (or higher) layer protocol semantics making it a "non-TCP-specific" solution, (B) the possibility of co-existence with network layer encryption, e.g., *IPsec* [RFC2401], and (C) that no per-flow state needs to be maintained in the network making it more scalable. The main contributions of this dissertation are the following:

1. The new concept of *flow-adaptive wireless links* and its application for fully-reliable flows. This work has been published in [LR99], [LRKOJ99], and [LKJK00].
2. Two new mechanisms for reliable end-to-end protocols, the *Eifel algorithm* and the *Eifel retransmission timer.* The former has been published in [LK00] while the latter is described in [LS99]. We have implemented both mechanisms for TCP, and refer to that implementation as *TCP-Eifel* that we have made publicly available [Lud99c].
3. New measurement methods and tools we developed throughout our work.

The remainder of this chapter describes and motivates the outline of this dissertation.

Chapter 2 provides required background. After we introduce related terminology in Section 2.1, the following three sections review those functions of the control schemes shown in Figure 1-1 that are relevant for our studies. Since all our “real-world” measurements in wireless networks were carried out using *GSM-CSD*, the *Circuit-Switched Data (CSD)* service implemented in the *Global System for Mobile communications (GSM),* we explain that network in more detail. In Section 2.5 we describe all inefficient cross-layer interactions in wireless networking that are known in literature. Related work is reviewed and evaluated in Section 2.6. We present a brief motivation and outline of the approach taken in this dissertation in Section 2.7.

Chapter 3 explains the analysis methodology we applied and the tools we developed to obtain the results presented in Chapter 4 and Chapter 5. We first motivate why we have mostly chosen a measurement-based analysis approach. In Section 3.1, we then explain the methodology we use in Section 4.3 to evaluate the benefit of link layer error recovery for reliable flows. We explain how we capture and analyze the error characteristics of the GSM-CSD wireless link. Our ReTracer tool is described, which we developed to reverse-engineer target metrics such as throughput given certain parameters like the link layer frame size. In Section 3.2, we explain the methodology we use in Section 4.2 to detect inefficient cross-layer interactions between TCP and the link layer error control implemented in GSM-CSD. We describe the tools we developed for that purpose: `rlpdump`, an event logging tool for the reliable link layer protocol implemented in GSM-CSD, and MultiTracer, used to correlated events on different protocol layers. There we also explain how to interpret TCP trace plots, which we often use to illustrate certain effects, problems, or solutions. In Section 3.3, we explain the methodology we use in Section 5.1 to study the problem of competing error recovery for the case of TCP, and to develop the *Eifel algorithm* in Section 5.2 that eliminates this problem. We explain how we used the `hiccup` tool that we developed to reproduce inefficient cross-layer interactions in a “non-wireless” but controllable network environment. In Section 3.4, we explain the methodology we use in Section 5.3 to study and reveal the problems of TCP-Lite’s retransmission timer. We use the same model in Section 5.4 to develop the *Eifel retransmission timer* that eliminates those problems. We explain the model that we developed to analyze those end-to-end retransmission timers, and the measurement setup we used to validate the correctness of the model. In Section 3.5, we provide a summary of the chapter.

Chapter 4 introduces the concept of flow-adaptive wireless links, and validates it for fully-reliable flows. In Section 4.1, we explain that concept and discuss its deployment concerns, and possible implementation alternatives. The key idea is that network end-points use the IP layer as a level of indirection through which their QoS requirements are signalled to each link layer along the path, on a per packet basis. This allows for a (wireless) link layer to adapt its error control schemes to meet those requirements while minimizing radio resource consumption. We argue against and illustrate the problems of running low link layer error recovery persistency for fully-reliable flows. In Section 4.2, we verify through measurements, that our solution of flow-adaptive wireless links eliminates all known inefficient cross-layer interactions except for the problem of competing error recovery. This study also delivered early indications that the retransmission timer implemented in TCP-Lite is too conservative. In Section 4.3, we show how the GSM-CSD wireless link can be adapted to optimize the end-to-end performance of bulk data flows. We use this case study to demonstrate that link layer error recovery over wireless links is essential for reliable flows to optimize their end-to-end performance. In Section 4.4, we provide a summary of the chapter.

Chapter 5 provides solutions to the remaining problems we identified in Chapter 4. In Section 5.1, we identify the retransmission ambiguity in TCP as the root of the problems caused by competing error recovery. In Section 5.2, we develop the *Eifel algorithm* that uses extra information in the TCP acknowledgements to resolve the retransmission ambiguity, and show how this is used to eliminate those problems. The Eifel algorithm only requires changes to the TCP sender implementation, not to the protocol itself. Thus, given this backwards compatibility and the fact that it does not change TCP's congestion control semantics, the new algorithm can be incrementally deployed. In Section 5.3, we analyze TCP-Lite's retransmission timer, and reveal a number of problems related to its definition and implementation. This explains why we had suspected that this timer is too conservative. In Section 5.4, we then propose an alternative retransmission timer, which we call the *Eifel retransmission timer*, that eliminates those problems. We demonstrate that the Eifel retransmission timer is a more precise predictor of an upper bound for the path's RTT. Combining both, the Eifel algorithm and the Eifel retransmission timer, we propose a new approach to designing retransmission timers. The idea is to let the timer become increasingly aggressive while adapting it to the measured fraction of spurious timeouts. We validated the correctness of our analysis by showing that the model- and the measurement-based approach leads to the same results. In Section 5.5, we provide a summary of the chapter.

Chapter 6 concludes this dissertation by summarizing our main results and outlining related but unsolved research problems that merit further exploration.

CHAPTER 2

Background

In this chapter, we explain the relevant technologies and related work for the background required in subsequent chapters. The first three sections describe the “three corners” of the triangle shown in Figure 1-1. In Section 2.5, we define the central problem (the center of Figure 1-1) this dissertation addresses. In Section 2.6, we briefly explain how related work has approached the problem, and provide an evaluation of the proposed solutions. In Section 2.6, we briefly motivate and outline the approach developed in the remainder of this dissertation.

2.1 Terminology

The *Internet* is an inter-connection of networks. A *network* connects *hosts* and networks are interconnected via *routers*. Communication in the Internet is based on the *Internet Protocol (IP)*, a network layer protocol defined by [RFC791], referred to as *IP-Version 4 (IPv4)*, and alternatively by [RFC2460], referred to as *IP-Version 6 (IPv6)*. A protocol data unit in IP is called a *packet*, consisting of an *IP header* followed by transport layer data[1]. The transport layer data may (in theory) be up to 64 KBytes large. The default size of the IP header is 20 bytes, and with option fields may be up to 60 bytes. 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 higher layer protocols such as the *Transmission Control Protocol (TCP)*. In particular, packets belonging to the same connection may take different routes to the destination and in practice sometimes do [Pax97d].

1. Unlike in [RFC1122] we do not distinguish between an IP datagram and an IP packet and always use the latter term.

In most cases, a host is a general purpose computer but it may also be a specialized appliance. Examples of a network are an Ethernet (multi-access), a dial-up line provided by a fixed or wireless telephone system (point-to-point), or a direct cable that connects two hosts via their serial line interfaces (point-to-point). The network that connects two hosts, two routers, or a host and a router is also called a *link* or *hop*. Communication across a link is provided by a link layer protocol. A protocol data unit in a link layer protocol is called a *frame*. A host connects to a link via a *network interface* (or just *interface*). Each interface on a host has an interface address that is unique in the corresponding network *and* an *IP address* that is unique in the Internet. The source and destination IP addresses are part of the IP header. The receiving IP layer uses the *protocol identifier* that is also part of the IP header to decide to which transport protocol, e.g., TCP, it is supposed to deliver the packet's payload. Transport layer service is provided through a *port* identified by a *port number* which is unique only in combination with an IP address and a protocol identifier. The source and destination port numbers are part of a transport protocol header.

A series of links connecting two hosts is called a *path*. Communication between two processes at each end of a path is referred to as *end-to-end* communication. Such a process is generally called a *network end-point* (or just *end-point*). End-to-end communication is provided by network (IP), transport and (optionally) application layer protocols, so-called end-to-end protocols. Thus, a network end-point can be a sending/receiving network (IP), transport or application layer, or a sending/receiving application.

An end-to-end stream of packets identified by the source and destination IP address, the source and destination port number, and the protocol identifier, is referred to as a *flow*. For example, the packets sent by a TCP sender to a TCP receiver corresponds to a flow; a TCP-based flow. A network end-point is tied to one specific flow and each flow has two network end-points, a sending and a receiving network end-point. Hence, a property of a flow may also be associated with a network end-point and vice versa. When speaking of such properties, we use the terms *network end-point* and *flow* interchangeable.

A *flow's bottleneck link* is the link on which the bandwidth available to the flow is the lowest of all links in the path. That bandwidth is also referred to as the *flow's available bandwidth* as it limits the end-to-end throughput that the flow may provide. The flow's available bandwidth can be very dynamic depending how many flows share the bottleneck link and how much of the bottleneck link's bandwidth those flows consume. We define as the *packet transmission delay* the time it takes to *successfully* transmit a packet over a given link excluding any queueing delay that may occur before the initial transmission of the packet. A flow's *round trip time (RTT)* (sometimes also called the path's RTT) is the time it takes to send a packet from one network end-point to the other, get it processed at the receiving end-point, and send another packet back to the end-point that sent the initial packet. A flow's RTT varies dynamically, depending on such factors as packet size (transmission delays), queueing delays the packets experience in the network, and processing required at the receiving end-point[1]. The packets a network end-point sends within the flow's RTT is called a *flight of packets* (or just *flight*). Those packets are also referred to as the packets a network end-point or flow has *in flight*. The number of packets a network end-point has in flight is called the *flow's load*[2].

Link layer and end-to-end protocols[3] have the following functions in common, not all of which have to be implemented (see [Tan89] or [Ste94] for more detail). We use the term *user process* to refer to the process that uses the protocol being described.

- *Framing* ensures that data units passed by a sending user process to a protocol are delivered as the same data units to the receiving user process. For example, if implemented at the link layer, it allows the receiving link layer to recognize the beginning and the end of an IP packet. This enables the receiving link layer to deliver each IP packet as a single unit to the receiving IP layer, i.e., the receiving user process.
- *Flow control* to prevent a fast sender to overflow a slow receiver.
- *Error detection* ensures that a protocol data unit received in error is discarded and is *not* delivered to the receiving user process. This function is provided by adding a checksum to each protocol data unit that the receiving protocol layer verifies.
- *Error recovery* - also known as *Automatic Repeat reQuest (ARQ)* - requires error detection and ensures that lost protocol data units and protocol data units received in error are retransmitted. We define as *error recovery persistency* the maximum number of retransmissions an error recovery scheme performs for a single protocol data unit before it is discarded. Alternatively, error recovery persistency may be defined as the maximum permissible delay that the error recovery scheme may introduce for a single protocol data unit before is discarded.
- *In-order delivery* ensures that the sequence of the data units passed by a sending user process to a protocol is preserved when those data units are delivered to the receiving user process.
- Removal of data that might have been duplicated during transmission by the protocol.

A protocol provides *reliable* service if it implements the latter five functions. It provides *unreliable* service if it implements error detection but not necessarily error recovery. It provides *transparent* service if it does not implement error detection. Either service may or may not sup-

1. For example, also including the delayed-ACK timer of 500 ms that may be used in TCP [RFC1122].
2. Some of those packets might have left the network already but because of the feedback delay (the RTT), the sending network end-point might not yet be aware of it. Also note, that the feedback might be provided explicitly, e.g., through ACKs as in TCP (see Section 2.2.1), or implicitly through (RTT and packet loss rate) measurement reports provided by the receiving network end-point as done for "TCP-friendly" flows (see Section 2.3.3).
3. We avoid this distinction in this context when it is clear that we speak of a link layer or end-to-end protocol.

port framing. A service that does *not* support framing is referred to as a *byte stream service*. Further, reliable protocols need to establish a *connection* at the beginning of each instance of communication. A connection is required to establish and maintain common protocol state (e.g., initial sequence numbers, flow control windows) between the sending and the receiving protocol layer.

The fact that IP does not need to preserve the packet order also allows for link layer protocols that provide reliable service to perform *out-of-order delivery* of correctly received IP packets, i.e., to not implement the above mentioned in-order delivery function. This provides for more memory-efficient link layer implementations. We still regard such a link layer protocol as providing reliable service, but make it explicit in the text when referring to that case.

We further distinguish between *fully-reliable* and *semi-reliable* service. When the error recovery persistency is reached, a protocol providing fully-reliable service terminates the connection (discards the common connection state) and indicates that event to the sending and the receiving user process. Those processes then "know" that data was lost and may or may not decide to re-initiate their communication. A protocol that provides semi-reliable service, on the other hand, does *not* terminate the connection when its error recovery persistency is reached, and does *not* indicate that event to the sending and the receiving user process. Instead, it just discards the corresponding protocol data unit and resumes transmission with the next one in sequence. We also use the terms (fully- or semi-) reliable, unreliable, or transparent in association with the term *protocol* depending on which service it provides, and with the term *flow* depending on which service the flow provides to the application. For example, the flow generated by a reliable multicast protocol, like [FJLMZ97], is considered reliable even if UDP is the underlying unreliable transport protocol. A flow generated by TCP is an example of a fully-reliable flow.

A link layer protocol that provides service directly to IP needs to implement framing and is called a *framing protocol*. Throughout this text we only refer to one framing protocol which is the *Point-to-Point Protocol (PPP)* [RFC1661] commonly used on dial-up and direct cable links. By default, PPP provides an unreliable service[1]. A framing protocol defines the link's *Maximum Transmission Unit (MTU)*, i.e., the size in bytes of the largest IP packet that can be transmitted on that link. The smallest MTU of all links of a path is called the *path MTU*. The IP layer includes a fragmentation function, referred to as *IP fragmentation* [RFC1122], that is used in case a packet is larger than the outbound link's MTU. IP fragmentation may be performed by the sending host's IP layer or any router's IP layer. If an IP packet needs to be fragmented, it is divided into smaller fragments and a copy of the IP header is prepended to each fragment. A fragment number is inserted into each of those headers so that the destination IP

1. [RFC1663] defines an extension for PPP to optionally provide reliable service. It is, however, rarely used.

layer can perform the reassembly of the original IP packet. IP fragments are transmitted and routed as regular IP packets.

We further classify flows according to the type of traffic they carry. *Bulk data flows* are generated by applications that need to transfer "large" amounts of data (e.g., file transfer or e-mail). The main Quality of Service (QoS) requirement of such flows, more precisely the QoS requirement of the corresponding application, is to maximize throughput, i.e., to transfer the *entire* data as fast as possible, while the end-to-end delay of an individual packet is less important[1]. We also speak of a *bulk data transfer* in this respect. *Interactive flows* are used for transactional (request/response-style) communication (e.g., remote terminal or banking applications). The main QoS requirement of such flows is to minimize the end-to-end delay of the packets belonging to a transaction, i.e., a low user level response time, while the end-to-end throughput that the flow may provide is less important. Bulk data and interactive flows are usually based on an end-to-end protocol that provides a fully-reliable service (e.g., TCP). *Real-time flows,* on the other hand, are usually based on an unreliable end-to-end protocol, e.g., the *User Datagram Protocol (UDP)* [RFC768]. They are generated by applications that are delay-sensitive (e.g., audio and video applications). An important class of real-time flows in the Internet are *rate-adaptive* real-time flows, e.g., those based on the *Real-Time Transport Protocol (RTP)* [RFC1889]. Applications that operate on such flows can adapt (to certain degrees) the output rate of their source codecs to the flow's available bandwidth. A comprehensive discussion of flow types and their QoS requirements can be found in [She95].

A *packet loss* is the event that a packet is sent into the Internet but does not reach the destination. A packet can get lost because it is dropped due to congestion (see Section 2.3) at an interface's in- or outbound buffer, or it is discarded due to transmission error by a link layer error detection function (see Section 2.4). We call the former event a *congestion loss* and the latter event an *error loss*.

2.2 End-to-End Error Recovery with TCP

2.2.1 Basic Operation

The basic functionality of TCP is defined by [RFC793], [RFC1122], and [RFC2581]. TCP extensions have been defined by [RFC1323], [RFC2018], and [RFC2481][2]. Those six recom-

1. In theory, it would not matter in a file transfer if the first packet reached the destination last. What usually matters is that the file transfer is completed in the shortest amount of time. In practice, the transport layer receive socket buffer required for packet re-sequencing places a limit on the maximum per packet delay that is tolerable without affecting performance. This limit is nevertheless low.
2. Currently in experimental status.

mendations have been proposed over a time frame of almost twenty years. During this time numerous TCP implementations for various operating systems have been developed. Some of these predate the more recent recommendations, and not every desirable TCP feature has been specified. Moreover, some TCP implementations are incorrect due to logic errors, misinterpretations of the specification, or conscious violations to gain better performance [Pax97b]. Consequently, many different "TCPs" exist today. Throughout this dissertation we refer to the so-called *TCP-Lite* implementation for the *Berkeley Software Distribution (BSD)* operating system documented in [WS95]. In the Internet research community, it is the current de facto standard for TCP implementations. It has been ported to various operating systems running daily on hundreds of thousands of servers and clients on the Internet. We omit the qualifier "-Lite" when discussing TCP in general as specified by the above listed recommendations.

TCP is a transport layer protocol that provides a fully-reliable byte stream service. It exchanges data with the user process through shared memory, so-called *(send and receive) socket buffers*. The size of TCP's socket buffers are usually determined by default settings of the operating system; commonly 8 or 16 KBytes. A protocol data unit in TCP is called a *segment*, consisting of a *TCP header* followed by application layer data. The default size of the TCP header is 20 bytes, and with option fields may be up to 60 bytes. Each segment is transmitted as a separate packet, and the receiving IP layer delivers it as a single unit to the receiving TCP layer. Thus, a segment does not require (begin/end) delimiters. Each byte in the application layer byte stream corresponds to a unique *sequence number* in a TCP connection. The header of each segment carries the sequence number of the first byte in the segment. The size of the application layer payload is variable but may not be larger than the *Maximum Segment Size (MSS)*[1]. The default MSS is 536 bytes derived from the default MTU size (576 bytes) which leaves space for default size TCP and IP headers. The MSS to be used by the TCP sender is usually announced by the TCP receiver during connection establishment through the *MSS option* in the TCP header. Nevertheless, it is limited by the outbound link's MTU (minus the size of the TCP/IP header). Alternatively, the sender may use the path MTU discovery procedure [RFC1191] to derive an appropriate MSS. The specification [RFC793] arbitrarily assumes a value of 2 minutes for the *Maximum Segment Lifetime (MSL)*. The MSL controls the maximum rate at which segments may be sent before the sequence numbers wrap around.

A TCP receiver sends *positive acknowledgements (ACKs)* for segments that are received correctly and in-order and *duplicate acknowledgements (DUPACKs)* for segments that are received correctly but out-of-order. No feedback is provided for segments received in error. ACKs may be generated for every segment, or for every other segment if the *delayed-ACK* mechanism [RFC1122] is used. DUPACKs may not be delayed. Both types of acknowledgements contain the so-called *ACK number* that is next sequence number that the TCP receiver

1. Note the slight illogicality: Although both the TCP header and the application layer payload together constitute a segment, the segment size, in particular the MSS, only applies to the payload.

expects to receive. A DUPACK contains the same ACK number as the last sent ACK. Thus, a DUPACK does not convey *which* segment was received correctly (unless Selective Acknowledgement Options [RFC2018] are used). The segments or bytes the TCP sender has sent and which are waiting to be acknowledged are called *outstanding*.

Two different error recovery strategies have been specified for TCP: (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 with the same ACK number have been received independent of the retransmission timer [Jac90a]. Section 2.2.2 provides a detailed description of TCP's retransmission timer. TCP's error recovery is fairly persistent. It retransmits a single segment twelve times which corresponds to roughly 9 minutes before the connection is aborted.

Flow control is provided through the well-known sliding window mechanism. ACKs sent by the TCP receiver carry the *advertised window*, which limits the number of bytes the TCP sender may have outstanding at any time. The advertised window (usually) corresponds to the size of TCP receiver's receive socket buffer. End-to-end protocols that implement sliding window flow control, like TCP, share an important *self-clocking property* [Jac88]. We explain this with Figure 2-1 (a modified version of a figure taken from [Jac88]) showing a schematic representation of a sender and a receiver on high bandwidth networks connected by a slow link, the bottleneck link, that is error-free. The vertical dimension is bandwidth, the horizontal dimension is time. Each of the shaded boxes is a packet. Because "*bandwidth x time = bits*", the area of each box is the packet size. Thus, a packet on the slow link (occupying less in the vertical

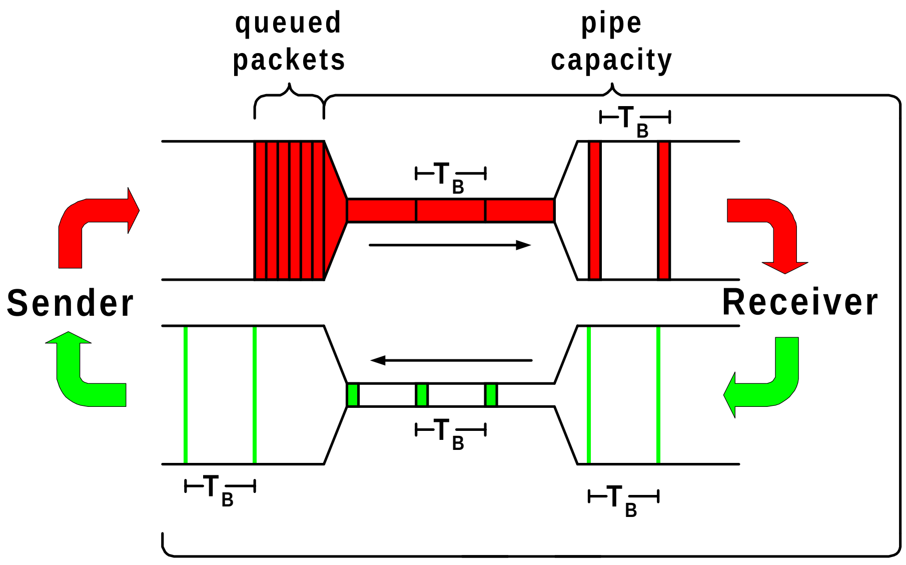


Figure 2-1: The "ACK Clock".

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

dimension) has to spread more in time (occupying more in the horizontal dimension). Figure 2-1 shows the ideal case in which a single sender fully utilizes the non-shared bottleneck link with a fixed bandwidth and always sends fixed size segments. In that case the ACK inter-arrival time at the sender is constant and equal to the packet transmission delay over the bottleneck link, T_B. This constant stream of returning ACKs is referred to as the *ACK clock*. The arrival of an ACK “moves the sliding window to the right” by one segment and “clocks out” a new segment. Consequently, for every packet that leaves the bottleneck link, a new packet arrives.

2.2.2 TCP-Lite’s Retransmission Timer

While data is outstanding the TCP sender samples the path’s RTT by timing the difference between sending a particular segment and receiving the corresponding ACK. Older TCP implementations only time one segment per RTT, whereas newer implementations use the timestamp option [RFC1323] to time every segment. Timing every segment allows much closer tracking of changes in the RTT. When using the timestamp option, the TCP sender writes the current value of a “timestamp clock” into a 12 bytes option field in the header of each outgoing segment. The receiver then echos those timestamps in the corresponding ACKs according to the rules defined in [RFC1323]. When receiving an ACK the TCP sender determines the RTT by calculating the difference between the current value of its “timestamp clock” and the timestamp echoed in the ACK. In the context of TCP, 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.

We refer to the *RTT sampling rate* as the number of RTT samples the TCP sender captures per RTT divided by the TCP sender’s load. In case the TCP sender times every segment and the TCP receiver acknowledges every segment, the RTT sampling rate is 1. If the TCP sender times every segment and the TCP receiver acknowledges every other segment (delayed-ACK), the RTT sampling rate is 1/2. If the TCP sender only times one segment per RTT, the RTT sampling rate is the reciprocal of the TCP sender’s load. The closer the RTT sampling rate is to 1 the more accurately the TCP sender measures the RTT.

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 RTT which - especially on an end-to-end path through the Internet - 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 transport protocol like TCP. It can greatly influence end-to-end 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, timeouts also trigger congestion control (explained in Section 2.3.2), 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 flows. But it also affects bulk data transfers whenever the TCP sender has exhausted the window limiting the number of outstanding segments before the retransmission timer expires.

In the following, we refer to TCP-Lite's retransmission timer as the *Lite-Xmit-Timer.* We use the index *L (Lite)* as a qualifier for a metric when referring to its definition or implementation. We omit that qualifier when discussing a particular metric in general. The following set of equations define RTO_L [JK92]. In its implementation, RTO_L is updated every time the TCP sender completes a new RTT measurement, denoted as RTT_{Sample}.

$$\begin{aligned}
DELTA_L &= RTT_{Sample} - SRTT_L \\
SRTT_L &= SRTT_L + \frac{1}{8} \times DELTA_L \\
RTTVAR_L &= RTTVAR_L + \frac{1}{4} \times (|DELTA_L| - RTTVAR_L) \\
RTO_L &= \mathrm{MAX}(SRTT_L + 4 \times RTTVAR_L, 2 \times ticks)
\end{aligned}$$

SRTT is the so-called *smoothed RTT estimator.* $SRTT_L$ is a low-pass filter that keeps a memory of a connection's RTT history with a fixed weighing factor of 7/8. $DELTA_L$ 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 [JK92] 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.* $REXMT_L$ is based on a so-called *heartbeat timer* provided by the BSD operating system implementing a timer granularity of 500 ms. The heartbeat timer expires *every* 500 ms, triggering a specific interrupt routine that updates the $REXMT_L$ (decrements it by one tick) of each active TCP connection. This is done independent of whether one of those $REXMT_L$ would actually go to zero or not. If

$REXMT_L$ was initialized with a value of one it could expire anywhere between 0 - 1 tick, because the initialization event is out of phase with the heartbeat timer. Therefore, a minimum of 2 ticks is required for RTO_L to prevent spurious timeouts in case $SRTT_L + 4 \times RTTVAR_L$ evaluates to one.

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 TCP sender's send socket 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. The following equation defines $REXMT_L$. 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.

$$REXMT_L = RTO_L$$

We briefly summarize related work concerning the Lite-Xmit-Timer. *Karn's algorithm* [KP87] must be implemented in TCP [RFC1122]. It prevents a clamped retransmission timer by ignoring the RTT_{Sample} derived from a retransmitted segment *and* doubling the RTO (*exponential timer backoff*) up to a maximum of two times MSL, 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 REXMT is clamped at too low a value. The authors of [BP95b] remove an inaccuracy in the implementation of RTO_L that made it more conservative then 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 [AP99] against a large set of real measurements. The authors conclude that the RTO minimum (*2 x ticks*, i.e., 1 second) 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, the study also concludes that the estimator gains and the RTT sampling rate have little influence on the Lite-Xmit-Timer's performance. We disagree with the latter conclusion and show in Section 5.3.2 that in fact the opposite is the case.

2.2.3 TCP/IP Header Compression

Although, TCP/IP header compression [RFC1144], [RFC2507] is a link layer mechanism, it has a lot to do with TCP's end-to-end error recovery. It is commonly implemented as part of a link's framing protocol, and is used to reduce the per packet overhead transmitted over the link. In the common case, a default size TCP/IP header of 40 bytes is compressed to 4 bytes. This

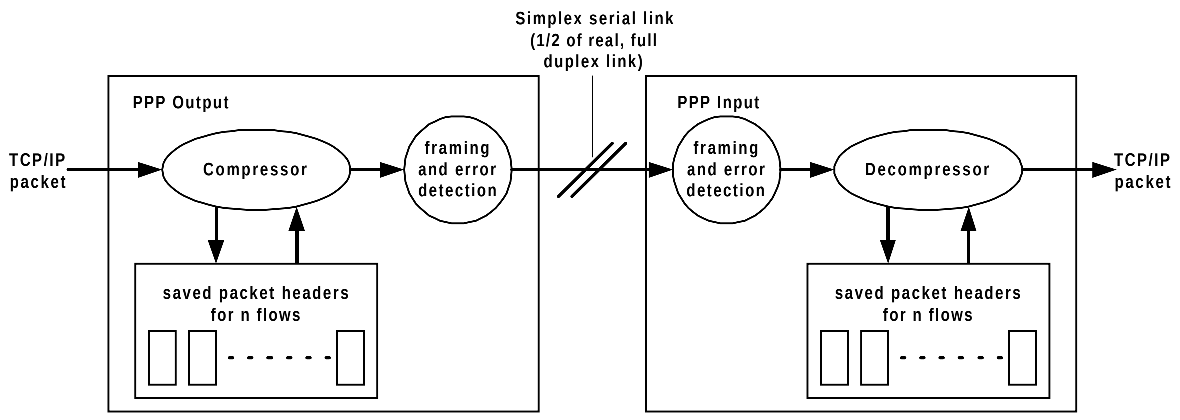

Simplex serial link
(1/2 of real, full
duplex link)
PPP Output
PPP Input
TCP/IP packet
Compressor
framing and error detection
framing and error detection
Decompressor
TCP/IP packet
saved packet headers for n flows
saved packet headers for n flows

Figure 2-2: Location of the TCP/IP header de-/compressor in the packet stream.

compression ratio is possible because - apart from the sequence number, the ACK number, and the checksums (one for the IP header and one for the TCP segment) - not much changes in the headers from packet to packet of the same flow. The advantages of header compression are cost savings on links with volume based charging and increased link throughput. For example, if the link's MTU is 296 bytes - a size commonly used on dial-up links - TCP/IP header compression increases the link's achievable throughput by more than ten percent. Figure 2-2 (taken from [RFC1144]) sketches the location of the compressor and the decompressor in the packet stream.

TCP/IP header compression is a *differential encoding* (also called *delta encoding*). After the first TCP/IP header of a flow has been transmitted uncompressed, only the encoded difference to the preceding header, the *delta*, is transmitted as the header of a packet. The decompressor derives the uncompressed header by applying the delta to the stored header of the packet that was last received for that flow. As with other differential encoding schemes, TCP/IP header compression relies on the fact that the deltas (more precisely the packets carrying the deltas) are not lost or reordered on the link between compressor and decompressor. A lost delta (packet) will lead to a series of false headers being generated by the decompressor, yielding

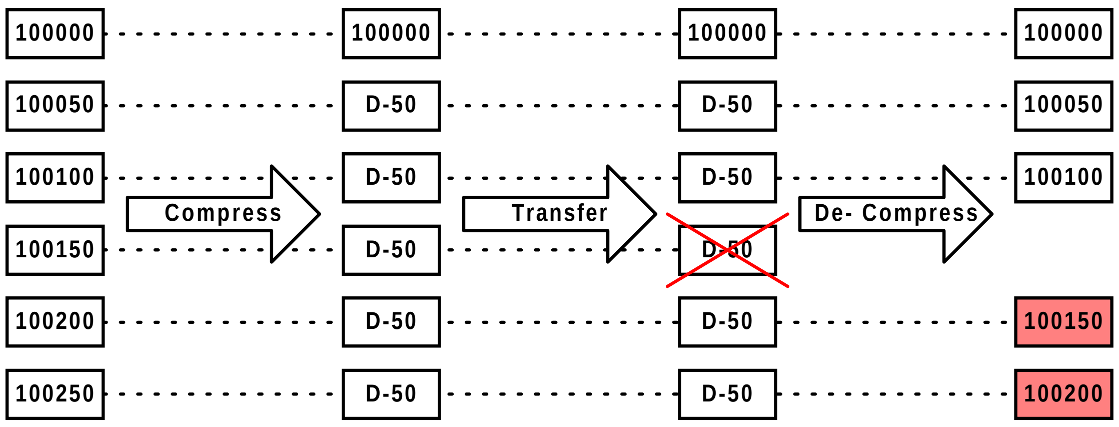


Figure 2-3: Compressor and decompressor losing synchronization.

TCP segments that have to be discarded at the TCP receiver because of checksum errors. This is sketched in Figure 2-3 showing an initial sequence number of 100,000 which increases by 50, the delta, from packet to packet. To resynchronize the compressor and the decompressor, [RFC1144] and [RFC2507] require that the TCP/IP headers of packets containing a retransmitted segment may not be compressed. Thus, once a delta is lost, an entire flight of segments is lost and has to be retransmitted. Even worse, since the TCP receiver does not provide feedback for erroneous TCP segments, the TCP sender is forced into a timeout. We have measured this effect and further discuss this issue in Section 4.2.3. [RFC1144] and [RFC2507] differ in their robustness against lost deltas. Whereas [RFC1144] cannot tolerate a single lost delta (the case shown Figure 2-3), [RFC2507] can tolerate the loss of a single lost delta - using the *twice algorithm* - but also loses synchronization once two deltas are lost back-to-back.

It is worth pointing out that header compression is an example of *layer violation*: a particular layer (in this case the link layer) inspects and interprets a higher layer's headers. Typical cases of layer violation are nodes in the network that require access to the headers of an end-to-end protocol.

2.3 End-to-End Congestion Control in the Internet

2.3.1 Objectives and Principles

A *best-effort* network like the Internet does not have the notion of admission control or resource reservation to control the imposed *network load*, i.e., the total number of packets that reside within the network. A best-effort network under high network load is called *congested.* If the network is in this state, host and/or router network interface buffers may overflow causing packets to be lost (dropped), i.e., congestion losses. Network end-points sharing a best-effort network need to respond to congestion by implementing *congestion control* to ensure network stability. Otherwise, the network may be driven into *congestion collapse*: the network load stays extremely high but throughput is reduced to close to zero [RFC896]. Thus, the objective of end-to-end congestion control is for network end-points to estimate (by probing the network) their available bandwidth while ensuring network stability.

In the following, we give a general description of congestion control in the Internet, and introduce related terminology. A more detailed description of these terms and concepts is provided in Section 2.3.2, where we explain how congestion control is implemented in TCP.

A congestion control scheme has three basic elements: (1) the network must have a *congestion signal* to inform the network end-points that congestion is occurring or about to occur, (2) the network end-points must have a policy to decrease their load on the network in response to the congestion signal, and (3) the network end-points must have a load increase policy in times when the congestion signal is not received as this may indicate that more bandwidth has become available at the bottleneck link. The latter is also referred to as *probing (for bandwidth)*. The key issue is the congestion signal. One distinguishes between *explicit* congestion signals issued by the network and *implicit* congestion signals inferred from certain network behavior by the network end-points. Routers in today's Internet do not issue explicit congestion signals[1], although this might be implemented in the future [RFC2481] (see Section 2.3.3).

Two approaches have been discussed for network end-points relying on an implicit congestion signal: delay-based [Jai89], [BP95a] and loss-based [Jac88], [Jac90a]. However, it is often not possible to draw sound conclusions from network delay measurements (e.g., see [BV99]). In particular, it is difficult to find characteristic measures such as the path's minimum RTT as required by [BP95a]. This may be due to route changes [Pax97d] or persistent congestion at the bottleneck link. Consequently, "packet loss" is the only signal that network end-points can confidently use as an indication of congestion. It is implemented either as a direct trigger (see Section 2.3.2) based on the detection of a lost packet, or an indirect trigger, based on a perceived packet loss rate (see Section 2.3.3) to reduce a flow's load. Such network end-points and their corresponding flows are *loss responsive*. In this dissertation, we only deal with loss responsive flows. We often omit the qualification "loss responsive" when talking of flows.

In the absence of an explicit congestion signal, an additive increase and a multiplicative decrease policy is required in an heterogeneous environment like the Internet to converge to network stability [Jac88], [CJ89].

We define as the *flow's pipe capacity* the minimum number of packets a flow needs to have in flight, i.e., the minimum load, to fully utilize its available bandwidth. Packets of a flow's load exceeding the flow's pipe capacity are queued in the network (see Figure 2-1). They contribute to network congestion and an increased end-to-end delay, which also affects the flow's own RTT. Ideally, a network end-point would not increase its load beyond its flow's pipe capacity. However, this is impossible with a congestion control scheme that only relies on an implicit congestion signal. With such a scheme the network end-points treat the network as a "black box", but the flow's pipe capacity can only be known by "looking into the black box".

A network end-point or flow is *network-limited* if its load is limited by congestion control. This property is commonly associated with bulk data and rate-adaptive real-time flows, rarely with interactive flows. Whether a network-limited flow fully utilizes its available bandwidth depends on the number of packets the flow may have in flight beyond its pipe capacity, i.e., the number of packets that may be queued in the network before a packet is dropped due to con-

1. At least after the *source quench* [Ste94] has been banned.

gestion. A network end-point or flow that is *not* network-limited is called *application-limited*. The load of such flows is limited by the rate at which the sending application can generate data and/or the rate at which the receiving application can consume the data. Examples include interactive TCP-based flows and rate-adaptive real-time flows of which the corresponding application can run its maximum rate (highest quality) source codec. Whether an application-limited flow fully utilizes its available bandwidth depends on the rate at which the sending/ receiving application generates/consumes data.

2.3.2 Congestion Control in TCP

The two error recovery strategies used in TCP (see Section 2.2.1) are coupled with TCP's congestion control scheme [Jac88], [Jac90a], [RFC2581] in the following way. After a timeout-based retransmission, the TCP sender decreases its load to one segment (see the 27th RTT in Figure 2-4). It then enters the *slow start* phase during which it increases the load exponentially until the load reaches one half of its value before the timeout (see the 30th RTT in Figure 2-4). The TCP sender then enters the *congestion avoidance* phase, where it increases the load linearly. The congestion avoidance phase is sometimes also called the *probing phase* because the TCP sender is probing the network for more bandwidth. After a DUPACK-based retransmission, the TCP sender halves its load (see the 6th, 11th, and 18th RTT in Figure 2-4), 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. Load above the "Pipe Capacity" line in Figure 2-4 is queued in the network.

The implementation of this congestion control scheme is intertwined with TCP's window-based flow control scheme through the use of two sender-side state variables: the *slow start threshold (ssthresh)* and the *congestion window (cwnd)*, which are both measured in bytes [Jac88]. A TCP sender is never allowed to have more bytes outstanding than the minimum of the advertised window and the congestion window. That is, a TCP sender's load is limited by

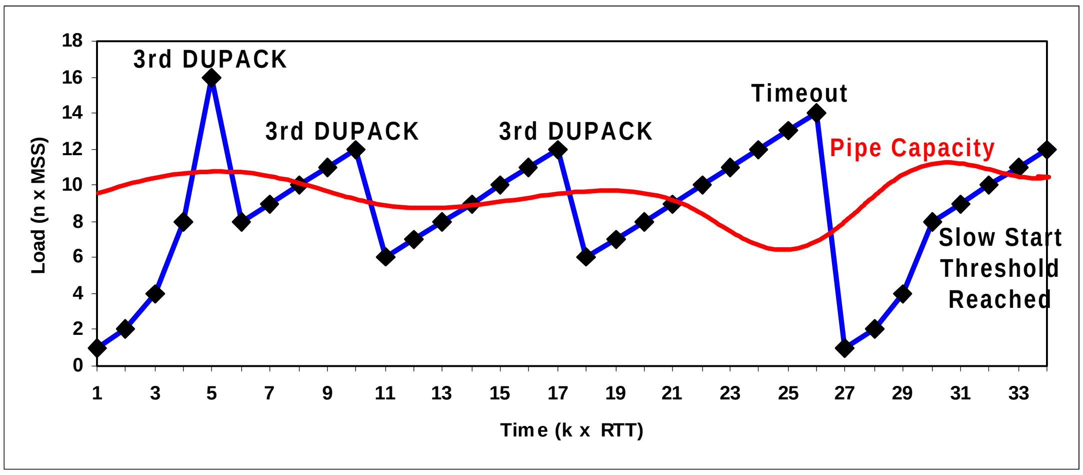


Figure 2-4: A sample evolution of the congestion window.

the flow control imposed by the receiver and by the congestion control (implicitly) imposed by the network. In the latter case a TCP flow is called network-limited as defined above, in the former case it is called *receiver-limited*. In addition, a TCP sender's load is limited by the size of the TCP sender's send socket buffer that is used to hold outstanding segments in case they need to be retransmitted. Such TCP flows are called *sender-limited*. Because the socket buffer sizes are controlled by the application, we consider receiver- and sender-limited TCP-based flows as special cases of application-limited flows.

Figure 2-5 shows the finite state machine[1] implemented for the TCP sender that results in the congestion control behavior depicted in Figure 2-4. When a connection is established, ssthresh and cwnd are initialized to 64 KBytes and MSS, respectively, and the connection enters the slow start phase. In that phase every ACK increases cwnd by one MSS causing an exponential increase in cwnd over RTT, i.e., cwnd is doubled per RTT[2].

As soon as cwnd exceeds ssthresh, the connection enters the congestion avoidance phase. In that phase every ACK increases cwnd by MSS^2/cwnd. This causes a linear increase in cwnd over RTT, i.e., cwnd is increased by one MSS per RTT[3]. When the third DUPACK is received that triggers the fast retransmit (either during slow start or during congestion avoidance), one half of the current value of cwnd is stored in ssthresh[4], and the connection enters the *fast recovery* phase [Jac90a].

