Documentation on the implementation of Congestion Control (CC) algorithms in ns-3

1. **Abstract**

2. **Introduction**

3. **Contents :**
    - Slow Start Algorithm
        - Parameters
        - Concept
        - Implementation in ns-3
    - Additive Increase Multiplicative Decrease (AIMD)
        - Concept
        - Implementation in ns-3
    - Fast Retransmit-Fast Recovery
        - Concept
            - Fast Retransmit
                - Concept
                - Implementation in ns-3
            - Parameters
            - Fast Recovery in TCP NewReno
            - Fast Recovery in SACK
                - Implementation in ns-3
    - Proportional Rate Reduction(PRR)
        - Parameters
        - Concept
        - Implementation in ns-3
    - Window Scaling
        - Concept
        - Implementation in ns-3

4. **References**

Abstract

TCP or Transmission Control Protocol provides reliability to data transferring in all end-to-end data stream services on the internet. The TCP congestion control algorithm is the key factor which plays a critical role in the level of performance and the regulates the amount of data stream within networks. With the network expansion and rapid increase of traffic, congestion control capabilities are becoming increasingly important in the TCP implementation. In this documentation, we are going to study different approaches to minimize the congestion control algorithms. Also, we will see the TCP behaviour and the approaches used to enhance the performance of TCP in ns3.

Introduction :

Congestion is a situation in a network in which more number of packets are in the network than it can be handled by the network resources. These resources include bandwidth of links,buffer space(memory) and processing capacity at intermediate nodes. Although resource allocation is necessary , the problem becomes more important as the increased load in the network because of issue of fairness. Also , the throughput of the network will also be affected .So, congestion control algorithms are used to maintain the number of packets into networks below the level at which the performance of the network is not affected.

TCP congestion control algorithms helps in preventing congestion or help minimize the congestion after it occurs. TCP was invented in 1983, but congestion control was not implemented in that time. TCP congestion control algorithm was introduced in 1988 in the form of TCP Tahoe. TCP Tahoe was the second version of TCP followed by TCP Reno which had a added feature of fast recovery. TCP Reno had many shortcomings which was improved by TCP NewReno. We will see the behaviour and implementation of different congestion control algorithms in ns-3. ns-3 is a discrete-event network simulator, targeted primarily for research and educational use.

Classes in ns-3 :

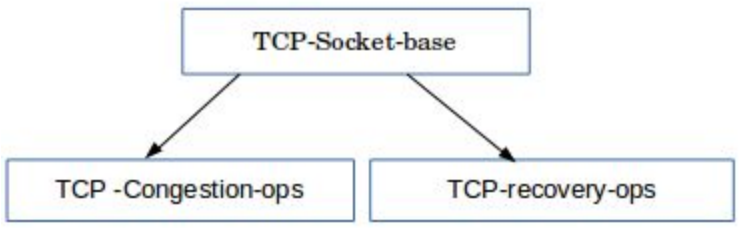
The source code for congestion control algorithms can be found in

ns-allinone-3.29/ns-3.29/src/internet/model/

`tcp-socket-base class:` This class in ns3 contains the base class for TCP packets which contains the essential components of TCP and provides a socket interface for upper layers to call. Some of the components of this call are fast recovery ,fast retransmit , connection orientation, window scaling option etc

`tcp-congestion-ops class:` This is an abstract class which provides an interface between the main socket code and congestion control. Some of the methods in this class are getssthresh() , increasewindow() , congestionavoidance() etc.

`Tcp-recovery-ops class :` The TcpRecoveryOps class provides the interface for various recovery algorithms used in recovery phase of TCP. It modifies the variable of TCP socket upon three conditions. E.g EnterRecovery() , DoRecovery() , ExitRecovery().



Relation between the three classes in ns-3

Slow-start algorithm :

Parameters

SEGMENT : A segment is ANY TCP/IP data or acknowledgment packet (or both).

SENDER MAXIMUM SEGMENT SIZE (SMSS) : The SMSS is the size of the largest segment that the sender can transmit.

RECEIVER MAXIMUM SEGMENT SIZE (RMSS):The RMSS is the size of the largest segment the receiver can receive.

RECEIVER WINDOW (RWND): The most recently advertised receiver window.

CONGESTION WINDOW (CWND): A TCP state variable that limits the amount of data a TCP can send before receiving an acknowledgement. At any given time, a TCP must not send data with a sequence number higher than the sum of the highest acknowledged sequence number and the minimum(cwnd,rwnd).

INITIAL WINDOW (IW) : The initial window is the size of the sender's congestion window after the three-way handshake is completed.

LOSS WINDOW (LW): The loss window is the size of the congestion window after a TCP sender detects loss using its retransmission/timeout Timer.

INFLIGHT / PIPE : The amount of data that has been sent but not yet cumulatively acknowledged.

Concept

Slow-start algorithm is used by the TCP sender to check the amount of outstanding data being injected in the network. In this algorithm, the sender gradually increases the transmission of packets into the network until it finds the network's capacity.

For implementing Slow-start algorithm, another TCP state variable Slow-start threshold(SSTHRESH) is used determine whether the slow start or congestion avoidance algorithm(discussed later) is used to control data transmission.

The initial value of SSTHRESH should be set arbitrarily high, but SSTHRESH must be reduced in response to congestion. Setting the SSTHRESH as high as possible allows the network to dictate the sending rate.

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

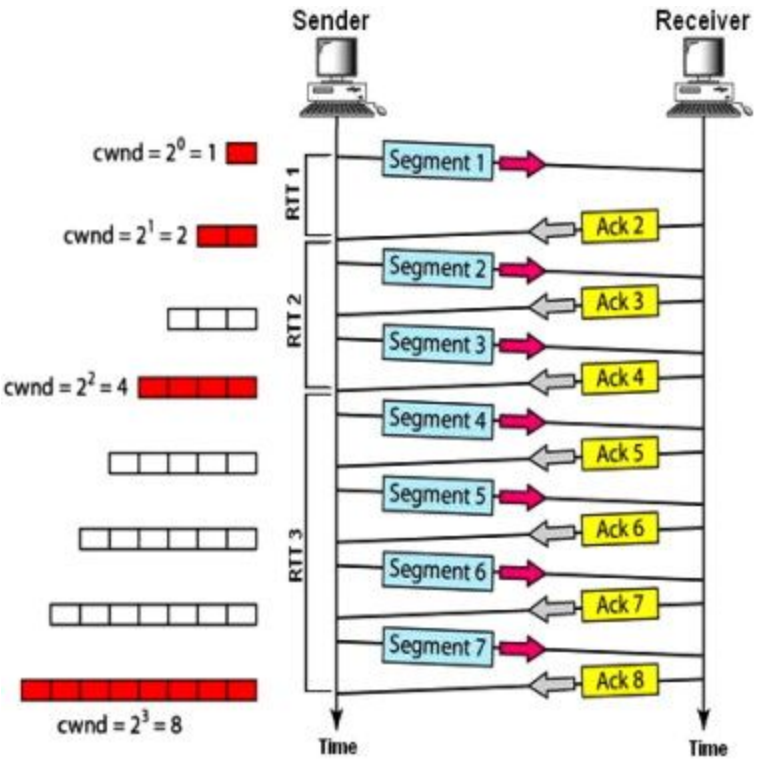
If SMSS > 2190 bytes:
IW = 2 * SMSS bytes and MUST NOT be more than 2 segments
If (SMSS > 1095 bytes) and (SMSS <= 2190 bytes):
IW = 3 * SMSS bytes and MUST NOT be more than 3 segments
If SMSS <= 1095 bytes:
IW = 4 * SMSS bytes and MUST NOT be more than 4 segments

