Round Trip Time Estimation and Timeout

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.

TCP maintains an average, called `EstimatedRTT`, of the `SampleRTT` values. Upon obtaining a new `SampleRTT`, TCP updates `EstimatedRTT` according to the following formula

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

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

`DevRTT`, as an estimate of how much **`SampleRTT`** typically deviates from **`EstimatedRTT`**:

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

Estimating Timeout

The timeout 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,

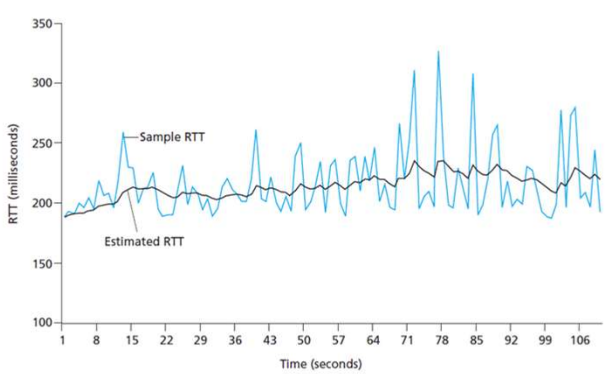

Figure 3.32 ♦ RTT samples and RTT estimates

$$\texttt{TimeoutInterval} = \texttt{EstimatedRTT} + 4 \cdot \texttt{DevRTT}$$

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.

Reliable Data Transfer in TCP

```
NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop (forever) {
    switch(event)

        event: data received from application above
            create TCP segment with sequence number NextSeqNum
            if (timer currently not running)
                start timer
            pass segment to IP
            NextSeqNum=NextSeqNum+length(data)
            break;

        event: timer timeout
            retransmit not-yet-acknowledged segment with
                smallest sequence number
            start timer
            break;

        event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase=y
                if (there are currently any not-yet-acknowledged segments)
                    start timer
                }
            break;

} /* end of loop forever */
```

Figure 3.33 ♦ Simplified TCP sender

Doubling the timeout interval: Each time TCP retransmits, it sets the next timeout interval to twice the previous value, rather than deriving it from the last `EstimatedRTT` and `DevRTT`

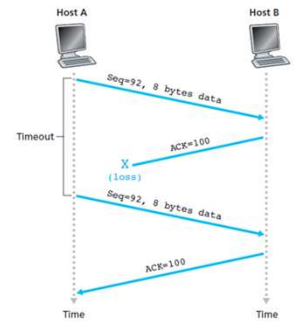

Figure 3.34 ♦ Retransmission due to a lost acknowledgment

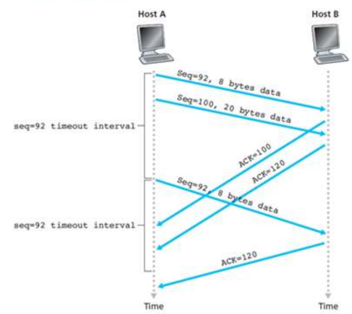

Figure 3.35 ♦ Segment 100 not retransmitted

Reliable Data Transfer: Fast Retransmit

Event	TCP Receiver Action
Arrival of in-order segment with expected sequence number. All data up to expected sequence number already acknowledged.	Delayed ACK. Wait up to 500 msec for arrival of another in-order segment. If next in-order segment does not arrive in this interval, send an ACK.
Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK transmission.	Immediately send single cumulative ACK, ACKing both in-order segments.
Arrival of out-of-order segment with higher-than-expected sequence number. Gap detected.	Immediately send duplicate ACK, indicating sequence number of next expected byte (which is the lower end of the gap).
Arrival of segment that partially or completely fills in gap in received data.	Immediately send ACK, provided that segment starts at the lower end of gap.

Table 3.2 ♦ TCP ACK Generation Recommendation [RFC 5681]

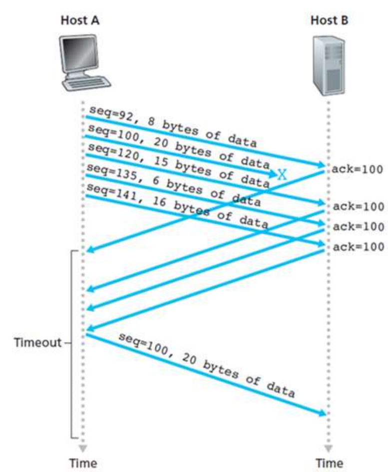

Figure 3.37 ♦ Fast retransmit: retransmitting the missing segment before the segment's timer expires

Reliable Data Transfer in TCP: Fast Retransmit

If the TCP sender receives three duplicate ACKs for the same data, it takes this as an indication that the segment following the segment that has been ACKed three times has been lost.