The purpose of the fast recovery phase is to reduce the load on the network by one half. This is accomplished by suppressing the transmission of new segments for the first half of the return-

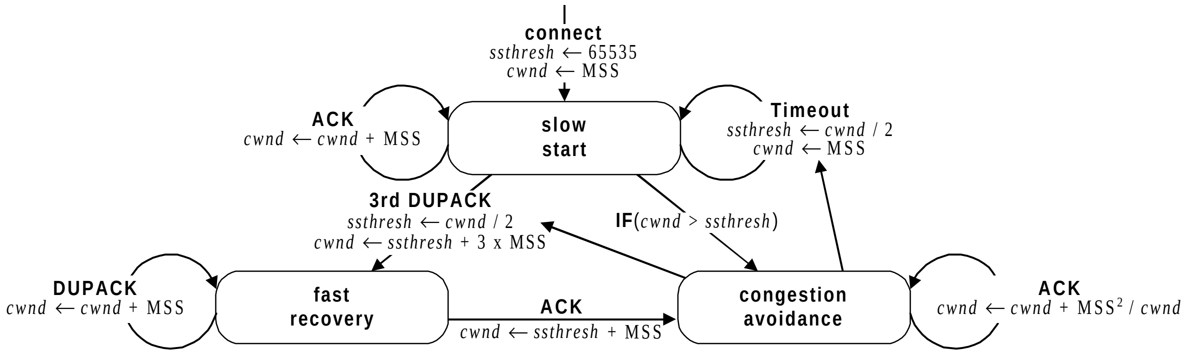


Figure 2-5: Congestion control in TCP.

1. To not confuse the diagram, we do not show the end state. For a complete finite state machine a transition ("disconnect") from each of the states shown in the diagram to the end state must be added.
2. Such an ACK clocks out two segments: one because the sliding window "moved right" by one segment and another one because cwnd increased by one segment. In case delayed-ACKs are used, even three segments are clocked out because in that case, the sliding window "moved right" by two segments.
3. MSS^2/cwnd equals 1/cwnd if cwnd is expressed in multiples of MSS, not bytes. The ACK at the end of each flight will consequently clock out two segments. No segment is sent for the other ACKs of a flight due to the Nagle algorithm [RFC896] which prevents TCP from sending less than a full-sized segment when the sender is expecting an ACK.
4. Note, that the value of ssthresh is always adjusted to a multiple of MSS and is bounded by a minimum of 2 x MSS.

ing DUPACKs (cwnd was set to (ssthresh + 3 x MSS), i.e., cwnd was halved and inflated for the first three DUPACKs) and by sending a new segment for each DUPACK of the second half (cwnd is inflated by MSS for every DUPACK returning after the third DUPACK). When the fast retransmit is acknowledged, cwnd is set to one half of what its value was before the fast retransmit was triggered plus MSS, and the connection enters the congestion avoidance phase.

When a timeout occurs (either during slow start or during congestion avoidance), one half of the current value of cwnd is stored in ssthresh, cwnd is set to MSS, and the connection enters the slow start phase. Note, that the congestion window is updated during the entire lifetime of the connection but only has an effect as long it is smaller than the advertised window, i.e., when the connection is network-limited.

2.3.3 New Developments

In this subsection we briefly explain the following three important developments related to congestion control in the Internet.

- Active Queue Management[1]
- Explicit Congestion Notification[2]
- “TCP-friendly” Congestion Control.

Our purpose is to show that even with the latest developments (1) “packet loss” will remain an important congestion signal in the Internet, (2) no mechanisms exists today that could be used by the network end-points to distinguish between a congestion loss and an error loss, and (3) also non-TCP flows, e.g., real-time flows respond to a packet loss by reducing their load.

It has become clear that TCP’s congestion control mechanisms, while necessary and powerful, are not sufficient to provide good service in all circumstances. Basically, there is a limit to how much control can be accomplished from the end-points of the network. Some mechanisms are needed in the routers to complement the network end-point’s congestion control mechanisms. Active queue management [RFC2309] is such a mechanism.

To a rough approximation, *queue management* algorithms manage the length of packet queues by dropping packets when necessary or appropriate. The traditional technique for managing router queue lengths is to set a maximum length (in terms of packets) for each queue, accept packets for the queue until the maximum length is reached, then reject (drop) subsequent incoming packets until the queue decreases because a packet from the queue has been transmit-

1. The description of active queue management is to a large extent drawn from the text of [RFC2309].
2. The description of explicit congestion notification is to a large extent drawn from the text of [RFC2481].

ted. This technique is known as *tail drop*, since the packet that arrived most recently (i.e., the one on the tail of the queue) is dropped when the queue is full. This method has an important drawback. It allows queues to maintain a full (or, almost full) status for long periods of time, since tail drop signals congestion (via a packet drop) only when the queue has become full. This prevents the queue from absorbing packets that arrive in bursts which is a great concern (see [RFC2309] for detail). The solution is for routers to drop packets before a queue becomes full, so that network end-points can respond to congestion before interface buffers overflow. Such a proactive approach is called *active queue management*. By dropping packets before interface buffers overflow, active queue management allows routers to control *when* and *how many* packets to drop. The main advantages are that packet bursts can be absorbed by the queue, i.e., do not have to be dropped. Furthermore, the smaller queues reduce the end-to-end delay, thus benefiting real-time flows.

Active queue management mechanisms may use one of several methods for indicating congestion to the network end-points. One is to use packet drops, as is currently done. However, active queue management allows the router to separate policies of queueing or dropping packets from the policies for indicating congestion. Thus, active queue management allows routers to explicitly mark packets to signal congestion, instead of relying solely on packet drops. The *Explicit Congestion Notification (ECN)* scheme proposed in [RFC2481] provides a congestion indication for incipient congestion where the notification can sometimes be through marking packets rather than dropping them. While ECN is inextricably tied up with active queue management at the router, the reverse does not hold; active queue management mechanisms have been developed and deployed independently from ECN, using packet drops as indications of congestion in the absence of ECN in the IP architecture.

The ECN scheme requires an ECN field in the IP header with two bits. The *ECN-Capable Transport (ECT)* bit would be set by the sending network end-point to indicate that both network end-points are ECN-capable. The *Congestion Experienced (CE*) bit would be set by the router to indicate congestion to the network end-points. Upon the receipt by an ECN-Capable network end-point of a single CE packet, the congestion control algorithms followed at the network end-points must be essentially the same as the congestion control response to a dropped packet. For example, in ECN-capable TCP, the TCP sender is required to halve its congestion window for any flight of segments containing either a packet drop or an ECN indication. It is important that the network end-points react to congestion at most once per RTT, to avoid reacting multiple times to repeated indications of congestion within a RTT.

It is important to note that ECN is not a replacement for “packet loss” as a congestion signal, nor is it a mechanism that could be used by the network end-points to distinguish between a congestion loss and an error loss. Non-ECN-capable routers may exist for a long time in the

Internet that only drop packets to indicate congestion, and ECN-capable routers under heavy network load may have no other choice but to drop packets.

The Internet is continuously changing. Non-TCP-based flows such as real-time flows are becoming increasingly important. To ensure network stability, such flows must also become loss responsive. Furthermore, because of the dominant role of TCP, the congestion control chosen for non-TCP-based flows must be equivalent to TCP's congestion control. Otherwise, if it was more aggressive, it would create an unfairness towards TCP flows, and vice versa if it was less aggressive. For that purpose, a rate-based equivalent to TCP's window-based congestion control scheme, the so-called *"TCP-friendly" congestion control scheme*, has been developed (e.g., see [MSMO97]). It should be used with rate-adaptive real-time flows.

The following formula yields the "TCP-friendly" packet send rate (also known as *TCP throughput equation*) for network-limited flows. It is determined by one constant, the MSS, and two variables, the RTT and the probing loss rate p described below.

$$\text{Packet-Send-Rate} = \frac{\text{Packets per Cycle}}{\text{Time per Cycle}} = \frac{MSS}{RTT} \times \sqrt{\frac{3}{2p}}$$

The exact derivation of the formula is not important for our work (see [MSMO97] for detail) but the idea behind it is relevant. The formula is derived from the ideal case where a single network-limited TCP bulk data transfer runs over a non-shared bottleneck link with a fixed bandwidth, the TCP sender always sends full-sized segments, and no packets are lost due to transmission error. In this case the TCP sender goes through periodic congestion avoidance cycles. With the additive increase policy of one packet per round trip time, as described above, this leads to a single dropped packet at the end of each cycle (see Figure 2-6). We call the reciprocal of the number of packets that are sent per cycle, including the dropped packet, the *flow's*

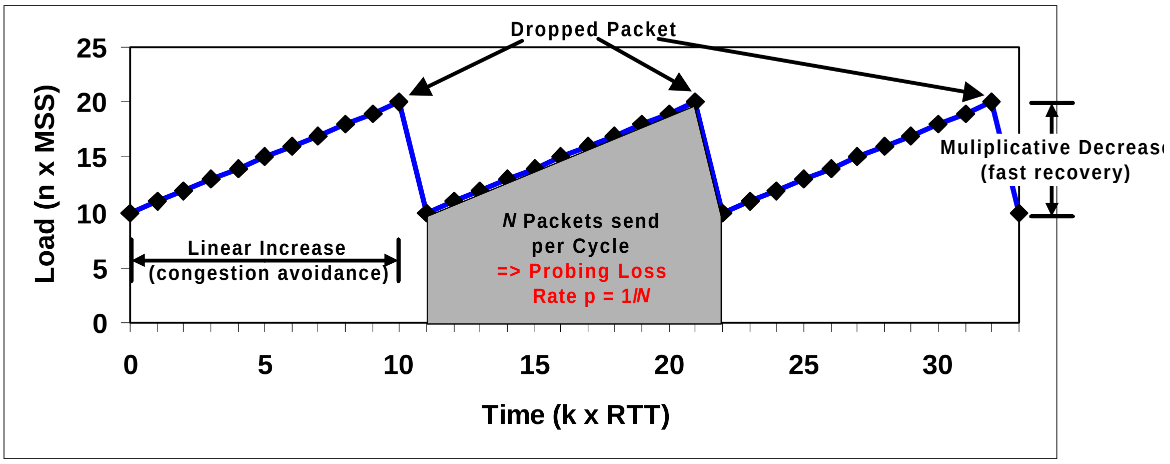


Figure 2-6: Deriving the "TCP-friendly" congestion control scheme.

probing loss rate p. Note that in practice a flow’s probing loss rate might vary considerably over time as the flow’s available bandwidth and/or the flow’s RTT changes.

The important observation for our work is that the evolution of the flow’s load as depicted in Figure 2-6 would be the same if those periodic packet losses were caused by transmission error, not congestion. We define as the *flow’s error loss rate* the rate at which packets of a given flow are lost due to transmission error. Note, that while the probing loss rate is a property that can only be associated with network-limited flows, the error loss rate can be associated with both network- and application-limited flows. With these terms the following first-order rule of thumb can be formulated: the throughput provided by a network-limited flow is insensitive to transmission errors as long as the flow’s error loss rate stays below the flow’s probing loss rate. This rule is valid as long as packet losses occur periodic. Otherwise, the error probability process might need to be considered, too.

2.4 Link Layer Error Control in Wireless Networks

A multitude of wireless networks exist today spanning a wide range features, e.g.:

- frequency band
- physical layer access (e.g., time division vs. code division)
- access vs. transit network
- cellular vs. trunk systems
- short vs. long range (for cellular systems this determines the cell size)
- multi-hop vs. single-hop radio
- degree of terminal mobility and the support for it from the network
- supported traffic type (data only, voice only, both)

Similarly, the link layer error control schemes implemented in those wireless networks can be very different, ranging from sophisticated to almost non-existent. Some design choices exclude certain link layer error control mechanisms, e.g., satellite links are often uni-directional excluding the possibility of link layer error recovery. Other design choices make some link layer error control techniques unnecessary, e.g., handover control is not an issue in geo-stationary satellite systems where a single satellite often covers the entire geographical area for which service needs to be provided.

A link that spans across the wireless segment(s), i.e., the air-interface(s), of a wireless network is called a *wireless link* as opposed to a *wireline link*. Likewise, we refer to *wireless networking* as any form of IP-based communication over paths that include wireless links. Independent of the features of a wireless network, wireless links are often problematic: whereas *Bit Error Rates (BER)* on today's wireline links can be neglected, this is not true for wireless links. In addition, when hosts are mobile, cell handovers (explained in Section 2.4.2) may cause data loss and some wireless networks may in certain situations only provide intermittent connectivity. We regard intermittent connectivity as a "long" transmission error that do not have to be treated different from "normal" transmission errors. All three cases

- may either cause packet loss due to transmission error, i.e., an error loss, or else,
- if such a loss can be prevented by link layer error control schemes, those may cause an increased packet transmission delay over the wireless link.

Our intention in this section is to give a brief overview of the basic link layer error control schemes that can be used to prevent an error loss. In the following subsections, we first describe each scheme in general. Then we exemplify how they are implemented in GSM-CSD, the Circuit-Switched Data service implemented in GSM, as that system supports all of the link layer error control schemes that are relevant for our work. Furthermore, the GSM-CSD system has been the basis for all of our measurement-based analysis of wireless links described in Chapter 3 and Chapter 4. We therefore provide a brief description of the GSM-CSD system first.

2.4.1 Circuit-Switched Data Transmission in GSM

Unlike earlier analog cellular telephone systems, data services are an integral part of a GSM digital cellular telephone network and are equally supported together with ordinary voice services. Figure 2-7 shows the basic components used for circuit-switched data transmission in GSM. A mobile host, a laptop or palmtop, is connected to the GSM network using a GSM mobile phone (*Mobile Station (MS)*) and a device running the *Terminal Adaptation Function (TAF)*. Unlike in first generation analog cellular systems, the TAF is *not* a modem. The modem (running the standard modem protocols V.42/V.32) resides in the network, in the *Interworking Function (IWF)* of the *Mobile Switching Centre (MSC)*. An MSC is a backbone telephone switch that routes circuits within the GSM network and also serves as a gateway to the fixed *Public Switched Telephone Network (PSTN)*. The radio interface is provided by a *Base Transceiver Station (BTS)* (or simply *base station*) which together with other BTSs is controlled by one *Base Station Controller (BSC)* which in turn is bundled with other BSCs by one MSC.

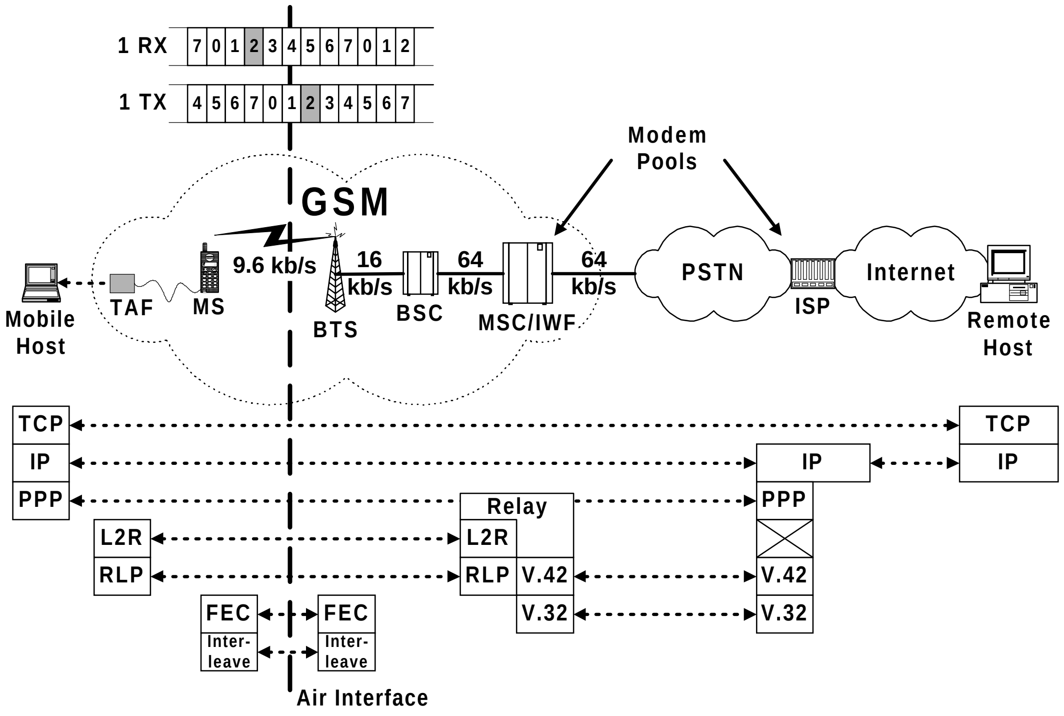


Figure 2-7: TCP/IP over GSM Circuit-Switched Data (CSD).

The GSM air-interface itself is based on frequency division of the *uplink*, i.e., from the mobile host to the network, and the *downlink*, i.e., from the network to the mobile host. Each carrier, comprising a pair of one up- and one downlink frequency, is time-shared among 8 users. At call-setup time a mobile phone is assigned a *channel*, defined as the tuple [carrier frequency number, time-slot number] that remains allocated for the entire duration of the call. The up- and downlink frequencies are offset by 3 time-slots so that a mobile phone never has to transmit and receive at the same time which reduces its complexity considerably. This is sketched in Figure 2-7. The maximum data rate over the air-interface is 9.6 Kbit/s synchronous (i.e., 1200 bytes/s). Optionally, a reliable link layer protocol called the *Radio Link Protocol (RLP)* (explained in Section 2.4.3) can be run between the TAF and the IWF, which is called the non-transparent data service. An additional protocol called the *Layer 2 Relay (L2R)* protocol is used by the non-transparent data service for flow control, and communicating status control signals between the TAF and the IWF.

Given this architecture a mobile host then uses a standard serial link protocol like PPP to connect to the Internet via GSM and a dial-in access fabric of an *Internet Service Provider (ISP)*. Thus, from the perspective of an ISP the GSM link appears just like any other dial-up connection. Additional details about how GSM-CSD is being enhanced to support higher bandwidth and direct Internet access is provided in [LR99].

2.4.2 Handover Control

Cellular networks provide service to a geographical area be dividing it into *cells*. Each cell is operated by one base station. Many cellular networks, e.g., GSM, support *terminal mobility* which among other functions (e.g., location updates or paging [MP92]) requires *handover control*. It allows a user to roam between cells while continuing an ongoing communication. In that case the mobile phone and the network perform an *handover*, which means that the communication with the base station of the cell the user left is switched to the base station of the cell the user entered. Depending on the wireless network, one distinguishes between different handover types: intra-BTS (when only the carrier within the cell is changed), inter-BTS, inter-BSC, and inter-MSC.

We call the *anchor point* that location in the network after which data sent downlink takes a different route after a handover has been performed. Error losses may occur during a handover to packets that are being transmitted over the air-interface (uplink and/or downlink), but also to packets queued downlink *after* the anchor point (see Figure 2-8). Those packets are either discarded or a (link layer) handover control mechanism ensures that they are transferred to the

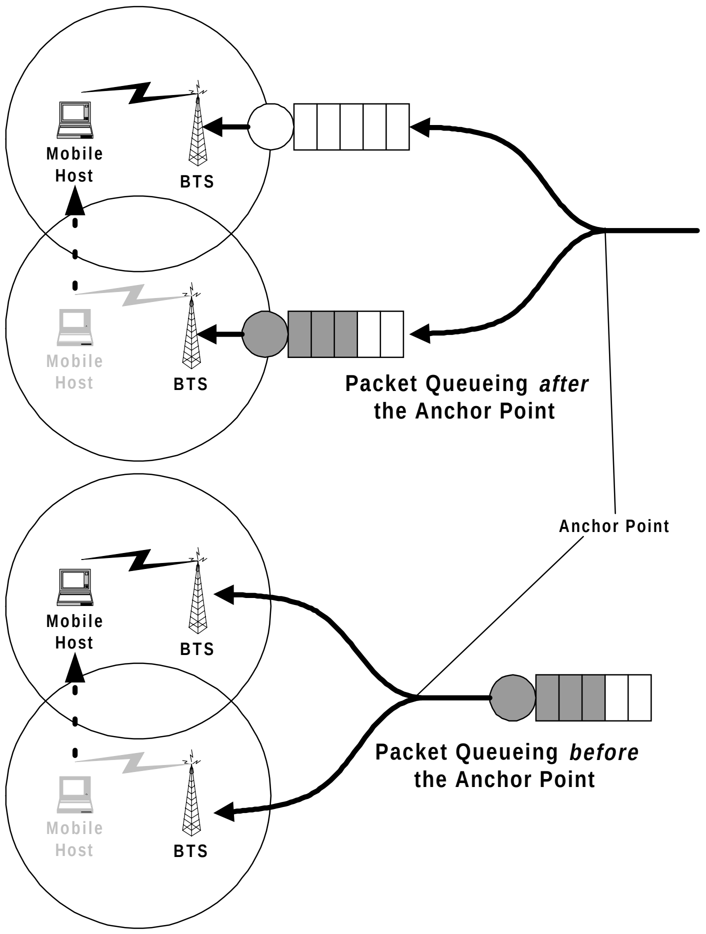

Mobile Host
BTS
Mobile Host
BTS
Packet Queueing *after* the Anchor Point
Anchor Point
Mobile Host
BTS
Packet Queueing *before* the Anchor Point
Mobile Host
BTS

Figure 2-8: Two Approaches to implement Handover Control.

new downlink queue. If the packets are discarded, we also speak of an error loss. In GSM-CSD, only packets that are being transmitted over the air-interface can be lost, because queuing always occurs *before* the anchor point, independent of the handover type. More precisely, the last downlink queue resides in the IWF which remains the same for the entire duration of a circuit-switched data call.

In summary, handovers may cause error losses. Alternatively, handover control mechanisms may increase the packet transmission delay in the event of an handover.

2.4.3 Link Layer Error Recovery

The Radio Link Protocol (RLP) [GSM04.22a], [GSM04.22b] is a link layer protocol that, like TCP, provides a reliable byte stream service. It uses selective reject and checkpointing for error recovery (see [GSM04.22a] or [Tan89] for detail on those mechanisms). RLP's error recovery persistency can be configured via a parameter that defines the maximum number of retransmissions of a single frame. The default of that parameter is 6 which increases the packet transmission delay by 2.5 s in the worst case. RLP may be implemented as a fully-reliable or a semi-reliable protocol. When implemented as a semi-reliable protocol, RLP resets the link, i.e., re-initializes the sequence numbers, when its error recovery persistency is reached.

The RLP frame size is fixed at 240 bits aligned to the FEC coder (explained in Section 2.4.4). Thus, an RLP frame does not require begin/end delimiters. RLP introduces an overhead of 48 bits per RLP frame[1], yielding a user data rate of 9.6 Kbit/s in the ideal case[2]. Note, that when accessing the Internet via GSM-CSD two link layer protocols are used, RLP and a framing protocol like PPP. Both link layer protocols have different termination points. The link is provided by PPP, while RLP only spans a section of that link. Because RLP is terminated in the IWF, it ensures reliability in the event of cell handovers. In fact, when RLP is running, the last downlink queue is the send buffer of the RLP sender.

In summary, link layer error recovery is a mechanism that, depending on its error recovery persistency, can reduce a flow's error loss rate. The drawbacks are a reduced user data rate due to the link layer protocol overhead, and that the packet transmission delay becomes variable (bounded by the error recovery persistency).

1. Of those 48 bits, 8 bits are actually introduced by L2R.
2. Note that the transparent (not running RLP) GSM-CSD service introduces a wasteful overhead of modem control information that also reduces the user data rate to 9.6 Kbit/s.

2.4.4 Forward Error Correction and Interleaving

The slot cycle time on a GSM-CSD channel is 5 ms on average. This allows 114 bits to be transmitted in each slot, yielding a gross data rate of 22.8 Kbit/s. The fundamental transmission unit in GSM is a *data block* (or simply *block*). The size of a data block encoded with the *Forward Error Correction (FEC)* scheme defined for GSM-CSD [GSM04.21] is 456 bits (the payload of 4 slots). The size of an unencoded data block is 240 bits, resulting in a data rate of 12 Kbit/s (240 bits every 20 ms).

An FEC scheme is used to protect data against bit transmission errors by adding redundant information (coding) before it is transmitted. The redundant information is used by the FEC decoder to reconstruct the unencoded data block. Whether the data is correctly reconstructed depends on the number and distribution of bit errors in the corresponding encoded data block after its transmission. Note that an FEC decoder cannot perform error detection, i.e., it cannot verify whether it has correctly decoded a received data block. In case the non-transparent GSM-CSD service is used, this responsibility is left to RLP. The performance (strength) of an FEC scheme, e.g., defined by the probability that data blocks are correctly decoded, depends on the characteristics of the physical medium and the amount of redundant information added per data block.

Interleaving is a technique that is used in combination with FEC to combat burst errors. Instead of transmitting a data block as one entity, it is divided into smaller fragments. Fragments from different data blocks are then interleaved before transmission. A few of these smaller fragments can be completely corrupted while the corresponding data block can still be reconstructed by the FEC decoder. The interleaving scheme chosen for GSM-CSD interleaves a single data block over 22 TDMA slots [GSM05.03]. Transmitting it as one entity would only require four consecutive slots. The disadvantage of this deep interleaving is that it introduces a significant one-way *latency*, i.e., a fixed delay component of the packet transmission delay that is independent of the packet size, of approximately 90 ms[1]. This high latency can have a significant adverse effect on interactive protocols [LR99]. Further details on FEC and interleaving can be found in [MP92].

In summary, FEC and interleaving are a mechanisms that can reduce a flow's error loss rate while keeping the packet transmission delay constant. The drawbacks are a reduced user data rate due to the FEC overhead, and (in some cases) additional latency introduced into the packet transmission delay by the interleaving scheme.

1. Note that voice is treated differently in GSM. Unencoded voice data blocks have a size of 260 bits and the interleaving depth is 8 slots.

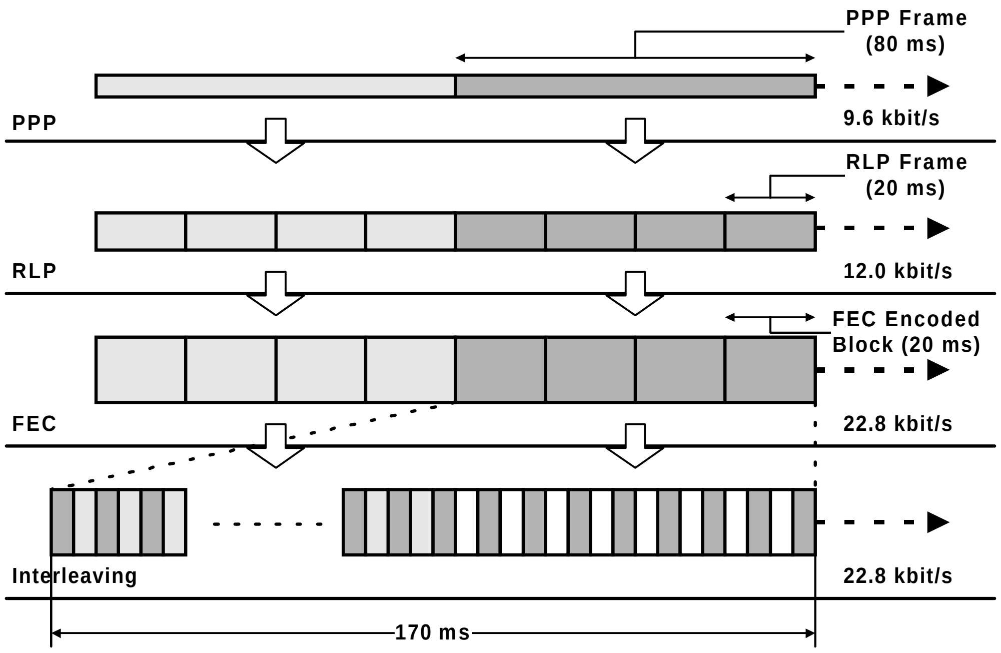


Figure 2-9: Error control in GSM Circuit-Switched Data.

Figure 2-9 exemplifies how two consecutive PPP frames are transmitted in GSM-CSD. As in Figure 2-1, the vertical dimension is bandwidth and the horizontal dimension is time. Each of the shaded boxes corresponds to a (PPP or RLP) frame, data block, or the transmission unit of the interleaver. The area of each box is the data unit size. In this example the PPP frames each occupies 80 ms in the time dimension. An RLP frame and an FEC block always occupies 20 ms. Both RLP and the FEC layer add overhead as explained above and thus more bandwidth is required for the same amount of PPP data. Figure 2-9 also indicates how interleaving adds latency thereby increasing the overall packet transmission delay over the air-interface to 170 ms.

2.5 The Problem: Inefficient Cross-Layer Interactions

In general terms, we speak of an *inefficient cross-layer interaction* when functions at different protocol layers interact in an unforeseen or inadvertent manner, which then results in reduced performance provided by the network and/or the network end-points. How much the performance is affected by inefficient cross-layer interactions depends on the frequency of their occurrence over a given period of time and the impact of each individual one.

A number of inefficient cross-layer interactions have been described in literature. For example, the authors of [FJ94] study the inefficient effects that synchronization of periodic routing messages cause, and show that this can be avoided by the addition of randomization to the traffic sources. The authors of [KM87] propose solutions to mitigate the inefficient effects that IP fragmentation causes for the processing performance of routers and the throughput performance provided by reliable end-to-end protocols.

In this section, we explain all known inefficient cross-layer interactions that may occur in wireless networking. Their root lies in the occurrence of one of the following two link layer events already described in Section 2.4:

- packet loss due to transmission error, i.e., error loss, and
- increased packet transmission delay caused by a link layer error control scheme.

Directly or indirectly via another link layer function, these may trigger end-to-end error and congestion control schemes. If triggered, this may reduce the performance provided by the respective network end-points. This concerns throughput, response times, or human perceived quality (e.g., for audio). In some cases, the triggering of an end-to-end control schemes is the result of a misinterpretation of the event by a network end-point. The key reason is that the Internet does not provide mechanisms for a network end-point to distinguish between

- congestion loss,
- error loss, and
- packets that are considered lost by a network end-point when in fact those packets are *not* lost but merely delayed in the network.

Moreover, it seems unlikely that such mechanisms will get developed (*and* deployed) soon (see also Section 2.3.3). That would require explicit signalling from a link layer to a network end-point. That is not always possible, e.g., when the signal needs to cross the link that is “down” which might have been the trigger for the signal in the first place. Such cross-layer signalling would also require that the link layer can identify the respective network end-point from information in the packet headers. That is also not always possible, e.g., when network layer encryption spans the link[1].

2.5.1 Underestimation of the Available Bandwidth

The fundamental problem of wireless networking is that a loss responsive network end-point might underestimate its available bandwidth. If this happens, it results in unnecessarily reduced application layer performance, particularly throughput, i.e., the corresponding applications get less than their fair share of the bottleneck link’s bandwidth. The reason is that error losses falsely trigger end-to-end congestion control, i.e., a loss responsive network end-point

1. Unless, the link layer was a trusted party and could decrypt and later again encrypt a flow’s packets.

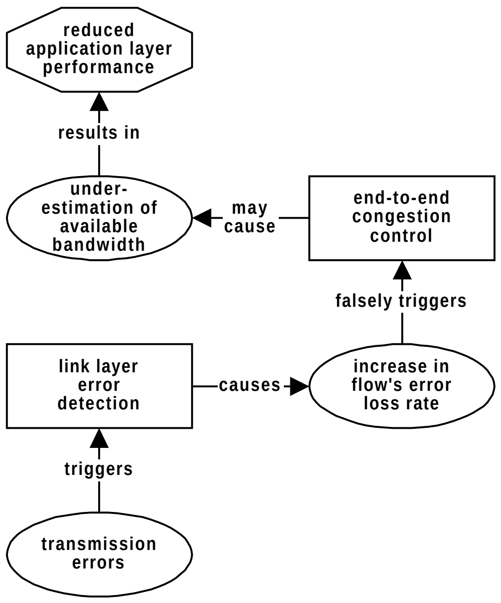

reduced application layer performance
results in
under-estimation of available bandwidth
may cause
end-to-end congestion control
falsely triggers
link layer error detection
causes
increase in flow's error loss rate
triggers
transmission errors

Figure 2-10: Cross-layer interaction leading to underestimating the available bandwidth.

mistakes the packet loss as a congestion signal. This inefficient cross-layer interaction is depicted in Figure 2-10.

However, transmission errors on a wireless link do not necessarily have to result in an underestimation of the available bandwidth. It depends on the flow's properties over its lifetime, mostly its error loss rate, load, and pipe capacity. In Section 2.3.3 we formulated the rule that the throughput provided by a network-limited flow is insensitive to transmission errors as long as the flow's error loss rate stays below the flow's probing loss rate[1]. Beyond that, it depends on the ratio of the flow's load to the flow's pipe capacity and how congestion control is implemented. For example in TCP, if packet losses were always detected by DUPACKs and the load at times when no packet losses occurred may exceed two times (or more) the pipe capacity, then the flow's error loss rate may even be larger then the flow's probing loss rate without causing underestimation of the available bandwidth. In that case, the TCP flow would always have more packets queued in the network than could drain from the bottleneck link during the fast recovery phase, i.e., the flow's available bandwidth would always be fully utilized. In Section 5.2.3 we provide a more illustrative analysis of this issue. For application-limited flows to which we cannot assign a probing loss rate, the problem is similar. For those, it solely depends on the ratio of the flow's load to the flow's pipe capacity as described above.

2.5.2 Inefficiency of End-to-End Error Control

Handling error losses by an end-to-end, instead of a link layer, function is often inefficient resulting in reduced application layer performance. Two such cases are depicted in

1. The probing loss rate the flow would have over the same wireless link in the (ideal) error-free case.

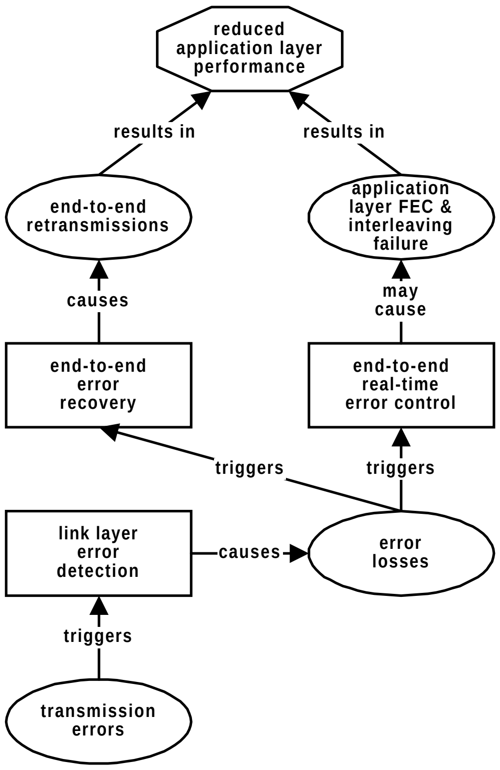

reduced application layer performance
results in
results in
end-to-end retransmissions
application layer FEC & interleaving failure
causes
may cause
end-to-end error recovery
end-to-end real-time error control
triggers
triggers
link layer error detection
causes
error losses
triggers
transmission errors

Figure 2-11: Cross-layer interaction caused by the inefficiency of end-to-end error control.

Figure 2-11. The figure does not show interactions with end-to-end congestion control which in addition gets falsely triggered causing the problems described in Section 2.5.1.

Lost packets of a reliable flow need to be retransmitted end-to-end. Thus, each packet that is lost has wasted resources on each link on the path's segment "before" the link where the loss occurred. If the bottleneck link is located on that segment than those lost packets have also wasted available bandwidth. When caused by error losses, this inefficiency could possibly have been prevented by link layer error recovery. However, this might in turn trigger the inefficient cross-layer interactions explained in Section 2.5.3 and Section 2.5.4.

Lost packets of a real-time flow lead to a reduced user level quality (e.g., the user perceived audio quality) if application layer FEC and interleaving (if those functions exist) fail to reconstruct the packet. When caused by error losses, this inefficiency could possibly have been prevented or mitigated by an appropriate link layer FEC and interleaving scheme.

2.5.3 Overly Strong Link Layer Error Control

The opposite of the case described in Section 2.5.2 is the case when link layer error control too strongly protects the wireless link. Figure 2-12 shows the different inefficient cross-layer interactions this may cause. First, FEC and interleaving increase the packet transmission delay by introducing latency. For example, in GSM-CSD a one-way latency of more than 90 ms is introduced (see Section 2.4.4). This might exceed the tolerable delay of some real-time applications. Although not shown in Figure 2-12, it also increases the response times that interactive reliable flows may provide [Bau97], [LR99].

Another variant of the same problem is when link layer error recovery is performed for real-time flows. In the case of a transmission error, this increases the packet transmission delay of the respective packet. That in turn may cause subsequent packets to also get delayed due to the link layer in-order delivery function, even if those packets were not affected by a transmission error. When the link layer in-order delivery function serves multiple different flows, this problem is known as *head of line blocking*. For example, error recovery may delay the delivery (to the receiving user process) of a large TCP segment belonging to one flow, while the link layer in-order delivery function blocks the delivery of subsequent packets belonging to other flows (e.g., small packets from a real-time flow or small ACKs from a second TCP flow) that have already correctly arrived at the receiving link layer. Those delays might exceed the tolerable delay of some real-time applications and/or disrupt the ACK clock of the second TCP flow. The latter (not shown in Figure 2-12) might lead to so-called *ACK compression* [ZSC91], [Mog92] which may cause further problems [Pax97c], e.g., that a TCP sender responds by sending a packet burst that might lead to multiple dropped packets.

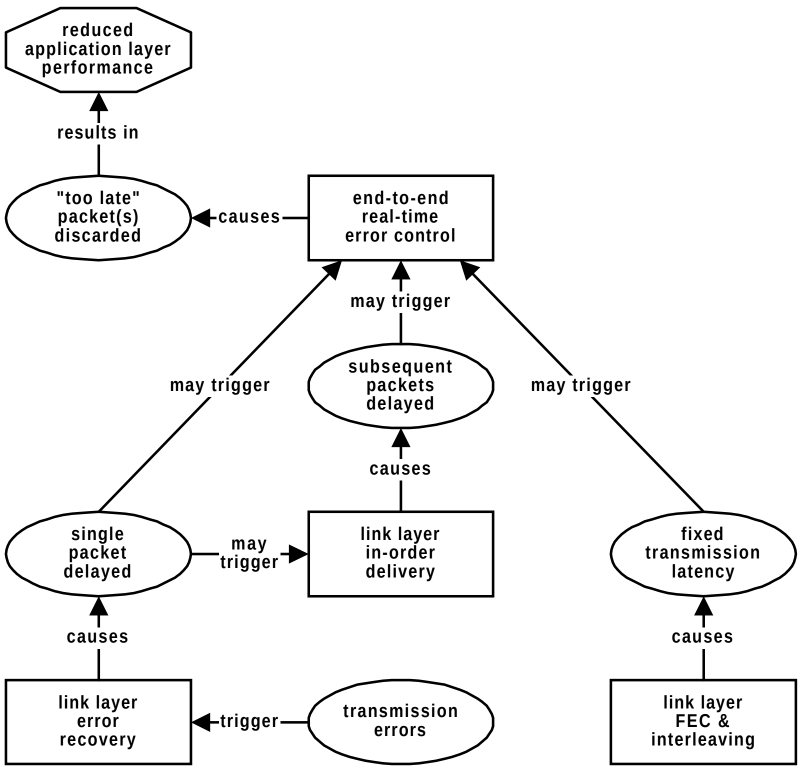

reduced application layer performance
results in
"too late" packet(s) discarded
causes
end-to-end real-time error control
may trigger
may trigger
may trigger
subsequent packets delayed
causes
single packet delayed
may trigger
link layer in-order delivery
fixed transmission latency
causes
causes
link layer error recovery
trigger
transmission errors
link layer FEC & interleaving

Figure 2-12: Cross-layer interaction caused by overly strong link layer error control.

2.5.4 Competing Error Recovery

Undesirable competition is introduced when error recovery is run both at the link layer and end-to-end. It might lead to the following wasteful situation. The link layer is retransmitting one or more packets, i.e., delaying the packet(s) in the network. Simultaneously, one of the network end-points considers the packet(s) lost, and falsely triggers that the same packets are retransmitted end-to-end. Those spurious end-to-end retransmissions reduce the end-to-end throughput and increase application layer response times. It may even occur that two or more copies of the same packet reside in the send buffer of the sending link layer at the same time.

In general, one could say the competing error recovery is caused by an inner control loop (link layer error recovery) reacting to the same signal as an outer control loop (end-to-end error recovery), without coordination between both loops (see Figure 2-13).

For TCP, two forms of inefficient cross-layer interactions can be caused by competing error recovery, depending on whether the link layer implements the in-order or out-of-order delivery function. Similar interactions are likely to exist for other reliable end-to-end protocols. Both cases are shown in Figure 2-14, and are explained in turn below. The figure does not show interactions with end-to-end congestion control which is falsely triggered if end-to-end error control is falsely triggered. This causes additional problems that are described in Section 2.5.1.

