# CHAPTER 9

# TCP Over Wireless


**January 19, 2010**

## 9.1 Introduction

TCP (Transmission Control Protocol) is a transport layer protocol used quite commonly in the internet. Since a significant fraction of the traffic in the internet is carried using TCP, it is important to consider performance of TCP in the context of wireless networks. TCP is designed to perform reliable in-order delivery, while using IP to route packets. As such, TCP should work correctly on a typical wireless network. However, packet losses due to transmission errors and mobility can adversely impact TCP performance, as we discuss later in this chapter. This chapter discusses the impact of such packet losses on TCP throughput, followed by selected approaches to improve TCP performance over wireless networks. We begin this chapter with a brief overview of salient features of TCP.

## 9.2 TCP Overview

Transmission control protocol (TCP) provides reliable in-order delivery of packets from a source to a destination. Also, TCP provides an end-to-end semantics, which implies that the receiver receives an acknowledgement for a packet only after the packet has been delivered to the destination. The jargon often used to refer to the TCP packets is "segment". However, we will use the term packet in our discussion. In addition to reliable in-order delivery, an important element of TCP is its congestion control mechanism. TCP is designed to respond to congestion in the network. Congestion occurs when the rate at which traffic is injected into the network exceeds the capacity of the network. With sustained congestion, the packet queues at some links in the network begin to grow, eventually resulting in packet

loss due to buffer overflow. TCP responds to such packet losses by reducing the sending rate, as discussed in this section.

We begin our overview on TCP with a discussion of acknowledgements used in TCP. Note that the description of TCP in this chapter is meant as a review; for more details, please refer to other texts devoted to TCP.

### 9.2.1 TCP Acknowledgements

Our discussion of TCP typically assumes that the TCP data flows from a certain sender to a certain receiver. In practice, the TCP connections are bi-directional (in the sense that the data can flow in both directions). However, our discussion will typically consider data flowing in one direction, with one of the end hosts identified as the TCP sender, and the other end host identified as the TCP receiver.

When a TCP receiver receives a packet from the sender, it may respond by sending an acknowledgement (abbreviated as Ack). The acknowledgement in TCP is cumulative in the sense that it acknowledges not just the receipt of a single packet, but the receipt of all packets up to a certain point in the TCP flow. For instance, the TCP ack may acknowledge that the receiver has received the first 20000 bytes. TCP sender assigns a sequence number to each byte of data sent by the sender. The receivers acknowledges the receipt of first 20000 bytes (bytes numbered 0 through 19999) by storing sequence number 20000 in the ack packet, which can be viewed as the next in-sequence byte expected by the receiver, but not yet received.

For future reference, note that all our examples will assume that each TCP data packet contains 1000 bytes. Thus, the first byte of each data packet will have sequence number equal to L thousand, for some non-negative integer value L. When discussing (or pictorially depicting) a packet, we will refer to data packet as L to indicate that it contains byte L*1000 through (L*1000+999). We will also use a similar abbreviated notation for acknowledgements. Since all data packets in our example will contain 1000 bytes, the TCP ack will always contain a sequence number of the form (L+1)*1000, indicating that the receiver has received all packets up to and including packet L, each containing 1000 bytes. We will label an ack simply as L in this case.

Using the above notation, Figure 9.1 shows acks sent by TCP receiver D in response to data packets received from TCP sender S. Host B is an intermediate host on the route from S to D. The data packets are shown above the links in the figure, whereas TCP acks are shown below the links. The status in Figure 9.1(a) is that D has already received data packets 0 through 36 and most recently sent an ack labeled 36, after having received data packet 36. Ack 36 acknowledges the receipt of all packets through packet 36 (again, recall that TCP uses byte sequence numbers, however, our notation is somewhat different).

Subsequently, when host D receives data packet 37, it sends Ack 37 in response, since host D has now received all packets through packet 37.

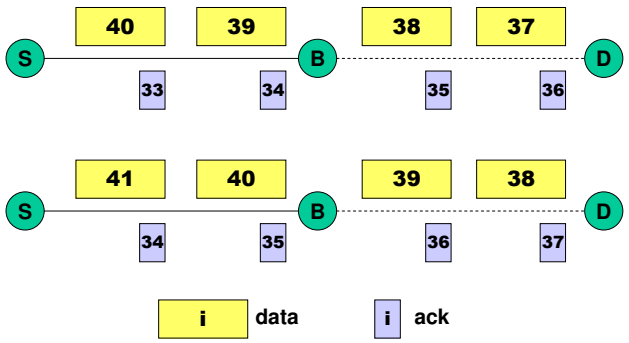



**Figure 9.1** Illustration of TCP acknowledgements (Ack)

In our example in Figure 9.1, packet 37 is the next in-sequence packet that host D expects after packet 36. Also, in this example, the receiver generates an ack after receiving each in-sequence packets. In reality, TCP receiver need not send an acknowledgement for each in-sequence packet. The "delayed ack" mechanism allows the TCP receiver, on receipt of a data packet, to delay sending ack until it receives a second data packet, or until a suitable timer expires, whichever occurs first. Thus, with delayed ack, the receiver may possibly send one ack for every other data packet received, reducing ack traffic, and the load on the reverse route from D to S. Such delaying of acks is acceptable, since the acknowledgements are cumulative. Figure 9.2 provides an illustration of delayed acks. In the first snapshot in the figure, host D has sent Ack 35 on receipt of data packet 35, but does not send any ack on receipt of packet 36. Subsequently, as shown in the second snapshot, when D receives data packet 37, it sends Ack 37, acknowledging all packets through 37, including data packet 36.

What happens when a packet is lost, for instance, due to transmission error on a wireless link? Consider the example in Figure 9.3. As shown in the first snapshot, D has just sent Ack 36 on receipt of data packet 36. The next in sequence packet 37 is lost due to transmission errors on the wireless link between hosts B and D (the dotted line depicts a wireless link). Thus, the next packet that host D receives reliably is packet 38. However, host D is not expecting this packet, hence, this is considered to be an out-of-order (OOO) packet. In response to such an OOO packet, as shown in the second snapshot in Figure 9.3, host D again sends Ack 36 to indicate that it has received all data through packet 36. Since host D has already once sent Ack 36, the second Ack 36 is said to be a *duplicate acknowledgement*, or dupack for short.

In the example in Figure 9.3, when host D receives packet 39, it will yet again send a dupack 36, since 39 is OOO, and the last in-sequence data packet received was 36. While the packet loss in our example was caused due to errors on a wireless link, such a loss can occur due to congestion as well (perhaps the more common case in many networks). For instance, packet 37 could have been lost if the buffer at host B were full when that packet arrives at B from S. In this case as well, the dupack shown in Figure 9.3 will be generated.

While packet loss can lead to generation of dupacks, out-of-order delivery of data also leads to dupacks, as illustrated next. Out-of-order delivery of TCP data packets can occur because IP (internet protocol) does not guarantee ordered delivery. Consider the illustration in Figure 9.4. The first snapshot in this figure shows that host D has just sent Ack 36 on receipt of packet 36 (all previous data packets have been received by D as well). Observe that on link BD, the order of packets 37 and 38 is switched; packet 37 is being delivered out of order by IP. In this case, when D receives packet 38, it will send dupack 36 (see the second snapshot in Figure 9.4). Later, on receipt of packet 37, host D will send Ack 38, since it has now received all packets through 38; this ack is labeled as new ack in the third snapshot in Figure 9.4.

Now, the number of dupacks that will be generated depends on how much out-of-order a packet is. Let us consider Figure 9.5. In this case, packet 37 is two places away from

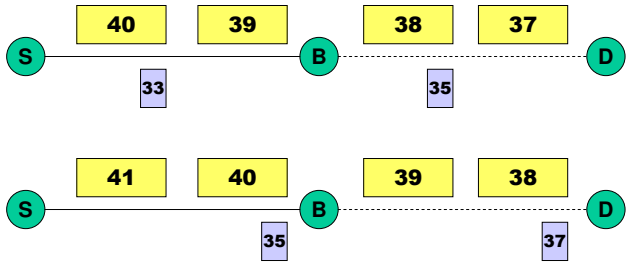



**Figure 9.2** Illustration of delayed acknowledgements

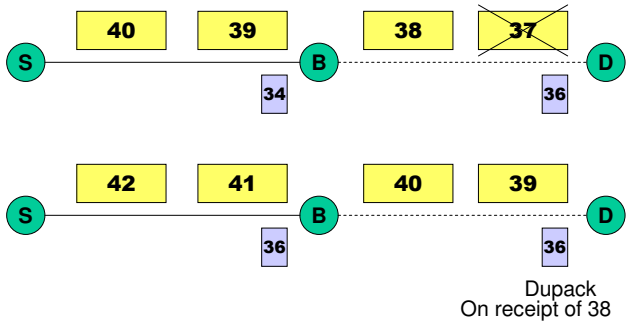



**Figure 9.3** Illustration of duplicate acknowledgement

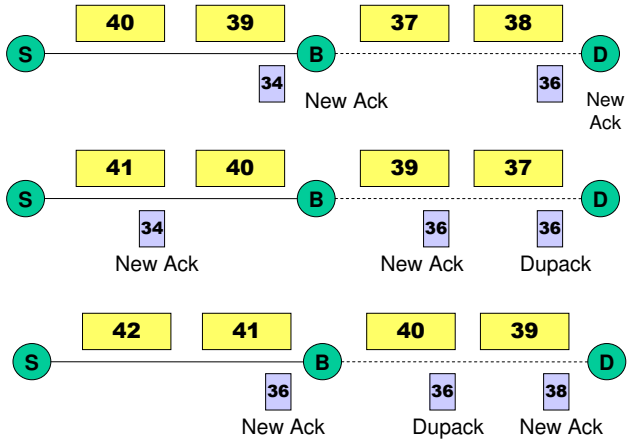



**Figure 9.4** Out-of-order delivery by network layer can lead to dupacks

its nominal position. Packet 37 should be delivered after packet 36. Instead, the network layer delivers the packets in the following order: 36, 38, 39, 37. Suppose that, as shown in the first snapshot in Figure 9.5, the receiver has already received packets through 36, and has just sent Ack 36. Subsequently, receiver D receives packets 38 and 39, respectively, and sends Ack 36 (dupacks) in response to each of these out-of-order packets. Note that dupacks are not delayed by the TCP receiver. In the last snapshot in Figure 9.5, host D receives TCP data packet 37, and then sends the new ack Ack 39, since it now has all TCP data packets through packet 39. Observe that, since packet 37 was delivered out-of-order by two positions, two dupacks were sent by the TCP receiver between the new acks Ack 36 and Ack 39.

