CSE 322 : Computer Networks Sessional

NS3 PROJECT : TCP-AR (Adaptive Reno)

1705044

Paper Topology

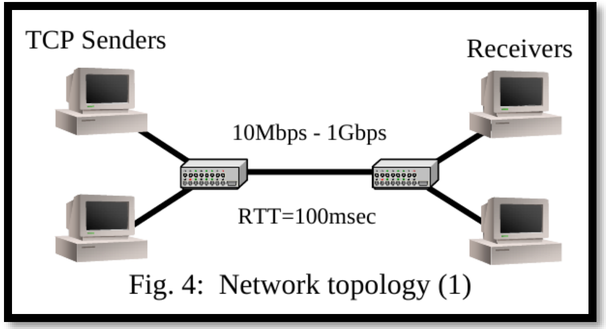
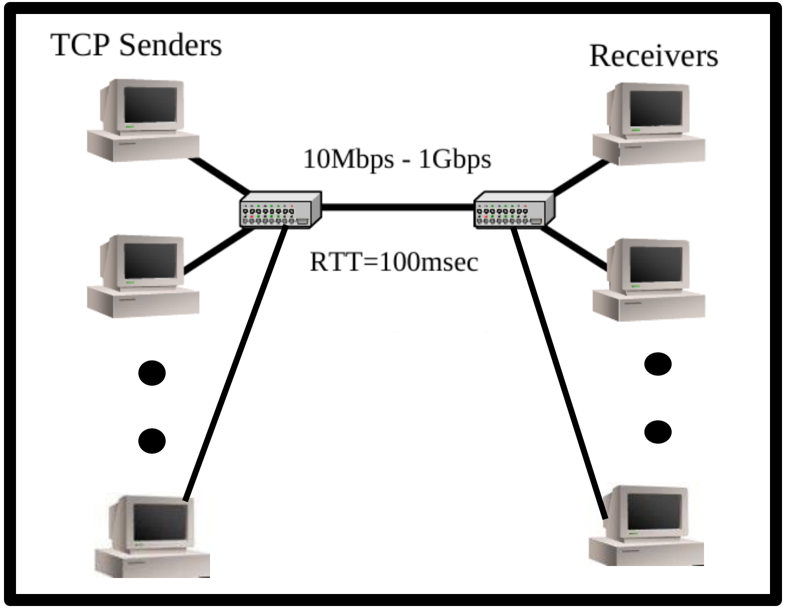


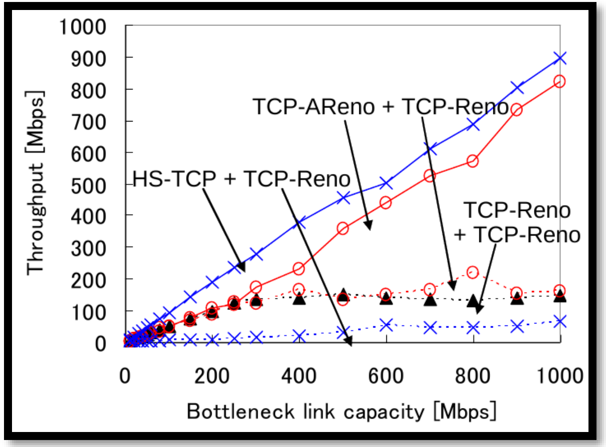
Fig. 4: Network topology (1)