If a link layer implements the in-order delivery function the receiving link layer delays all packets of a flow until they have been received correctly and in-order (assuming that the error recovery persistency is not reached and that no alternative route exists between the TCP sender and receiver). During this time the TCP sender does not receive any of the corresponding ACKs[1]. It is forced into a spurious timeout (and subsequent slow start) that will trigger the

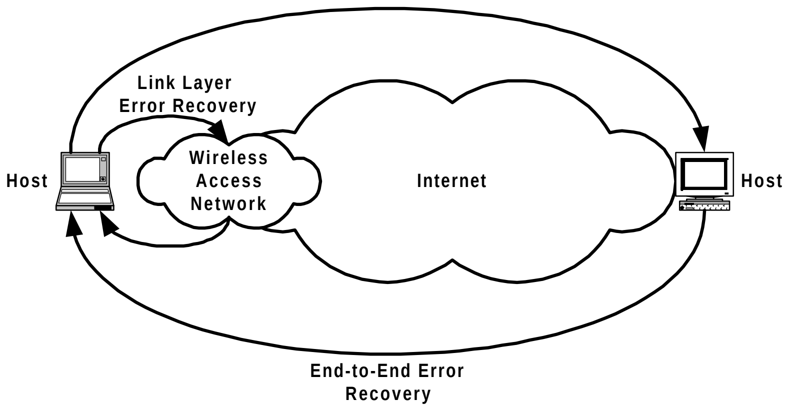


Figure 2-13: Uncoordinated control loops reacting to the same signal.

1. Either the segments containing the data have not yet reached the TCP receiver because they are queued at the link layer, or the corresponding ACKs itself are queued at the link layer.

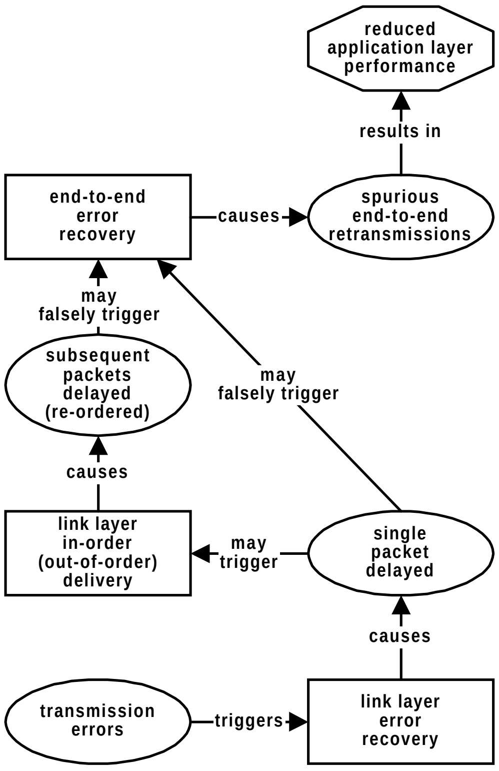

reduced application layer performance
results in
end-to-end error recovery
causes
spurious end-to-end retransmissions
may falsely trigger
subsequent packets delayed (re-ordered)
may falsely trigger
causes
link layer in-order (out-of-order) delivery
may trigger
single packet delayed
causes
transmission errors
triggers
link layer error recovery

Figure 2-14: Cross-layer interaction leading to competing error recovery.

spurious retransmission of the oldest outstanding segment. In Section 5.1, we show that TCP has another fundamental problem that eventually leads to the (go-back-N-style) spurious retransmission of an entire flight of packets after a spurious timeout.

A *packet re-ordering* event occurs when a packet arrives at the receiving network end-point after one or more packet(s), that had left the sending end-point 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.

If a link layer implements the out-of-order delivery function, it may re-order packets. If segments of a TCP connection are re-ordered beyond the DUPACK-Threshold, i.e., if the re-ordering length is equal or greater than three (the commonly implemented DUPACK-Threshold), this triggers a *spurious fast retransmit*, i.e., a spurious retransmission that was *not* triggered by a spurious timeout, and subsequently the fast recovery phase. Also in Section 5.1, we study this inefficient cross-layer interaction in detail.

2.5.5 Failure of Link Layer Differential Encodings

The problem that uncorrected transmission errors cause for link layer differential encodings, such as TCP/IP header compression, has already been described in Section 2.2.3. The ineffi-

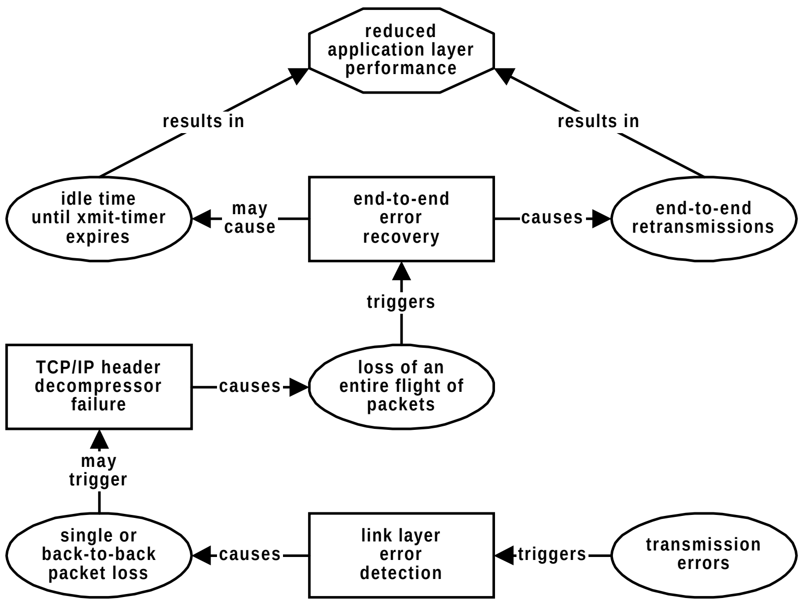

reduced application layer performance
results in
results in
idle time until xmit-timer expires
may cause
end-to-end error recovery
causes
end-to-end retransmissions
triggers
TCP/IP header decompressor failure
causes
loss of an entire flight of packets
may trigger
single or back-to-back packet loss
causes
link layer error detection
triggers
transmission errors

Figure 2-15: Cross-layer interaction leading to the failure of TCP/IP header compression.

cient cross-layer interaction that this causes is depicted in Figure 2-15. The figure does not show interactions with end-to-end congestion control which in addition gets falsely triggered causing the problems described in Section 2.5.1. The arrow “may trigger” is labeled that way because [RFC1144] cannot tolerate a single packet loss, whereas [RFC2507] cannot tolerate it if two or more packets are lost back-to-back. We have measured that effect and illustrate it in Section 4.2.3 and Section 4.3.3.

Failure of the TCP/IP header decompressor reduces the end-to-end throughput and increases application layer response times in two ways. First the end-to-end retransmissions reduce the performance as described in Section 2.5.2. Secondly, the TCP sender is forced into a timeout (see Section 2.2.3), which may result in an idle time where no packet is in transit end-to-end until the retransmission timer expires. This idle time can be “long” because of the overly conservative retransmission timer that is implemented in TCP. The latter is further studied in Section 5.3.

2.6 Related Work

While we are not aware of related work that studies the problem of loss responsive flows over wireless links in general, the particular problem of TCP over wireless links has been widely investigated. In Section 2.6.1 we briefly summarize and classify the solutions that have been proposed in literature, and point out advantages and problems that are specific to each approach. We ignore proposals that suggest protocols that result in flows which are not loss responsive, e.g., [CLZ87]. In Section 2.6.2, we evaluate those approaches with respect to which of the inefficient cross-layer interactions explained in Section 2.5 they solve, and which deployment concerns they raise.

2.6.1 Classification of Existing Approaches

We have categorized the proposed solutions in Figure 2-16. Note that the shaded areas indicate whether a transport protocol or its implementation must be changed, or whether transport protocol dependent state has to be maintained in the network.

Pure end-to-end approaches do not maintain transport layer state in the network and make no assumptions about the existence of dedicated link layer (e.g., error recovery) or network layer (e.g., cell handover indications) support. This category includes (1) existing end-to-end protocols (e.g., TCP); (2) extensions of existing end-to-end protocols and/or their implementation (e.g., [RFC2018], [RFC2481], [SF98], and our own proposal explained in Section 5.2); and (3) new or not widely deployed end-to-end protocols (e.g., [CLZ87]). Adding selective acknowledgements (SACK) to TCP [RFC2018] is a way to deal with transmission errors over unreliable wireless links [BPSK96]. In case a particular segment must be retransmitted more than once, [SF98] proposes a further enhancement to the TCP sender assuming a SACK-capable TCP receiver.

Hard-state transport layer approaches encompass all forms of "splitting" by running a separate transport layer connection over the wireless link. Those approaches require a Performance Enhancing Proxy (PEP) in the network that has access to transport layer headers. The concept was initially proposed in [BB95], and has been used in other work including satellite access links [HK99]. Any protocol can be chosen for the wireless link, e.g., [Bau97], [BS97] and [KRLKA97] suggest combining splitting with a link layer approach. Some split solutions [Bau97], [BB95], [HK99], [KRLKA97] violate the end-to-end semantics of TCP's error con-

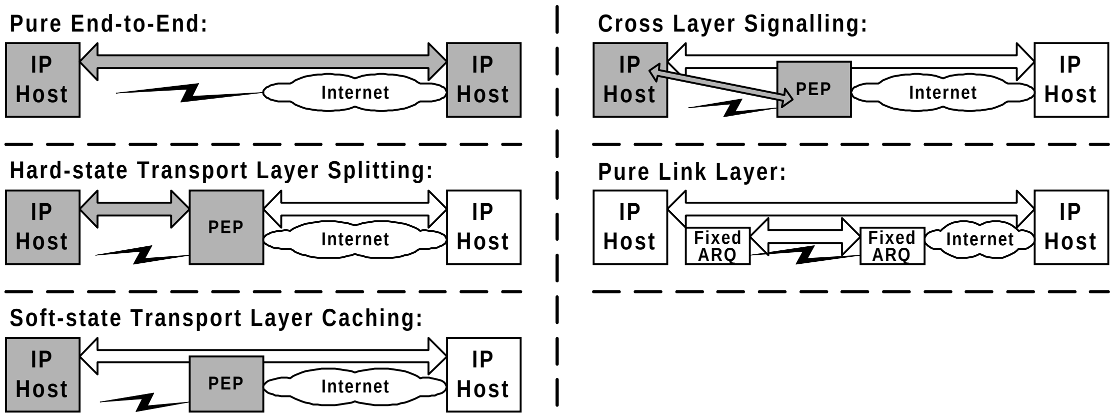


Figure 2-16: Existing approaches to solve "TCP over Wireless".

trol scheme. They allow the PEP to send ACKs back to the TCP sender before the corresponding segments have reached the TCP receiver. The solution proposed in [BS97] implements splitting while maintaining these end-to-end semantics. It targets the problem of frequent and/or long disconnections. In case of disconnections, the PEP issues ACKs, which shrink the advertised window to zero. This forces the TCP sender into *persist mode* [Ste94]. In this mode the TCP sender does not suffer from timeouts nor the exponential backoff of the RTO. The major benefit of hard-state transport layer solutions is that the end-to-end flow is shielded from error losses on the wireless link, and that the flow can fully utilize its available bandwidth over the entire path.

The Snoop protocol developed in [BSK95] and extended in [BK98] implements “TCP-aware” local error recovery (as discussed further below, we avoid calling this *link layer* error recovery as we associate a different meaning with that term). It also requires a PEP in the network that has access to transport layer headers. Its advantage over split solutions is that the transport layer state maintained in the network is *soft*, i.e., it is not crucial for the end-to-end connection. This eases the implementation of handover control schemes. However, the Snoop protocol has limitations. When sending *to* the mobile host, packets dropped due to congestion at a bottleneck link between the wireless link and the mobile host (i.e., when the wireless link is not the last-hop link) are mistaken for error losses by the TCP-aware cache (the *snoop agent*). The congestion signal, the three DUPACKs, is not propagated back to the sender. For packets sent *from* the mobile host, the proposed *Explicit Loss Notification (ELN)* scheme [BK98] causes a problem. If the wireless link itself, or any other link between the mobile host and the wireless link, becomes the bottleneck, congestion losses cannot be distinguished from error losses[1]. An ELN is sent in either case by the snoop agent, and the sender relies on external means to get the congestion signal (e.g., the source quench). Consequently, in both cases the end-to-end semantics of TCP’s congestion control scheme are violated. A “fix” has been proposed in [BK98] by implementing the snoop agent symmetrically on *both* sides of the wireless link. The ELN scheme can then be used to unambiguously discriminate between error losses and congestion losses. However, the approach is equivalent to link layer error recovery (segmentation and reassembly could be added to the snoop protocol) leaving questionable the benefit of “TCP-awareness”.

Cross layer approaches make the flow’s sender aware of the properties of the wireless link. This is achieved by having the link or network layer signal the transport layer about specific events, like link outages, error losses, or cell handovers, so that it can adapt accordingly. Also those approaches require a PEP in the network that has access to transport layer headers. The solution proposed in [DMT96] uses the *Internet Control Message Protocol (ICMP)* [RFC792] to out-of-band signal active receivers that the link is in a bad state (e.g., a link outage). The

1. In Section 4.2.2 these effects were measured where packets got dropped locally at the mobile host because of congestion at the first-hop wireless link.

receiver reflects the signal to the sender using a dedicated TCP option field. In the network studied in [DMT96], the reverse path did not traverse the "problem link". The ELN scheme proposed in [BK98] is similar, but uses in-band signals (ELNs are piggy-backed onto certain DUPACKs as a TCP option[1]) to inform the TCP sender about error losses. This assumes perfect knowledge of such events, which in practice is difficult to implement. [CI95] focuses on the problem of data loss caused by cell handovers. The solution does *not* require access to transport layer headers in the network, but instead requires support from the mobility management function provided by the wireless network. It suggests informing the TCP sender about a cell handover to trigger TCP's fast retransmit algorithm and thereby avoids idle waits for TCP's retransmission timer to expire to recover the loss.

Pure link layer approaches do not maintain transport layer state in the network and make no assumptions about the existence of dedicated support from an end-to-end protocol. In particular - and as opposed to our solution of flow-adaptive wireless links explained in Chapter 4 - pure link layer approaches do not derive any decision from information in a flow's headers. Link layer solutions in general (not only the "pure" ones) aim at hiding the artifacts of the wireless link to higher layer flows. The techniques include adaptive FEC, interleaving, adaptive power control, and link layer error recovery protocols [GSM04.22a], [GSM04.22b], [GSM04.60], [Kar93], [BDSZ94], [IEEE802.11]. Some wireless networks use some of these, e.g., early commercially available *Wireless Local Area Networks (WLANs)*, while others implement combinations, e.g., GSM-CSD. None of the variations of the Snoop protocol are considered as *link layer* error recovery (rather local error recovery) solutions. The basic difference is that true link layer solutions are not tied to the semantics of any higher layer protocol. We also consider approaches to make TCP/IP header compression more robust against lost packets (deltas) as link layer solutions, e.g., the twice algorithm (see Section 2.2.3) and the header request mechanism proposed in [RFC2507]. However, in Section 4.3.3 we point out the limitations of the twice algorithm. Moreover, we believe that the header request mechanism will have problems over wireless links with high transmission error rates where many of the header requests itself will be lost. Link layer error recovery implemented below TCP/IP header compression is more effective. This is further discussed in Section 4.3.3.

For the case of TCP, the problem of competing error recovery is investigated in [DCY93]. The study concludes that at small link transmission error rates almost all packets retransmitted at the link layer are also retransmitted by the TCP sender (!). However, the presented analysis and simulation are flawed. The mathematical analysis assumes that the RTO is fixed. This is an unrealistic precondition that increases the likelihood of spurious timeouts. The simulation ignores increases in the RTT and RTO due to congestion; packets are retransmitted entirely at the link layer, i.e., link layer segmentation is ignored; the error recovery is a stop-and-wait protocol; and only 2 instead of 4 times the variation is used for calculating the RTO (see

1. This requires that the IP *and* the TCP checksum be re-computed.

Section 2.2.2). These are again unrealistic preconditions. To the contrary, link layer error recovery can yield excellent throughput for bulk data transfers without interfering with TCP's error recovery as we show in Chapter 4. Although not explicitly focusing on the problem of competing error recovery, the studies in [BBKT97], [ES98], and [Mey99] confirm our result.

2.6.2 Evaluation

In this section, we evaluate the solutions of the five categories defined in Section 2.6.1 with respect to which of the inefficient cross-layer interactions explained in Section 2.5 they solve, and which deployment concerns they raise. Thus, we pose the questions: Is there at least one proposal in the given category that ...

- solves the problem of underestimating the available bandwidth?
- minimizes end-to-end error control inefficiencies?
- equally supports real-time and reliable, e.g., TCP-based, flows?[1]
- eliminates competing error recovery should the solution be applied to a wireless link running link layer error recovery?
- prevents TCP/IP header decompressor failures?

In addition, we define two more metrics to evaluate which architectural impact a given solution has, i.e., which deployment concerns it raises. Solutions that require changes to an end-to-end protocol, or implementations thereof, rely on a large scale effort to be incorporated into the operating system software of Internet hosts and network components (see the shaded boxes in Figure 2-16). If those changes are required at *both* network end-points this becomes an immense deployment concern considering the large number of hosts - especially the millions of WWW servers - that are connected to the Internet. Confining those changes to the “mobile end-point” has been the main motivation for PEP-based approaches and is one of their key advantages. However, PEPs create another deployment concern which it that PEPs cannot interoperate with network layer encryption, e.g., IPsec[2]. The importance of this aspect is discussed in Section 2.7. Hence, we ask the questions: Is there at least one proposal in the given category that

- confines the required changes of an end-to-end protocol or its implementation to the “mobile end-point”?
- can interoperate with network layer encryption assuming that nodes that a solution might require in the network are untrusted, i.e., cannot decrypt/encrypt a flow's packets?

1. Since all solutions mentioned in the literature are TCP-specific and the interaction described in Section 2.5.3 only applies to real-time flows, we have adapted this question accordingly.
2. Unless, the PEP was a trusted party and could decrypt and later again encrypt a flow's packets.

Table 2-17 summarizes our evaluation including that of our own approach that is briefly motivated and outlined in Section 2.7. We comment the table below.

	Solves underestimation of available bandwidth?	**Minimizes end-to-end inefficiencies?**	**Equally supports real-time and reliable flows?**	**Eliminates competing error recovery?**	**Prevents TCP/IP header decompressor failures?**	**Requires changes *only* to mobile end-point?**	**Interoperates with network layer encryption?**
Pure End-to-End	***no***	***no***	***no***	***no***	***no***	***no***	yes
Hard-State Transport Layer PEP	yes	yes	***no***	yes	yes	yes	***no***
Soft-State Transport Layer PEP	yes	***no***	***no***	***no***	***no***	yes	***no***
Cross-Layer PEP	yes	***no***	***no***	***no***	***no***	yes	***no***
Pure Link Layer	yes	yes	***no***	***no***	yes	yes	yes
Our Approach (see Section 2.7): Flow-Adaptive Link + TCP-Eifel	yes	yes	yes	yes	yes	***no***	yes

Tabelle 2-17: Evaluation of related work.

TCP-SACK as one of the pure end-to-end solutions improves TCP's error recovery as it can more quickly recover from multiple lost segments in a single RTT. It also makes TCP's congestion control more robust as it treats such an event as *one* congestion signal instead of one signal for each lost segment as done in "standard" TCP. Nevertheless, TCP-SACK cannot prevent underestimation of the available bandwidth by the TCP sender, it can be inefficient over wireless links if not supported by link layer error recovery, and it cannot eliminate competing error recovery when run over a reliable wireless link. Also, an end-to-end protocol cannot make TCP/IP header compression more robust.

Hard-state transport layer PEPs have a number of advantages over soft-state transport layer or cross-layer PEPs. They can minimize end-to-end error control inefficiencies when a protocol is run over the wireless link that is optimized accordingly, e.g., [Bau97], [HK99], [KRLKA97]. Those solutions also separate the end-to-end and the link layer error recovery control loops (compare Figure 2-14), and thereby eliminate competition between both error recovery schemes. Furthermore, failures of the TCP/IP header decompressor can be eliminated by running a reliable link layer protocol link with sufficient error recovery persistency on the wireless.

The key advantage of link layer solutions in general (not only the “pure” ones) is that the local knowledge about the continuously changing error characteristics of the wireless link can be exploited to optimize error control efficiency overall, including that of end-to-end error control. We show this in Chapter 4. The major drawback of *pure* link layer error control schemes is, however, that they do not adapt to a flow’s QoS requirements but instead are “hard-wired”, i.e., cannot be changed (e.g., [Kar93]), or can only be reconfigured from one (physical) connection to the next (e.g., RLP; see Section 2.4.3). Consequently, a flow that is best served with persistent link layer error recovery cannot share the link with a real-time flow intolerable of delays introduced by link layer retransmissions. On the other hand, an application might tolerate higher loss rates in return for higher available bit rates than provided by the link’s FEC scheme. The solution proposed in [Kar93] strikes a good compromise between supporting TCP-based flows while minimizing interactions for delay-sensitive flows. It proposes semi-reliable link layer error recovery with a low error recovery persistency. However, that solution cannot in all situations prevent that the available bandwidth is underestimated, and it may cause failure of the TCP/IP header decompressor. As that solution explicitly emphasizes end-to-end error recovery it may also cause end-to-end error recovery inefficiencies. On the other hand, [GSM04.22b] and [GSM04.60] can be “tuned” from the mobile host to work well for TCP-based flows by configuring a high error recovery persistency. Although that eliminates the above mentioned problems of [Kar93], it may now cause inefficient cross-layer interactions for real-time flows as described in Section 2.5.3, and also cannot avoid competing error recovery.

2.7 Brief Motivation and Outline of our Approach

PEP-based solutions not only have the problems discussed in the preceding section, but are often regarded as being “inelegant” because they violate *the* fundamental design principle in data communications: *protocol layering.* The principle says that a service provided by a certain protocol layer may not depend on the semantics of a protocol on a different layer. For example, a service that is specific to a particular link is by definition a link layer service. According to the protocol layering principle it may not depend on network, transport, or application layer protocol semantics. Those semantics are often conveyed via the respective protocol headers. However, PEPs provide a link layer service but *are* dependent on transport layer protocol semantics and *do rely* on access to the information in the transport protocol’s headers. In Section 2.2.3, we have already encountered such an example of layer violation: TCP/IP header compression.

Our goal is to find a solution that eliminates *all* the inefficient cross-layer interactions explained in Section 2.5 *while* satisfying the paradigm of protocol layering. Beyond academic

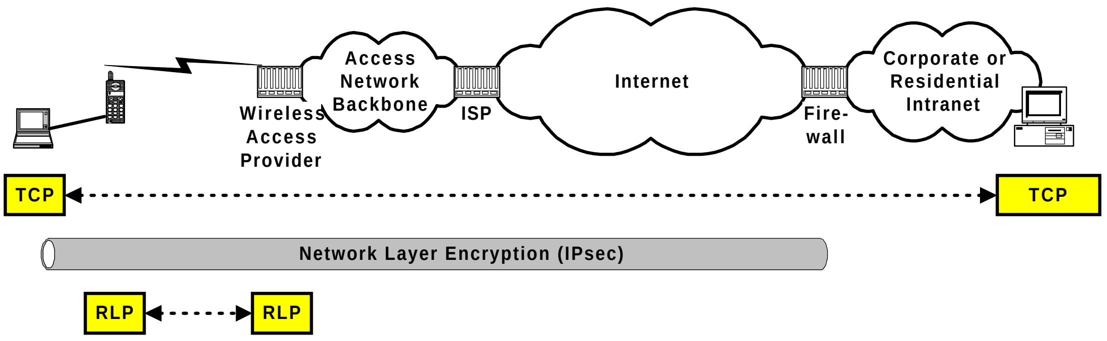


Figure 2-18: Getting secure access to an Intranet with an IPsec tunnel.

ambition there is a more pragmatic reason for why it is desirable to find a solution that conforms with protocol layering: network layer encryption for IP, *IPsec* [RFC2401]. PEP-based solutions are rendered useless if IPsec spans across an untrusted PEP, because IPsec encrypts the transport layer protocol headers. This means that a link layer cannot even identify a flow's end-points. We believe that IPsec will become an important cornerstone of the future Internet, especially for mobile users who are free to roam between multiple wireless access providers and ISPs to get access to their corporate and/or residential Intranets[1] (see Figure 2-18). Those users (usually) do not trust their wireless access providers, their ISPs, nor anybody on the "open" Internet when accessing their Intranets. IPsec provides them with the capability to secure their data against such untrusted parties by encrypting all traffic at the network layer either end-to-end, or between a host and a firewall.

In finding our solution, we were not discouraged by concerns about deploying changes to an end-to-end protocol or its implementation (see the "no" in the last row of Table 2-17) as discussed in the preceding section. Nevertheless, with our proposed solutions we obey the *Internet Engineering Task Force's (IETF)* key design principle: *incremental deployment*. The principle says that any change to the Internet, in particular to its end-to-end protocols, must be backwards compatible and may not interfere with existing mechanism, e.g., TCP's congestion control behavior. Hence, we asked ourselves the following questions.

- What function, in general, is missing in the Internet's design that would allow to optimize the end-to-end performance that loss responsive flows could provide when running across wireless links?
- Given that such a function is, in fact, missing and was added to the Internet, what, in particular, is wrong with TCP or its implementations that decreases its end-to-end performance when it runs over wireless links, and how can that be solved?

Chapter 4 is dedicated to finding answers to the first question, while we study the second question in Chapter 5.

1. An Intranet is a "closed" inter-network that is connected to the rest of the "open" Internet via firewalls that perform access control of in- and outbound flows.

CHAPTER 3

Analysis Methodology

In this chapter, we explain the methods and tools we used to obtain the results presented in Chapter 4 and Chapter 5. Our analysis approach is strongly based on measurements. This is mainly motivated by the fact that simulators for wireless networking were not sufficiently developed when we started our work. Also, with the globally deployed GSM-CSD systems we had a *real* (wide-area) wireless network available for our study, about which little was known with respect to inefficient interactions with end-to-end protocols. Hence, we used the GSM-CSD system as a case study of a wireless link. Our measurement-based approach gave us the unique opportunity to capture the aggregate of real-world effects such as noise, interference, fading, and shadowing. This is a key advantage over simulations as unrealistic assumptions about the error characteristics of a wireless channel can completely change the results of a performance analysis. This often leads to inadequate design decisions as we demonstrate in Chapter 4.

Furthermore, we believe that results obtained by measurement are often more convincing than those obtained by simulations. The reason is that it is much easier to make mistakes in simulations that lead to wrong conclusions than it is when performing measurements. Experimental measurements often expose effects that may not be visible using simulations alone. This may be due to implementation errors, or the fact that a simulator has abstracted to many details, i.e., does not implement all the relevant features of a real system. In Section 3.2.5 and Section 4.2.3 we present examples of problems that would have been difficult to detect by simulations.

Certainly, a measurement-based analysis approach has a number of drawbacks. First, it requires the availability of a real system. Measuring the performance of new features of a network that is still in the design phase is not possible. Also, the process of performing measurements is often a time intensive task while a large base of measurements is required to draw general conclusions when the error characteristics of a wireless link are crucial. Another problem for measurements is that real systems often only allow a limited parameterization.

3.1 Evaluating Error Recovery Strategies

In this section, we explain the methodology we use in Section 4.3 to evaluate the benefit of link layer error recovery for reliable flows. With this analysis we address the problems of “underestimation of available bandwidth” (see Section 2.5.1), “inefficiency of end-to-end error control” (see Section 2.5.2), and also the problem of “failure of link layer differential encodings” (see Section 2.5.5). This work has been published in [LKJ99] and [LKJK00]. In Section 3.1.1 we provide general information about how we collected measurements in GSM-CSD that applies to both this section and Section 3.2.

3.1.1 Collecting Link Layer Traces in GSM-CSD

Performance measurements involving wireless links add a complex dimension to the characteristics with which links are usually described. In addition to the simpler parameters of link bit rate and link latency, the error characteristics play a crucial role as motivated in the next subsection. The error characteristics of a wireless channel over a certain period of time can be captured by a bit error trace. A *bit error trace* contains information about whether a particular bit was transmitted correctly. The average *Bit Error Rate (BER)* is the first-order metric commonly used to describe such a trace. The same approach can be applied at the block level (see Section 2.4.4). Hence, a *block erasure trace* contains information about whether a particular data block was correctly transmitted and the *BLock Erasure Rate (BLER)* denotes the average rate at which block erasures, i.e., FEC decoding failures, occur in such a trace.

All our measurements involving a wireless link were carried out in commercially deployed GSM-CSD systems. Most of the measurements were carried out in the San Francisco Bay Area. In addition, we have collected traces at other places in the U.S. and also in Sweden and Germany. Nevertheless, apart from the effects mentioned in Section 3.2.5, we did not find any differences between the various countries, or more precisely, between the manufacturers of the GSM network components and the frequencies used for operation. It is important to point out that the error characteristics we have captured in the form of block erasure traces are only valid for the particular FEC and interleaving scheme implemented in GSM-CSD (see Section 2.4.4). Nevertheless, we believe that the results presented in Chapter 4 provide new insights into how to optimize this widely deployed system. These suggest techniques that can be used to design future wireless links, e.g., the GSM packet-switched data service which implements a similar FEC scheme [GSM05.03] and similar link layer error recovery [GSM04.60].

We are not interested in identifying physical link factors that cause measured block erasures. Rather, we are interested in the aggregate result captured by block erasure traces. This is similar to the approach of trace-based mobile network emulation as proposed in [NSNK97]. To collect block erasure traces, we ported the RLP (see Section 2.4.3) implementation of a commercially available GSM data PC-Card to the BSD/386 Version 3.0 operating system. We also developed a protocol monitor for RLP that we call `rlpdump`[1]. It logs whether a received block could be correctly reconstructed by the FEC decoder. This is possible because every RLP frame corresponds to an FEC encoded data block. Thus, a received block suffered an erasure whenever the corresponding RLP frame has a frame checksum error. In addition, `rlpdump` logs time/sequence information, i.e., which frame number was sent at which time, and also exceptional events, like selective rejects, retransmissions, flow control signals (XON/XOFF), and RLP link resets in both the send and the receive direction. For a given RLP connection such information makes up what we refer to as an *RLP trace*. Unfortunately, we were not able to log internal receiver signal strength measurements from the mobile phone to correlate them with the block erasure traces. Instead, we read the mobile phone's visual receiver signal strength indicator ranging from 1 - 5. The receiver signal strength is used in Section 3.1.3 and Section 3.2.3 to categorize measurements.

3.1.2 Analysis Goals, Assumptions, and Approach

Our goal is to evaluate the performance of the following two error recovery strategies. Without a PEP in the network, these are the only alternatives that exist for reliable data transfer over a path that includes a wireless link.

- End-to-end error recovery complemented with link layer error recovery running over the wireless link.
- Pure end-to-end error recovery.

In Section 2.6.1, “pure end-to-end” implied that no assumptions are made about the existence of dedicated support from the link layer, e.g., error recovery. Nevertheless, throughout the rest of this dissertation, when we use the term “pure end-to-end error recovery” we imply that the wireless link is *not* protected by link layer error recovery.

In Section 4.3, we perform the evaluation of the two error recovery strategies through a case study of the GSM-CSD wireless link. We first investigate the impact of changing the (fixed) RLP frame size on application layer throughput and the consumption of radio resources (e.g., spectrum and transmission power). We then quantify the benefits of link layer error recovery by comparing it against the performance of pure end-to-end error recovery. There we show that at least on some wireless links, e.g., a GSM-CSD link, the end-to-end performance that a reliable flow can provide can only by optimized by complementing end-to-end with link layer error recovery.

1. `rlpdump` was implemented by Bela Rathonyi at Ericsson Mobile Communications AB, Sweden. Keith Sklower at U.C. Berkeley assisted in porting the RLP code to the BSD system.

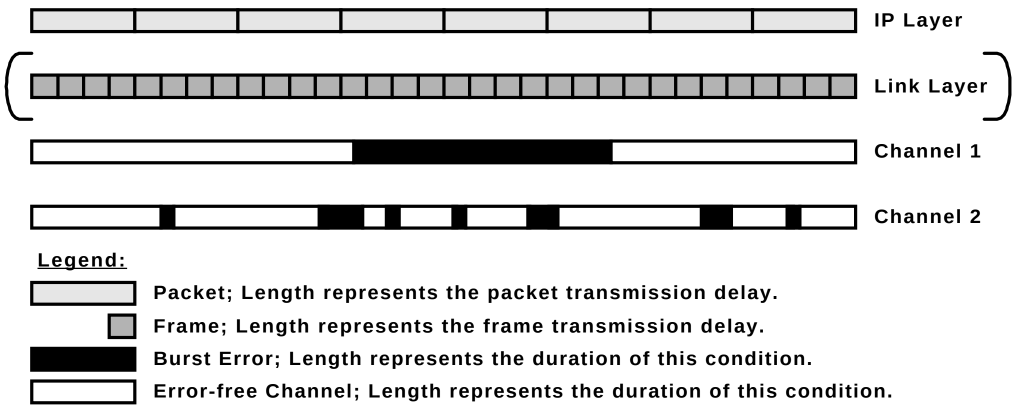

IP Layer
Link Layer
Channel 1
Channel 2
Legend:
Packet; Length represents the packet transmission delay.
Frame; Length represents the frame transmission delay.
Burst Error; Length represents the duration of this condition.
Error-free Channel; Length represents the duration of this condition.

Figure 3-1: Two different channel error characteristics.

The performance difference between the two protocol design alternatives depends on the wireless channel's time varying error characteristics versus the channel's packet transmission delay. This is sketched in Figure 3-1, where "burst error" denotes time intervals during which data in transit is corrupted to the extent that it cannot be recovered at the receiving link layer (FEC decoder). With respect to GSM-CSD, a burst error corresponds to a series of back-to-back block erasures where the channel is error-free before and after that series. A wireless channel's error characteristic can be described by the length of burst errors and their correlation expressing the degree of clustering. Link layer error recovery is less effective on wireless links where the length of burst errors is large compared to the packet transmission delay (see "Channel 1" in Figure 3-1). In this case, pure end-to-end error recovery often yields higher throughput results by saving link layer protocol overhead. Another case is sketched with "Channel 2" in Figure 3-1 where the length of burst errors is small compared to the packet transmission delay and the burst errors often occur isolated. In this case, the link layer overhead is likely to be amortized when the "right" frame size is chosen. Studying this trade-off requires a realistic error characterization of the wireless channel which motivates our measurement-based analysis approach.

The key premise for our analysis is a model of a bulk data flow based on a fully-reliable protocol such as TCP. As pointed out in Section 2.1, the main QoS requirement of bulk data flows is to maximize throughput. Fully-reliable flows have the additional QoS requirement that the transfer *must* be reliable, i.e., the transfer fails if the data is corrupted or incomplete when received by the destination. To compare throughput among the two error recovery strategies, we assume that the GSM-CSD wireless link is the path's bottleneck link, and that the bulk data flow is the only flow that utilizes the bottleneck link. Using the ReTracer tool explained in Section 3.1.4, we perform a best-case analysis on the basis of block erasure traces we had collected a priori as described in Section 3.1.3. The best-case analysis assumes that the bulk data transfer always fully utilizes the wireless bottleneck link, i.e., utilizes the link 100 percent.

We redefine the term *utilization* for our purposes as follows.

- Given a time period of length *T*, the utilization of a link is the fraction of *T* during which *useful* data, i.e., excluding packets/frames which had already been successfully transmitted[1], is transmitted over the link, divided by *T*.

For link layer error recovery, the best-case analysis implies (1) the use of a selective reject based protocol, like RLP; and (2) an “infinite” error recovery persistency[2]. It also requires the use of large enough windows to allow the link layer sender to always fully utilize the link. This avoids the *stalled window* condition, where the sender must interrupt transmission due to flow control, i.e., when the receive buffer of the link layer receiver is exhausted to buffer additional frames. For a bulk data flow that implements congestion control similar to TCP, the best-case analysis implies that the flow’s maximum load must exceeds two times (or more) the flow’s pipe capacity as explained in Section 2.5.1.

The best-case assumption ignores inefficient interactions with end-to-end congestion control schemes that may lead to an underestimation of the available bandwidth. For TCP over RLP in GSM-CSD, this is valid as we show in Section 4.2. For pure end-to-end error recovery, however, this is often not the case as discussed in Section 2.5.1. Nevertheless, a best-case study indicates the theoretical maximum application layer throughput that pure end-to-end error recovery could provide. Moreover, the application layer throughput that we determine in Section 4.3 under the given assumptions, directly translates into radio resource consumption. For example, if transport layer sender A only achieves half the throughput that sender B achieves, it is using twice as much radio resources, i.e., it needs to transmit twice as many data blocks. This may happen if sender A has to rely on pure end-to-end error recovery, and has to retransmit packets of which only a small fraction of the corresponding original transmission was corrupted on the unreliable wireless link[3].

3.1.3 Measurement Platform

Our measurement platform is depicted in Figure 3-2 (simplified from Figure 2-7). A single-hop path connects the mobile to a fixed host which terminates the GSM-CSD connection. As explained in the preceding subsection, we were only interested in capturing block erasure traces, not in studying protocol interactions. We therefore used `ping` described in [Ste94] as a traffic generation tool because the underlying end-to-end protocol (ICMP) is unresponsive to

1. This can, e.g., happen in TCP which exhibits go-back-N behavior after spurious timeouts as explained in Section 5.1.
2. Throughout our measurements the highest number of retransmissions for a single RLP frame was 12. Thus, in GSM-CSD an “infinite” error recovery persistency (the RLP parameter N2) can be approximated with a maximum number of retransmissions of 12 + n for some small value of n.
3. E.g., if only a single byte of a 1500 bytes packet gets corrupted during transmission over an unreliable link, then still the entire packet has to be retransmitted.

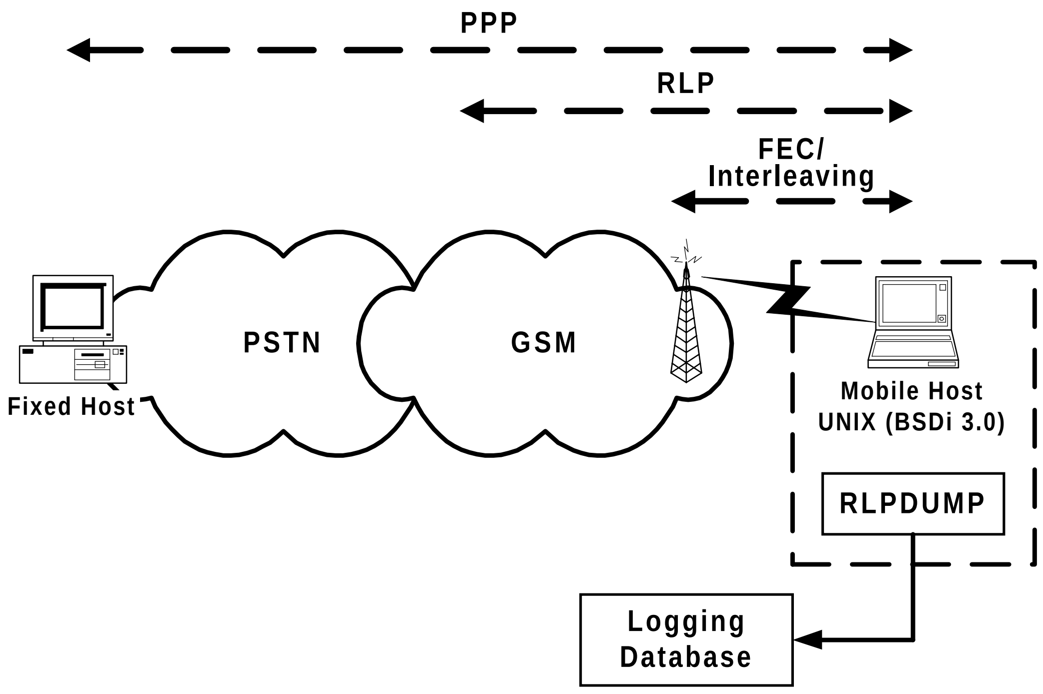


Figure 3-2: The measurement platform.

packet losses. The `ping` sender sends an ICMP packet *every* second that is echoed by the `ping` receiver. Thus, when the ICMP packet size is configured so that the corresponding packet transmission delay exceeds one second, `ping` can be used as an infinite and uninterrupted traffic source[1].

We then generated continuous traffic with `ping` and used `rlpdump` to capture the corresponding block erasure traces. That way we have collected block erasure traces for over 500 minutes of “air-time” and distinguish between measurements where the host was stationary versus mobile when driving in a car. All stationary measurements were taken in the exact same location. We categorized the measurements as follows.