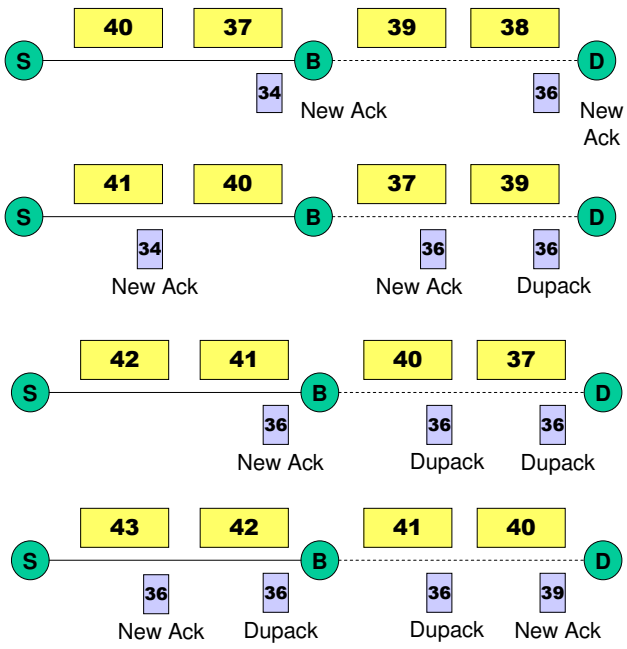



**Figure 9.5** Number of dupacks depends on how much out-of-order a packet is delivered

### 9.2.2 Detection of Packet Loss

TCP uses two approaches for detecting a packet loss: timeouts and duplicate acks. Let us consider the timeout mechanism first. Our discussion will adhere to TCP specification in spirit, but not in the exact details. Essentially, when TCP sender sends a data packet L, it sets a timer to go off after a suitable retransmission timeout (RTO) interval. If an ack that acknowledges packet L is not received before the timer expires, then the TCP sender assumes that packet L is lost. Note that Ack M will acknowledge packet L so long as L $\leq$ M. After the loss of packet L is inferred on a timeout, the TCP sender retransmits packet L, and sets the retransmissions timer again. However, the RTO value is now chosen to be double the previously used value. This is referred to as exponential backoff. If the timer were to expire again before getting an ack for packet L, then the packet would be retransmitted yet again, again doubling the timeout value. Thus, consecutive failures to receive an ack for a data packet result in successively longer timeout intervals (RTO) being used for the retransmission attempts.

The above exponential backoff procedure explains how the timeout interval is chosen for a retransmission. How do we choose the timeout for the first transmission of a packet? If the timeout value is too small, the sender will unnecessarily timeout, and retransmit a packet that is not lost. On the other hand, if the timeout interval is too large, then the delay before detection of a packet loss will be larger. To address these concerns, TCP chooses the timeout interval as a function of the actual delays in the network. On receipt of each ack, the TCP sender can determine an estimate for the round-trip time (RTT), which is the time required for a data packet to reach from S to D, and an ack acknowledging that packet to reach from D to S (the estimated RTT is affected by the use of the delayed ack mechanism). The TCP sender maintains a moving average of the RTT samples, and also keeps track of the *mean deviation* (similar to standard deviation, but simpler to calculate) of the RTT. The RTO value for first transmission of a packet is chosen as *mean RTT + 4*mean deviation*. Thus, the RTT estimates as well as the variance of the RTT estimates both affect the RTO value. In addition, a lower bound is imposed on the RTO value, independent of the RTT samples.

The RTO interval used in the above timeout mechanism is often much longer than the actual instantaneous RTT, since RTO accounts for variance in RTT as well. This can potentially lead to long delays in detection of packet loss. An alternative mechanism is suggested by the illustration of dupacks caused by a packet loss. From the earlier examples, it should be clear that when a packet (or a sequence of consecutive packets) is lost, the next TCP data packet received by the TCP receiver will trigger a dupack from the receiver. On receipt of this dupack, say, dupack L, the source can infer that packet L+1 is lost. While this may appear to be a reasonable conclusion, sometimes the conclusion is erroneous. In particular, even when a packet is not lost, but merely delivered out-of-order by the network layer, a dupack is generated by the receiver as we have seen previously. Thus, a dupack may result due to a lost packet or out-of-order delivery by lower layers. Assuming that every

dupack is an indication of a packet loss will result in an erroneous conclusion whenever the network layer delivers packets in the wrong order. Clearly, in case of out-of-order delivery, there is no need to retransmit any packet, since no packet is lost. TCP uses a somewhat arbitrary rule to try to differentiate between the two causes. In particular, if the sender receives at least 3 dupacks consecutively, say, dupack L, then the sender concludes that packet L+1 is lost. Thus, even if the network layer delivers packets out-of-order, so long as a packet is not delivered "too" out-of-order (specifically, within two positions of its nominal position, leading to at most two dupacks), TCP sender will not mistake out-of-order delivery for packet loss.

Detection of packet loss following 3 dupacks can often occur much sooner than detection using the timeout mechanism, particularly when packet losses occur in small bursts (including the case of single packet loss). However, when losses occur in large bursts, the loss may not be detected using dupacks. For instance, consider the possibility that link BD fails for a long interval of time, causing loss of all in transit TCP data packets starting from packet 37 (lower layers will normally discard packets when the link failure is detected). In this case, since data packets are not reaching the receiver, it will not generate dupacks (generation of dupacks requires the receiver to receive an out-of-order packet). Thus, in this case, the TCP sender will detect the packet loss only when a retransmission timeout occurs.

### 9.2.3 Congestion Control

TCP sender uses a sliding window mechanism to determine which packets may be transmitted to the receiver. Only packets within this window may be transmitted by the sender. When a new ack is received, the window slides to include new data. For instance, Figure 9.6 shows the sender's sliding window of size 5 packets before and after the sender receives ack 6 (we will specify window size in number of packets, however, in reality, TCP measures window size in bytes, since TCP data packets are not fixed size).

The size of sender's sliding window is bounded by two parameters: buffer space available at the receiver, and congestion window maintained by the sender. The buffer space available at a TCP receiver is communicated to the TCP sender by means of the *receiver's advertised window* field included in TCP acks. For simplicity, we will assume that the sliding window size is equal to the congestion window (that is, the buffer size at the receiver is not a constraint). Under this assumption, in our subsequent discussion, we will refer to the sender's window as the congestion window.

We say that a data packet is "outstanding" when the packet has been sent by the TCP sender, but it has not received an acknowledgement for that packet. Since only the packets within the congestion window can be transmitted at any given time, only a congestion window worth of data can be outstanding at any time. That is, only the packets in the congestion window could have been transmitted, but not yet acknowledged. Another way to

look at this is that (for our example with window size of 5 packets), after sending the first packet (with smallest sequence number) within the congestion window, the sender may send at most 4 more packets (since congestion window size is 5 packets) without having received an ack for the first packet. Since it requires RTT duration to receive an ack after sending the data packet, the above discussion implies that, within one RTT, the sender may only transmit one congestion window (cwnd) worth of data. Thus, the throughput achieved by TCP is upper bounded by cwnd/RTT. This observation is used to implement the congestion control mechanism in TCP. In particular, observe that, if cwnd is chosen to be too small, then the available capacity may not be utilized. On the other hand, if cwnd is too large, then the TCP sender will send too many packets within a RTT leading to queueing in the network, and possibly packet loss. What is the right window size? Recall that the sliding window mechanism allows the sender to send all the packets within the congestion window even if the first packet in the window is not acknowledged. Given that the round-trip time is RTT, and the capacity of the route from S to D is C, what is the right congestion window size? For simplicity, let us assume that the acks are sent on a separate route, and that route is not a bottleneck to TCP performance. Now, the time required for the first data packet in the congestion window to reach the receiver, and for the resulting ack from the receiver to reach the sender will be the round-trip time RTT of interest here. How much data can the sender possibly deliver to the sender during this RTT interval, if the route capacity is C ? Clearly, the amount of "outstanding data" (that has not been acknowledged as per the sender's knowledge) can be at most RTT*C. If the sender attempts to send more data than RTT*C, the excess amount will have to be buffered at the bottleneck link on the route (a link with capacity C can only transmit RTT*C during RTT interval). On the other hand, if the sender sends less than RTT*C data during the RTT interval, the route capacity will be under-utilized. The quantity RTT*C is often called the "delay-bandwidth" product for the route. The term bandwidth, strictly speaking, refers to a slice of the frequency spectrum; however, the term is also loosely used to refer to the available capacity. The intended meaning of the term should be clear from the context. The above discussion, thus, implies that the ideal congestion window size is equal to the delay-bandwidth product for the route taken by the TCP flow.

