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

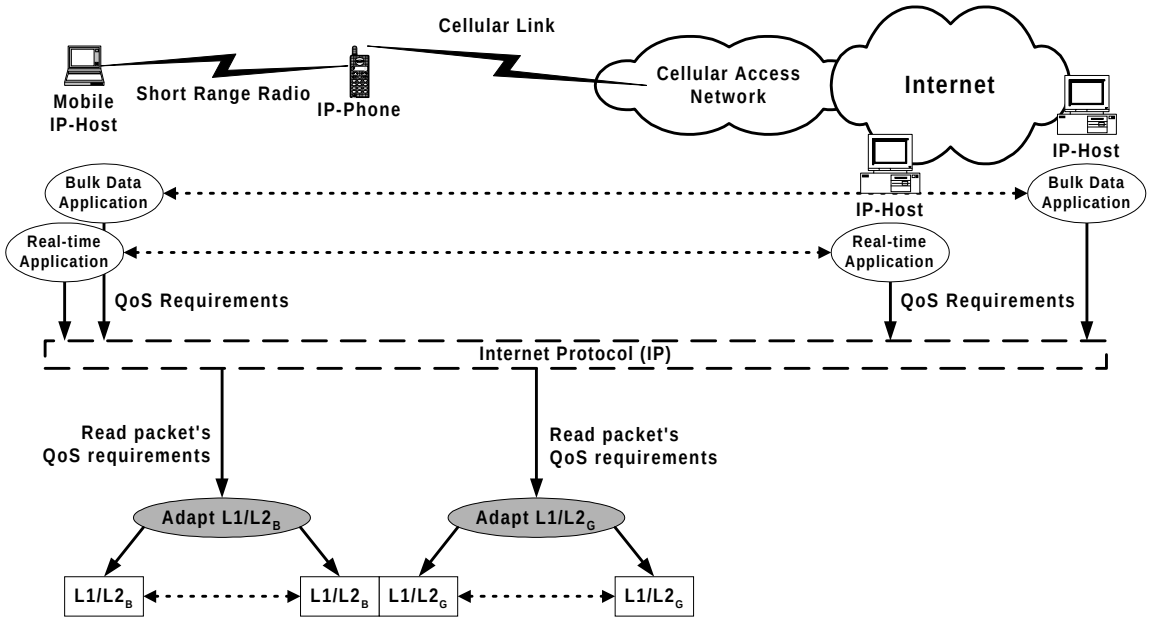

Figure 4-1: The concept of flow-adaptive wireless links.

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.