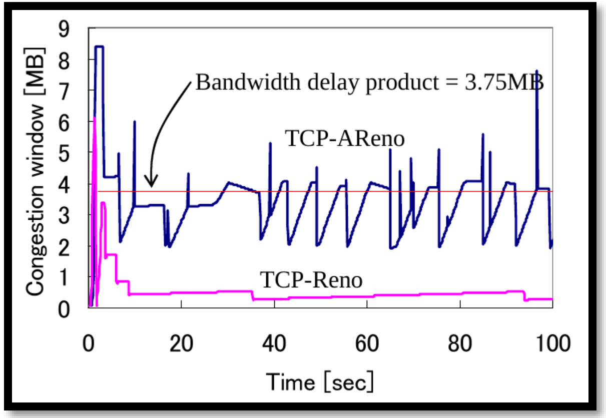


My Implementation

Paper Experiments

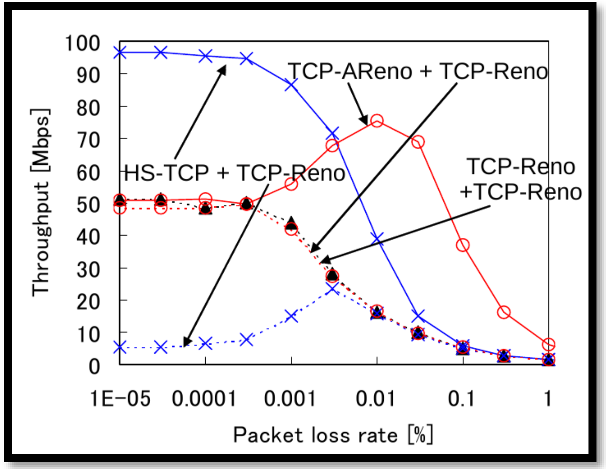
Throughput vs bottleneck link capacity



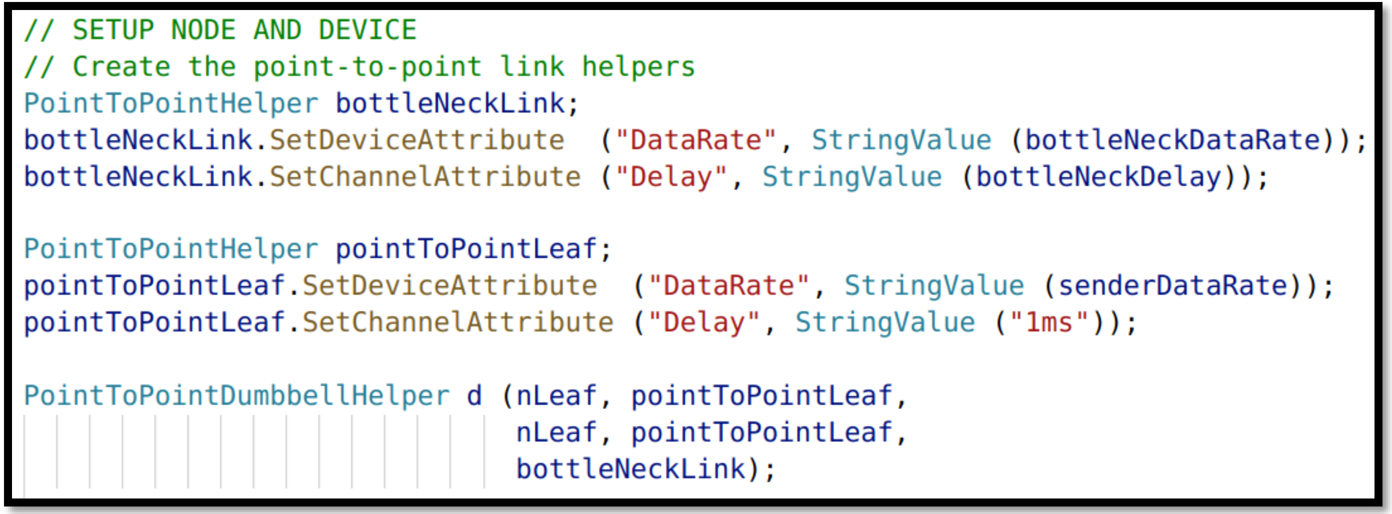


Congestion Window vs Time

Throughput vs packet loss rate



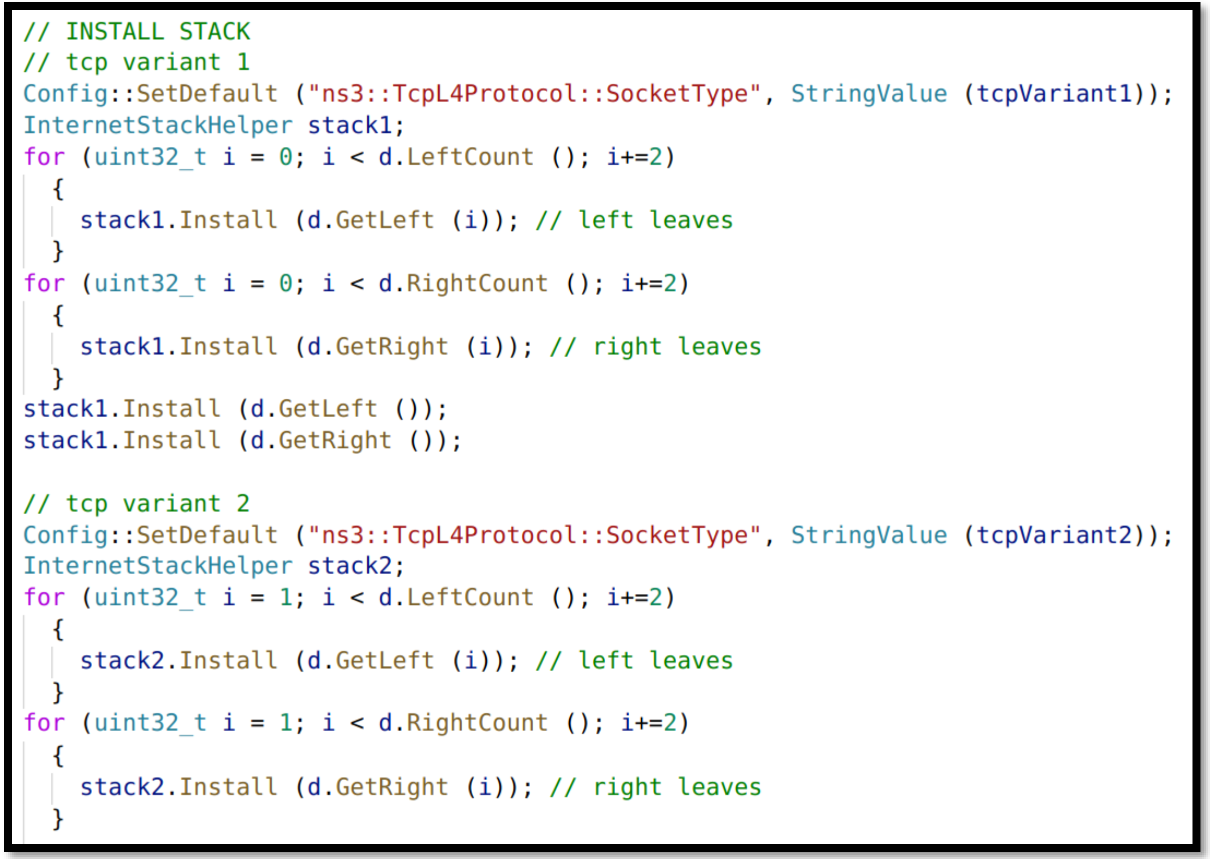
Building Topology



```
// SETUP NODE AND DEVICE
// Create the point-to-point link helpers
PointToPointHelper bottleNeckLink;
bottleNeckLink.SetDeviceAttribute  ("DataRate", StringValue (bottleNeckDataRate));
bottleNeckLink.SetChannelAttribute ("Delay", StringValue (bottleNeckDelay));

PointToPointHelper pointToPointLeaf;
pointToPointLeaf.SetDeviceAttribute  ("DataRate", StringValue (senderDataRate));
pointToPointLeaf.SetChannelAttribute ("Delay", StringValue ("1ms"));

PointToPointDumbbellHelper d (nLeaf, pointToPointLeaf,
                              nLeaf, pointToPointLeaf,
                              bottleNeckLink);
```

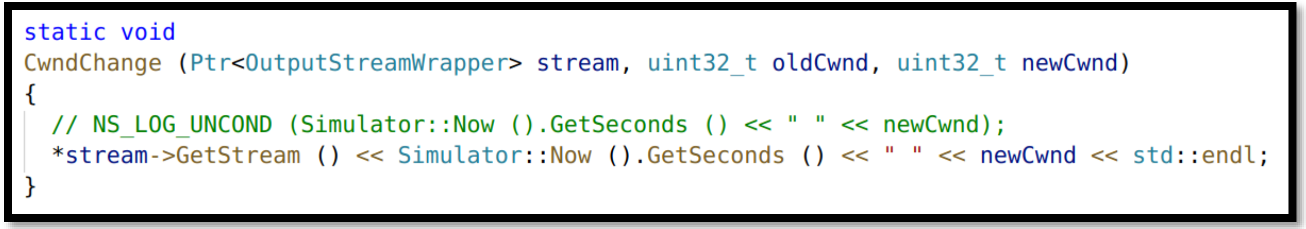
Building Topology

2 Different TCP variant for odd and even nodes

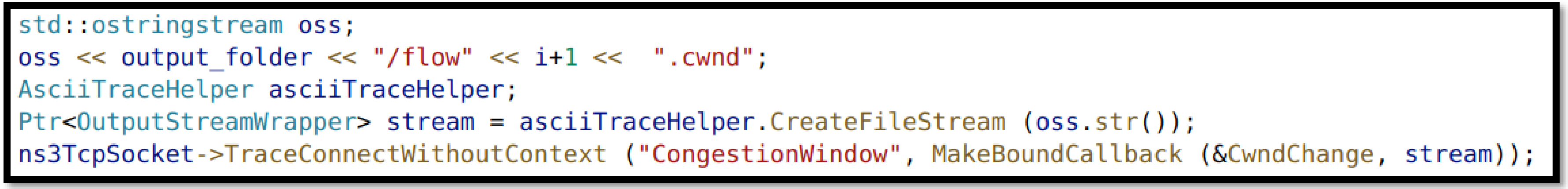


```
// INSTALL STACK
// tcp variant 1
Config::SetDefault ("ns3::TcpL4Protocol::SocketType", StringValue (tcpVariant1));
InternetStackHelper stack1;
for (uint32_t i = 0; i < d.LeftCount (); i+=2)
  {
    stack1.Install (d.GetLeft (i)); // left leaves
  }
for (uint32_t i = 0; i < d.RightCount (); i+=2)
  {
    stack1.Install (d.GetRight (i)); // right leaves
  }
stack1.Install (d.GetLeft ());
stack1.Install (d.GetRight ());

// tcp variant 2
Config::SetDefault ("ns3::TcpL4Protocol::SocketType", StringValue (tcpVariant2));
InternetStackHelper stack2;
for (uint32_t i = 1; i < d.LeftCount (); i+=2)
  {
    stack2.Install (d.GetLeft (i)); // left leaves
  }
for (uint32_t i = 1; i < d.RightCount (); i+=2)
  {
    stack2.Install (d.GetRight (i)); // right leaves
  }
```

