CHAPTER 2

# Background

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

## 2.1 Terminology

The *Internet* is an inter-connection of networks. A *network* connects *hosts* and networks are interconnected via *routers*. Communication in the Internet is based on the *Internet Protocol (IP)*, a network layer protocol defined by [RFC791], referred to as *IP-Version 4 (IPv4)*, and alternatively by [RFC2460], referred to as *IP-Version 6 (IPv6)*. A protocol data unit in IP is called a *packet*, consisting of an *IP header* followed by transport layer data[1]. The transport layer data may (in theory) be up to 64 KBytes large. The default size of the IP header is 20 bytes, and with option fields may be up to 60 bytes. The IP protocol is *connection-less* and as such does not guarantee *in-order delivery* of packets. That is, the sequence of packets as generated by the source does not need to be preserved when the packets are delivered to the destination. That responsibility is left to higher layer protocols such as the *Transmission Control Protocol (TCP)*. In particular, packets belonging to the same connection may take different routes to the destination and in practice sometimes do [Pax97d].

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 2.2 End-to-End Error Recovery with TCP

### 2.2.1 Basic Operation

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

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

2. Currently in experimental status.

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

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

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

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

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

Two different error recovery strategies have been specified for TCP: (1) timeout-based retransmission, and (2) DUPACK-based retransmission. In the latter case a retransmission - a so-called *fast retransmit* - is triggered when three[1] successive DUPACKs with the same ACK number have been received independent of the retransmission timer [Jac90a]. Section 2.2.2 provides a detailed description of TCP's retransmission timer. TCP's error recovery is fairly persistent. It retransmits a single segment twelve times which corresponds to roughly 9 minutes before the connection is aborted.

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

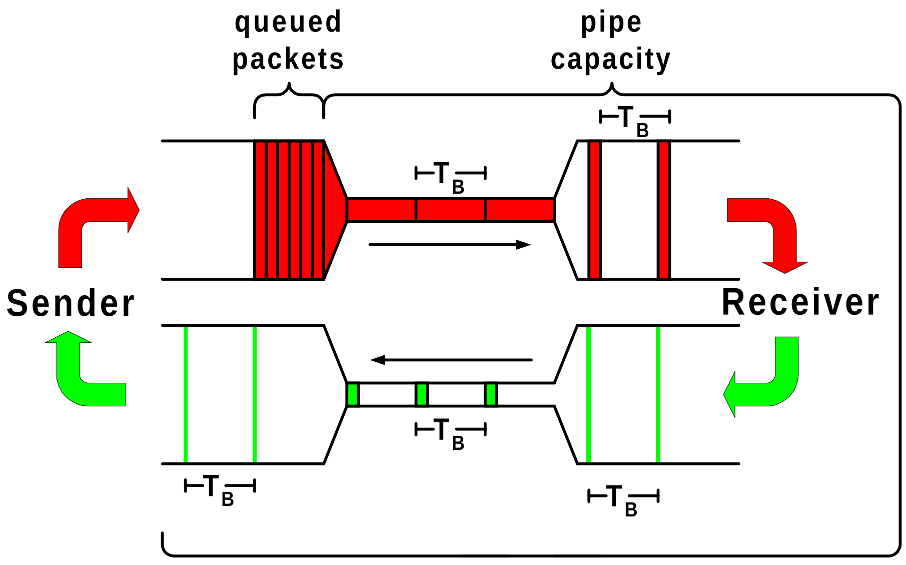

**Figure 2-1:** The "ACK Clock".

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

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

### 2.2.2 TCP-Lite's Retransmission Timer

While data is outstanding the TCP sender samples the path's RTT by timing the difference between sending a particular segment and receiving the corresponding ACK. Older TCP implementations only time one segment per RTT, whereas newer implementations use the timestamp option [RFC1323] to time every segment. Timing every segment allows much closer tracking of changes in the RTT. When using the timestamp option, the TCP sender writes the current value of a "timestamp clock" into a 12 bytes option field in the header of each outgoing segment. The receiver then echos those timestamps in the corresponding ACKs according to the rules defined in [RFC1323]. When receiving an ACK the TCP sender determines the RTT by calculating the difference between the current value of its "timestamp clock" and the timestamp echoed in the ACK. In the context of TCP, we speak of "the RTT" when referring to the RTT of the last segment for which the sender received the ACK, *independent* of whether the sender had timed that segment to derive the RTT.

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

The *retransmission timeout value (RTO)* is the time that elapses after a packet has been sent until the sender considers it lost and therefore retransmits it. This event is called a *timeout*. The RTO is a prediction of the upper limit of the RTT which - especially on an end-to-end path through the Internet - may vary considerably for various reasons. The time that remains until the timeout for a packet occurs is maintained by the *retransmission timer state (REXMT)*. Thus, the RTO is the REXMT's initial value. We use the term *retransmission timer* to refer to the combination of REXMT and RTO.