The slow start algorithm is used when CWND < SSTHRESH. But ,when CWND = SSTHRESH , the sender may use either slow-start or congestion avoidance algorithm.

During slow start, a TCP increments CWND by at most SMSS bytes for each ACK received that cumulatively acknowledges new data. Slow start ends when CWND exceeds SSTHRESH (or, optionally when it reaches it) or when congestion is observed.
This can be expressed by

CWND + = min(SMSS,N)

Where N is the number of freshly acknowledged data.



slow-start algorithm

Credit : https://www.sciencedirect.com/science/article/pii/S1319157812000146

Implementation in ns-3

Slow-start is implemented according to RFC 5681 in ns3 in the following location

ns-allinone-3.29/ns-3.29/src/internet/model/tcp-congestion-ops.cc

```
uint32_t
TcpNewReno::SlowStart (Ptr<TcpSocketState> tcb, uint32_t
segmentsAcked)
{
  NS_LOG_FUNCTION (this << tcb << segmentsAcked);
```

```
  if (segmentsAcked >= 1)
    {
      tcb->m_cWnd += tcb->m_segmentSize;
      NS_LOG_INFO ("In SlowStart, updated to cwnd " << tcb->m_cWnd
<< " ssthresh " << tcb->m_ssThresh);
      return segmentsAcked - 1;
    }
```

In which SlowStart is a virtual member function of class TcpNewReno in tcp-congestion-ops.h , where TCPNewReno is inherited from class TcpCongestionOps in tcp-congestion-ops.h .

Congestion Avoidance:

Concept

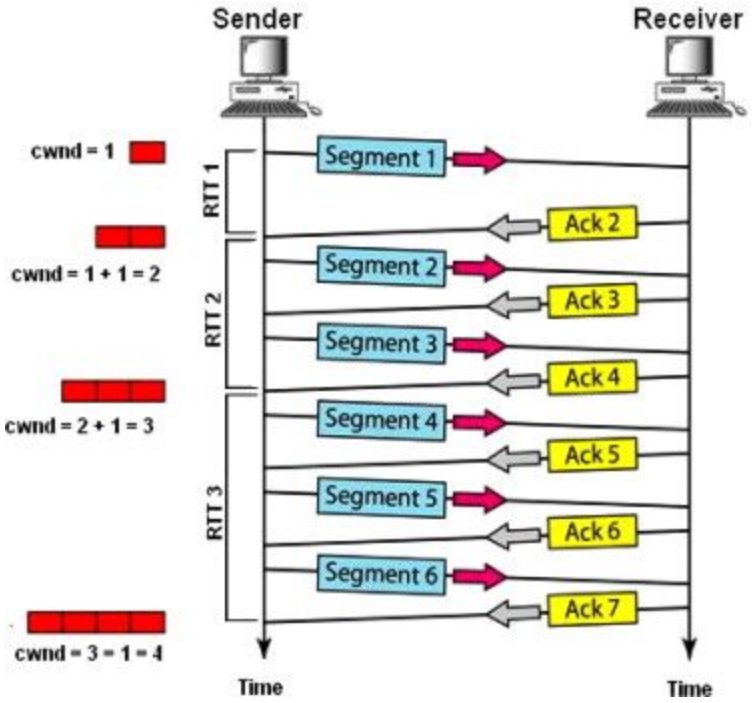
Congestion avoidance starts when CWND exceeds SSTHRESH (or, optionally when it reaches it). In this algorithm, the CWND is incremented upto 1 SEGMENT per round-trip time. The algorithm continues until congestion is detected.
The TCP uses the following formula to update CWND during congestion avoidance period

CWND += SMSS*SMSS / CWND

When the TCP sender experience segment loss due to timeout timer , the value of SSTHRESH is set as the following

SSTHRESH = max(FLIGHT SIZE / 2 , 2*SMSS)

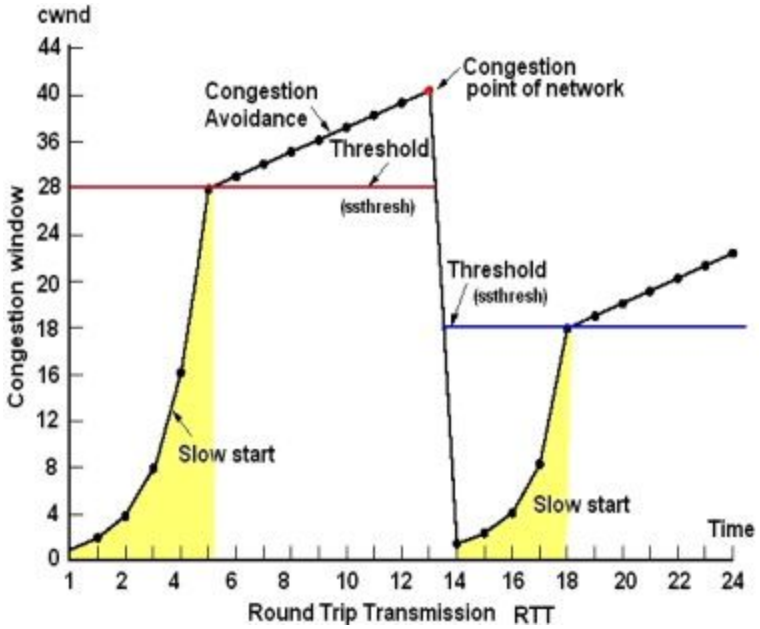
If a TCP sender detects segment loss using the timeout timer and the given segment has already been retransmitted by way of the timeout timer at least once, the value of SSTHRESH is held constant.