In the case that three duplicate ACKs are received, the TCP sender performs a **fast retransmit** [RFC 5681], retransmitting the missing segment *before* that segment's timer expires.

```
event: ACK received, with ACK field value of y
            if (y > SendBase) {
                    SendBase=y
                    if (there are currently any not yet
                                acknowledged segments)
                        start timer
                    }
            else { /* a duplicate ACK for already ACKed
                    segment */
                increment number of duplicate ACKs
                    received for y
                if (number of duplicate ACKS received
                    for y==3)
                    /* TCP fast retransmit */
                    resend segment with sequence number y
                }
            break;
```


TCP Flow Control Service:

TCP provides a **flow-control service** to its applications to eliminate the possibility of the sender overflowing the receiver's buffer. Flow control is thus a speed-matching service—matching the rate at which the sender is sending against the rate at which the receiving application is reading.

TCP provides flow control by having the *sender* maintain a variable called the **receive window**. Informally, the receive window is used to give the sender an idea of how much free buffer space is available at the receiver.

Because TCP is full-duplex, the sender at each side of the connection maintains a distinct receive window.

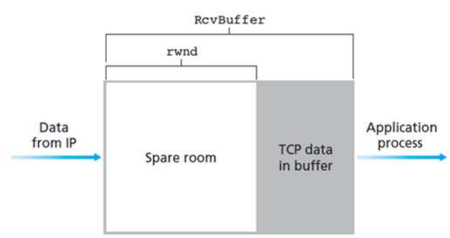

Figure 3.38 ♦ The receive window (`rwnd`) and the receive buffer (`RcvBuffer`)

Because TCP is not permitted to overflow the allocated buffer, we must have

`LastByteRcvd – LastByteRead ≤ RcvBuffer`

`rwnd = RcvBuffer – [LastByteRcvd – LastByteRead]`

Host B tells Host A how much spare room it has in the connection buffer by placing its current value of `rwnd` in the receive window field of every segment it sends to A. Initially, Host B sets `rwnd = RcvBuffer`.

TCP: Flow Control:

Host A in turn keeps track of two variables, `LastByteSent` and `Last-Byte Acked`, which have obvious meanings. Note that the difference between these two variables, `LastByteSent – LastByteAcked`, is the amount of unacknowledged data that A has sent into the connection.

$$\texttt{LastByteSent} - \texttt{LastByteAcked} \leq \texttt{rwnd}$$

By keeping the amount of unacknowledged data less than the value of `rwnd`, Host A is assured that it is not overflowing the receive buffer at Host B.

TCP Connection Termination

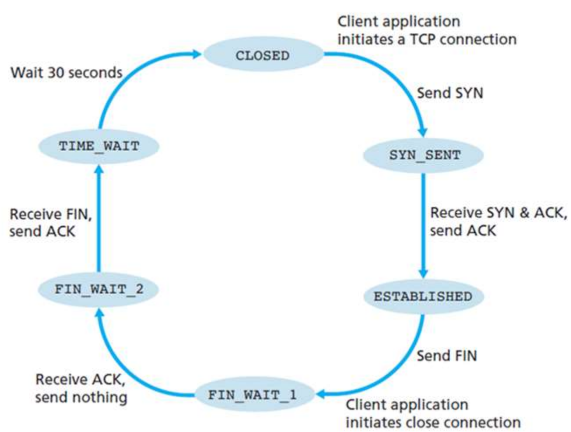

Figure 3.41 ♦ A typical sequence of TCP states visited by a client TCP

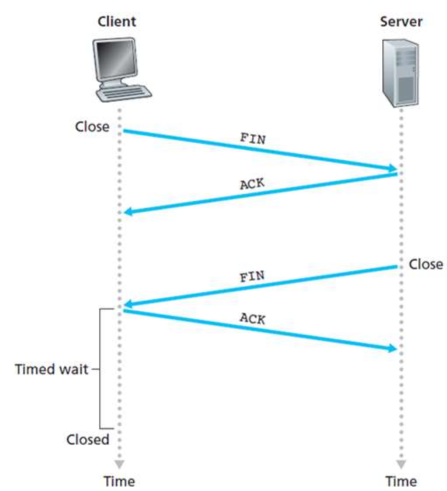

Figure 3.40 ♦ Closing a TCP connection

TCP Connection Termination

when a host sends a reset segment, it is telling the source "I don't have a socket for that segment. Please do not resend the segment."

