TRANSMISSION CONTROL PROTOCOL
By Mr. Vishal Jaiswal

Transmission Control Protocol (TCP) is a connection-oriented, reliable protocol. TCP explicitly defines connection establishment, data transfer, and connection teardown phases to provide a connection-oriented service.

TCP Congestion -

Congestion refers to a network state where - The message traffic becomes so heavy that it slows down the network response time. Congestion is an important issue that can arise in Packet Switched Network. **Congestion leads to the loss of packets in transit.**

- Congestion is an important issue that can arise in **Packet Switched Network.**
- Congestion leads to the loss of packets in transit.
- So, it is necessary to control the congestion in network.
- It is not possible to completely avoid the congestion.

TCP Congestion Control :

The important service done by TCP is providing end to end transmission. This service has a significant role in congestion control in the network. Congestion free network which is an ideal network means taking the advantage of all the bandwidth in the network.

The sender's window size is determined not only by the receiver but also by congestion in the network. The sender has two pieces of information: the receiver-advertised window size and the congestion window size. The actual size of the window is the minimum of these two.

Actual window size = minimum (rwnd, cwnd)

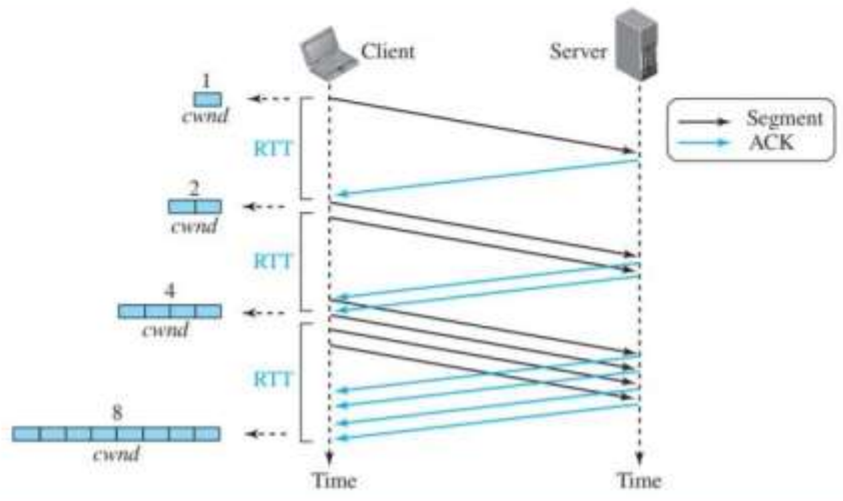
Congestion Policy :

TCP's general policy for handling congestion is based on three phases: slow start, congestion avoidance, and congestion detection. In the slow-start phase, the sender starts with a very slow rate of transmission, but increases the rate rapidly to reach a threshold. When the threshold is reached, the data rate is reduced to avoid congestion. Finally if congestion is detected, the sender goes back to the slow-start or congestion avoidance phase based on how the congestion is detected.

Slow Start: Exponential Increase :

The slow-start algorithm is based on the idea that the size of the congestion window (cwnd) starts with one maximum segment size (MSS), but it increases one MSS each time an acknowledgment arrives.

Figure 24.29 *Slow start, exponential increase*



the size of the congestion window in this algorithm is a function of the number of ACKs arrived and can be determined as follows.

If an ACK arrives, $cwnd = cwnd + 1$.

If we look at the size of the *cwnd* in terms of round-trip times (RTTs), we find that the growth rate is exponential in terms of each round trip time, which is a very aggressive approach:

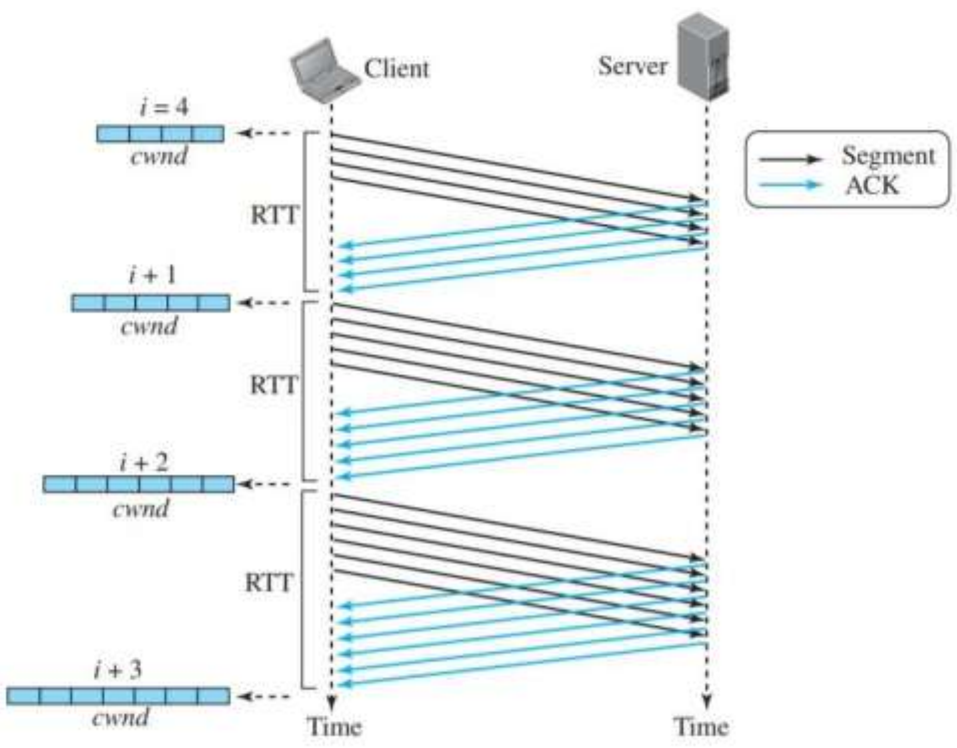
Start	$\rightarrow$	$cwnd = 1 \rightarrow 2^0$
After 1 RTT	$\rightarrow$	$cwnd = cwnd + 1 = 1 + 1 = 2 \rightarrow 2^1$
After 2 RTT	$\rightarrow$	$cwnd = cwnd + 2 = 2 + 2 = 4 \rightarrow 2^2$
After 3 RTT	$\rightarrow$	$cwnd = cwnd + 4 = 4 + 4 = 8 \rightarrow 2^3$

“ In the slow-start algorithm, the size of the congestion window increases exponentially until it reaches a threshold ”

Congestion Avoidance: Additive Increase :

To avoid congestion before it happens, we must slow down this exponential growth. TCP defines another algorithm called congestion avoidance, which increases the cwnd additively instead of exponentially. When the size of the congestion window reaches the slow-start threshold in the case where cwnd = i, the slow-start phase stops and the additive phase begins. In this algorithm, each time the whole “window” of segments is acknowledged, the size of the congestion window is increased by one. A window is the number of segments transmitted during RTT.

Figure 24.30 *Congestion avoidance, additive increase*



The sender starts with cwnd = 4. This means that the sender can send only four segments. After four ACKs arrive, the acknowledged segments are purged from the window, which means there is now one extra empty segment slot in the window. The size of the congestion window is also increased by 1. The size of window is now 5.

After sending five segments and receiving five acknowledgments for them, the size of the congestion window now becomes 6, and so on. In other words, the size of the congestion window in this algorithm is also a function of the number of ACKs that have arrived and can be determined as follows: If an ACK arrives, cwnd 5 cwnd 1 (1/cwnd). The size of the window increases only 1/cwnd portion of MSS (in bytes). In other words, all segments in the previous window should be acknowledged to increase the window 1 MSS bytes. If we look at the size of the cwnd in terms of round-trip times (RTTs), we find that the growth rate is linear in terms of each round-trip time, which is much more conservative than the slow-start approach.

Start	$\rightarrow$	$cwnd = i$
After 1 RTT	$\rightarrow$	$cwnd = i + 1$
After 2 RTT	$\rightarrow$	$cwnd = i + 2$
After 3 RTT	$\rightarrow$	$cwnd = i + 3$

In the congestion-avoidance algorithm, the size of the congestion window increases additively until congestion is detected.

Congestion Detection: Multiplicative Decrease :

If congestion occurs, the congestion window size must be decreased. The only way the sender can guess that congestion has occurred is by the need to retransmit a segment. However, retransmission can occur in one of two cases: when a timer times out or when three Duplicate ACKs are received. In both cases, the size of the threshold is dropped to one-half, a multiplicative decrease.

TCP implementations have two reactions :

1. If a time-out occurs, there is a stronger possibility of congestion; a segment has probably been dropped in the network, and there is no news about the sent segments.

In this case TCP reacts strongly:

a. It sets the value of the threshold to one-half of the current window size.

b. It sets cwnd to the size of one segment.

c. It starts the slow-start phase again.

2. If three ACKs are received, there is a weaker possibility of congestion; a segment may have been dropped, but some segments after that may have arrived safely since three ACKs are received. This is called fast transmission and fast recovery.

In this case, TCP has a weaker reaction:

a. It sets the value of the threshold to one-half of the current window size.

b. It sets cwnd to the value of the threshold.