Instead of using estimates of delay and bandwidth to predict the suitable congestion window size, TCP uses a probing mechanism. In particular, on receipt of each new ack (which indicates successful delivery of new data to the receiver), the TCP sender increases its congestion window. On the other hand, whenever a packet loss is detected (either by receipt of 3 dupacks, or by a timeout), the TCP sender decreases the congestion window. TCP uses a *additive increase, multiplicative decrease* algorithm for adjusting the congestion window. That is, on successful delivery of new data, the congestion window is increased (additively) by a small amount, while on packet loss, the congestion window is decreased much more dramatically (multiplicatively), reducing to half its value (on receipt of 3 dupacks), or to the minimum possible value (on a timeout) of 1 Maximum Segment Size (MSS), which we will refer as 1 packet. The window growth on the reception of a new ack occurs at different rates

depending on whether the congestion window is less than a so-called *slow start threshold (ssthresh)* or not. The slow-start threshold (ssthresh) itself is also modified on a packet loss. We will omit the discussion of these details here. Figure 9.7 depicts the idealized evolution of congestion window for a hypothetical TCP connection, with time being measured in RTTs. Note that while cwnd is less than ssthresh, the cwnd doubles after each RTT (during the so-called slow-start phase), whereas it grows linearly with time during the congestion avoidance phase that follows. The important point to remember here is that the congestion window decreases significantly after a packet loss.

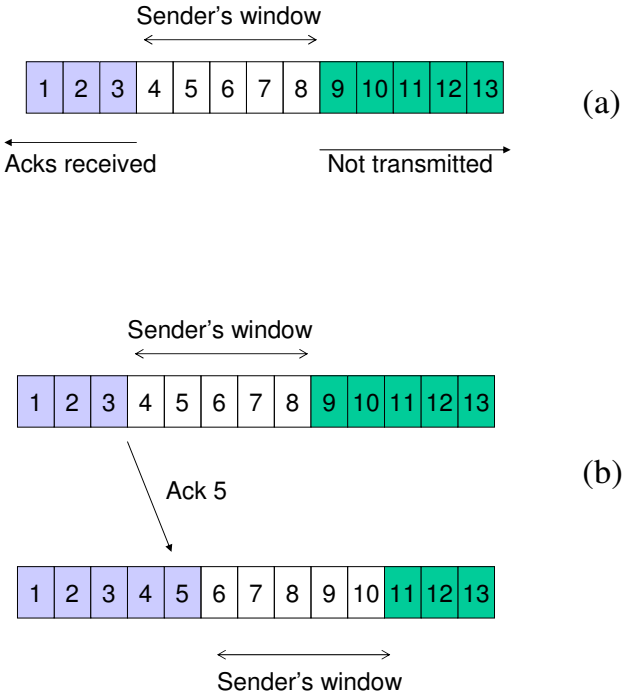



**Figure 9.6** Sliding window

On the detection of a packet loss due to three dupacks, the sender retransmits the lost packet, as seen previously. This is called *fast retransmit*, since the loss detection is presumably faster than that using a timeout. The fast retransmit is followed by fast recovery, during which the cwnd grows linearly with time. Important issue to note here is that, after the packet loss, at the start of fast recovery, the congestion window is reduced to half its value prior to the detection of the packet loss (by three dupacks).

With the above discussion, we conclude our brief overview of TCP. Readers who are unfamiliar with TCP may wish to read other texts on this topic for further details.

## 9.3 Performance of TCP over wireless links

We will consider two types of wireless networks. Our initial discussion will focus on the case of TCP flows for which the last link on the route from the TCP source to TCP receiver is a lossy wireless link. In Figure 9.8, host S is the TCP source, and D the TCP receiver. Link BD is a wireless link that is prone to transmission errors. The first snapshot in the figure shows that host D has just sent a new ack 36, and that data packet 37 is lost on the wireless link due to transmission errors. In the second snapshot, no new packet is reliably received by host D, hence it does not send any ack (the *new ack* shown in snapshot (b) is the new ack sent in snapshot (a)). As snapshots (c), (d) and (e) show, host D sends dupack 36 on the reception of packets 38, 39, and 40, respectively. When these three dupacks reach host S, it will detect the loss of packet 37, and perform fast retransmit. Since host S has detected packet loss on receipt of 3 dupacks, the congestion window used by host S will be reduced in half.

Now, the reason for reducing congestion window in response to a packet loss is that TCP conservatively treats each packet loss as having occurred due to congestion. Reducing congestion window reduces the load on the network, alleviating the congestion. However, when packet loss occurs due to transmission errors unrelated to congestion, TCP's response is still to reduce congestion window quite significantly. The reduced window may quite possibly be much smaller than the ideal cwnd value. This leads to lower throughput until the cwnd eventually grows closer to the optimal value. Figure 9.9 shows the throughput of a TCP connection for a 2 Mbps full-duplex wireless link. The horizontal axis in the graph shows inverse of the error rate, and vertical axis shows TCP throughput. In this case, no packets are lost due to congestion. TCP throughput drops with increasing error rate. Of course, with increasing error rate, we expect the throughput to degrade, since not all packets are delivered reliably on the wireless link. For instance, if 10% of the TCP data packets are lost on the wireless link, then one may anticipate 10% throughput degradation compared to the performance on a loss-free link. However, as seen in Figure 9.9, the throughput degradation can be far worse. This result occurs due to the TCP sender reducing its congestion window

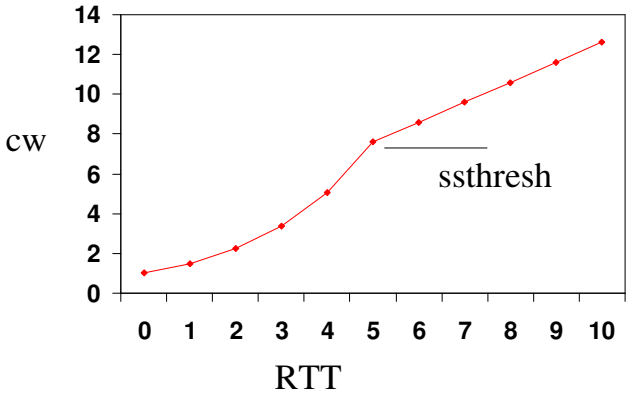



**Figure 9.7** Congestion window dynamics

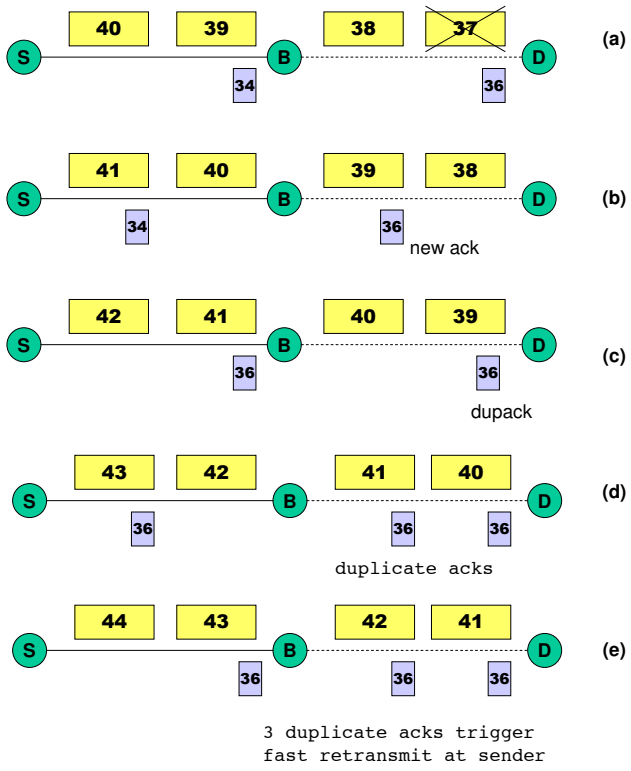



**Figure 9.8** Fast retransmit due to packet loss

in response to the packet losses on the wireless link, even though the losses may not have occurred due to congestion.

Packet losses due to transmission errors can also cause a timeout at the TCP sender. This may occur, for instance, if several data packets are lost consecutively on the wireless link. If an insufficient number of packets from a congestion window is delivered reliably, then the resulting dupacks (if any) may not be sufficient to result in a fast retransmit. Thus, the only way the TCP sender will detect a packet loss in this case is when a timeout eventually occurs. Again, TCP sender will reduce its congestion window to a small value, which is not always necessary in response to transmission errors, since the network may not really be congested.

Much of the rest of this chapter deals with approaches to improve TCP performance over routes in which the last hop is an error-prone wireless link, and other links are error-free. We will refer to such routes as "last-hop wireless routes". In addition, towards the end of the chapter, we will address the issue of TCP performance over multi-hop wireless networks.

## 9.4 Improving TCP Performance over Last-Hop Wireless Routes

As the above discussion indicates, the TCP throughput degrades over wireless links because of two factors:

- Packets losses occur due to a cause (specifically, transmission errors) that is unrelated to congestion, and
- TCP responds to such packet losses by reducing the congestion window, which is an inappropriate response to non-congestion related packet losses.

The above simple observation suggests different ways of improving TCP performance. We first present two classifications of the different approaches, followed by some example techniques for improving TCP performance.

### Classification 1

This classification is based on the type of action taken to improve TCP performance. Two types of actions can be designed to address each of the factors above:

- Hide wireless packet loss from the TCP sender: In this case, since the TCP sender will be unaware of packet losses due to transmission errors, it will not respond inappropriately, avoiding significant throughput degradation.