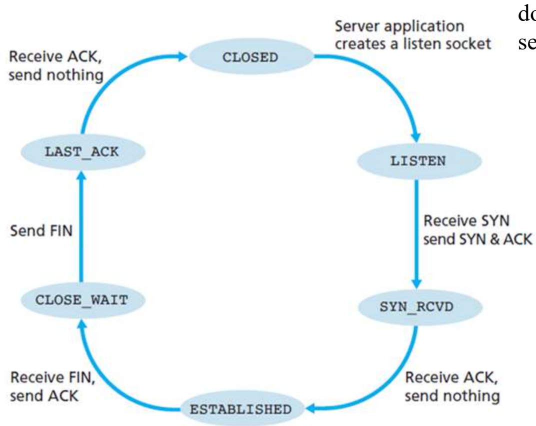

Figure 3.42 ♦ A typical sequence of TCP states visited by a server-side TCP

TCP Congestion Control

TCP sender perceives that there is little congestion on the path between itself and the destination, then the TCP sender increases its send rate; if the sender perceives that there is congestion along the path, then the sender reduces its send rate.

But this approach raises three questions.
1. First, how does a TCP sender limit the rate at which it sends traffic into its connection?
2. Second, how does a TCP sender perceive that there is congestion on the path between itself and the destination?
3. And third, what algorithm should the sender use to change its send rate as a function of perceived end-to-end congestion?

Specifically, the amount of unacknowledged data at a sender may not exceed the minimum of `cwnd` and `rwnd`, that is:

```
LastByteSent – LastByteAcked min{cwnd, rwnd}
```

TCP Congestion Control

TCP uses acknowledgments to trigger (or clock) its increase in congestion window size,TCP is said to be **self-clocking**.

A lost segment implies congestion, and hence, the TCP sender's rate should be decreased when a segment is lost.

An acknowledged segment indicates that the network is delivering the sender's segments to the receiver, and hence, the sender's rate can be increased when an ACK arrives for a previously unacknowledged segment.

Bandwidth probing. Given ACKs indicating a congestion-free source-to-destination path and loss events indicating a congested path, TCP's strategy for adjusting its transmission rate is to increase its rate in response to arriving ACKs until a loss event occurs, at which point, the transmission rate is decreased.

TCP Congestion control algorithm has three components
Slow Start, Congestion Avoidance and Fast Recovery

TCP Congestion Control

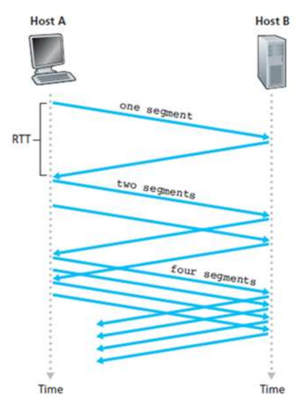

Figure 3.51 ♦ TCP slow start

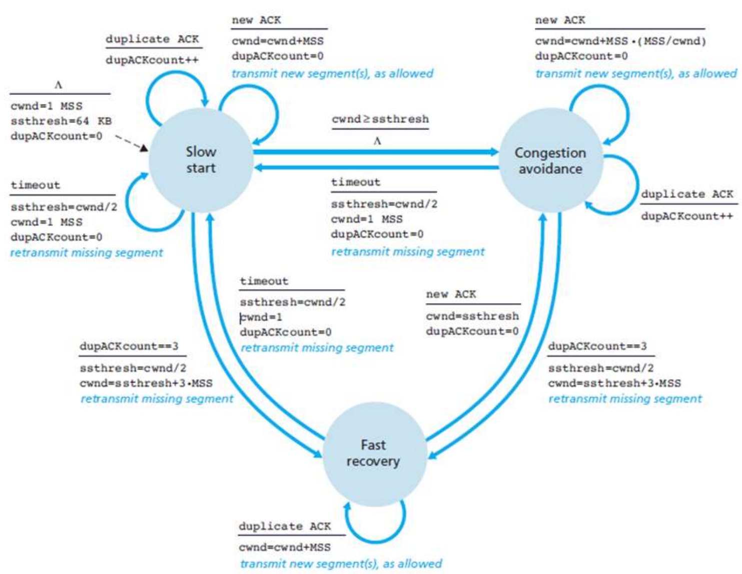

Figure 3.52 ♦ FSM description of TCP congestion control

https://tools.ietf.org/html/rfc2001

TCP Congestion Control: Reno and Tahoe

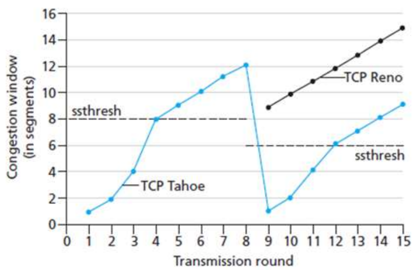

Figure 3.53 ♦ Evolution of TCP's congestion window (Tahoe and Reno)