A. Stationary in an area with good receiver signal strength (3 - 4): 258 minutes.

B. Stationary in an area with poor receiver signal strength (1 - 2): 215 minutes.

C. Mobile in an area with mediocre receiver signal strength (2 - 4): 44 minutes.

3.1.4 The ReTracer Tool

Clearly, the size of an RLP frame does not need to match the size of an unencoded data block. Any multiple of the size of an unencoded data block would have been a valid design choice. In

1. This causes the sending host’s outbound interface buffer to constantly overflow leading to many dropped `ping` packets, but that did not matter in this case.

fact, a multiple of 2 has been chosen for the new version of RLP [GSM04.22b] in the next generation of GSM-CSD, which also uses a weaker FEC scheme [GSM04.21]. Larger frames introduce less relative overhead per frame, but an entire frame has to be retransmitted even if only a single data block incurs an erasure. Applying our technique of *retrace analysis*, we study this trade-off based on the block erasure traces we had collected a priori in environments *A* - *C* (see above). For that purpose we developed a tool we call *ReTracer*[1] that automatically performs the retrace analysis. Based on a given block erasure trace and a given bulk data transfer size, ReTracer reverse-engineers the value of target metrics (e.g., channel throughput or number of retransmissions). It emulates RLP while assuming a particular fixed frame size and fixed per frame overhead. We then iterate the retrace analysis over a range of RLP frame sizes, defined as multiples of the data block size. We can thereby find the frame size that maximizes the bulk data throughput for a particular block erasure trace.

We use different block erasure traces for our analysis. *trace_A* is a concatenation of all block erasure traces we collected in environment *A*. Likewise, *trace_B* and *trace_C* are concatenations of all block erasure traces we collected in environment *B* and *C*, respectively. We then choose an appropriate bulk data size to cover the entire trace (e.g., for *trace_B* a size corresponding to a transmission time of 215 min was chosen). Once the retrace analysis reaches the end of a trace, it wraps around to its beginning. In addition, we investigate the impact of error burstiness, i.e., the extent to which the distribution of block erasures within a trace influences our results. For that purpose, we artificially generated three more “non-bursty” block erasure traces, *trace_A_even*, *trace_B_even* and *trace_C_even*. These have the same BLER as the corresponding real traces, but with an even block erasure distribution, i.e., those traces have single and isolated block erasures with a constant distance from each other.

3.2 Detecting Inefficient Cross-Layer Interactions

In this section, we explain the methodology we use in Section 4.2 to study in general the inefficient cross-layer interactions that may occur when running TCP-based bulk data transfers over RLP in GSM-CSD. This work has been published in [LRKOJ99]. Also, in Section 3.2.1 we explain how to interpret TCP trace plots. In Chapter 4 and Chapter 5 we often use TCP trace plots to illustrate certain effects, problems, or solutions.

1. ReTracer was implemented by Almudena Konrad at U.C. Berkeley.

3.2.1 How to Read TCP Trace Plots

A *trace* is a series of events that was measured over time for a particular connection of a given protocol layer at the sender (called a *sender trace*), the receiver (called a *receiver trace*), or a node in the connection's path. A *trace plot* is a graphical representation of a trace. Trace plots 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. We mostly deal with TCP traces, but in some cases correlate them with RLP traces that we captured using `rlpdump` as described in Section 3.2.4. A TCP trace captures the times (timestamps) when a segment or an ACK is transmitted or received. In a trace this is represented by the tuple <timestamp, sequence number> or <timestamp, ACK number>, respectively (see Section 2.2.1 for the definition of sequence number and ACK number). In the plots we label the graphs comprising points corresponding to

- segments sent by the TCP sender as *Snd_Data* (or *TcpSnd_Data*),
- ACKs received by the TCP sender as *Snd_Ack* (or *TcpSnd_Ack*),
- segments received by the TCP receiver as *Rcv_Data* (or *TcpRcv_Data*), and
- ACKs sent by the TCP receiver as *Rcv_Ack* (or *TcpRcv_Ack*).

To avoid that the sender and receiver plots overlap when shown in the same plot we offset the sequence number space of the TCP receiver trace by 10,000 bytes. In our measurements, the clocks of the sending and the receiving host were *not* synchronized. 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 connect request (SYN) and when it arrives at the receiver. Thus, apart from clock drifts on both hosts, the receiver trace is offset by the one-way delay of the initial SYN.

Capturing TCP traces requires an extension of the operating system kernel that does the logging of relevant information, and a user level process to control the kernel extension and to transfer the logged data into user space. For the BSD system these two functions had already been developed[1]: the *BSD Packet Filter (BPF)* [MJ93], and `tcpdump` [JLM], respectively. The output generated by `tcpdump` can then be reformatted (we used our own scripts for that purpose) according to the input format required by standard plotting tools such as Xgraph [Xg].

A number of characteristics can be read off TCP trace plots. As an example, Figure 3-3 shows a section of a TCP sender trace. As each point in the Snd_Data graph corresponds to a

1. Those tools are publicly available and have been extensively used, tested, and enhanced by the Internet research community.

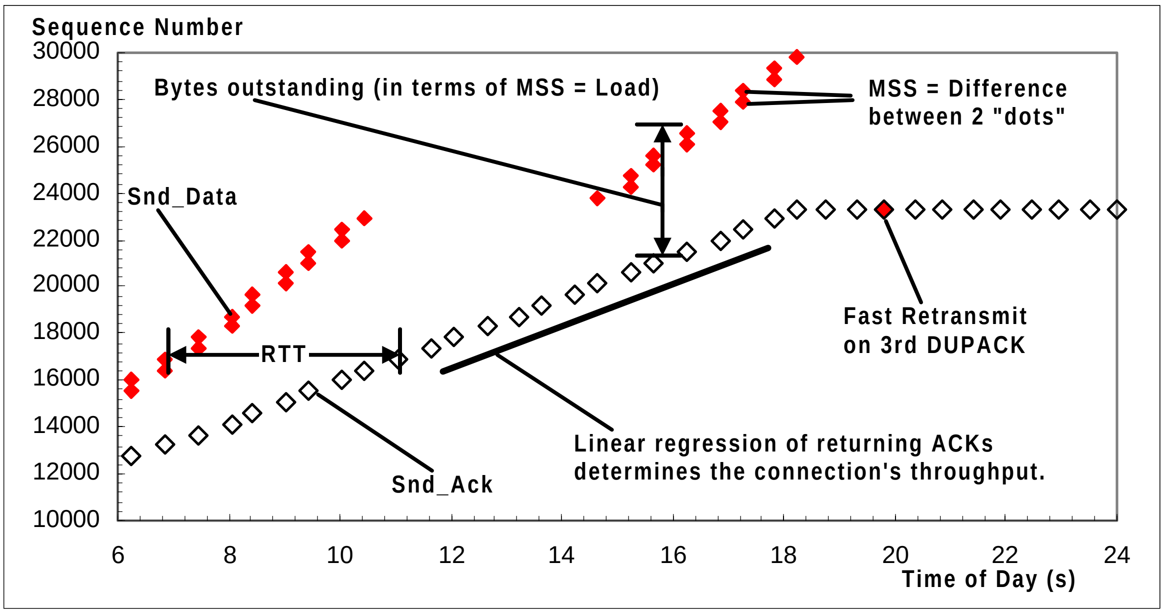


Figure 3-3: A TCP sender trace plot.

sequence number, the difference between two succeeding segments is the segment size of the first segment. During bulk data transfer it is usually the connection's Maximum Segment Size (MSS). Also the flow's load (usually the number of bytes outstanding divided by the MSS) and the flow's RTT can be read off the plot as indicated in Figure 3-3. The rate at which ACKs return to the sender can be determined by linear regression of the Snd_Ack trace. It corresponds to the flow's available bandwidth at that time due to the self-clocking property of TCP (see Section 2.2.1). The ACK clock itself can be seen from the fact that no segment is sent between arrivals of ACKs, i.e., each ACK clocks out one or more segments.

Figure 3-3 shows a special case. The TCP connection has just been established and the sender is in the slow start phase where every ACK clocks out two segments[1]. One because the ACK advanced the window and another one because the congestion window was increased by one. 10.5 s into the connection the sending host's interface buffer overflows and one packet is dropped. In response the *tcp_quench()* function [WS95] resets the TCP sender's congestion window to $1 \times MSS$. The following eight ACKs grow the congestion window back to $17 \times MSS$ allowing that another segment is sent at about 15 s into the connection. Shortly after the 18th second the DUPACKs for the dropped segment return to the sender. The third DUPACK triggers a fast retransmit in the 20th second.

Figure 3-4 shows an example where both the sender and the receiver traces are correlated in the same plot. This measurement was collected in the simple network shown in Figure 3-8 that we explain in Section 3.3.2. The plot shows the typical sender and receiver traces of a network-

1. In this case, the TCP receiver acknowledges the receipt of *every* segment because the ACK interarrival time is larger then the delayed-ACK timer of 200 ms used in TCP-Lite.

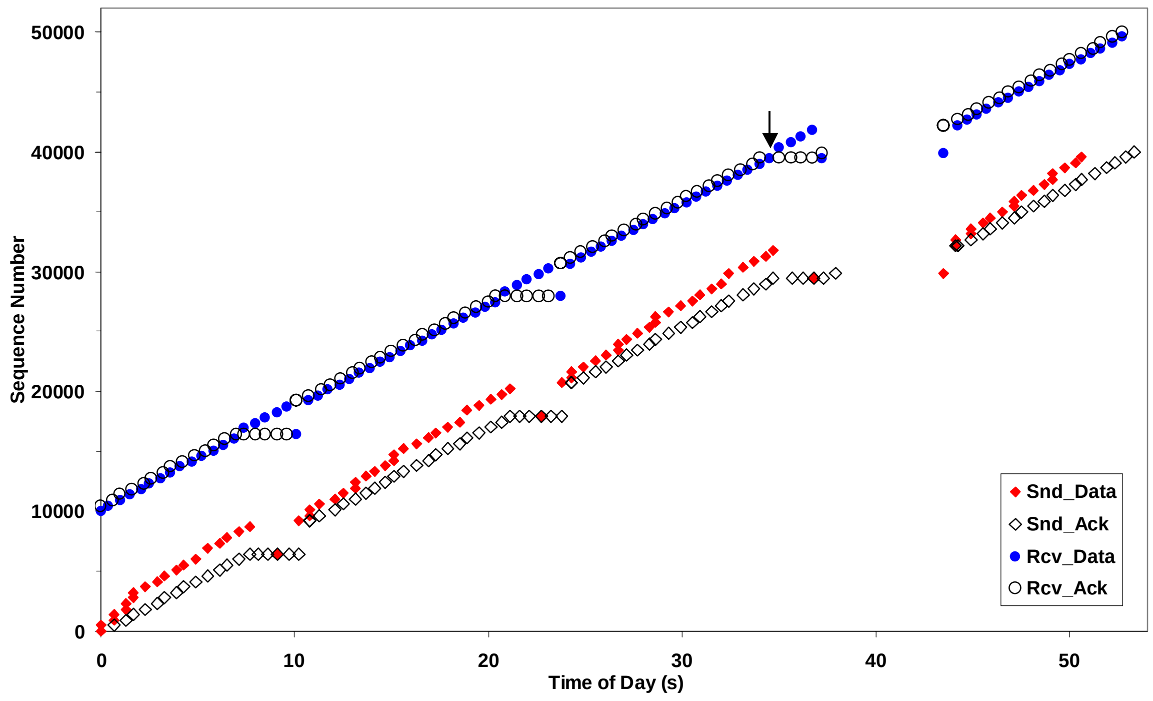


Figure 3-4: A TCP sender and receiver trace plot.

limited TCP connection. The sender periodically: grows its load linearly during the congestion avoidance phase, drops a single packet, goes into fast recovery (triggered after a fast retransmit), and then goes into congestion avoidance again. This plot should be compared with Figure 2-6 which is an alternative representation of a network-limited TCP connection. In the 35th second a relatively rare event happens that we captured by chance. A segment with a checksum error arrives at the TCP receiver[1]. This can be seen from the fact that the TCP receiver does not provide any feedback, i.e., neither sends an ACK nor a DUPACK, upon its arrival (see arrow in the plot). The lost segment triggers the fast retransmit 37.5 s into the connection. The segment following the one that was received in error was dropped due to congestion. However, because at that time not enough packets are in flight to generate three DUPACKs, that segment has to be recovered by a timeout that occurs in the 43th second.

3.2.2 Analysis Goals, Assumptions, and Approach

The main focus of our analysis is to study inefficient cross-layer interactions that may occur when running TCP-based bulk data transfers over RLP in GSM-CSD. The results of this study are described in Section 4.2. We were only interested in “stable” connections that lasted long enough to allow for all TCP sender state variables (e.g., retransmission timer, slow-start threshold, etc.) to converge from their initialization values to a stable range of operation. We therefore performed a series of large bulk data transfers ranging in size from 230 KBytes to

1. Apparently this error had not been detected by PPP’s error detection function.

1.5 MBytes. In Section 4.2.1 we report on the throughput that TCP achieved in those measurements. However, throughput itself is not sufficient information to determine whether TCP and RLP interacted in an inefficient way. For example, a throughput of one half of the theoretical maximum could either mean that the radio conditions were so poor that RLP had to retransmit every other frame or it could indicate competing error recovery between TCP and RLP.

Utilization as defined in Section 3.1.2 is the key performance metric that can be used to determine whether a data transfer suffered from inefficient TCP/RLP interactions or not. If the TCP sender fully utilizes the bandwidth provided by RLP (which may vary over time due to RLP retransmissions) then this indicates optimal performance and rules out inefficient interactions between the two protocols. There are only two ways that utilization may not be optimal: (1) the TCP sender leaves the link (RLP) idle, or (2) the TCP sender sends spurious retransmissions. We used the MultiTracer tool explained in Section 3.2.4 to check for these two cases in all the measurements we had collected a priori as described in Section 3.2.3. That way we isolated the traces where utilization was 95 percent or less, and further investigated those to identify the causes of the degraded performance.

Note that utilization can never be exactly 100 percent because of TCP's initial slow-start phase and the 3-way handshake required for both the TCP connection establishment and the disconnection phase [Ste94]. In our measurement platform, however, the effect of slow-start is negligible because the pipe capacity of a TCP flow over a GSM-CSD link is already reached with 2 - 3 segments, even when using a small MSS. Also, these effects are amortized when performing large bulk data transfers (as done here). Measuring utilization has the added advantage that it is independent of protocol overhead. Thus, parameters like the Maximum Transmission Unit (MTU) configured for PPP, the PPP framing overhead, and whether TCP/IP header compression was used or not, do not affect utilization as defined here.

3.2.3 Measurement Platform

The platform that we developed for measurement collection is depicted in Figure 3-5. The gray shaded area indicates a possible extension to the setup that we have not implemented. It would generate input for trace replay in a simulation environment allowing to reproduce various effects that were measured in reality. The measurement platform extends the setup shown in Figure 3-2 by the capability to capture TCP traces with `tcpdump` in addition to capturing RLP traces with `rlpdump`, and to correlate all traces onto the same time axis. Since we wanted to isolate the TCP/RLP interactions we continued using a single-hop path. Although it might in some cases be reverse-engineered, `tcpdump` does not provide information about the TCP sender state variables, such as the congestion window, the slow start threshold, and the retransmission timeout value. We therefore used `tcpstats` [Pad98], a UNIX kernel instrumentation tool that traces these TCP sender state variables. As for the measurements described

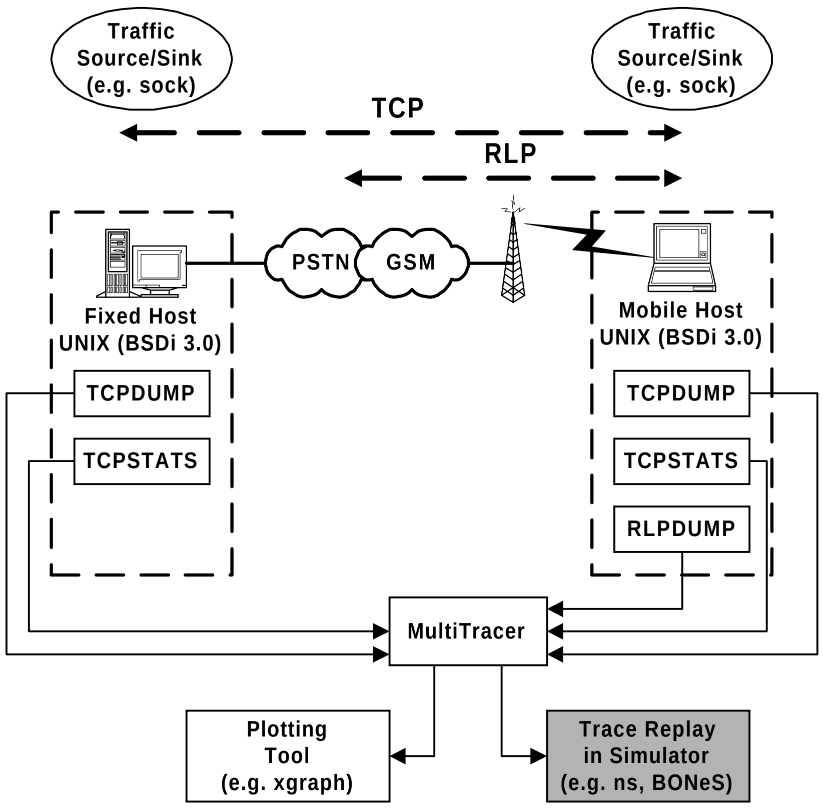

Traffic Source/Sink (e.g. sock)
Traffic Source/Sink (e.g. sock)
TCP
RLP
PSTN
GSM
Fixed Host UNIX (BSDi 3.0)
Mobile Host UNIX (BSDi 3.0)
TCPDUMP
TCPSTATS
TCPDUMP
TCPSTATS
RLPDUMP
MultiTracer
Plotting Tool (e.g. xgraph)
Trace Replay in Simulator (e.g. ns, BONeS)

Figure 3-5: Measurement platform and tools.

in Section 3.1.3, we needed a traffic generation tool for bulk data transfers. Only this time we needed one that was based on TCP. For that purpose we used the `sock` tool described in [Ste94].

Overall, we captured six hours of traces that we used for our analysis. Four hours were measured in environments with good and two hours in environments with poor receiver signal strength. Although in most of our measurements the mobile host was stationary, we also measured while walking (indoor and outdoor) or driving in a car. We categorized the measurements as follows.

D. Environments with good receiver signal strength (3 - 4): 4 hours.

E. Environments with poor receiver signal strength (1 - 2): 2 hours.

It is important to point out that, as reported in [KRLKA97], we also had situations where the GSM call, i.e., the physical connection, was dropped during a measurement. In almost all cases, this happened when the receiver signal was very low. Apparently, radio coverage was insufficient in those environments. As this data would have introduced an unrealistic bias into our analysis, we excluded those traces from the analysis in Section 4.2.

3.2.4 The MultiTracer Tool

Altogether `tcpdump`, `tcpstats` and `rlpdump` generate a total of up to 300 bytes/s of trace data for a GSM-CSD connection that is running at about 10 Kbit/s. It was therefore essential to develop a post-processing tool that enabled the rapid correlation and representation of collected trace data in a comprehensive graphical manner for trace analysis. We call this tool *MultiTracer*[1]. MultiTracer is a set of script files that converts the trace data into the input format required by a plotting tool such as Xgraph. MultiTracer also automatically determines the utilization of each measurement indicating whether a data transfer suffered from inefficient TCP/RLP interactions as explained in Section 3.2.2. For that purpose MultiTracer inspects the RLP trace to determine idle phases at the RLP sender, and it inspects the TCP traces for spurious retransmissions.

In addition to the labeling scheme described in Section 3.2.1 we label the graphs comprising points corresponding to

- the congestion window at the TCP sender as *TcpSnd_cwnd*,
- frames sent by the RLP sender for the first time as *RlpSnd_Data*,
- retransmitted frames sent by the RLP sender as *RlpSnd_Xmit*,
- flow control signals (XON/XOFF) sent by the RLP receiver as *RlpRcv_XON* and *RlpRcv_XOFF*,
- RLP link resets as *RlpSnd_Rst*.

MultiTracer generates more information (e.g., RTT, SRTT, RTO), but for this analysis we only use the items listed above. To correlate RLP and TCP traces, MultiTracer uses the TCP sequence number space. Note, however, that in all plots the *RlpSnd_**, *TcpSnd_**, and *TcpRcv_** graphs are offset by 10,000 bytes from each other in the plots so that the graphs do not overlap. All traces are loosely synchronized with respect to the TCP connect request as described in Section 3.2.1.

To demonstrate the capability to correlate and visualize multi-layer traces we show a typical measurement in Figure 3-6. The three rectangles in Figure 3-6 indicate sections of this plot that are “zoomed in” for detailed analysis in the following subsection and in Section 4.2. This particular measurement yielded optimal throughput performance. This can be seen from the fact that the TCP receiver continuously receives data. Linear regression of the *RlpSnd_Data* graph shows that throughput provided by RLP is almost 960 bytes/s which is equivalent to a bit rate of 9.6 Kbit/s asynchronous. Likewise, the trendline through *TcpRcv_Data* yields a throughput of 848 bytes/s. This is what we expected as TCP/IP header compression was not used for this

1. MultiTracer was implemented by Almudena Konrad at U.C. Berkeley.

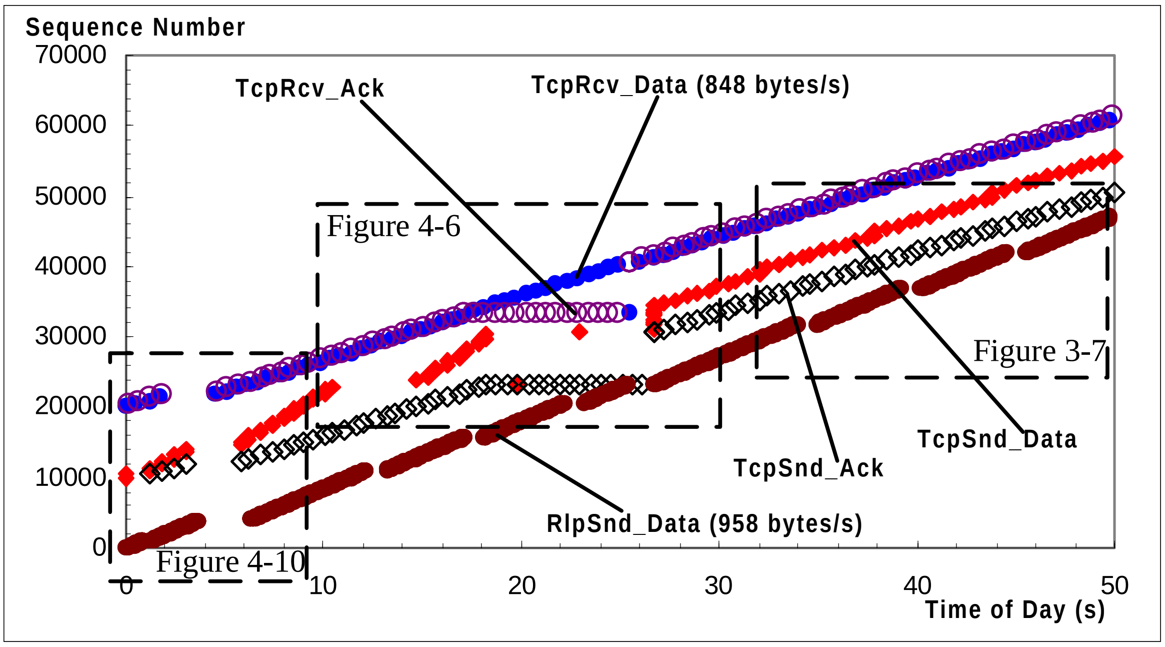


Figure 3-6: A typical multi-layer trace plot.

measurement and the overhead per MSS of 460 bytes was 59 bytes (40 bytes for the IP and TCP headers, 12 bytes timestamp option and 7 bytes PPP overhead). Thus, the TCP sender optimally utilized the bandwidth provided by RLP. Note that the graph for *RlpSnd_data* always has a larger slope than the TCP graphs because it includes the TCP, IP, and PPP overhead.

3.2.5 Detected "Implementation Bugs" in GSM

The maximum data rate provided by RLP is 1200 bytes/s. We were therefore surprised when we saw the gaps in the *RlpSnd_Data* graphs in some of our traces. However, after we traced the flow control messages at the L2R protocol (see Section 2.4.1) it became clear what was occurring. Due to limitations in some commercial GSM networks, the data rate appears to be limited to only 960 bytes/s (9.6 Kbit/s asynchronous).

In these networks, the RLP sender is flow controlled from the remote side so that the average data rate becomes 960 bytes/s. Figure 3-7 shows that the RLP sender sends at the maximum rate of almost 1200 bytes/s at times when it is not flow controlled, but the linear regression line shows that the real throughput is throttled by 20 percent down to about 960 bytes/s. However, as can be seen from Figure 3-6, the periodic gaps of 950 - 1300 ms did not trigger spurious timeouts in TCP.

As mentioned in Section 2.4.3, RLP can be implemented to provide fully-reliable service. In that case the data call is completely dropped when the error recovery persistency is reached. We have measured this effect several times in some commercial GSM networks. Simply drop-

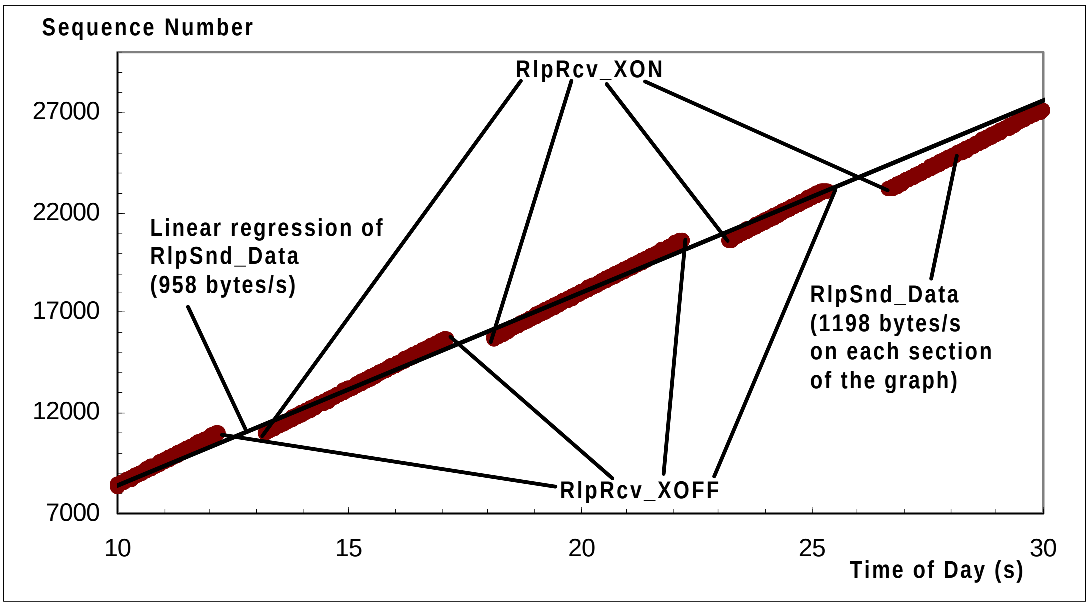


Figure 3-7: L2R flow control (zoom of Figure 3-6).

ping the call is, however, an unacceptable alternative. Not only will the user in many cases have to re-initiate the data transfer (e.g., a file transfer), but will also be charged for air time that yielded an unsuccessful transmission. Implementing RLP to provide semi-reliable service is therefore more “user friendly”.

3.3 Reproducing Inefficient Cross-Layer Interactions

In this section, we explain the methodology we use in Section 5.1 and Section 5.2 to study and solve the problem of competing error recovery for the case of TCP. With this analysis we particularly address the problem of “competing error recovery” (see Section 2.5.4). This work has been published in [LK00].

3.3.1 Analysis Goals, Assumptions, and Approach

The goal of our analysis is to study the impact of competing error recovery on TCP’s operation. In TCP, competing error recovery can cause spurious retransmissions as explained in Section 2.5.4. Those can be triggered by spurious timeouts or packet re-ordering events. In the latter case, we speak of spurious fast retransmits to distinguish those from spurious retransmissions that have been triggered by spurious timeouts.

Spurious timeouts have not generally been a concern in the past. They are rare over all wireline paths [Pax97d], as well as on path’s that include reliable wireless links that do not lose connectivity as we show in Section 4.2. This is due to the fact that the retransmission timer implemented in TCP-Lite is overly conservative as we show in Section 5.3. 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[1]. 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.

Spurious fast retransmits can occur if a link layer implements the out-of-order delivery function, or if packets of the same flow are routed differently in the Internet. It is difficult to evaluate how serious this problem is in the Internet today. For wireless links that implement the out-of-order delivery function, we are not aware of any study that investigates this problem. For the other case, some studies [Pax97d] conclude that spurious fast retransmits occur rarely, while other studies [BPS99] 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 (e.g., for fault tolerance) and load balancing is performed across those links/paths on the aggregate traffic, packet re-orderings will occur more frequently.

To study competing error recovery in TCP, we setup a “clean” environment in which measurements are not blurred by uncontrolled effects like delay variations, or packet losses commonly found in the Internet. We then used the `hiccup` tool explained in Section 3.3.3 to artificially introduce excessive packet delays and/or packet re-orderings to trigger spurious retransmissions.

3.3.2 Measurement Platform

We used a single-hop, all wireline path for our experiments consisting of two hosts (BSD/386 Version 3.0) inter-connected via a direct cable connection running PPP at 9.6 Kbit/s with an MTU of 512 bytes. In all measurements the TCP timestamp option was enabled. The TCP receiver advertised a window of 8496 bytes $(18 \times MSS)$. We always measured a single connection at a time, and the pipe capacity was two segments. The size of the sending host’s interface buffer (IFQ_MAXLEN [WS95]) which in BSD-derived systems is maintained in terms of IP packets was used to limit the number of queued packets. For example, an interface buffer size of 12 allows 12 packets to be queued before a packet (tail-)drop occurs. We used the inter-

1. Note that GSM-CSD is *not* a packet-radio network.

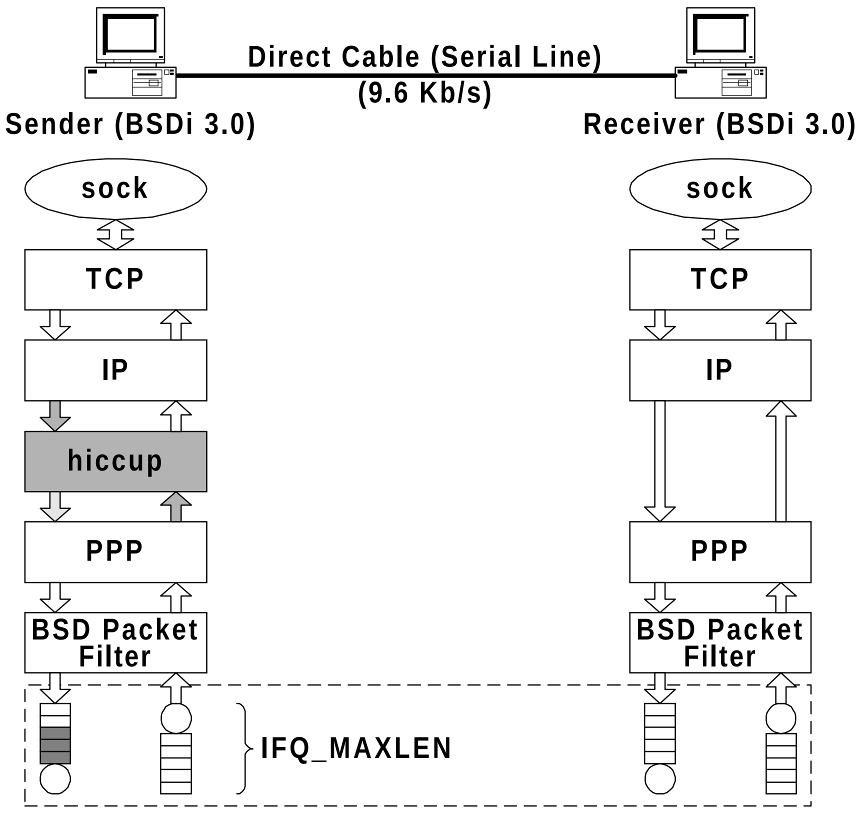


Figure 3-8: Measurement Setup.

face buffer size to trigger certain effects explained in Section 5.1. We used `sock` for bulk data traffic generation.

3.3.3 The Hiccup Tool

We developed a tool called `hiccup`[1] 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 according to a user-specified re-ordering length. When generating transient delays, `hiccup` can additionally be provided with an “expiration time” after which each packet is dropped from the queue after its arrival[2]. The default “expiration time” is indefinite. In that case packets are never dropped by `hiccup`. We have used this feature to demonstrate in Section 4.1.4 the problems that less persistent link layer error recovery may cause for fully-reliable end-to-end protocols such as TCP. Effectively, `hiccup` emulates a semi-reliable link layer protocol with a configurable error recovery persistency, and with the in-order or the out-of-order delivery function.

1. `hiccup` was implemented by Keith Sklower at U.C. Berkeley.
2. Note, that this feature results in a drop-from-front as opposed to a tail-drop queue management scheme.

The location of `hiccup` in the protocol stack is important to understand the trace plots (e.g., see Figure 5-2) in Section 5.1 and Section 5.2. Outbound packets queued by `hiccup` are logged as a single burst by the BSD Packet Filter (BPF) 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 BPF. 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.

3.4 Analyzing TCP's Retransmission Timer

In this section, we explain the methodology we use in Section 5.3 to study TCP-Lite's retransmission timer, the Lite-Xmit-Timer (see Section 2.2.2). We also use that methodology to develop a new retransmission timer for TCP in Section 5.4. With this analysis we address the problem of "competing error recovery" (see Section 2.5.4). This work has been described in [LS99].

Our experience with measuring TCP, especially the large amount of delay variation that is required to trigger a spurious timeout in TCP (see Section 4.2 and Section 5.1), led us to believe that something was wrong with the Lite-Xmit-Timer. We suspected that it was overly conservative. We therefore analyzed the Lite-Xmit-Timer and confirmed our conjecture. For that purpose, we developed a model of the class of network-limited TCP bulk data transfers in steady state which we describe in Section 3.4.1 and Section 3.4.2. In Section 3.4.3, we describe the measurement setup that was used for validation purposes.

3.4.1 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 [Pax97a]. Consequently, *the* typical TCP connection does not exist. This makes it particularly difficult to validate the design of 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.

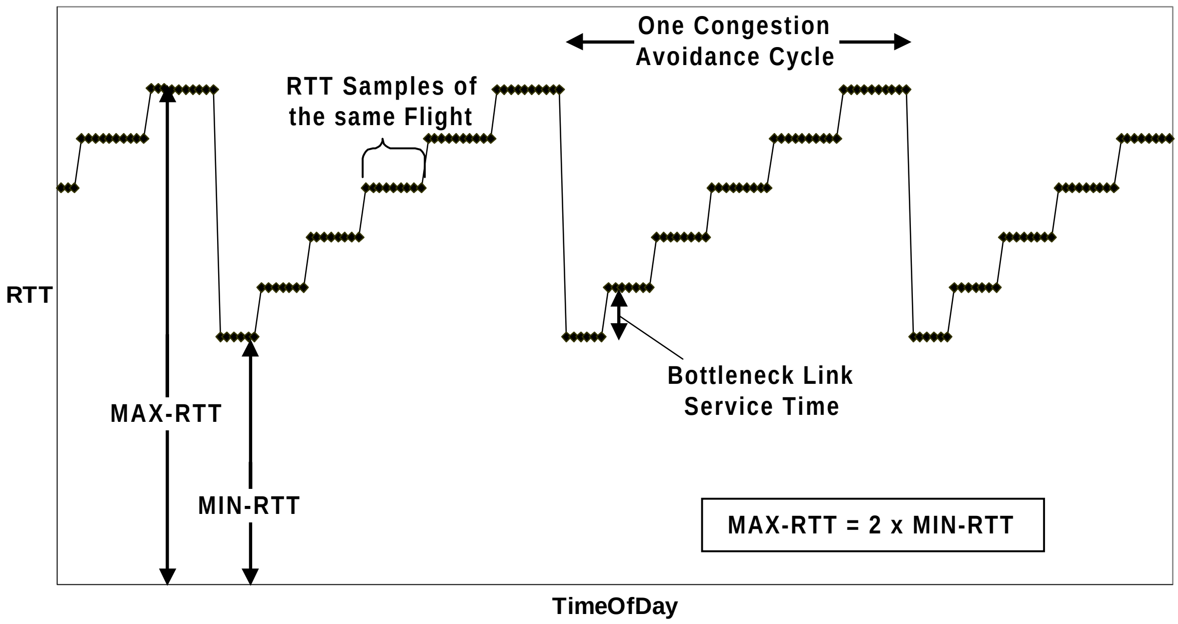

One Congestion Avoidance Cycle
RTT Samples of the same Flight
RTT
MAX-RTT
MIN-RTT
Bottleneck Link Service Time
MAX-RTT = 2 x MIN-RTT
TimeOfDay

Figure 3-9: The RTT in steady state.

We study the class of network-limited TCP bulk data transfers in steady state. In this case the TCP 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 the sender always sends fixed size segments. In addition, we assume that the sender fully utilizes (as defined in Section 3.1.2) the bottleneck link at any point in time. The latter has the effect that whenever the sender increases its 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 3-9. Given these 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 3-9 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.

3.4.2 Model-based Analysis

Given an RTT that evolves in a deterministic and recurrent manner as outlined in Section 3.4.1, 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 spreadsheet [Lud99a].

We make the following additional assumptions:

- In our model, we assume that every segment is timed to measure the RTT and that the receiver acknowledges every segment, i.e., we assume an RTT sampling rate of one.
- We assume that congestion is signalled explicitly at the end of each congestion avoidance cycle instead of through a dropped packet (see Section 2.3). This simplifies the model-based analysis without limiting it.
- To make our model independent of the impact of the timer granularity (see Section 2.2.2) we model time in terms of ticks which can be arbitrarily defined.

On our spreadsheet, 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 spreadsheet 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 spreadsheet to reflect a specific connection, i.e., a specific evolution of RTT. In the following we refer to such an instantiation of the spreadsheet as “the model”. The mentioned parameters itself are less important for our analysis. What matters is the flow’s load at the end of each congestion avoidance cycle. This is further discussed in Chapter 5.

Using spreadsheet software as a modeling tool for our purpose has a number of advantages. First, spreadsheet software usually includes graphing components which greatly ease the analysis. Second, debugging is implicitly supported as the spreadsheet itself reflects the history of the sender-side connection state, i.e., the value of the modeled state variables over time. The greatest advantage over techniques like simulations, however, is the little processing time required to determine target metrics over time for a given parameter set. Once the parameters have been specified, a graph of interest can be viewed instantly.

3.4.3 Measurement-based Analysis

We perform a measurement-based analysis to validate our model-based analysis. Thus, 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 the preceding two subsections. For that purpose, we used the measurement setup described in Abschnitt 3.3.2 without `hiccup`, an MTU of 1500 bytes, and a link speed of 2.4 Kbit/s. In addition, we set the size of the interface buffer to 40 packets. We chose those settings to produce RTTs that are several multiples of the timer granularity used in TCP-Lite (500 ms) to study the RTO at a sufficient resolution. With these settings, the RTT at the end of a congestion avoidance cycle is about 250 s (40 packets of 1500 bytes draining from the interface buffer at 240 bytes/s).

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 DUPACKs as the congestion signal. The minor impact of this difference is discussed in Abschnitt 5.4.2.

3.5 Summary

In this chapter, we explained the methods and tools we used to obtain the results presented in Chapter 4 and Chapter 5. Most of our studies are based on measurements. To support this work, we have developed four new tools:

- `rlpdump`,
- `hiccup`,
- ReTracer, and
- MultiTracer.

Many thanks to Almudena Konrad, Bela Rathonyi, and Keith Sklower who contributed to the tools' development.

The first two tools are kernel extensions of the BSD system with a corresponding user level process. `rlpdump` logs RLP and block erasure traces. `hiccup` emulates a semi-reliable link layer protocol with a configurable error recovery persistency, and with the in-order or out-of-order delivery function. It is used in a controlled, "non-wireless" environment. `hiccup` can also be used to artificially generate excessive packet delays and/or packet re-orderings during a measurement to trigger spurious retransmissions in TCP.

The latter two tools are script files required for post-processing of TCP and RLP traces providing the ability to visualize them correlated in time and at multiple levels of detail. Using these tools, we have post-processed and analyzed a large base of traces representing a variety of mobile data uses (e.g., stationary indoors, walking, driving, etc.).

We use `rlpdump` and MultiTracer in Section 4.2 to study in general inefficient cross-layer interactions that may occur when running TCP-based bulk data transfers over RLP in GSM-CSD. We use `rlpdump` and ReTracer in Section 4.3 to evaluate the benefit of link layer error recovery for reliable flows. This analysis addresses the problems of “underestimation of available bandwidth” (see Section 2.5.1), “inefficiency of end-to-end error control” (see Section 2.5.2), and also the problem of “failure of link layer differential encodings” (see Section 2.5.5). We use `hiccup` in Section 5.1 and Section 5.2 to further study and solve the problem of “competing error recovery” (see Section 2.5.4) for the case of TCP.