congestion avoidance algorithm

Credit : https://www.sciencedirect.com/science/article/pii/S1319157812000146

The following graph showing the slow-start and congestion avoidance algorithm



Credit : https://www.sciencedirect.com/science/article/pii/S1319157812000146

Implementation in ns-3

Congestion avoidance is implemented according to RFC 5681 in ns3 in the following location

ns-allinone-3.29/ns-3.29/src/internet/model/tcp-congestion-ops.cc

```
void
TcpNewReno::CongestionAvoidance (Ptr<TcpSocketState> tcb, uint32_t
segmentsAcked)
{
  NS_LOG_FUNCTION (this << tcb << segmentsAcked);

  if (segmentsAcked > 0)
    {
      double adder = static_cast<double> (tcb->m_segmentSize *
tcb->m_segmentSize) / tcb->m_cWnd.Get ();
      adder = std::max (1.0, adder);
      tcb->m_cWnd += static_cast<uint32_t> (adder);
      NS_LOG_INFO ("In CongAvoid, updated to cwnd " << tcb->m_cWnd
<<
                   " ssthresh " << tcb->m_ssThresh);
    }
}
```

In which CongestionAvoidance is a virtual member function of class TcpNewReno in tcp-congestion-ops.h.

Fast Retransmit-Fast Recovery

Concept of Fast Retransmit

Fast Retransmit is a function in which the TCP sender instantly retransmits the lost segment which got lost instead for waiting for timeout timer to expire.

Whenever the sender receives 3 duplicate ACK, it is an signal to the sender about the packet loss. After receiving 3 duplicate ACK, TCP sender checks the value of RECOVER is less than cumulative acknowledgement. If so, then set

RECOVER = Highest Sequence Numbered segment Transmitted

SSTHRESH = max (INFLIGHT / 2, 2*SMSS)

Then the TCP sender performs a retransmission of segment what appears to be the missing segment, without waiting for the timeout timer to expire.

After that it enters fast recovery phase .

Implementation in ns3

It is implemented in ns3 at location

ns-allinone-3.29/ns-3.29/src/internet/model/tcp-socket-base.cc

```
void
TcpSocketBase::DoRetransmit ()
{
NS_LOG_FUNCTION (this);
bool res;
SequenceNumber32 seq;

res = m_txBuffer->NextSeg (&seq, false);
if (!res)
{
seq = m_txBuffer->HeadSequence ();
}
NS_ASSERT (m_sackEnabled || seq == m_txBuffer->HeadSequence ());

NS_LOG_INFO ("Retransmitting " << seq);
m_tcb->m_nextTxSequence = seq;
uint32_t sz = SendDataPacket (m_tcb->m_nextTxSequence,
m_tcb->m_segmentSize, true);

NS_ASSERT (sz > 0);
}
```

Parameters

The following parameters are used for implementation of SACK in ns-3.

highSack: This variable holds the highest sequence number that has been SACKed.

retxThresh / DupThresh : This variable stores the number of duplicate acknowledgments required before fast retransmit is triggered.

highTx: This variable holds the highest sequence number that has transmitted.

highRxt: This variable stores the highest sequence number that has been retransmitted.

rescueRxt: This variable holds the highest sequence number that has been optimistically retransmitted. This variable allows one extra retransmission in order to sustain the ACK clock, which helps preventing the retransmission timer from being triggered.

inFastRec: This boolean variable determines if the TCP is currently in fast recovery phase.

cw**nd**: This variable represents the congestion window used for the TCP's sending rate.

pipe: This variable holds the number of data octets that are currently in transit.

Fast Recovery in TCP NewReno

New-Reno enters into fast-retransmit when it receives multiple duplicate packets , but it doesn't exit fast-recovery until all the data which was out standing at the time it entered fast- recovery is acknowledged. When a fresh acknowledge is received, then there are two cases : partial and full acknowledgement.

Full Acknowledgement : If this ACK acknowledges all of the data up to and including RECOVER, then the ACK acknowledges all the intermediate segments sent between the original transmission of the lost segment and the receipt of the third duplicate ACK. Then , set

CWND = SSTHRESH

And exit from fast recovery.

Partial Acknowledgement : If this ACK does not acknowledge all of the data up to and including RECOVER, then this is a partial ACK. In this case, retransmit the first unacknowledged segment.

For each partial ACK ,

CWND += SMSS

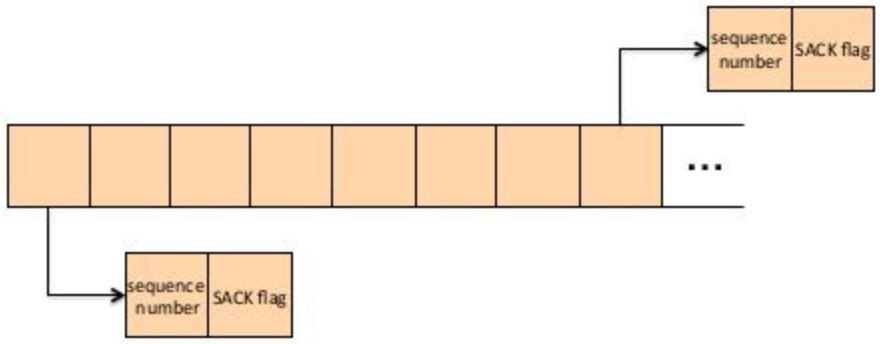
Which inflates the the CWND.

Fast Recovery by SACK

For SACK , both the endpoints i.e sender and the receiver exchange the SACK-permitted option in the SYN segment to indicate that the SACK option can be used after the establishment of the connection. If both agree, the receiver can include a SACK option into an ACK segment addressed to the sender. The objective is to carry extended information about the acknowledgment itself. In particular, the information is about the out-of-sequence blocks of data received and then queued. In this way, the sender exactly knows what blocks it should retransmit and which are correctly received, avoiding unnecessary retransmissions. When the receiver gets the missing segments, it naturally acknowledges the data.

The main features of SACK are (i) the TCP does not retransmit the segments marked as correctly received in the SACK block and (ii) the TCP does not consider the correctly received

blocks in the count of outstanding bytes.These differences avoid unnecessary retransmissions and allow to send new data for each received block. Overall, SACK recovery algorithm make TCP way more efficient when there are multiple losses inside a single window of transmission.
The scoreboard which is used by a TCP SACK sender to keep track of SACK information received from the receiver side, is implemented as a list of scoreboard entries in which each entry contains the sequence number of a transmitted segment sequence number and a SACK flag to indicate its status .