- Make TCP sender aware of the cause of packet loss: In contrast to the above approaches, the second class of schemes allow the TCP sender to become aware of the packet losses due to errors, but also provide the TCP sender additional information about the cause of the packet losses. Knowing the cause, the TCP sender can respond appropriately for each type of packet loss (i.e., reduce congestion window only if the packet loss is deemed to be due to congestion).

**Classification 2**

Clearly, to improve TCP performance in presence of errors, we may need to make some protocol changes. The second classification is based on the location where such changes are made:

- Modify TCP sender only: A significant amount of wireless TCP traffic is likely to be from web servers to wireless web clients. In such cases, much of the data is transferred with the web server acting as the TCP sender. If by making changes to the TCP implementation at the sender we can improve TCP performance, then this improvement is attained without having to change the code at large number of wireless client devices.

- Modify TCP receiver only: Knowing that the wireless device will operate as the TCP receiver for the most part, the designers may want to incorporate wireless-specific optimizations in the TCP receiver code. Such changes can potentially benefit TCP performance without having to modify the code on other (non-wireless) devices.

- Modify only intermediate hosts: For instance, if performance of TCP can be improved by incorporating intelligence only at intermediate nodes (for instance, host B in our examples), then such benefits can potentially be attained without modifying TCP at all.

- Hybrid schemes combining different approaches above.

In addition to the location where changes are made, the protocol layers at which the changes are made also affect usefulness of the schemes. Several schemes to improve TCP performance have been proposed in the past. In this chapter, we discuss a few selected schemes, beginning with link layer mechanisms.

## 9.5 Link Layer Recovery

The degradation of TCP performance over wireless links occurs due to transmissions errors that result in packet losses. A natural solution to this problem is to improve the quality of the wireless link, either by reducing the occurrence of packet losses due to errors, or by hiding such losses from the TCP sender. Two link layer approaches to recover from errors may be used: forward error correction and link layer retransmissions.

### Forward Error Correction (FEC)

In this case, the sender encodes the transmitted packet, incorporating some redundancy, such that the packet can be decoded correctly by the receiver despite the occurrence of some transmission errors. We discussed Hamming codes in Chapter 2. In practice more complex codes are typically used for forward error correction in wireless networks.

### Link Layer Retransmissions

An alternative to FEC is to incorporate only limited redundancy to allow the receiver to detect errors in the received packet, as opposed to being able to correct the errors. The level of redundancy required is again a function of the desired error detection capabilities. If we want to guarantee that up to $t$ bit errors can always be detected, then it is necessary to employ a code with minimum distance $t + 1$. Observe that $t$-error detection requires significantly smaller minimum distance (and hence lower level of redundancy) compared to $t$-error correction.

Once an error is detected (by the receiver) in the received packet, the transmitter (at the other end of the wireless link) can be informed to retransmit the corrupted packet. The mechanism to inform the transmitter may be implicit, such as absence of a link layer acknowledgement, or explicit, such as a negative (link layer) acknowledgement. For our illustrations, let us assume that the link layer at the receiver sends a link layer acknowledgement for each link layer packet (often referred to as a frame) received correctly, and sends no response when a corrupted frame is received (this is similar to the IEEE 802.11 acknowledgement mechanism). The benefit of link layer retransmissions is that, in absence of errors, this approach incurs little overhead (only to facilitate error detection). Additional overhead in the form of retransmissions is incurred when errors actually occur. The retransmissions do incur additional delay. However, since link layer retransmissions provide "*local recovery*" from errors, there is potential for recovering from the packet loss sooner than using the "end-to-end" recovery performed using TCP retransmissions.

We illustrate the advantage of local recovery over end-to-end recovery using Figures 9.10 and 9.11. Figure 9.10 shows the case when link layer retransmissions or FEC are

not employed. In this case, the TCP source S sends data packets 6, 7, 8 and 9 at time 1000 ms, 1002 ms, 1004 ms and 1006 ms, respectively, which reach host B at 1040 ms, 1042 ms, 1044 ms, and 1046 ms, respectively. Packet 6 is lost due to transmission errors on link BD, but packets 7, 8 and 9 reach the receiver at 1043 ms, 1045 ms and 1047 ms, respectively (we assume a much shorter delay on link BD as compared to route from S to B, which may contain multiple hops not shown in the figure). The duplicate acks sent by receiver D in response to receipt of packets 7, 8, and 9 (since packet 6 was lost) reach source S at 1084 ms, 1086 ms, and 1088 ms, respectively. Thus, at 1088 ms, source S can retransmit packet 6, 88 ms after its original transmission. This retransmitted packet will reach D 42 ms later at time 1130 ms (in the absence of further packet loss).

Now consider Figure 9.11. In this case, after having sent packet 6 to host S at 1040 ms, the link layer at host B expects to receive a link layer acknowledgement (not shown in the figure) for packet 6 by 1042 ms (allowing for 2 ms round-trip delay on the wireless link). Since packet 6 was corrupted on the wireless link, host D does not send a link layer ack for this packet, and host B will retransmit the packet at 1042 ms, which will reach D around 1043 ms. Thus, the *recovery* of the lost packet can occur by 1043 ms with link layer retransmission scheme, as opposed to 1130 ms with the end-to-end recovery by TCP.

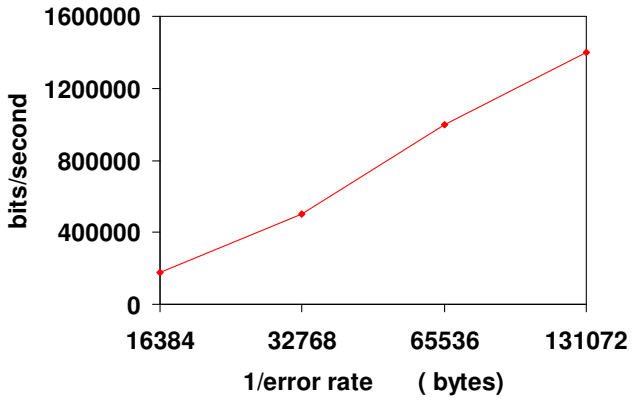



**Figure 9.9** Performance of TCP in presence of errors: The results presented here correspond to a certain two-hop network, with route S-B-D used between source S and destination D, with link BD being a wireless link.