The retransmission timer is a key feature of a reliable transport protocol like TCP. It can greatly influence end-to-end performance. A too optimistic retransmission timer often expires prematurely. Such an event is called a *spurious timeout*. It causes unnecessary traffic, so-called *spurious retransmissions*, reducing a connection’s effective throughput. In TCP, timeouts also trigger congestion control (explained in Section 2.3.2), that may additionally reduce the end-to-end throughput. A retransmission timer that is too conservative may cause long idle times before the lost packet is retransmitted. This can also degrade performance. This is obvious for interactive flows. But it also affects bulk data transfers whenever the TCP sender has exhausted the window limiting the number of outstanding segments before the retransmission timer expires.

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

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

SRTT is the so-called *smoothed RTT estimator.* $SRTT_L$ is a low-pass filter that keeps a memory of a connection’s RTT history with a fixed weighing factor of 7/8. $DELTA_L$ is the difference between the latest $RTT_{Sample}$ and the current $SRTT_L$. RTTVAR is the so-called *smoothed RTT deviation estimator.* Through RTTVAR, the RTO accounts for variations in RTT. $RTTVAR_L$ is a low-pass filter that keeps a memory of a connection’s RTT deviation history with a fixed weighing factor of 3/4. We refer to the constants 1/4 and 1/8 as the *estimator gains* and to the constant 4 as the *variation weight*. Little motivation other than implementation efficiency is provided in [JK92] for this particular set of constants.

REXMT and RTO are maintained in multiples of *ticks*, i.e., some fraction of a second that is operating system dependent. This is also referred to as the *timer granularity.* $REXMT_L$ is based on a so-called *heartbeat timer* provided by the BSD operating system implementing a timer granularity of 500 ms. The heartbeat timer expires *every* 500 ms, triggering a specific interrupt routine that updates the $REXMT_L$ (decrements it by one tick) of each active TCP connection. This is done independent of whether one of those $REXMT_L$ would actually go to zero or not. If

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

We call the time that has elapsed since a segment was sent the *age of a segment*. Likewise we refer to the *oldest outstanding segment* as that segment in the TCP sender's send socket buffer with the highest age. That segment also carries the lowest sequence number of all outstanding segments. It is the segment that gets retransmitted when REXMT expires. TCP-Lite maintains a single REXMT per TCP connection. The following equation defines $REXMT_L$. When a segment is sent and $REXMT_L$ is not active, it is started (initialized with $RTO_L$). When an ACK arrives that acknowledges the oldest outstanding segment and more segments are still outstanding, $REXMT_L$ is re-initialized with $RTO_L$.

$$REXMT_L = RTO_L$$

We briefly summarize related work concerning the Lite-Xmit-Timer. *Karn's algorithm* [KP87] must be implemented in TCP [RFC1122]. It prevents a clamped retransmission timer by ignoring the $RTT_{Sample}$ derived from a retransmitted segment *and* doubling the RTO (*exponential timer backoff*) up to a maximum of two times MSL, i.e., 240 seconds, each time REXMT expires for the same segment. This makes it possible to eventually collect a valid $RTT_{Sample}$ again. Otherwise the sender might get stuck retransmitting the oldest outstanding segment while REXMT is clamped at too low a value. The authors of [BP95b] remove an inaccuracy in the implementation of $RTO_L$ that made it more conservative then intended in its definition. This has been updated accordingly in later TCP implementations (e.g., in the FreeBSD operating system). Through trace-driven simulation, the Lite-Xmit-Timer and some of its variations are evaluated in [AP99] against a large set of real measurements. The authors conclude that the RTO minimum (*2 x ticks*, i.e., 1 second) dominates the performance of the Lite-Xmit-Timer and that its performance can be further increased when a timer granularity of 100 ms or less is implemented. However, the study also concludes that the estimator gains and the RTT sampling rate have little influence on the Lite-Xmit-Timer's performance. We disagree with the latter conclusion and show in Section 5.3.2 that in fact the opposite is the case.

### 2.2.3 TCP/IP Header Compression

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

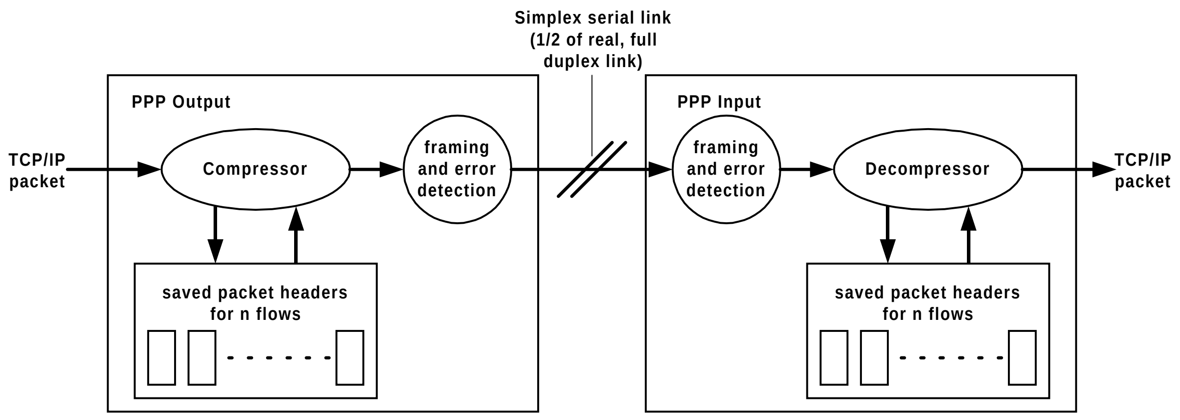
Simplex serial link
(1/2 of real, full
duplex link)
PPP Output
PPP Input
TCP/IP packet
Compressor
framing and error detection
framing and error detection
Decompressor
TCP/IP packet
saved packet headers for n flows
saved packet headers for n flows

