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.

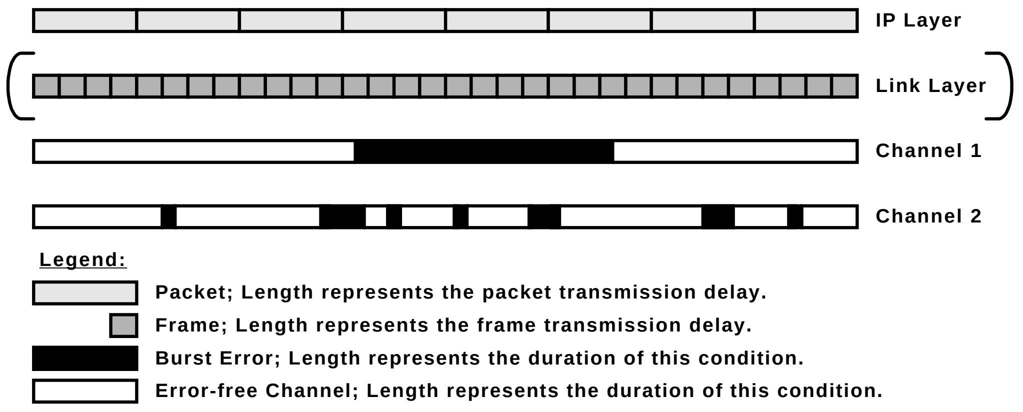
IP Layer
Link Layer
Channel 1
Channel 2
Legend:
Packet; Length represents the packet transmission delay.
Frame; Length represents the frame transmission delay.
Burst Error; Length represents the duration of this condition.
Error-free Channel; Length represents the duration of this condition.

**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.

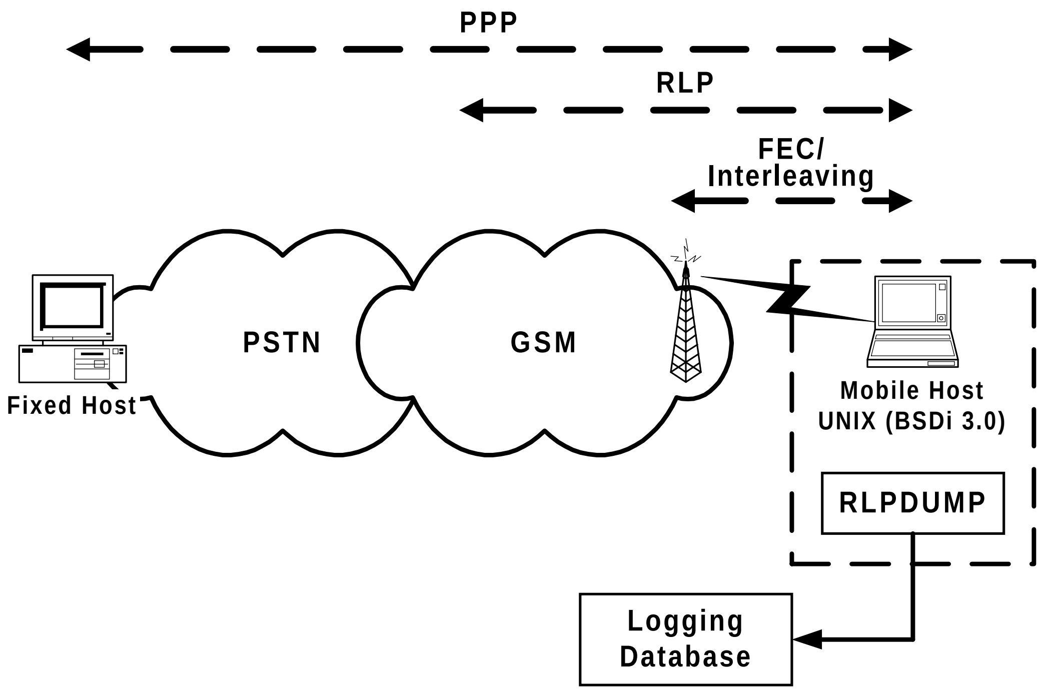

**Figure 3-2:** The measurement platform.

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.