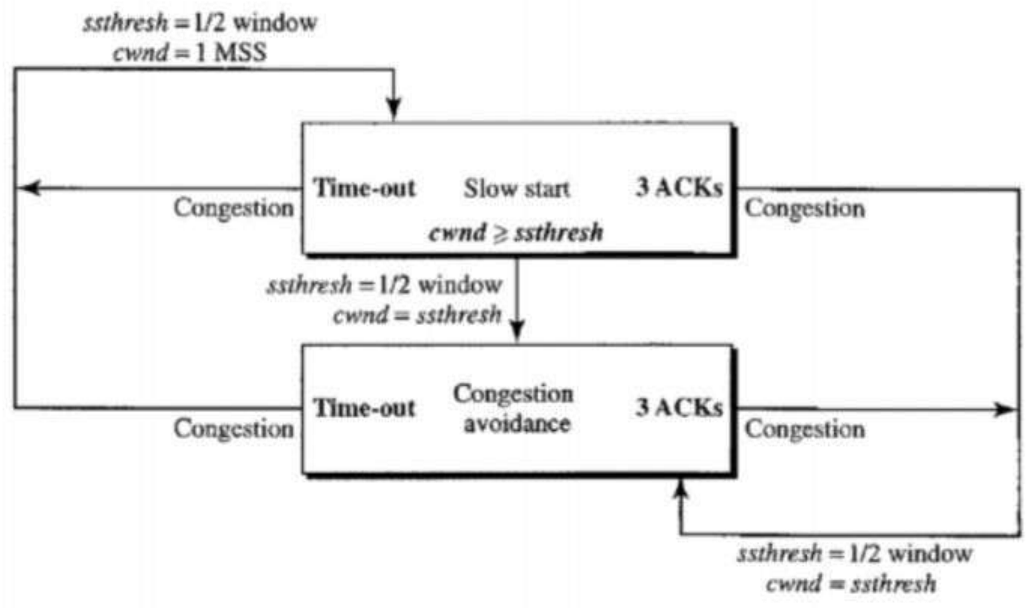
c. It starts the congestion avoidance phase.

An implementations reacts to congestion detection in one of the following ways :

If detection is by time-out, a new slow-start phase starts.

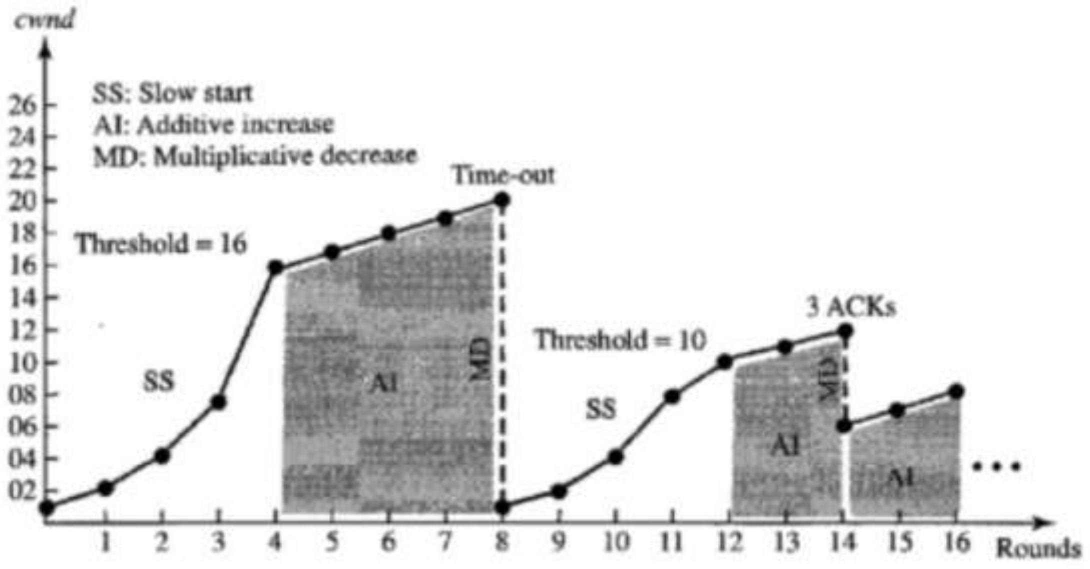
If detection is by three ACKs, a new congestion avoidance phase starts.

Figure 24.10 *TCP congestion policy summary*



Example : We assume that the maximum window size is 32 segments. The threshold is set to 16 segments (one-half of the maximum window size). In the slow-start phase the window size starts from 1 and grows exponentially until it reaches the threshold. After it reaches the threshold, the congestion avoidance (additive increase) procedure allows the window size to increase linearly until a timeout occurs or the maximum window size is reached. In Figure 24.11, the time-out occurs when the window size is 20. At this moment, the multiplicative decrease procedure takes over and reduces the threshold to one-half of the previous window size. The previous window size was 20 when the time-out happened so the new threshold is now 10. TCP moves to slow start again and starts with a window size of 1, and TCP moves to additive increase when the new threshold is reached. When the window size is 12, a three duplicate ACKs event happens. The multiplicative decrease procedure takes over again. The threshold is set to 6 and TCP goes to the additive increase phase this time. It remains in this phase until another time- out or another three duplicate ACKs happen.