Calculate Metrics



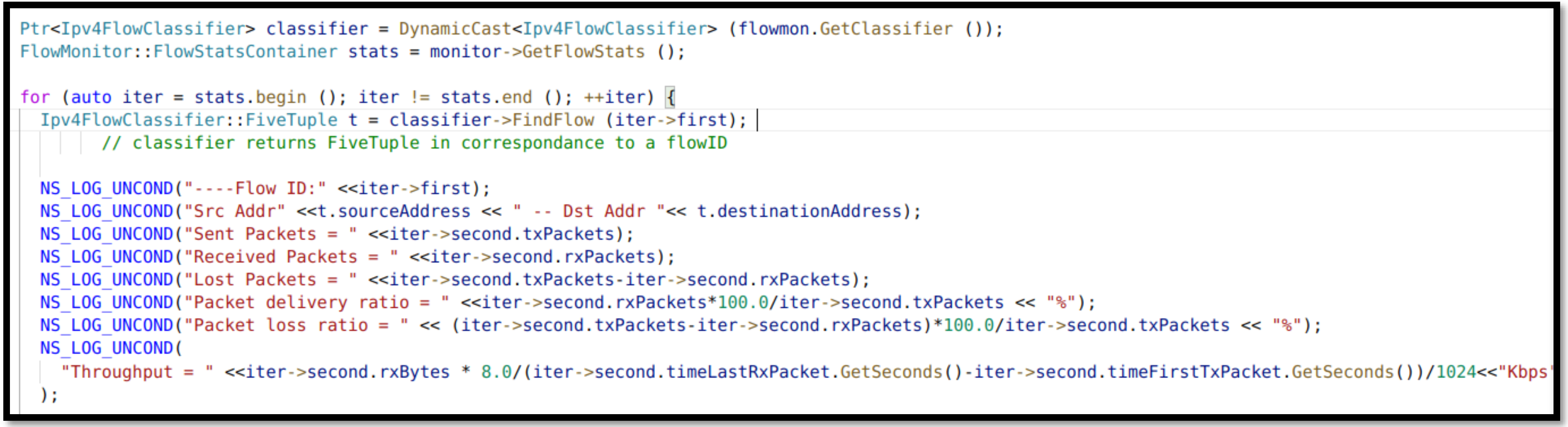
```
static void
CwndChange (Ptr<OutputStreamWrapper> stream, uint32_t oldCwnd, uint32_t newCwnd)
{
  // NS_LOG_UNCOND (Simulator::Now ().GetSeconds () << " " << newCwnd);
  *stream->GetStream () << Simulator::Now ().GetSeconds () << " " << newCwnd << std::endl;
}
```



```
std::ostringstream oss;
oss << output_folder << "/flow" << i+1 <<  ".cwnd";
AsciiTraceHelper asciiTraceHelper;
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream (oss.str());
ns3TcpSocket->TraceConnectWithoutContext ("CongestionWindow", MakeBoundCallback (&CwndChange, stream));
```

Calculate New Congestion Window for each flow

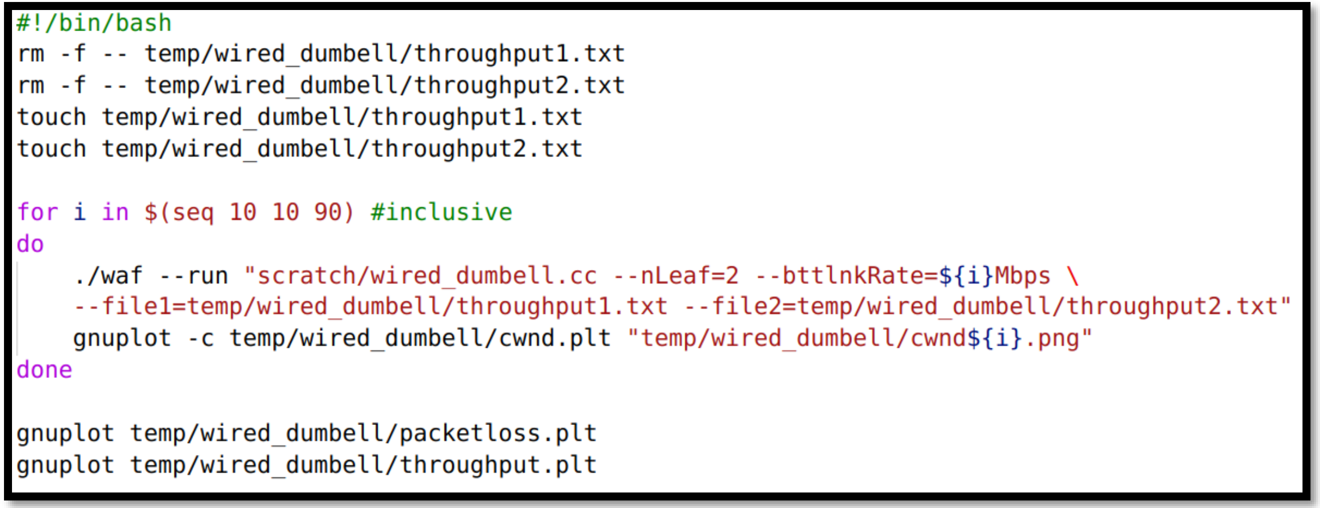
Calculate Metrics



```
Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier> (flowmon.GetClassifier ());
FlowMonitor::FlowStatsContainer stats = monitor->GetFlowStats ();

for (auto iter = stats.begin (); iter != stats.end (); ++iter) {
  Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow (iter->first);
        // classifier returns FiveTuple in correspondance to a flowID

  NS_LOG_UNCOND("----Flow ID:" <<iter->first);
  NS_LOG_UNCOND("Src Addr" <<t.sourceAddress << " -- Dst Addr "<< t.destinationAddress);
  NS_LOG_UNCOND("Sent Packets = " <<iter->second.txPackets);
  NS_LOG_UNCOND("Received Packets = " <<iter->second.rxPackets);
  NS_LOG_UNCOND("Lost Packets = " <<iter->second.txPackets-iter->second.rxPackets);
  NS_LOG_UNCOND("Packet delivery ratio = " <<iter->second.rxPackets*100.0/iter->second.txPackets << "%");
  NS_LOG_UNCOND("Packet loss ratio = " << (iter->second.txPackets-iter->second.rxPackets)*100.0/iter->second.txPackets << "%");
  NS_LOG_UNCOND(
    "Throughput = " <<iter->second.rxBytes * 8.0/(iter->second.timeLastRxPacket.GetSeconds()-iter->second.timeFirstTxPacket.GetSeconds())/1024<<"Kbps
  );
```

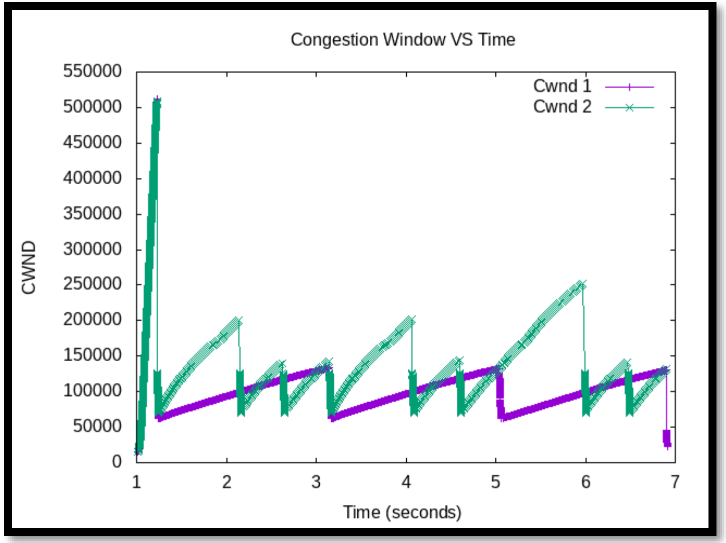
Calculate Throughput and package drop rate for each flow