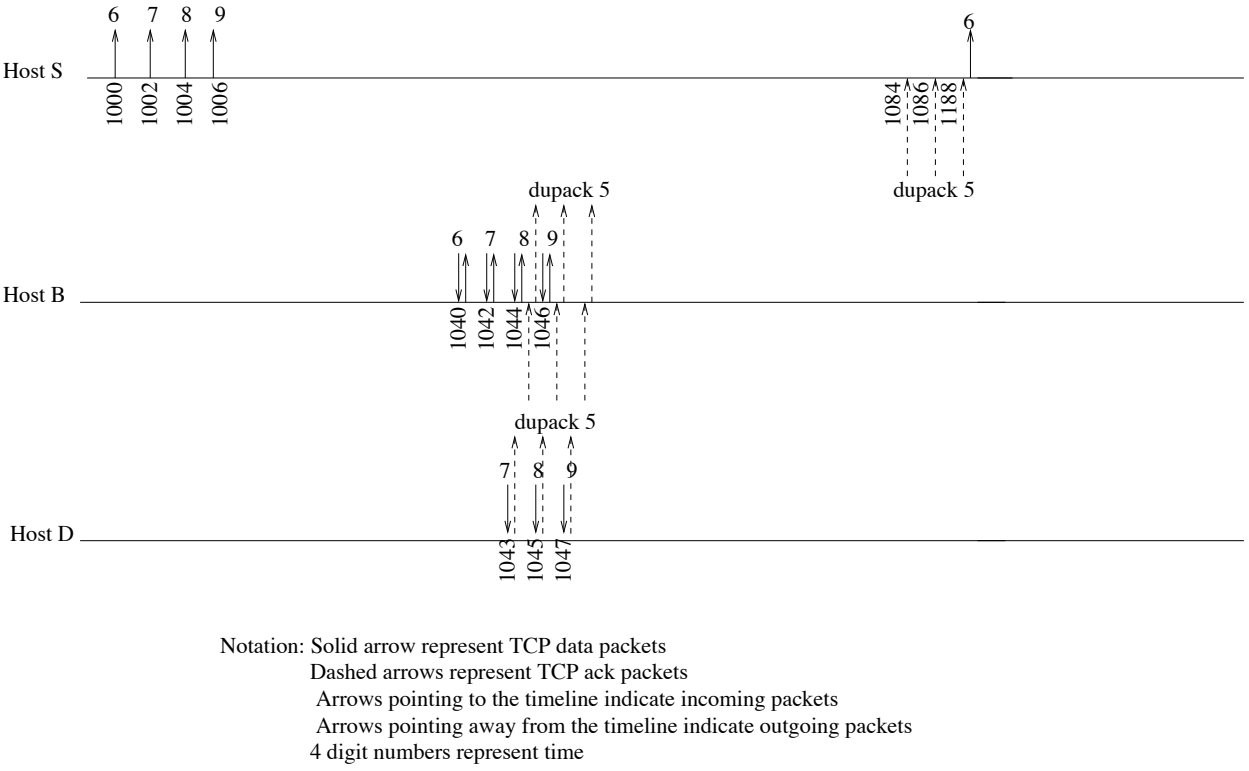



**Figure 9.10** End-to-end recovery using TCP retransmissions

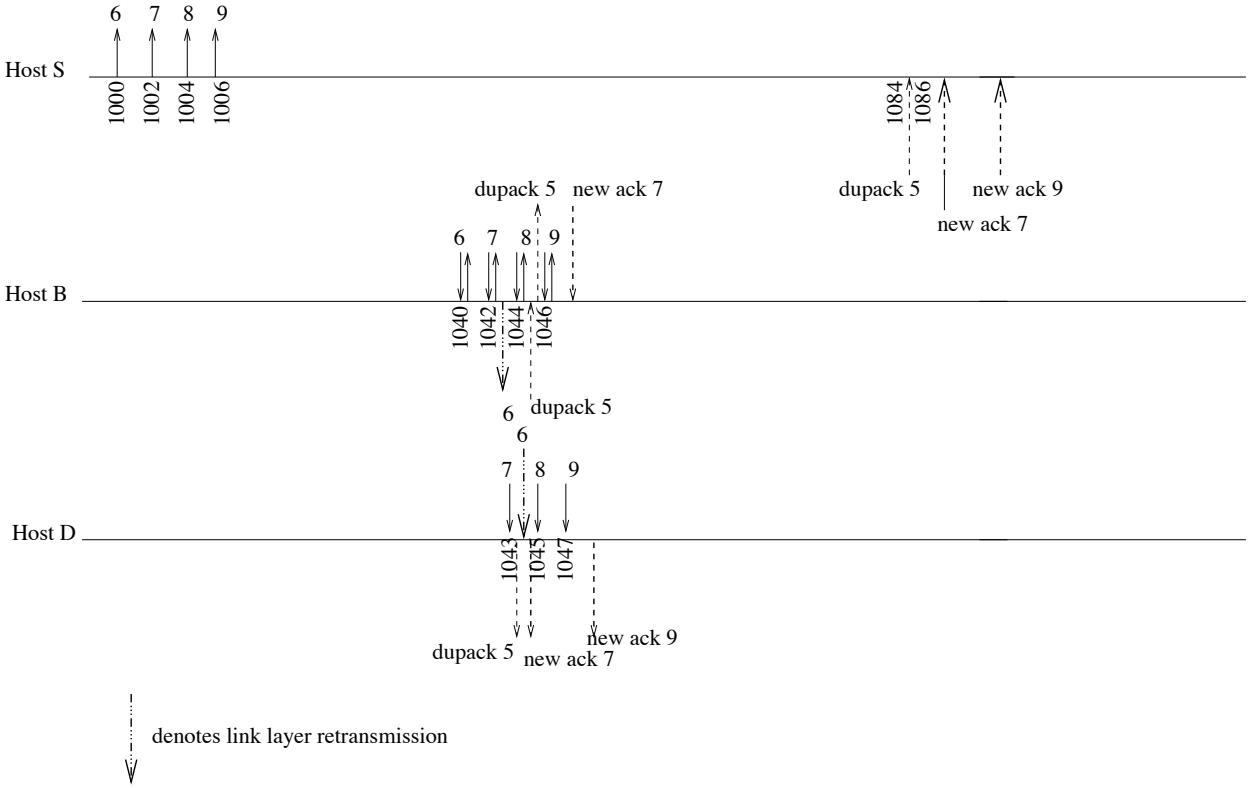



**Figure 9.11** Benefit of local recovery using link layer retransmissions

Figure 9.12 pictorially illustrates the solutions that use link layer transmissions. The figure shows protocol layers at hosts S, B and D, where S is the TCP sender, D is the TCP receiver, and B is an intermediate host, with link BD being a wireless link. Link layer retransmissions (rxmt) are incorporated on the wireless link. To be able to implement the link layer retransmission, the transmitter on the wireless link, namely, host B, must maintain a buffer for packets that have been transmitted on the wireless link, but are not known to have been delivered successfully. This buffer is shown in the figure as link layer state. Note that a single buffer is used, independent of the number of TCP connections to host D via host B. In essence, the link layer at host B does not distinguish between different TCP connections. Similarly, the link layer at host D must maintain a buffer too, for packets that may be sent from D to B at the link layer; this buffer is not shown in the figure. Observe that the packets from S to D reach up to the network layer at host B (for routing purposes), and then go down the layers again until reaching the physical layer, and then transmitted on the wireless channel.

As seen above, link layer retransmission can be implemented provided that an error detection mechanism is also provided to allow detection of corrupted packets. In general, a combination of FEC and retransmissions may also be used. FEC may be used to recover from a small number of errors without having to retransmit a packet, whereas the corrupted packet can be retransmitted in case of excessive errors.

Let us now identify the circumstances under which link layer transmissions is likely to help improve TCP performance substantially. One issue that affects the performance improvement is whether the link layer delivers packets in-order, or as they arrive (potentially out-of-order).

### In-Order versus Out-of-Order Delivery at the Link Layer

Figure 9.13 illustrates in-order delivery of frames at the link layer, despite retransmissions. The frames show two sequence numbers, the number shown on the left in a packet is a link layer (LL) sequence number, and the other one is a TCP sequence number. The link layers normally use their own numbering scheme for the frames, independent of the protocol that may be used at the transport layer. To avoid ambiguity in this discussion, we will prefix a sequence number with LL or TCP. In our example in Figure 9.13, frame LL-23 (that is, the frame with link layer sequence number 23) is corrupted on its first transmission; however, the retransmission is delivered correctly. The order in which frames are received by host D's link layer is shown in the figure. Also shown is the order in which packets are delivered by the link layer to the higher layers at host D. Notice that, although frame LL-24 was received prior to the correct reception of frame LL-23, the delivery of this frame is delayed until frame LL-23 is reliably received. This delay is necessary to ensure ordered delivery of frames from the link layer to the higher layers.

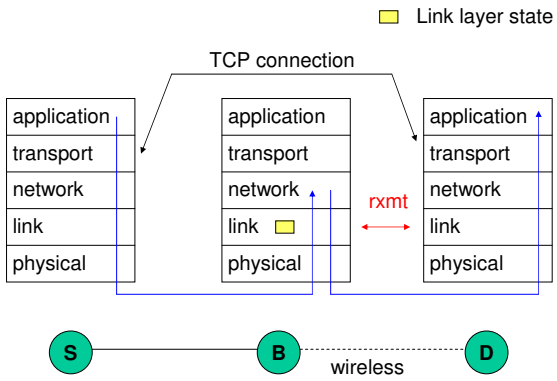
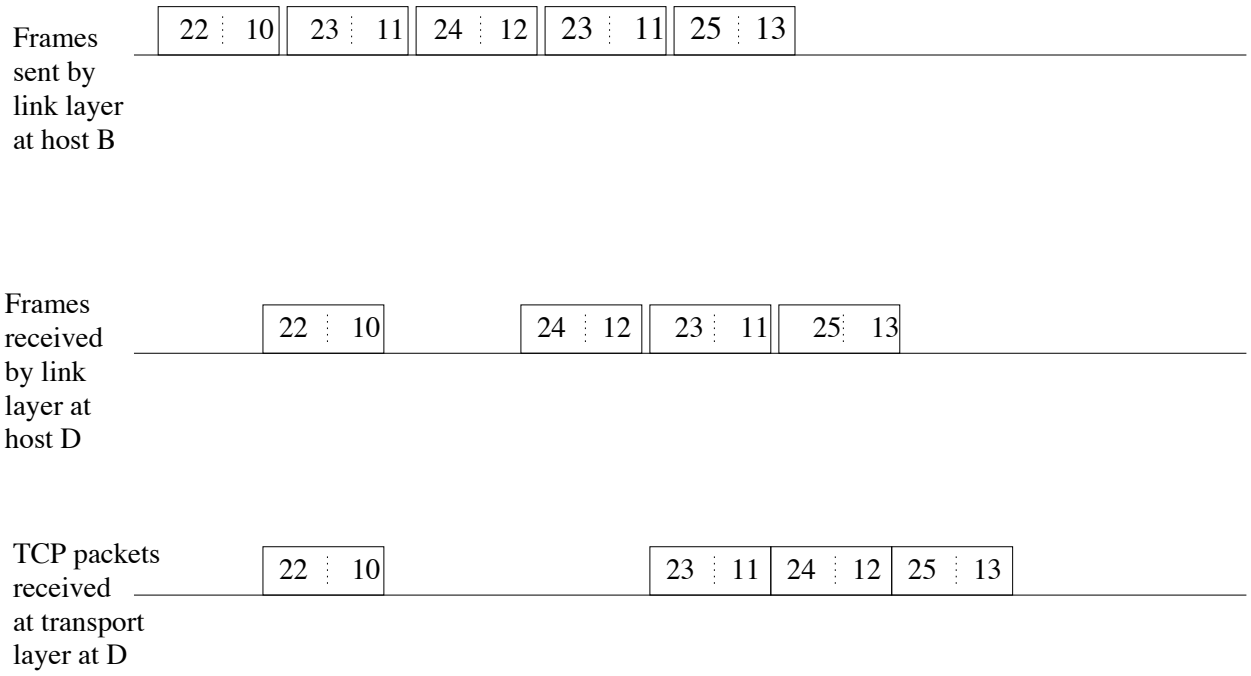



**Figure 9.12** Protocol stack illustrating link layer retransmissions




(link layer headers, although shown here, are removed before sending the packets to transport layer)

**Figure 9.13** In-order delivery at the link layer

Why would host B transmit frame LL-24 prior to retransmitting frame LL-23? Why not use the stop-and-go protocol at the link layer? In the stop-and-go protocol, the transmitter (host B in our case) does not transmit a new frame until it has been able to deliver the previous frame reliably, possibly after retransmissions. Thus, after host B has sent frame LL-23 to host D, host B expects to get an acknowledgement (link layer ack) from host D. Failing to receive the ack within a suitable time interval, host B will retransmit the frame. This will continue until host B is able to receive an ack for the frame (alternatively, after certain number of attempts, host B may deem the link as broken, drop the frame, and inform the upper layer of the link failure). Such a stop-and-go protocol ensures that the link layer at host D will not receive frame LL-24 before frame LL-23 (unless frame 23 is dropped). Thus, the link layer achieves in-order delivery. In this case, there is no need to delay received frames at the link layer at host D, since the received frames will be necessarily in-order. In fact, this is indeed the approach taken in some practical systems, notable among them is IEEE 802.11a/b/g. In case of IEEE 802.11a/b/g, the maximum distance between hosts is up to a few hundred meters, limiting the propagation delay to be a few microseconds. Also, the maximum bit rate is under 100 Mbps. Clearly, in this case, the delay-bandwidth product of the wireless link is quite small, and the use of a stop-and-go protocol does not sacrifice performance significantly (the per-frame link layer acknowledgements do incur some performance cost).

