APPENDIXES
APPENDIX A

TCP NEWRENO HEADER

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2010 Adrian Sai-wah Tam
 *
 * This program is free software; you can redistribute it and/or
modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-
1307  USA
 *
 * Author: Adrian Sai-wah Tam <adrian.sw.tam@gmail.com>
 */

#ifndef TCP_NEWRENO_H
#define TCP_NEWRENO_H

#include "tcp-socket-base.h"

namespace ns3 {

/**
 * \ingroup socket
 * \ingroup tcp
 *
 * \brief An implementation of a stream socket using TCP.
 *
 * This class contains the NewReno implementation of TCP, as of
\RFC{2582}.
 */
class TcpNewReno : public TcpSocketBase
{
public:
  /**
   * \brief Get the type ID.
   * \return the object TypeId
   */
  static TypeId GetTypeId (void);
  /**
   * Create an unbound tcp socket.
   */
  TcpNewReno (void);
  /**
   * \brief Copy constructor
```

```
     * \param sock the object to copy
     */
    TcpNewReno (const TcpNewReno& sock);
    virtual ~TcpNewReno (void);

    // From TcpSocketBase
    virtual int Connect (const Address &address);
    virtual int Listen (void);

protected:
    virtual uint32_t Window (void); // Return the max possible number
of unacked bytes
    virtual Ptr<TcpSocketBase> Fork (void); // Call
CopyObject<TcpNewReno> to clone me
    virtual void NewAck (SequenceNumber32 const& seq); // Inc cwnd and
call NewAck() of parent
    virtual void DupAck (const TcpHeader& t, uint32_t count);  //
Halving cwnd and reset nextTxSequence
    virtual void Retransmit (void); // Exit fast recovery upon
retransmit timeout

    // Implementing ns3::TcpSocket -- Attribute get/set
    virtual void     SetSegSize (uint32_t size);
    virtual void     SetInitialSSThresh (uint32_t threshold);
    virtual uint32_t GetInitialSSThresh (void) const;
    virtual void     SetInitialCwnd (uint32_t cwnd);
    virtual uint32_t GetInitialCwnd (void) const;
private:
    /**
     * \brief Set the congestion window when connection starts
     */
    void InitializeCwnd (void);

protected:
    TracedValue<uint32_t>  m_cWnd;         //!< Congestion window
    TracedValue<uint32_t>  m_ssThresh;     //!< Slow Start Threshold
    uint32_t               m_initialCWnd;  //!< Initial cWnd value
    uint32_t               m_initialSsThresh;  //!< Initial Slow Start
Threshold value
    SequenceNumber32       m_recover;      //!< Previous highest Tx
seqnum for fast recovery
    uint32_t               m_retxThresh;   //!< Fast Retransmit
threshold
    bool                   m_inFastRec;    //!< currently in fast
recovery
    bool                   m_limitedTx;    //!< perform limited
transmit
};

} // namespace ns3

#endif /* TCP_NEWRENO_H */
```

APPENDIX B

TCP NEWRENO CONGESTION CONTROL SOURCE CODE

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2010 Adrian Sai-wah Tam
 *
 * This program is free software; you can redistribute it and/or
modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-
1307  USA
 *
 * Author: Adrian Sai-wah Tam <adrian.sw.tam@gmail.com>
 */

#define NS_LOG_APPEND_CONTEXT \
  if (m_node) { std::clog << Simulator::Now ().GetSeconds () << "
[node " << m_node->GetId () << "] "; }

#include "tcp-newreno.h"
#include "ns3/log.h"
#include "ns3/trace-source-accessor.h"
#include "ns3/simulator.h"
#include "ns3/abort.h"
#include "ns3/node.h"

NS_LOG_COMPONENT_DEFINE ("TcpNewReno");

namespace ns3 {

NS_OBJECT_ENSURE_REGISTERED (TcpNewReno);

TypeId
TcpNewReno::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::TcpNewReno")
    .SetParent<TcpSocketBase> ()
    .AddConstructor<TcpNewReno> ()
    .AddAttribute ("ReTxThreshold", "Threshold for fast retransmit",
                   UintegerValue (3),
                   MakeUintegerAccessor (&TcpNewReno::m_retxThresh),
                   MakeUintegerChecker<uint32_t> ())
    .AddAttribute ("LimitedTransmit", "Enable limited transmit",
                   BooleanValue (false),
                   MakeBooleanAccessor (&TcpNewReno::m_limitedTx),
                   MakeBooleanChecker ())
    .AddTraceSource ("CongestionWindow",
                     "The TCP connection's congestion window",
```

```
                        MakeTraceSourceAccessor (&TcpNewReno::m_cWnd))
    .AddTraceSource ("SlowStartThreshold",
                        "TCP slow start threshold (bytes)",
                        MakeTraceSourceAccessor
(&TcpNewReno::m_ssThresh))
  ;
  return tid;
}

TcpNewReno::TcpNewReno (void)
  : m_retxThresh (3), // mute valgrind, actual value set by the
attribute system
    m_inFastRec (false),
    m_limitedTx (false) // mute valgrind, actual value set by the
attribute system
{
  NS_LOG_FUNCTION (this);
}

TcpNewReno::TcpNewReno (const TcpNewReno& sock)
  : TcpSocketBase (sock),
    m_cWnd (sock.m_cWnd),
    m_ssThresh (sock.m_ssThresh),
    m_initialCWnd (sock.m_initialCWnd),
    m_initialSsThresh (sock.m_initialSsThresh),
    m_retxThresh (sock.m_retxThresh),
    m_inFastRec (false),
    m_limitedTx (sock.m_limitedTx)
{
  NS_LOG_FUNCTION (this);
  NS_LOG_LOGIC ("Invoked the copy constructor");
}

TcpNewReno::~TcpNewReno (void)
{
}

/* We initialize m_cWnd from this function, after attributes
initialized */
int
TcpNewReno::Listen (void)
{
  NS_LOG_FUNCTION (this);
  InitializeCwnd ();
  return TcpSocketBase::Listen ();
}

/* We initialize m_cWnd from this function, after attributes
initialized */
int
TcpNewReno::Connect (const Address & address)
{
  NS_LOG_FUNCTION (this << address);
  InitializeCwnd ();
  return TcpSocketBase::Connect (address);
}

/* Limit the size of in-flight data by cwnd and receiver's rxwin */
uint32_t
TcpNewReno::Window (void)
```

```
{
  NS_LOG_FUNCTION (this);
  return std::min (m_rWnd.Get (), m_cWnd.Get ());
}

Ptr<TcpSocketBase>
TcpNewReno::Fork (void)
{
  return CopyObject<TcpNewReno> (this);
}

/* New ACK (up to seqnum seq) received. Increase cwnd and call
TcpSocketBase::NewAck() */
void
TcpNewReno::NewAck (const SequenceNumber32& seq)
{
  NS_LOG_FUNCTION (this << seq);
  NS_LOG_LOGIC ("TcpNewReno receieved ACK for seq " << seq <<
                " cwnd " << m_cWnd <<
                " ssthresh " << m_ssThresh);

  // Check for exit condition of fast recovery
  if (m_inFastRec && seq < m_recover)
    { // Partial ACK, partial window deflation (RFC2582 sec.3 bullet
#5 paragraph 3)
      m_cWnd += m_segmentSize - (seq - m_txBuffer.HeadSequence ());
      NS_LOG_INFO ("Partial ACK in fast recovery: cwnd set to " <<
m_cWnd);
      m_txBuffer.DiscardUpTo(seq);  //Bug 1850: retransmit before
newack
      DoRetransmit (); // Assume the next seq is lost. Retransmit
lost packet
      TcpSocketBase::NewAck (seq); // update m_nextTxSequence and
send new data if allowed by window
      return;
    }
  else if (m_inFastRec && seq >= m_recover)
    { // Full ACK (RFC2582 sec.3 bullet #5 paragraph 2, option 1)
      m_cWnd = std::min (m_ssThresh.Get (), BytesInFlight () +
m_segmentSize);
      m_inFastRec = false;
      NS_LOG_INFO ("Received full ACK. Leaving fast recovery with
cwnd set to " << m_cWnd);
    }

  // Increase of cwnd based on current phase (slow start or
congestion avoidance)
  if (m_cWnd < m_ssThresh)
    { // Slow start mode, add one segSize to cWnd. Default m_ssThresh
is 65535. (RFC2001, sec.1)
      m_cWnd += m_segmentSize;
      NS_LOG_INFO ("In SlowStart, updated to cwnd " << m_cWnd << "
ssthresh " << m_ssThresh);
    }
  else
    { // Congestion avoidance mode, increase by
(segSize*segSize)/cwnd. (RFC2581, sec.3.1)
      // To increase cwnd for one segSize per RTT, it should be
(ackBytes*segSize)/cwnd
```