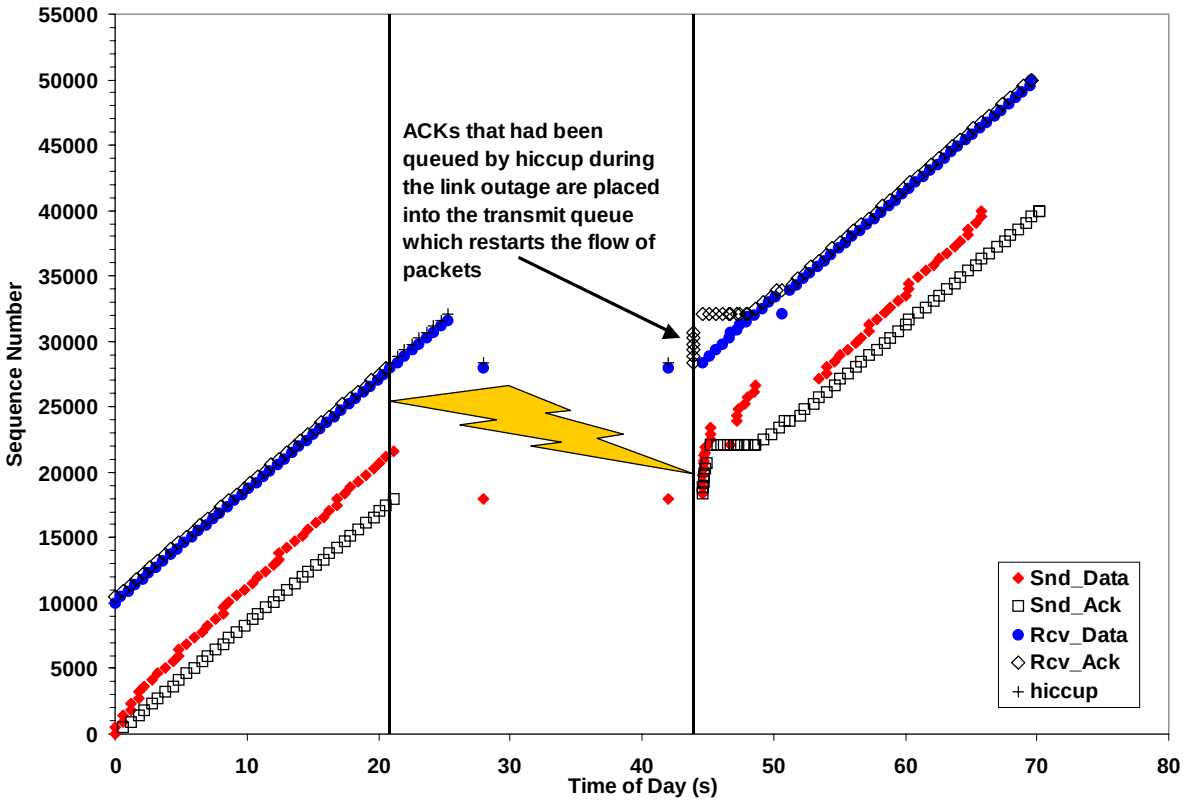

Figure 4-3: Link outage with high link layer error recovery persistency.

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.

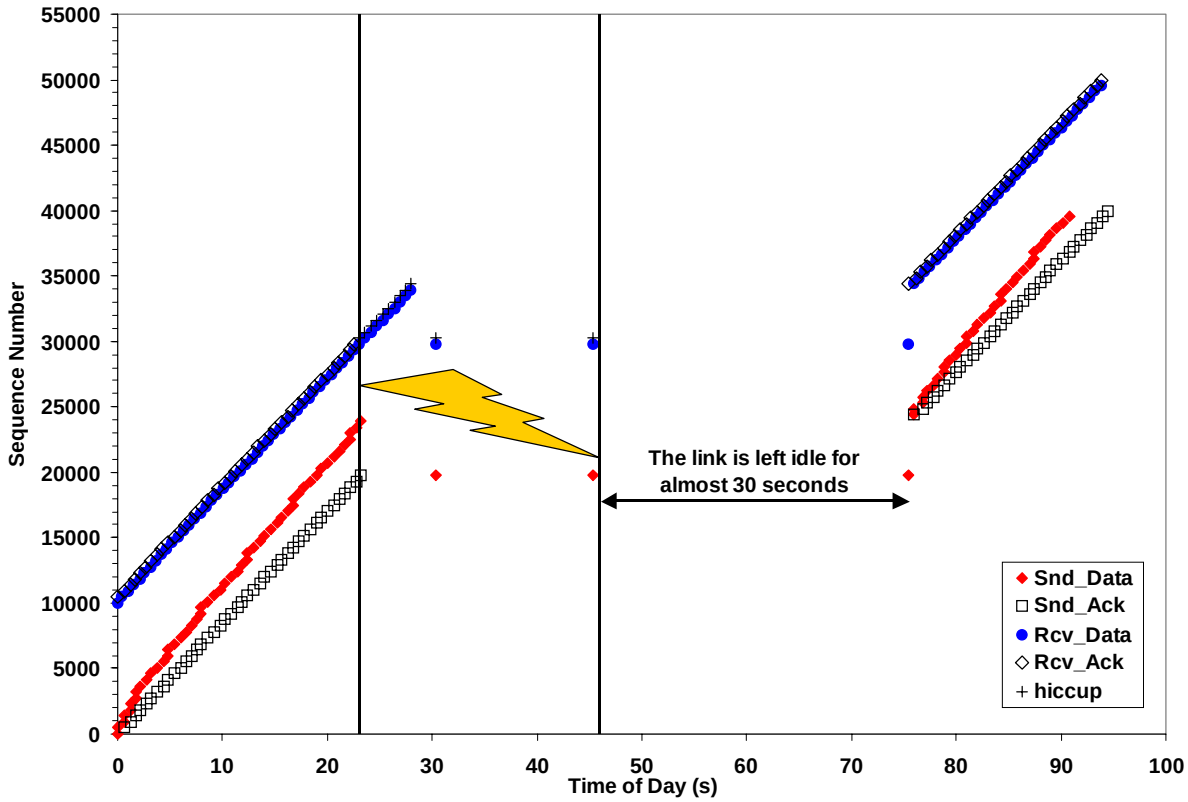

Figure 4-4: Link outage with low link layer error recovery persistency.