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

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

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

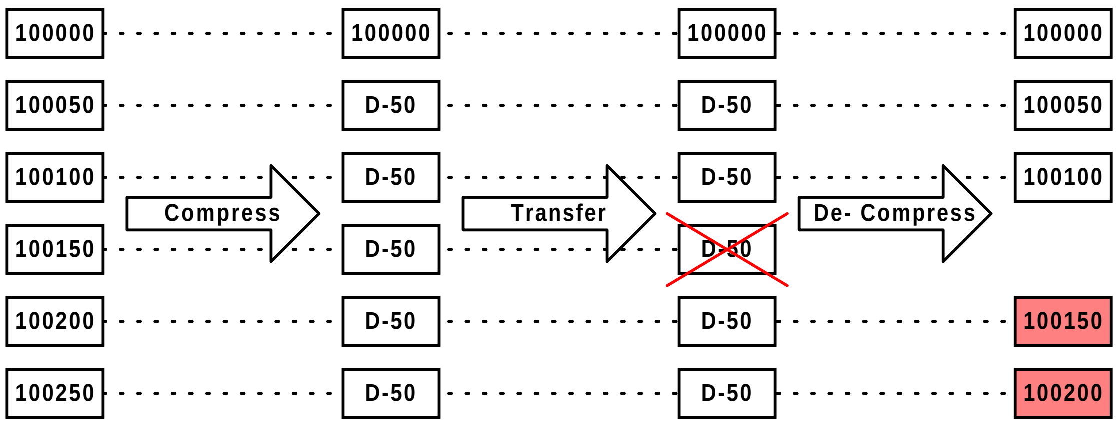

**Figure 2-3:** Compressor and decompressor losing synchronization.

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

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

## 2.3 End-to-End Congestion Control in the Internet

### 2.3.1 Objectives and Principles

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

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

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

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

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

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

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

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

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