Figure 24.11 *Congestion example*



Fast Recovery :

The **fast-recovery** algorithm is optional in TCP. The old version of TCP did not use it, but the new versions try to use it. It starts when three duplicate ACKs arrive, which is interpreted as light congestion in the network. Like congestion avoidance, this algorithm is also an additive increase, but it increases the size of the congestion window when a duplicate ACK arrives (after the three duplicate ACKs that trigger the use of this algorithm).

We can say :

" If a duplicate ACK arrives, cwnd 5 cwnd 1 (1 / cwnd) "

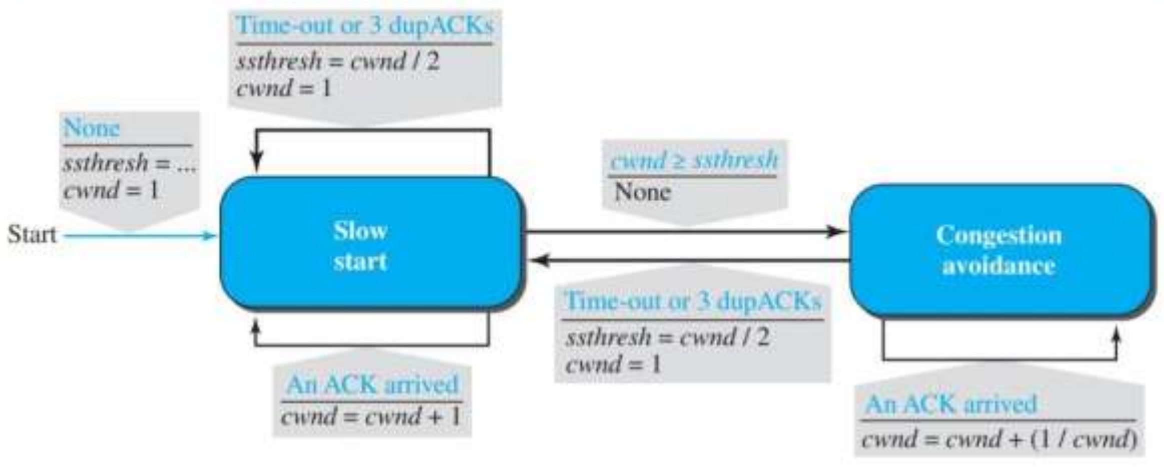
Policy Transition :

We discussed three congestion policies in TCP. Now the question is when each of these policies is used and when TCP moves from one policy to another. To answer these questions, we need to refer to three versions of TCP: Taho TCP, Reno TCP, and New Reno TCP.

Taho TCP :

The early TCP, known as Taho TCP, used only two different algorithms in their congestion policy: slow start and congestion avoidance. We use Figure 24.31 to show the FSM for this version of TCP. However, we need to mention that we have deleted some small trivial actions, such as incrementing and resetting the number of duplicate ACKs, to make the FSM less crowded and simpler.

Figure 24.31 *FSM for Taho TCP*



Taho TCP treats the two signs used for congestion detection, time-out and three duplicate ACKs, in the same way. In this version, when the connection is established, TCP starts the slow-start algorithm and sets the ssthresh variable to a pre-agreed value (normally a multiple of MSS) and the cwnd to 1 MSS. In this state, as we said before, each time an ACK arrives, the size of the congestion window is incremented by 1. We know that this policy is very aggressive and exponentially increases the size of the window, which may result in congestion. If congestion is detected (occurrence of time-out or arrival of three duplicate ACKs), TCP immediately interrupts this aggressive growth and restarts a new slow start algorithm by limiting the threshold to half of the current cwnd and resetting the congestion window to 1. In other words, not only does TCP restart from scratch, but it also learns how to adjust the threshold. If no congestion is detected while reaching the threshold, TCP learns that the ceiling of its ambition is reached; it should not continue at this speed. It moves to the congestion avoidance state and continues in that state.

In the congestion-avoidance state, the size of the congestion window is increased by 1 each time a number of ACKs equal to the current size of the window has been received. For example, if the window size is now 5 MSS, five more ACKs should be received before the size of the window becomes 6 MSS. Note that there is no ceiling for the size of the congestion window in this state; the conservative additive growth of the congestion window continues to the end of the data transfer phase unless congestion is detected.