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

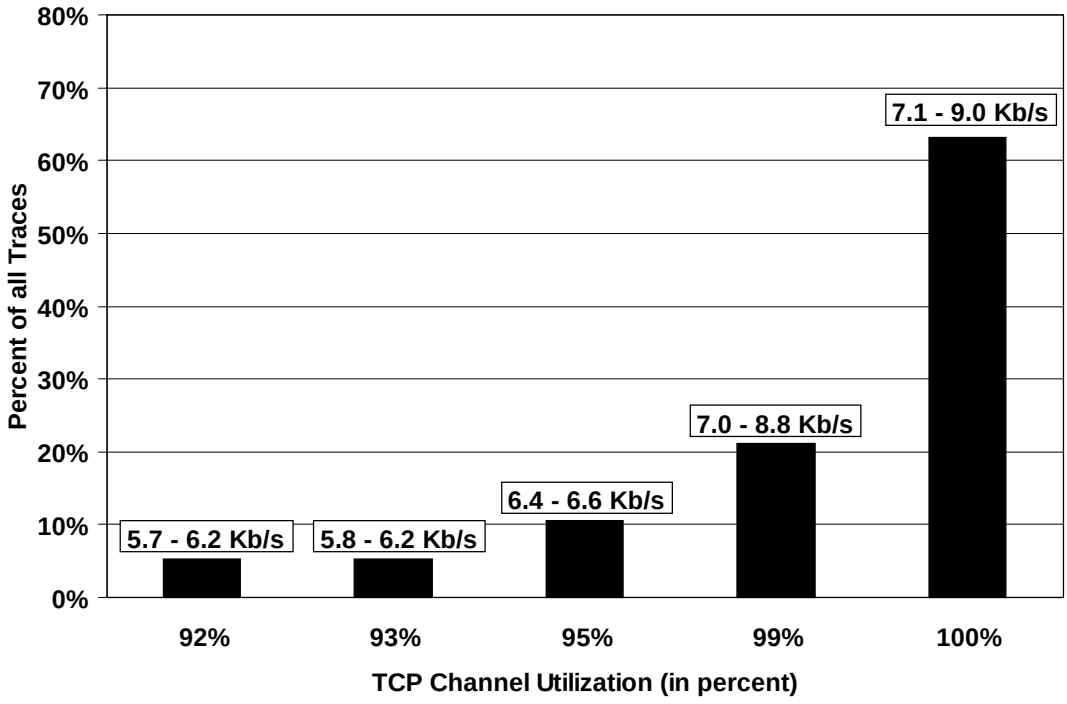

Figure 4-5: TCP channel utilization.

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.

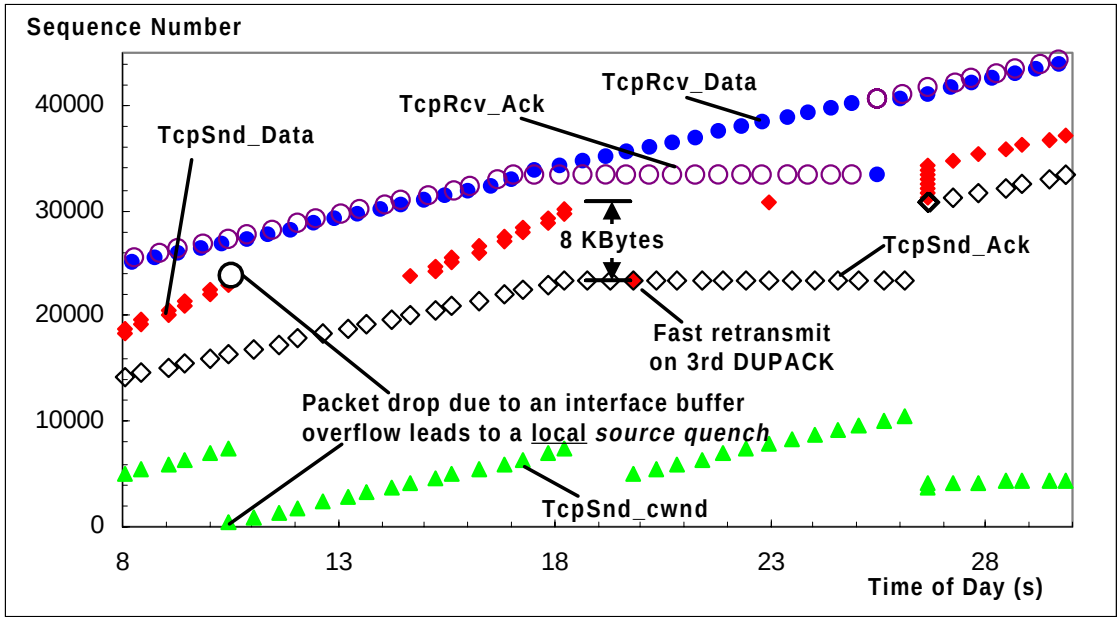

Figure 4-6: Local buffer overflow (zoom of Figure 3-6).