### 2.3.2 Congestion Control in TCP

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

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

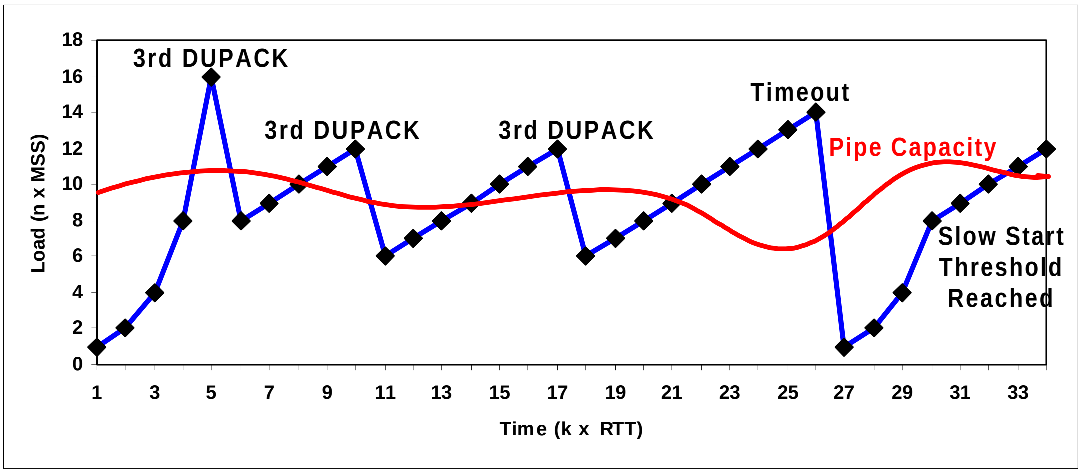

**Figure 2-4:** A sample evolution of the congestion window.

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

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

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

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

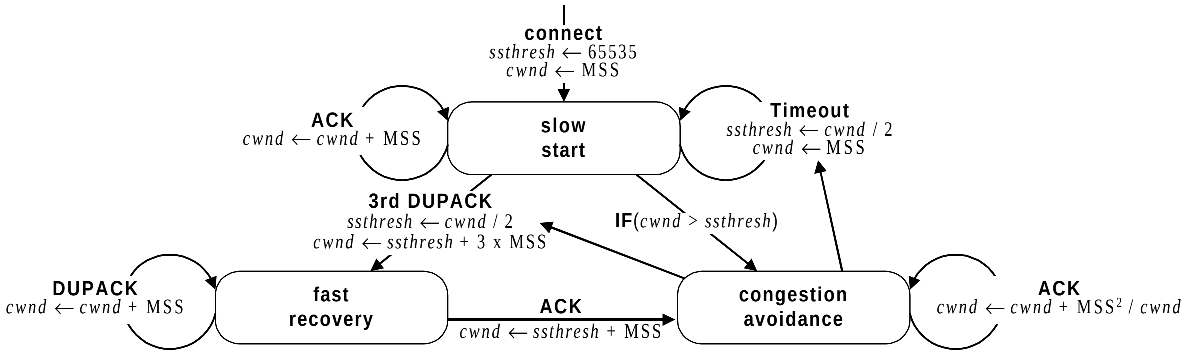

**Figure 2-5:** Congestion control in TCP.

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

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

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

### 2.3.3 New Developments

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

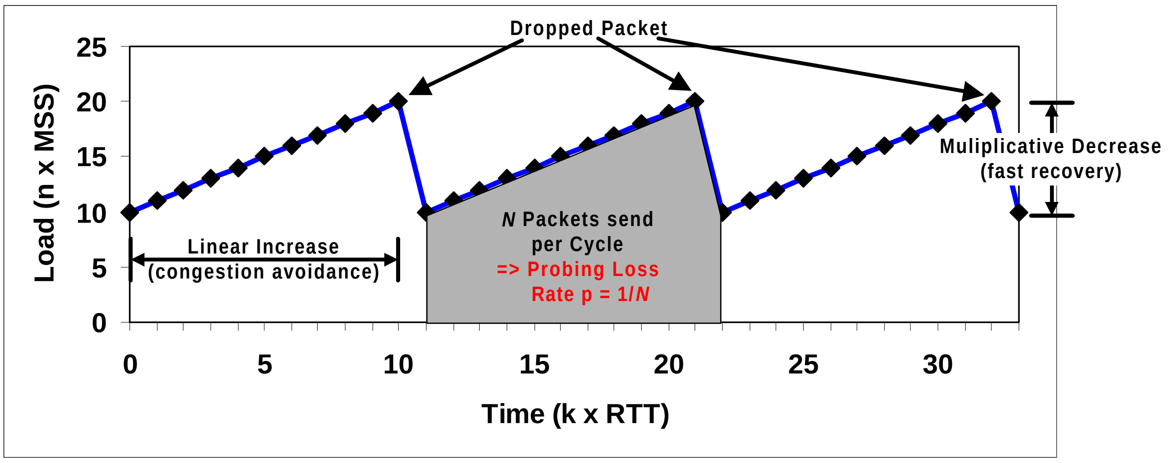

**Figure 2-6:** Deriving the "TCP-friendly" congestion control scheme.

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

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