The scoreboard structure

Credit : https://pdfs.semanticscholar.org/267e/aae7022052216be5566fafc928dd1626de03.pdf

Algorithm for SACK

1. If the sender receive any ACK containing SACK information, the scoreboard must be updated using the Update ().

2. If the incoming ACK is a DUPACKs and the TCP is not currently in loss recovery, the TCP increases DupAcks by one and take the following steps:
   a.) If DupAcks >= DupThresh, go to step (4)

   b.)If DupAcks < DupThresh but IsLost (HighSack + 1) returns true , go to step(4)
   It tells that 3 DUPACK had arrived above last current cumulative acknowledgment point

3. TCP may transmit previously unsent data segments except that the number of octets which may be sent is governed by pipe and cwnd as follows:

   a.) Set HighRxt = HighSack

   b.) Run SetPipe ( )
   This function traverses the sequence space from HighSack to HighTx

and sets “pipe” variable to an estimate of the number of segments that are currently in transit between the TCP sender and the TCP receiver.

i) If IsLost (S1) returns false , increment pipe by 1.

ii) If S1 <= HighRxt , increment pipe by 1.

Where S1 is sequence space between HighSack and HighTx that has not been SACKed.

c.) If (cwnd - pipe) >= 1 SMSS
transmit up to 1 SMSS of data starting with the octet HighTx+1 and update HighTx to reflect this transmission and return to 3(b).

4. Invoke fast retransmit and enter loss recovery

a.) set RecoveryPoint = HighTx .

b.) Set SSTHRESH = CWND = (INFLIGHT / 2)

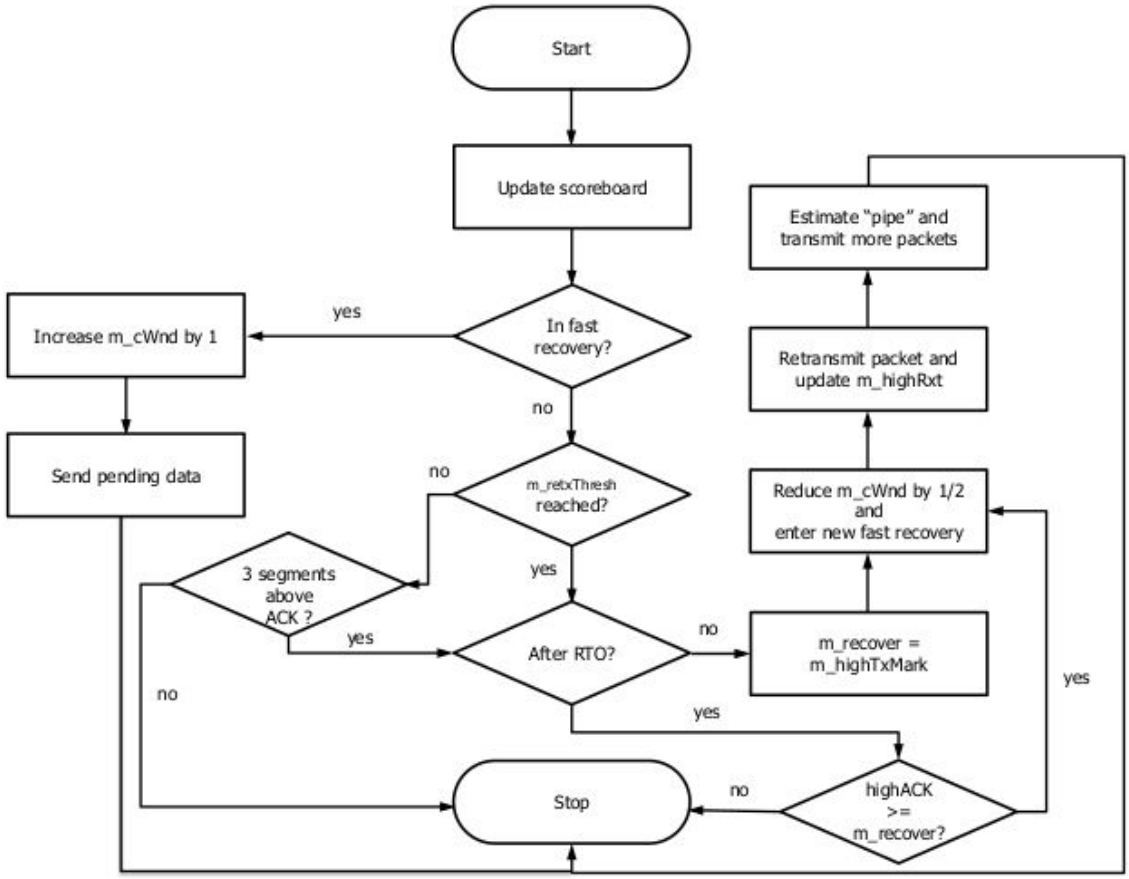
c.) Retransmit the first data segment presumed dropped , the segment starting with sequence number HighSack + 1.

d.) Run SetPipe( )

e.) If (CWND - pipe) >= 1 SMSS , the sender could transmit more than one segments, if possible.

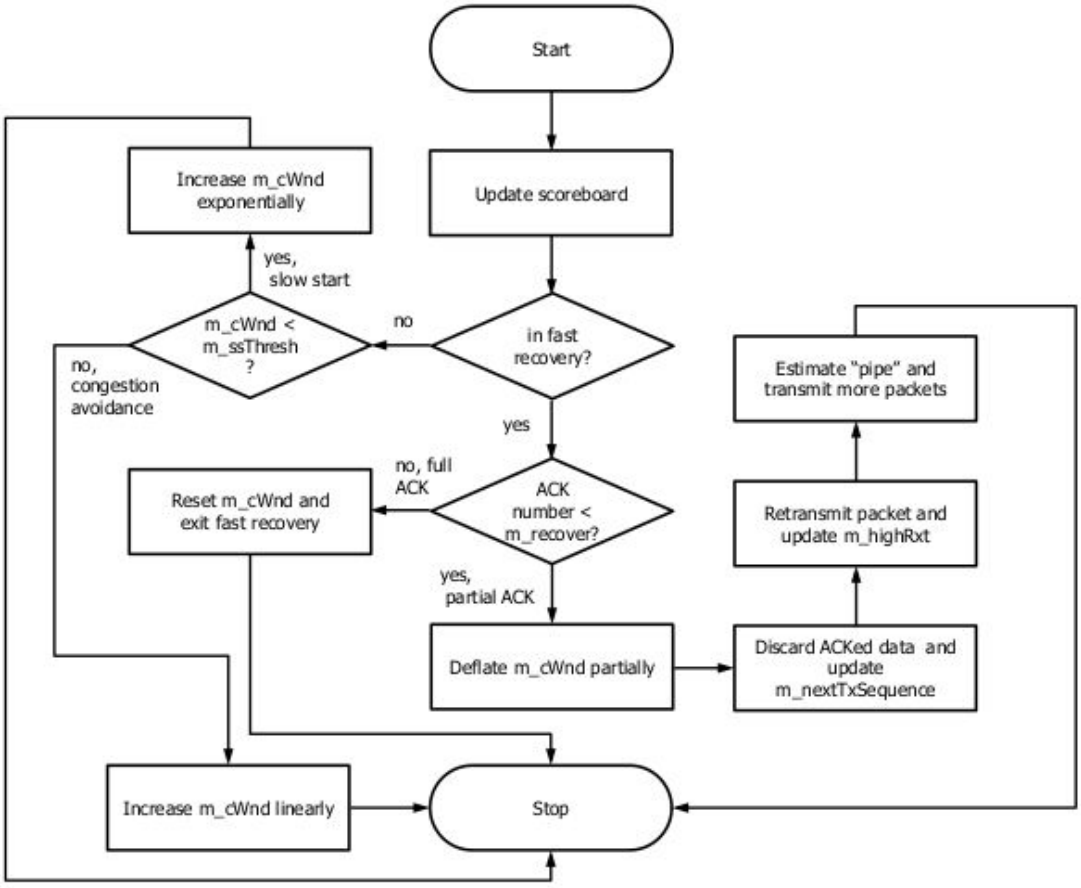
5. If HighSack >= RecoveryPoint , exit from loss recovery .