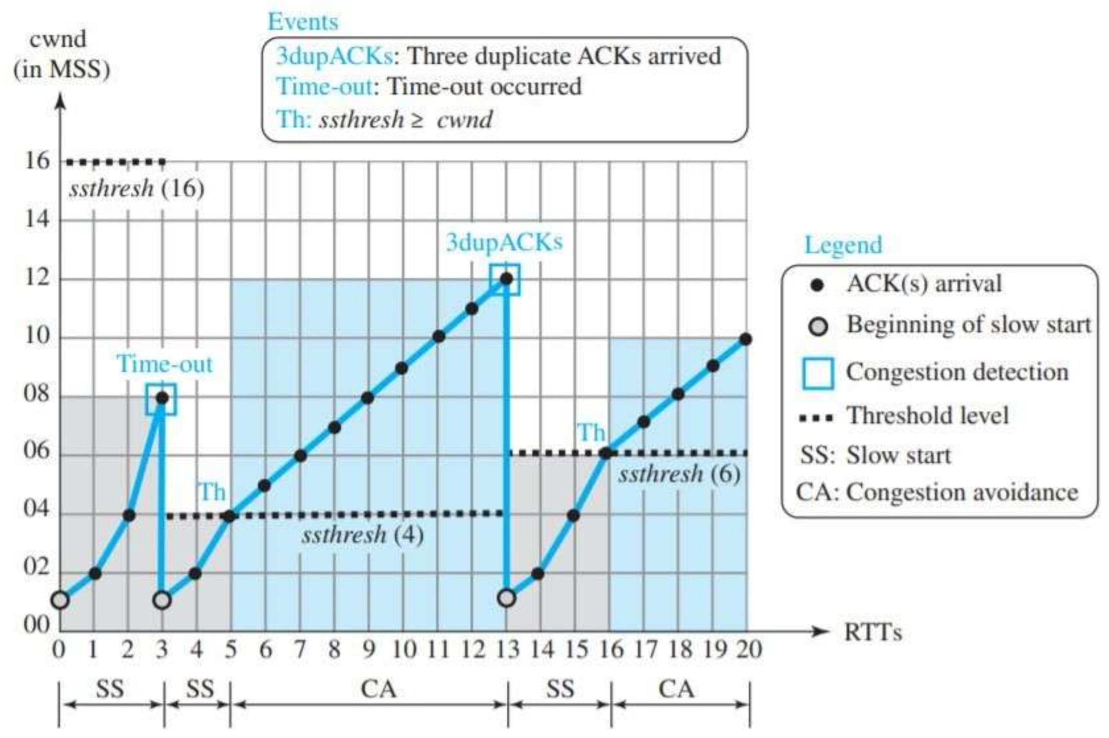
If congestion is detected in this state, TCP again resets the value of the ssthresh to half of the current cwnd and moves to the slow-start state again. Although in this version of TCP the size of ssthresh is continuously adjusted in each congestion detection, this does not mean that it necessarily becomes lower than the previous value. For example, if the original ssthresh value is 8 MSS and congestion is detected when TCP is in the congestion avoidance state and the value of the cwnd is 20, the new value of the ssthresh is now 10, which means it has been increased.

Example :

Figure 24.32 shows an example of congestion control in a Taho TCP. TCP starts data transfer and sets the ssthresh variable to an ambitious value of 16 MSS. TCP begins at the slow-start (SS) state with the cwnd = 1. The congestion window grows exponentially, but a time-out occurs after the third RTT (before reaching the threshold). TCP assumes that there is congestion in the network. It immediately sets the new ssthresh = 4 MSS (half of the current cwnd, which is 8) and begins a new slow-start (SA) state with cwnd = 1 MSS. The congestion window grows exponentially until it reaches the newly set threshold. TCP now moves to the congestion-avoidance (CA) state and the congestion window grows additively until it reaches cwnd = 12 MSS. At this moment, three duplicate ACKs arrive, another indication of congestion in the network. TCP again halves the value of ssthresh to 6 MSS and begins a new slow-start

(SS) state. The exponential growth of the cwnd continues. After RTT 15, the size of cwnd is 4 MSS. After sending four segments and receiving only two ACKs, the size of the window reaches the ssthresh (6) and TCP moves to the congestion-avoidance state. The data transfer now continues in the congestion avoidance (CA) state until the connection is terminated after RTT 20.