In addition to measurements, we use a model-based analysis approach to study TCP-Lite’s retransmission timer in Section 5.3, and to develop a new retransmission timer for TCP in Section 5.4. For that purpose we modeled on a spreadsheet the RTT, the RTO, and all other relevant sender-side connection state variables for the class of network-limited TCP bulk data transfers in steady state. In Section 5.3.5, we validate our model-based analysis through measurements in a real network that yield the same results. This analysis addresses the problem of “competing error recovery” (see Section 2.5.4).

CHAPTER 4

Flow-Adaptive Wireless Links

In this chapter, we propose the concept of *flow-adaptive wireless links* to eliminate the inefficient cross-layer interactions explained in Section 2.5. The key idea is that network end-points use the IP layer as a level of indirection through which their QoS requirements are signalled to each link layer along the path, on a per packet basis. This allows for a (wireless) link layer to adapt its error control schemes to meet those requirements while efficiently utilizing radio resources. Subsequently, we focus on fully-reliable flows, in particular on bulk data flows, leaving the question of appropriate adaptation of link layer error control for real-time flows for future research. The work presented in this chapter has been published in [LR99], [LRKOJ99], [Lud99b], and [LKJK00].

A novelty of our solution is that for fully-reliable flows, we recommend a maximum link layer error recovery persistency that corresponds to TCP's Maximum Segment Lifetime (MSL) of 2 minutes. In Section 4.1.4, we argue and demonstrate that this approach provides higher end-to-end performance than solutions that use a lower link layer error recovery persistency. Through a case study of TCP over RLP in GSM-CSD, we verify in Section 4.2 that flow-adaptive wireless links combined with the recommended link layer adaptations eliminate all known inefficient cross-layer interactions except for the problem of competing error recovery.

In Section 4.3, we show how the GSM-CSD wireless link can be adapted to optimize the end-to-end performance of bulk data flows. This again validates the benefits of providing service differentiation by adapting link layer error control. It also demonstrates that link layer error recovery over wireless links is essential for reliable flows to optimize their end-to-end performance while efficiently utilizing radio resources. Implementing an optimal solution only from the end points of a path seems impossible; even if knowledge about the time varying error characteristics of each wireless link in a path was available. We believe that a similar line of argument applies to real-time flows.

4.1 Extending the Differentiated Service Framework

4.1.1 Providing Differentiated Service through Link Layer Error Control

The motivation behind flow-adaptive wireless links is that the more information about its QoS requirements a flow[1] makes available to a (wireless) link layer, the better the link layer can adapt its error control schemes to meet those requirements while efficiently utilizing radio resources. The latter is crucial because spectrum required for a wireless link is often expensive and mobile radio transceivers consume precious battery power. In the next subsection we discuss the QoS requirements that are relevant in this context and recommend matching adaptation of link layer error control.

The key idea is that network end-points use the IP layer as a level of indirection through which their QoS requirements are signalled to each link layer along the path, on a per packet basis. For that purpose, the QoS requirements are communicated *only* through the IP headers, in particular those parts of the IP header that are not encrypted by IPsec, e.g., the *Differentiated Service Field (DS Field)* [RFC2474]. This approach is the key to making our solution independent from transport (or higher) layer protocol semantics, and allowing co-existence with IPsec.

Flow-adaptive wireless links emphasize link layer error control - in particular error recovery - as a necessary complement to end-to-end error control. This is orthogonal to the “end-to-end argument” [SRC84] that calls link layer error control “an incomplete version of the function

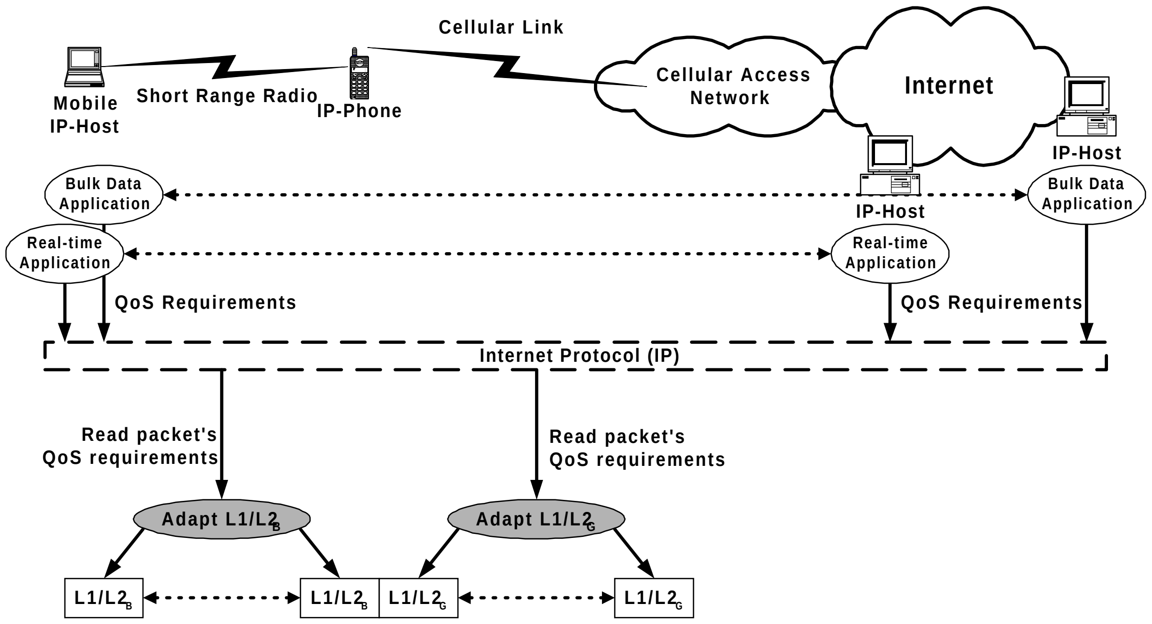


Figure 4-1: The concept of flow-adaptive wireless links.

1. Remember from Section 2.1 that we use the terms *network end-point* and *flow* interchangeable

[error control] provided by the communication system [that] may be useful as a performance enhancement". We believe that carrying a network end-point's QoS requirements as part of the flow's packet headers and accordingly adapting lower layer functions, such as error control schemes, advances the discussion presented in [SRC84].

The concept of flow-adaptive wireless links is exemplified in Figure 4-1. It shows a mobile host that connects to the Internet through a cellular network. The mobile host communicates with two different hosts on the Internet simultaneously resulting in a bulk data and a real-time flow. The connection between the mobile host and the cellular network access device (labeled as IP-phone in Figure 4-1) is yet a second (different) wireless link. Both wireless links are flow-adaptive but independent of each other. Each link deploys its own set of physical and link layer error control schemes called $L1/L2_B$ and $L1/L2_G$ which are optimized for the particular wireless bearer.

Similar models have been proposed: the type-of-service marking model [RFC1349] and the differentiated service architecture [RFC2475] that is based on the mentioned DS Field. Whereas the type-of-service marking has never been widely deployed, the differentiated service architecture is likely to become the standard mechanism for offering more services in the Internet other than just the best-effort service. However, both models are limited to providing service differentiation only by adapting network layer forwarding behavior implemented through functions such as routing, scheduling, and queue management. Our solution should be understood as an extension of these models to additionally provide service differentiation by adapting link layer error control.

Numerous solutions have been proposed in literature to dynamically adapt link (and physical[1]) layer error control schemes. However, as opposed to our solution those mechanisms have been designed to provide fixed QoS requirements (e.g., a fixed maximum residual bit error rates or fixed link latency) and only adapt to quality changes of the wireless bearer. All flows that share such a wireless link receive the same service. This "one size fits all" philosophy is the root of the inefficient cross-layer interactions *"inefficiency of end-to-end error control"* (see Section 2.5.2), and *"overly strong link layer error control"* (see Section 2.5.3). Flow-adaptive wireless links eliminate those inefficient cross-layer interactions as explained in the following subsection.

4.1.2 Defining Service Classes and Matching Link Layer Adaptations

Table 4-2 shows the QoS requirements of the flow types we defined in Section 2.1. These QoS requirements are similar to the type-of-service markings defined in [RFC1349]. Applications that are based on bulk data flows usually require a fully-reliable end-to-end connection. They

1. For example, power control, or spreading factor control in code division multiple access networks.

usually do not have a minimum end-to-end throughput requirement but instead “take what they can get” (best-effort), i.e., their main QoS requirement is to maximize throughput while the end-to-end delay of individual packets is less important. Consequently, bulk data flows mostly utilize their available bandwidth. Also, applications that are based on interactive flows usually require a fully-reliable end-to-end connection. Unlike bulk data flows, however, those flows mostly do not utilize their available bandwidth, i.e., end-to-end throughput is less important. Instead, those applications’ main QoS requirement is to minimize response time although they usually do not have a minimum end-to-end delay requirement (best-effort).

	Minimize Per Packet Delay	**Maximize Flow’s Throughput**	**Reliability**[a]
Bulk Data (e.g., file transfer, e-mail)	not important	yes (best-effort)	fully-reliable
Interactive (e.g., remote terminal, e-banking)	yes (best-effort)	not important	fully-reliable
Real-time (e.g., telephony, radio, stock quote broadcasting)	yes (maximum is application dependent)	depends on flow type	*not* fully-reliable

Tabelle 4-2: Differentiating between flow types.

a. fully-reliable, or not fully-reliable

The QoS requirements of applications that rely on real-time flows are more difficult to define because those applications can have various degrees of delay, throughput, and reliability requirements. For example, traditional telephony requires a fixed end-to-end throughput. It fails if the flow’s available bandwidth falls below and it cannot utilize available bandwidth that exceeds that fixed value. Rate-adaptive audio applications, on the other hand, can operate over a range of available bandwidths. Also, traditional telephony is fairly delay-sensitive (the per packet end-to-end delay may not exceed about 100 ms) whereas streaming applications like IP-based radio can tolerate per packet end-to-end delays on the order of a few seconds due to the implementation of playback buffers. Other real-time applications can even tolerate higher end-to-end delays, e.g., applications that periodically broadcast refresh messages to update obsolete information like stock quotes. The only thing that real-time applications have in common is that they do *not* require a fully-reliable end-to-end connection. Nothing more can be said about their reliability requirement in general. For example, some applications implement error-resilient codecs for which corrupted application data is often still useful. For those codecs to become effective a transparent end-to-end (and link layer) service is required. Real-time applications that are based on non-error-resilient codecs rely on the fact that received application data is error-free and therefore require a semi-reliable or an unreliable end-to-end service.

Below, we propose four new DS Field values that allow a differentiation between four different service classes, and recommend matching adaptations of link layer error control. Note that this does not define how a link's resources are supposed to be allocated among those four service classes in case of congestion. Various alternatives exist for that purpose (e.g., see [FJ95]).

- *Fully-reliable & throughput dominated (bulk data)*
  Packets belonging to this service class should be transmitted with a maximum link layer error recovery persistency of up to the MSL of 2 minutes. The link layer should perform out-of-order delivery as long as this does not interfere with differential encoding schemes[1]. All other link and physical error control schemes should be adapted to provide highest throughput.
- *Fully-reliable & delay dominated (interactive)*
  Packets belonging to this service class should be transmitted with a maximum link layer error recovery persistency of up to the MSL of 2 minutes. The link layer should perform out-of-order delivery as long as this does not interfere with differential encoding schemes. All other link and physical error control schemes should be adapted to provide lowest per packet delay.
- *Real-time & error-resilient*
  Packets belonging to this service class should be transmitted with a "low" link layer error recovery persistency and FEC and interleaving should be adapted to provide "low" delay[2]. For error-resilient codecs to become effective a transparent link layer service is required.
- *Real-time & non-error-resilient*
  Packets belonging to this service class should be transmitted with a "low" link layer error recovery persistency and FEC and interleaving should be adapted to provide "low" delay. A semi-reliable or unreliable link layer service is required as the receiving application cannot tolerate errors in the application data.

We do not further consider real-time flows in this dissertation. This is left for future research. This is also why the above recommendations for real-time flows are rather vague. Throughout the rest of this dissertation we focus on fully-reliable flows, in particular on bulk data flows. Although a validation is outside the scope of this dissertation, we expect that the difference between the two fully-reliable service classes (bulk data and interactive) is that the interactive service class requires a stronger protection of the wireless link (e.g., through stronger FEC) to minimize the per packet delay.

1. In case non-resequencing TCP/IP header compression is run, out-of-order delivery should only be performed for packets belonging to different flows. Note, that [RFC2507] provides a resequencing feature while [RFC1144] does not.
2. Ideally, each packet's IP header carried a "time-to-live" field in milliseconds with respect to a global clock that could be used to limit the maximum delay these error control schemes may introduce. Since such a field does not exist in the IP header today, a conservative maximum delay (e.g., 50 ms) should be assumed.

The important aspect for our work and the novelty of our solution is that we can distinguish between bulk data, interactive, and “other” flows at the link layer for the purpose of adapting link layer error control. This is the key to eliminating the inefficient cross-layer interactions we explained in Section 2.5. In particular it ensures that link layer error recovery, a link layer in-order delivery function, and/or a too large interleaving depth does not interfere with the delay bounds required by real-time flows. This eliminates the problem of *“overly strong link layer error control”* (see Section 2.5.3). Distinguishing between bulk data and interactive flows further allows to better meet those flows’ QoS requirements (maximize throughput versus minimizing per packet delay) as shown in Section 4.3.

The second novelty of our solution is that for fully-reliable flows, we recommend a maximum link layer error recovery persistency corresponding to TCP’s Maximum Segment Lifetime (MSL). This value has been arbitrarily defined as 2 minutes in [RFC793]. The MSL has also been chosen as an upper bound for the reassembly timeout after IP fragmentation [RFC1122]. We are not aware of any proposal or implementation of a link layer protocol that treats fully-reliable flows this way. We explain the motivation for choosing such a high link layer error recovery persistency in Section 4.1.4.

For fully-reliable flows our solution eliminates the following inefficient cross-layer interactions:

- *“underestimation of the available bandwidth”* (see Section 2.5.1)
  This problem is eliminated because the highly persistent link layer error recovery reduces a flow’s error loss rate to the lowest possible value. Thus, only in extreme situations can the network end-points receive a false congestion signal. A long link outage (longer than 2 minutes) is an examples of such an extreme situation. However, in that case it is best to signal congestion, anyway, so that the network end-points can adapt to a conservative packet send rate.
- *“inefficiency of end-to-end error control”* (see Section 2.5.2)
  This problem is eliminated because the highly persistent link layer error recovery reduces the number of end-to-end retransmissions triggered by error losses to the lowest possible value. Thus, in the common case, end-to-end retransmissions are either spurious or triggered by congestion losses. In Section 5.2, we propose an enhancement to TCP that also eliminates avoidable spurious retransmissions.
- *“failure of link layer differential encodings”* (see Section 2.5.5)
  This problem is eliminated because the highly persistent link layer error recovery ensures that “deltas” (e.g., compressed TCP/IP headers) are never lost or only lost in extreme situations like the above mentioned long link outage. Consequently, our approach obsoletes efforts to make TCP/IP header compression robust against packet losses, e.g., the Twice algorithm and the header request mechanism proposed in [RFC2507].

Hence, for fully-reliable flows our solution of flow-adaptive wireless links combined with the recommended link layer adaptations eliminates all known inefficient cross-layer interactions except for the problem of *"competing error recovery"* (see Section 2.5.4). Yet, we believe that that problem has to be solved by making the respective fully-reliable end-to-end protocol more robust. In Section 5.2, we therefore propose a solution that eliminates that problem for TCP.

4.1.3 Deployment Concerns and Implementation Alternatives

There are certainly deployment concerns associated with our concept of flow-adaptive wireless links. First of all, it requires the implementation and deployment of two things: (1) new wireless link layers that are capable of providing the required service differentiation, and (2) an extended network/link layer interface via which the network layer can request an appropriate service class for the transmission of an IP packet. Both do not exist today but would need to be deployed in future wireless systems. Nevertheless, this is not such a severe problem because it only causes "local" changes, i.e., it only requires changes to hosts and routers that are connected to such a wireless link.

A more problematic and fundamental deployment concern is that network end-points today do not explicitly include their QoS requirements in their flows' packet headers. This prevents the necessary differentiation between IP packets at the network/link layer interface. Solving this problem requires changes at *every* network end-point that may be located at the end of a path that includes a wireless link, i.e., essentially every host on the Internet. Hence, it requires standardization, adoption, and deployment not only of the differentiated services framework but also of the extensions we have proposed in the preceding two subsections.

It is, however, possible to sidestep this deployment concern with a "layer-violating trick". A link layer implementation can inspect the protocol identifier in the IP header of each packet to distinguish between TCP, UDP, and ICMP packets. TCP packets clearly belong to a fully-reliable flow. The implementation could further "guess" that UDP packets belong to a real-time flow that is "very" delay-sensitive, e.g., using a the above mentioned maximum delay bound of 50 ms, and that ICMP packets belong to a less delay-sensitive real-time flow. To further distinguish between TCP packets that belong to a bulk data versus those belonging to an interactive flow, the implementation could check the TCP port numbers, e.g., telnet (interactive) and ftp (bulk data) servers have uniquely defined port numbers. This idea has already been implemented for certain link layer framing protocols (see note on pp. 35 of [Ste94]) to control scheduling algorithms that prioritize packets belonging to interactive over those belonging to bulk data flows. In [LR99] we have developed a solution that applies this idea for the purpose of adapting link layer error control, i.e., a solution to implement a flow-adaptive wireless link.

However, this trick has a number of major problems. First, it violates the paradigm of protocol layering as discussed in Section 2.7 preventing interoperability with IPsec. The reason is that IPsec encrypts both the protocol identifier in the IP header[1], and also the port numbers in the TCP header. Again, this prevents the necessary differentiation between IP packets at the network/link layer interface. Secondly, not every UDP-based flow is a real-time flow as some application layer protocols build end-to-end reliability on top of UDP (e.g., [FJLMZ97], [RFC1831], [RFC1094]). Thus, such flows would be falsely treated as real-time flows when they should have been treated as bulk data or interactive flows.

4.1.4 Link Layer Error Recovery Persistency for Fully-Reliable Flows

There have been debates [ES98], [Kar93], [LRKOJ99], [PILC] about how persistent link layer error recovery should be implemented for fully-reliable flows. The "end-to-end argument" [SRC84] tells us that it is not worth the effort to implement perfect reliability at the link layer. Yet, our design should eliminate non-congestion related packet losses to avoid interference with end-to-end congestion control schemes as explained in Section 2.5.1. Implementing a low error recovery persistency as proposed in [Kar93] is a compromise that avoids this conflict by emphasizing end-to-end error recovery. However, this approach has fundamental problems.

First, the link layer sender has no way to decide *when* to "give up" and discard the packet to, e.g., stay within the bounds of TCP's retransmission timer and/or to reduce the flow's error loss rate below the flow's probing loss rate. This is not feasible as it requires knowledge of the flow's RTT, which cannot be known at the link layer (unless it was explicitly signalled from a network end-point). Also, the higher the rate of error losses, the more inefficient cross-layer interactions this causes with link layer differential encodings (see Section 2.5.5 and Section 4.2.3). A low error recovery persistency therefore requires strong FEC, i.e., more channel overhead, to keep the rate of error losses low. Together with the non-data-preserving property[2] of semi-reliable link layer error recovery, this cannot yield optimal end-to-end throughput.

A more fundamental problem occurs in case of temporary link outages, e.g., when a user temporarily roams into an area without wireless connectivity. In this case, all of a flow's packets that are in flight are eventually queued at the wireless link where the outage occurred. A link layer that implements a low error recovery persistency discards those packets over a short period of time. This may cause a phase where the link is left idle due the sending network end-point waiting for a possibly backed-off retransmission timer to expire before the next packet is sent. During that time the link may have already become available again. In case of TCP this

1. Actually the value of the protocol identifier field in the IP header is changed to reflect one of the two protocol identifiers defined for IPsec.
2. Those fragments of an IP packet that had already been transmitted before the packet is finally discarded after the error recovery persistency was reached have wasted resources of the wireless link.

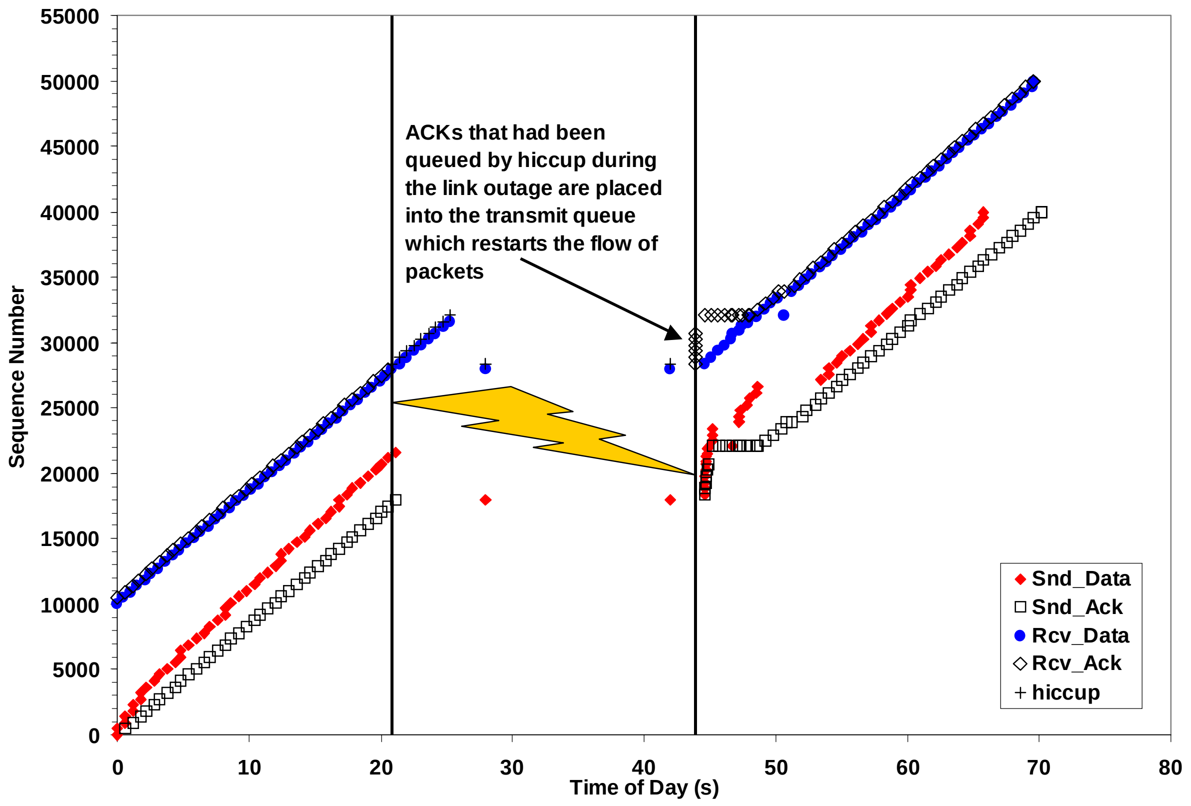


Figure 4-3: Link outage with high link layer error recovery persistency.

idle phase can be up to 64 seconds (the maximum RTO) long. If, on the other hand, packets are still queued at the wireless link, the end-to-end flow of packets is re-started immediately after the link has become available. We demonstrate both cases based on TCP using the `hiccup` tool and the experimental network explained in Section 3.3. We run two bulk data transfers and transmit 40 KBytes each time. In both cases we use `hiccup` to emulate a link outage of 23 s on the TCP receiver's outbound interface, i.e., in the direction that ACKs are sent. The other direction is not affected[1]. In both cases this leads to two spurious timeouts. Note that although we demonstrate these effects with bulk data transfers the same problem exists for interactive flows.

In the first case (see Figure 4-3), `hiccup` emulates a semi-reliable link layer with an error recovery persistency of 2 min. During the link outage all ACKs sent by the TCP receiver are queued by `hiccup`. None of those packets is discarded due to the high error recovery persistency that is emulated. Hence, as soon as the link outage is over, the queued ACKs are transmitted and arrive at the TCP sender as a single burst (this phenomena is called ACK compression [ZSC91], [Mog92]). This immediately restarts the flow of packets, and the bulk data transfer terminates about 70 s after it was initiated. Figure 4-3 also illustrates the go-back-N

1. We excluded that direction from the link outage to make retransmissions visible in the trace plots. The same problem occurs if the link outage affects both directions, but it cannot be illustrated so easily.

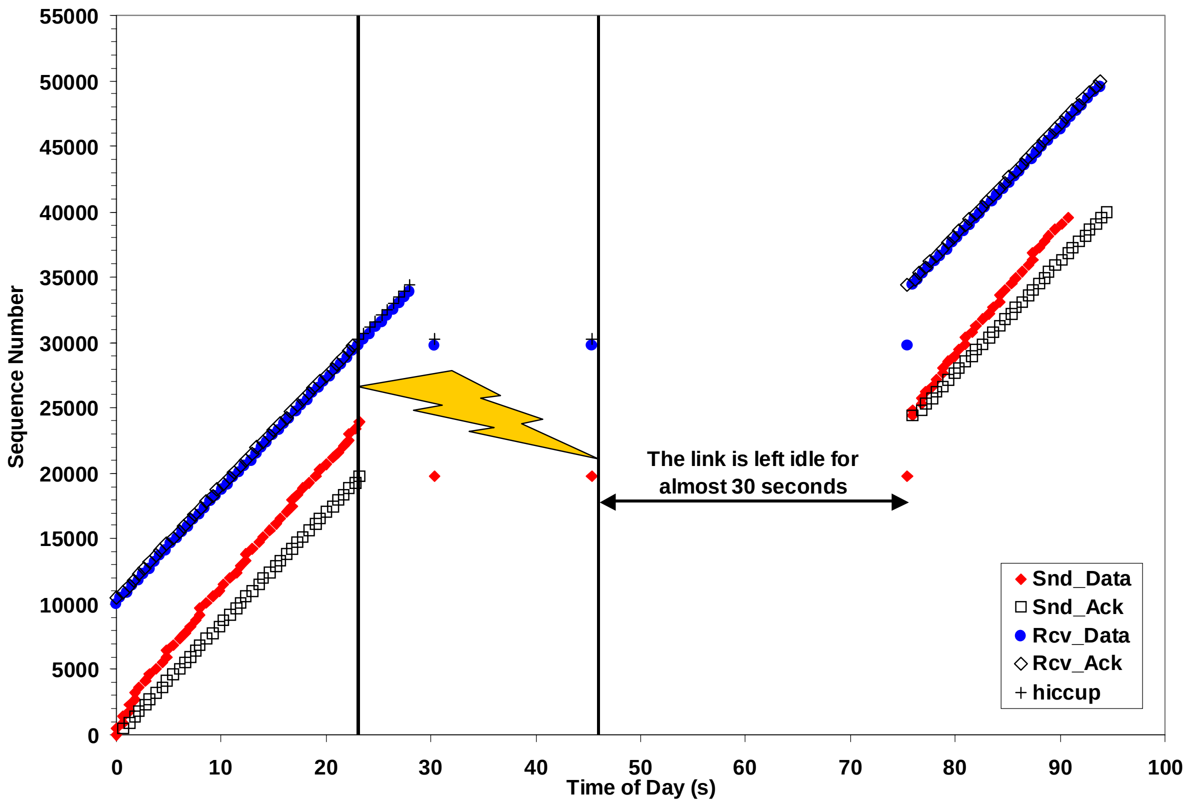


Figure 4-4: Link outage with low link layer error recovery persistency.

behavior in TCP that occurs after spurious timeouts. The Eifel algorithm presented in Section 5.2 eliminates this inefficiency.

In the second case (see Figure 4-4), `hiccup` emulates a semi-reliable link layer with an error recovery persistency of 500 ms. During the link outage all ACKs sent by the TCP receiver are eventually discarded by `hiccup`. Hence, when the link outage is over, there is no ACK queued that could restart the flow of packets. Instead, the TCP sender is forced into a third timeout. The resulting third retransmission finally restarts the flow of packets, however, only after the link was left idle for almost 30 s. Consequently, the bulk data transfer takes about 25 s longer as compared to the first case.

Highly persistent link layer error recovery for fully-reliable flows has none of the problems explained above. It ensures as far as possible that any loss at the link is caused by congestion[1] which is the right signal to give to the senders of loss-responsive flows. For the same reasons that favor highly persistent link layer error recovery for fully-reliable flows, should wireless networks implement mechanisms to support lossless intra- (and if possible also inter-) system cell handovers for data belonging to fully-reliable flows.

1. Apart from the more unlikely events of link layer error detection failures.

4.2 Real-World Interactions between TCP and RLP

In Section 4.1, we concluded that for fully-reliable flows our solution eliminates all known inefficient cross-layer interactions except for the problem of *"competing error recovery"* (see Section 2.5.4). To validate that conclusion and to evaluate how serious the problem of competing error recovery is in a real-world wireless network like GSM-CSD, we study potentially inefficient interactions between TCP and RLP during bulk data transfers. While we show that competing error recovery is not a problem for TCP in GSM-CSD, our multi-layer tracing approach explained in Section 3.2 allowed us to detect some unexpected results. Firstly, we observe the negative impact that overbuffered links has on end-to-end performance. Secondly, RLP link resets lead to large amounts of data being lost due to an interaction with TCP/IP header compression; a problem that is aggravated by overbuffered links.

4.2.1 Interactions are Rare

In general, we have found that TCP and RLP rarely interact in an inefficient way. As depicted in Figure 4-5, in almost 85 percent of all our measurements, the utilization (as defined in Section 3.1.2) of the GSM-CSD channel was 98 percent or more[1]. Even in those measurements where we detected inefficient protocol interactions, the utilization never dropped below

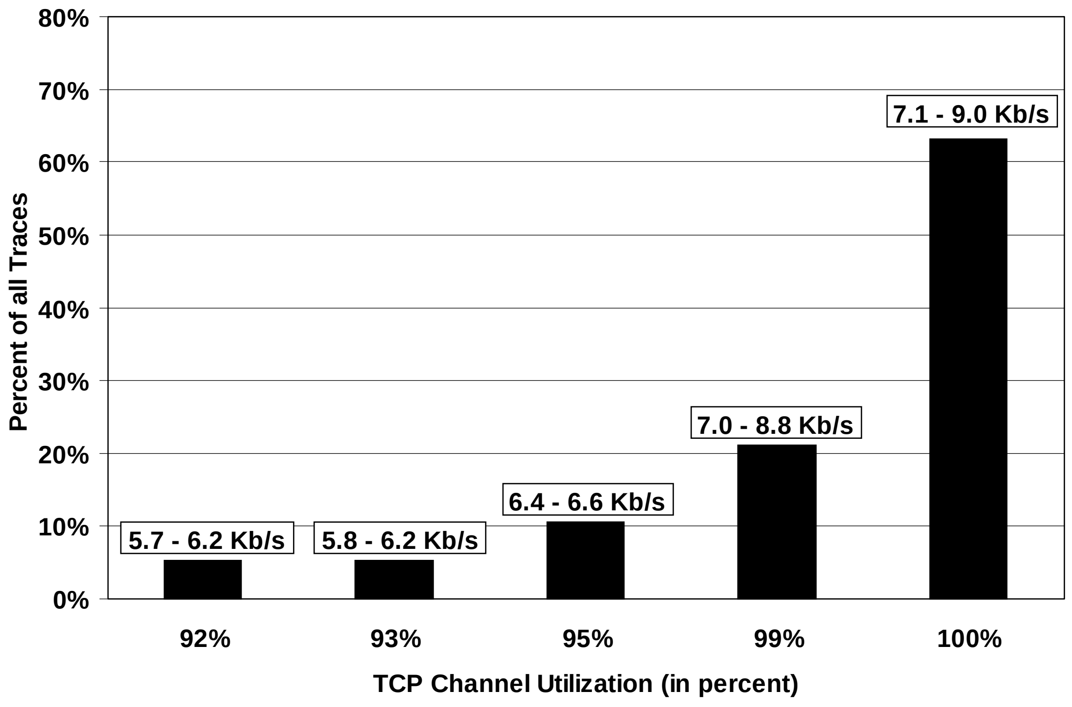


Figure 4-5: TCP channel utilization.

1. Note that we use "rounded" figures. For example, a measured utilization between 99 and 100 percent is counted as 100 percent.

91 percent. We had not expected to observe such high figures, given that one third of all measurements were taken in an environment with poor receiver signal strength (see Section 3.2.3). In such an environment we had expected to find cases of competing error recovery between TCP and RLP. In fact, we only found two such incidents. All measurements that yielded a utilization of 95 percent or less suffered from the impact of RLP link resets when TCP/IP header compression [RFC1144] was used. This is further explained in Section 4.2.3.

Figure 4-5 also shows the throughput range that `sock` achieved for measurements that yielded the same utilization. Taking protocol overhead into account, the throughput was mostly close to the bit rate of the channel[1]. These results confirm similar findings from [Bau97] and [KRLKA97]. However, unlike in those studies, our tools provided us with the unique opportunity to measure utilization in addition to throughput. Thus, we could determine that a measurement (using TCP/IP header compression) which resulted in a throughput of only 7.0 Kbit/s, but yielded an utilization of 99 percent must have suffered from a non-optimal radio connection. Consequently, the RLP sender must have retransmitted a higher number of frames. The overall throughput results, however, suggest that the GSM-CSD channel is over-protected with FEC. This topic is further studied in Section 4.3.

4.2.2 Excessive Queueing

One problem that is evident in the trace plots is the large mismatch between the pipe capacity and the load that the TCP sender puts onto the network. The pipe capacity in this network is already reached with 2 segments, assuming an MTU of 512 bytes. However, as can be seen in Figure 4-6, the TCP sender increases its load up to 8 KBytes or 16 segments. As explained in Section 2.3, the TCP sender has no way to determine the pipe capacity and, thus, will periodically increase its congestion window (the load) until the TCP receiver's advertised window is reached. The latter usually corresponds to the default socket buffer size of commonly 8 or 16 KBytes.

In the measurement platform shown in Figure 3-5, packets are queued in the mobile host's outbound interface buffer. For downlink transmission, those packets would be queued at the other side of the bottleneck link, e.g., at the ISP shown in Figure 2-7[2]. Thus, the core of the problem is a largely overbuffered link. The default interface buffer size in the BSD system is 50 packets. Obviously, this is an inappropriate size for a mobile host that usually does not have a large number of simultaneous connections.

1. Note that some measurements were done with and others without TCP/IP header compression. Also, some commercial GSM networks provide a user rate of 1200 bytes/s, whereas others only provide 960 bytes/s (see Section 3.2.5).
2. This is why Internet Service Providers (ISPs) often configure their equipment to not allow more than 3 - 4 packets worth of buffer space per access line into their modem pool.

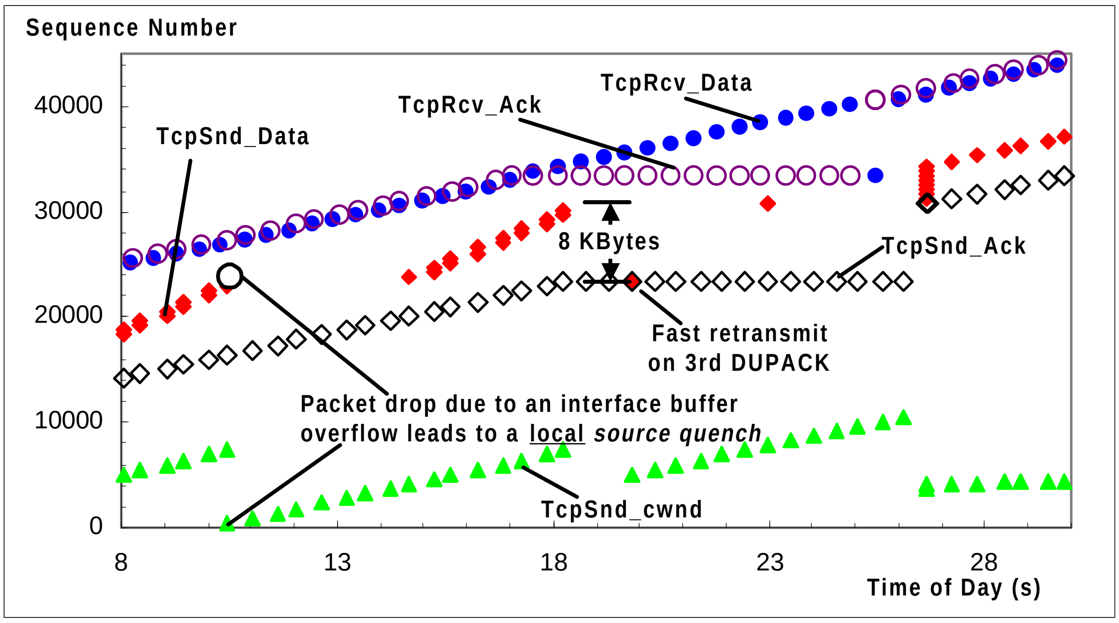


Figure 4-6: Local buffer overflow (zoom of Figure 3-6).

We have purposefully compiled a kernel of the BSD/386 Version 3.0 operating system with an interface buffer that was smaller than the socket buffer size of 8 KBytes to provoke a *local* packet drop as shown in Figure 4-6. In response, the *tcp_quench()* function [WS95] resets the TCP sender's congestion window to $1 \times MSS$ (we have explained this part of the trace already in Section 3.2.1). After about one half of the current RTT, the sender can again send additional segments until the DUPACKs for the dropped packet trigger the fast retransmit algorithm. This leads to setting the congestion window to one half of its value before the local drop occurred (see the TcpSnd_cwnd graph). At this point, the sender has reached the advertised window and cannot send additional segments (which it could have otherwise) while further DUPACKs return during the fast recovery phase. Thus, when the retransmission is acknowledged 26.5 s into the connection, a burst of 8 segments is sent out by the TCP sender at once.

As can be seen from the TCP receiver trace, excessive queuing, the ups and downs of the congestion window at the TCP sender, and even retransmissions do not much degrade throughput performance. But excessive queueing has a number of other negative effects:

- It massively inflates the RTT. In fact, a second TCP connection established over the same link is likely to suffer from a timeout on the initial connect request (SYN). That timeout would occur because it takes longer to drain the queue packets (here up to 14 x MTU or 7 KBytes) on a 960 bytes/s link than the commonly used initial setting for TCP's RTO of 6 seconds.
- If the timestamp option is not used, the RTT is sampled less frequent, leading to an inaccurate RTO.

- An inflated RTT inevitably leads to an inflated RTO, which can have a significant negative impact on TCP's performance, e.g., in case of multiple losses of the same packet. The negative impact results from the exponential timer backoff explained in Section 2.2.2 and is illustrated in Figure 4-9.
- For downlink transmissions (e.g., web browsing), where no appropriate limit is imposed onto the outbound interface buffer of the bottleneck router, the data in the queued packets may become obsolete (e.g., when a user aborts the download of a web page in favor of another one). In that case, the "stale data" must first drain from the queue, which in case of a low bandwidth link, may take on the order of several seconds.
- In case of a handover during downlink data transfer many packets may have to be discarded or transferred to the new downlink queue (see Section 2.4.2).

A simple solution to these problems is to statically adjust the interface buffer size to the order of the interface's bit rate (An interface buffer of 50 packets is certainly too large for an interface bit rate of 9.6 Kbit/s.). A more advanced solution is to deploy active queue management (see Section 2.3.3) at both sides of the bottleneck link. The goal is to adapt the buffer size available for queueing to the bit rate of the interface, a given worst-case RTT, and the number of connections actively sharing the link. Combining active queue management with an explicit congestion notification mechanism (see Section 2.3.3) would further improve network performance as fewer packets would have to be dropped and retransmitted. In fact, we regard it as imperative that these mechanisms be implemented at both ends of wide-area wireless links, which we believe will be the bottleneck in a future Internet. Given that those solutions already exist we do not further consider the problem of excessive queueing.

4.2.3 The Impact of RLP Link Resets

One of the key findings of our multi-layer tracing analysis is an understanding of the impact of RLP link resets (see Section 2.4.3) when TCP/IP header compression is used to reduce the per segment overhead. We want to point out, though, that RLP link resets are very rare events. We have captured 14 resets, all of which occurred when the receiver signal strength was extremely low. In all cases, the link reset was triggered because RLP's default error recovery persistency (see Section 2.4.3) was reached, i.e., a frame had to be retransmitted more than 6 times. Our results suggest that this default value is too low and needs to be increased. TCP connections before and after the link reset usually progress without problems and there is no apparent reason why the link should be reset.