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

  // Complete newAck processing
  TcpSocketBase::NewAck (seq);
}

/* Cut cwnd and enter fast recovery mode upon triple dupack */
void
TcpNewReno::DupAck (const TcpHeader& t, uint32_t count)
{
  NS_LOG_FUNCTION (this << count);
  if (count == m_retxThresh && !m_inFastRec)
    { // triple duplicate ack triggers fast retransmit (RFC2582 sec.3
bullet #1)
      m_ssThresh = std::max (2 * m_segmentSize, BytesInFlight () /
2);
      m_cWnd = m_ssThresh + 3 * m_segmentSize;
      m_recover = m_highTxMark;
      m_inFastRec = true;
      NS_LOG_INFO ("Triple dupack. Enter fast recovery mode. Reset
cwnd to " << m_cWnd <<
                   ", ssthresh to " << m_ssThresh << " at fast
recovery seqnum " << m_recover);
      DoRetransmit ();
    }
  else if (m_inFastRec)
    { // Increase cwnd for every additional dupack (RFC2582, sec.3
bullet #3)
      m_cWnd += m_segmentSize;
      NS_LOG_INFO ("Dupack in fast recovery mode. Increase cwnd to "
<< m_cWnd);
      SendPendingData (m_connected);
    }
  else if (!m_inFastRec && m_limitedTx && m_txBuffer.SizeFromSequence
(m_nextTxSequence) > 0)
    { // RFC3042 Limited transmit: Send a new packet for each
duplicated ACK before fast retransmit
      NS_LOG_INFO ("Limited transmit");
      uint32_t sz = SendDataPacket (m_nextTxSequence, m_segmentSize,
true);
      m_nextTxSequence += sz;                    // Advance next tx
sequence
    };
}

/* Retransmit timeout */
void
TcpNewReno::Retransmit (void)
{
  NS_LOG_FUNCTION (this);
  NS_LOG_LOGIC (this << " ReTxTimeout Expired at time " <<
Simulator::Now ().GetSeconds ());
  m_inFastRec = false;
```

```
  // If erroneous timeout in closed/timed-wait state, just return
  if (m_state == CLOSED || m_state == TIME_WAIT) return;
  // If all data are received (non-closing socket and nothing to
send), just return
  if (m_state <= ESTABLISHED && m_txBuffer.HeadSequence () >=
m_highTxMark) return;

  // According to RFC2581 sec.3.1, upon RTO, ssthresh is set to half
of flight
  // size and cwnd is set to 1*MSS, then the lost packet is
retransmitted and
  // TCP back to slow start
  m_ssThresh = std::max (2 * m_segmentSize, BytesInFlight () / 2);
  m_cWnd = m_segmentSize;
  m_nextTxSequence = m_txBuffer.HeadSequence (); // Restart from
highest Ack
  NS_LOG_INFO ("RTO. Reset cwnd to " << m_cWnd <<
               ", ssthresh to " << m_ssThresh << ", restart from
seqnum " << m_nextTxSequence);
  m_rtt->IncreaseMultiplier ();             // Double the next RTO
  DoRetransmit ();                          // Retransmit the packet
}

void
TcpNewReno::SetSegSize (uint32_t size)
{
  NS_ABORT_MSG_UNLESS (m_state == CLOSED, "TcpNewReno::SetSegSize()
cannot change segment size after connection started.");
  m_segmentSize = size;
}

void
TcpNewReno::SetInitialSSThresh (uint32_t threshold)
{
  NS_ABORT_MSG_UNLESS (m_state == CLOSED, "TcpNewReno::SetSSThresh()
cannot change initial ssThresh after connection started.");
  m_initialSsThresh = threshold;
}

uint32_t
TcpNewReno::GetInitialSSThresh (void) const
{
  return m_initialSsThresh;
}

void
TcpNewReno::SetInitialCwnd (uint32_t cwnd)
{
  NS_ABORT_MSG_UNLESS (m_state == CLOSED,
"TcpNewReno::SetInitialCwnd() cannot change initial cwnd after
connection started.");
  m_initialCWnd = cwnd;
}

uint32_t
TcpNewReno::GetInitialCwnd (void) const
{
  return m_initialCWnd;
}
```

```
void
TcpNewReno::InitializeCwnd (void)
{
  /*
   * Initialize congestion window, default to 1 MSS (RFC2001, sec.1)
and must
   * not be larger than 2 MSS (RFC2581, sec.3.1). Both m_initiaCWnd
and
   * m_segmentSize are set by the attribute system in ns3::TcpSocket.
   */
  m_cWnd = m_initialCWnd * m_segmentSize;
  m_ssThresh = m_initialSsThresh;
}