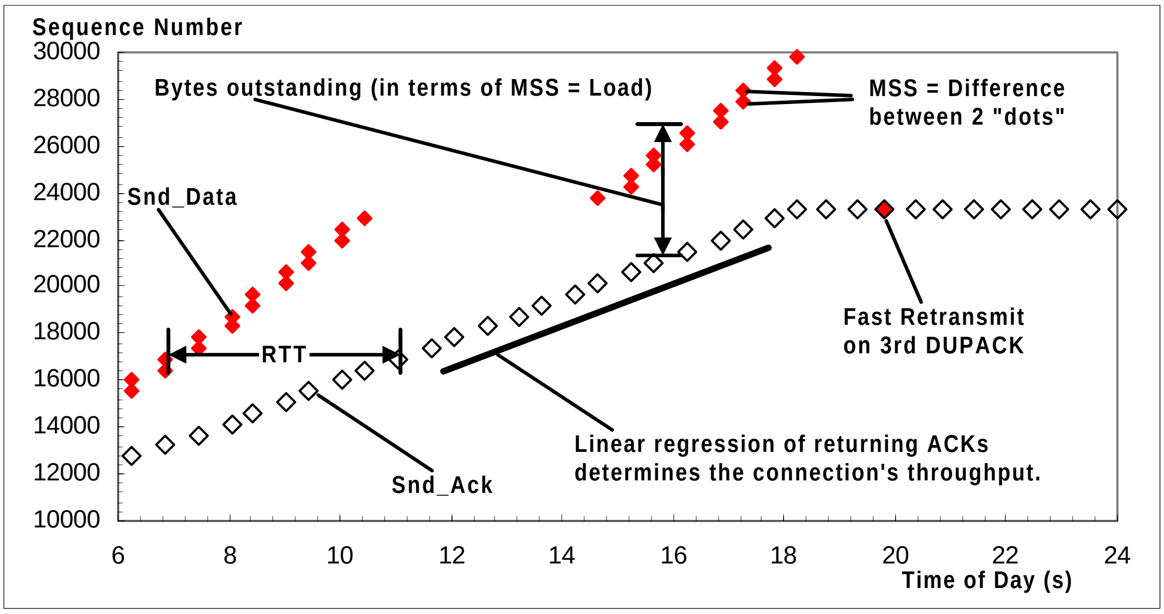

**Figure 3-3:** A TCP sender trace plot.

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.