Figure 4-7 depicts this problem as perceived by the TCP receiver. We have only plotted the ACKs generated by the receiver and the RLP link resets (of which we captured two within 100 seconds). As can be seen, the first link reset leads to a gap of 11 s and 18 s for the second reset. During both gaps, no data is received correctly as neither ACKs or DUPACKs are sent

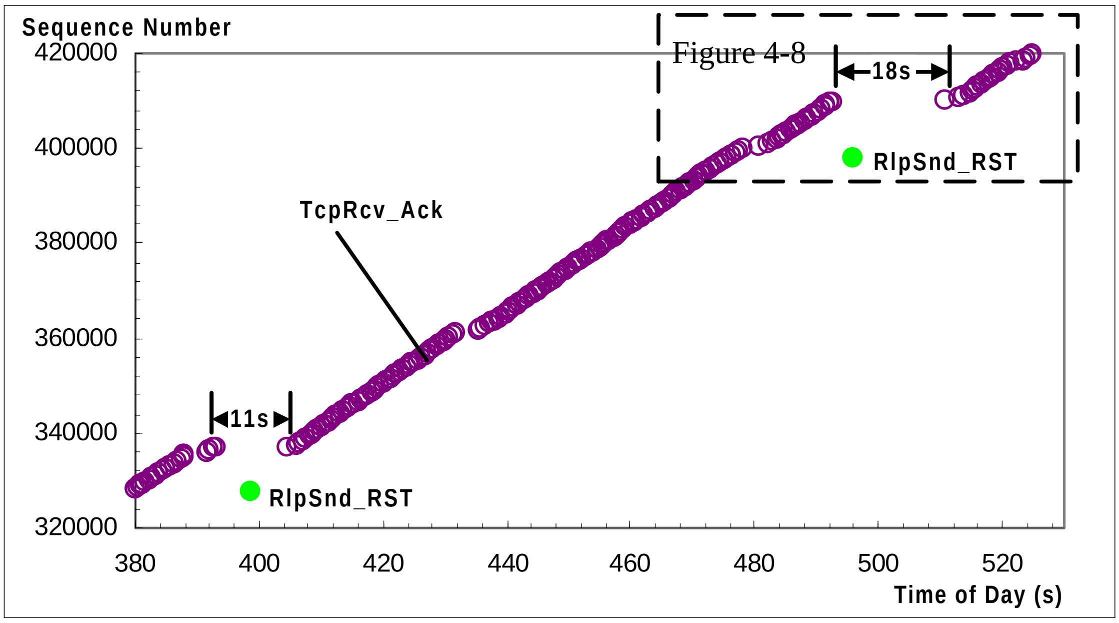


Figure 4-7: Header decompressor failures.

by the TCP receiver. Consequently, the throughput during the interval depicted in Figure 4-7 drops by more than 25 percent down to 634 bytes/s.

Figure 4-8 shows a detailed examination of what happens after the RLP link reset. The reset apparently caused the loss of 5 segments. Remember from Section 2.4.3 that RLP provides a byte stream service. Thus, if only the first or last few bytes of a PPP frame are lost when the RLP sender and receiver flush their buffers after the link reset, the whole PPP frame is discarded by the PPP receiver because of a checksum error. In this case, it causes the TCP/IP

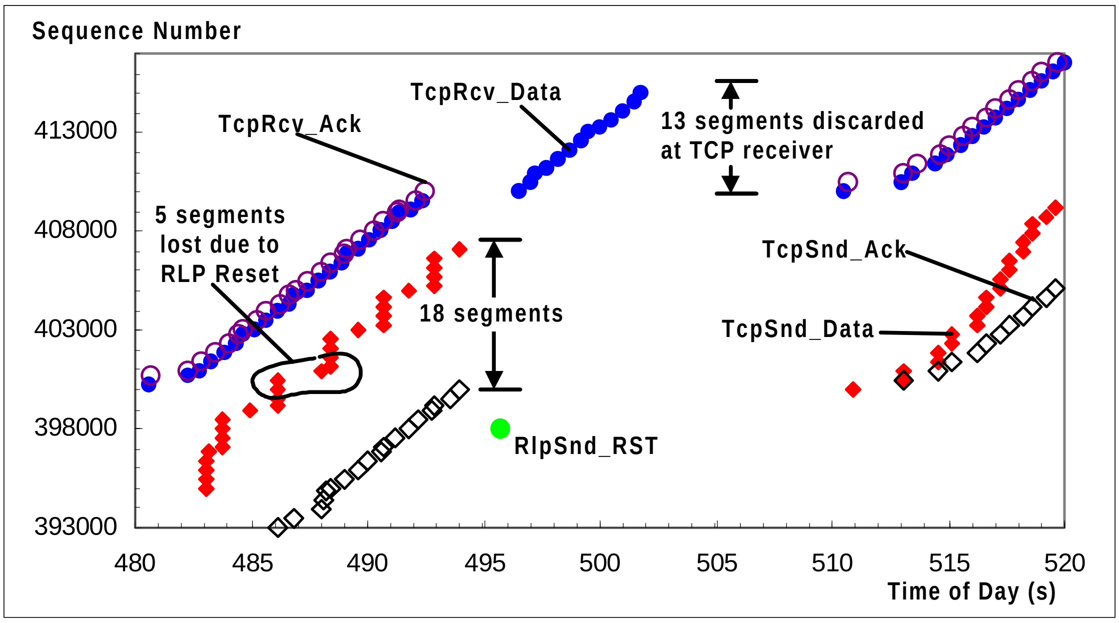


Figure 4-8: Zoom of Figure 4-7.

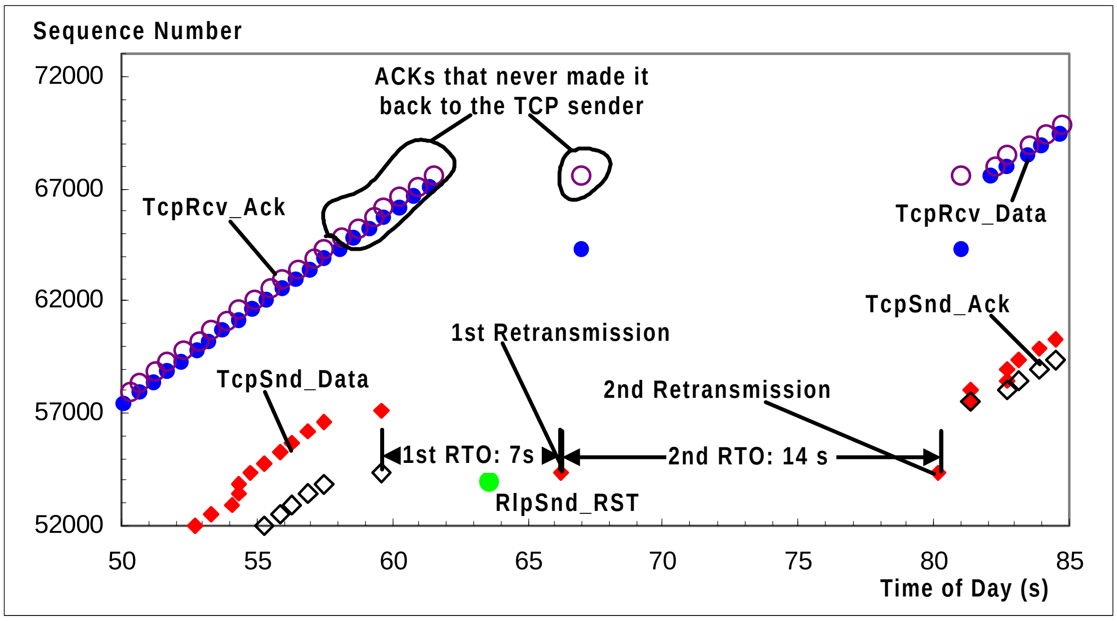


Figure 4-9: Exponential retransmission timer back-off.

header decompressor to be off by 5 segments, so that segment i + 5 is falsely decoded as segment i and so forth. For that reason, thirteen of the segments shown in the plot are not acknowledged by the TCP receiver because they are discarded due to checksum error. These segments should actually have been plotted with an offset of 5 x MSS parallel to the y-axis.

Another variant of the same problem is shown in Figure 4-9. This time ACKs get lost, including the one for the first retransmission; again due to a RLP link reset. This loss leads to an exponential backoff of TCP's retransmission timer. Since the retransmission timer value is significantly inflated due to the excessive queueing explained in the preceding subsection, it causes a particularly long idle phase.

4.2.4 Competing Error Recovery Only in Pathological Cases

Since RLP implements the in-order delivery function, competing error recovery between TCP and RLP can only cause spurious timeouts. (In Section 2.5.4, we mentioned spurious fast retransmits as another potential cause of competing error recovery.) A spurious timeout can be easily seen in a TCP trace plot: the ACK for a correctly received segment reaches the TCP sender *after* the retransmission timer covering that segment has expired. We only found two such instances in all our traces. However, both times the spurious timeout occurred at the beginning of the connection when the TCP sender had not yet converged to an appropriate retransmission timer value. Also, both times the receiver signal strength was very low and the RLP sender had performed many retransmissions at that time. All other timeouts we found were related to RLP link resets. In contrast, we found several instances that show that TCP-Lite's retransmission timer is conservative enough to even allow for extra delays due to link

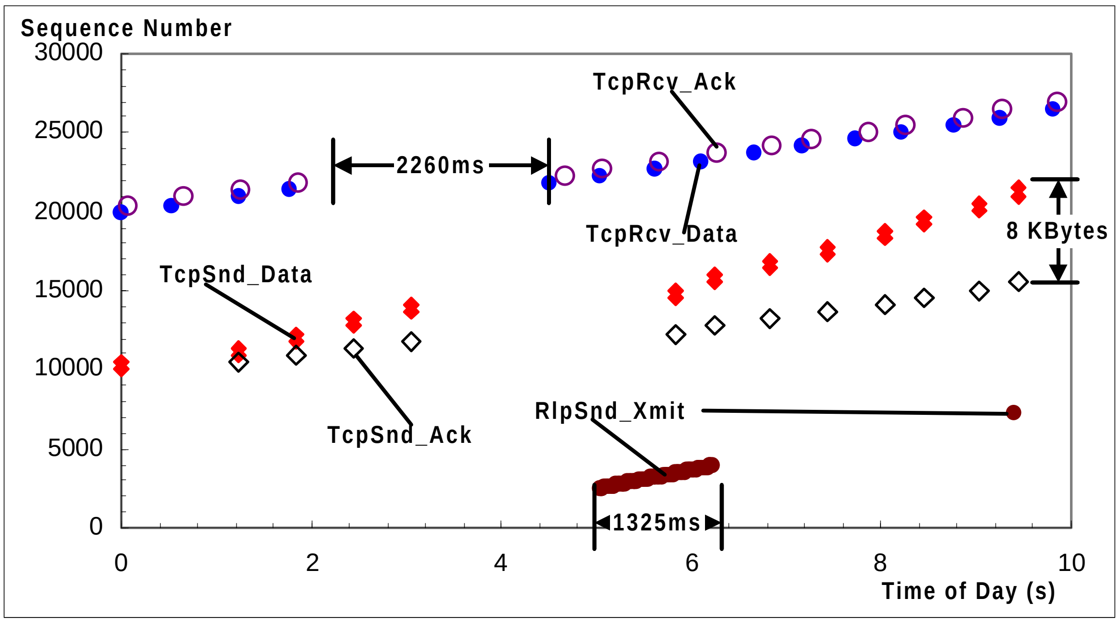


Figure 4-10: First 10 seconds of the trace in Figure 3-6.

layer error recovery beyond 1200 ms. This is depicted in Figure 4-10 which shows a burst of retransmissions on the RLP layer causing a delay of 1325 ms. This leads to an idle phase of 2260 ms during which no data arrives at the TCP receiver. One reason for the difference in these values is that the end of a segment could have been affected by the retransmissions, which would require a full round-trip time on RLP layer (about 400 ms [LR99]). It cannot be the case that the returning ACKs were delayed in addition to the segment, as the plot shows no sign of ACK compression.

We were curious to understand why [KRLKA97] *did* find so many spurious timeouts in their study which used almost the same network setup as ours. The authors of that study believe that these spurious timeouts were caused by excessive RLP retransmissions, i.e., because of competing error recovery between TCP and RLP. While it appears as if our results contradict those results, this is not the case. The reason apparently lies in differences between the implementations of TCP that were used in both studies. Some implementations of TCP seem to maintain a more aggressive retransmission timer than others. Moreover, in our measurements the TCP timestamp option was enabled, yielding a more accurate estimation of the RTT and consequently also a more accurate retransmission timer. Timing every segment instead of only one segment per RTT (which is done when the timestamp option is not used) enables a TCP sender to more quickly adapt the retransmission timer to sudden delay increases. Thus, we believe that timing every segment is an attractive enhancement for TCP in a wireless environment.

4.3 Optimizing Wireless Links for Bulk Data Flows

In Section 4.1, we concluded that our solution of distinguishing between flow types allows to better meet each flow's QoS requirements by appropriately adapting link layer error control. In this section, we validate this approach for the case of bulk data flows using the methodology explained in Section 3.1. We show that the throughput of the GSM-CSD channel can be improved by up to 25 percent by increasing the (fixed) RLP frame size. Furthermore, we show that pure end-to-end error recovery, i.e., when the wireless link is *not* protected by link layer error recovery, fails as a general solution to optimize end-to-end performance provided by bulk data flows.

4.3.1 Block Erasure Rates and Burstiness

Deriving the overall BLERs for *trace_A*, *trace_B* and *trace_C* (defined in Section 3.1.3) would have delivered little useful information. Instead, we also capture how "fast" the BLER changes over time in a given radio environment. We therefore divide each trace into one minute sub-traces and treat each of those independently.

Figure 4-11 summarizes the BLERs that we have determined in this manner. The BLERs for the sub-traces of *trace_A* are not shown because we found *trace_A* to be almost free of block erasures: over 96 percent of all sub-traces do not have a single block erasure and the remaining ones have a BLER of less then 0.0025. This result shows how strongly the GSM-CSD channel

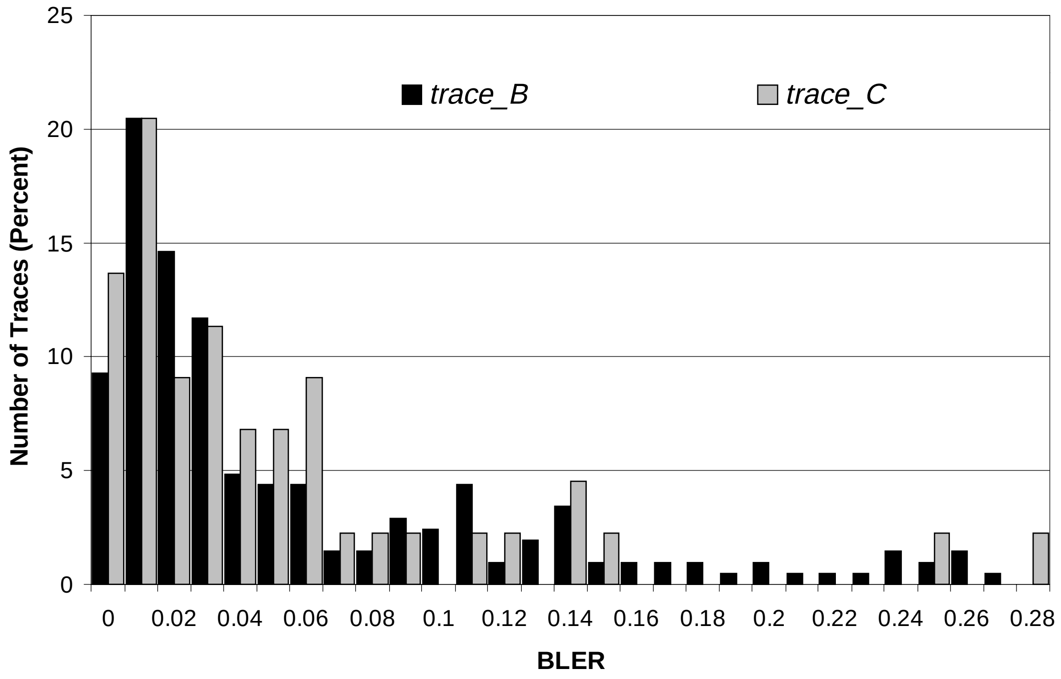


Figure 4-11: Measured BLERs.

is protected by FEC and interleaving, leaving little error recovery work for RLP. This is especially striking because radio environment *A* was far from ideal, as it only provided a receiver signal strength of 3 - 4. Many radio environments provide a maximum receiver signal strength of 5. This suggests that a weaker FEC scheme and/or a larger RLP frame size would increase channel throughput. The results for *trace_B* and *trace_C* are similar but different from the results for *trace_A*. In these, over 30 percent of all sub-traces have no single block erasure or a BLER of less then 0.01. But overall the BLERs vary considerably and can be as high as 0.28. These large variations take place over time scales of one minute, which corresponds to 3000 RLP frames. This is “slow” enough to make adaptive error control schemes applicable even within the same radio environment. This is important because otherwise such schemes would only be effective if the mobile user changed location to a different radio environment. The reason is that adaptive error control schemes only adapt with a certain latency, which depends on the delay required to feedback channel state information. We leave the study the potential of adaptive frame length control (e.g., proposed in [ES98] and [LS98]) to increase channel throughput for future research. This decision is partly driven by our measurement-based analysis approach and the fact that we are currently not able to implement schemes like adaptive FEC in our measurement platform.

Figure 4-12 shows the cumulative distribution function for the burst error lengths, i.e., the number of consecutive blocks that suffered an erasure, for *trace_B* and *trace_C*. There was no point in showing this for *trace_A*, as it was basically error-free. Over 50 percent of burst errors are only 1 or 2 blocks long. Longer error bursts are more common when the mobile host is stationary, e.g., in *trace_B* less than 5 percent of all error bursts are larger than 26 blocks, whereas

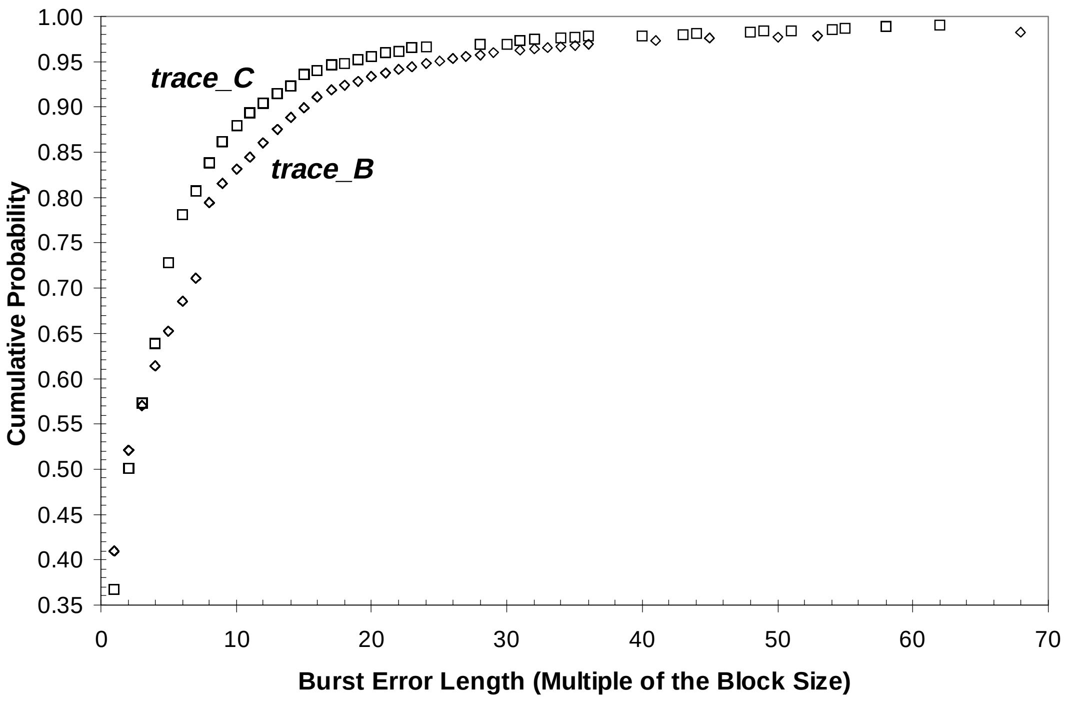


Figure 4-12: Burst error length distribution.

in *trace_C* this number drops to 18. This comparison is valid as the BLERs of both traces are of the same order. As discussed in Section 3.1.2, the distributions shown in Figure 4-12 do not sufficiently describe the wireless channel's error characteristic. They do not show whether the burst errors occur in clusters or isolated, i.e., the correlation between error bursts is not captured. In the following section, we show how the (fixed) frame size that maximizes channel throughput can be used to quantify this correlation.

4.3.2 Error Burstiness Allows for Larger Frames

In this section, we determine the fixed RLP frame size that maximizes channel throughput in the radio environments *A*, *B*, and *C*. For that purpose, we perform the retrace analysis described in Section 3.1.4 leading to the results shown in Figure 4-13. An optimal frame size of 1410 bytes yields a throughput of 1423 bytes/s for *trace_A* and a frame size of 210 bytes maximizes throughput to 1295 bytes/s for *trace_C*. The results for *trace_C* are close to those of *trace_B*. The gradual performance improvements in the case of *trace_A* rapidly decrease above a frame size of 210 bytes. A frame size of 210 bytes still yields a throughput of 1392 bytes/s. This indicates that for an adaptive frame length control algorithm, it would be sufficient to adapt the frame size in a range of about 30 - 210 bytes.

A key result of our analysis is that the (fixed) frame size chosen for RLP was overly conservative. Increasing it to 210 bytes improves the channel throughput by at least 18 and up to 23 percent depending on the radio environment[1]. We were not able to verify which studies led to the decision to standardize an RLP frame size of 30 bytes. However, our results show that

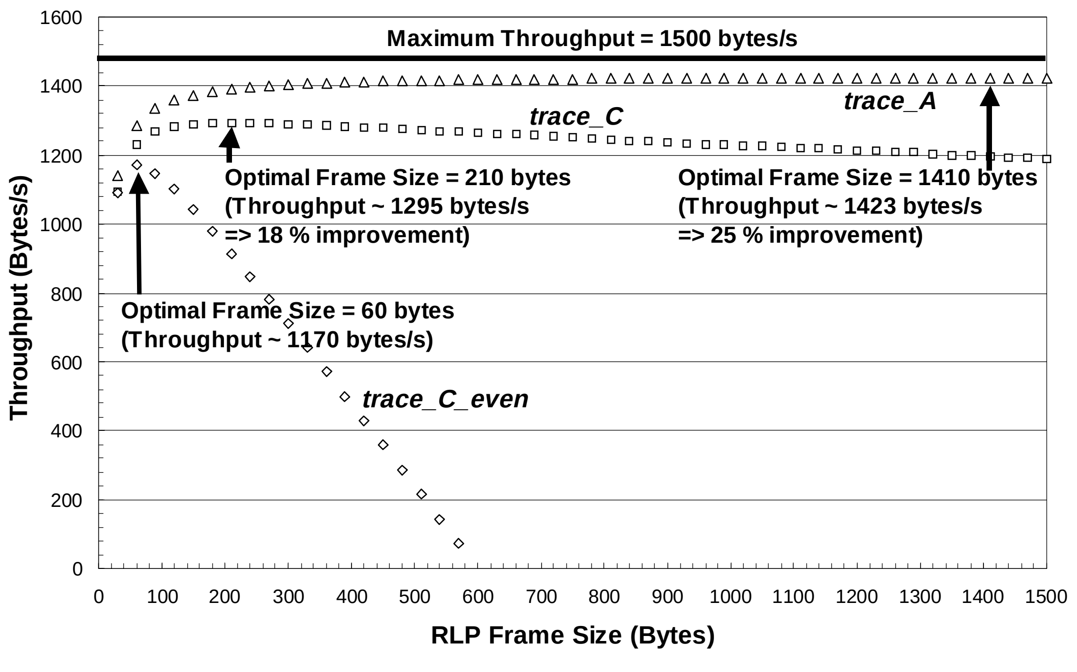


Figure 4-13: Throughput versus frame size.

they must have been based on an unrealistic error model of the GSM-CSD radio channel. This highlights the importance of measurement-based analysis of protocol performance over wireless links.

Another result is that the error burstiness on the GSM-CSD channel allows for larger frame sizes than if block erasures are not bursty. This effect can seen by comparing the graphs *trace_C* and *trace_C_even* in Figure 4-13. The retrace analysis for *trace_C_even* yields an optimal frame size of only 60 bytes (*trace_B* and *trace_B_even* are similar). One could view the quotient of the optimal frame size for an error trace (bit error trace or block erasure trace) and the corresponding "*_even*" trace as the *burst error factor*. The closer a trace's burst error factor is to 1, the less the corresponding channel exhibited error burstiness. Note, that the burst error factor also depends on the per frame overhead chosen for the retrace analysis. To eliminate this dependency, one could base the definition of the burst error factor on a retrace analysis that assumes a per frame overhead of zero.

4.3.3 The Failure of Pure End-to-End Error Recovery

Based on *trace_C*, we perform the best-case analysis described in Section 3.1.2 using TCP as an example of a pure end-to-end error recovery protocol. For that purpose we repeat the retrace analysis assuming a per MTU overhead of 47 bytes (20 bytes TCP header, 20 bytes IP header, and 7 bytes of PPP overhead). The retrace analysis shows that the end-to-end throughput is maximized with an MTU size of 690 bytes. The reason for the difference with the corresponding analysis for RLP is the larger overhead per transmission unit. The first row of Table 4-14 shows the result for commonly used MTU sizes. The second row shows the end-to-end throughput that is achieved when running RLP with a frame size of 210 bytes, providing a channel throughput of 1295 bytes/s (see Figure 4-13).

	MTU 296 bytes	**MTU 576 bytes**	**MTU 1500 bytes**
Pure End-to-End (No TCP/IP header compression)	1151 5.2%↑	1219 2.4%↑	1196
End-to-End with RLP (No TCP/IP header compression)	1094	1191	4.9%↓ 1255
End-to-End with RLP (With TCP/IP header compression)	↓7.6% 1239	↓3.8% 1265	↓7.4% 1284

Tabelle 4-14: Application layer throughput in bytes/s.

1. For example, for *trace_A*, the retrace analysis yields a throughput of 1392 bytes/s for a frame size of 210 bytes and a throughput of 1138 bytes/s for a frame size of 30 bytes/s. For *trace_B* and *trace_C*, these frame sizes yield a throughput of 1295 bytes/s and 1096 bytes/s, respectively.

Pure end-to-end error recovery achieves a 2.4 and 5.2 percent increase in best-case application layer throughput for MTU sizes of 576 and 296 bytes, respectively. This shows that pure end-to-end error recovery consumes less radio resource for these MTU sizes as discussed in Section 3.1.2. However, even when TCP-SACK is used, it is unlikely that the advantage in end-to-end throughput would be achieved in practice, due to interference with TCP's end-to-end congestion control scheme. The benefit of link layer error control becomes evident with larger MTU sizes (e.g., the commonly used 1500 bytes - see Table 4-14) and when IP header compression is used over the wireless link[1].

For pure end-to-end error recovery, TCP/IP header compression, as defined in [RFC1144] and [RFC2507], are not an option because of the frequent error losses (see Section 2.5.5). While the Twice algorithm proposed in [RFC2507] is more robust, it has problems when two or more packets with compressed headers are lost back-to-back. However, this is a likely event for the GSM-CSD wireless link (if not protected by RLP) as shown in Figure 4-15. The cumulative distribution of the number of back-to-back packet losses shows that 66, 59, and 48 percent of all such losses have a length of 2 or larger for an MTU of size 296, 576, and 1500 bytes, respectively. Alternatively, [RFC2507] also defines a header request mechanism but as our results show, link layer error recovery would be more appropriate on this wireless link.

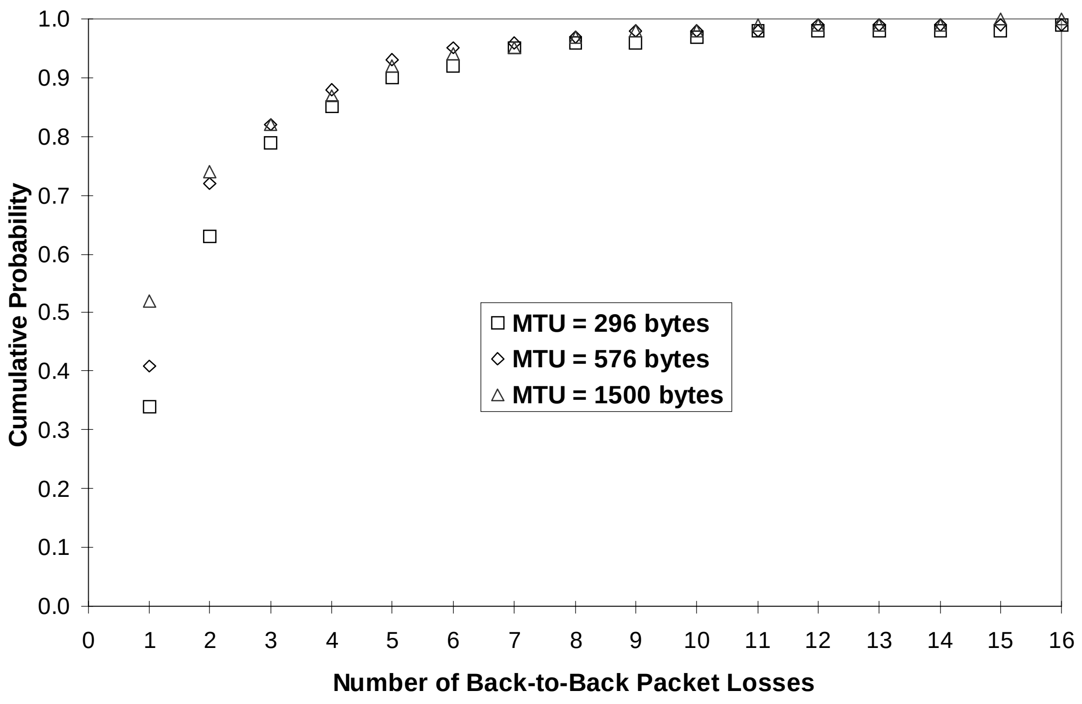


Figure 4-15: Distribution of back-to-back packet losses.

1. In this case, we assume that the TCP/IP header is compressed to 6 bytes. Although compressed TCP/IP headers are typically 4 bytes long, a network-limited TCP connection drops one packet - in the ideal case - per congestion avoidance cycle. This causes one packet to be sent with a full header (40 bytes), and 2 packets - after the packet loss and after the retransmission - to be sent with a compressed header of 7 bytes. Given the pipe capacity of a GSM-CSD link this leads to an average of about 6 bytes.

One could argue in favor of pure end-to-end error recovery by requiring the wireless link's MTU to be set to small values. Transport protocols like TCP could then use the *MSS option* (see Section 2.2.1) or *path MTU discovery* [RFC1191] to adapt the path's MTU accordingly. However, that does not work when the link's end points (e.g., the PPP peers) are not "aware" that the link includes a wireless segment as in GSM-CSD (see Figure 2-7). Also, the path's MTU cannot be re-negotiated during a connection in current transport protocols[1] when the wireless link's error characteristics change.

Link layer error recovery does not have these problems. It is independent of MTU sizes and also interworks well with IP header compression. Future systems favor link layer error recovery even more. Weaker FEC schemes are being deployed[2], which further decrease the throughput optimal frame size on those wireless links. Also, IPv6 requires a minimum MTU of 1280 bytes and recommends an MTU of 1500 bytes or more on links such as GSM-CSD.

Another shortcoming of pure end-to-end error recovery is that each retransmission has to traverse the entire path. This is depicted in Figure 4-16 for *trace_C*, showing the number of retransmissions (as a fraction of the overall number of transmissions) that are required for a range of different MTU sizes. The commonly used MTU size of 1500 and 576 bytes would cause 18 and 12 percent retransmissions, respectively. Such flows impose an unfair load on a best-effort network, such as the Internet, and also on shared wireless access links. Apart from

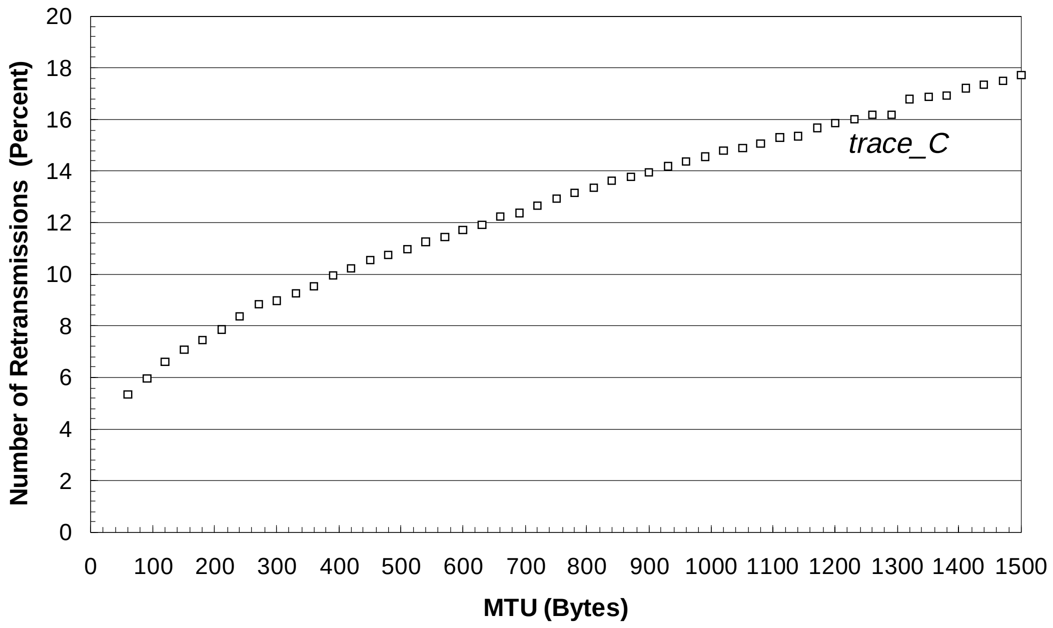


Figure 4-16: Number of end-to-end retransmissions.

1. Implementing such a mechanism would also be a poor design choice as optimizing a link's frame length is not an end-to-end issue.
2. Weaker FEC schemes are used in the new GSM-CSD service [GSM04.21] and the GSM packet-switched data service [GSM05.03].

fairness, a higher fraction of retransmissions also decreases the end-to-end throughput if the corresponding packets had already traversed the bottleneck link regardless of where it is located in the path. This is a common situation when, e.g., data is downloaded from the Internet and the last-hop is an unreliable wireless link. End-to-end error recovery complemented with link layer error recovery running over the wireless link "typically" does not require a single end-to-end retransmission as we have shown in Section 4.2.

4.4 Summary

In this chapter, we introduced the concept of flow-adaptive wireless links that provides service differentiation by adapting link layer error control. The key advantages of our solution is its independence from transport (or higher) layer protocol semantics, and the possibility of co-existence with IPsec. Through measurements in GSM-CSD, we have validated the concept, and show that for fully-reliable flows our solution eliminates all known inefficient cross-layer interactions except for the problem of competing error recovery. In a first study of this problem, we find that even long transient delays on the end-to-end path, do not trigger spurious timeouts in TCP. This made us suspicious that something was wrong with the retransmission timer implemented in TCP-Lite. In Chapter 5, we therefore study in detail both the problem of spurious retransmissions in TCP and TCP-Lite's retransmission timer.

We showed that pure end-to-end error recovery, i.e., when the wireless link is *not* protected by link layer error recovery, fails as a general solution for optimizing end-to-end throughput when wireless links form parts of the path. Fundamental problems are that the path's MTU is often too large to yield efficient error recovery, and that network end-points are not capable of dynamically adapting their MTU to changing local error characteristics on (possibly multiple) wireless links. In many cases, this leads to decreased end-to-end throughput, an unfair load on a best-effort network, such as the Internet, and a waste of valuable radio resources. In fact, we show that highly persistent link layer error recovery over wireless links is essential for fully-reliable flows to avoid these problems.

Our results also suggest that the GSM-CSD channel is over-protected with FEC, and that the default error recovery persistency standardized for RLP is too low. We show that the throughput of the GSM-CSD channel can be improved by up to 25 percent by increasing the (fixed) RLP frame size to reduce the relative per packet overhead. These results highlight the importance of measurement-based analysis of protocol performance over wireless links.

CHAPTER 5

TCP-Eifel

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

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

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

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

5.1 Problems of TCP-Lite's Error Recovery

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

5.1.1 Spurious Timeouts

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

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

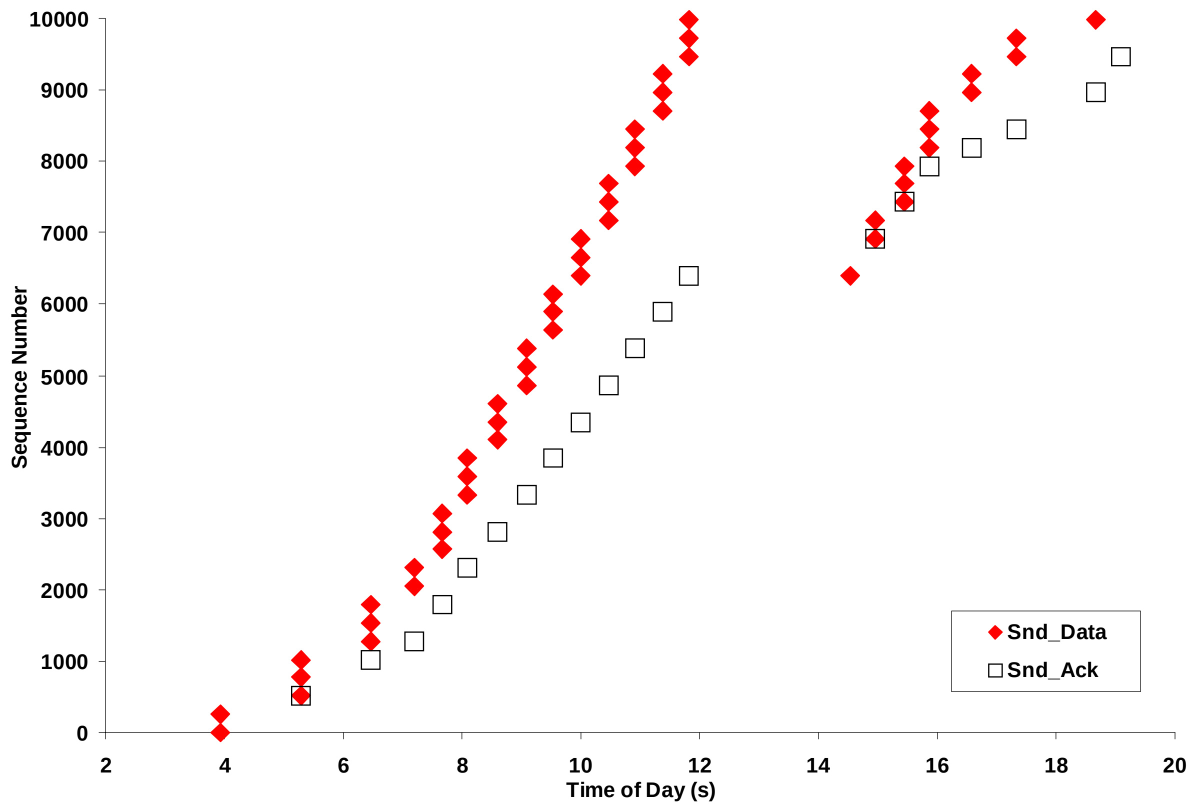


Figure 5-1: Go-back-N after a spurious timeout (sender trace).