} // namespace ns3
```

APPENDIX C

TCPW HEADER

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2013 ResiliNets, ITTC, University of Kansas
 *
 * This program is free software; you can redistribute it and/or
modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-
1307  USA
 *
 * Authors: Siddharth Gangadhar <siddharth@ittc.ku.edu>, Truc Anh N.
Nguyen <annguyen@ittc.ku.edu>,
 * and Greeshma Umapathi
 *
 * James P.G. Sterbenz <jpgs@ittc.ku.edu>, director
 * ResiliNets Research Group  http://wiki.ittc.ku.edu/resilinets
 * Information and Telecommunication Technology Center (ITTC)
 * and Department of Electrical Engineering and Computer Science
 * The University of Kansas Lawrence, KS USA.
 *
 * Work supported in part by NSF FIND (Future Internet Design)
Program
 * under grant CNS-0626918 (Postmodern Internet Architecture),
 * NSF grant CNS-1050226 (Multilayer Network Resilience Analysis and
Experimentation on GENI),
 * US Department of Defense (DoD), and ITTC at The University of
Kansas.
 */

#ifndef TCP_WESTWOOD_H
#define TCP_WESTWOOD_H

#include "tcp-socket-base.h"
#include "ns3/packet.h"

namespace ns3 {

/**
 * \ingroup socket
 * \ingroup tcp
 *
 * \brief An implementation of a stream socket using TCP.
 *
 * This class contains the implementation of TCP Westwood and
Westwood+.
 *
```

```
 * Westwood and Westwood+ employ the AIAD (Additive Increase/Adaptive
Decrease)
 * congestion control paradigm. When a congestion episode happens,
 * instead of halving the cwnd, these protocols try to estimate the
network's
 * bandwidth and use the estimated value to adjust the cwnd.
 * While Westwood performs the bandwidth sampling every ACK
reception,
 * Westwood+ samples the bandwidth every RTT.
 *
 * The two main methods in the implementation are the CountAck (const
TCPHeader&)
 * and the EstimateBW (int, const, Time). The CountAck method
calculates
 * the number of acknowledged segments on the receipt of an ACK.
 * The EstimateBW estimates the bandwidth based on the value returned
by CountAck
 * and the sampling interval (last ACK inter-arrival time for
Westwood and last RTT for Westwood+).
 */
class TcpWestwood : public TcpSocketBase
{
public:
  /**
   * \brief Get the type ID.
   * \return the object TypeId
   */
  static TypeId GetTypeId (void);

  TcpWestwood (void);
  /**
   * \brief Copy constructor
   * \param sock the object to copy
   */
TcpWestwood (const TcpWestwood& sock);
virtual ~TcpWestwood (void);

/**
 * \brief Protocol variant (Westwood or Westwood+)
 */
enum ProtocolType
{
  WESTWOOD,
  WESTWOODPLUS
};

  /**
   * \brief Filter type (None or Tustin)
   */
  enum FilterType
  {
    NONE,
    TUSTIN
  };

  // From TcpSocketBase
  virtual int Connect (const Address &address);
  virtual int Listen (void);

protected:
```

```
  virtual uint32_t Window (void); // Return the max possible number
of unacked bytes
  virtual Ptr<TcpSocketBase> Fork (void); // Call
CopyObject<TcpTahoe> to clone me
  virtual void NewAck (SequenceNumber32 const& seq); // Inc cwnd and
call NewAck() of parent
  virtual void DupAck (const TcpHeader& t, uint32_t count);  // Treat
3 dupack as timeout
  virtual void Retransmit (void); // Retransmit time out

  /**
   * Process the newly received ACK
   *
   * \param packet the received ACK packet
   * \param tcpHeader the header attached to the ACK packet
   */
  virtual void ReceivedAck (Ptr<Packet> packet, const TcpHeader&
tcpHeader);

  /**
   * Estimate the RTT, record the minimum value,
   * and run a clock on the RTT to trigger Westwood+ bandwidth
sampling
   * \param header the packet header
   */
  virtual void EstimateRtt (const TcpHeader& header);

  // Implementing ns3::TcpSocket -- Attribute get/set
  virtual void     SetSegSize (uint32_t size);
  virtual void     SetInitialSSThresh (uint32_t threshold);
  virtual uint32_t GetInitialSSThresh (void) const;
  virtual void     SetInitialCwnd (uint32_t cwnd);
  virtual uint32_t GetInitialCwnd (void) const;

private:
  /**
   * Initialize cwnd at the beginning of a connection
   */
  void InitializeCwnd (void);

  /**
   * Calculate the number of acknowledged packets upon the receipt of
an ACK packet
   *
   * \param tcpHeader the header of the received ACK packet
   * \return the number of ACKed packets
   */
  int CountAck (const TcpHeader& tcpHeader);

  /**
   * Update the total number of acknowledged packets during the
current RTT
   *
   * \param acked the number of packets the currently received ACK
acknowledges
   */
  void UpdateAckedSegments (int acked);

  /**
   * Estimate the network's bandwidth
```

```
   *
   * \param acked the number of acknowledged packets returned by
CountAck
   * \param tcpHeader the header of the packet
   * \param rtt the RTT estimation
   */
  void EstimateBW (int acked, const TcpHeader& tcpHeader, Time rtt);

  /**
   * Tustin filter
   */
  void Filtering (void);

protected:
  TracedValue<uint32_t>  m_cWnd;                //!< Congestion
window
  TracedValue<uint32_t>  m_ssThresh;            //!< Slow Start
Threshold
  uint32_t               m_initialCWnd;         //!< Initial cWnd
value
  uint32_t               m_initialSsThresh;     //!< Initial Slow
Start Threshold value
  bool                   m_inFastRec;           //!< Currently in
fast recovery if TRUE

  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
  Time                   m_minRtt;              //!< Minimum RTT
  double                 m_lastAck;             //!< The time last
ACK was received
  SequenceNumber32       m_prevAckNo;           //!< Previously
received ACK number
  int                    m_accountedFor;        //!< The number of
received DUPACKs
  enum ProtocolType      m_pType;               //!< 0 for
Westwood, 1 for Westwood+
  enum FilterType        m_fType;               //!< 0 for none, 1
for Tustin

  int                    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+

};

} // namespace ns3

#endif /* TCP_WESTWOOD_H */
```