If an RTO occurs during loss recovery , RecoveryPoint must be set to HighTx , further a new value of RecoveryPoint is saved and loss recovery algorithm to be terminated.



The SackDupAck flowchart

Credit : https://pdfs.semanticscholar.org/267e/aae7022052216be5566fafc928dd1626de03.pdf



The NewAck flowchart

Credit : https://pdfs.semanticscholar.org/267e/aae7022052216be5566fafc928dd1626de03.pdf

Implementation in ns3

It is implemented in ns3 at location

ns-allinone-3.29/ns-3.29/src/internet/model/tcp-socket-base.cc

ns-allinone-3.29/ns-3.29/src/internet/model/tcp-recovery-ops.cc

Some of the functions used are :

`IsLost( S )`:This method determines if a sequence number 'S' is considered to be lost. This turns out to be true when 3 DupAcks are received.

`Update ( )`: This method is called on the arrival of an ACK and scans through the scoreboard to discard entries with sequence numbers smaller than the acknowledgment number in the received ACK segment.

`NextSeg ( )`: This routine uses the scoreboard data structure maintained by the Update() function to determine what to transmit based on the SACK information that has arrived from the data receiver.

`SetPipe ( )`**:** This method computes the number of outstanding segments. The scoreboard is traversed from the beginning to the end and only those entries with the SACK flag off and keys fall between highAck and m highTxMark are considered.