However, the stop-and-go protocol is not appropriate for links with higher delay-bandwidth products. For instance, consider a wireless link where the link delay is of the order of 10 ms, and the bit rate is 10 Mbps. For this link, the delay-bandwidth product is 100 Kbits, which is equivalent to many 1000 byte frames. Note that the link delay may also be due to processing delay at the receiver (e.g., decoding), in addition to propagation delay. For such a link, if we use stop-and-go protocol, the link will be kept idle most of the time, reducing efficiency (and throughput) significantly. On links with large delay-bandwidth product, a sliding window protocol (similar to TCP) is more appropriate than stop-and-go. With the use of a sliding window at the link layer, the scenario depicted in Figure 9.13 can potentially occur. Specifically, if the delay-bandwidth product for the link layer is of the order of $W$ frames, then a transmission of a given frame, and its retransmission (or two consecutive retransmissions) may be separated by $W - 1$ frames, if we want to maximize the channel utilization.

When $W$ is larger than 1, should we require the link layer to deliver frames in-order? It is important to emphasize here that the ordering we are referring to is at the link layer. If, somehow, host B in our example receives packets from S out-of-order, then that is the order in which the link layer at B will deliver them to D. When $W$ is large, the link layer should use a protocol such as the sliding window protocol to maximize efficiency. At the same time, the link layer can provide in-order delivery by the use of buffering. In our example above, the link layer at host D buffers frame 24 until frame 23 is received reliably. When this frame is received reliably, frames 23 and 24 are released to the upper layer in that order, achieving

in-order delivery at the link layer. So long as the link layer delivers frames in-order, the transmission errors on the wireless link will not cause dupacks from the TCP receiver at host D. So, does in-order delivery solve our problem? There are two potential concerns:

- In-order delivery requires buffering of packets as we have seen above. In addition, as seen in our example, there is also some delay in delivery of the packets, since the delivery must wait for the retransmission to succeed We can impose an upper bound on how long a packet may be buffered at the receiver, in order to avoid waiting indefinitely in case of link failures. Potentially, the delay can become long enough that the TCP sender may timeout while the wireless link layer is attempting to deliver the packet to the receiver. This would result in retransmission by the TCP sender, in addition to the link layer retransmissions. This phenomenon is sometimes referred to as "interference" between TCP retransmissions and link layer retransmissions (this should not be confused with interference in SINR calculations). The likelihood of TCP timeout, while the link layer is still attempting to retransmit the packet, is higher when the wireless link delay represents a large fraction of the end-to-end delay, or when link layer retransmissions of a packet are also erroneous.

- Not all traffic on the wireless link may require in-order delivery of information, or be able to tolerate the delay incurred due to buffering for in-order delivery. In particular, for voice traffic, an application layer (or transport layer) redundancy mechanism may be used to improve reliability, instead of relying on link layer mechanisms. In this case, imposing reliable in-order delivery at the link layer may worsen the perceived voice quality. One solution to this dilemma is to selectively use retransmission or in-order delivery for TCP traffic, but not for other traffic that does not need these features. However, this will require the link layer to be aware of the transport layer requirements.

  Nevertheless, practical implementations do often provide in-order delivery at the link layer. This solution is particularly attractive for wireless links with relatively small retransmission delays.

Many other solutions have been developed for the case when the link layer delivery may possibly be out-of-order, or no link layer retransmission capability is available. We now discuss some such solutions below.

## 9.6 Split Connection Approach

The split connection approach does not rely on the availability of a link layer retransmission mechanism. Instead, it achieves a similar result by splitting the TCP connection into

two connections. Recall that, in our example, the TCP connection is between the transport layers at hosts S and D, where the route includes the wireless link BD. The split connection approach replaces this single TCP connection by a TCP connection between S and B (no wireless links on the route taken by this connection), and a TCP connection between B and D (going over the wireless link). At the intermediate host B, which is one endpoint of the wireless link, packets from the receive buffer of the first TCP connection are moved to the send buffer of the other TCP connection, as the send buffer space permits.

Figure 9.14 illustrates the split connection approach. As shown, now there are two TCP connections replacing our original TCP connection. Also, the intermediate host B must maintain per-TCP connection state (namely, send and receive buffers) for each TCP connection that traverses the wireless link BD. This is in contrast to the link layer retransmission scheme discussed earlier, which does not maintain separate state for each transport layer connection. When a packet is lost due to transmission errors on link BD, the TCP sender of the second TCP connection (over link BD) will detect the packet loss and retransmit the lost packet, in addition to reducing its congestion window. However, sender S is shielded from any losses on link BD, since the TCP connection originating at S now terminates at host B.

Since the TCP sender at host S does not need to respond to the losses on the wireless link, the TCP connection between S and B does not incur performance degradation due to any wireless loss. However, the TCP connection on link BD may possibly have a reduced throughput due to the wireless losses, since the TCP sender at host B will react to these losses by reducing its congestion window. However, the resulting detrimental impact is likely to be smaller than in the case of the original TCP connection from S to D because the delay over link BD is smaller than that over route SD. This implies that, even if the TCP connection on link BD reduces its congestion window in response to a wireless loss, it will sooner reach the optimal congestion window. Recall that the TCP congestion window growth depends on the RTT (smaller RTT resulting in faster growth). Therefore, the smaller RTT over link BD (as compared to route SD) helps in reaching the suitable congestion window sooner. This reduces the period during which the TCP connection over BD loses throughput.

The split connection approach can thus significantly improve performance. It achieves this benefit by retaining the *local recovery* advantage of link layer retransmissions. The split connection approach, however, has some potential disadvantages:

- Violation of end-to-end semantics: Consider the example shown in Figure 9.15(a). As shown, host B has already received packets 39 and 40 on the SB TCP connection. As a result, B sends a TCP ack 40 to host S. However, as shown, host B is yet to transmit packet 40 to host D on the BD TCP connection. Thus, host S might receive ack 40 even before that packet has reached the intended destination D. This violates TCP's end-to-end semantics, and any application relying on this semantics may break if we replace standard TCP by split TCP.

- Hard state at intermediate host: In Figure 9.15(a), packets 39 and 40 are shown buffered at host B. When host S receives ack 40, it no longer needs to store packets 39 and 40 in its own buffer. Now, we have a situation where packets 39 and 40 are not yet delivered to the destination D, but the source S has already discarded the packets. If host B were to fail and lose the contents of its buffers, packet 39 and 40 will not be delivered to the destination. This violates the correctness of TCP, since packet delivery is no longer reliable. The standard TCP protocol ensures reliable delivery so long as the source and destinations (S and D) do not fail, and there is a route between the two hosts. With split TCP, we also require an intermediate host to maintain TCP state reliably.

- Increased overhead with mobility: A consequence of the fact that the state at host B is hard is that, with mobility, this state may have to be transferred to another host. Suppose that host B is a base station, and that host D is a mobile host as shown in Figure 9.15(b). If host D were to move out-of-range of base station B, and move closer to base station C, then the state at host B (particularly, packets 39 and 40) will have to be transferred to host C, the new base station. Otherwise, as in the previous scenario, these packets may not be delivered to host D, violating TCP's reliability guarantee.

- Asymmetric links: When the forward and reverse paths between hosts S and D do not both traverse the wireless link, split TCP is not as effective. Consider the example in Figure 9.16. Here, the link between B and D is an asymmetric satellite link – D can receive packets from B, but does not have the capability to transmit back to B. Thus, D can receive TCP data packets on the BD link. However, the TCP acks have to be sent on another link, namely, link DU, which may be a dialup connection (over telephone lines). In this case, we cannot obtain a significant benefit by splitting the TCP connection at host B. With split connection, the TCP acks for the second connection will have to follow a relatively long route, not yielding the local recovery advantage expected of split TCP.