APPENDIX D

TCPW CONGESTION CONTROL SOURCE CODE

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2013 ResiliNets, ITTC, University of Kansas
 *
 * This program is free software; you can redistribute it and/or
modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-
1307  USA
 *
 * Authors: Siddharth Gangadhar <siddharth@ittc.ku.edu>, Truc Anh N.
Nguyen <annguyen@ittc.ku.edu>,
 * and Greeshma Umapathi
 *
 * James P.G. Sterbenz <jpgs@ittc.ku.edu>, director
 * ResiliNets Research Group  http://wiki.ittc.ku.edu/resilinets
 * Information and Telecommunication Technology Center (ITTC)
 * and Department of Electrical Engineering and Computer Science
 * The University of Kansas Lawrence, KS USA.
 *
 * Work supported in part by NSF FIND (Future Internet Design)
Program
 * under grant CNS-0626918 (Postmodern Internet Architecture),
 * NSF grant CNS-1050226 (Multilayer Network Resilience Analysis and
Experimentation on GENI),
 * US Department of Defense (DoD), and ITTC at The University of
Kansas.
 */

#define NS_LOG_APPEND_CONTEXT \
  if (m_node) { std::clog << Simulator::Now ().GetSeconds () << "
[node " << m_node->GetId () << "] "; }

#include "tcp-westwood.h"
#include "ns3/log.h"
#include "ns3/trace-source-accessor.h"
#include "ns3/simulator.h"
#include "ns3/abort.h"
#include "ns3/node.h"
#include "ns3/sequence-number.h"
#include "rtt-estimator.h"

NS_LOG_COMPONENT_DEFINE("TcpWestwood");

namespace ns3 {
```

```
NS_OBJECT_ENSURE_REGISTERED(TcpWestwood);

TypeId
TcpWestwood::GetTypeId (void)
{
  static TypeId tid = TypeId("ns3::TcpWestwood")
      .SetParent<TcpSocketBase>()
      .AddConstructor<TcpWestwood>()
      .AddTraceSource("CongestionWindow", "The TCP connection's
congestion window",
                      MakeTraceSourceAccessor(&TcpWestwood::m_cWnd))
      .AddTraceSource ("SlowStartThreshold",
                       "TCP slow start threshold (bytes)",
                       MakeTraceSourceAccessor
(&TcpWestwood::m_ssThresh))
      .AddAttribute("FilterType", "Use this to choose no filter or
Tustin's approximation filter",
                      EnumValue(TcpWestwood::TUSTIN),
MakeEnumAccessor(&TcpWestwood::m_fType),
                      MakeEnumChecker(TcpWestwood::NONE, "None",
TcpWestwood::TUSTIN, "Tustin"))
      .AddAttribute("ProtocolType", "Use this to let the code run as
Westwood or WestwoodPlus",
                      EnumValue(TcpWestwood::WESTWOOD),
                      MakeEnumAccessor(&TcpWestwood::m_pType),
                      MakeEnumChecker(TcpWestwood::WESTWOOD,
"Westwood",TcpWestwood::WESTWOODPLUS, "WestwoodPlus"))
      .AddTraceSource("EstimatedBW", "The estimated bandwidth",

MakeTraceSourceAccessor(&TcpWestwood::m_currentBW));
  return tid;
}

TcpWestwood::TcpWestwood (void) :
  m_inFastRec(false),
  m_currentBW(0),
  m_lastSampleBW(0),
  m_lastBW(0),
  m_minRtt(0),
  m_lastAck(0),
  m_prevAckNo(0),
  m_accountedFor(0),
  m_ackedSegments(0),
  m_IsCount(false)
{
  NS_LOG_FUNCTION (this);
}

TcpWestwood::TcpWestwood (const TcpWestwood& sock) :
  TcpSocketBase(sock),
  m_cWnd(sock.m_cWnd),
  m_ssThresh(sock.m_ssThresh),
  m_initialCWnd(sock.m_initialCWnd),
  m_initialSsThresh (sock.m_initialSsThresh),
  m_inFastRec(false),
  m_currentBW(sock.m_currentBW),
  m_lastSampleBW(sock.m_lastSampleBW),
  m_lastBW(sock.m_lastBW),
  m_minRtt(sock.m_minRtt),
  m_lastAck(sock.m_lastAck),
```

```
    m_prevAckNo(sock.m_prevAckNo),
    m_accountedFor(sock.m_accountedFor),
    m_pType(sock.m_pType),
    m_fType(sock.m_fType),
    m_IsCount(sock.m_IsCount)
{
  NS_LOG_FUNCTION (this);
  NS_LOG_LOGIC ("Invoked the copy constructor");
  NS_LOG_INFO ("m_minRtt at copy constructor" << m_minRtt);
}

TcpWestwood::~TcpWestwood (void)
{
}

int
TcpWestwood::Listen (void)
{
  NS_LOG_FUNCTION (this);
  InitializeCwnd();
  return TcpSocketBase::Listen();
}

int
TcpWestwood::Connect (const Address & address)
{
  NS_LOG_FUNCTION (this << address);
  InitializeCwnd();
  return TcpSocketBase::Connect(address);
}

uint32_t
TcpWestwood::Window (void)
{
  NS_LOG_FUNCTION (this);
  return std::min (m_rWnd.Get (), m_cWnd.Get ());
}

Ptr<TcpSocketBase>
TcpWestwood::Fork (void)
{
  NS_LOG_FUNCTION (this);
  return CopyObject<TcpWestwood>(this);
}

void
TcpWestwood::NewAck (const SequenceNumber32& seq)
{ // Same as Reno
  NS_LOG_FUNCTION (this << seq);
  NS_LOG_LOGIC ("TcpWestwood receieved ACK for seq " << seq <<
                " cwnd " << m_cWnd <<
                " ssthresh " << m_ssThresh);

  // Check for exit condition of fast recovery
  if (m_inFastRec)
    {// First new ACK after fast recovery, reset cwnd as in Reno
      m_cWnd = m_ssThresh;
      m_inFastRec = false;
      NS_LOG_INFO ("Reset cwnd to " << m_cWnd);
    };
```

```
  // Increase of cwnd based on current phase (slow start or
congestion avoidance)
  if (m_cWnd < m_ssThresh)
    { // Slow start mode, add one segSize to cWnd as in Reno
      m_cWnd += m_segmentSize;
      NS_LOG_INFO ("In SlowStart, updated to cwnd " << m_cWnd << "
ssthresh " << m_ssThresh);
    }
  else
    { // Congestion avoidance mode, increase by
(segSize*segSize)/cwnd as in Reno
      double adder = static_cast<double> (m_segmentSize *
m_segmentSize) / m_cWnd.Get();
      adder = std::max(1.0, adder);
      m_cWnd += static_cast<uint32_t>(adder);
      NS_LOG_INFO ("In CongAvoid, updated to cwnd " << m_cWnd << "
ssthresh " << m_ssThresh);
    }