Full Pipeline



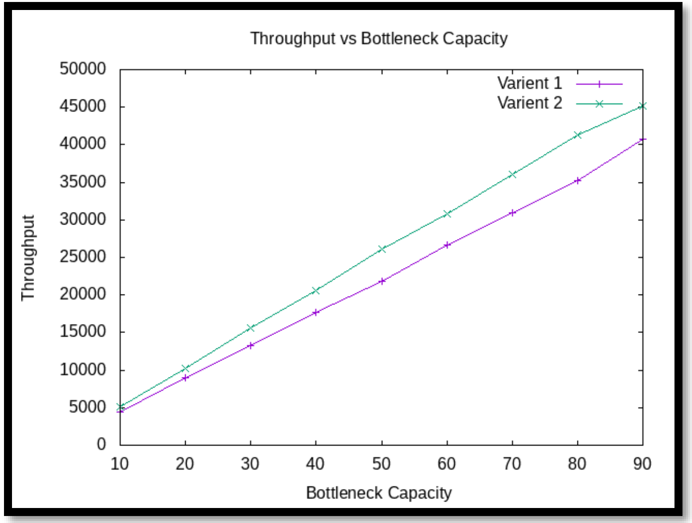
```
#!/bin/bash
rm -f -- temp/wired_dumbell/throughput1.txt
rm -f -- temp/wired_dumbell/throughput2.txt
touch temp/wired_dumbell/throughput1.txt
touch temp/wired_dumbell/throughput2.txt

for i in $(seq 10 10 90) #inclusive
do
    ./waf --run "scratch/wired_dumbell.cc --nLeaf=2 --bttlnkRate=${i}Mbps \
    --file1=temp/wired_dumbell/throughput1.txt --file2=temp/wired_dumbell/throughput2.txt"
    gnuplot -c temp/wired_dumbell/cwnd.plt "temp/wired_dumbell/cwnd${i}.png"
done

gnuplot temp/wired_dumbell/packetloss.plt
gnuplot temp/wired_dumbell/throughput.plt
```