Figure 24.32 *Example 24.9*



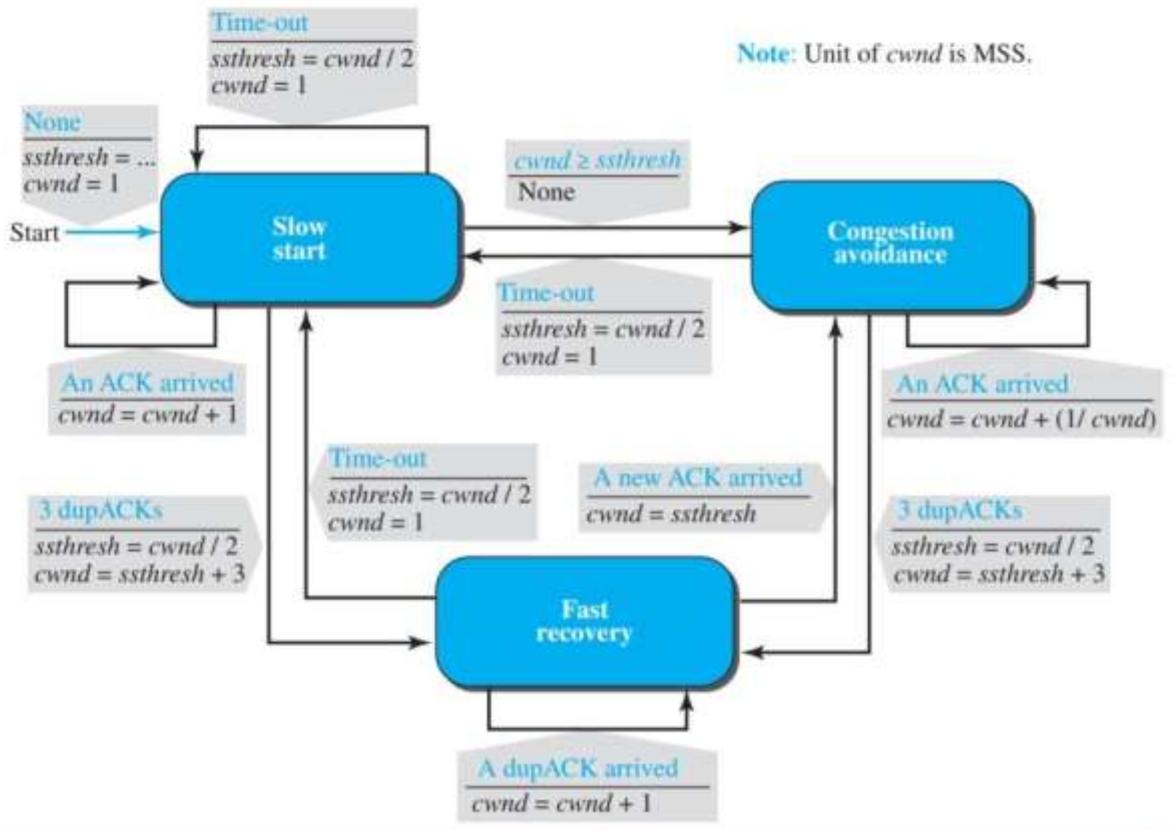
Reno TCP :

A newer version of TCP, called Reno TCP, added a new state to the congestion-control FSM, called the fast-recovery state. This version treated the two signals of congestion,time-out and the arrival of three duplicate ACKs, differently. In this version, if a time-out occurs, TCP moves to the slow-start state (or starts a new round if it is already in this state); on the other hand, if three duplicate ACKs arrive, TCP moves to the fast-recovery state and remains there as long as more duplicate ACKs arrive. The fast-recovery state is a state somewhere between the slow-start and the congestion-avoidance states. It behaves like the slow start, in which the cwnd grows exponentially, but the cwnd starts with the value of ssthresh plus 3 MSS (instead of 1). When TCP enters the fast-recovery state, three major events may occur. If duplicate ACKs continue to arrive, TCP stays in this state, but the cwnd grows exponentially. If a time-out occurs, TCP assumes that there is real congestion in the network and moves to the slow-start state. If a new (non duplicate) ACK arrives, TCP moves to the congestion-avoidance state, but deflates the size of the cwnd to the ssthresh value, as though the three duplicate ACKs have not occurred, and transition is from the slow-start state to the congestion-avoidance state. Figure 24.33 shows the simplified FSM for Reno TCP. Again, we have removed some trivial events to simplify the figure and discussion.

Example :

Figure 24.34 shows the same situation as Figure 24.32, but in Reno TCP. The changes in the congestion window are the same until RTT 13 when three duplicate ACKs arrive. At this moment, Reno TCP drops the ssthresh to 6 MSS (same as Taho TCP), but it sets the cwnd to a much higher value (ssthresh + 3 = 9 MSS) instead of 1 MSS. Reno TCP now moves to the fast recovery state. We assume that two more duplicate ACKs arrive until RTT 15, where cwnd grows exponentially. In this moment, a new ACK (not duplicate) arrives that announces the receipt of the lost segment. Reno TCP now moves to the congestion-avoidance state, but first deflates the congestion window to 6 MSS (the ssthresh value) as though ignoring the whole fast-recovery state and moving back to the previous track.