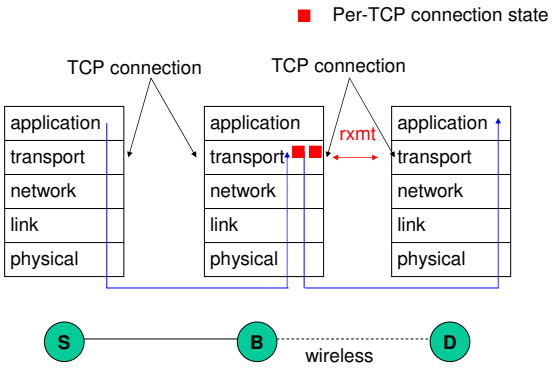



**Figure 9.14** Protocol stack illustrating the split connection approach

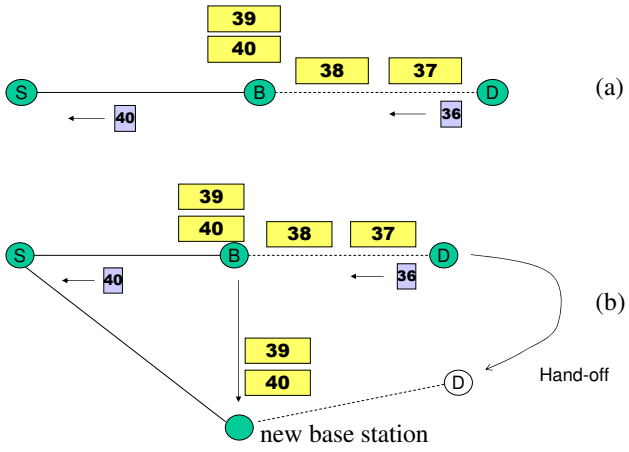



**Figure 9.15** Disadvantages of split connection approach

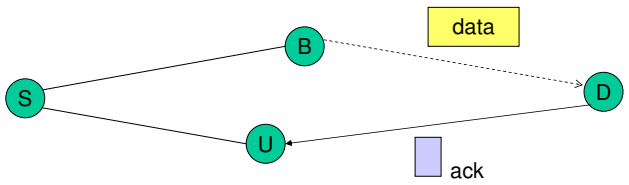



**Figure 9.16** Impact of link asymmetry on the split connection approach

## 9.7 TCP-Aware Link Layer Retransmission

While split TCP does improve performance, it may result in violation of the end-to-end semantics and reliability guarantees, as seen above. An alternative mechanism is to implement link layer retransmissions, but in a TCP-aware manner, such that the link layer takes actions that are appropriate for TCP. Figure 9.17 illustrates the protocol stack for this approach. Unlike split TCP, here the TCP connection is not split into two TCP connections. This helps preserve the end-to-end semantics. Also, the state at intermediate host B is soft state – loss of this state may result in degradation in performance, but not a loss of correctness. An important difference from the link layer retransmission scheme discussed previously is that in the TCP-aware scheme, the link layer maintains per-TCP connection state. Clearly, to be able to implement this approach, the link layer needs to be able to view the TCP headers. This will not be feasible if the TCP headers are encrypted (when using IPsec, for instance) and the intermediate host is not trusted. Similar to the link layer schemes, the retransmissions in response to wireless losses typically occur only on the wireless link, preserving the local recovery advantage.

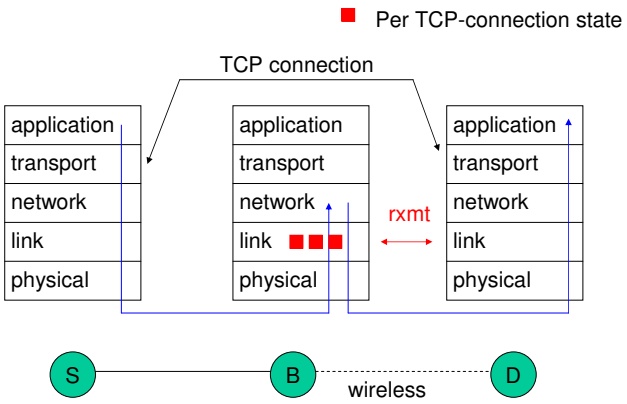



**Figure 9.17** TCP-aware link layer

The TCP-aware link layer retransmission approach works as follows. Similar to the previous link layer scheme, the link layer at host B buffers TCP packets so that they can be retransmitted if needed on the wireless link. However, this scheme does not use link layer acknowledgements, instead relying on TCP acknowledgements to determine whether a certain TCP packet has been delivered successfully on the wireless link or not. The protocol is implemented at the intermediate host, such as host B in our example. In this case, host B buffers packets that it has received from S, but that are not known to have reached host D yet. When the link layer at host B determines that a packet was lost before reaching host D, it transmits that packet provided that the packet is available in the buffer at host B. On the other hand, if the packet did not reach B either (because it was lost before reaching B) then, of course, host B cannot transmit that packet.

Let us consider an example, starting with the first snapshot in Figure 9.18. In this example, all sequence numbers are TCP sequence numbers. As shown in the snapshot, host D has just sent ack 36 on reception of data packet 36 from S. Host B has just received TCP ack 34 and forwarded it to host S. On reception of TCP ack 34, host B learns that all TCP packets through 34 have been received by D. Therefore, it need not keep them in its local buffer anymore. On the other hand, it has received TCP packets 35 through 38 from S, but has not yet seen a TCP ack from D for these packets. Consequently, host B buffers these packets. The buffered packets at B are shown above host B in the figure. Since host B decides which packets to remove from the buffer based on the TCP acks it receives from D, host B needs to be able to read the TCP sequence number in the TCP ack.

Since packet 37 is corrupted on the wireless link, no TCP ack is generated when packet 37 is delivered at D, as seen in snapshot (b) in Figure 9.18. Next, as seen in snapshot (c), when host B receives packet 39, the packet is added to the local buffer. Also, since B receives Ack 36 from D, host B removes packets 35 and 36 from its local buffer. As seen above, data packet 37 is lost on the wireless link due to transmission errors. Subsequently, when host D receives packet 38, host D will send dupack 36, since 36 is the last in-order

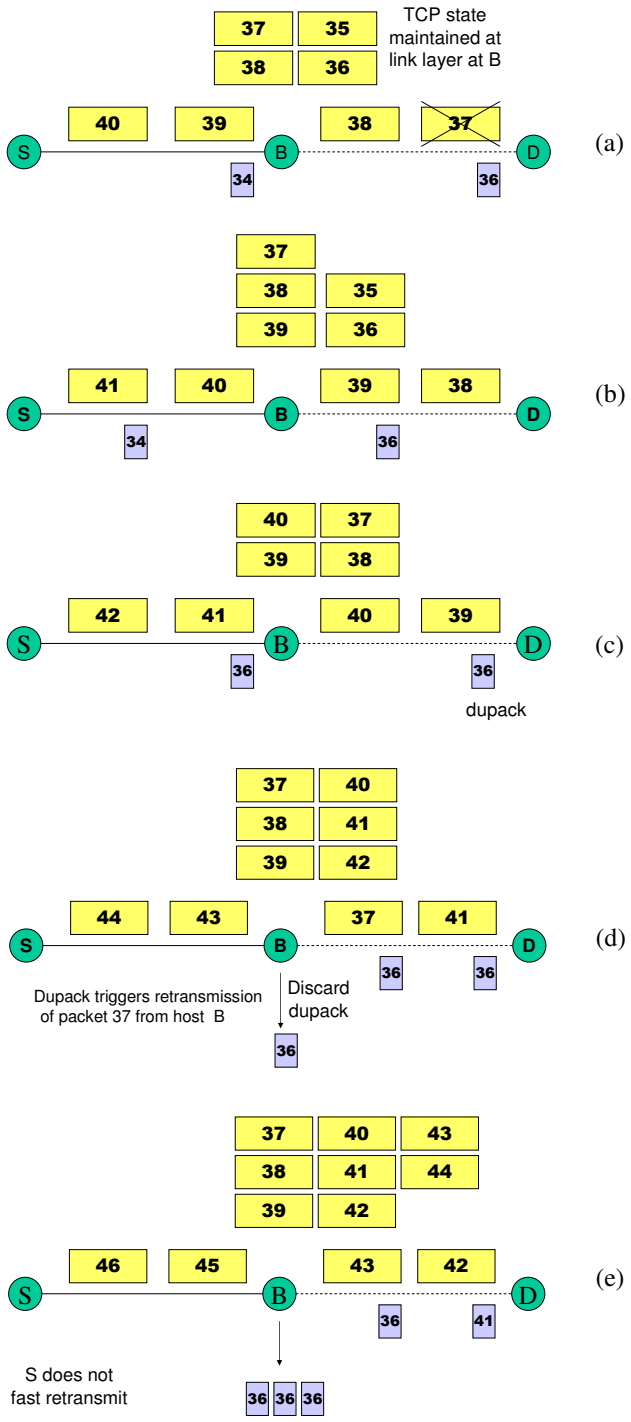



**Figure 9.18** Illustration of TCP-aware link layer retransmission scheme

packet received by host D. Similarly, receipt of TCP packets 39, 40 and 41 will also generate TCP dupack 36 from D. In the meantime, host B is receiving other packets from host S, which are then buffered at host B.

Snapshot (d) in Figure 9.18 illustrates the key action taken by the TCP-aware link layer retransmission scheme. Observe that, when host B receives the first of the dupack 36 sent by host D, host B realizes that host D has not yet received packet 37, but did receive a subsequent packet.

Now, packet 37 is in host B's local buffer. Thus, it is clear that packet 37 was transmitted to D by host B, but not received reliably by host D. In this case, host B detects the loss on the wireless link, which is presumed to be due to errors, not congestion. Host B then transmits data packet 37 from its local buffer. In this manner, this scheme provides local recovery of losses on the wireless link. Ordinarily, host B being an intermediate host on the route from D to S, it will simply forward the dupack on its route to host S. However, forwarding the dupacks to S will eventually trigger a fast retransmit from sender S, as well as a reduction of congestion window at S. To avoid this, particularly since host B is able to retransmit the lost packet, host B discards the dupacks in addition to retransmitting packet 36 to host D. Snapshot (d) shows that host B has retransmitted packet 37, and also discarded one dupack 36. Subsequently, host B will receive other dupack 36 sent by host D, and discards these as well. It continues to buffer new packets received from host S. Until B receives a new ack from D, no packets can be removed from the buffer.

