**TCP Slow Start**

**According to RFC 2581**

1. The congestion window (cwnd) is a sender-side limit on the amount of data the sender can transmit into the network before receiving an acknowledgment (ACK), while the receiver’s advertised window (rwnd) is a receiver-side limit on the amount of outstanding data. The minimum of cwnd and rwnd governs data transmission.

2. Another state variable, the slow start threshold (ssthresh), is used to determine whether the slow start or congestion avoidance algorithm is used to control data transmission.

(a) The slow start algorithm is used : cwnd < ssthresh

(b) The congestion avoidance algorithm is used : cwnd > ssthresh

(c) Either slow start or congestion avoidance : cwnd == ssthresh

3. During slow start, a TCP increments cwnd by at most SMSS bytes for each ACK received that acknowledges new data. Slow start ends when cwnd exceeds ssthresh or when congestion is observed.

4. During congestion avoidance, cwnd is incremented by 1 full-sized segment per round-trip time (RTT).

5. Congestion avoidance continues until congestion is detected. One formula commonly used to update cwnd during congestion avoidance is : **cwnd += SMSS\*SMSS/cwnd**

6. The value of ssthresh : **ssthresh = max (FlightSize / 2, 2\*SMSS)**

where, FlightSize is the amount of outstanding data in the network.

**According to RFC 5681**

7. Slow start additionally serves to start the "ACK clock" used by the TCP sender to release data into the network in the slow start, congestion avoidance, and loss recovery algorithms.

8. IW, the initial value of cwnd, MUST be set using the following guidelines as an upper bound.

(a) If SMSS > 2190 bytes: **IW = 2 \* SMSS** bytes and MUST NOT > 2 segments.

(b) If (SMSS > 1095 bytes) and (SMSS <= 2190 bytes): **IW = 3 \* SMSS** bytes and MUST NOT be more than 3 segments.

(c) If SMSS <= 1095 bytes: **IW = 4 \* SMSS** bytes and MUST NOT > 4 segments.

9. During Slow Start, it is recommended that **cwnd += min (N, SMSS)**, where N is the number of previously unacknowledged bytes acknowledged in the incoming ACK.

**Slowstart Implementation in ns-3**

1. T**cp-congestion-ops**

TcpNewReno is the default congestion control algorithm implemented in ns-3. It follows RFC 5681.

if (tcb->m\_cWnd < tcb->m\_ssThresh)

if (segmentsAcked >= 1)

{

tcb->m\_cWnd += tcb->m\_segmentSize;

return segmentsAcked - 1;

}

2. T**cp-bic (TCP Binary Increase Congestion control)**

Taking as a starting point the current window value and as a target point the last maximum window value (i.e. the cwnd value just before the loss event) a binary search technique is used to update the cwnd value at the midpoint between the two, directly or using an additive increase strategy if the distance from the current window is too large.

It inherits TCP New Reno (tcp-congestion-ops) for slow start and congestion avoidance implementation. If **m\_lastMaxCwnd == 0**, then behave as NewReno and increase cwnd 5% per RTT.

3. **TcpHighSpeed**

TCP HighSpeed is designed for high-capacity channels or, in general, for TCP connections with large congestion windows. Conceptually, with respect to the standard TCP, HighSpeed makes the cwnd grow faster during the probing phases and accelerates the cwnd recovery from losses. It works as ,

**cwnd = cwnd + (a(cwnd))/cwnd**

The function a() is calculated using a fixed RTT the value 100 ms (the lookup table for this function is taken from RFC 3649). For each congestion event, the slow start threshold is decreased by a value that depends on the size of the slow start threshold itself. Then, the congestion window is set as,

**cwnd = (1 – b(cwnd)) \* cwnd**

**Both function a() & b() are defined in tcp-highspeed**

TcpHighSpeed::TableLookupA (uint32\_t w)

TcpHighSpeed::TableLookupB (uint32\_t w)

4. **TcpHybla**

The key idea behind TCP Hybla is to obtain for long RTT connections the same instantaneous transmission rate of a reference TCP connection with lower RTT. Additional attribute “RRTT” and TraceSource “Rho” introduced in TCP Hybla.

Rho calculated as : m\_rho = max (rtt.GetMilliSeconds () / m\_rRtt.GetMilliSeconds (), 1.0)

**In slow start phase (m\_cwnd <=m\_ssThresh)**

if (segmentsAcked >= 1)

m\_cWnd = min (m\_cWnd + (increment \* m\_segmentSize), m\_ssThresh);

where, increment = pow(2 , m\_rho) – 1.0; {## increment = 2^RHO – 1}

5. **Tcp-htcp**

H-TCP has been designed for high BDP (Bandwidth-Delay Product) paths. It is a dual mode protocol. In normal conditions, it works like traditional TCP with the same rate of increment and decrement for the congestion window. However, in high BDP networks, when it finds no congestion on the path after “deltal” seconds, it increases the window size based on the alpha function as:

m\_alpha = (1 + 10 \* diffSec + 0.25 \* (diffSec \* diffSec));

where, diffSec = m\_delta – m\_deltaL (in Seconds)

m\_deltal = threshold in seconds for switching between the modes

m\_delta = the elapsed time from the last congestion.