Figure 24.33 *FSM for Reno TCP*


Note: Unit of cwnd is MSS.
None / ssthresh = ... cwnd = 1
Start
Time-out / ssthresh = cwnd / 2 cwnd = 1
Slow start
cwnd ≥ ssthresh / None
Congestion avoidance
Time-out / ssthresh = cwnd / 2 cwnd = 1
An ACK arrived / cwnd = cwnd + 1
An ACK arrived / cwnd = cwnd + (1/ cwnd)
Time-out / ssthresh = cwnd / 2 cwnd = 1
A new ACK arrived / cwnd = ssthresh
3 dupACKs / ssthresh = cwnd / 2 cwnd = ssthresh + 3
3 dupACKs / ssthresh = cwnd / 2 cwnd = ssthresh + 3
Fast recovery
A dupACK arrived / cwnd = cwnd + 1

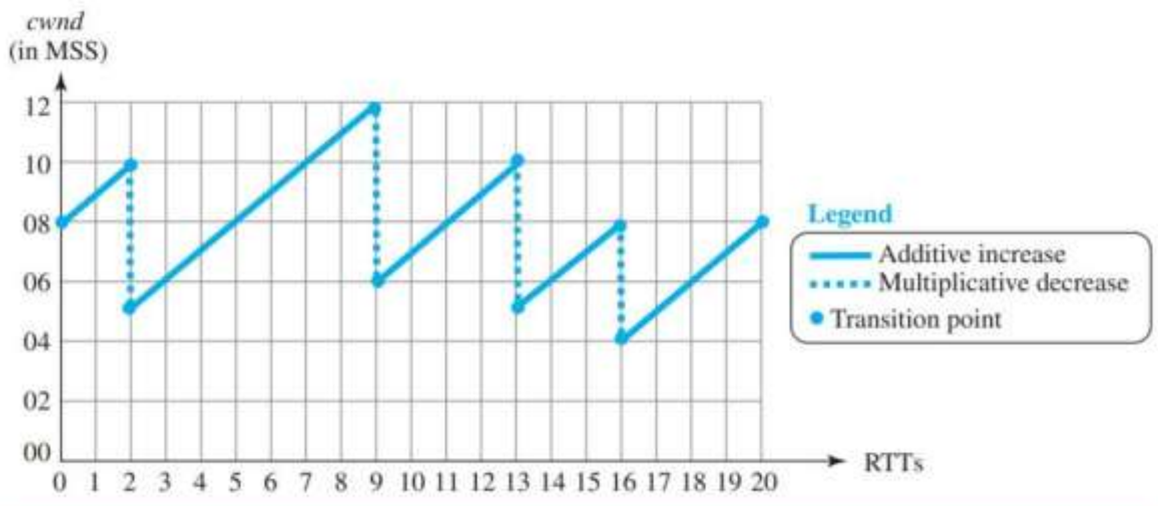
New Reno TCP :

A later version of TCP, called NewReno TCP, made an extra optimization on the Reno TCP. In this version, TCP checks to see if more than one segment is lost in the current window when three duplicate ACKs arrive. When TCP receives three duplicate ACKs, it retransmits the lost segment until a new ACK (not duplicate) arrives. If the new ACK defines the end of the window when the congestion was detected, TCP is certain that only one segment was lost. However, if the ACK number defines a position between the retransmitted segment and the end of the window, it is possible that the segment defined by the ACK is also lost. NewReno TCP retransmits this segment to avoid receiving more and more duplicate ACKs for it.

Additive Increase, Multiplicative Decrease :

Out of the three versions of TCP, the Reno version is most common today. It has been observed that, in this version, most of the time the congestion is detected and taken care of by observing the three duplicate ACKs. Even if there are some time-out events, TCP recovers from them by aggressive exponential growth. In other words, in a long TCP connection, if we ignore the slow-start states and short exponential growth during fast recovery, the TCP congestion window is cwnd = cwnd + (1 / cwnd) when an ACK arrives (congestion avoidance), and cwnd = cwnd / 2 when congestion is detected, as though SS does not exist and the length of FR is reduced to zero. The first is called additive increase; the second is called multiplicative decrease. This means that the congestion window size, after it passes the initial slow-start state, follows a saw tooth pattern called additive increase, multiplicative decrease (AIMD), as shown in Figure 24.35.

Figure 24.35 *Additive increase, multiplicative decrease (AIMD)*



Comparison Among TCP Taho , Reno TCP and New Reno TCP :

Table 1. The four phases of congestion control algorithm

Phase 1	Slow Start Algorithm
Phase 2	Congestion avoidance Algorithm
Phase 3	Fast retransmission Algorithm
Phase 4	Fast recovery Algorithm