In processAck( ) , the `uint32_t oldDupAckCount` stores the number of duplicate acknowledgement. The `m_tcb->m_lastAckedSeq` stores the the sequence of last segment acknowledged.The `m_tcb->m_highTxMark` is the highest sequence number of the segment that is transmitted.
If this ACK acknowledges all of the data up to and including recover, then the ACK acknowledges all the intermediate segments sent between the original transmission of the lost segment and the receipt of the third duplicate ACK.

```
void
TcpSocketBase::ProcessAck (const SequenceNumber32 &ackNumber, bool
scoreboardUpdated,
                    const SequenceNumber32 &oldHeadSequence)
{
  NS_LOG_FUNCTION (this << ackNumber << scoreboardUpdated);
   bool exitedFastRecovery = false;
  uint32_t oldDupAckCount = m_dupAckCount;
  m_tcb->m_lastAckedSeq = ackNumber;

  bool isDupack = m_sackEnabled ?
    scoreboardUpdated
    : ackNumber == oldHeadSequence &&
    ackNumber < m_tcb->m_highTxMark;

  NS_LOG_DEBUG ("ACK of " << ackNumber <<
              " SND.UNA=" << oldHeadSequence <<
              " SND.NXT=" << m_tcb->m_nextTxSequence <<
              " in state: " << TcpSocketState::TcpCongStateName[m_tcb->m_congState] <<
              " with m_recover: " << m_recover);

  if (isDupack)
```

```
  {

    DupAck ();
  }

 if (ackNumber == oldHeadSequence
    && ackNumber == m_tcb->m_highTxMark)
  {
        return;
  }
 else if (ackNumber == oldHeadSequence
        && ackNumber > m_tcb->m_highTxMark)
  {

    NS_LOG_DEBUG ("Update nextTxSequence manually to " << ackNumber);
    m_tcb->m_nextTxSequence = ackNumber;
  }
 else if (ackNumber == oldHeadSequence)
  {
       m_congestionControl->PktsAcked (m_tcb, 1, m_tcb->m_lastRtt);
  }
 else if (ackNumber > oldHeadSequence)
  {
    uint32_t bytesAcked = ackNumber - oldHeadSequence;
    uint32_t segsAcked  = bytesAcked / m_tcb->m_segmentSize;
    m_bytesAckedNotProcessed += bytesAcked % m_tcb->m_segmentSize;

    if (m_bytesAckedNotProcessed >= m_tcb->m_segmentSize)
      {
        segsAcked += 1;
        m_bytesAckedNotProcessed -= m_tcb->m_segmentSize;
      }

      if (!isDupack)
      {
        m_dupAckCount = 0;
      }

        if (ackNumber < m_recover && m_tcb->m_congState ==
TcpSocketState::CA_RECOVERY)
      {
        if (!m_sackEnabled)
          {
            NS_LOG_INFO ("Partial ACK. Manually setting head as lost");
            m_txBuffer->MarkHeadAsLost ();
          }
        else
```

```
        {
                m_txBuffer->DeleteRetransmittedFlagFromHead ();
        }
      DoRetransmit (); // Assume the next seq is lost. Retransmit lost packet
      m_tcb->m_cWndInfl = SafeSubtraction (m_tcb->m_cWndInfl, bytesAcked);
      if (segsAcked >= 1)
        {
          m_recoveryOps->DoRecovery (m_tcb, bytesAcked, m_txBuffer->GetSacked ());
        }

      m_congestionControl->PktsAcked (m_tcb, 1, m_tcb->m_lastRtt);
      NewAck (ackNumber, m_isFirstPartialAck);

      if (m_isFirstPartialAck)
        {
          NS_LOG_DEBUG ("Partial ACK of " << ackNumber <<
                        " and this is the first (RTO will be reset);"
                        " cwnd set to " << m_tcb->m_cWnd <<
                        " recover seq: " << m_recover <<
                        " dupAck count: " << m_dupAckCount);
          m_isFirstPartialAck = false;
        }
      else
        {
          NS_LOG_DEBUG ("Partial ACK of " << ackNumber <<
                        " and this is NOT the first (RTO will not be reset)"
                        " cwnd set to " << m_tcb->m_cWnd <<
                        " recover seq: " << m_recover <<
                        " dupAck count: " << m_dupAckCount);
        }
    }
      else if (ackNumber < m_recover && m_tcb->m_congState ==
TcpSocketState::CA_LOSS)
    {
      m_congestionControl->PktsAcked (m_tcb, segsAcked, m_tcb->m_lastRtt);
      m_congestionControl->IncreaseWindow (m_tcb, segsAcked);

      NS_LOG_DEBUG (" Cong Control Called, cWnd=" << m_tcb->m_cWnd <<
                    " ssTh=" << m_tcb->m_ssThresh);
      if (!m_sackEnabled)
        {
          NS_ASSERT_MSG (m_txBuffer->GetSacked () == 0,
                         "Some segment got dup-acked in CA_LOSS state: " <<
                         m_txBuffer->GetSacked ());
        }
      NewAck (ackNumber, true);
    }
```

```
    else
     {
       if (m_tcb->m_congState == TcpSocketState::CA_OPEN)
        {
          m_congestionControl->PktsAcked (m_tcb, segsAcked, m_tcb->m_lastRtt);
        }
       else if (m_tcb->m_congState == TcpSocketState::CA_DISORDER)
        {
          if (segsAcked >= oldDupAckCount)
            {
              m_congestionControl->PktsAcked (m_tcb, segsAcked - oldDupAckCount,
m_tcb->m_lastRtt);
            }

          if (!isDupack)
            {
              m_congestionControl->CongestionStateSet (m_tcb, TcpSocketState::CA_OPEN);
              m_tcb->m_congState = TcpSocketState::CA_OPEN;
              NS_LOG_DEBUG (segsAcked << " segments acked in CA_DISORDER, ack of "
<<
                            ackNumber << " exiting CA_DISORDER -> CA_OPEN");
            }
          else
            {
              NS_LOG_DEBUG (segsAcked << " segments acked in CA_DISORDER, ack of "
<<
                            ackNumber << " but still in CA_DISORDER");
            }
        }
             else if (m_tcb->m_congState == TcpSocketState::CA_RECOVERY)
        {
          m_isFirstPartialAck = true;

             segsAcked = static_cast<uint32_t>(ackNumber - m_recover) /
m_tcb->m_segmentSize;
          m_congestionControl->PktsAcked (m_tcb, segsAcked, m_tcb->m_lastRtt);
          m_congestionControl->CwndEvent (m_tcb,
TcpSocketState::CA_EVENT_COMPLETE_CWR);
          m_congestionControl->CongestionStateSet (m_tcb, TcpSocketState::CA_OPEN);
          m_tcb->m_congState = TcpSocketState::CA_OPEN;
          exitedFastRecovery = true;
          m_dupAckCount = 0; // From recovery to open, reset dupack

          NS_LOG_DEBUG (segsAcked << " segments acked in CA_RECOVER, ack of " <<
                        ackNumber << ", exiting CA_RECOVERY -> CA_OPEN");
        }
       else if (m_tcb->m_congState == TcpSocketState::CA_LOSS)
```

```
        {
          m_isFirstPartialAck = true;

          segsAcked = (ackNumber - m_recover) / m_tcb->m_segmentSize;

          m_congestionControl->PktsAcked (m_tcb, segsAcked, m_tcb->m_lastRtt);

          m_congestionControl->CongestionStateSet (m_tcb, TcpSocketState::CA_OPEN);
          m_tcb->m_congState = TcpSocketState::CA_OPEN;
          NS_LOG_DEBUG (segsAcked << " segments acked in CA_LOSS, ack of" <<
                        ackNumber << ", exiting CA_LOSS -> CA_OPEN");
        }

      if (exitedFastRecovery)
        {
          NewAck (ackNumber, true);
          m_recoveryOps->ExitRecovery (m_tcb);
          NS_LOG_DEBUG ("Leaving Fast Recovery; BytesInFlight() = " <<
                        BytesInFlight () << "; cWnd = " << m_tcb->m_cWnd);
        }
      else
        {
          m_congestionControl->IncreaseWindow (m_tcb, segsAcked);

          m_tcb->m_cWndInfl = m_tcb->m_cWnd;

          NS_LOG_LOGIC ("Congestion control called: " <<
                        " cWnd: " << m_tcb->m_cWnd <<
                        " ssTh: " << m_tcb->m_ssThresh <<
                        " segsAcked: " << segsAcked);

          NewAck (ackNumber, true);
        }
    }
  }
}
```

```
void
TcpSocketBase::EnterRecovery ()
{
  NS_LOG_FUNCTION (this);
  NS_ASSERT (m_tcb->m_congState != TcpSocketState::CA_RECOVERY);

  NS_LOG_DEBUG
(TcpSocketState::TcpCongStateName[m_tcb->m_congState] << " ->
CA_RECOVERY");

  if (!m_sackEnabled)
    {

      m_txBuffer->AddRenoSack ();
      m_txBuffer->MarkHeadAsLost ();
    }
  else
    {
      if (!m_txBuffer->IsLost (m_txBuffer->HeadSequence ()))
        {
                m_txBuffer->MarkHeadAsLost ();
        }
    }

    m_recover = m_tcb->m_highTxMark;

  m_congestionControl->CongestionStateSet (m_tcb,
TcpSocketState::CA_RECOVERY);
  m_tcb->m_congState = TcpSocketState::CA_RECOVERY;

   uint32_t bytesInFlight = m_sackEnabled ? BytesInFlight () :
BytesInFlight () + m_tcb->m_segmentSize;
  m_tcb->m_ssThresh = m_congestionControl->GetSsThresh (m_tcb,
bytesInFlight);
  m_recoveryOps->EnterRecovery (m_tcb, m_dupAckCount,
UnAckDataCount (), m_txBuffer->GetSacked ());

  NS_LOG_INFO (m_dupAckCount << " dupack. Enter fast recovery
mode." <<"Reset cwnd to " << m_tcb->m_cWnd << ", ssthresh to " <<
m_tcb->m_ssThresh << " at fast recovery seqnum " << m_recover <<
 " calculated in flight: " << bytesInFlight);


  DoRetransmit ();

}
```

```
void
TcpSocketBase::DupAck ()
{
      NS_LOG_FUNCTION (this);
      if (m_tcb->m_congState == TcpSocketState::CA_LOSS)
    {
      return;
    }
     if (m_tcb->m_congState != TcpSocketState::CA_RECOVERY)
    {
      ++m_dupAckCount;
    }

  if (m_tcb->m_congState == TcpSocketState::CA_OPEN)
    {
      NS_ASSERT_MSG (m_dupAckCount == 1, "From OPEN->DISORDER but
with " <<m_dupAckCount << " dup ACKs");

      m_congestionControl->CongestionStateSet (m_tcb,
TcpSocketState::CA_DISORDER);
      m_tcb->m_congState = TcpSocketState::CA_DISORDER;

      NS_LOG_DEBUG ("CA_OPEN -> CA_DISORDER");
    }


     if (m_tcb->m_congState == TcpSocketState::CA_RECOVERY)
    {
      if (!m_sackEnabled)
        {
                    m_txBuffer->AddRenoSack ();
        }
      m_recoveryOps->DoRecovery (m_tcb, 0, m_txBuffer->GetSacked
());
      NS_LOG_INFO (m_dupAckCount << " Dupack received in fast
recovery mode."
                   "Increase cwnd to " << m_tcb->m_cWnd);
    }
     …
      else if (m_tcb->m_congState == TcpSocketState::CA_DISORDER)
    {
      if ((m_dupAckCount == m_retxThresh) && (m_highRxAckMark >=
m_recover))
        {
```

```
          EnterRecovery ();
          NS_ASSERT (m_tcb->m_congState ==
TcpSocketState::CA_RECOVERY);
        }
      else if (m_txBuffer->IsLost (m_highRxAckMark +
m_tcb->m_segmentSize))
        {
          EnterRecovery ();
          NS_ASSERT (m_tcb->m_congState ==
TcpSocketState::CA_RECOVERY);
        }
      else
        {
           if (!m_sackEnabled && m_limitedTx)
            {
              m_txBuffer->AddRenoSack ();
            }
        }
```