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

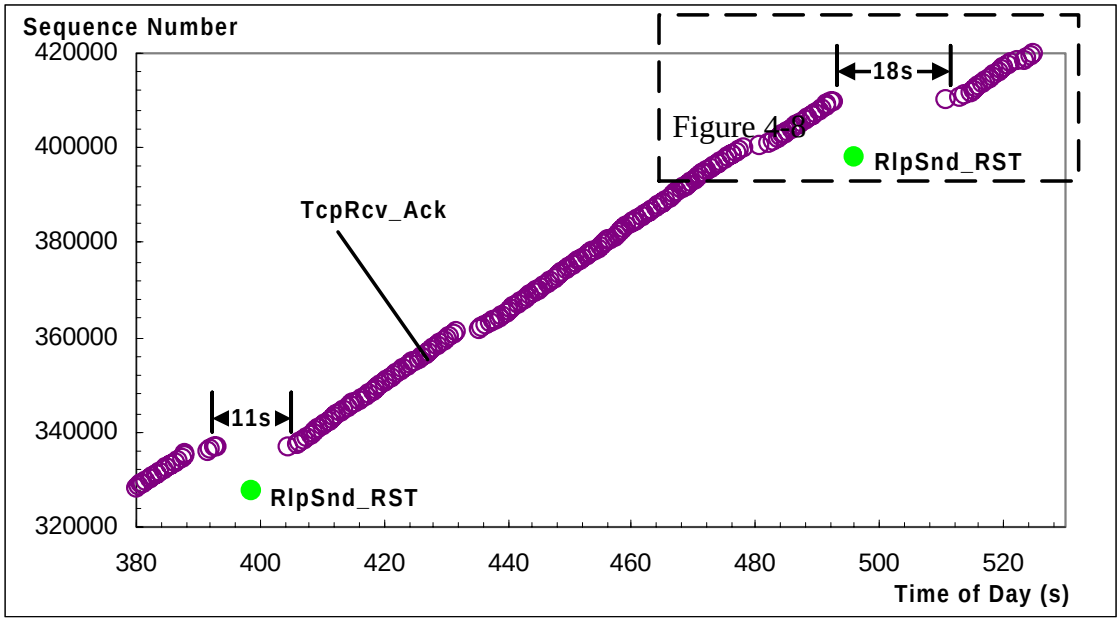

Figure 4-7: Header decompressor failures.

