Workflow of TCP in NS3

Congestion Control Perspective

Subsection: B2

Group: 6-2

Student IDs:

1705091

1705103

1705119

Evolution of TCP Vegas from TCP Tahoe

TCP Tahoe:

- congestion window (cwnd) increases **exponentially** in **slow start** phase
- cwnd increases **linearly** in **congestion avoidance** phase

TCP Reno:

- **Fast Recovery** and **Fast Retransmit**. Use of **3 duplicate ACKs** to detect congestion
- cwnd falls to **half** instead of 1

TCP NewReno:

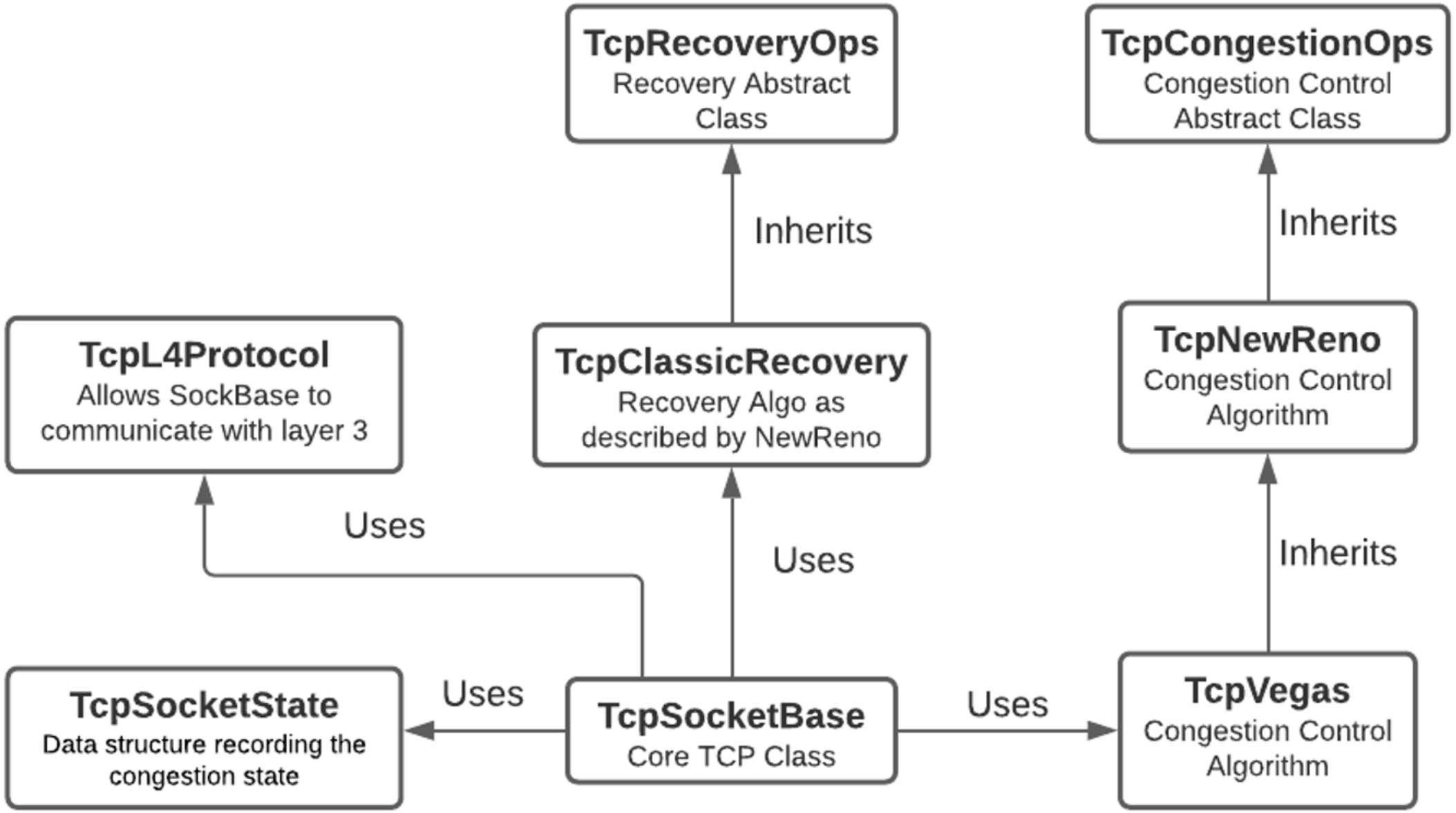
- **cwnd inflation** during **Fast Recovery**
- Fast Recovery phase ends when all segments that might have been lost are ACKed

