# **TCPW BR**: A Wireless Congestion Control Scheme Base on RTT

Project Update 1
Saem Hasan - 1705027

# Add Parameter

### *3.1 Congestion level division*

The first step is to determine an accurate RTT estimation scheme. At present, there are many versions for detecting RTT values. This paper uses the method of timeout retransmission timer to predict the round trip delay [Leu, Jenq and Jiang (2011)]. SRTT + (1-β)×$RTT_{NEW}$→×SRTT where SRTT is a smooth RTT estimate and $RTT_{NEW}$ represents the current RTT value (take β=1/8).

Substituting the measured RTT values into the following weighted average mathematical expressions (1) and (2) indirectly reflects network congestion. The specific practices are as follows:

$$F = RTT_{max} - RTT_{min} \quad (1)$$

$$R = (RTT - RTT_{min})/F \quad (2)$$

Here, F is the variation range of RTT, and $RTT_{max}$ and $RTT_{min}$ respectively represent the maximum and minimum values of the measured RTT during the transmission of the TCP data segment; RTT is the RTT value of the current time measured according to the current segment. R∈[0,1] indicates the extent to which the currently confirmed data segment is used in the network transmission process. The smaller the R, the less time the data segment spends, and the network is idle; otherwise, the network is more congested. Divide R into 4 levels L, as shown in Tab. 2, where a higher level indicates a greater likelihood of congestion. The maximum value of the round trip delay is set to a value not greater than the timeout timer.

**Table 2:** Congestion level classification

| R | [0,0.25] | (0.25,0.5] | (0.5,0.75] | (0.75,1] |
|---|---|---|---|---|
| L | 1 | 2 | 3 | 4 |

# Add Parameter

Help

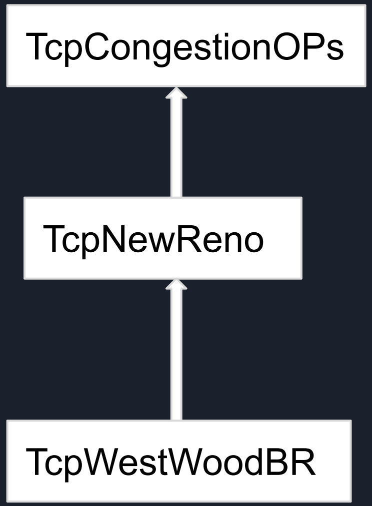
estwood.cc | tcp-socket-state.h | tcp-westwood.h | tcp-congestion-ops.h | tcp-congestion-ops.cc | tcp-socket-base.h

src > internet > model > tcp-socket-state.h > {} ns3 > TcpSocketState

```
TracedValue<DataRate>  m_pacingRate {0};         //!< Current Pacing rate
uint16_t               m_pacingSsRatio {0};      //!< SS pacing ratio
uint16_t               m_pacingCaRatio {0};      //!< CA pacing ratio
bool                   m_paceInitialWindow {false}; //!< Enable/Disable pacing for the initial window

Time                   m_minRtt  {Time::Max ()};   //!< Minimum RTT observed throughout the connection

+ Time                 m_maxRtt
                                                 //!< Bytes in flight
                                              ;  //!< Last RTT sample collected

Ptr<TcpRxBuffer>       m_rxBuffer;               //!< Rx buffer (reordering buffer)

EcnMode_t              m_ecnMode {ClassicEcn}; //!< ECN mode
UseEcn_t               m_useEcn {Off};         //!< Socket ECN capability
```

# New Class and Header File

+ tcp-westwood-br.h
+ tcp-westwood-br.cc



# Implementation

```
class TcpWestwoodBR : public TcpNewReno{
    public:
    //public methods
    private:
    //private parameters
}
```

# Private Parameters

```
TracedValue<double>     m_currentBW;                  //!< Current value of the estimated BW
 double                  m_lastSampleBW;               //!< Last bandwidth sample
 double                  m_lastBW;                     //!< Last bandwidth sample after being
filtered
 enum FilterType         m_fType;                      //!< 0 for none, 1 for Tustin

 uint32_t                m_ackedSegments;              //!< The number of segments ACKed
between RTTs
 Time                    m_lastAck;
```

# Private Methods

```
//Update the total number of acknowledged packets during the current RTT
void UpdateAckedSegments (int acked);


//Estimate the network's bandwidth

void EstimateBW (const Time& rtt, Ptr<TcpSocketState> tcb);
```

# Public Methods

```
static TypeId GetTypeId (void); //Get the type ID.
```

```
TcpWestwood (void);
TcpWestwood (const TcpWestwood& sock);//Copy constructor
```

```
~TcpWestwood (void);
```

```
Ptr<TcpCongestionOps> Fork ();
```

# Add Public Method

### *3.1 Congestion level division*

The first step is to determine an accurate RTT estimation scheme. At present, there are many versions for detecting RTT values. This paper uses the method of timeout retransmission timer to predict the round trip delay [Leu, Jenq and Jiang (2011)]. SRTT + $(1\text{-}\beta)\times RTT_{NEW} \rightarrow \times SRTT$ where SRTT is a smooth RTT estimate and $RTT_{NEW}$ represents the current RTT value (take $\beta=1/8$).