In `DupAck()`, the congestion state checked. The packet is also checked not only considered to be lost but also disordered.

//Implementation in tcp-recovery-ops.cc

```
void
TcpClassicRecovery::EnterRecovery (Ptr<TcpSocketState> tcb,
uint32_t dupAckCount,uint32_t unAckDataCount, uint32_t
lastSackedBytes)
{
  NS_LOG_FUNCTION (this << tcb << dupAckCount << unAckDataCount <<
lastSackedBytes);
  NS_UNUSED (unAckDataCount);
  NS_UNUSED (lastSackedBytes);
  tcb->m_cWnd = tcb->m_ssThresh;
  tcb->m_cWndInfl = tcb->m_ssThresh + (dupAckCount *
tcb->m_segmentSize);
}
```

The above code is called from `EnterRecovery()` and `DupAck()` from tcp-socket-base.cc.

```
void
TcpClassicRecovery::DoRecovery (Ptr<TcpSocketState> tcb, uint32_t
lastAckedBytes,uint32_t lastSackedBytes)
{
  NS_LOG_FUNCTION (this << tcb << lastAckedBytes <<
lastSackedBytes);
  NS_UNUSED (lastAckedBytes);
  NS_UNUSED (lastSackedBytes);
  tcb->m_cWndInfl += tcb->m_segmentSize;
}
```

The above code is called from `ProcessAck()` from tcp-socket-base.cc.

```
void
TcpClassicRecovery::ExitRecovery (Ptr<TcpSocketState> tcb)
{
  NS_LOG_FUNCTION (this << tcb);
    tcb->m_cWndInfl = tcb->m_ssThresh.Get ();
}
```

Proportional Rate Reduction(PRR)

Parameters

recoverfs: value of pipe at the start of recovery phase

prr_delivered: total bytes delivered during recovery phase

prr_out: total bytes sent during recovery phase

sndcnt: represents number of bytes to be sent in response to each ACK

limit : data sending limit for a reduction bound

DeliveredData : The total number of bytes that the current ACK indicates have been delivered to the receiver.

Concept

On detecting a packet loss through 3 duplicate ACK , the sender retransmits the lost packets , updates the SSTHRESH and enters into recovery phase. It is one of the few recovery algorithms proposed along with the likes of rate halving and Selective ACK based recovery.

In Selective ACK based recovery , when bulk packets are lost/dropped due to congested network the value of INFLIGHT packets decreases much below CWND. This enables the sender to aggressively transmit the the packets into the network which makes the stability of the network.

In rate halving algorithm, the sender does not send a burst of packets when the value of INFLIGHT data falls much below CWND. Instead, it sends one new packet per ACK in the network and continues to do so till the end of the recovery phase which is very conservative in nature.

In PRR, CWND is updated based on the values of INFLIGHT data and SSTHRESH.
If the value of INFLIGHT > SSTHRESH,

sndcnt = ceil (prr_delivered * ssthresh / recoverfs) - prr_out

CWND = INFLIGHT + sndcnt

Else if SSTHRESH > INFLIGHT

CWND is updated using one of the PRR with Conservative Reduction Bound (PRR-CRB) or PRR with Slow Start Reduction Bound (PRR-SSRB).

For PRR-CRB,

limit = prr_delivered − prr_out

sndcnt = min (SSTHRESH − INFLIGHT , limit)

cwnd = pipe + sndcnt

For PRR-SSRB,

limit = max (prr_delivered − prr_out, DeliveredData) + SMSS

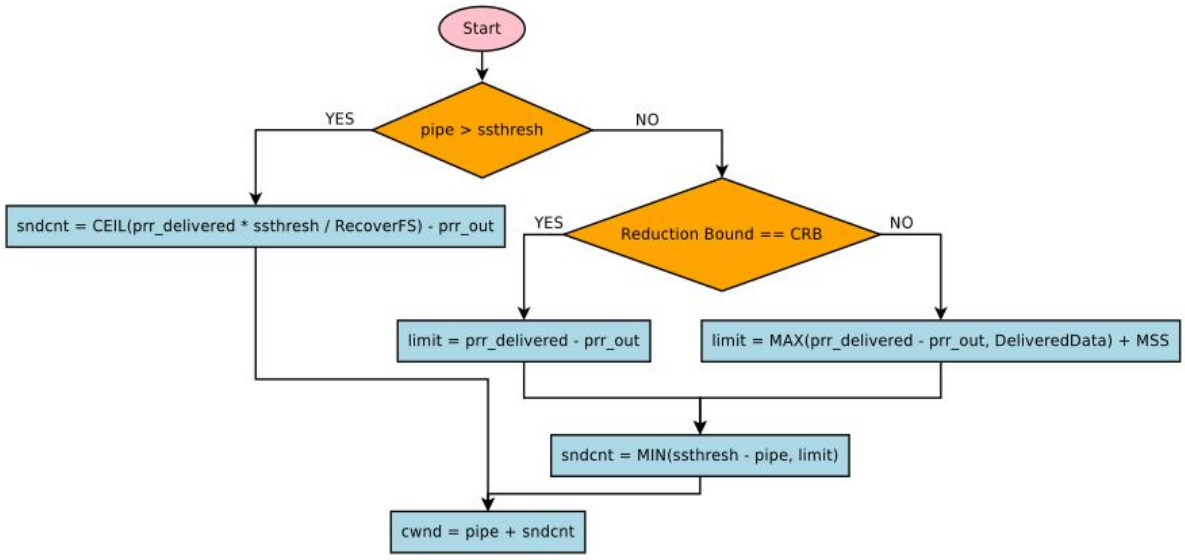
sndcnt = min (SSTHRESH − INFLIGHT , limit)

CWND = pipe + sndcnt

Among all these algorithms , the SACK is most aggressive while Rate Halving is most conservative when INFLIGHT is very less than CWND.