TCP Vegas:

- Use **packet delay** to determine **transmission rate** in **congestion avoidance** phase

In this presentation we will see how all these concepts are implemented in NS3

Class Diagram of Relevant Classes



TCP Socket State Implementation

TCP Socket State - Important Attributes

```
typedef enum {
  CA_OPEN,        // normal state, no hint of congestion
  CA_DISORDER,    // some dupacks seen, possibility of congestion
  CA_CWR,         // cWnd reduced due to ECN
  CA_RECOVERY,    // cWnd reduced, now fast-retransmitting.
  CA_LOSS,        // cWnd reduced due to RTO timeout
} TcpCongState_t;
```

TCP Socket State - Important Attributes

```
m_cWnd              // Congestion window
m_ssThresh          // Slow start threshold
m_initialCWnd       // Initial cWnd value
m_initialSsThresh   // Initial Slow Start Threshold value
m_segmentSize       // Segment size
m_lastAckedSeq      // Last sequence ACKed
m_highTxMark        // Highest seqno ever sent
m_nextTxSequence    // Next seqnum to be sent
m_minRtt            // Minimum RTT observed
m_bytesInFlight     // Bytes in flight
```

TCP Socket Base Implementation

TCP Socket Base - Initialization and Default Values

TcpL4Protocol :: CreateSocket function creates an **object** of **TCP Socket Base** class

Following functions determine the default attribute values of SocketBase:

- TcpSocketBase :: GetTypeId
- TcpL4Protocol :: GetTypeId

TCP Socket Base - Initialization and Default Values

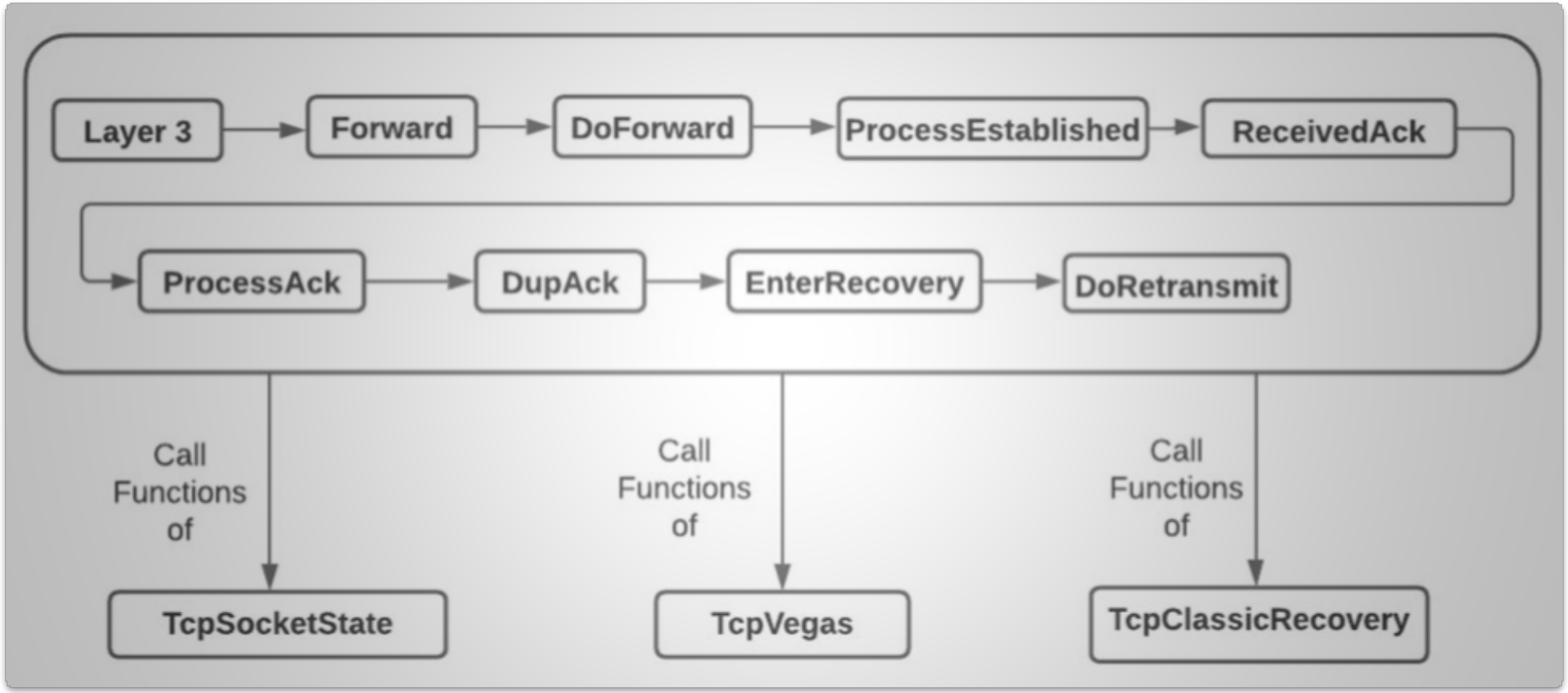
We can **override** the **default** values from **user code**.