  // Complete newAck processing
  TcpSocketBase::NewAck(seq);
}

void
TcpWestwood::ReceivedAck (Ptr<Packet> packet, const TcpHeader&
tcpHeader)
{
  NS_LOG_FUNCTION (this);
  int acked = 0;
  if ((0 != (tcpHeader.GetFlags () & TcpHeader::ACK)) &&
tcpHeader.GetAckNumber() >= m_prevAckNo)
    {// It is a duplicate ACK or a new ACK. Old ACK is ignored.
      if (m_pType == TcpWestwood::WESTWOOD)
        {// For Westwood, calculate the number of ACKed segments and
estimate the BW
          acked = CountAck (tcpHeader);
          EstimateBW (acked, tcpHeader, Time(0));
        }
      else if (m_pType == TcpWestwood::WESTWOODPLUS)
        {// For Westwood+, calculate the number of ACKed segments and
update m_ackedSegments
          if (m_IsCount)
            {
              acked = CountAck (tcpHeader);
              UpdateAckedSegments (acked);
            }
        }
    }

  TcpSocketBase::ReceivedAck (packet, tcpHeader);
}

void
TcpWestwood::EstimateBW (int acked, const TcpHeader& tcpHeader, Time
rtt)
{
  NS_LOG_FUNCTION (this);
  if (m_pType == TcpWestwood::WESTWOOD)
    {
```

```
      // Get the time when the current ACK is received
      double currentAck = static_cast<double>
(Simulator::Now():GetSeconds());
      // Calculate the BW
      m_currentBW = acked * m_segmentSize / (currentAck - m_lastAck);
      // Update the last ACK time
      m_lastAck = currentAck;
    }
  else if (m_pType == TcpWestwood::WESTWOODPLUS)
    {
      // Calculate the BW
      m_currentBW = m_ackedSegments * m_segmentSize /
rtt.GetSeconds();
      // Reset m_ackedSegments and m_IsCount for the next sampling
      m_ackedSegments = 0;
      m_IsCount = false;
    }

  // Filter the BW sample
  Filtering();
}

int
TcpWestwood::CountAck (const TcpHeader& tcpHeader)
{
  NS_LOG_FUNCTION (this);

  // Calculate the number of acknowledged segments based on the
received ACK number
  int cumul_ack = (tcpHeader.GetAckNumber() - m_prevAckNo) /
m_segmentSize;

  if (cumul_ack == 0)
    {// A DUPACK counts for 1 segment delivered successfully
      m_accountedFor++;
      cumul_ack = 1;
    }
  if (cumul_ack > 1)
    {// A delayed ACK or a cumulative ACK after a retransmission
     // Check how much new data it ACKs
      if (m_accountedFor >= cumul_ack)
        {
          m_accountedFor -= cumul_ack;
          cumul_ack = 1;
        }
      else if (m_accountedFor < cumul_ack)
        {
          cumul_ack -= m_accountedFor;
          m_accountedFor = 0;
        }
    }

  // Update the previous ACK number
  m_prevAckNo = tcpHeader.GetAckNumber();

  return cumul_ack;
}

void
TcpWestwood::UpdateAckedSegments (int acked)
```

```
{
  m_ackedSegments += acked;
}

void
TcpWestwood::DupAck (const TcpHeader& header, uint32_t count)
{
  NS_LOG_FUNCTION (this << count << m_cWnd);

  if (count == 3 && !m_inFastRec)
    {// Triple duplicate ACK triggers fast retransmit
     // Adjust cwnd and ssthresh based on the estimated BW
      m_ssThresh = uint32_t(m_currentBW * static_cast<double>
(m_minRtt.GetSeconds()));
      if (m_cWnd > m_ssThresh)
        {
          m_cWnd = m_ssThresh;
        }
      m_inFastRec = true;
      NS_LOG_INFO ("Triple dupack. Enter fast recovery mode. Reset
cwnd to " << m_cWnd <<", ssthresh to " << m_ssThresh);
      DoRetransmit ();
    }
  else if (m_inFastRec)
    {// Increase cwnd for every additional DUPACK as in Reno
      m_cWnd += m_segmentSize;
      NS_LOG_INFO ("Dupack in fast recovery mode. Increase cwnd to "
<< m_cWnd);
      SendPendingData (m_connected);
    }
}

void
TcpWestwood::Retransmit (void)
{
  NS_LOG_FUNCTION (this);
  NS_LOG_LOGIC (this << " ReTxTimeout Expired at time " <<
Simulator::Now ().GetSeconds ());
  m_inFastRec = false;

  // If erroneous timeout in closed/timed-wait state, just return
  if (m_state == CLOSED || m_state == TIME_WAIT)
    return;
  // If all data are received, just return
  if (m_txBuffer.HeadSequence() >= m_nextTxSequence)
    return;

  // Upon an RTO, adjust cwnd and ssthresh based on the estimated BW
  m_ssThresh = std::max (static_cast<double> (2 * m_segmentSize),
m_currentBW.Get() * static_cast<double> (m_minRtt.GetSeconds()));
  m_cWnd = m_segmentSize;

  // Restart from highest ACK
  m_nextTxSequence = m_txBuffer.HeadSequence();
  NS_LOG_INFO ("RTO. Reset cwnd to " << m_cWnd <<
       ", ssthresh to " << m_ssThresh << ", restart from seqnum " <<
m_nextTxSequence);

  // Double the next RTO
  m_rtt->IncreaseMultiplier();
```

```
    // Retransmit the packet
    DoRetransmit();
}

void
TcpWestwood::EstimateRtt (const TcpHeader& tcpHeader)
{
    NS_LOG_FUNCTION_NOARGS ();

    // Calculate m_lastRtt
    TcpSocketBase::EstimateRtt (tcpHeader);

    // Update minRtt
    if (m_minRtt == Time (0))
      {
        m_minRtt = m_lastRtt;
      }
    else
      {
        if (m_lastRtt < m_minRtt)
          {
            m_minRtt = m_lastRtt;
          }
      }

    // For Westwood+, start running a clock on the currently estimated
RTT if possible
    // to trigger a new BW sampling event
    if (m_pType == TcpWestwood::WESTWOODPLUS)
      {
        if(m_lastRtt != Time (0) && m_state == ESTABLISHED &&
!m_IsCount)
          {
            m_IsCount = true;
            m_bwEstimateEvent.Cancel();
            m_bwEstimateEvent = Simulator::Schedule (m_lastRtt,
&TcpWestwood::EstimateBW,this,m_ackedSegments,tcpHeader,m_lastRtt);
          }
      }
}

void
TcpWestwood::Filtering ()
{
    NS_LOG_FUNCTION (this);

    double alpha = 0.9;

    if (m_fType == TcpWestwood::NONE)
      {
      }
    else if (m_fType == TcpWestwood::TUSTIN)
      {
        double sample_bwe = m_currentBW;
        m_currentBW = (alpha * m_lastBW) + ((1 - alpha) * ((sample_bwe
+ m_lastSampleBW) / 2));
        m_lastSampleBW = sample_bwe;
        m_lastBW = m_currentBW;
      }
```