Eventually, packet 37 retransmitted by host B reaches host D. Previous to this, host D has received all packets through 41 except packet 37. Therefore, on receipt of packet 37, host D sends ack 41, as seen in snapshot (e) in Figure 9.18. When host B receives ack 41, it can remove all data packets through 41 from its local buffer. All new acks, including this ack 41, are forwarded to source S.

As seen above, by retransmitting the lost packet from its local buffer, and dropping the dupacks, host B is able to achieve reliable delivery of packet 37, and also hide the packet loss from TCP source S. This ensures that the TCP source S does not respond inappropriately to the loss on the wireless link.

A few issues need further attention. First, what happens if the wireless link loses so many packets that no dupack is received by host B? To allow the link layer to detect such packet loss, host B will need to utilize a timeout-based scheme. This may be implemented similar to the TCP timeout mechanism, except that in this case the round-trip delay on the wireless link will have to be used to determine the suitable timeout interval at host B.

The second issue is that, as described above host B discards dupack 36. What happens if the retransmitted packet 36 is lost too? This will also lead to additional dupacks. If host B were to simply discard these dupacks, then it will not be able to quickly detect the loss of retransmitted packet 36 (it may detect this eventually using a timeout). To allow host B to detect the loss of a retransmitted packet, host B should not discard more dupacks than the

number of TCP packets sent between the two transmissions of the lost packet (packet 37 above). With these modifications, the above scheme can often recover from wireless losses without the TCP sender S detecting the loss.

## 9.8 Impact of Long Outages

Consider two scenarios, which both result in similar detrimental impact on TCP:

- Scenario 1: Consider a wireless host using a cellular link. When the host passes through a tunnel, or behind an obstacle, there is a possibility of link outage for non-negligible durations. A similar situation also arises in case the route includes a satellite links, and the satellite link encounters long durations of poor quality.
- Scenario 2: Consider a mobile ad hoc network using a reactive routing protocol. When an existing route breaks, it may take the network layer a substantial amount of time to rediscover a new valid route.

In both cases above, the TCP sender may timeout before packet delivery resumes to the receiver, as illustrated in Figure 9.19(a). As seen in this example, when a route breaks (either because the cellular link becomes unavailable, or due to host mobility in ad hoc networks), packets en route to the destination cannot be delivered, and are discarded by the lower layers (here we assume that the lower layers do not indefinitely try to retransmit packets to the host). Subsequently, the route is repaired. However, the TCP sender will not make an attempt to transmit a packet until a timeout occurs. Thus, the TCP connection will not obtain useful throughput until the TCP sender times out. Clearly, while the route is broken, we cannot expect TCP to be able to send packets, however, the loss of throughput between route repair and TCP timeout is unnecessary, and is caused by TCP's congestion control mechanisms.

This detrimental impact can be more pronounced with larger route repairs delays. Let us consider the case where the route is not yet repaired when TCP sender times out (Figure 9.19(b)). Thus, the retransmission following the timeout is lost as well. Again, TCP sender will not attempt to send another packet until the next timeout occurs. However, due to the exponential backoff mechanism, the next timeout interval is twice as long as the initial timeout interval. In this case, there is a potential for loss of throughput for a longer duration of time, as seen in Figure 9.19(b).

The performance degradation above occurs because the TCP sender is unaware that the packet loss occurred due to route failure, as opposed to congestion. Thus, TCP relies on the timeout mechanism to determine when to make the next transmission attempt. To alleviate this performance impact, one solution is to send TCP sender an *explicit notification*

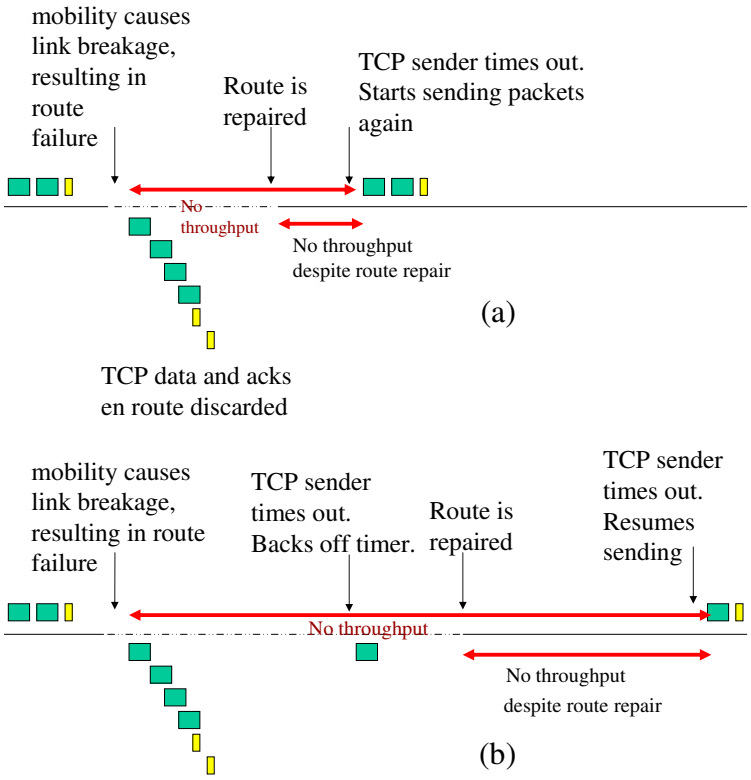



**Figure 9.19** Impact of long outage on TCP

that the route has been repaired. In case of cellular networks, this notification could potentially be generated by the base station. In case ad hoc networks, the network layer (possibly at the sender) will have to send a notification to the transport layer that the route has been discovered. This may be implemented in the form of an ICMP message. On receiving the notification, the TCP sender may resume transmission immediately instead of waiting for the timeout to occur. In other words, the sender can pretend that the timeout occurred at the time of the reception of the explicit notification. This explicit notification approach can indeed improve TCP performance, however, it requires additional signaling, in the form of the explicit notification, between TCP and the lower layers.

A secondary issue to consider in the context of explicit notification is the choice of congestion window after the TCP sender receives the explicit notification. In the case when long interval of time elapses between route failure and repair, clearly, the congestion window used prior to the failure may not be appropriate for the state of the network after route repair (even if the new route is same as the old route). Secondly, when the new route is different from the old route, it may have a different level of congestion than the old route, even if the route repair delay happens to be short. A conservative response after route repair is to initiate slow-start, analogous to the standard TCP response after a timeout.

In case of ad hoc networks, the behavior illustrated in Figure 9.19 may interact detrimentally with route caching for reactive protocols. With route caching, hosts utilize routes from their cache, when the route in use breaks. However, there is a risk that the cached route is broken too. In this case, after each timeout, a new route from the cache may be used, only to discover after another (larger) timeout interval that it is broken as well. Due to exponential backoff, the timeout value, and therefore, the time between consecutive retransmission attempts increases. Together, exponential backoff and stale caches can degrade performance significantly. The use of explicit notifications can reduce the time between successive attempts, however, stale caches will still result in the retransmissions being sent on broken routes. Thus, it is important to improve route cache accuracy to avoid detrimental impact on TCP performance.

## 9.9 TCP over Ad Hoc Networks

In the previous discussion, we have already considered the impact of route repair latency, as well as interaction with route caching. In ad hoc networks, TCP performance is also affected by the number of hops on the wireless route. To understand the impact of number of hops, let us consider routes with various number of hops. We will consider the chain network illustrated in Figure 9.20, where host 1 is the TCP source, and host n is the TCP destination, in case of a chain with n hosts. Assume that IEEE 802.11 DCF (or similar) MAC protocol is being used with half-duplex wireless interfaces, with single interface per

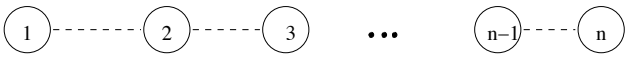



**Figure 9.20** A chain topology

host (all hosts being on the same channel). Assume that simultaneous transmissions on two links can be successful only if they are separated by at least two hops; links closer than that pose too much interference to each other leading to packet loss.

Let us start with a chain network with just two hosts (single hop route). In this case, the TCP data packets from 1 to 2, and TCP acks from 2 to 1 have to share the wireless channel. Let us now consider the TCP throughput in this case as the baseline for comparison with longer chains. Now, consider a 2-hop chain (3 hosts). In this case, the TCP packets have to travel from 1 to 3 via host 2. Since host 2 is half-duplex, we can either have transmission on link 1-2 or link 2-3. This results in degradation in throughput by approximately a factor of 2 when compared to the single hop chain. Note that this effect will occur for UDP and TCP both, and is not caused by TCP behavior as such. In case of the 3 hop chain (with 4 hosts), since links 1-2 and 3-4 are too close to each other, we cannot have simultaneous successful transmissions on those two links. Effectively, we can only schedule reliable transmissions on any one link at a given time. This degrades throughput by roughly a factor of 3 compared to the baseline single-hop scenario. The interference between links 1-2 and 3-4 is sometimes referred to as *self-interference*, since packets from the same TCP flow transmitted on those two links interfere with each other. In case of the 4-hop network (with 5 hosts), links 1-2 and 4-5 can potentially transmit packets simultaneously. Thus, we do not anticipate more degradation due to self-interference than that in the 3-hop network. However, the greater number of hops increases the round trip delay, and that degrades throughput somewhat, since TCP's congestion window growth is slower with larger RTT.

If the hosts were to be closer to each other, and if interference between links separated by 3 hops were to result in collisions, then the degradation in throughput due to self-interference will continue up to a 5 hop networks. Clearly, whether the interference is too high or not depends on path losses as well as transmission rate and power, but the example here illustrates that self-interference phenomenon degrades throughput when number of hops on the route is greater than one.