For example, the following two lines can be used to **override** the **default congestion control** and **recovery algorithms**

- **Config::SetDefault ("ns3::TcpL4Protocol::SocketType", StringValue ("ns3::TcpVegas"))**
- **Config::SetDefault ("ns3::TcpL4Protocol::RecoveryType", StringValue ("ns3::TcpClassicRecovery "))**

TCP Socket-Base Function Call Sequence

- When **ACK** is **received**, it passes through different functions of **SocketBase**
- Functions of **SocketState**, **CongestionAlgo**, **RecoveryAlgo** are called from **SocketBase**
- This **total system** works together to **implement Congestion Control**



TcpSocketBase :: ForwardUp

- Congestion Control Algorithms might use **Explicit Congestion Notification**
- Congestion Control Algorithm is **notified** about **congestion detected by ECN**
- Here, has **receiver** can detect ECN and inform sender about it.

```
void ForwardUp (packet, Ipv4Header) {
  if ( "ECN Enabled & CE bit set")
    ECN_STATE = "ECN_CE";
    congestionAlgo.CwndEvent();

  DoForwardUp (packet);
}
```

TcpSocketBase :: DoForwardUp

- Take **different actions** depending on whether **connection** is **being established** or **already established**

```
void DoForwardUp (packet) {
  if ( "Sender reduced CW in response to previous CE" && "current CE bit not SET" )
      ECN_STATE = ECN_IDLE

  if ( "SYN bit sit : connection being established")
    Initialize congestionState.cwnd, congestionState.ssThresh, Estimate RTT

  else if( "ACK bit Set")
    Estimate RTT, Update RWND

  if( tcpState==ESTABLISHED ) ProcessEstablished(packet)
  else if( tcpState==SYN_SENT ) ProcessSynSent(packet)
  else if( tcpState==SYN_RVD ) ProcessSynRcvd(packet)
}
```

TcpSocketBase :: ProcessEstablished

- **ACK.SEG** : Sequence Number of Acknowledgement
- **SND.UNA** : First Unacknowledged Sequence Number in transmission buffer
- **HighTxMark** : Highest Sequence Number ever transmitted

```
void ProcessEstablished (packet) {
  if ( "ACK Flag Set" )
    if( "ACK.SEG < SND.UNA" ) // ACK is Stale, Ignore
    else if( "ACK.SEG > HighTxMark" ) // Errorenous Case
    else ReceivedAck(packet)
  else if( "other Flag conditions" )
    // handled accordingly, irrelevant to Congestion Control
}
```

TcpSocketBase :: ReceivedAck

- Congestion can be detected through **Explicit** or **Implicit** means

```
void ReceivedAck (packet) {
  // Discard delivered segments from tranmission buffer

  if( "check if previously started congestion recovery is over")
    congestionState.state = "CA_OPEN"
    congestionState.cwd = congestionState.ssThresh()
    recoveryAlgo.ExitRecovery()
    congestionAlgo.CwndEvent()

  if( "ECN type congestion detection from receiver" ) EnterCwr()

  ProcessAck(packet) // handle implicit type congestion detection
}
```