PRR-SSRB is aggressive but lesser than SACK based loss recovery and PRR-CRB is conservative but lesser than Rate Halving.

Flow chart



Flow diagram for the PRR algorithm

Credit : https://www.researchgate.net/publication/325330598_Proportional_rate_reduction_for_ns-3_TCP

Implementation in ns3

PRR is implemented in ns3 in

ns-allinone-3.29/ns-3.29/src/internet/model/tcp-prr-recovery.cc

```
void
TcpPrrRecovery::EnterRecovery (Ptr<TcpSocketState> tcb, uint32_t
dupAckCount,uint32_t unAckDataCount, uint32_t lastSackedBytes)
{
  NS_LOG_FUNCTION (this << tcb << dupAckCount << unAckDataCount <<
lastSackedBytes);
  NS_UNUSED (dupAckCount);

  m_prrOut = 0;
  m_prrDelivered = 0;
  m_recoveryFlightSize = unAckDataCount;
  m_previousSackedBytes = lastSackedBytes;

  DoRecovery (tcb, 0, lastSackedBytes);
}
```

This the beginning of recovery phase in which the values of variables are initialized.
Where ,
`m_recoveryFlightSize` is defined above as recoverfs and `m_previousSackedBytes` is the last segment that gets acknowledged. `m_prrOut` and `m_prrDelivered` are initialized as 0(zero) because while entering the recovery phase as no packets are transmitted and delivered at the time of entering the recovery phase. After initializing the variables, `DoRecovery()` function is called.

```
void
TcpPrrRecovery::DoRecovery (Ptr<TcpSocketState> tcb, uint32_t
lastAckedBytes,uint32_t lastSackedBytes)
{
  NS_LOG_FUNCTION (this << tcb << lastAckedBytes <<
lastSackedBytes);
  uint32_t lastDeliveredBytes;
  int changeInSackedBytes = int (lastSackedBytes -
m_previousSackedBytes);
```

```
  lastDeliveredBytes = lastAckedBytes + changeInSackedBytes > 0 ?
lastAckedBytes + changeInSackedBytes : 0;
  m_previousSackedBytes = lastSackedBytes;
  m_prrDelivered += lastDeliveredBytes;
```

We are calculating storing the number of packets that are freshly acknowledged in `changeInSackedByte`. DeliveredData is the defined in the snippet as `lastDeliveredBytes`. In `lastDeliveredBytes` we are aggregating the previous acknowledged data with new acknowledged data. In `m_prrDelivered` , the DeliveredData is added with the previous value of `m_prrDelivered` to get the recent result.

Now, we are comparing the values of SSTHRESH and INFLIGHT data.
If SSTHRESH < INFLIGHT , then the following will be executed.

```
int sendCount;
  if (tcb->m_bytesInFlight > tcb->m_ssThresh)
    {
       sendCount = std::ceil (m_prrDelivered * tcb->m_ssThresh *
1.0 / m_recoveryFlightSize) - m_prrOut;
    }
```

Here, `m_bytesInFlight` and `sendCount` is defined as INFLIGHT data and sndcnt above respectively.

Else if , SSTHRESH > INFLIGHT , then one the following will be executed.

```
 if (m_reductionBoundMode == CRB)
        {
          limit = m_prrDelivered - m_prrOut;
        }
      else if (m_reductionBoundMode == SSRB)
        {
          limit = std::max (m_prrDelivered - m_prrOut,
lastDeliveredBytes) + tcb->m_segmentSize;
        }
      sendCount = std::min (limit, static_cast<int>
(tcb->m_ssThresh - tcb->m_bytesInFlight));
    }
```

```
```

The value of limit is different for PRR-CRB and PRR-SSRB.

```
sendCount = std::max (sendCount, static_cast<int> (m_prrOut > 0 ? 0
: tcb->m_segmentSize));
  tcb->m_cWnd = tcb->m_bytesInFlight + sendCount;
  tcb->m_cWndInfl = tcb->m_cWnd;
```

The above snippet sets the value of CWND after exiting from PRR recovery.

Windows Scaling

Concept

The original TCP specification allocated 16 bits for advertising the receiver window size ,upper bound 16 bits = 65535 bytes. which turns out to be not enough to get optimal performance.
RFC 1323 was drafted which allows us to raise the max receiver window size from 65535 bytes to 1GB. The window scaling option is communicated during the three handshake and carries a value that represents the number of bits to left shift the 16 bit window size field. TCP window scaling options can be checked in linux by

$> sysctl net.ipv4.tcp_window_scaling

Implementation in ns3

This is how the scaling factor is calculated:

```
uint8_t scale = 0;
```

```
while (maxSpace > m_maxWinSize)
      {
      maxSpace = maxSpace >> 1;
      ++scale;
      }
```

Here scale is the scaling factor, maxSpace is the size of segment

Calculation of advertised window size:

```
w >>= m_rcvWindShift;
```

Here w is window size, m_rcvWindShift is the number of bits by which the advertised window size has to be right shifted before being put on the packet header.

References

1. https://www.sciencedirect.com/science/article/pii/S1319157812000146
2. https://tools.ietf.org/html/rfc5681 “TCP Congestion Control”
3. https://www.researchgate.net/publication/325330598_Proportional_rate_reduction_for_ns-3_TCP
4. https://vimeo.com/album/4649484/video/223666358” Mohit Tahiliani, ns-3 training 2017, session 6 (TCP)”
5. https://www.nsnam.org/doxygen/ “ns-3 documentation”
6. https://tools.ietf.org/html/rfc6675 “A conservative loss recovery algorithm based on selective acknowledgement(SACK) for TCP”
7. High Performance Browser Networking (O'Reilly) “window scaling”
8. https://pdfs.semanticscholar.org/267e/aae7022052216be5566fafc928dd1626de03.pdf ”An Implementation of the SACK-Based Conservative Loss Recovery Algorithm for TCP in ns-3 ”