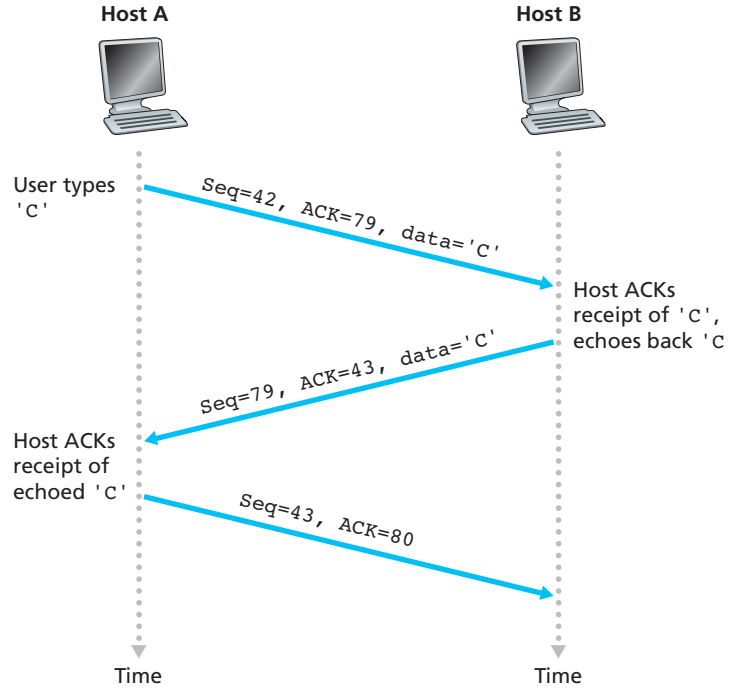

Figure 3.31 ♦ Sequence and acknowledgment numbers for a simple Telnet application over TCP

The third segment is sent from the client to the server. Its sole purpose is to acknowledge the data it has received from the server. (Recall that the second segment contained data—the letter 'C'—from the server to the client.) This segment has an empty data field (that is, the acknowledgment is not being piggybacked with any client-to-server data). The segment has 80 in the acknowledgment number field because the client has received the stream of bytes up through byte sequence number 79 and it is now waiting for bytes 80 onward. You might think it odd that this segment also has a sequence number since the segment contains no data. But because TCP has a sequence number field, the segment needs to have some sequence number.

3.5.3 Round-Trip Time Estimation and Timeout

TCP, like our `rdt` protocol in Section 3.4, uses a timeout/retransmit mechanism to recover from lost segments. Although this is conceptually simple, many subtle issues arise when we implement a timeout/retransmit mechanism in an actual protocol such as TCP. Perhaps the most obvious question is the length of the timeout

intervals. Clearly, the timeout should be larger than the connection's round-trip time (RTT), that is, the time from when a segment is sent until it is acknowledged. Otherwise, unnecessary retransmissions would be sent. But how much larger? How should the RTT be estimated in the first place? Should a timer be associated with each and every unacknowledged segment? So many questions! Our discussion in this section is based on the TCP work in [Jacobson 1988] and the current IETF recommendations for managing TCP timers [RFC 6298].

Estimating the Round-Trip Time

Let's begin our study of TCP timer management by considering how TCP estimates the round-trip time between sender and receiver. This is accomplished as follows. The sample RTT, denoted `SampleRTT`, for a segment is the amount of time between when the segment is sent (that is, passed to IP) and when an acknowledgment for the segment is received. Instead of measuring a `SampleRTT` for every transmitted segment, most TCP implementations take only one `SampleRTT` measurement at a time. That is, at any point in time, the `SampleRTT` is being estimated for only one of the transmitted but currently unacknowledged segments, leading to a new value of `SampleRTT` approximately once every RTT. Also, TCP never computes a `SampleRTT` for a segment that has been retransmitted; it only measures `SampleRTT` for segments that have been transmitted once [Karn 1987]. (A problem at the end of the chapter asks you to consider why.)

Obviously, the `SampleRTT` values will fluctuate from segment to segment due to congestion in the routers and to the varying load on the end systems. Because of this fluctuation, any given `SampleRTT` value may be atypical. In order to estimate a typical RTT, it is therefore natural to take some sort of average of the `SampleRTT` values. TCP maintains an average, called `EstimatedRTT`, of the `SampleRTT` values. Upon obtaining a new `SampleRTT`, TCP updates `EstimatedRTT` according to the following formula:

$$\texttt{EstimatedRTT} = (1 - \alpha) \cdot \texttt{EstimatedRTT} + \alpha \cdot \texttt{SampleRTT}$$

The formula above is written in the form of a programming-language statement—the new value of `EstimatedRTT` is a weighted combination of the previous value of `EstimatedRTT` and the new value for `SampleRTT`. The recommended value of α is $\alpha = 0.125$ (that is, 1/8) [RFC 6298], in which case the formula above becomes:

$$\texttt{EstimatedRTT} = 0.875 \cdot \texttt{EstimatedRTT} + 0.125 \cdot \texttt{SampleRTT}$$

Note that `EstimatedRTT` is a weighted average of the `SampleRTT` values. As discussed in a homework problem at the end of this chapter, this weighted average puts more weight on recent samples than on old samples. This is natural, as the

PRINCIPLES IN PRACTICE

TCP provides reliable data transfer by using positive acknowledgments and timers in much the same way that we studied in Section 3.4. TCP acknowledges data that has been received correctly, and it then retransmits segments when segments or their corresponding acknowledgments are thought to be lost or corrupted. Certain versions of TCP also have an implicit NAK mechanism—with TCP's fast retransmit mechanism, the receipt of three duplicate ACKs for a given segment serves as an implicit NAK for the following segment, triggering retransmission of that segment before timeout. TCP uses sequences of numbers to allow the receiver to identify lost or duplicate segments. Just as in the case of our reliable data transfer protocol, `rdt3.0`, TCP cannot itself tell for certain if a segment, or its ACK, is lost, corrupted, or overly delayed. At the sender, TCP's response will be the same: retransmit the segment in question.