TcpSocketBase :: ProcessAck

Actual function is 297 lines! Greatly simplified here to show just relevant function calls

```
void ProcessAck(packet)
  if( "ACK.SEG==SND.UNA" && "ACK.SEG<socketState.highTxMark" ) DupAck()
  else if( "ACK.SEG == SND.UNA" ) congestionAlgo.PktsAcked(socketState)
  else if ( "ACK.SEG > SND.UNA" )
    dupAckCount = 0 // because ACK is cumulative
    congestionAlgo.PktsAcked(socketState)
    if ( "socketState.state == CA_RECOVERY" ) recoveryAlgo.DoRecovery(socketState)
    else congestionAlgo.IncreaseWindow(socketState)

    if( "ACK causes CA_RECOVERY to end" )
      congestionState.cwd = congestionState.ssThresh()
      recoveryAlgo.ExitRecovery()
    if( "ACK causes any congestion state change" )
      congestionAlgo.CwndEvent()
      congestionAlgo.CongestionStateSet("x") // "x" is the new congestion state
      congestionState.state = "x"
```

TcpSocketBase :: DupAck

```
void DupAck () {
  if ( "congestionState.state == CA_RECOVERY" )
    recoveryAlgo.DoRecovery(congestionState)
  else
    dupAckCount++;

  if ( "congestionState.state == CA_OPEN" ) congestionState.state = "CA_DISORDER"

  else if ( "congestionState.state == CA_DISORDER" )
    if( "dupAckCount > dupAckThreshold" )
          EnterRecovery()
}
```

TcpSocketBase :: EnterRecovery

```
void EnterRecovery () {
  congestionState.state = "CA_RECOVERY"
  congestionAlgo.CongestionStateSet(congestionState)

  congestionState.ssThresh = congestionAlgo.GetSsThresh(congestionState, bytesInFlight())

  recoveryAlgo.EnterRecovery(congestionState)

  // Retransmit the first data segment presumed dropped
  // For NewReno, it is SND.UNA
  DoRetransmit ();
}
```

TCP Recovery Ops Implementation

TCP Recovery Ops - Declaration

- It is an **abstract class**
- **Inherited** by specific **recovery algorithm** classes
- For example:
  - TcpClassicRecovery
  - TcpPrrRecovery

```
class TcpRecoveryOps : public Object {
public:
  static TypeId GetTypeId (void);
  TcpRecoveryOps ();
  virtual ~TcpRecoveryOps ();

  virtual void EnterRecovery (socketState) = 0;
  virtual void DoRecovery (socketState) = 0;
  virtual void ExitRecovery (socketState) = 0;
};
```

TCP Classic Recovery Implementation

TCP Classic Recovery - Inherits TCP Recovery Ops

- These 3 functions were **pure virtual** in **TCP Recovery Ops**
- **TCP Classic Recovery** defines these functions

```
void EnterRecovery (socketState) {
  socketState.cWnd = socketState.ssThresh;
  socketState.cWndInfl = socketState.ssThresh + (dupAckCount * tcb->m_segmentSize);
}

void DoRecovery (socketState) {
  socketState.cWndInfl += socketState.segmentSize;
}

void ExitRecovery (socketState) {
  socketState.cWndInfl = socketState.ssThresh;
}
```