**The cwnd calculated as** :

adder = ((m\_segmentSize \* m\_segmentSize) + (m\_cwnd \* m\_alpha)) / m\_cwnd;

adder = max (1.0, adder);

m\_cwnd += adder;

6. **Tcp-Illinois.** It is especially targeted at high-speed, long-distance networks. It uses both loss and delay as congestion signals to achieve a better throughput than the standard TCP for high-speed networks.

if (cWnd < m\_winThresh)

{

NS\_LOG\_INFO ("cWnd < winThresh, set alpha & beta to base values");

m\_alpha = m\_alphaBase;

m\_beta = m\_betaBase;

}

segmentsAcked = TcpNewReno::SlowStart (tcb, segmentsAcked);

**Calculation of ssthresh:**

segBytesInFlight = bytesInFlight / m\_segmentSize;

ssThresh = max (2.0, (1.0 - m\_beta) \* segBytesInFlight;

7. **Tcp-Scalable**

Scalable improves TCP performance to better utilize the available bandwidth of a highspeed wide area network by altering NewReno congestion window adjustment algorithm. When congestion has not been detected, for each ACK received in an RTT, Scalable increases its cwnd per:

cWnd = cWnd + 0.01

**Calculation of ssThresh**

segCwnd = bytesInFlight / m\_segmentSize;

b = 1.0 – m\_mdFactor; (#m\_mdfactor: Multiplicative Decrease factor=0.125)

ssThresh = max (2.0, segCwnd \* b);

**Congestion window**

if (segCwnd != oldCwnd)

m\_cWnd = segCwnd \* m\_segmentSize;

8. **Tcp-Vegas**

TCP Vegas is a pure delay-based congestion control algorithm implementing a proactive scheme that tries to prevent packet drops by maintaining a small backlog at the bottleneck queue.

(a) Vegas calculations will be performed only if enough RTT samples received to ensure that at least 1 of those samples was not from a delayed ACK. Till that time, it behaves as TCP- NewReno.

(b) Once enough RTT samples to perform Vegas algorithm has been received then it has to be determined whether cwnd should be increased or decreased based on the calculated difference between the expected rate and actual sending rate and the predefined thresholds (alpha, beta, and gamma).

segCwnd = GetCwndInSegments ();

tmp = m\_baseRtt.GetSeconds () / m\_minRtt.GetSeconds ();

targetCwnd = segCwnd \* tmp;

diff = segCwnd – targetCwnd;

if (diff > m\_gamma && (m\_cWnd < m\_ssThresh))

{

segCwnd = min (segCwnd, targetCwnd + 1);

m\_cWnd = segCwnd \* m\_segmentSize;

m\_ssThresh = GetSsThresh (tcb, 0);

}

else if (m\_cWnd < m\_ssThresh)

{

segmentsAcked = TcpNewReno::SlowStart (tcb, segmentsAcked);

}

**GetSsThresh()**

max (min (m\_ssThresh.Get (), m\_cWnd.Get () - m\_segmentSize), 2 \* m\_segmentSize);

**Note**

(a) To avoid congestion, Vegas linearly increases/decreases its congestion window to ensure the diff value fall between the two predefined thresholds, alpha and beta.

(b) Diff and gamma are used to determine when Vegas should change from its slow-start mode to linear increase/decrease mode.

9. **Tcp-Veno**

TCP Veno enhances Reno algorithm for more effectively dealing with random packet loss in wireless access networks by employing Vegas’s method in estimating the backlog at the bottleneck queue to distinguish between congestive and non-congestive states.

tmp = m\_baseRtt.GetSeconds () / m\_minRtt.GetSeconds ();

targetCwnd = segCwnd \* tmp;

if (segCwnd >= targetCwnd);

// Calculate the difference between actual and target cwnd//

m\_diff = segCwnd – targetCwnd;

There is an attribute as threshold for congestion detection : m\_beta(6) //default value =6

TcpVeno makes decision on cwnd modification based on the calculated **m\_diff** and its predefined threshold **m\_beta**.

**Note:**

(a) The additive increase algorithm of Reno increments cwnd by 1/cwnd for every other new ACK received after the available bandwidth has been fully utilized. Otherwise, Veno increases its cwnd by 1/cwnd upon every new ACK receipt as in Reno.

(b) In the multiplicative decrease algorithm, when Veno is in the non-congestive state, i.e. when (m\_diff < m\_beta), Veno decrements its cwnd by 1/5. Otherwise TcpVeno halves its sending rate as in Reno.