```
}

void
TcpWestwood::SetSegSize (uint32_t size)
{
  NS_ABORT_MSG_UNLESS(m_state == CLOSED, "TcpWestwood::SetSegSize()
cannot change segment size after connection started.");
  m_segmentSize = size;
}

void
TcpWestwood::SetInitialSSThresh (uint32_t threshold)
{
  NS_LOG_FUNCTION (this);
  NS_ABORT_MSG_UNLESS (m_state == CLOSED, "TcpWestwood::SetSSThresh()
cannot change initial ssThresh after connection started.");
  m_initialSsThresh = threshold;
}

uint32_t
TcpWestwood::GetInitialSSThresh (void) const
{
  NS_LOG_FUNCTION (this);
  return m_initialSsThresh;
}

void
TcpWestwood::SetInitialCwnd (uint32_t cwnd)
{
  NS_ABORT_MSG_UNLESS(m_state == CLOSED,
"TcpWestwood::SetInitialCwnd() cannot change initial cwnd after
connection started.");
  m_initialCWnd = cwnd;
}

uint32_t
TcpWestwood::GetInitialCwnd (void) const
{
  NS_LOG_FUNCTION (this);
  return m_initialCWnd;
}

void
TcpWestwood::InitializeCwnd(void)
{
  NS_LOG_FUNCTION (this);
  /*
   * Initialize congestion window, default to 1 MSS (RFC2001, sec.1)
and must
   * not be larger than 2 MSS (RFC2581, sec.3.1). Both m_initiaCWnd
and
   * m_segmentSize are set by the attribute system in ns3::TcpSocket.
   */
  m_cWnd = m_initialCWnd * m_segmentSize;
  m_ssThresh = m_initialSsThresh;
}

} // namespace ns3
```

APPENDIX E

PETRA CONGESTION CONTROL SOURCE CODE

```
#define NS_LOG_APPEND_CONTEXT \
  if (m_node) { std::clog << Simulator::Now ().GetSeconds () << "
[node " << m_node->GetId () << "] "; }

#include "tcp-westwood.h"
#include "ns3/log.h"
#include "ns3/trace-source-accessor.h"
#include "ns3/simulator.h"
#include "ns3/abort.h"
#include "ns3/node.h"
#include "ns3/sequence-number.h"
#include "rtt-estimator.h"

NS_LOG_COMPONENT_DEFINE("TcpPETRA");

namespace ns3 {

NS_OBJECT_ENSURE_REGISTERED(TcpPETRA);

TypeId
TcpPETRA::GetTypeId (void)
{
  static TypeId tid = TypeId("ns3::TcpPETRA")
      .SetParent<TcpSocketBase>()
      .AddConstructor<TcpPETRA>()
      .AddTraceSource("CongestionWindow", "The TCP connection's
congestion window",
                      MakeTraceSourceAccessor(&TcpPETRA::m_cWnd))
      .AddAttribute("FilterType", "Use this to choose no filter or
Tustin's approximation filter",
                    EnumValue(TcpPETRA::TUSTIN),
MakeEnumAccessor(&TcpPETRA::m_fType),
                    MakeEnumChecker(TcpPETRA::NONE, "None",
TcpPETRA::TUSTIN, "Tustin"))
      .AddAttribute("ProtocolType", "Use this to let the code run as
PETRA or WestwoodPlus",
                    EnumValue(TcpPETRA::PETRA),
                    MakeEnumAccessor(&TcpPETRA::m_pType),
                    MakeEnumChecker(TcpPETRA::PETRA,
"PETRA",TcpWestwood::WESTWOODPLUS, "WestwoodPlus"))
      .AddTraceSource("EstimatedBW", "The estimated bandwidth",
                      MakeTraceSourceAccessor(&TcpPETRA::m_currentBW));
  return tid;
}

TcpPETRA::TcpPETRA (void) :
  m_inFastRec(false),
  m_currentBW(0),
  m_lastSampleBW(0),
  m_lastBW(0),
  m_minRtt(0),
  m_lastAck(0),
  m_prevAckNo(0),
  m_accountedFor(0),
```

```
    m_ackedSegments(0),
    m_IsCount(false)
{
    NS_LOG_FUNCTION (this);
}

TcpPETRA::TcpPETRA (const TcpPETRA& sock) :
    TcpSocketBase(sock),
    m_cWnd(sock.m_cWnd),
    m_ssThresh(sock.m_ssThresh),
    m_initialCWnd(sock.m_initialCWnd),
    m_inFastRec(false),
    m_currentBW(sock.m_currentBW),
    m_lastSampleBW(sock.m_lastSampleBW),
    m_lastBW(sock.m_lastBW),
    m_minRtt(sock.m_minRtt),
    m_lastAck(sock.m_lastAck),
    m_prevAckNo(sock.m_prevAckNo),
    m_accountedFor(sock.m_accountedFor),
    m_pType(sock.m_pType),
    m_fType(sock.m_fType),
    m_IsCount(sock.m_IsCount)
{
    NS_LOG_FUNCTION (this);
    NS_LOG_LOGIC ("Invoked the copy constructor");
    NS_LOG_INFO ("m_minRtt at copy constructor" << m_minRtt);
}

TcpPETRA::~TcpPETRA (void)
{
}
double last_EBW(0);
bool IsInitial(true);
int
TcpPETRA::Listen (void)
{
    NS_LOG_FUNCTION (this);
    InitializeCwnd();
    return TcpSocketBase::Listen();
}

int
TcpPETRA::Connect (const Address & address)
{
    NS_LOG_FUNCTION (this << address);
    InitializeCwnd();
    return TcpSocketBase::Connect(address);
}

uint32_t
TcpPETRA::Window (void)
{
    NS_LOG_FUNCTION (this);
    return std::min (m_rWnd.Get (), m_cWnd.Get ());
}