TCP also uses pipelining, allowing the sender to have multiple transmitted but yet-to-be-acknowledged segments outstanding at any given time. We saw earlier that pipelining can greatly improve a session's throughput when the ratio of the segment size to round-trip delay is small. The specific number of outstanding, unacknowledged segments that a sender can have is determined by TCP's flow-control and congestion-control mechanisms. TCP flow control is discussed at the end of this section; TCP congestion control is discussed in Section 3.7. For the time being, we must simply be aware that the TCP sender uses pipelining.

more recent samples better reflect the current congestion in the network. In statistics, such an average is called an **exponential weighted moving average (EWMA)**. The word "exponential" appears in EWMA because the weight of a given `SampleRTT` decays exponentially fast as the updates proceed. In the homework problems you will be asked to derive the exponential term in `EstimatedRTT`.

Figure 3.32 shows the `SampleRTT` values and `EstimatedRTT` for a value of $\alpha = 1/8$ for a TCP connection between `gaia.cs.umass.edu` (in Amherst, Massachusetts) to `fantasia.eurecom.fr` (in the south of France). Clearly, the variations in the `SampleRTT` are smoothed out in the computation of the `EstimatedRTT`.

In addition to having an estimate of the RTT, it is also valuable to have a measure of the variability of the RTT. [RFC 6298] defines the RTT variation, `DevRTT`, as an estimate of how much `SampleRTT` typically deviates from `EstimatedRTT`:

$$\texttt{DevRTT} = (1 - \beta) \cdot \texttt{DevRTT} + \beta \cdot |\texttt{SampleRTT} - \texttt{EstimatedRTT}|$$

Note that `DevRTT` is an EWMA of the difference between `SampleRTT` and `EstimatedRTT`. If the `SampleRTT` values have little fluctuation, then `DevRTT` will be small; on the other hand, if there is a lot of fluctuation, `DevRTT` will be large. The recommended value of β is 0.25.

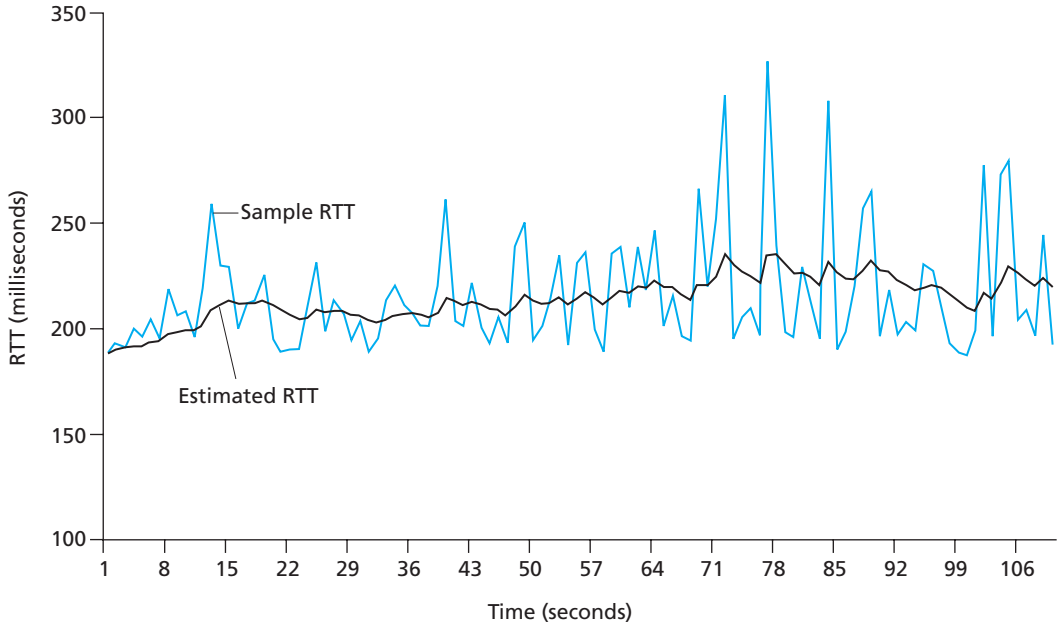

Figure 3.32 ♦ RTT samples and RTT estimates

Setting and Managing the Retransmission Timeout Interval

Given values of `EstimatedRTT` and `DevRTT`, what value should be used for TCP's timeout interval? Clearly, the interval should be greater than or equal to `EstimatedRTT`, or unnecessary retransmissions would be sent. But the timeout interval should not be too much larger than `EstimatedRTT`; otherwise, when a segment is lost, TCP would not quickly retransmit the segment, leading to large data transfer delays. It is therefore desirable to set the timeout equal to the `EstimatedRTT` plus some margin. The margin should be large when there is a lot of fluctuation in the `SampleRTT` values; it should be small when there is little fluctuation. The value of `DevRTT` should thus come into play here. All of these considerations are taken into account in TCP's method for determining the retransmission timeout interval:

```
TimeoutInterval = EstimatedRTT + 4 · DevRTT
```

An initial `TimeoutInterval` value of 1 second is recommended [RFC 6298]. Also, when a timeout occurs, the value of `TimeoutInterval` is doubled to avoid a premature timeout occurring for a subsequent segment that will soon be acknowledged. However, as soon as a segment is received and `EstimatedRTT` is updated, the `TimeoutInterval` is again computed using the formula above.