## 2.4 Link Layer Error Control in Wireless Networks

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

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

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

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

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

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

### 2.4.1 Circuit-Switched Data Transmission in GSM

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

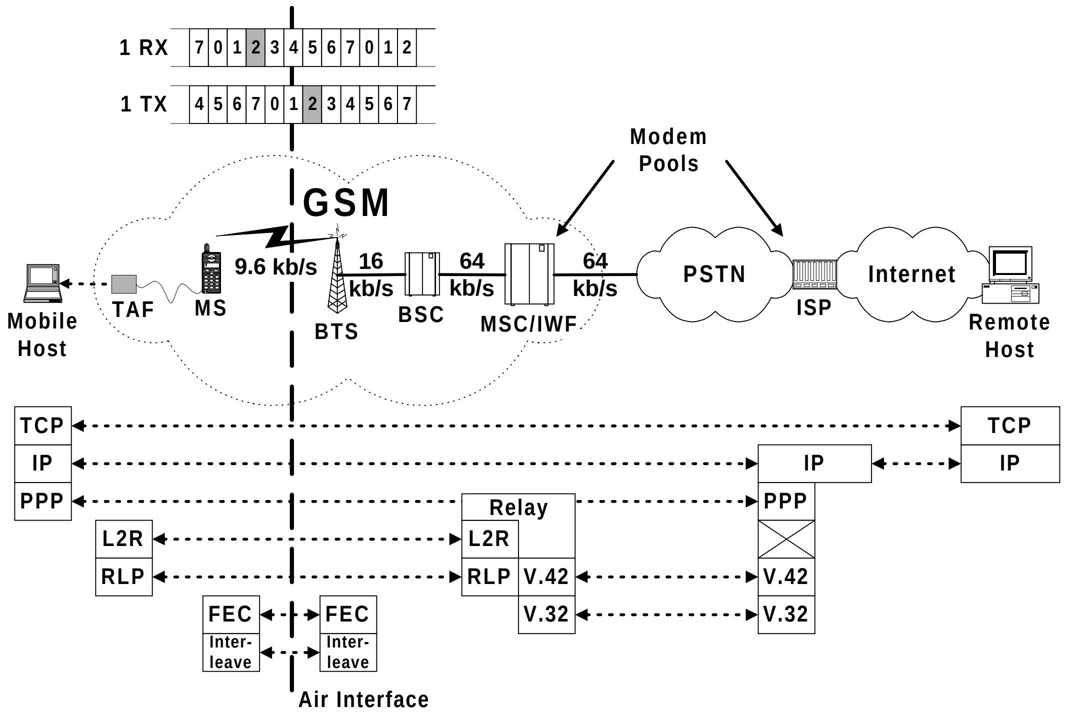

**Figure 2-7:** TCP/IP over GSM Circuit-Switched Data (CSD).

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

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

### 2.4.2 Handover Control

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

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

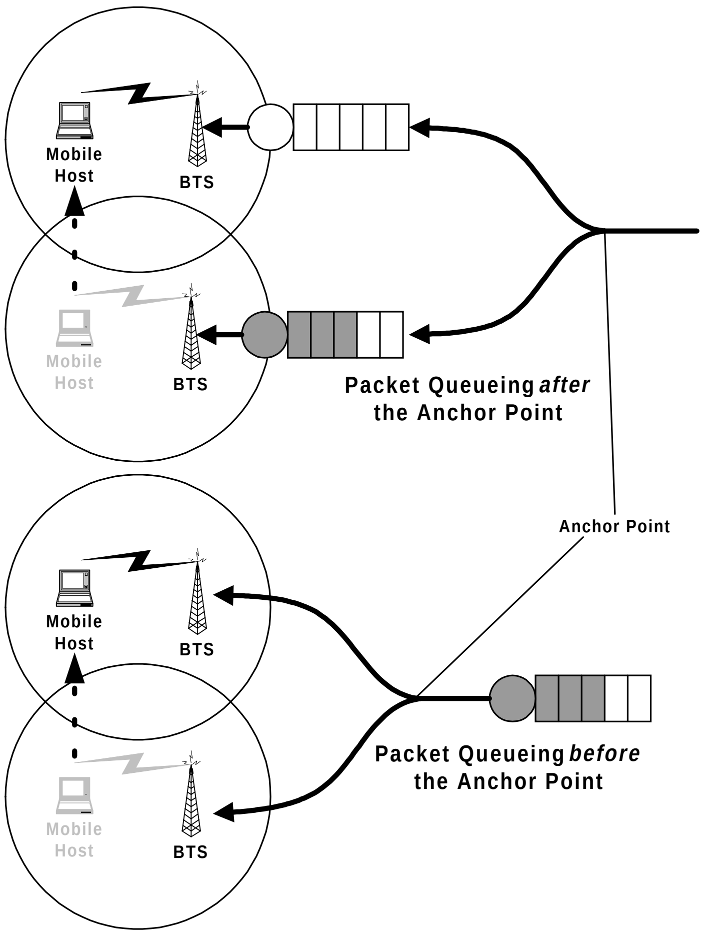
Mobile Host
BTS
Mobile Host
BTS
Packet Queueing *after* the Anchor Point
Anchor Point
Mobile Host
BTS
Mobile Host
BTS
Packet Queueing *before* the Anchor Point