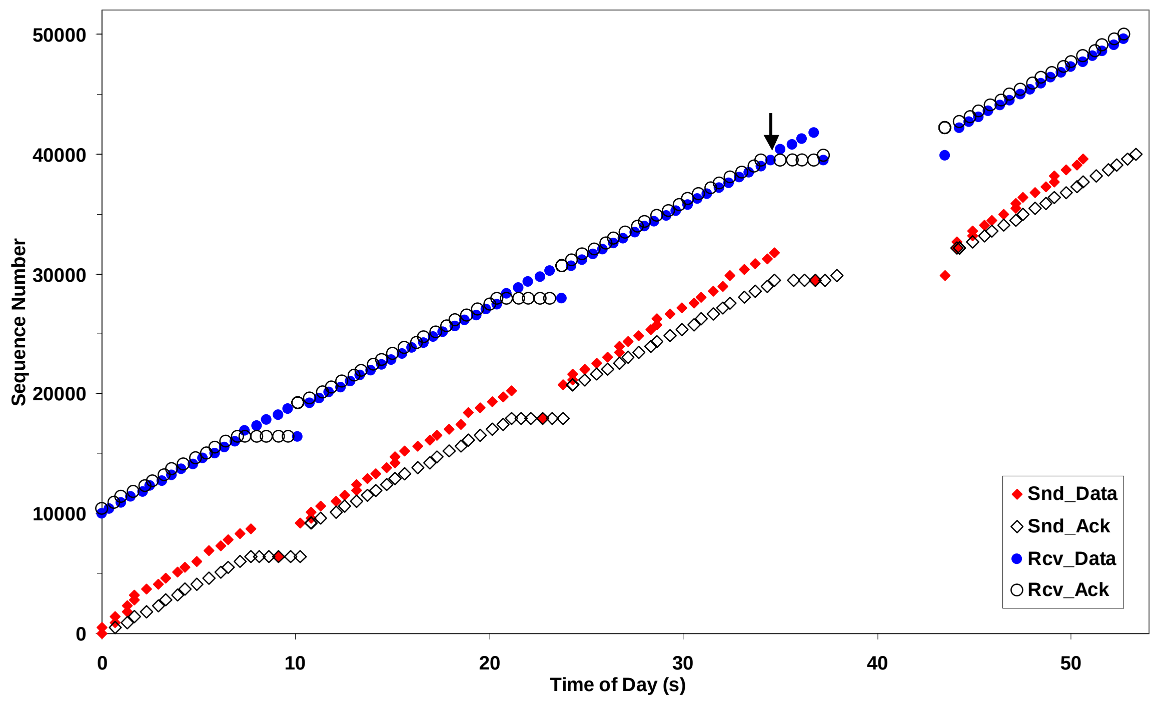

**Figure 3-4:** A TCP sender and receiver trace plot.

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

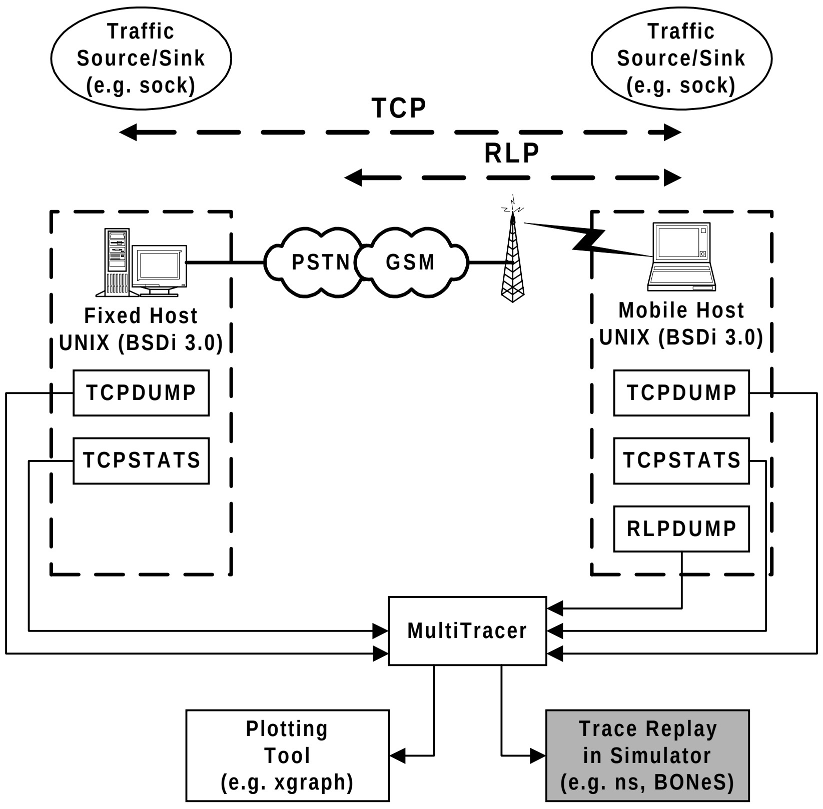
Traffic Source/Sink (e.g. sock)
Traffic Source/Sink (e.g. sock)
TCP
RLP
PSTN
GSM
Fixed Host UNIX (BSDi 3.0)
Mobile Host UNIX (BSDi 3.0)
TCPDUMP
TCPSTATS
TCPDUMP
TCPSTATS
RLPDUMP
MultiTracer
Plotting Tool (e.g. xgraph)
Trace Replay in Simulator (e.g. ns, BONeS)