TCP Congestion Ops Implementation

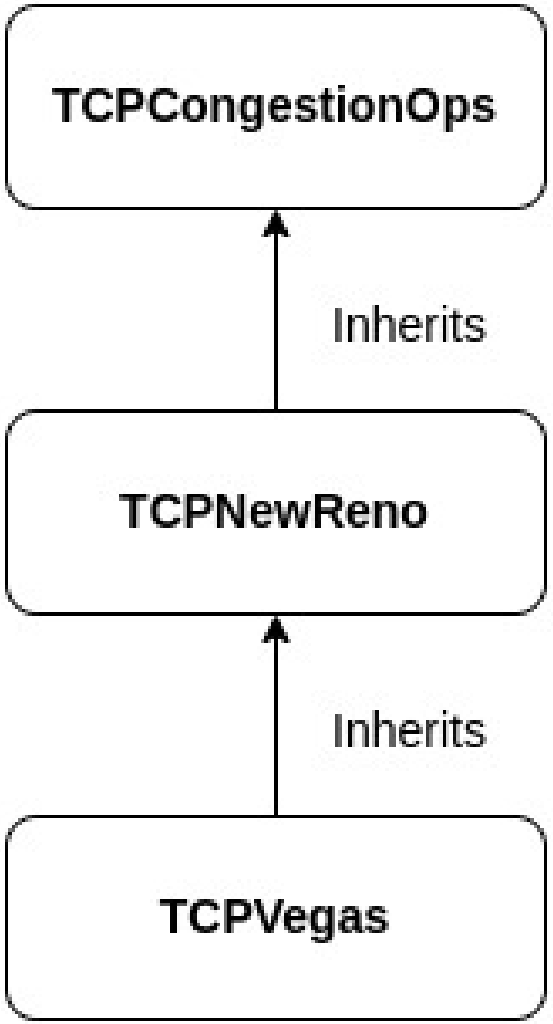
TCP Congestion Ops - Declaration

- It an **abstract class**.
- Most functions are either **pure virtual** or has **empty definition**
- The various congestion control algorithms that **inherit** TCP Congestion Ops implement the functions as required.

```
class TcpCongestionOps : public Object {
public:
  static TypeId GetTypeId (void)
  TcpCongestionOps ()
  virtual ~TcpCongestionOps ()
  virtual std::string GetName () const = 0

  virtual void Init (socketState) {}
  virtual uint32_t GetSsThresh (socketState) = 0
  virtual void IncreaseWindow (socketState, segmentsAcked)
  virtual void PktsAcked (socketState, segmentsAcked, rtt)
  virtual void CongestionStateSet (socketState, newState)
  virtual void CwndEvent (socketState, event)
  virtual bool HasCongControl () const
  virtual void CongControl ()
};
```