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

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

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

### 2.4.3 Link Layer Error Recovery

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

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

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

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

### 2.4.4 Forward Error Correction and Interleaving

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

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

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

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

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

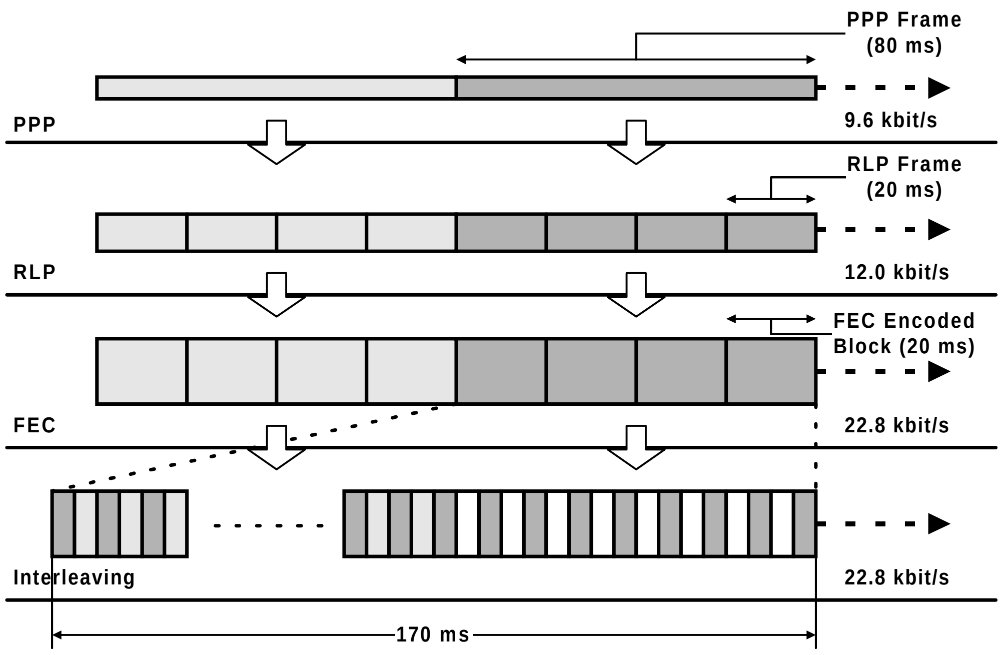

**Figure 2-9:** Error control in GSM Circuit-Switched Data.

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

## 2.5 The Problem: Inefficient Cross-Layer Interactions

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

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

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

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

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

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

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

## 2.5.1 Underestimation of the Available Bandwidth

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

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

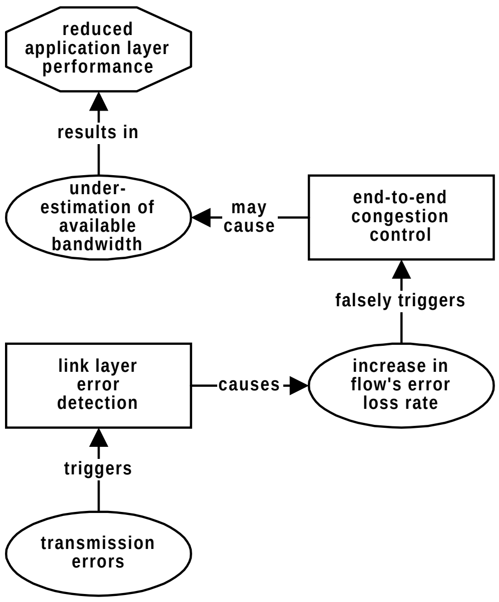
reduced application layer performance
results in
under-estimation of available bandwidth
may cause
end-to-end congestion control
falsely triggers
link layer error detection
causes
increase in flow's error loss rate
triggers
transmission errors

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

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

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

## 2.5.2 Inefficiency of End-to-End Error Control

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

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

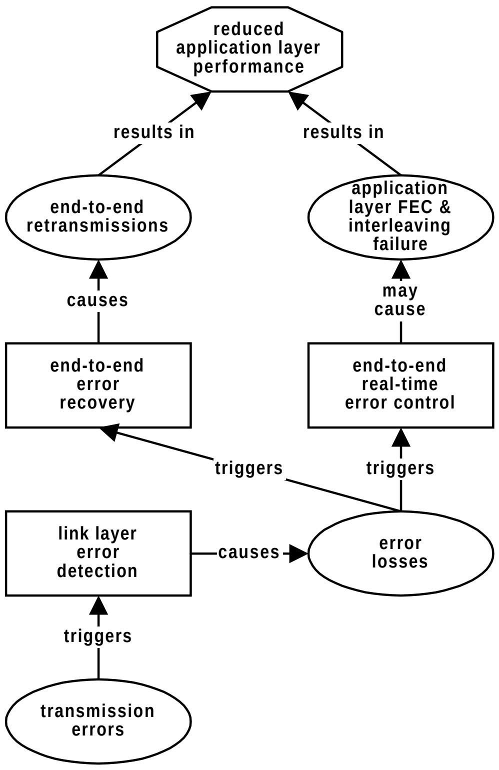
reduced application layer performance
results in
results in
end-to-end retransmissions
application layer FEC & interleaving failure
causes
may cause
end-to-end error recovery
end-to-end real-time error control
triggers
triggers
link layer error detection
causes
error losses
triggers
transmission errors