**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.

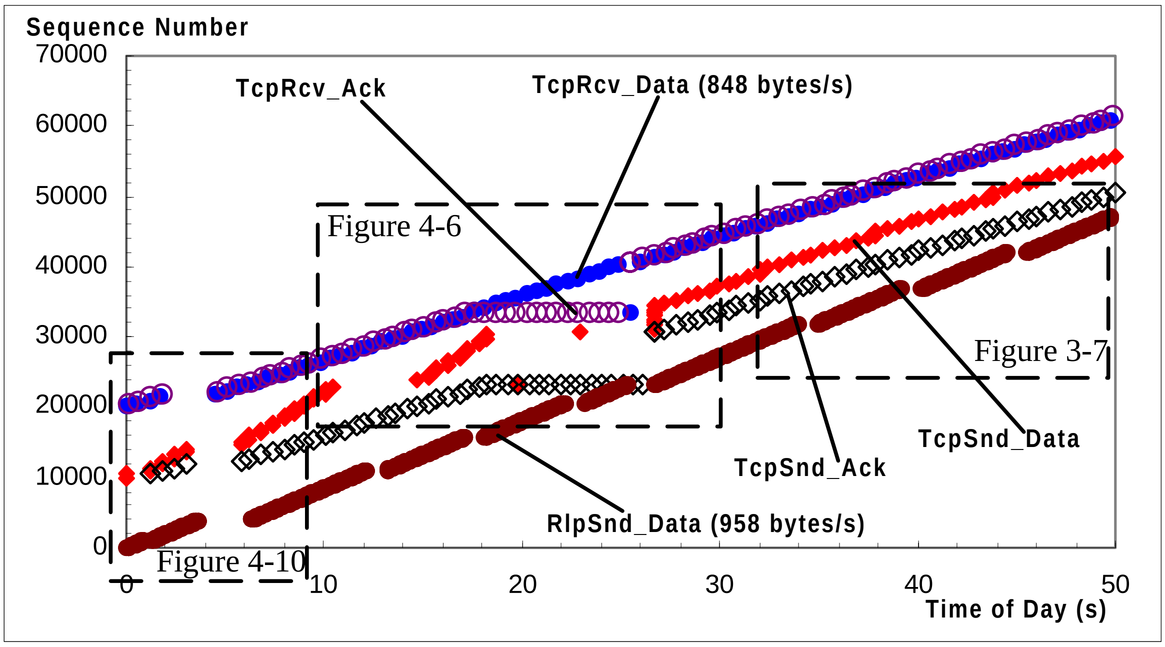

**Figure 3-6:** A typical multi-layer trace plot.

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-

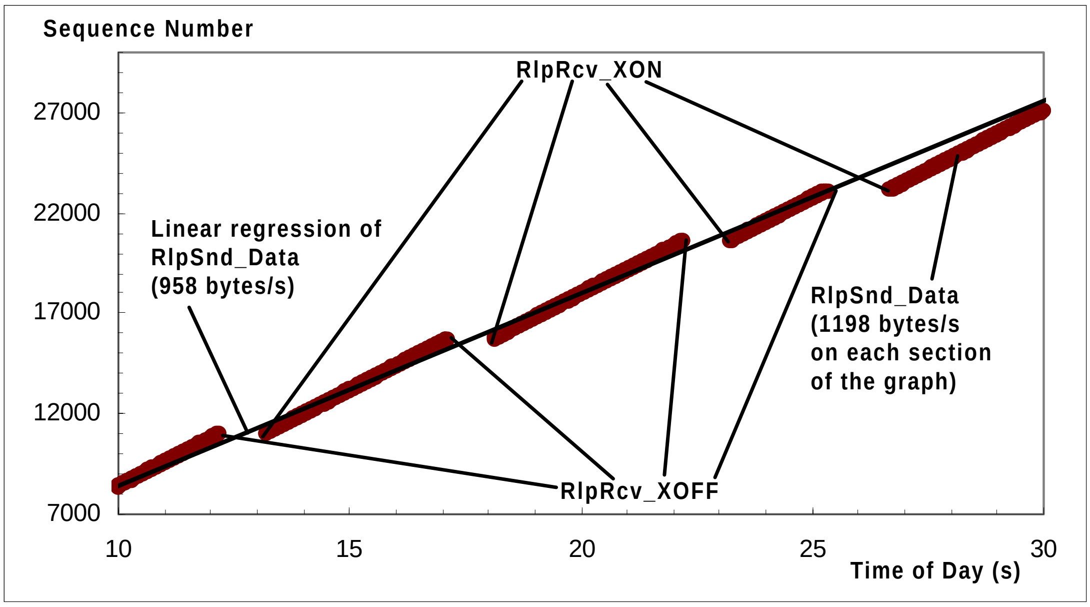

**Figure 3-7:** L2R flow control (zoom of Figure 3-6).

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.