Ptr<TcpSocketBase>
TcpPETRA::Fork (void)
{
    NS_LOG_FUNCTION (this);
```

```
    return CopyObject<TcpPETRA>(this);
}

void
TcpPETRA::NewAck (const SequenceNumber32& seq)
{ // Same as Reno
  NS_LOG_FUNCTION (this << seq);
  NS_LOG_LOGIC ("TcpPETRA receieved ACK for seq " << seq <<
                " cwnd " << m_cWnd <<
                " ssthresh " << m_ssThresh);
  uint32_t EBW= m_currentBW * static_cast<double>
(m_minRtt.GetSeconds());
   // Check for exit condition of fast recovery
   if (m_inFastRec && seq < m_prevAckNo)
     {
        m_cWnd += m_segmentSize - (seq - m_txBuffer.HeadSequence ());
        NS_LOG_INFO ("Partial ACK in fast recovery: cwnd set to " <<
m_cWnd);
        m_txBuffer.DiscardUpTo(seq);  //Bug 1850:  retransmit before
newack
        DoRetransmit (); // Assume the next seq is lost. Retransmit
lost packet
        TcpSocketBase::NewAck (seq); // update m_nextTxSequence and
send new data if allowed by window
       return;
     }
      else if (m_inFastRec && seq >= m_prevAckNo)
       {// First new ACK after fast recovery, reset cwnd as in Reno
       m_cWnd = EBW;
       m_inFastRec = false;
       NS_LOG_INFO ("Reset cwnd to " << m_cWnd);
       //FlightSzie+=m_cWnd;
      };
   if (BytesInFlight () >= EBW)
   { // Congestion avoidance mode, increase by (segSize*segSize)/cwnd
as in Reno
       double adder = static_cast<double> (m_segmentSize *
m_segmentSize) / m_cWnd.Get();
       adder = std::max(1.0, adder);
       m_cWnd += static_cast<uint32_t>(adder);
       NS_LOG_INFO ("In CongAvoid, updated to cwnd " << m_cWnd << "
ssthresh " << m_ssThresh);
       //FlightSzie+=m_cWnd;
     }
  // Increase of cwnd based on current phase (slow start or
congestion avoidance)
  else if ( EBW /BytesInFlight () > 2)
     { // Faster start mode, add two segSize to cWnd
       m_cWnd += 2*m_segmentSize;
       NS_LOG_INFO ("In SlowStart, updated to cwnd " << m_cWnd << "
ssthresh " << m_ssThresh);
       //FlightSzie+=m_cWnd;
     }
  else
     { // Normal slow start
       m_cWnd += m_segmentSize;
     }

  // Complete newAck processing
```

```
    TcpSocketBase::NewAck(seq);
  }

  void
  TcpPETRA::ReceivedAck (Ptr<Packet> packet, const TcpHeader&
  tcpHeader)
  {
    NS_LOG_FUNCTION (this);
    int acked = 0;
    if ((0 != (tcpHeader.GetFlags () & TcpHeader::ACK)) &&
  tcpHeader.GetAckNumber() >= m_prevAckNo)
      {// It is a duplicate ACK or a new ACK. Old ACK is ignored.
        if (m_pType == TcpPETRA::PETRA)
          {// For PETRA, calculate the number of ACKed segments and
  estimate the BW
            acked = CountAck (tcpHeader);
            EstimateBW (acked, tcpHeader, Time(0));
          }
        else if (m_pType == TcpWestwood::WESTWOODPLUS)
          {// For Weswood+, calculate the number of ACKed segments and
  update m_ackedSegments
            if (m_IsCount)
              {
                acked = CountAck (tcpHeader);
                UpdateAckedSegments (acked);
              }
          }
      }
    if (IsInitial==true)
    m_ssThresh = std::max (static_cast<double> (m_ssThresh),
    m_currentBW.Get() * static_cast<double> (m_minRtt.GetSeconds()));
    TcpSocketBase::ReceivedAck (packet, tcpHeader);
  }

  void
  TcpPETRA::EstimateBW (int acked, const TcpHeader& tcpHeader, Time
  rtt)
  {
    double alpha = 0.9;
    NS_LOG_FUNCTION (this);
    if (m_pType == TcpPETRA::PETRA)
      {
        // Get the time when the current ACK is received
        double currentAck = static_cast<double>
  (Simulator::Now().GetSeconds());
        // Calculate the BW
        m_currentBW = acked * m_segmentSize / (currentAck - m_lastAck);
        // Update the last ACK time
        if (IsInitial==true)
        m_currentBW = (alpha*m_currentBW) + (1- alpha)*last_EBW;
        m_lastAck = currentAck;
        last_EBW = m_currentBW;
      }
    else if (m_pType == TcpWestwood::WESTWOODPLUS)
      {
        // Calculate the BW
        m_currentBW = m_ackedSegments * m_segmentSize /
  rtt.GetSeconds();
        // Reset m_ackedSegments and m_IsCount for the next sampling
        m_ackedSegments = 0;
```

```
      m_IsCount = false;
    }

  // Filter the BW sample
  Filtering();
}

int
TcpPETRA::CountAck (const TcpHeader& tcpHeader)
{
  NS_LOG_FUNCTION (this);
  // Calculate the number of acknowledged segments based on the
received ACK number
  int cumul_ack = (tcpHeader.GetAckNumber() - m_prevAckNo) /
m_segmentSize;

  if (cumul_ack == 0)
    {// A DUPACK counts for 1 segment delivered successfully
      m_accountedFor++;
      cumul_ack = 1;
    }
  if (cumul_ack > 1)
    {// A delayed ACK or a cumulative ACK after a retransmission
      // Check how much new data it ACKs
      if (m_accountedFor >= cumul_ack)
        {
          m_accountedFor -= cumul_ack;
          cumul_ack = 1;
        }
      else if (m_accountedFor < cumul_ack)
        {
          cumul_ack -= m_accountedFor;
          m_accountedFor = 0;
        }
    }

  // Update the previous ACK number
  m_prevAckNo = tcpHeader.GetAckNumber();

  return cumul_ack;
}

void
TcpPETRA::UpdateAckedSegments (int acked)
{
  m_ackedSegments += acked;
}

void
TcpPETRA::DupAck (const TcpHeader& header, uint32_t count)
{
  Time Frtt;
  Frtt=m_lastRtt-m_rtt->GetCurrentEstimate();
  NS_LOG_FUNCTION (this << count << m_cWnd);
  if  (((4-count) * m_lastRtt >= m_rtt->RetransmitTimeout()) &&
!m_inFastRec)
      {
       m_ssThresh = m_currentBW * static_cast<double>
(m_minRtt.GetSeconds());
```

```
  if (m_cWnd > m_ssThresh)
    {
      m_cWnd = m_ssThresh;
    }
  m_prevAckNo = m_highTxMark;
  m_inFastRec = true;
  DoRetransmit ();
}
else if (count == 3 && !m_inFastRec)
 // Triple duplicate ACK triggers fast retransmit
 // Adjust cwnd and ssthresh based on the estimated BW
  {
    m_ssThresh = m_currentBW * static_cast<double>
(m_minRtt.GetSeconds());
    if (m_cWnd > m_ssThresh)
      {
        m_cWnd = m_ssThresh;
      }
    m_prevAckNo = m_highTxMark;
    m_inFastRec = true;
    NS_LOG_INFO ("Triple dupack. Enter fast recovery mode. Reset
cwnd to " << m_cWnd <<
                 ", ssthresh to " << m_ssThresh << " at fast
recovery seqnum " << m_prevAckNo);
    DoRetransmit ();
    }
  else if (m_inFastRec)
    {// Increase cwnd for every additional DUPACK as in Reno
     if (Frtt >0)
     m_cWnd += 2*m_segmentSize;
     else
     m_cWnd += m_segmentSize;