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

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

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

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

### 2.5.3 Overly Strong Link Layer Error Control

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

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

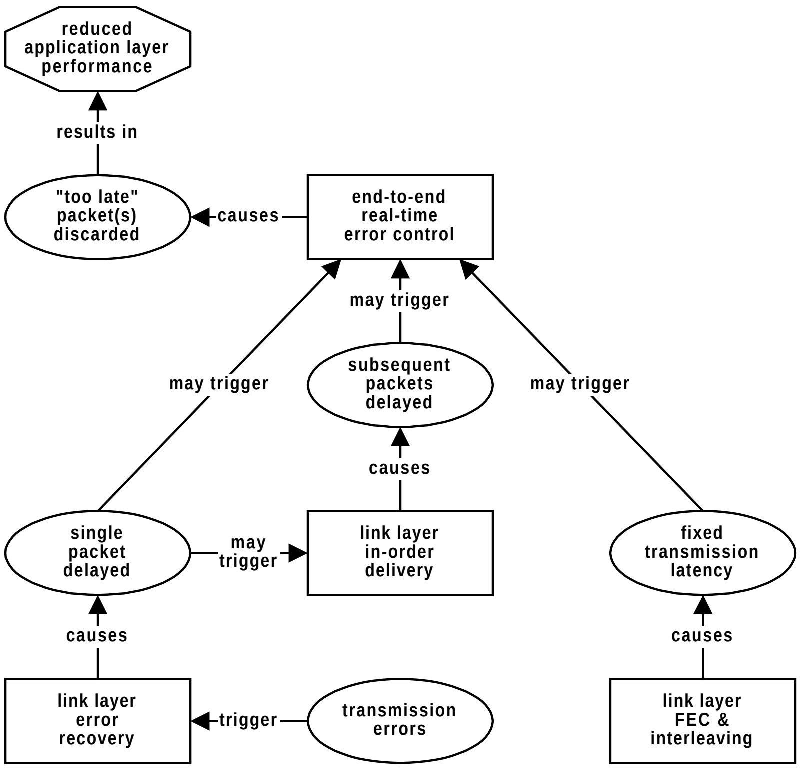
reduced application layer performance
results in
"too late" packet(s) discarded
causes
end-to-end real-time error control
may trigger
subsequent packets delayed
may trigger
may trigger
causes
single packet delayed
may trigger
link layer in-order delivery
fixed transmission latency
causes
causes
link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

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

### 2.5.4 Competing Error Recovery

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

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

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

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

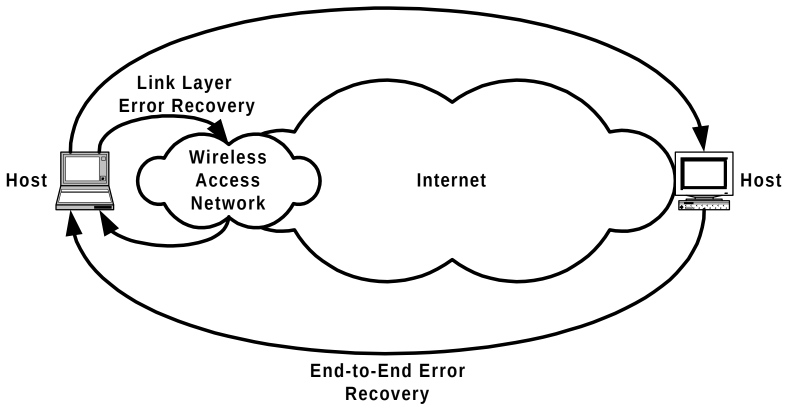

**Figure 2-13:** Uncoordinated control loops reacting to the same signal.

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

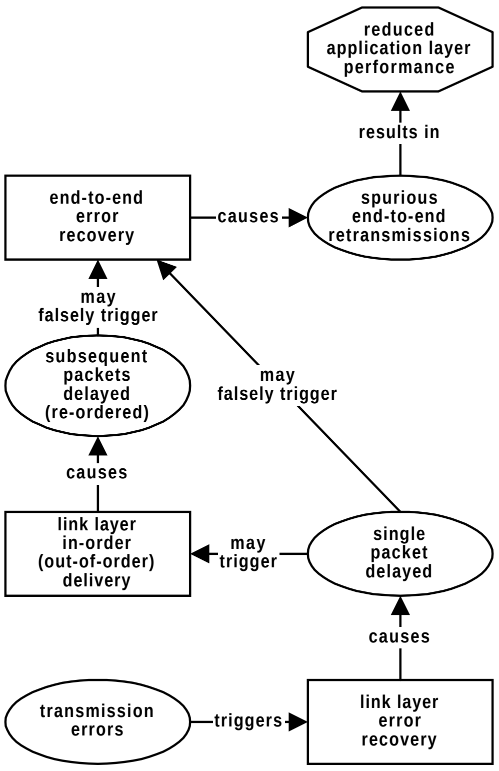
reduced application layer performance
results in
end-to-end error recovery
causes
spurious end-to-end retransmissions
may falsely trigger
subsequent packets delayed (re-ordered)
may falsely trigger
causes
link layer in-order (out-of-order) delivery
may trigger
single packet delayed
causes
transmission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

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

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

A *packet re-ordering* event occurs when a packet arrives at the receiving network end-point after one or more packet(s), that had left the sending end-point later, have already arrived. We call the number of packets that had already arrived out-of-order, the *re-ordering length*. For example, if packets 1 - 10 are sent but packet 1 arrives last, then the re-ordering length is 9.