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

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

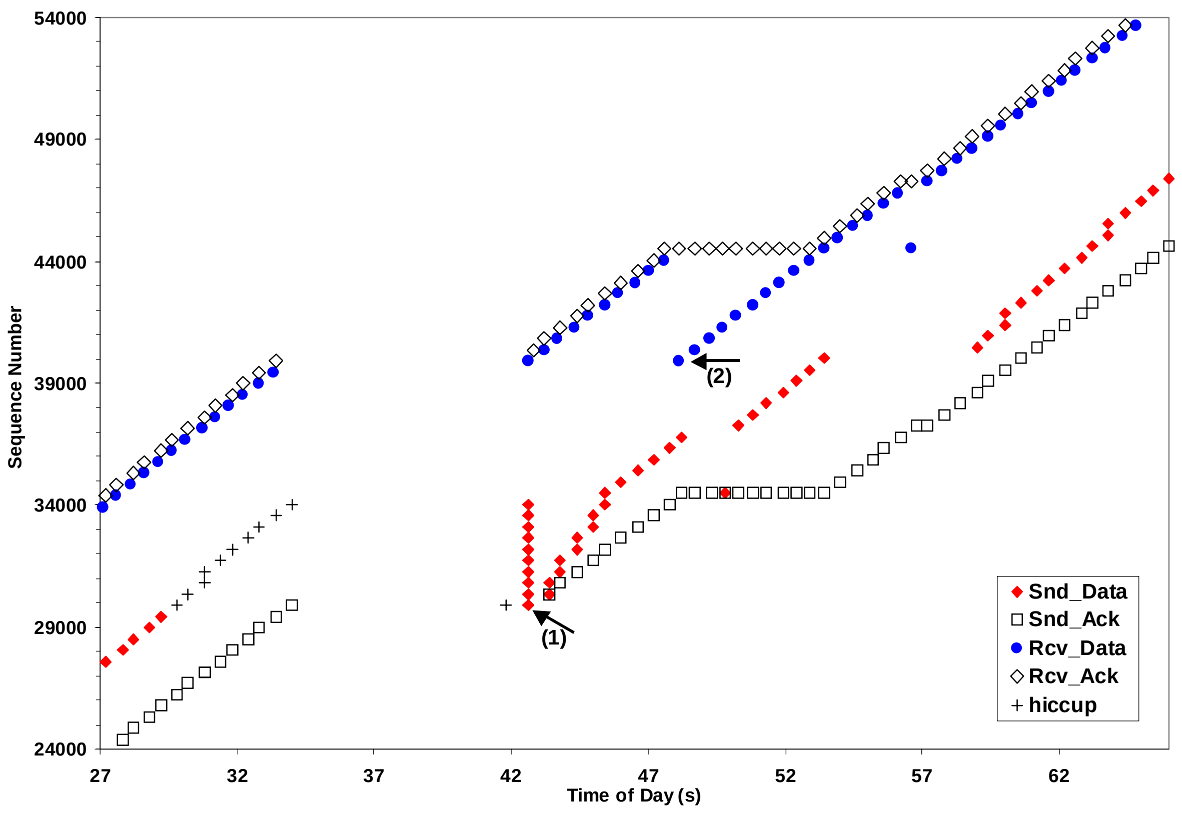


Figure 5-2: Go-back-N after a spurious timeout (sender and receiver traces)

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

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

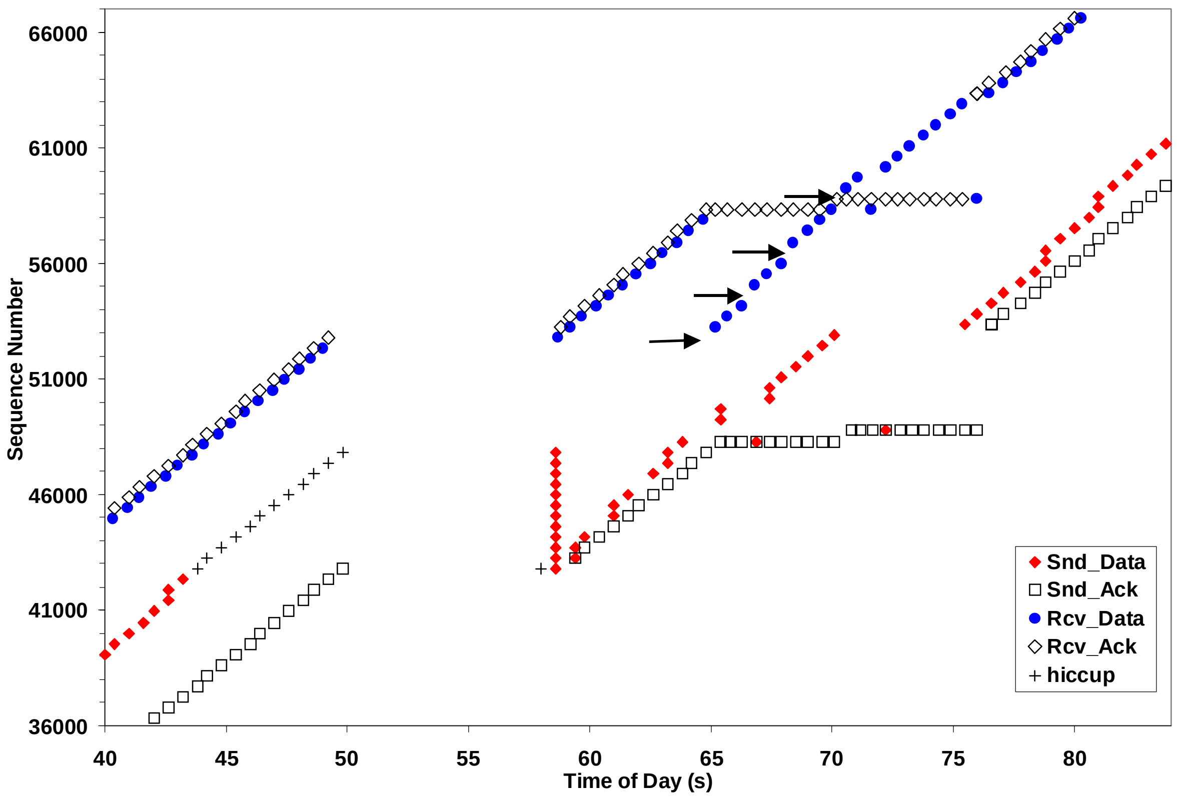


Figure 5-3: Go-back-N after a spurious timeout causing a *real* packet drop.

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

5.1.2 Spurious Fast Retransmits

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

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

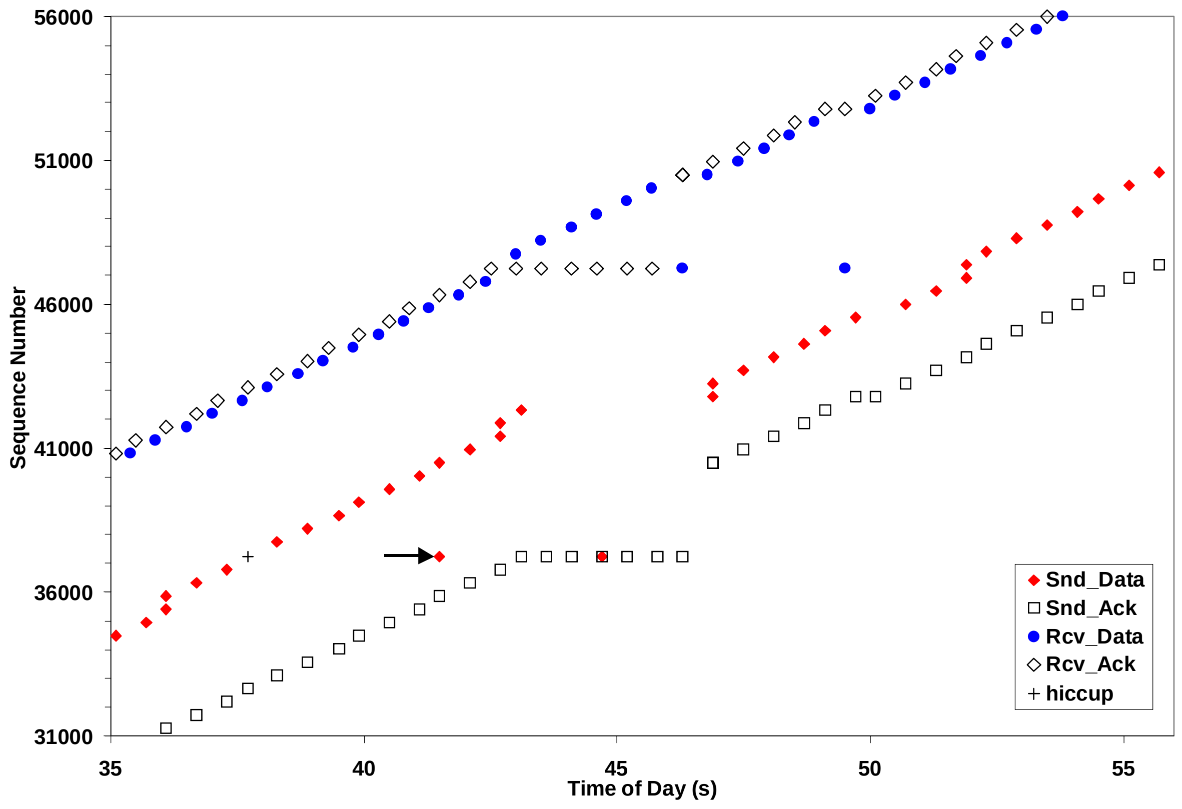


Figure 5-4: The effect of packet re-ordering.

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

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

5.2 The Eifel Algorithm

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

5.2.1 Resolving the Retransmission Ambiguity

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

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

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

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

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

5.2.2 The Sender’s Response

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

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

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

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

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

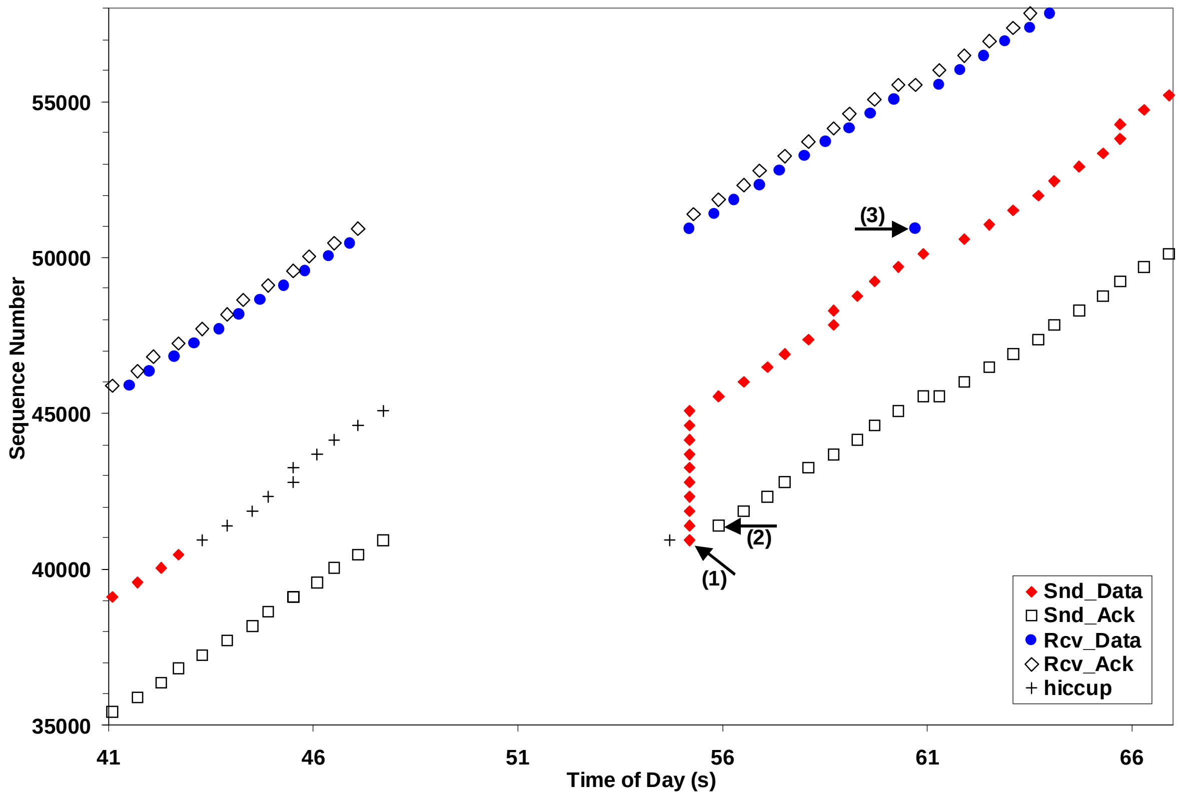


Figure 5-5: Response after spurious timeout.

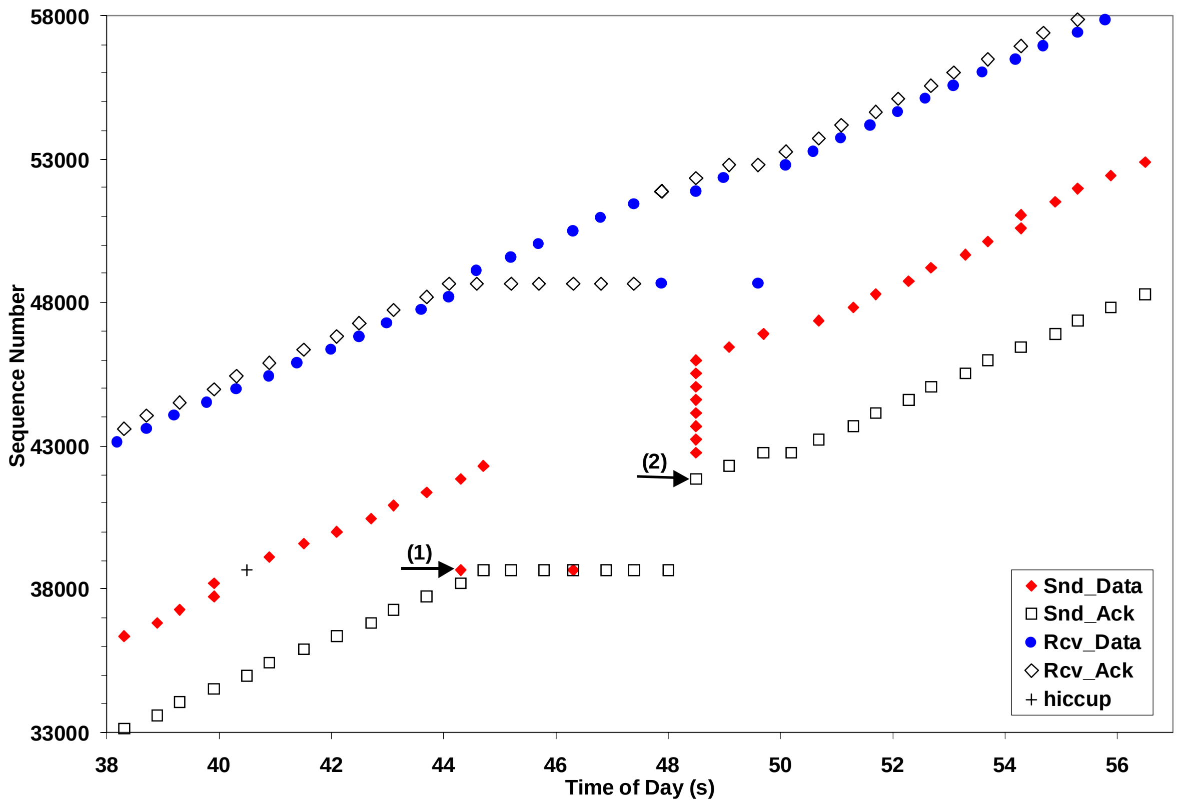


Figure 5-6: Response after a spurious fast retransmit.

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

5.2.3 Performance Evaluation

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

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

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

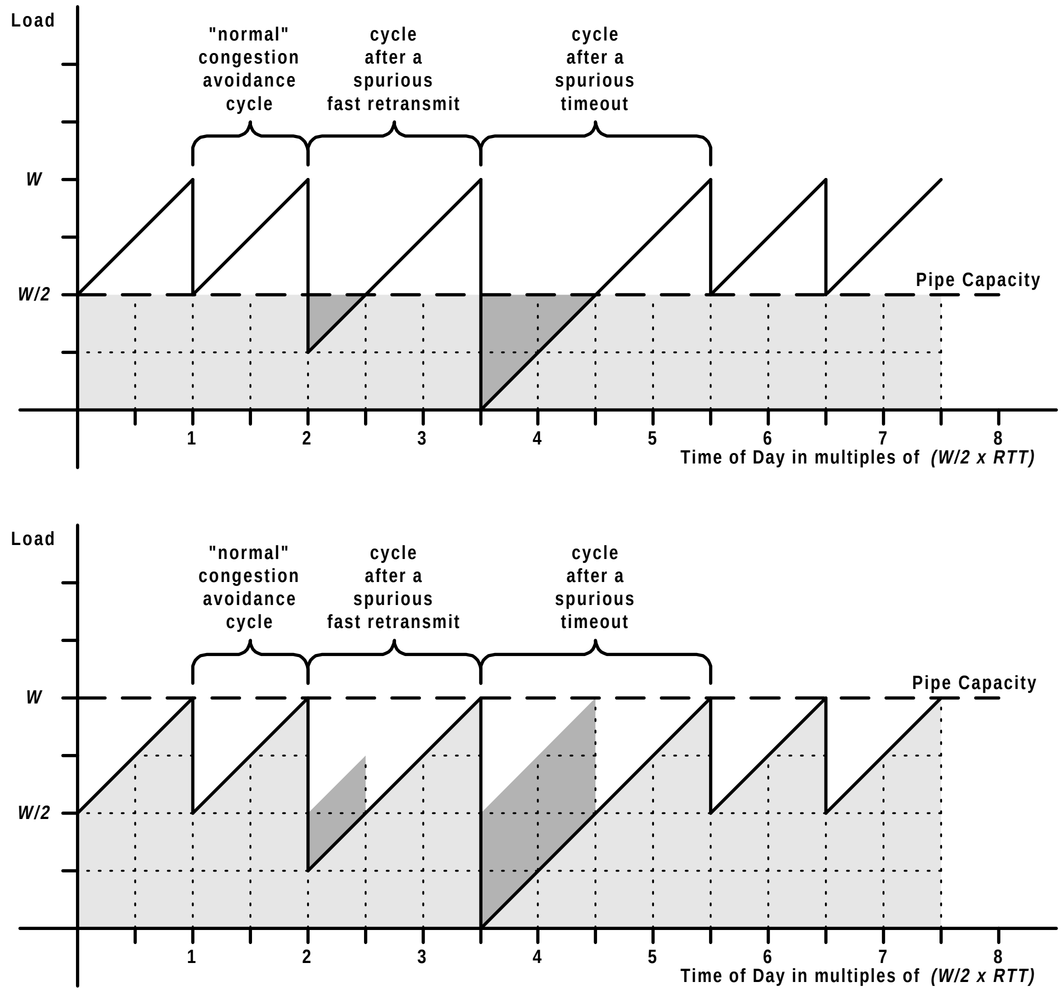


Figure 5-7: Approximating the performance loss.

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

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

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

5.3 Problems of TCP-Lite's Retransmission Timer

In this section, we explain four major problems of the Lite-Xmit-Timer. The first two are fundamental flaws in the definition of RTO_L while the latter two concern the implementation of $REXMT_L$. While the first, third, and fourth problems make the Lite-Xmit-Timer more conservative, the second problem makes it more aggressive. However, the latter is usually outweighed by the other three factors.

5.3.1 Prediction Flaw when the RTT Drops

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

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

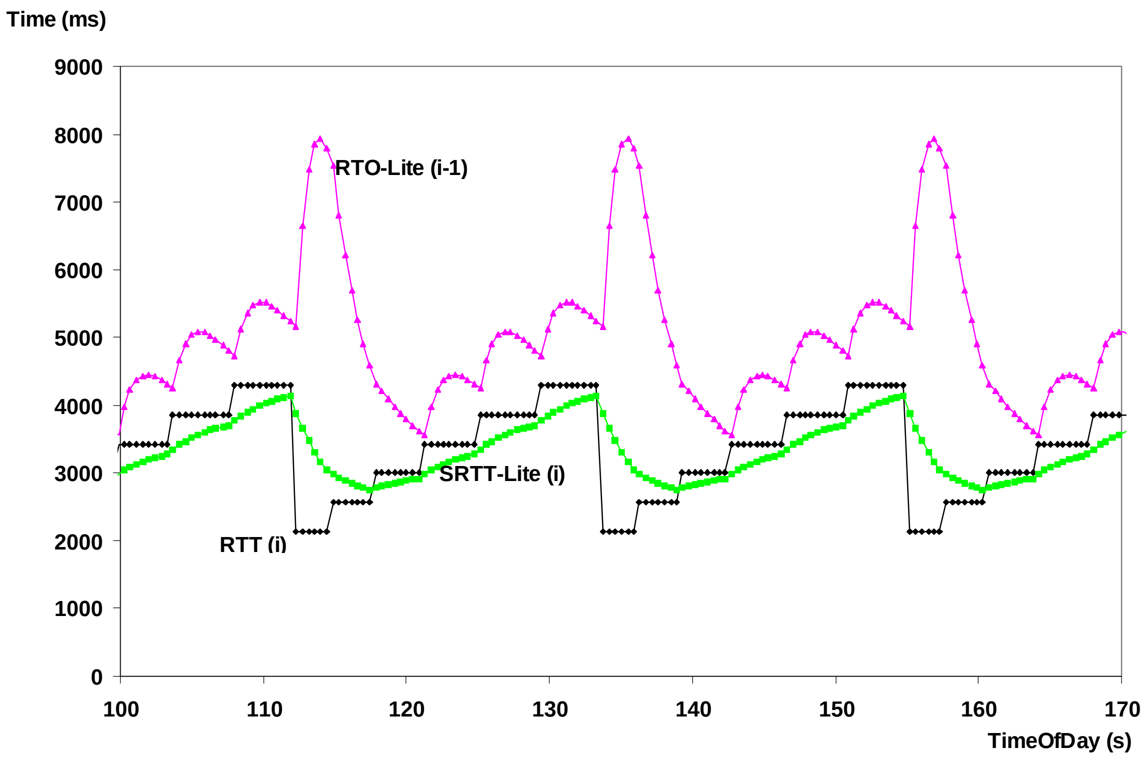


Figure 5-8: Prediction Flaw in *RTO_L*.

5.3.2 Failure of the "Magic Numbers"

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

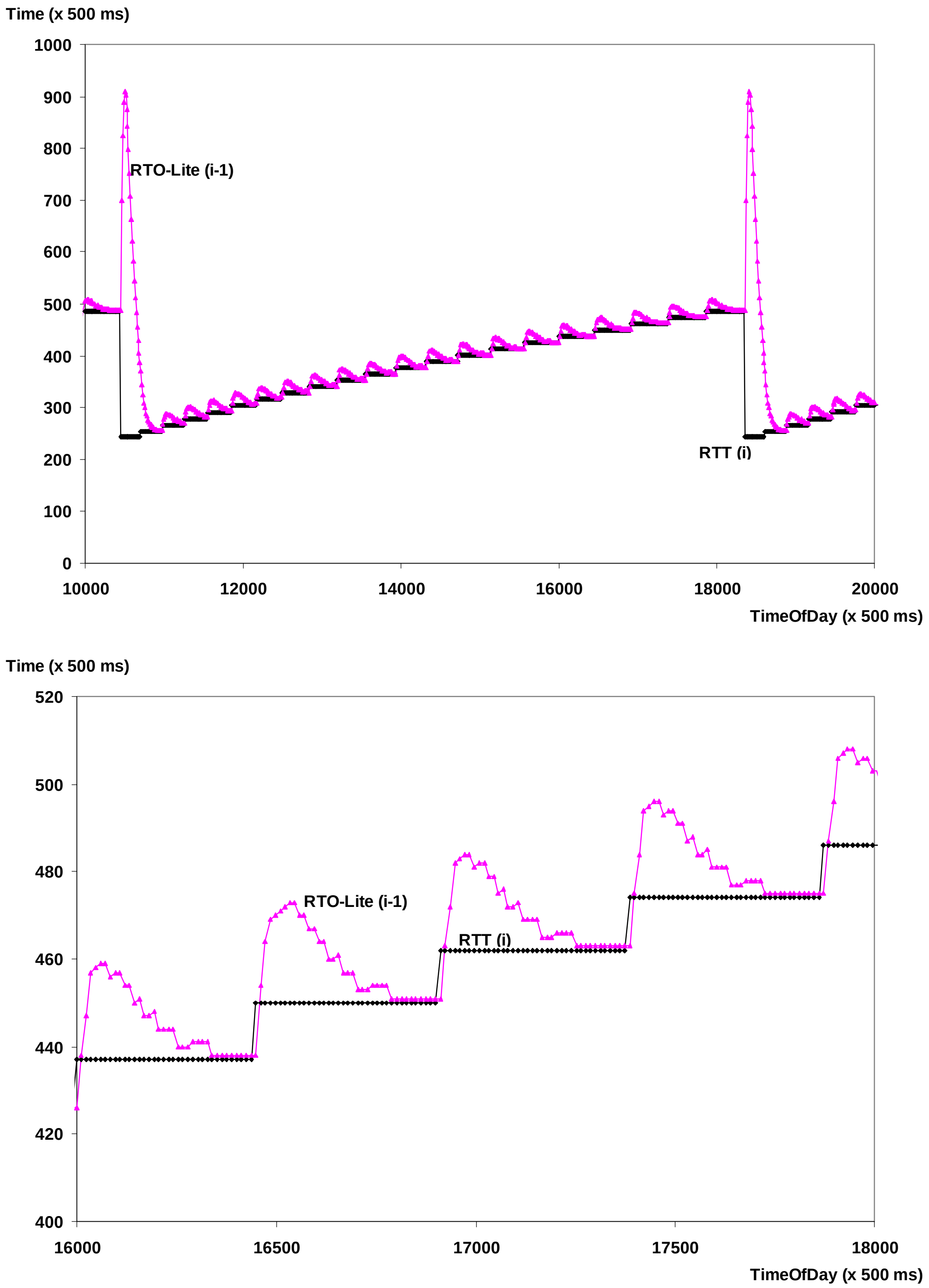


Figure 5-9: A Collapsed RTO_L (model).

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

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

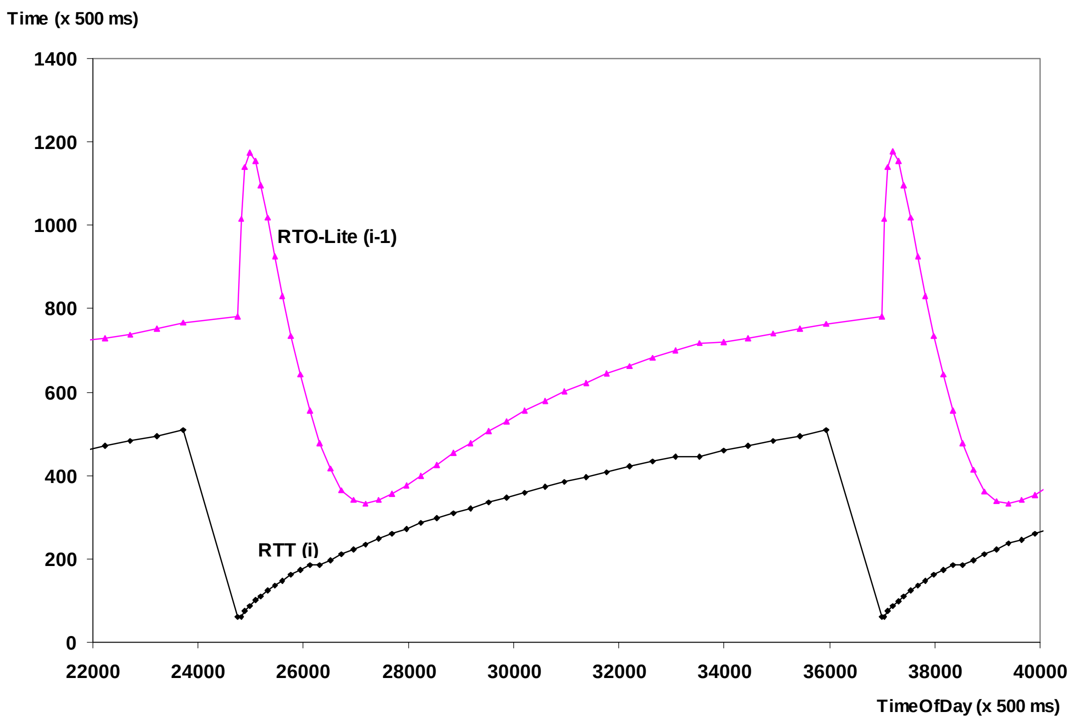


Figure 5-10: RTO_L when timing one segment per RTT.

5.3.3 The "REXMT-Restart Bug"

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

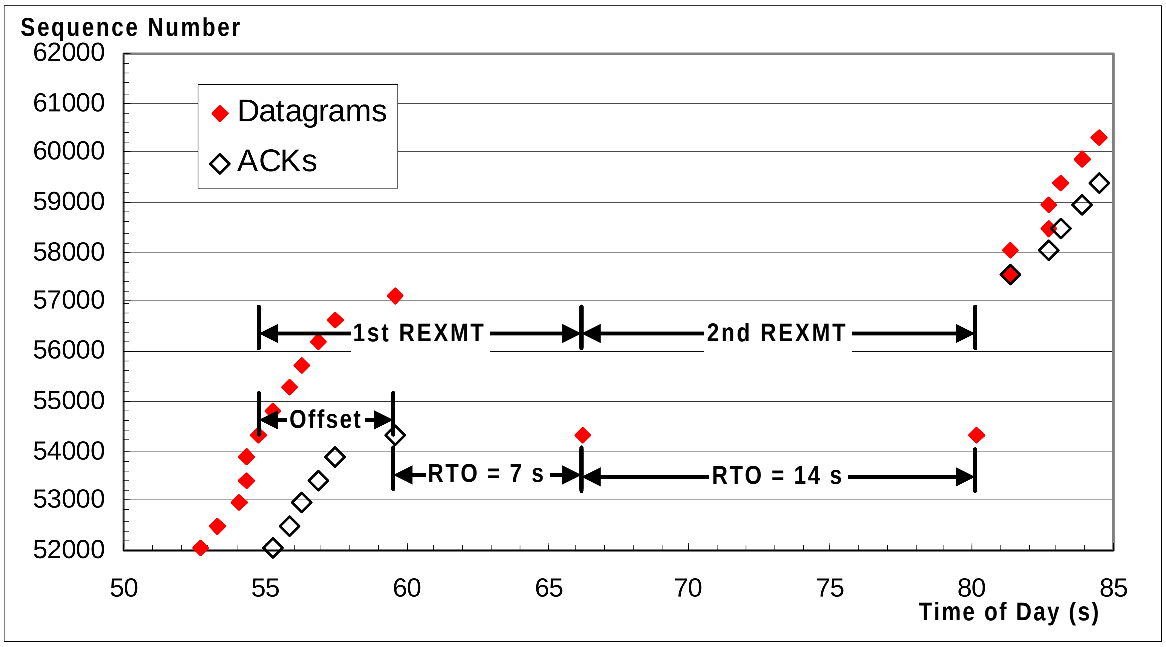


Figure 5-11: The “REXMT-Restart Bug”.

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

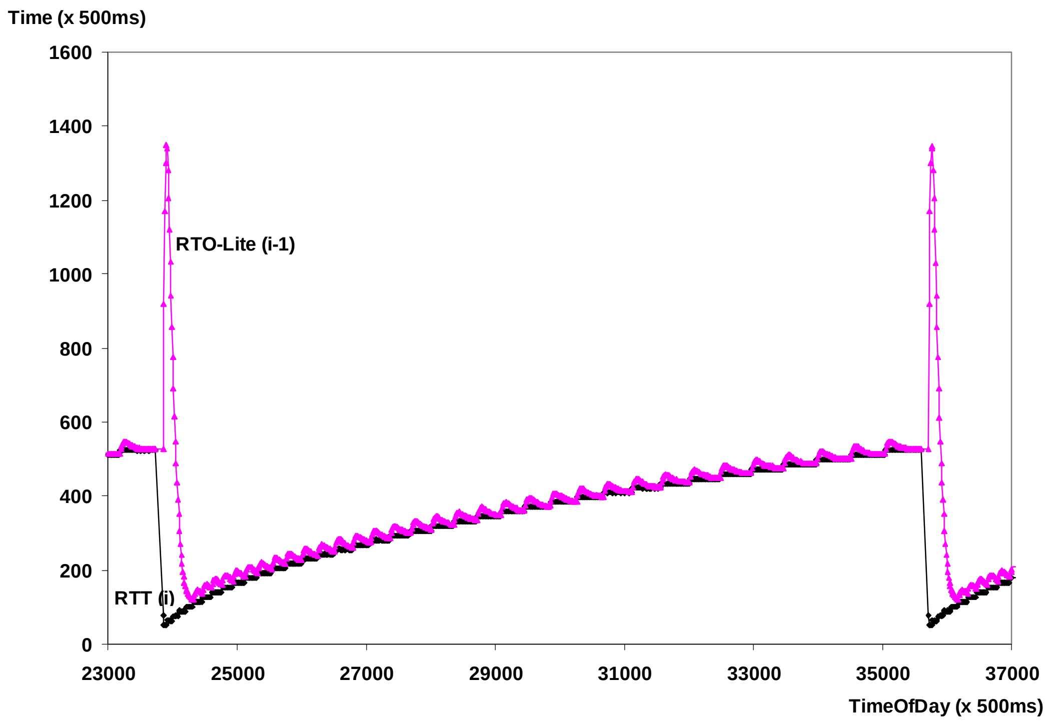
5.3.4 Timer Granularity

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

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

5.3.5 Validating the Model

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



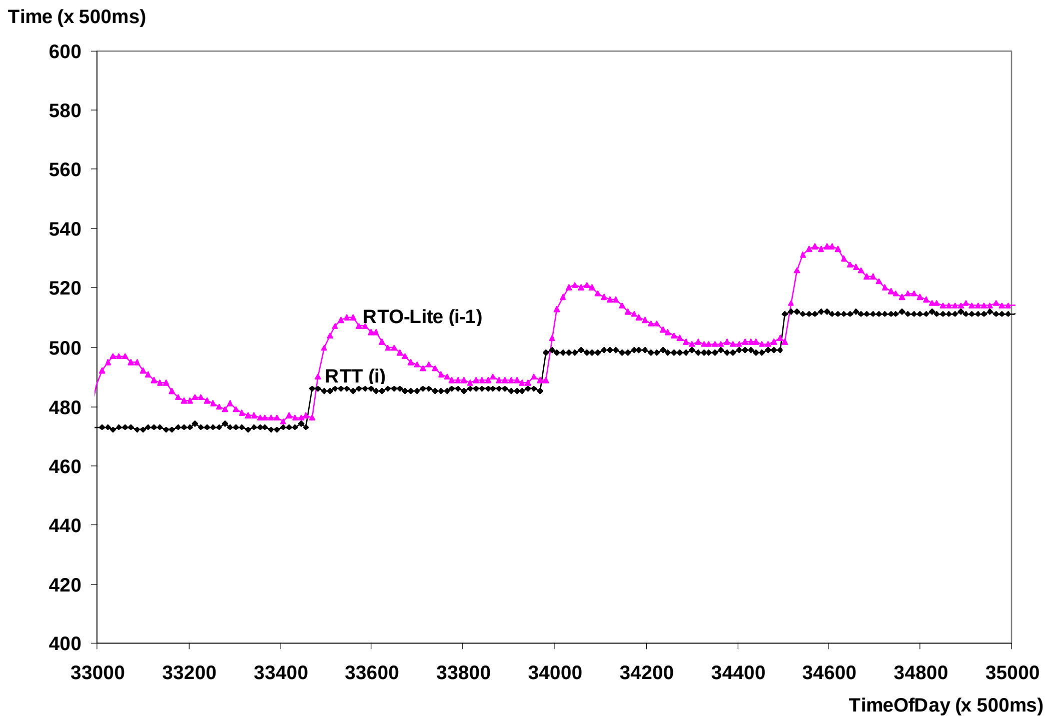


Figure 5-12: A Collapsed RTO_L (measured).

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

5.4 The Eifel Retransmission Timer

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

$$
\begin{aligned}
DELTA_E &= RTT_{Sample} - SRTT_E \\
FLIGHT_E &= \mathrm{MAX}\left(SSTHRESH, \frac{CWND}{2}\right) + 1 \\
GAIN_E &= \begin{cases} \dfrac{1}{FLIGHT_E}, & \text{if RTT Sampling Rate } = 1 \\ \dfrac{2}{FLIGHT_E}, & \text{if RTT Sampling Rate } = \dfrac{1}{2} \\ \dfrac{1}{3}, & \text{if 1 RTT Sample per RTT} \end{cases} \\
\overline{GAIN_E} &= \begin{cases} GAIN_E, & \text{if } (DELTA_E - RTTVAR_E) \geq 0 \\ GAIN_E{}^2, & \text{if } (DELTA_E - RTTVAR_E) < 0 \end{cases} \\
SRTT_E &= SRTT_E + GAIN_E \times DELTA_E \\
RTTVAR_E &= \begin{cases} RTTVAR_E + \overline{GAIN_E} \times (DELTA_E - RTTVAR_E), & \text{if } DELTA_E \geq 0 \\ RTTVAR_E, & \text{if } DELTA_E < 0 \end{cases} \\
RTO_E &= \mathrm{MAX}\left(\left(SRTT_E + \frac{1}{GAIN_E} \times RTTVAR_E\right), RTT_{Sample} + (2 \times ticks)\right)
\end{aligned}
$$

5.4.1 Predicting a Decreasing RTT

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

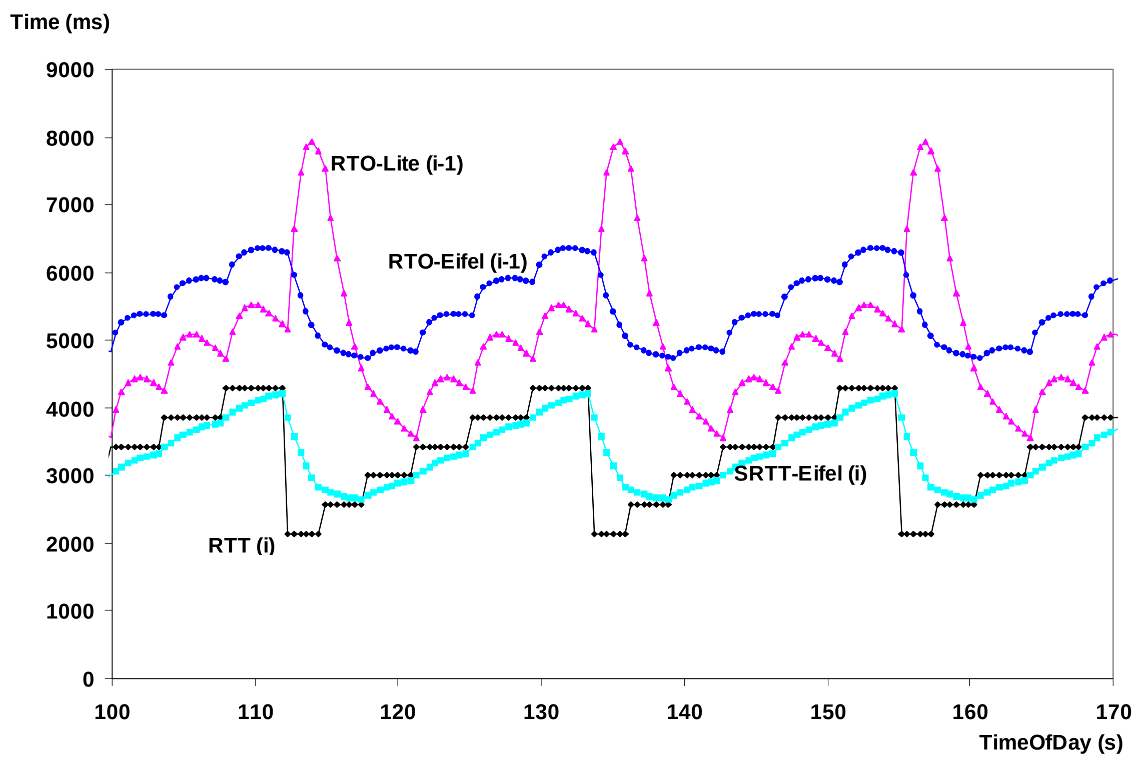


Figure 5-13: Fixing the Prediction Flaw with $RTTVAR_E$.

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

5.4.2 Scaling the Estimator Gains and the Variation Weight

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

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

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

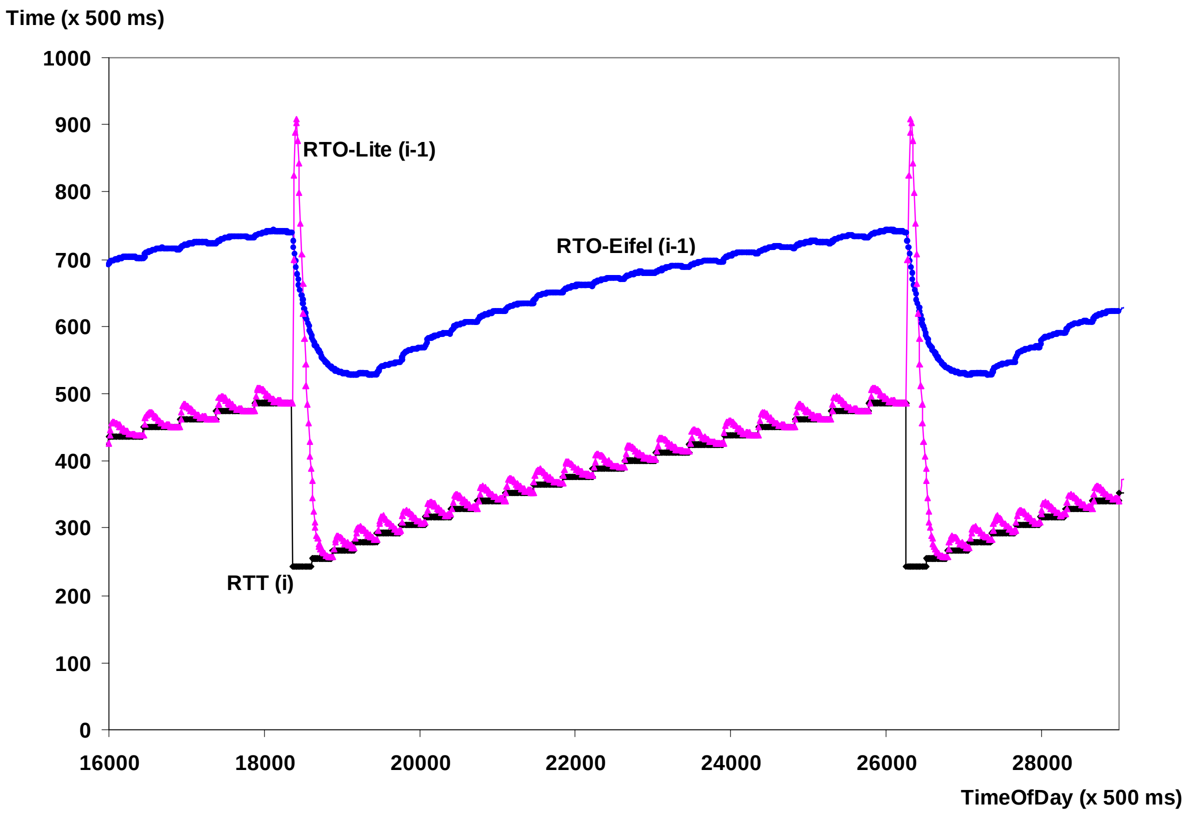


Figure 5-14: RTO_E scales with the flow's load (model).