Metric Graphs

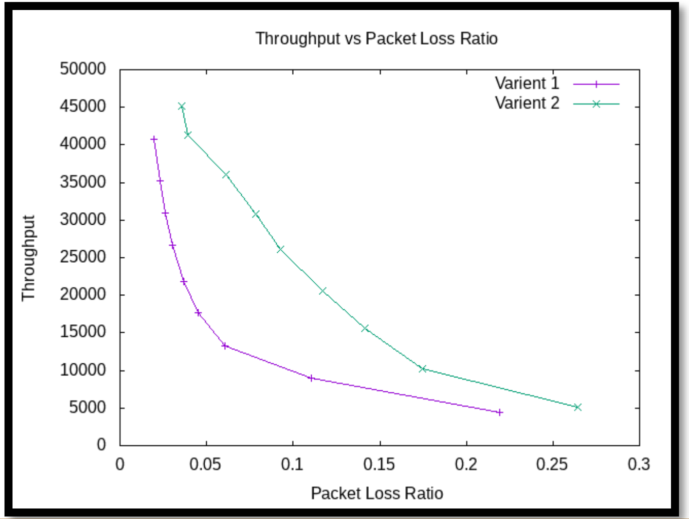


Congestion Window vs Time

Throughput vs bottleneck link capacity



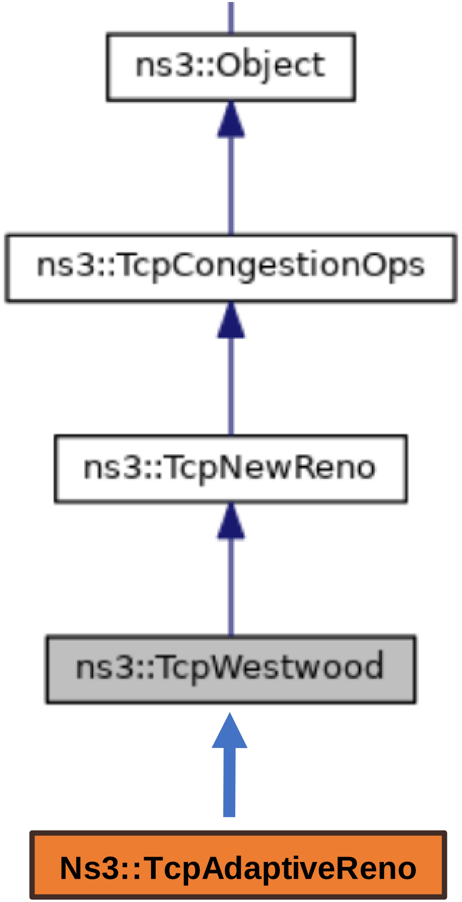
Throughput vs packet loss rate



TCP-AdaptiveReno

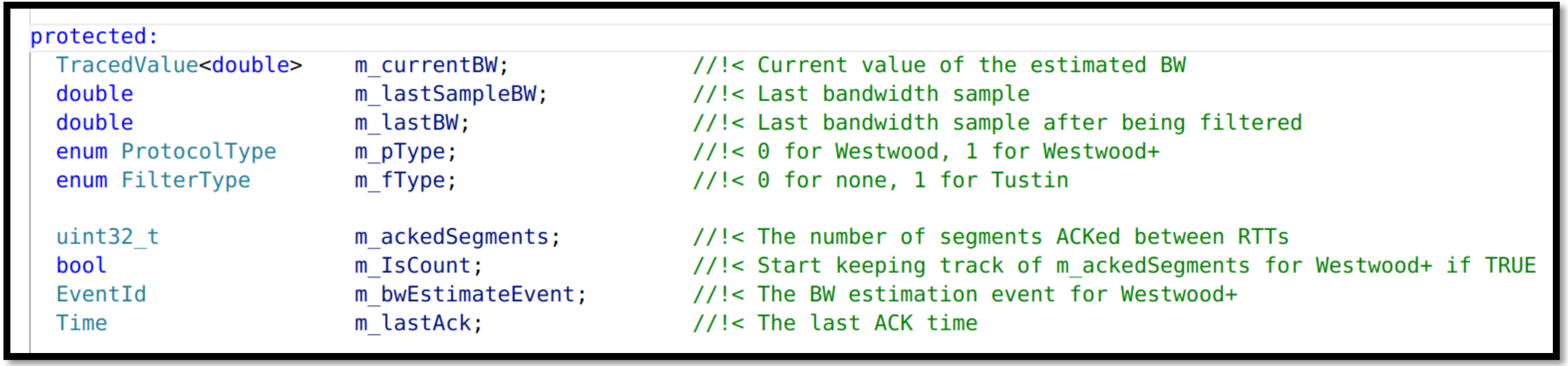
Implementation Questions

Class Hierarchy



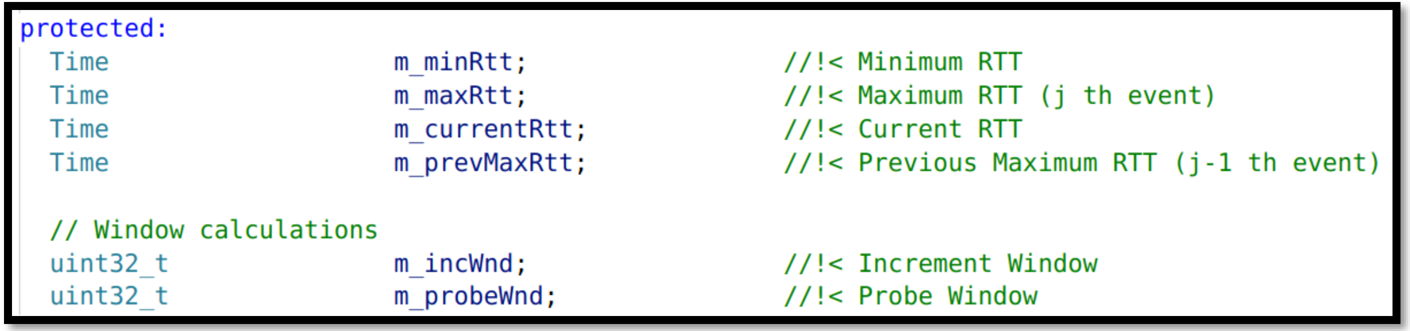
Variables

TCP Westwood



```
protected:
  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 ProtocolType      m_pType;                 //!< 0 for Westwood, 1 for Westwood+
  enum FilterType        m_fType;                 //!< 0 for none, 1 for Tustin

  uint32_t               m_ackedSegments;         //!< The number of segments ACKed between RTTs
  bool                   m_IsCount;               //!< Start keeping track of m_ackedSegments for Westwood+ if TRUE
  EventId                m_bwEstimateEvent;       //!< The BW estimation event for Westwood+
  Time                   m_lastAck;               //!< The last ACK time
```

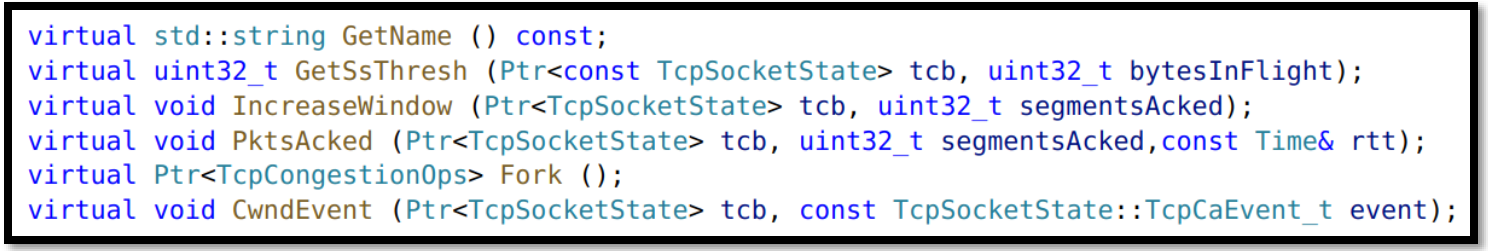
TCP AdaptiveReno



```
protected:
  Time                      m_minRtt;                 //!< Minimum RTT
  Time                      m_maxRtt;                 //!< Maximum RTT (j th event)
  Time                      m_currentRtt;             //!< Current RTT
  Time                      m_prevMaxRtt;             //!< Previous Maximum RTT (j-1 th event)

  // Window calculations
  uint32_t                  m_incWnd;                 //!< Increment Window
  uint32_t                  m_probeWnd;               //!< Probe Window
```

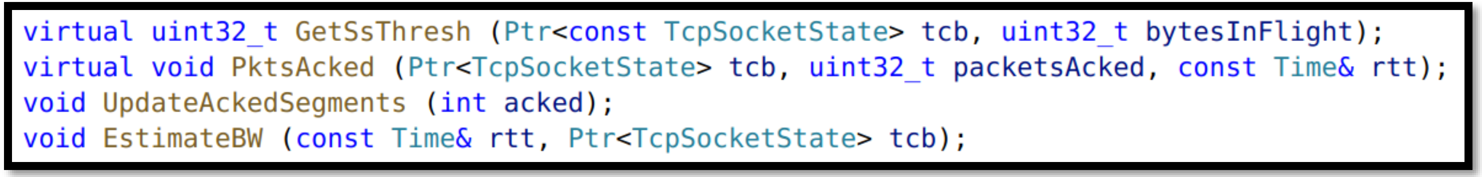
Functions

TCP NewReno



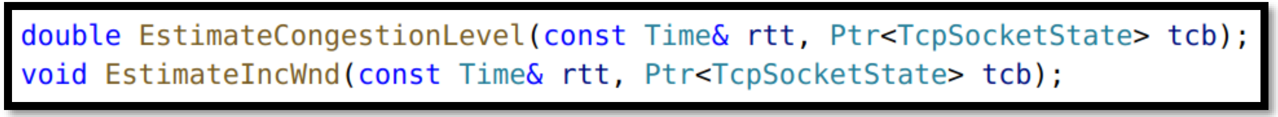
```
virtual std::string GetName () const;
virtual uint32_t GetSsThresh (Ptr<const TcpSocketState> tcb, uint32_t bytesInFlight);
virtual void IncreaseWindow (Ptr<TcpSocketState> tcb, uint32_t segmentsAcked);
virtual void PktsAcked (Ptr<TcpSocketState> tcb, uint32_t segmentsAcked,const Time& rtt);
virtual Ptr<TcpCongestionOps> Fork ();
virtual void CwndEvent (Ptr<TcpSocketState> tcb, const TcpSocketState::TcpCaEvent_t event);
```

TCP Westwood



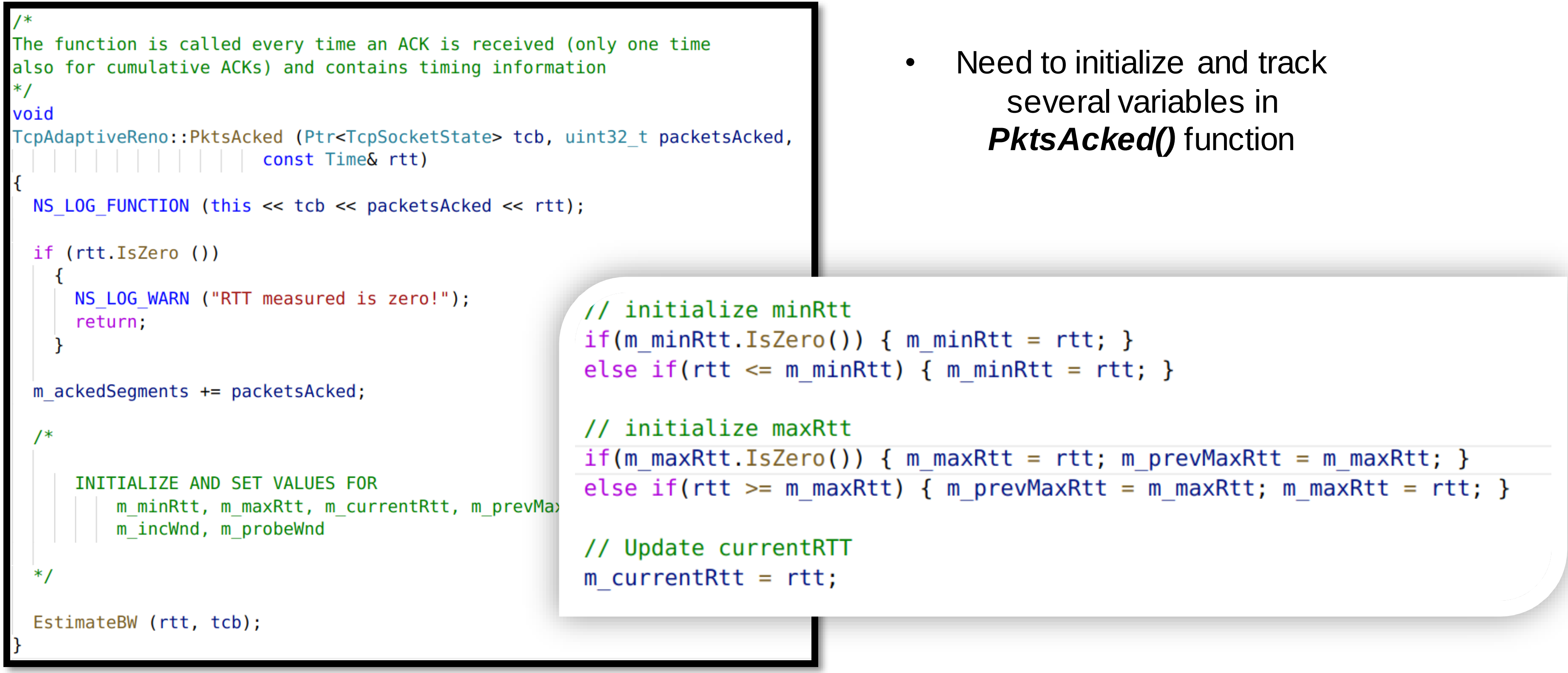
```
virtual uint32_t GetSsThresh (Ptr<const TcpSocketState> tcb, uint32_t bytesInFlight);
virtual void PktsAcked (Ptr<TcpSocketState> tcb, uint32_t packetsAcked, const Time& rtt);
void UpdateAckedSegments (int acked);
void EstimateBW (const Time& rtt, Ptr<TcpSocketState> tcb);
```