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

## 2.5.5 Failure of Link Layer Differential Encodings

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

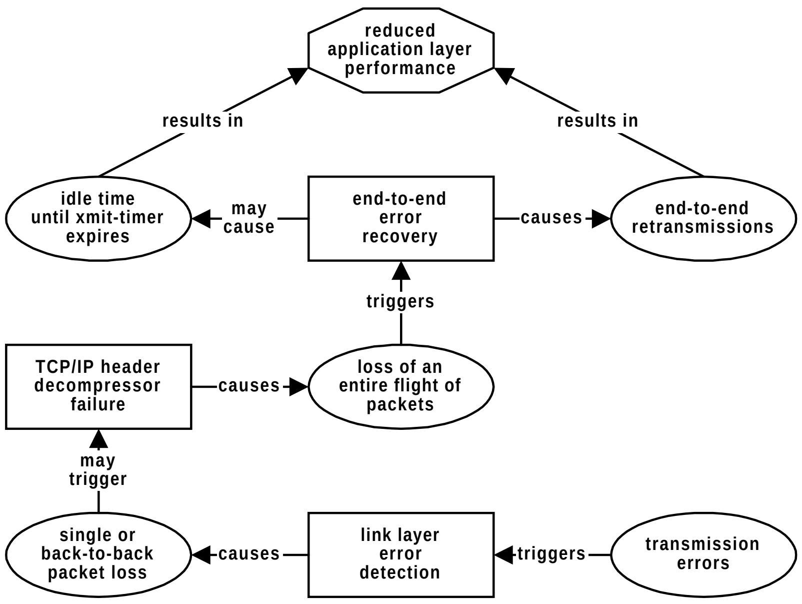
reduced application layer performance
results in
results in
idle time until xmit-timer expires
may cause
end-to-end error recovery
causes
end-to-end retransmissions
triggers
TCP/IP header decompressor failure
causes
loss of an entire flight of packets
may trigger
single or back-to-back packet loss
causes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

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

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

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

## 2.6 Related Work

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

### 2.6.1 Classification of Existing Approaches

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

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

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

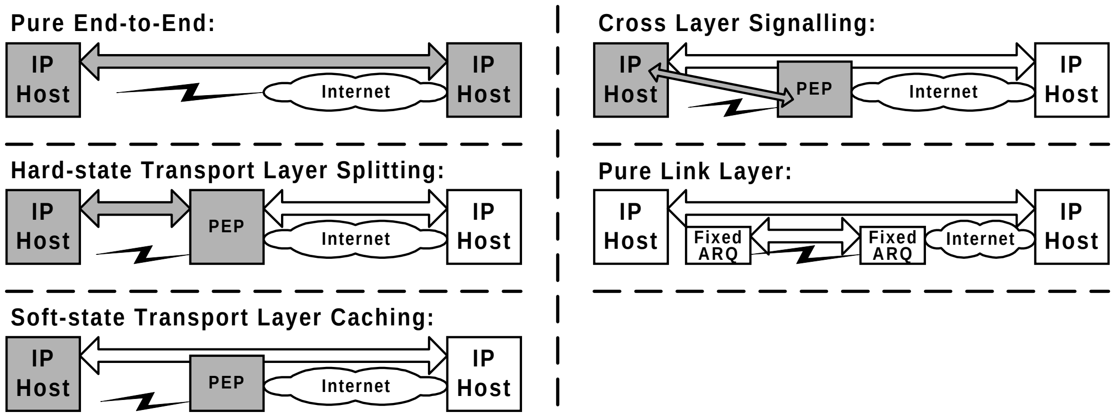

**Figure 2-16:** Existing approaches to solve "TCP over Wireless".

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

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

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

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

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

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

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

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

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

## 2.6.2 Evaluation

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

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

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

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

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

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

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

**Tabelle 2-17:** Evaluation of related work.

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

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

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

## 2.7 Brief Motivation and Outline of our Approach

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

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

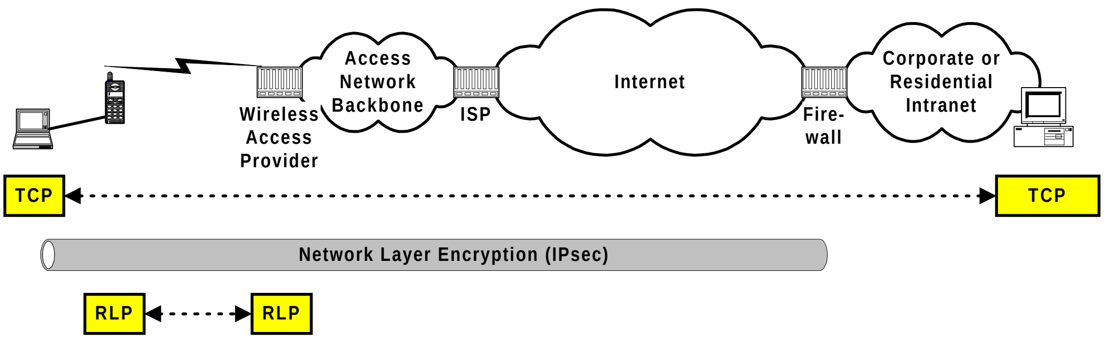

**Figure 2-18:** Getting secure access to an Intranet with an IPsec tunnel.

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

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

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

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

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