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

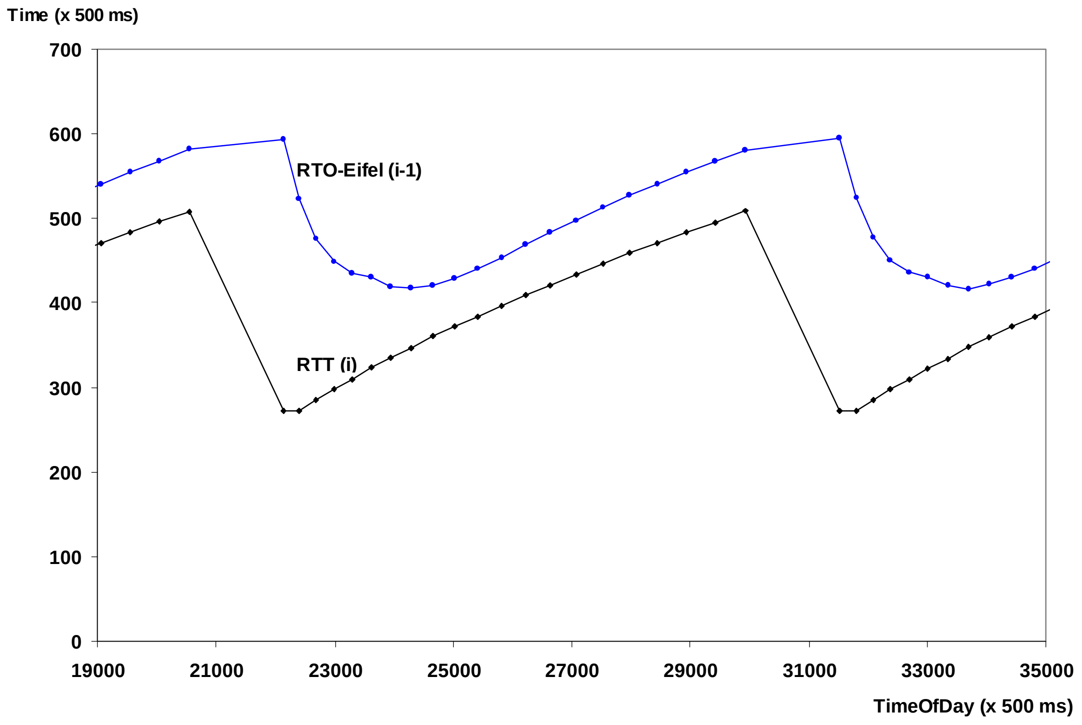


Figure 5-15: RTO_E when timing one segment per RTT.

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

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

5.4.3 Shock Absorbers

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

5.4.4 The RTO Minimum

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

5.4.5 Implementing REXMT Precisely

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

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

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

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

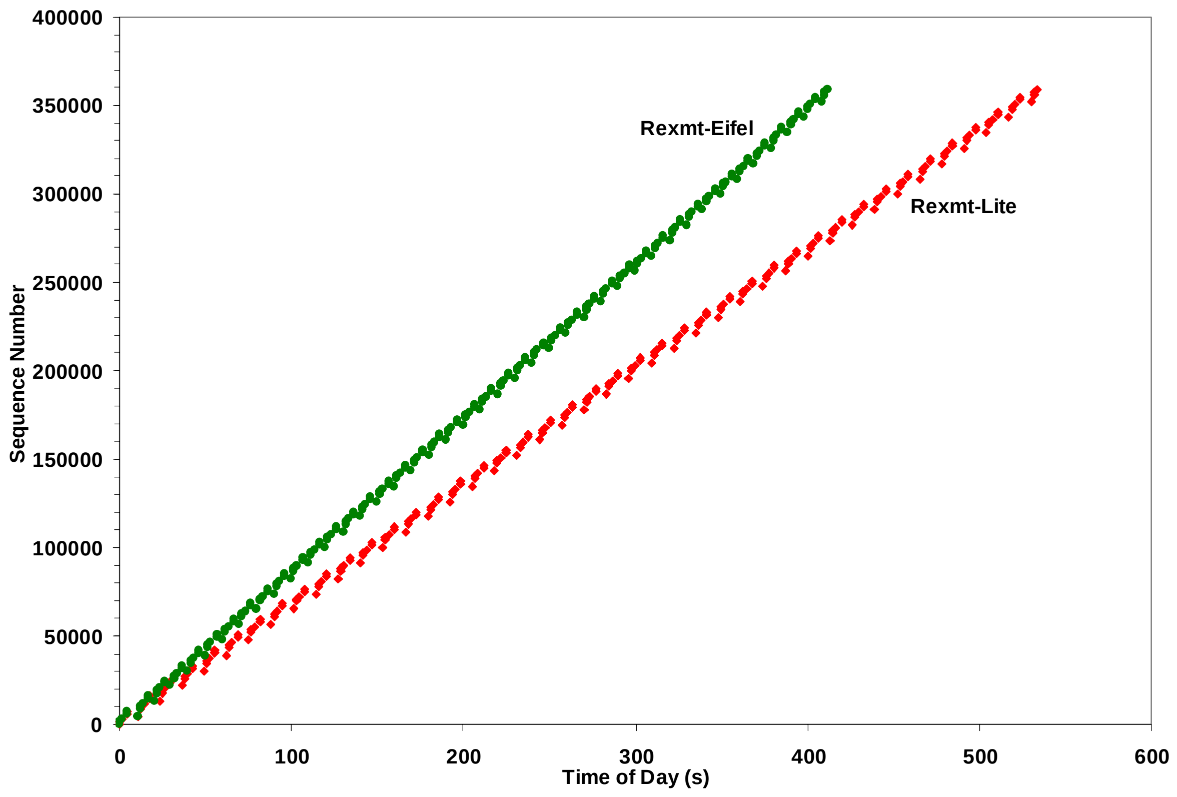
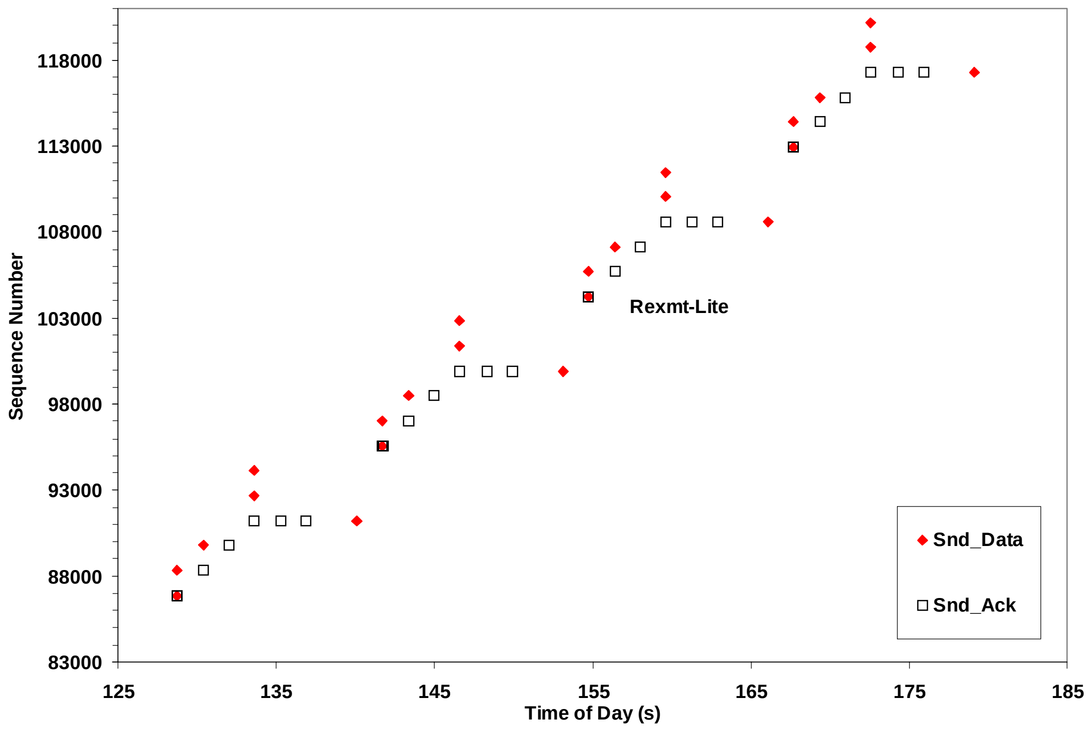


Figure 5-16: Restarting $REXMT_E$ precisely.

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



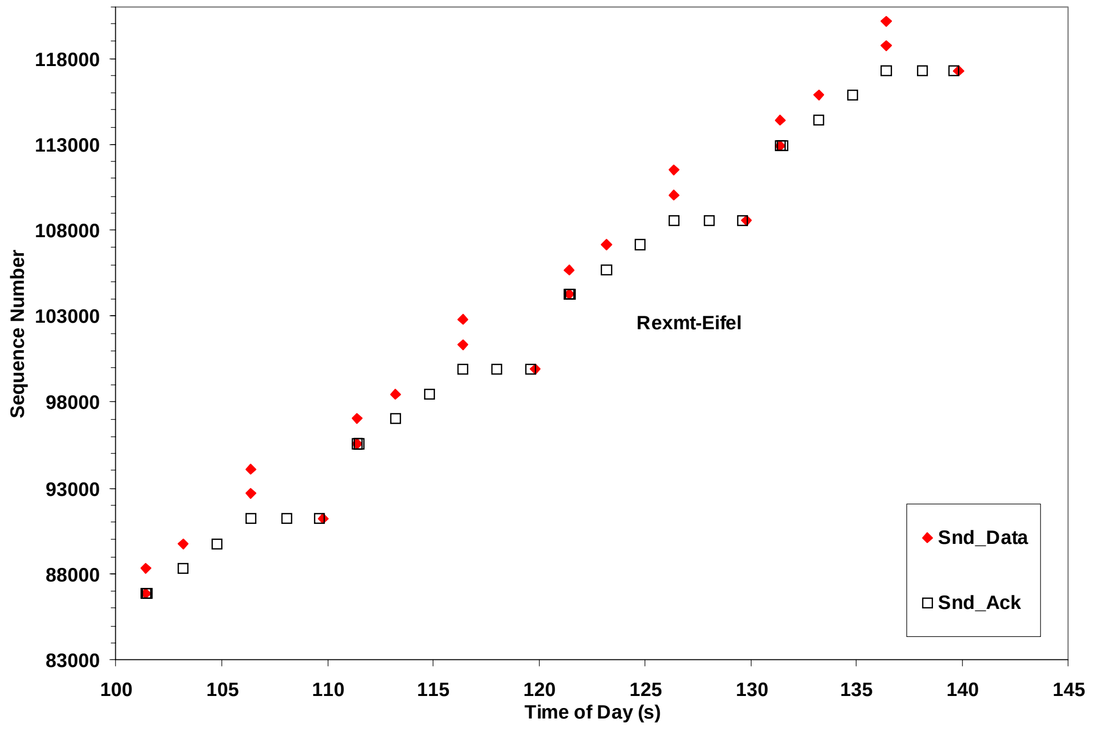


Figure 5-17: Zoom of the graphs shown in Figure 5-16.

5.4.6 Adapting to Spurious Timeouts

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

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

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

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

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

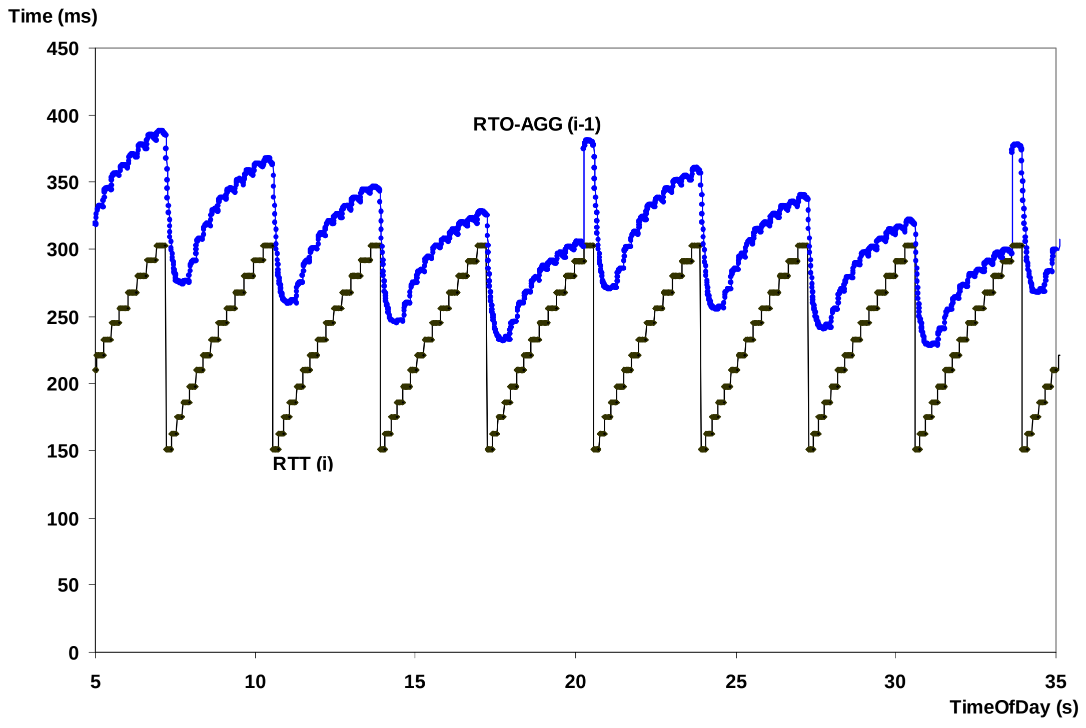


Figure 5-18: A self-trained RTO.

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

5.4.7 Validating the Implementation of RTO-Eifel

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

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

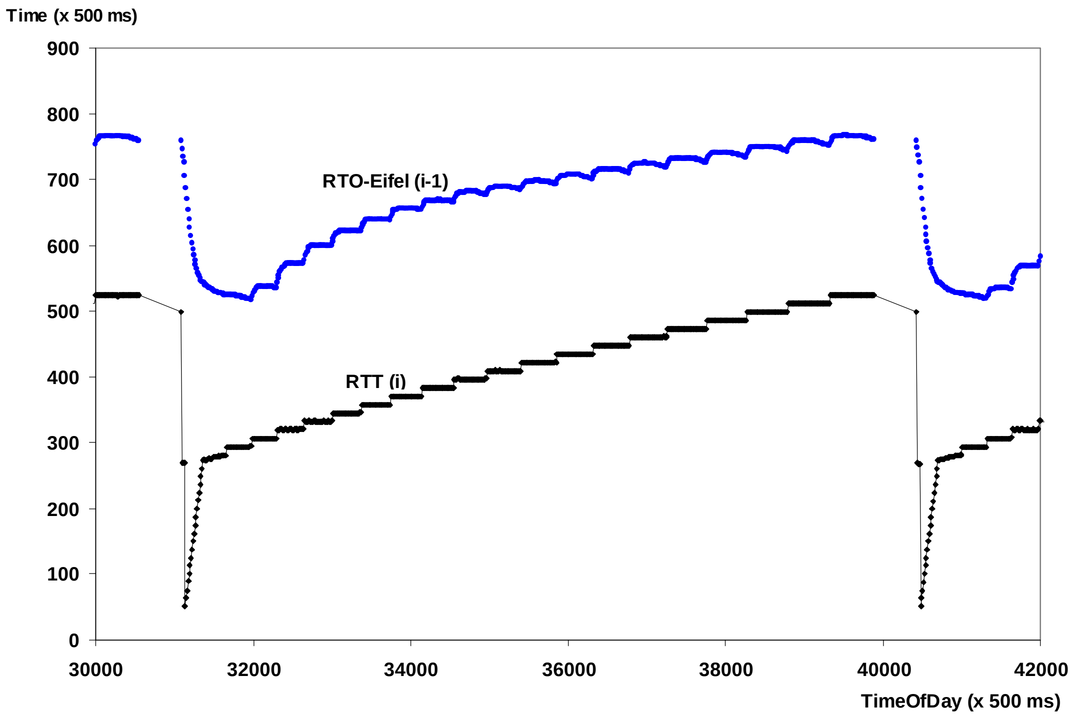


Figure 5-19: RTO_E scales with the flow's load (measured).

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

5.5 Summary

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

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

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

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

CHAPTER 6

Conclusion

A commonly used link layer design philosophy says “leave the link layer dumb but simple”. The term “dumb” in this context stands for independence from any kind of higher layer information about a flow’s nature[1]. This approach has been explicitly applied to the design of many wireless networks that have been deployed in recent years, e.g., IS-95 [Kar93] and WLAN [IEEE802.11]. The “end-to-end argument” [SRC84] is usually quoted to advocate this design philosophy. In a nutshell, the reasoning is that appropriate error control can *only* be implemented on an end-to-end basis, since only the network end-points have sufficient information to perform this task. Despite its attractiveness, we showed that the simplicity of such link layers comes at a cost: reduced end-to-end performance due to inefficient cross-layer interactions and a waste of radio resources. Earlier work addressing these problems either fails to eliminate all inefficiencies, or resorts to Performance Enhancing Proxies (PEPs) that are dependent on transport (or higher) layer protocol semantics and cannot interoperate with network layer encryption as long as they are untrusted.

The concept of *flow-adaptive wireless links* is a new link layer design philosophy we have proposed. It eliminates all known inefficient cross-layer interactions, with the exception of the problem of competing error recovery which we solved with the *Eifel algorithm*. We argued why carrying a network end-point’s QoS requirements as part of the flow’s packet headers and accordingly adapting link layer error control, is orthogonal to the “end-to-end argument”. Moreover, our solution has the key advantage that it avoids PEPs and their drawbacks. We showed the feasibility of our solution by applying it for the class of fully-reliable flows. We demonstrated that highly persistent link layer error recovery is required to optimize the end-to-end performance provided by such flows while efficiently utilizing radio resources.

1. TCP/IP header compression is the exception.

Our measurement results revealed that the GSM-CSD wireless link is over-protected with forward error correction, and that the default value standardized for the link layer error recovery persistency is too low. We also demonstrated that the throughput of the GSM-CSD wireless link can be improved by up to 25 percent by increasing the (fixed) link layer frame size to reduce the relative per packet overhead. These results highlight the importance of measurement-based analysis in wireless networking. It would have been difficult to obtain those results through simulations. The reason is that such an analysis is highly dependent on the error characteristics of the wireless link that are difficult to model with sufficient accuracy. We therefore believe that for wireless systems it is particularly important that prototypes are developed early in the design process so that measurement-based performance studies can be carried out.

We proposed two new mechanisms, the Eifel algorithm and the Eifel retransmission timer, that improve the performance of end-to-end error recovery protocols independent of whether they are run over wireless links. Both mechanisms have been implemented and evaluated on the basis of TCP. The implementation, TCP-Eifel, is publicly available [Lud99c], as well as the spreadsheet-based model we developed for the timer analysis [Lud99a].

The Eifel algorithm uses extra information in the TCP header to eliminate the problems caused by competing error recovery. It only requires changes to the TCP sender implementation, and can be incrementally deployed. In situations where competing error recovery is likely to occur, e.g., in packet-radio networks where the wireless connectivity is often intermittent, the algorithm can improve the end-to-end throughput up to several tens of percent. Another key novelty is that the Eifel algorithm provides for the implementation of optimistic retransmission timers, because it reduces the penalty of underestimating the round-trip time to a single spurious retransmission (in the common case).

The Eifel retransmission timer takes advantage of this feature by becoming increasingly optimistic while adapting to the measured fraction of spurious timeouts. In addition, it eliminates four major problems of the current de facto implementation of TCP's retransmission timer which we have revealed in our work. We demonstrated that the Eifel retransmission timer is a more precise predictor of an upper bound for a path's round-trip time while reacting quicker to packet losses. For network-limited bulk data transfers that do not have enough packets in flight to trigger TCP's fast retransmit algorithm, the Eifel retransmission timer increased the end-to-end throughput by 30 percent in our measurement setup. We validated the correctness of our analysis by showing that the measurements based on the timer implementations yielded the same results that we had predicted based on our timer models.

Optimizing the end-to-end performance of loss responsive *real-time* flows over paths that include wireless links remains an open research problem. Although, we argued that the concept of flow-adaptive wireless links provides an appropriate framework that accommodates such flows, we have not further studied this problem. We are also not aware of related work that addresses it. Below, we outline two issues that need to be resolved in this context.

The performance of loss responsive real-time flows can suffer from the same, and potentially more, inefficient cross-layer interactions that we described in our work. Thus, it needs to be studied how to appropriately adapt link layer error control schemes to minimize such inefficiencies. For example, “how much delay may link layer error control introduce before end-to-end delay bounds are exceeded?”, or “which error loss rates can the network end-point tolerate before it starts underestimating the available bandwidth?”.

Another fundamental problem is the lack of support for bit-error-resilient audio/video codecs. While these are state-of-the-art in traditional circuit-switched wireless networks like GSM, they cannot be used in today’s Internet. The problem is that both transport *and* link layer protocols implemented in the Internet perform error detection, i.e., packets received by an Internet application is always bit-error-free. This effectively disables the key feature of a bit-error-resilient codec, and can greatly reduce user-perceived (audio/video) quality. Solving this problem requires that a transparent service is provided both end-to-end and at the link layer. Yet, transport protocols that use weak checksums, e.g., UDP, rely on strong link layer error detection. Thus, network end-points need to be able to request or disable such a transparent service.

We believe these problems merit further exploration because finding feasible solutions is crucial for true “IP over wireless” to become a reality.

APPENDIX A

Glossar

A

ACK	Positive ACKnowledgement	10
ARQ	Automatic Repeat reQuest	7

B

BER	Bit Error Rates	24
BLER	BLock Erasure Rate	46
BPF	BSD Packet Filter	52
BSC	Base Station Controller	24
BSD	Berkeley Software Distribution	10
BTS	Base Transceiver Station	24

C

CE	Congestion Experienced	21
CSD	Circuit-Switched Data	3
CWND	congestion window	18

D

DS Field	Differentiated Service Field	68
DUPACK	DUPlicate ACKnowledgement	10

E

ECN	Explicit Congestion Notification	21
ECT	ECN-Capable Transport	21
ELN	Explicit Loss Notification	38

F

FEC	Forward Error Correction	28

G

GSM	Global System for Mobile communications	3

I

ICMP	Internet Control Message Protocol	38
IETF	Internet Engineering Task Force	43
IP	Internet Protocol	1
IPsec	Secure IP	43
IPv4	IP-Version 4	5
IPv6	IP-Version 6	5
ISP	Internet Service Provider	25
IWF	Interworking Function	24

L

L2R	Layer 2 Relay	25

M

MS	Mobile Station	24
MSC	Mobile Switching Centre	24
MSL	Maximum Segment Lifetime	10
MSS	Maximum Segment Size	10
MTU	Maximum Transmission Unit	8

P

PEP	Performance Enhancing Proxy	2
PPP	Point-to-Point Protocol	8
PSTN	Public Switched Telephone Network	24

R

REXMT	Retransmission Timer State	12
RLP	Radio Link Protocol	25
RTO	Retransmission Timeout Value	12
RTP	Real-Time Transport Protocol	9
RTT	Round Trip Time	6

RTTVAR	Smoothed RTT Deviation Estimator	13

S

SACK	Selective Acknowledgements	37
SRTT	Smoothed RTT Estimator	13
SSTHRESH	Slow Start Threshold	18
SYN	TCP's connect request	52

T

TAF	Terminal Adaptation Function	24
TCP	Transmission Control Protocol	2

U

UDP	User Datagram Protocol	9

W

WLAN	Wireless Local Area Network	39
WWW	World Wide Web	2

APPENDIX B

References

B.1 Research Papers and Books

[AP99] M. Allman, V. Paxson, *On Estimating End-to-End Network Path Properties,* In Proceedings of ACM SIGCOMM 99.

[BB95] A. Bakre, B. R. Badrinath, *I-TCP: Indirect TCP for Mobile Hosts,* In Proceedings of ICDCS 95, pp. 136-143, May 1995.

[BSK95] H. Balakrishnan, S. Seshan, R. H. Katz, *Improving reliable transport and handoff performance in cellular wireless networks,* ACM/Baltzer Wireless Networks Journal, Vol. 1, No. 4, pp. 469-481, December 1995.

[BPSK96] H. Balakrishnan, V. Padmanabhan, S. Seshan, R. H. Katz, *A Comparison of Mechanisms for Improving TCP Performance over Wireless Links,* In Proceedings of ACM SIGCOMM 96.

[BK98] H. Balakrishnan, R. H. Katz, *Explicit Loss Notification and Wireless Web Performance,* In Proceedings of IEEE GLOBECOM 98.

[Bau97] 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.

[BPS99] J.C.R. Bennett, C. Partridge and N. Shectman, *Packet Reordering is Not Pathological Network Behavior,* In IEEE/ACM Transactions on Networking, December 1999.

[BBKT97] P. Bhagwat, P. Bhattacharya, A. Krishna, S. K. Tripathi, *Using channel state dependent packet scheduling to improve TCP throughput over wireless LANs,* ACM/Baltzer Wireless Networks Journal, Vol. 3, No. 1, January 1997.

[BDSZ94] V. Bharghavan, A. Demers, S. Shenker, and L. Zhang, *MACAW: A media access protocol for wireless LANs,* In Proceedings of ACM SIGCOMM 94.

[BV99] S. Biaz, N. H. Vaidya, *Is the Round-trip Time Correlated with the Number of Packets in Flight?*, Computer Science Dept., Texas A&M University, Technical Report 99-006, March 1999.

[BP95a] 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, pp. 1465-1480, October 1995.

[BP95b] L. S. Brakmo, L. L. Peterson, *Performance Problems in BSD4.4 TCP*, ACM Computer Communication Review, Vol. 25, No. 5, October 1995.

[BS97] K. Brown, S. Singh, *M-TCP: TCP for Mobile Cellular Networks*, ACM Computer Communication Review, Vol. 27, No. 5, October 1997.

[CI95] R. Cáceres, L. Iftode, *Imroving the Performance of Reliable Transport Protocols in Mobile Computing Environments*, IEEE Journal of Selected Areas in Communication, Vol. 13, No. 5, pp. 850-857, June 1995.

[CJ89] D.-M. Chiu, R. Jain, *Analysis of Increase and Decrease Algorithms for Congestion Avoidance in Computer Nerwork*, Computer Networks and ISDN Systems, Vol. 17, 1989.

[CLZ87] D. D. Clark, M. L. Lambert, L. Zhang, *NETBLT: A bulk data transfer protocol*, In Proceedings of ACM SIGCOMM 87.

[DCY93] A. DeSimone, M. C. Chuah, O.-C. Yue, *Throughput Performance of Transport-Layer Protocols over Wireless LANs*, In Proceedings of IEEE GLOBECOM 93.

[DMT96] R. C. Durst, G. J. Miller, E. J. Travis, *TCP Extensions for Space Communications*, In Proceedings of ACM MOBICOM 96.

[ES98] D. A. Eckhardt, P. Steenkiste, *Improving Wireless LAN Performance via Adaptive Local Error Control*, In Proceedings of IEEE ICNP 98.

[FJ94] S. Floyd and V. Jacobson, *The Synchronization of Periodic Routing Messages*, IEEE/ACM Transactions on Networking, Vol. 2, No. 2, April 1994.

[FJ95] S. Floyd, V. Jacobson, *Link-sharing and Resource Management Models for Packet Networks*, IEEE/ACM Transactions on Networking, Vol. 3, No. 4, August 1995.

[FJLMZ97] S. Floyd, V. Jacobson, C. Liu, S. McCanne, L. Zhang, *A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing*, IEEE/ACM Transactions on Networking, Vol. 5, No. 6, December 1997.

[FMMPR99] 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.

[HK99] T. R. Henderson, R. H. Katz, *Transport Protocols for Internet-Compatible Satellite Networks*, IEEE Journal of Selected Areas in Communication, Vol. 17, No. 2, pp. 326-344, February 1999.

[Jac88] V. Jacobson, *Congestion Avoidance and Control*, In Proceedings of ACM SIGCOMM 88.

[Jac90a] V. Jacobson, *Modified TCP Congestion Avoidance Algorithm*, end2end mailing list (ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail), April 30, 1990.

[JK92] V. Jacobson, M. J. Karels, *Congestion Avoidance and Control,* Revised version of [Jac88], 1992.

[Jai89] R. Jain, *A Delay-Based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks*, ACM Computer Communication Review, Vol. 19, No. 5, October 1989.

[KM87] C. A. Kent, J. C. Mogul, *Fragmentation Considered Harmful,* In Proceedings of ACM SIGCOMM 87.

[KP87] P. Karn, C. Partridge, *Improving Round-Trip Time Estimates in Reliable Transport Protocols,* In Proceedings of ACM SIGCOMM 87.

[Kar93] P. Karn, *The Qualcomm CDMA Digital Cellular System,* In Proceedings of the USENIX Mobile and Location-Independent Computing Symposium, pp. 35-39, USENIX Association, August 1993.

[KRLKA97] M. Kojo, K. Raatikainen, M. Liljeberg, J. Kiiskinen, T. Alanko, *An Efficient Transport Service for Slow Wireless Telephone Links,* IEEE Journal of Selected Areas in Communication, Vol. 15, No. 7, pp. 1337-1348, September1997.

[LS98] P. Lettieri, M. B. Srivastava, *Adaptive Frame Length Control for Improving Wireless Link Throughput, Range, and Energy Efficiency,* pp. 564-571, In Proceedings of IEEE INFOCOM 98.

[LNQ97] R. Ludwig, N. Niebert, R. Quinet, *Radio Webs - Support Architecture for Mobile Web Access,* pp. 262-271, European Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services, IDMS 97.

[LT97] R. Ludwig, D. Turina, *Link Layer Analysis of the General Packet Radio Service for GSM,* pp. 525-530, In Proceedings of IEEE ICUPC 97.

[LR99] R. Ludwig, B. Rathonyi, *Link Layer Enhancements for TCP/IP over GSM,* pp. 415-422, In Proceedings of IEEE INFOCOM 99.

[LRKOJ99] R. Ludwig, B. Rathonyi, A. Konrad, K. Oden, A. Joseph, *Multi-Layer Tracing of TCP over a Reliable Wireless Link,* pp. 144-154, In Proceedings of ACM SIGMETRICS 99.

[Lud99b] R. Ludwig, *A Case for Flow-Adaptive Wireless Links,* Technical Report UCB//CSD-99-1053, University of California at Berkeley, May 1999.

[LKJ99] R. Ludwig, A. Konrad, A. D. Joseph, *Optimizing the End-to-End Performance of Reliable Flows over Wireless Links*, pp. 113-119, In Proceedings of ACM/IEEE MOBICOM 99.

[LKJK00] R. Ludwig, A. Konrad, A. D. Joseph, R. H. Katz, *Optimizing the End-to-End Performance of Reliable Flows over Wireless Links,* To appear in ACM/Baltzer Wireless Networks Journal (Special issue: Selected papers from ACM/IEEE MOBICOM 99), revised version of [LKJ99].

[LK00] R. Ludwig, R. H. Katz, *The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions*, ACM Computer Communication Review, Vol. 30, No. 1, January 2000.

[LS99] R. Ludwig, K. Sklower, *The Eifel Retransmission Timer*, Work in progress, available at http://iceberg.cs.berkeley.edu, November 1999.

[MSMO97] M. Mathis, J. Semke, J. Mahdavi, T. Ott, *The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm*, ACM Computer Communication Review, Vol. 27, No. 3, July 1997.

[MJ93] S. McCanne, V. Jacobson, *The BSD Packet Filter: A New Architecture for User-Level Packet Capture*, In Proceedings of the 1993 Winter USENIX Conference.

[Mey99] M. Meyer, *TCP Performance over GPRS*, In Proceedings of IEEE WCNC 99.

[Mog92] J. Mogul, *Observing TCP Dynamics in Real Networks*, In Proceedings of ACM SIGCOMM 92.

[MP92] M. Mouly, M.-B. Pautet, *The GSM System for Mobile Communications*, Cell & Sys, France 1992.

[NSNK97] B. D. Noble, M. Satyanarayanan, G. T. Nguyen, R. H. Katz, *Trace-Based Mobile Network Emulation*, In Proceedings of ACM SIGCOMM 97.

[Pax97a] V. Paxson, *Measurements and Analysis of End-to-End Internet Dynamics*, Ph. D. dissertation, University of California, Berkeley, April 1997.

[Pax97b] V. Paxson, *Automated Packet Trace Analysis of TCP Implementations*, In Proceedings of ACM SIGCOMM 97.

[Pax97c] V. Paxson, *End-to-End Internet Packet Dynamics*, In Proceedings of ACM SIGCOMM 97.

[Pax97d] V. Paxson, *End-to-End Routing Behavior in the Internet*, IEEE/ACM Transactions on Networking, Vol. 5, No. 5, October 1997.

[PILC] The Mailing List of the PILC Working Group (Performance Implications of Link Characteristics) of the Internet Engineering Task Force, http://pilc.grc.nasa.gov/pilc/list/archive/

[SRC84] J. H. Saltzer, D. P. Reed, D. D. Clark, *End-To-End Arguments in System Design*, ACM Transactions on Computer Systems, Vol. 2, No. 4, November 1984.

[SF98] N. K. G. Samaraweera, G. Fairhurst, *Reinforcement of TCP Error Recovery for Wireless Communication*, ACM Computer Communication Review, Vol. 28, No. 2, April 1998.

[She95] S. Shenker, *Fundamental Design Issues for the Future Internet*, IEEE Journal of Selected Areas in Communication, Vol. 13, No. 7, pp. 1176-1188, Sep.1995.

[SFHKK95] O. Spaniol, A. Fasbender, S. Hoff, J. Kaltwasser, J. Kassubek, *Impacts of Mobility on Telecommunication and Data Communication Networks*, IEEE Personal Communications, Vol. 2, No. 5, pp. 20-33, Oktober 1995.

[SFHKK96] O. Spaniol, A. Fasbender, S. Hoff, J. Kaltwasser, J. Kassubek, *Wireless Networks and OSI: New Challenges for Protocol Stack Design*, Applied Microwave & Wireless, Spring 1996.

[Ste94] W. R. Stevens, *TCP/IP Illustrated, Volume 1 (The Protocols),* Addison Wesley, November 1994.

[Tan89] A. Tanenbaum, *Computer Networks*, 2nd Edition, Prentice Hall, 1989.

[TMW97] K. Thompson, G. J. Miller, R. Wilder, *Wide-Area Internet Traffic Patterns and Characteristics,* IEEE Network , Vol. 11, No. 6, November/December 1997.

[WS95] G. R. Wright, W. R. Stevens, *TCP/IP Illustrated, Volume 2 (The Implementation),* Addison Wesley, January 1995.

[ZSC91] L. Zhang, S. Shenker, D. Clark, *Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic*, In Proceedings of ACM SIGCOMM 91.

B.2 Standards, Recommendations, and Drafts

[GSM04.22a] ETSI, *Radio Link Protocol for data and telematic services on the Mobile Station - Base Station System (MS-BSS) interface and the Base Station System - Mobile Switching Center (BSS-MSC) interface*, GSM Specification 04.22, Version 5.0.0, December 1995.

[GSM04.22b] ETSI, *Digital cellular communications system (Phase 2+); Radio Link Protocol for data and telematic services on the Mobile Station - Base Station System (MS-BSS) interface and the Base Station System - Mobile Switching Center (BSS-MSC) interface,* GSM Specification 04.22, Version 6.1.0, November 1998.

[GSM04.21] ETSI, *Digital cellular communications system (Phase 2+); Rate adaption on the Mobile Station - Base Station System (MS - BSS) Interface,* GSM Specification 04.21, Version 7.0.0, October 1998.

[GSM04.60] ETSI, *Digital cellular communications system (Phase 2+); General Packet Radio Service (GPRS); Mobile Station (MS) Base Station System (BSS) interface; Radio Link Control / Medium Access Control (RLC/MAC) protocol,* GSM Specification 04.60, Version 6.1.0, August 1998.

[GSM05.03] ETSI, *Digital cellular communications system (GSM Radio Access Phase 3); Channel coding*, GSM Specification 05.03, Version 6.0.0, January 1998.

[IEEE802.11] IEEE Local and Metropolitan Area Network Standards Committee, *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, IEEE Std 802.11-1997, The Institute of Electrical and Electronics Engineers, New York, 1997.

[ISO8073] ISO/IEC, *Information processing systems - Open Systems Interconnection - Connection oriented transport protocol specification*, International Standard ISO/IEC 8073, December 1988.

[RFC768] J. Postel, *User Datagram Protocol*, RFC 768, August 1980.

[RFC791] J. Postel, *Internet Protocol*, RFC 791, September 1981.

[RFC792] J. Postel, *Internet Control Message Protocol*, RFC 792, September 1981.

[RFC793] J. Postel, *Transmission Control Protocol*, RFC 793, September 1981.

[RFC896] J. Nagle, *Congestion Control in IP/TCP Internet-works*, RFC 896, January 1984.

[RFC1094] Sun Microsystems Inc., *NFS: Network File System Protocol Specification*, RFC 1094, March 1989.

[RFC1122] R. Braden, *Requirements for Internet Hosts - Communication Layers*, RFC 1122, October 1989.

[RFC1144] V. Jacobson, *Compressing TCP/IP Headers for Low-Speed Serial Links*, RFC 1144, February 1990.

[RFC1191] J. Mogul, S. Deering, *Path MTU Discovery*, RFC 1191, November 1990.

[RFC1323] V. Jacobson, R. Braden, D. Borman, *TCP Extensions for High Performance*, RFC 1323, May 1992.

[RFC1349] P. Almquist, *Type of Service in the Internet Protocol Suite*, RFC1349, July 1992.

[RFC1661] W. Simpson, *The Point-to-Point Protocol*, RFC 1661, July 1994.

[RFC1663] D. Rand, *PPP Reliable Transmission*, RFC 1663, July 1994.

[RFC1831] R. Srinivasan, *RPC: Remote Procedure Call Protocol Specification Version 2*, RFC 1831, August 1995.

[RFC1889] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobsen, *RTP: A Transport Protocol for Real-Time Applications*, RFC 1889, January 1996.

[RFC2002] C. Perkins, *IP Mobility Support*, RFC 2002, October 1996.

[RFC2018] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, *TCP Selective Acknowledgement Options*, RFC 2018, October 1996.

[RFC2309] B. Braden, et al., *Recommendations on Queue Management and Congestion Avoidance in the Internet*, RFC 2309, April 1998.

[RFC2401] S. Kent, R. Atkinson, *Security Architecture for the Internet Protocol*, RFC 2401, November 1998.

[RFC2460] S. Deering, R. Hinden, *Internet Protocol, Version 6 (IPv6) Specification*, RFC 2460, December 1998.

[RFC2474] K. Nichols, S. Blake, F. Baker, D. Black, *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*, RFC 2474, December 1998.

[RFC2475] S. Blake, et al., *An Architecture for Differentiated Services*, RFC 2475, December 1998.

[RFC2481] K. K. Ramakrishnan, S. Floyd, *A Proposal to add Explicit Congestion Notification (ECN) to IP*, RFC 2481, January 1999.

[RFC2507] M. Degermark, B. Nordgren, S. Pink, *IP Header Compression*, RFC 2507, February 1999.

[RFC2581] M. Allman, V. Paxson, W. Stevens, *TCP Congestion Control*, RFC 2581, April 1999.

B.3 Software

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

[Lud99a] R. Ludwig, *Model of the TCP Sender Connection State in Equilibrium*, available at http://iceberg.cs.berkeley.edu, January 1999.

[Lud99c] R. Ludwig, TCP-Eifel, Patches for FreeBSD, available at http://iceberg.cs.berkeley.edu, October 1999.

[Pad98] Padmanabhan V., `tcpstats`, Appendix A of Ph. D. dissertation, University of California, Berkeley, September 1998.

[Xg] Xgraph, available at http://jean-luc.ncsa.uiuc.edu/Codes/xgraph/index.html.