TCP AdaptiveReno



```
double EstimateCongestionLevel(const Time& rtt, Ptr<TcpSocketState> tcb);
void EstimateIncWnd(const Time& rtt, Ptr<TcpSocketState> tcb);
```

Modifications



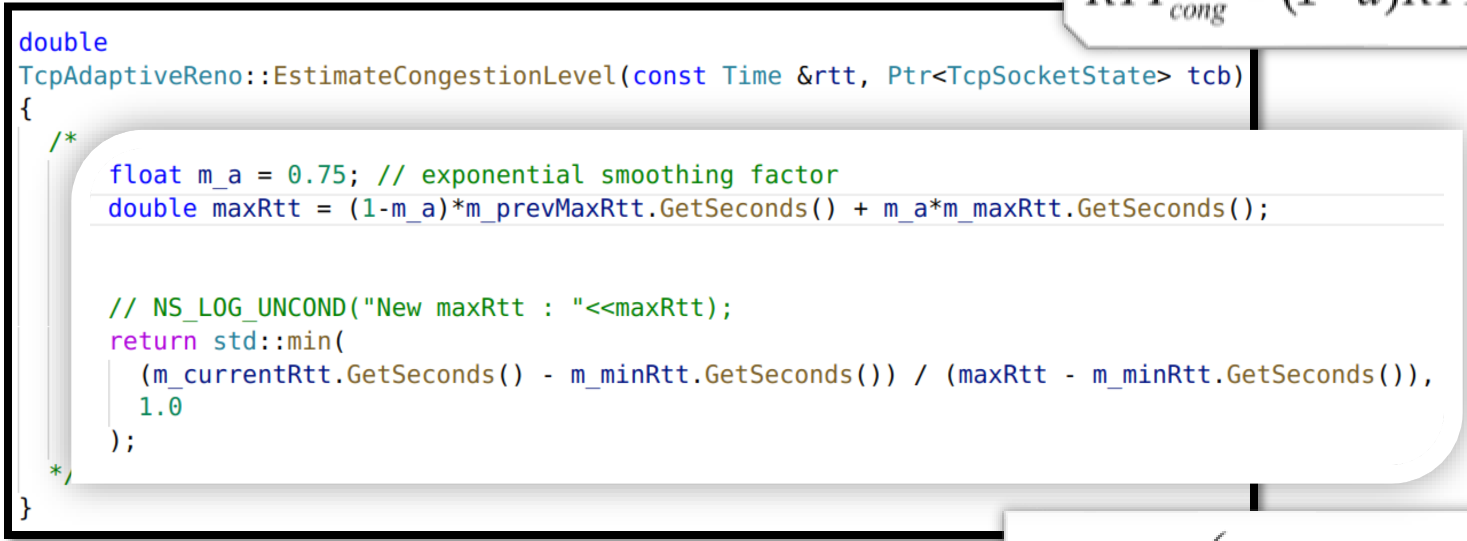
```
/*
The function is called every time an ACK is received (only one time
also for cumulative ACKs) and contains timing information
*/
void
TcpAdaptiveReno::PktsAcked (Ptr<TcpSocketState> tcb, uint32_t packetsAcked,
                            const Time& rtt)
{
  NS_LOG_FUNCTION (this << tcb << packetsAcked << rtt);

  if (rtt.IsZero ())
    {
      NS_LOG_WARN ("RTT measured is zero!");
      return;
    }

  m_ackedSegments += packetsAcked;

  /*

      INITIALIZE AND SET VALUES FOR
          m_minRtt, m_maxRtt, m_currentRtt, m_prevMax
          m_incWnd, m_probeWnd

  */

  EstimateBW (rtt, tcb);
}
```

```
// initialize minRtt
if(m_minRtt.IsZero()) { m_minRtt = rtt; }
else if(rtt <= m_minRtt) { m_minRtt = rtt; }

// initialize maxRtt
if(m_maxRtt.IsZero()) { m_maxRtt = rtt; m_prevMaxRtt = m_maxRtt; }
else if(rtt >= m_maxRtt) { m_prevMaxRtt = m_maxRtt; m_maxRtt = rtt; }

// Update currentRTT
m_currentRtt = rtt;
```