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

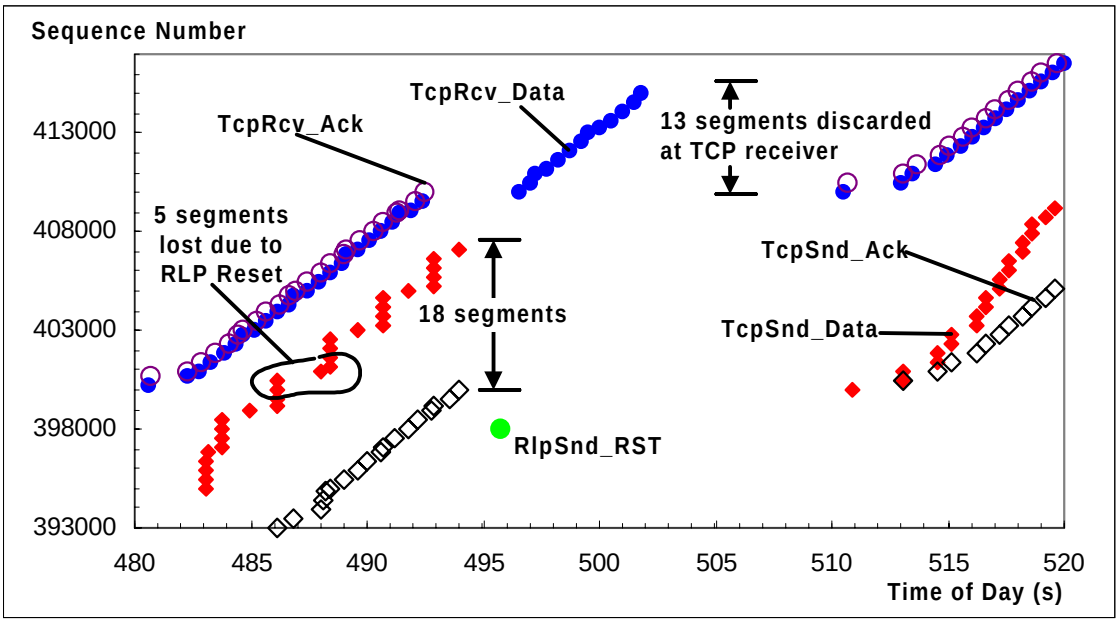

Figure 4-8: Zoom of Figure 4-7.

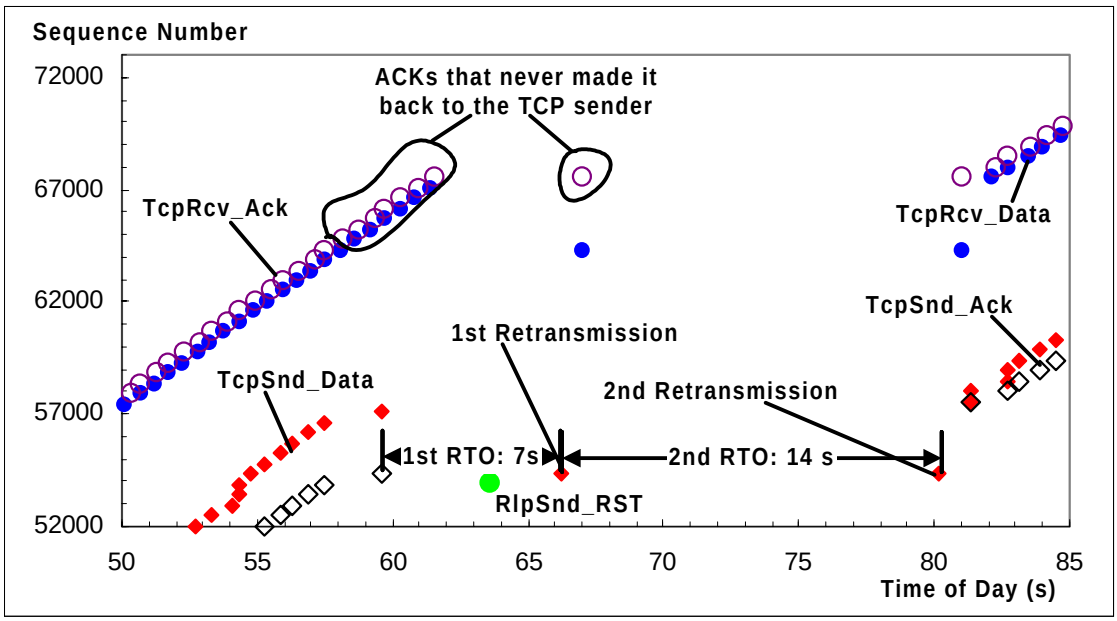

Figure 4-9: Exponential retransmission timer back-off.