     NS_LOG_INFO ("Dupack in fast recovery mode. Increase cwnd to "
<< m_cWnd);
     SendPendingData (m_connected);
     }
  else if (!m_inFastRec  && m_txBuffer.SizeFromSequence
(m_nextTxSequence) > 0)
    { // RFC3042 Limited transmit: Send a new packet for each
duplicated ACK before fast retransmit
      NS_LOG_INFO ("Limited transmit");
      uint32_t sz = SendDataPacket (m_nextTxSequence, m_segmentSize,
true);
      m_nextTxSequence += sz;                    // Advance next tx
sequence
    };
}

void
TcpPETRA::Retransmit (void)
{
  NS_LOG_FUNCTION (this);
  NS_LOG_LOGIC (this << " ReTxTimeout Expired at time " <<
Simulator::Now ().GetSeconds ());
  m_inFastRec = false;

  // If erroneous timeout in closed/timed-wait state, just return
  if (m_state == CLOSED || m_state == TIME_WAIT)
    return;
```

```
  // If all data are received, just return
  if (m_txBuffer.HeadSequence() >= m_nextTxSequence)
    return;

  // Upon an RTO, adjust cwnd and ssthresh based on the estimated BW
  m_ssThresh = std::max (static_cast<double> (2 * m_segmentSize),
m_currentBW.Get() * static_cast<double> (m_minRtt.GetSeconds()));
  if (BytesInFlight () < m_ssThresh && m_lastRtt <= m_rtt-
>GetCurrentEstimate())
   m_cWnd =  BytesInFlight ()/2;
   //else if (m_cWnd < m_ssThresh)
  else if (m_lastRtt<= m_rtt->GetCurrentEstimate())
   m_cWnd =BytesInFlight ()/4;
  else
  {
  // Double the next RTO
  m_rtt->IncreaseMultiplier();
  m_cWnd = m_segmentSize;
  }
  m_nextTxSequence = m_txBuffer.HeadSequence();
  NS_LOG_INFO ("RTO. Reset cwnd to " << m_cWnd <<
       ", ssthresh to " << m_ssThresh << ", restart from seqnum " <<
m_nextTxSequence);
  // Retransmit the packet
  DoRetransmit();
}

void
TcpPETRA::EstimateRtt (const TcpHeader& tcpHeader)
{
  NS_LOG_FUNCTION_NOARGS ();

// Calculate m_lastRtt
TcpSocketBase::EstimateRtt (tcpHeader);

// Update minRtt
if (m_minRtt == 0)
  {
    m_minRtt = m_lastRtt;
  }
else
  {
    if (m_lastRtt < m_minRtt)
      {
        m_minRtt = m_lastRtt;
      }
  }

  // For Westwood+, start running a clock on the currently estimated
RTT if possible
  // to trigger a new BW sampling event
  if (m_pType == TcpWestwood::WESTWOODPLUS)
   {
    if(m_lastRtt != 0 && m_state == ESTABLISHED && !m_IsCount)
      {
        m_IsCount = true;
        m_bwEstimateEvent.Cancel();
        m_bwEstimateEvent = Simulator::Schedule (m_lastRtt,
&TcpWestwood::EstimateBW,this,m_ackedSegments,tcpHeader,m_lastRtt);
      }
```

```
    }
}

void
TcpPETRA::Filtering ()
{
  NS_LOG_FUNCTION (this);

  double alpha = 0.9;

  if (m_fType == TcpPETRA::NONE)
    {
    }
  else if (m_fType == TcpPETRA::TUSTIN)
    {
      double sample_bwe = m_currentBW;
      m_currentBW = (alpha * m_lastBW) + ((1 - alpha) * ((sample_bwe
+ m_lastSampleBW) / 2));
      m_lastSampleBW = sample_bwe;
      m_lastBW = m_currentBW;
    }
}

void
TcpPETRA::SetSegSize (uint32_t size)
{
  NS_ABORT_MSG_UNLESS(m_state == CLOSED, "TcpPETRA::SetSegSize()
cannot change segment size after connection started.");
  m_segmentSize = size;
}

void
TcpPETRA::SetSSThresh (uint32_t threshold)
{
  NS_LOG_FUNCTION (this);
  m_ssThresh = threshold;
}

uint32_t
TcpPETRA::GetSSThresh (void) const
{
  NS_LOG_FUNCTION (this);
  return m_ssThresh;
}

void
TcpPETRA::SetInitialCwnd (uint32_t cwnd)
{
  NS_ABORT_MSG_UNLESS(m_state == CLOSED, "TcpPETRA::SetInitialCwnd()
cannot change initial cwnd after connection started.");
  m_initialCWnd = cwnd;
}

uint32_t
TcpPETRA::GetInitialCwnd (void) const
{
  NS_LOG_FUNCTION (this);
  return m_initialCWnd;
}
```

```
void
TcpPETRA::InitializeCwnd(void)
{
  NS_LOG_FUNCTION (this);
  /*
   * Initialize congestion window, default to 1 MSS (RFC2001, sec.1)
and must
   * not be larger than 2 MSS (RFC2581, sec.3.1). Both m_initiaCWnd
and
   * m_segmentSize are set by the attribute system in ns3::TcpSocket.
   */
  m_cWnd = m_initialCWnd * m_segmentSize;
}

} // namespace ns3
```

APPENDIX F:
LIST OF PUBLICATIONS

Al-Hasanat, M., Seman, K., & Saadan, K. (August 12-15, 2014). *Enhanced TCP Westwood Congestion Control Mechanism over Wireless Networks.* Paper presented at the International Conference on Advanced Technology and Science (ICAT'14), Antalya, Turkey.

Al-Hasanat, M., Seman, K., & Saadan, K. (2014). Enhanced TCP Westwood Slow Start Phase. *Transactions on Networks And Communications, 2*(5), pp. 194-200. http://dx.doi.org/10.14738/tnc.25.601

Al-Hasanat, M., Seman, K., & Saadan, k. (April 21-23, 2015). *Enhanced TCPW's fast retransmission and fast recovery mechanism over high bit errors networks.* Paper presented at the International Conference on Computer, Communications, and Control Technology (I4CT), Kuching, Malaysia.