- Need to initialize and track several variables in ***PktsAcked()*** function

Modifications

$$RTT_{cong}^{j} = (1-a)RTT_{cong}^{j-1} + aRTT^{j}$$



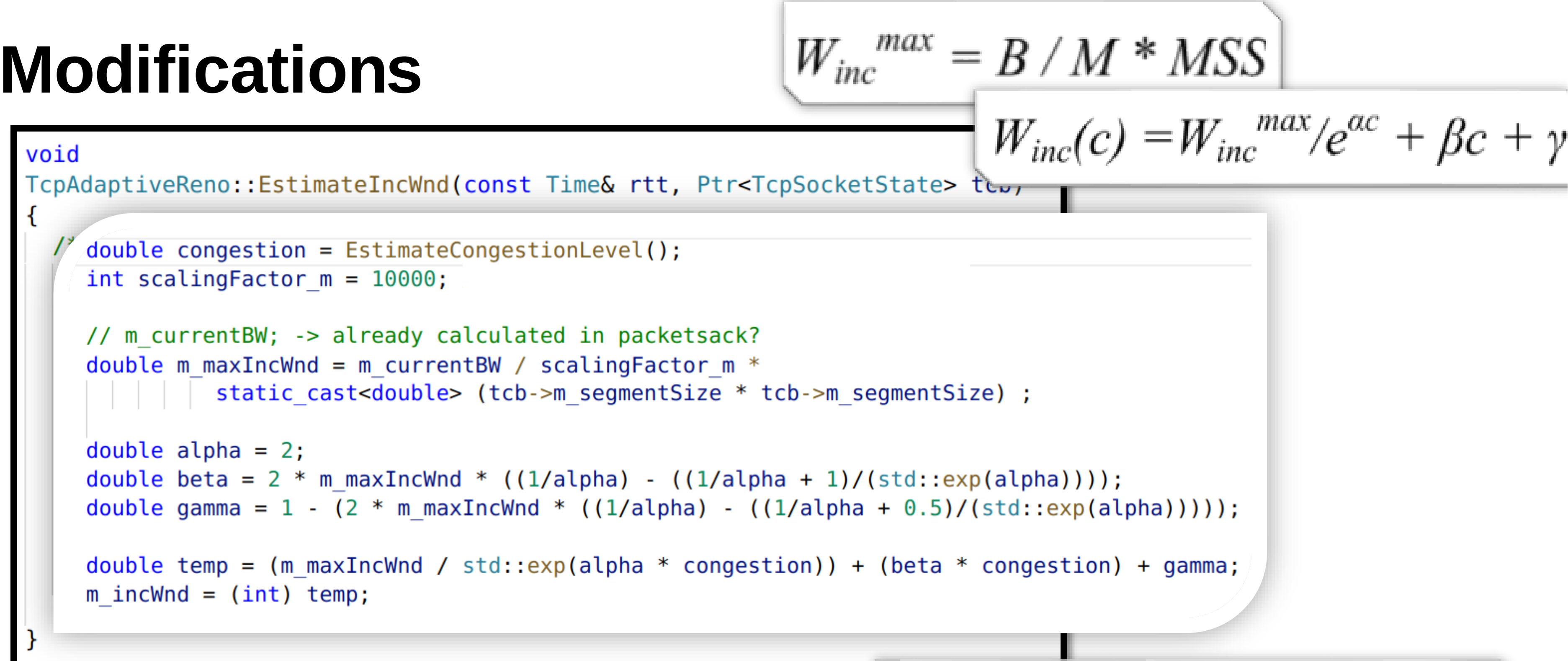
```
double
TcpAdaptiveReno::EstimateCongestionLevel(const Time &rtt, Ptr<TcpSocketState> tcb)
{
  /*
  */
}
```

```
float m_a = 0.75; // exponential smoothing factor
double maxRtt = (1-m_a)*m_prevMaxRtt.GetSeconds() + m_a*m_maxRtt.GetSeconds();


// NS_LOG_UNCOND("New maxRtt : "<<maxRtt);
return std::min(
  (m_currentRtt.GetSeconds() - m_minRtt.GetSeconds()) / (maxRtt - m_minRtt.GetSeconds()),
  1.0
);
```

$$c = \min\left(\frac{RTT - RTT_{\min}}{RTT_{cong} - RTT_{\min}}, \quad 1\right)$$

- Retrieve the current congestion level in ***EstimateCongestionFunction()*** function

Modifications



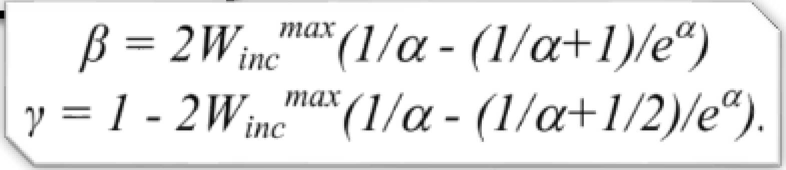
$$W_{inc}^{max} = B / M * MSS$$

$$W_{inc}(c) = W_{inc}^{max} / e^{\alpha c} + \beta c + \gamma$$

```
void
TcpAdaptiveReno::EstimateIncWnd(const Time& rtt, Ptr<TcpSocketState> tcb)
{
  /*
}
```

```
double congestion = EstimateCongestionLevel();
int scalingFactor_m = 10000;

// m_currentBW; -> already calculated in packetsack?
double m_maxIncWnd = m_currentBW / scalingFactor_m *
          static_cast<double> (tcb->m_segmentSize * tcb->m_segmentSize) ;

double alpha = 2;
double beta = 2 * m_maxIncWnd * ((1/alpha) - ((1/alpha + 1)/(std::exp(alpha))));
double gamma = 1 - (2 * m_maxIncWnd * ((1/alpha) - ((1/alpha + 0.5)/(std::exp(alpha)))));

double temp = (m_maxIncWnd / std::exp(alpha * congestion)) + (beta * congestion) + gamma;
m_incWnd = (int) temp;
```

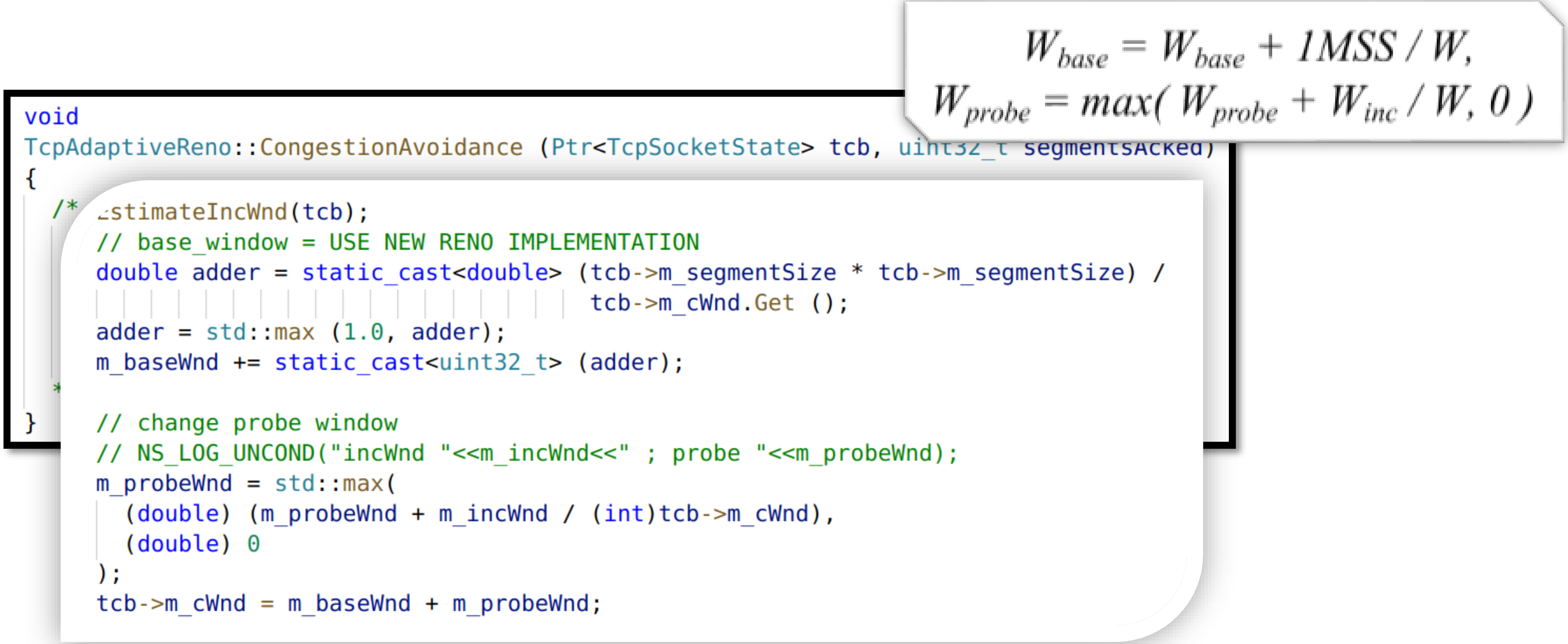
- Retrieve the current increment window size in ***EstimateIncWindow()*** function