Substituting the measured RTT values into the following weighted average mathematical expressions (1) and (2) indirectly reflects network congestion. The specific practices are as follows:

$$F = RTT_{max} - RTT_{min} \tag{1}$$

$$R = (RTT - RTT_{min})/F \tag{2}$$

Here, F is the variation range of RTT, and $RTT_{max}$ and $RTT_{min}$ respectively represent the maximum and minimum values of the measured RTT during the transmission of the TCP data segment; RTT is the RTT value of the current time measured according to the current segment. $R \in [0,1]$ indicates the extent to which the currently confirmed data segment is used in the network transmission process. The smaller the R, the less time the data segment spends, and the network is idle; otherwise, the network is more congested. Divide R into 4 levels L, as shown in Tab. 2, where a higher level indicates a greater likelihood of congestion. The maximum value of the round trip delay is set to a value not greater than the timeout timer.

**Table 2:** Congestion level classification

| R | [0,0.25] | (0.25,0.5] | (0.5,0.75] | (0.75,1] |
|---|---|---|---|---|
| L | 1 | 2 | 3 | 4 |

# Public Methods

```
uint32_t GetSsThresh (Ptr<const TcpSocketState> tcb, uint32_t bytesInFlight);

void IncreaseWindow (Ptr<TcpSocketState> tcb, uint32_t segmentsAcked);

void PktsAcked (Ptr<TcpSocketState> tcb, uint32_t packetsAcked,
                            const Time& rtt);

uint32_t SlowStart (Ptr<TcpSocketState> tcb, uint32_t segmentsAcked);

void CongestionAvoidance (Ptr<TcpSocketState> tcb, uint32_t segmentsAcked);

int GetCongestionLevel(Ptr<const TcpSocketState> tcb, const Time& rtt); //for L

int GetGrowthFactor(Ptr<const TcpSocketState> tcb, const Time& rtt); // for P
```

The algorithm is described as follows:

```
(1) Each time an ACK of a new data segment is received,
    If (congestion level=1||congestion level=2)//Think it is wireless packet loss, mild congestion
        Cwnd=Cwnd+1;
    If (Cwnd>Ssthresh)
        Cwnd=Cwnd+(1/Cwnd)*p;
(2) After receiving a duplicate ACK before timing out
    If (duplicate ACK=3&& congestion level=1)
        Fast retransmission;
        Quick recovery
    If (duplicate ACK=2&& (congestion level=3||congestion level=4))//Think it is a congestion packet
    Slow start or congestion avoidance;
      Cwnd=Cwnd*p;
      Ssthresh =(BWE*RTTmin)/seg_size;
    If (Cwnd>Ssthresh) then Cwnd=Ssthresh;
    If (duplicate ACK=3&& congestion level>2)
    Slow start or congestion avoidance;
      Cwnd=Cwnd*p;
      Ssthresh=(BWE*RTTmin)/seg_size;
    If (Cwnd>Ssthresh) then Cwnd=Ssthresh;
```

lp

twood.cc | tcp-socket-base.cc × | tcp-socket-state.h | tcp-westwood.h | tcp-congestion-ops.h | tcp

c > internet > model > tcp-socket-base.cc > {} ns3 > ProcessAck(const SequenceNumber32 &, bool, uint32_t, const SequenceNur

```
          }
        }
      }
}

/* Process the newly received ACK */
void
TcpSocketBase::ReceivedAck (Ptr<Packet> packet, const TcpHeader& tcpHeader)
{
  NS_LOG_FUNCTION (this << tcpHeader);

  NS_ASSERT (0 != (tcpHeader.GetFlags () & TcpHeader::ACK));
  NS_ASSERT (m_tcb->m_segmentSize > 0);

  uint32_t previousLost = m_txBuffer->GetLost ();
  uint32_t priorInFlight = m_tcb->m_bytesInFlight.Get ();
```

The algorithm is described as follows:

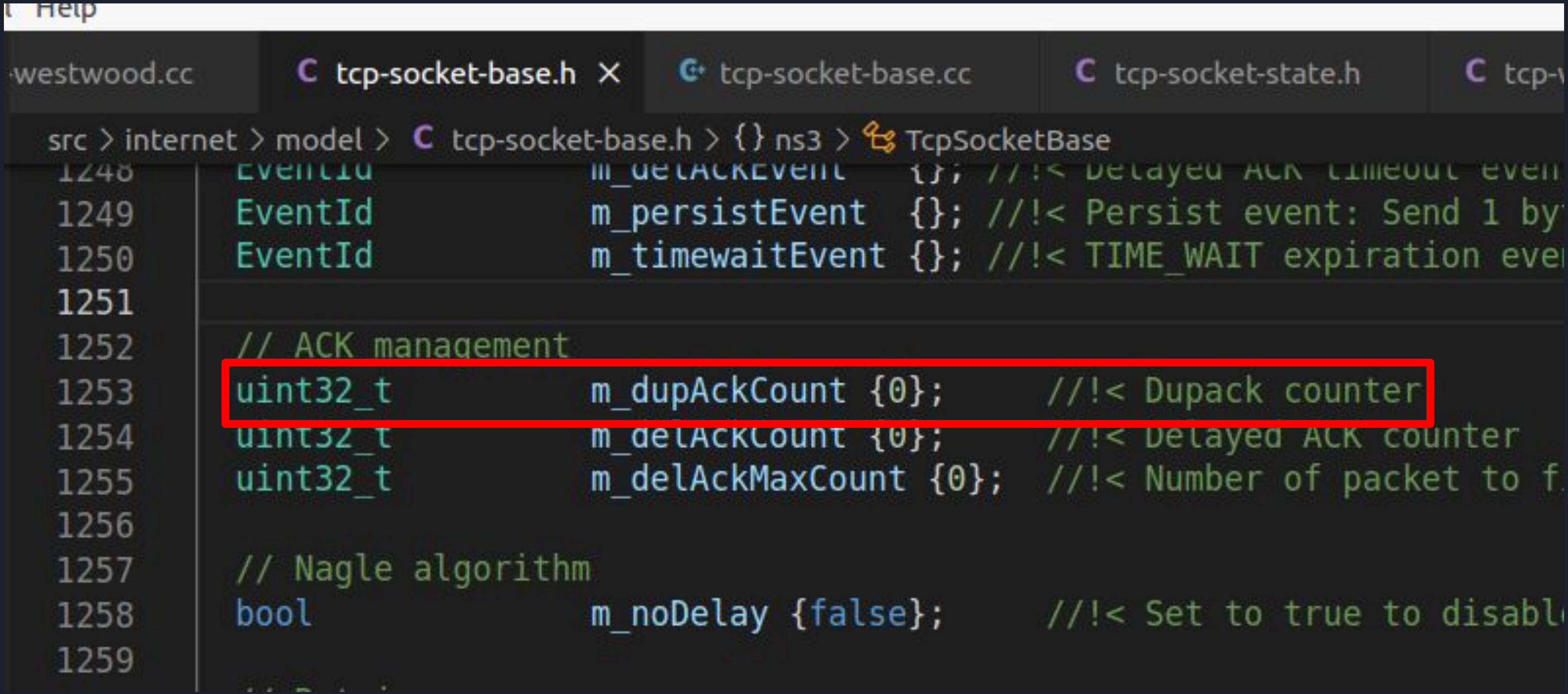
```
(1) Each time an ACK of a new data segment is received,
    If (congestion level=1||congestion level=2)//Think it is wireless packet loss, mild congestion
        Cwnd=Cwnd+1;
    If (Cwnd>Ssthresh)
        Cwnd=Cwnd+(1/Cwnd)*p;
(2) After receiving a duplicate ACK before timing out
    If (duplicate ACK=3&& congestion level=1)
        Fast retransmission;
        Quick recovery
    If (duplicate ACK=2&& (congestion level=3||congestion level=4))//Think it is a congestion packet
    Slow start or congestion avoidance;
      Cwnd=Cwnd*p;
      Ssthresh =(BWE*RTTmin)/seg_size;
    If (Cwnd>Ssthresh) then Cwnd=Ssthresh;
    If (duplicate ACK=3&& congestion level>2)
    Slow start or congestion avoidance;
      Cwnd=Cwnd*p;
      Ssthresh=(BWE*RTTmin)/seg_size;
    If (Cwnd>Ssthresh) then Cwnd=Ssthresh;
```

Help

estwood.cc | tcp-socket-base.cc | tcp-socket-state.h | tcp-westwood.h | tcp-congestion-ops.h

src > internet > model > tcp-socket-base.cc > {} ns3 > DupAck(uint32_t)

```cpp
    // (4.5) Proceed to step (C)
    // these steps are done after the ProcessAck function (SendPendingData)
}

void
TcpSocketBase::DupAck (uint32_t currentDelivered)
{
  NS_LOG_FUNCTION (this);
  // NOTE: We do not count the DupAcks received in CA_LOSS, because we
  // don't know if they are generated by a spurious retransmission or because
  // of a real packet loss. With SACK, it is easy to know, but we do not consider
  // dupacks. Without SACK, there are some euristics in the RFC 6582, but
  // for now, we do not implement it, leading to ignoring the dupacks.
  if (m_tcb->m_congState == TcpSocketState::CA_LOSS)
    {
      return;
    }

  // RFC 6675, Section 5, 3rd paragraph:
```


Help
westwood.cc
tcp-socket-base.h
tcp-socket-base.cc
tcp-socket-state.h
src > internet > model > tcp-socket-base.h > {} ns3 > TcpSocketBase
EventId m_persistEvent {}; //!< Persist event: Send 1 by
EventId m_timewaitEvent {}; //!< TIME_WAIT expiration eve
// ACK management
uint32_t m_dupAckCount {0}; //!< Dupack counter
uint32_t m_delAckCount {0}; //!< Delayed ACK counter
uint32_t m_delAckMaxCount {0}; //!< Number of packet to f
// Nagle algorithm
bool m_noDelay {false}; //!< Set to true to disabl