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

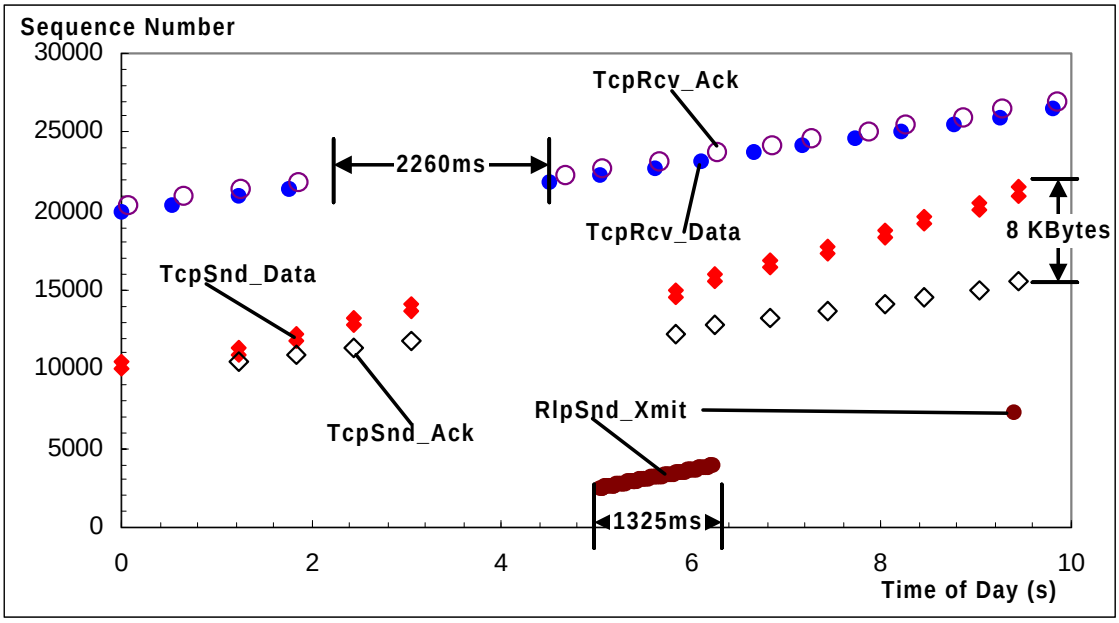

Figure 4-10: First 10 seconds of the trace in Figure 3-6.

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

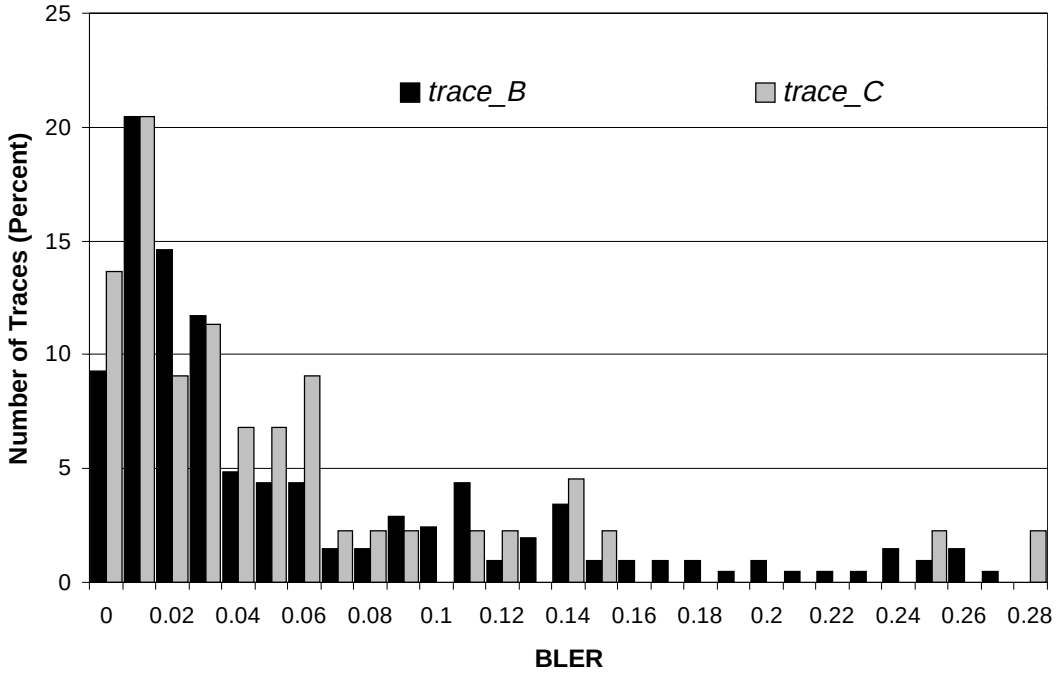

Figure 4-11: Measured BLERs.