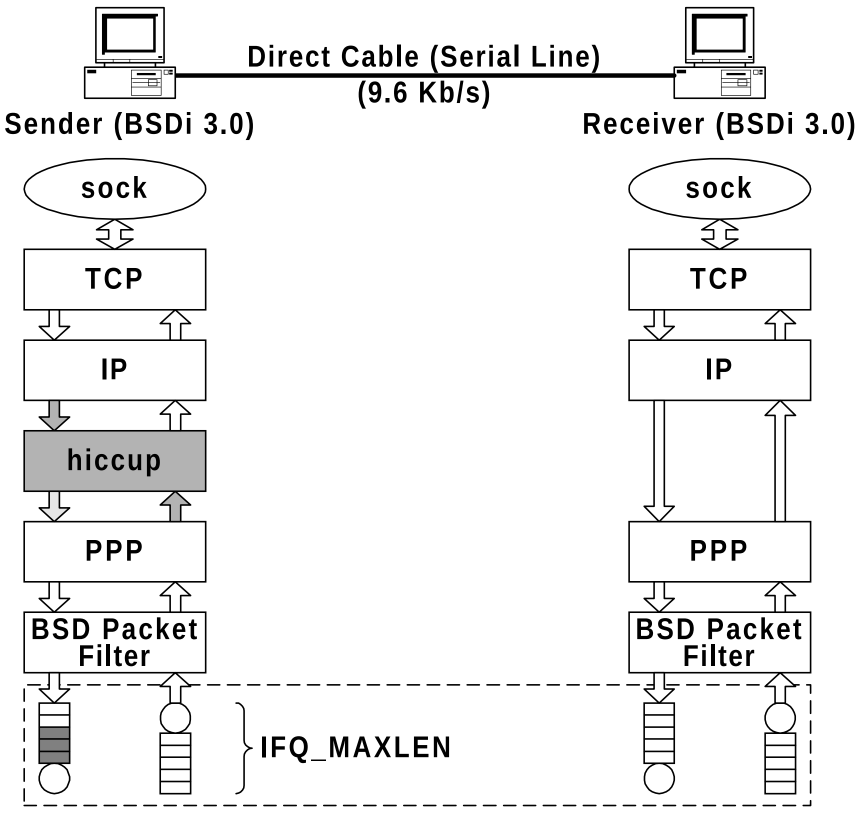

**Figure 3-8:** Measurement Setup.

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.

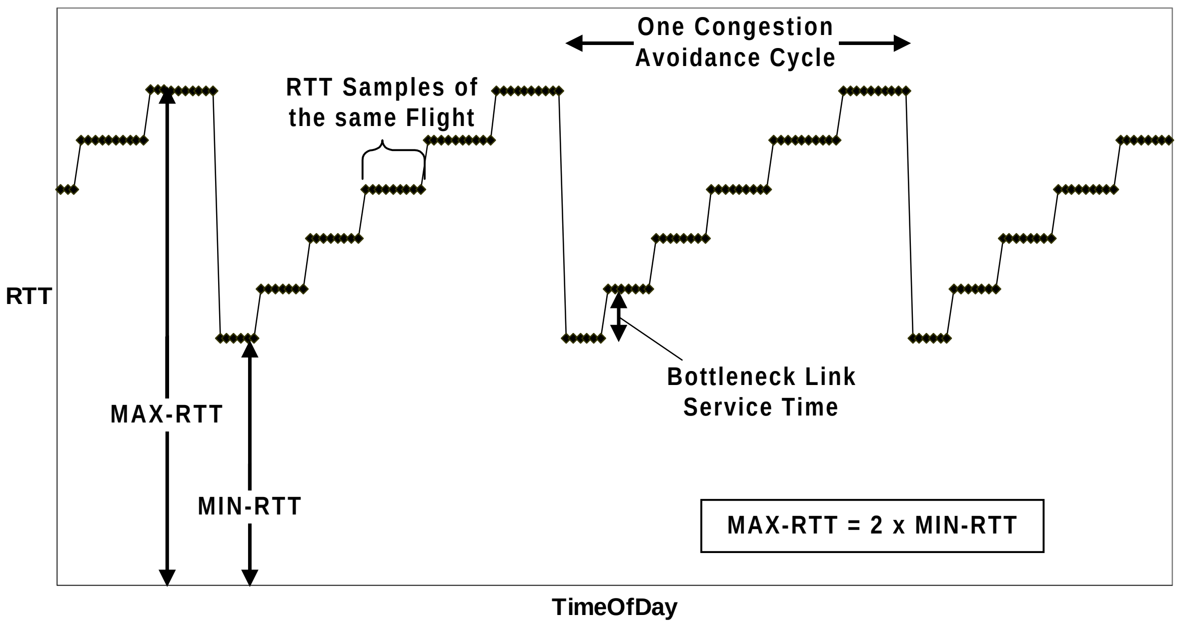

**Figure 3-9:** The RTT in steady state.

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).