Inheritance Diagram

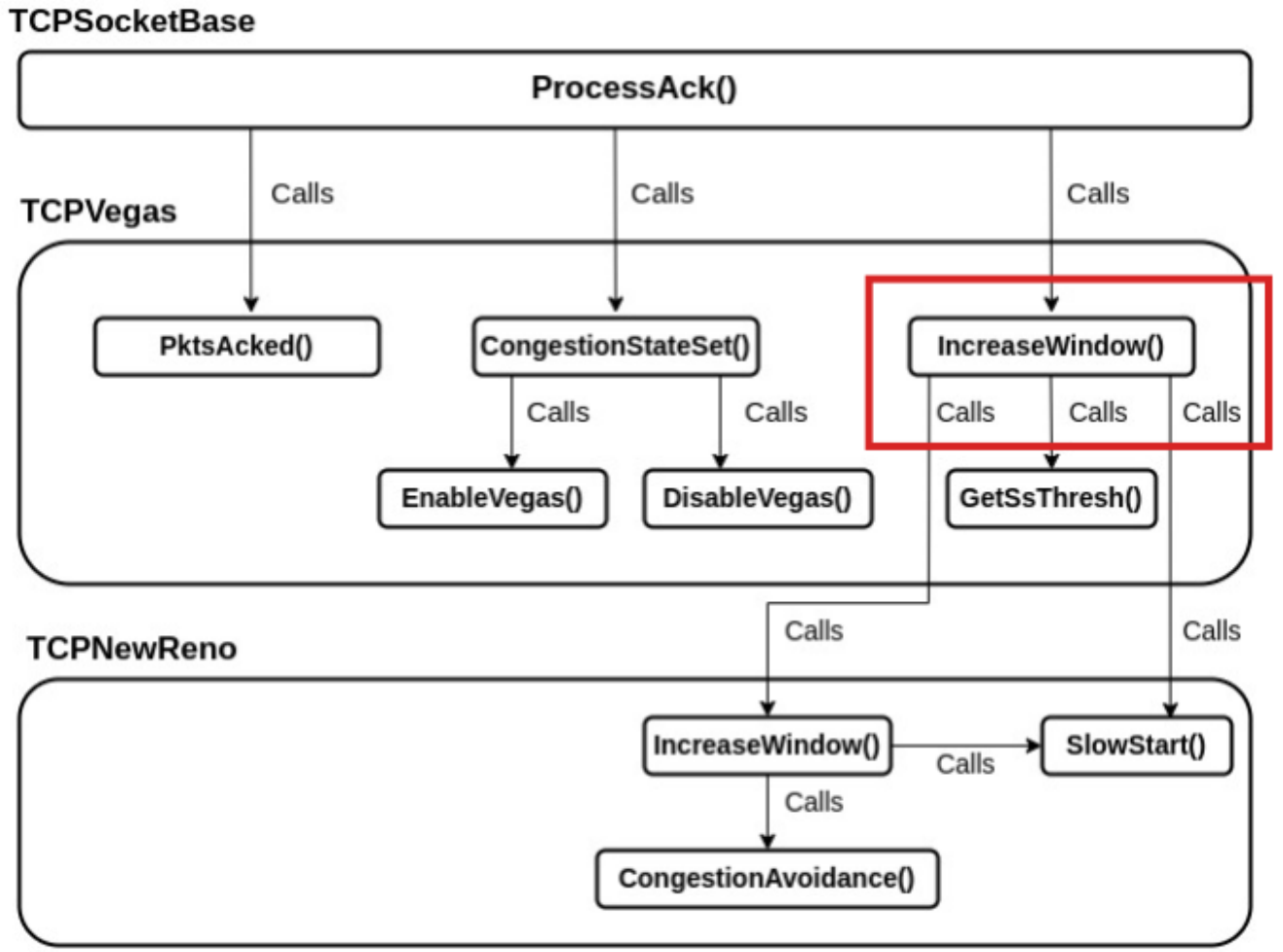


TCP Vegas Implementation

TcpVegas - Attributes

```
TypeId
TcpVegas::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::TcpVegas")
    .SetParent<TcpNewReno> ()
    .AddConstructor<TcpVegas> ()
    .SetGroupName ("Internet")
    .AddAttribute ("Alpha", "Lower bound of packets in network",
                   UintegerValue (2),
                   MakeUintegerAccessor (&TcpVegas::m_alpha),
                   MakeUintegerChecker<uint32_t> ())
    .AddAttribute ("Beta", "Upper bound of packets in network",
                   UintegerValue (4),
                   MakeUintegerAccessor (&TcpVegas::m_beta),
                   MakeUintegerChecker<uint32_t> ())
    .AddAttribute ("Gamma", "Limit on increase",
                   UintegerValue (1),
                   MakeUintegerAccessor (&TcpVegas::m_gamma),
                   MakeUintegerChecker<uint32_t> ())
  ;
  return tid;
}
```

TcpVegas - Function Call Sequence



TcpVegas :: PktsAcked

- called every time an ACK is received
- Contains timing information

```
void
TcpVegas::PktsAcked (Ptr<TcpSocketState> tcb, uint32_t segmentsAcked,
                     const Time& rtt)
{
  NS_LOG_FUNCTION (this << tcb << segmentsAcked << rtt);

  if (rtt.IsZero ())
    {
      return;
    }

  m_minRtt = std::min (m_minRtt, rtt);
  NS_LOG_DEBUG ("Updated m_minRtt = " << m_minRtt);

  m_baseRtt = std::min (m_baseRtt, rtt);
  NS_LOG_DEBUG ("Updated m_baseRtt = " << m_baseRtt);

  // Update RTT counter
  m_cntRtt++;
  NS_LOG_DEBUG ("Updated m_cntRtt = " << m_cntRtt);
}
```

Minimum of all RTT measurements within last RTT.

Minimum of all Vegas RTT measurements seen during connection

Number of RTT measurements during last RTT

TcpVegas :: CongestionStateSet

```
void
TcpVegas::CongestionStateSet (Ptr<TcpSocketState> tcb,
                              const TcpSocketState::TcpCongState_t newState)
{
  NS_LOG_FUNCTION (this << tcb << newState);
  if (newState == TcpSocketState::CA_OPEN)
    {
      EnableVegas (tcb);
    }
  else
    {
      DisableVegas ();
    }
}
```

m_doingVegasNow = true;

m_doingVegasNow = false;