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

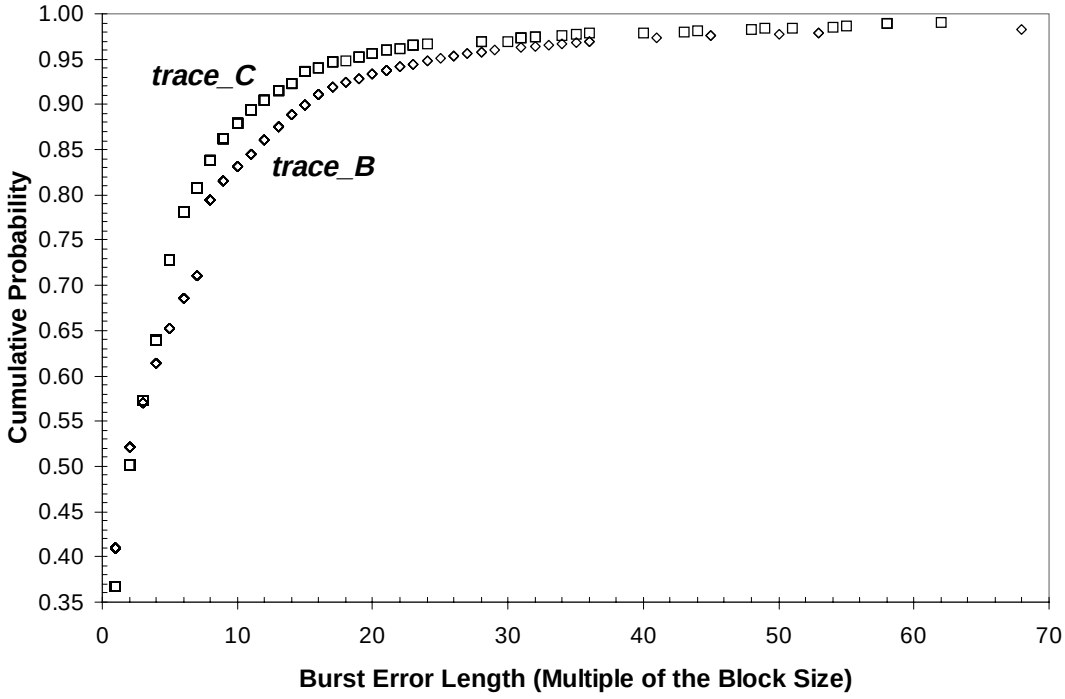

Figure 4-12: Burst error length distribution.

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

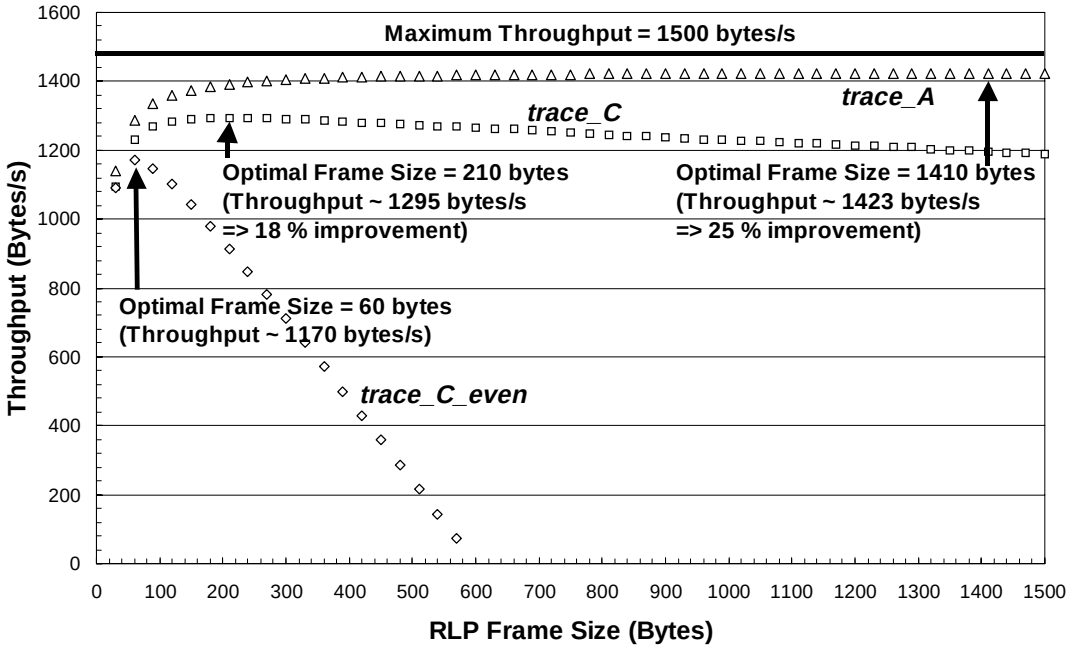

Figure 4-13: Throughput versus frame size.

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	1219	1196
	5.2% ↑	*2.4%* ↑	*4.9%* ↓
End-to-End with RLP (No TCP/IP header compression)	1094	1191	1255
	7.6% ↓	*3.8%* ↓	*7.4%* ↓
End-to-End with RLP (With TCP/IP header compression)	1239	1265	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.