$$\beta = 2W_{inc}^{max}(1/\alpha - (1/\alpha+1)/e^{\alpha})$$

$$\gamma = 1 - 2W_{inc}^{max}(1/\alpha - (1/\alpha+1/2)/e^{\alpha}).$$

Modifications



$$W_{base} = W_{base} + 1MSS / W,$$

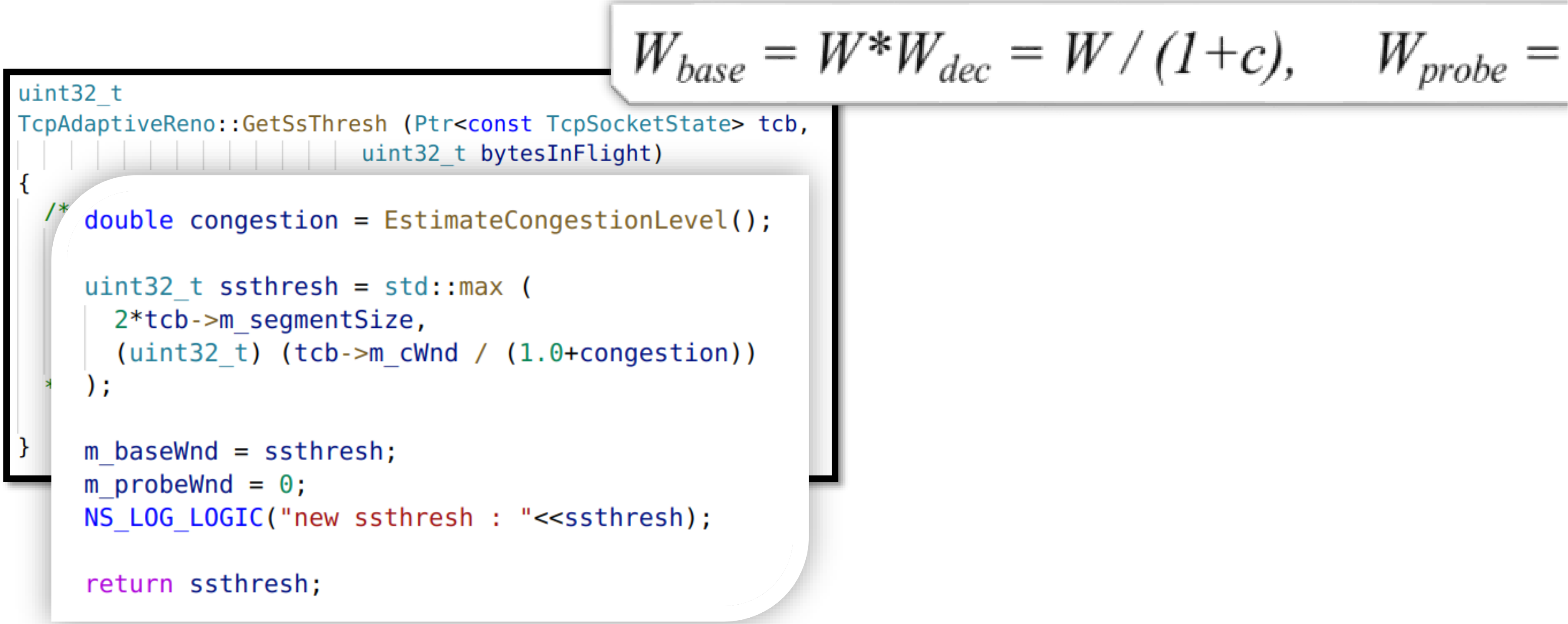
$$W_{probe} = max(\ W_{probe} + W_{inc} / W,\ 0\ )$$

```
void
TcpAdaptiveReno::CongestionAvoidance (Ptr<TcpSocketState> tcb, uint32_t segmentsAcked)
{
  /*

  *
}
```

```
EstimateIncWnd(tcb);
// base_window = USE NEW RENO IMPLEMENTATION
double adder = static_cast<double> (tcb->m_segmentSize * tcb->m_segmentSize) /
                                   tcb->m_cWnd.Get ();
adder = std::max (1.0, adder);
m_baseWnd += static_cast<uint32_t> (adder);

// change probe window
// NS_LOG_UNCOND("incWnd "<<m_incWnd<<" ; probe "<<m_probeWnd);
m_probeWnd = std::max(
  (double) (m_probeWnd + m_incWnd / (int)tcb->m_cWnd),
  (double) 0
);
tcb->m_cWnd = m_baseWnd + m_probeWnd;
```

- Calculate the current window size for the Congestion Avoidance Phase in ***CongestionAvoidance()*** function

Modifications



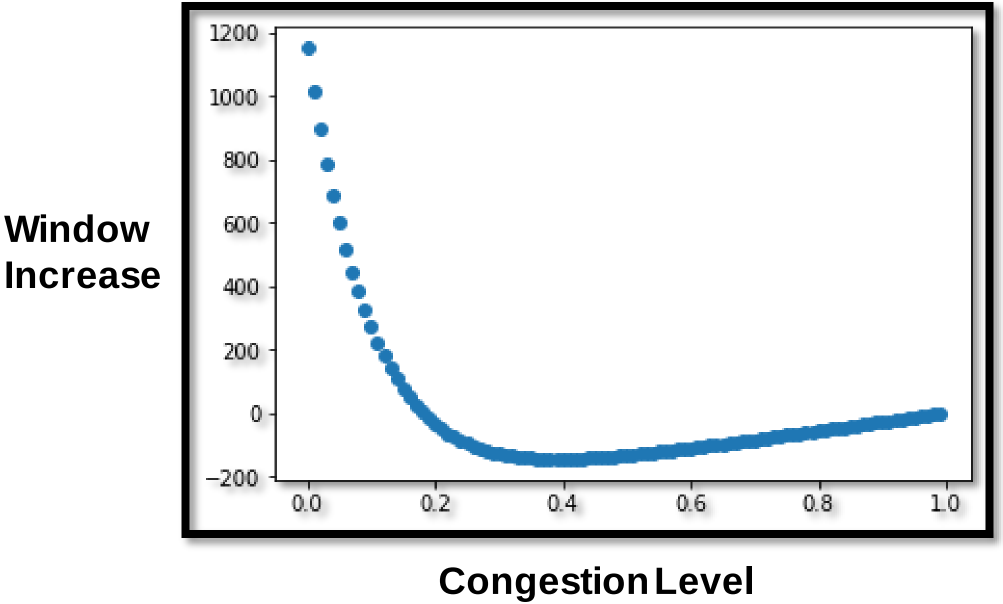
$$W_{base} = W*W_{dec} = W / (1+c), \qquad W_{probe} =$$

```
uint32_t
TcpAdaptiveReno::GetSsThresh (Ptr<const TcpSocketState> tcb,
                              uint32_t bytesInFlight)
{
  /*

  *
}
```

```
double congestion = EstimateCongestionLevel();

uint32_t ssthresh = std::max (
  2*tcb->m_segmentSize,
  (uint32_t) (tcb->m_cWnd / (1.0+congestion))
);

m_baseWnd = ssthresh;
m_probeWnd = 0;
NS_LOG_LOGIC("new ssthresh : "<<ssthresh);

return ssthresh;
```

- Calculate the Slow Start Threshold in the ***GetSsThresh()*** function

Query



- W_inc is never positive as congestion level is always > 0.4 in congestion avoidance phase

Reference

- H. Shimonishi and T. Murase, “**Improving efficiency friendliness tradeoffs of TCP congestion control algorithm**,” in Proc. IEEE GLOBECOM, 2005.

- Multiple NS3 Tutorials and documentations.

Paper Link

Thank you

Any Questions?