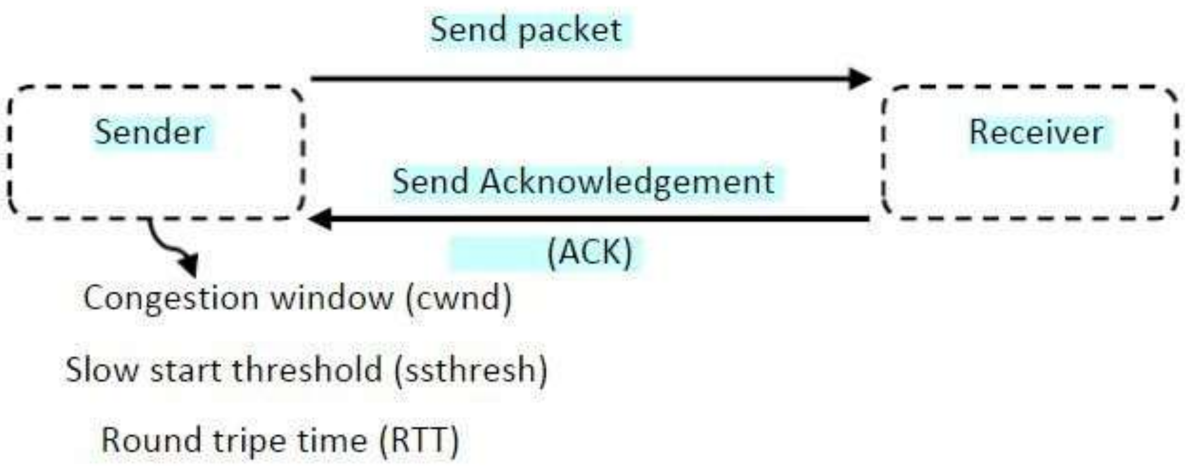


Fig. 1 Common factors used in congestion control

TCP Taho :

TCP Taho uses only the first three algorithm mentioned in Table 1 which are Slow Start, Congestion Avoidance, and Fast Retransmit . To simplify the idea discussed in algorithm steps with and without congestion will be shown here.

A) The procedure without congestion or loss packet will be as following (All terms taken from Fig.1). The procedure start with congestion window of size equal to one packet and any time we have a complete transmission (the source received an acknowledgement for sent packet before timeout status) the congestion window grow by one packet.

cwnd<= ssthresh
Connection start : cwnd=1 packet.
Each ACK : cwnd+=1 packet.

B) The procedure when we have any one of the following states: the congestion window exceeds the ssthresh, timeout, or receives three duplicate acknowledgements by source.

ssthresh = cwnd/2
cwnd = 1
Congestion avoidance
Fast retransmission
Slow start state: when ACK received for retransmitted packet.

TCP Reno :

TCP Reno uses the entire four algorithms for congestion control mentioned in Table 1. The new procedure looks like the one used in TCP Tahoe when respond to timeout. But for three duplicate acknowledgements another procedure will be used which is like following:

ssthresh = cwnd/2.
cwnd = ssthresh + 3. (fast recovery)
first ACK: cwnd +=1.
next ACK cwnd = ssthresh.

The simulation uses two senders and two receivers to test different scenarios of congestion control and Round-trip time RTT. According to the results of simulation, the raise of RTT occurs only when there is a congestion in the network. The raise in RTT property is used to timing the sender with the difference of RTT in current cycle and the previous one. The negative sum of RTT differences means we have no congestion in the current cycle of transmission while the positive sum of RTT differences means the transmission is in congestion state or will have a congestion in next cycles. The algorithm of TCP Reno will be like the following:

```
RTT_diff_sum=0;
when the sender recieves a ACK;
    RTT_diff = RTT - RIT_last;
    RTT_diff_sum + = RTT_diff;
    if(dup_ACKs> =3)
(if(RTT_diff_sum >O) llin congestion state
{ssthresh=min(cwnd,rcv_window )/2;
cwnd = ssthresh+ 3*MSS; //lower transmission rate
    retransmit the lost packet;
} else  //in non-congestion state
{ only retransmit the lost packet;
```

```
 }
}
```

TCP Reno is not the best choice to use with wireless network. The new procedure doesn’t provide a way to differentiate between packet loss caused by congestion in network and packet loss when network suffer from random bit error in wireless links.

TCP New Reno :

A very novel TCP version has been proposed. The design provides a new mechanism to differentiate between packets loss caused by high bit error and packets loss caused by network congestion. Sender, receiver and middle router all of these parts cooperate to detect congestion and control it.

The new modified Transmission Control Protocol (TCP Reno) is able to monitor the loss of wireless packets in real time. By detecting a router's buffer mechanism in response to congestion occupancy, the modified Reno is able to monitor wireless packet loss; thus being able to react accordingly and decrease the rate of wireless package loss. This is important because when high volumes of information packets are sent it can have problems reaching the desired recipient. Communication over wireless links, between computer networks and various systems, is filled with random rates of high bit error and connectivity that is intermittent due to the frequency of handoffs.

Mechanisms such as random early detection (RED) were used to find possible problematic packets in their early stages.

The modified TCP Reno uses Explicit Congestion Notification (ECN), which is an extension of REDs. This mechanism can allow the system to properly tell the difference between random wireless link errors and errors caused by the congestion of network links. It can also monitor the rate at which wireless packets are lost in a manner that helps the sender select the appropriate segment size at the right moment when packet loss is identified. These modifications to Reno, after plenty of tests and simulations, have shown to have merit and even improve the TCP efficiency; it can do this with no changes in the protocol itself which makes the modified Reno easy enough to use.