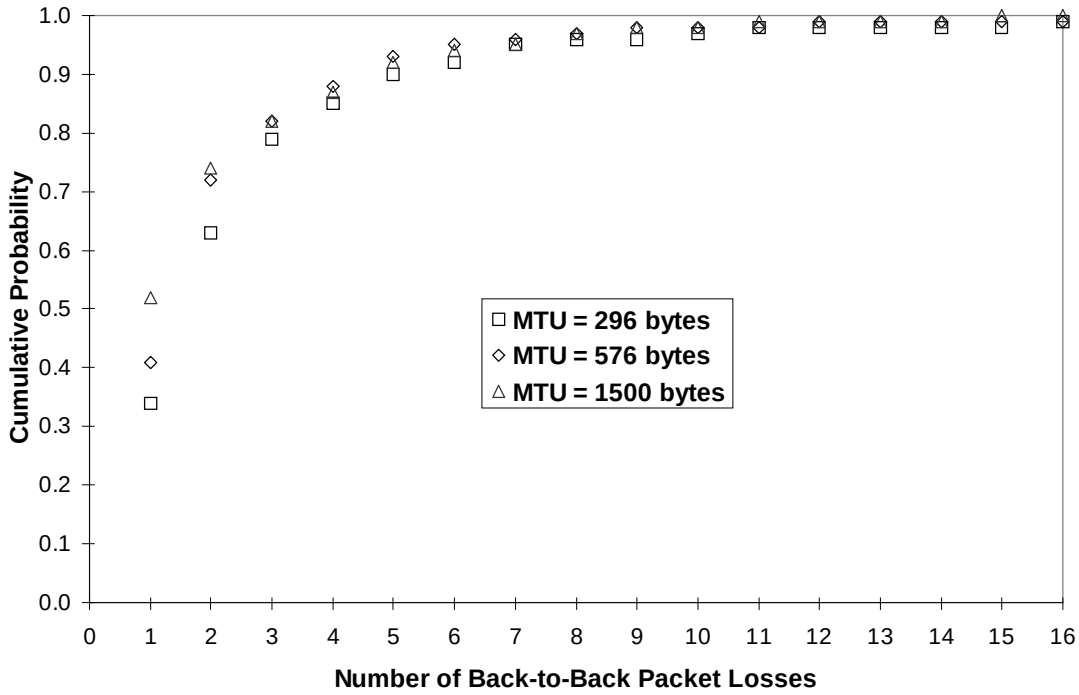

Figure 4-15: Distribution of back-to-back packet losses.

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

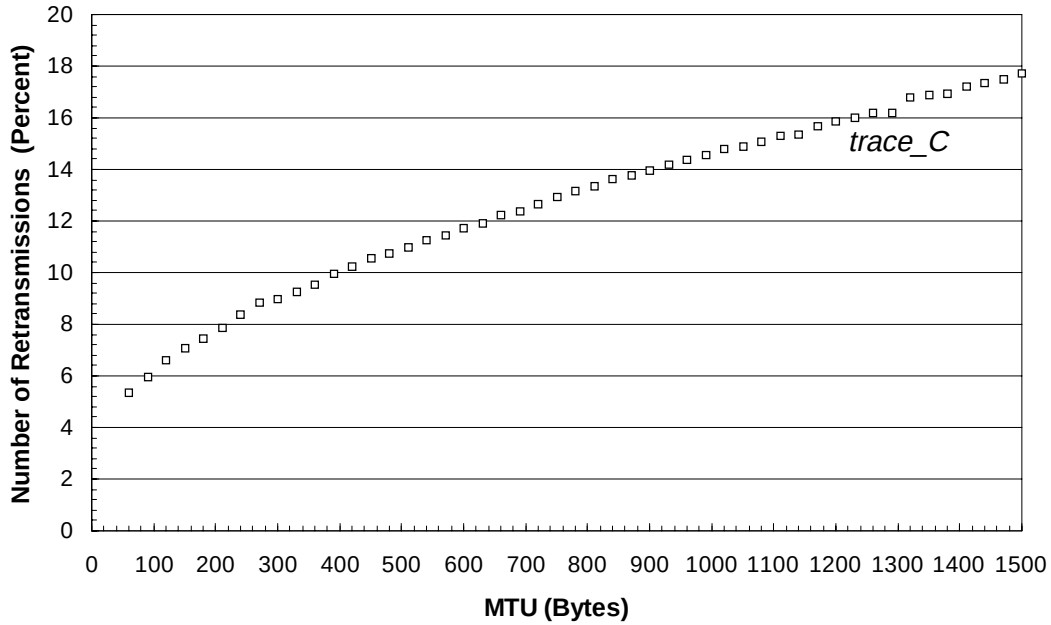

Figure 4-16: Number of end-to-end retransmissions.

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.