TcpVegas :: IncreaseWindow

```
void
TcpVegas::IncreaseWindow (Ptr<TcpSocketState> tcb, uint32_t segmentsAcked)
{
  if ("Not vegas") TcpNewReno::IncreaseWindow (tcb, segmentsAcked);

  if ("Vegas cycle has finished") {
      if ("Not enough RTT samples") TcpNewReno::IncreaseWindow (tcb, segmentsAcked);
      else "enough RTT Samples"
        {
          // Calculate targetcwnd
          // Calculate diff = segcWnd - targetcwnd

          if ("Window size is increasing fast") /* Change to Linear Increase/Decrease Mode */

          else if ("Slow Start") TcpNewReno::SlowStart (tcb, segmentsAcked);

          else
            {     // Linear increase/decrease mode
              if ("too much packet in network")  // decrease cwnd
              else if ("too few packet in network")   //increase cwnd
              else  "All okay"
            }
          // update ssThresh
        }
      // Reset cntRtt & minRtt every RTT
    }
  else if ("Vegas cycle not finished")  TcpNewReno::SlowStart (tcb, segmentsAcked);
}
```

Congestion Avoidance in TCP Vegas

Implemented in IncreaseWindow()

diff Calculation

- continuously samples the RTT and computes the actual throughput and compares it with the expected throughput

```
double tmp = m_baseRtt.GetSeconds () / m_minRtt.GetSeconds ();
targetCwnd = static_cast<uint32_t> (segCwnd * tmp);
diff = segCwnd - targetCwnd;
```

Amount of extra packet queued at the bottleneck

Congestion Avoidance in TCP Vegas

Implemented in IncreaseWindow()

Three Conditions:

- Potential Congestion — diff > Gamma and cwnd < ssThresh
- No Congestion — diff < Gamma and cwnd < ssThreash
- Linear Increase/Decrease

Congestion Avoidance in TCP Vegas

Implemented in IncreaseWindow()

No Congestion

-> Slow Start

```
else if (tcb->m_cWnd < tcb->m_ssThresh)
  {     // Slow start mode
    TcpNewReno::SlowStart (tcb, segmentsAcked);
  }
```

Congestion Avoidance TCP Vegas

Implemented in IncreaseWindow()

Potential Congestion

-> Increase/Decrease Mode

```
if (diff > m_gamma && (tcb->m_cWnd < tcb->m_ssThresh))
  {
    // slow-start to linear increase/decrease mode
    segCwnd = std::min (segCwnd, targetCwnd + 1);
    tcb->m_cWnd = segCwnd * tcb->m_segmentSize;
    tcb->m_ssThresh = GetSsThresh (tcb, 0);
  }
```

Congestion Avoidance in TCP Vegas

Implemented in IncreaseWindow()

Increase/Decrease Mode

Linearly Decreases

```
if (diff > m_beta)
  {
    // We are going too fast, so we slow down
    segCwnd--;
    tcb->m_cWnd = segCwnd * tcb->m_segmentSize;
    tcb->m_ssThresh = GetSsThresh (tcb, 0);
  }
else if (diff < m_alpha)
  {
    // We are going too slow (having too little data in the network),
    // so we speed up.
    segCwnd++;
    tcb->m_cWnd = segCwnd * tcb->m_segmentSize;
  }
else
  {
    // We are going at the right speed
  }
```

Congestion Avoidance in TCP Vegas

Implemented in IncreaseWindow()

Increase/Decrease Mode

Linearly Increases 

```
if (diff > m_beta)
  {
    // We are going too fast, so we slow down
    segCwnd--;
    tcb->m_cWnd = segCwnd * tcb->m_segmentSize;
    tcb->m_ssThresh = GetSsThresh (tcb, 0);
  }
else if (diff < m_alpha)
  {
    // We are going too slow (having too little data in the network),
    // so we speed up.
    segCwnd++;
    tcb->m_cWnd = segCwnd * tcb->m_segmentSize;
  }
else
  {
    // We are going at the right speed
  }
```

Congestion Avoidance in TCP Vegas

Implemented in IncreaseWindow()

Increase/Decrease Mode

Does Nothing

```
if (diff > m_beta)
  {
    // We are going too fast, so we slow down
    segCwnd--;
    tcb->m_cWnd = segCwnd * tcb->m_segmentSize;
    tcb->m_ssThresh = GetSsThresh (tcb, 0);
  }
else if (diff < m_alpha)
  {
    // We are going too slow (having too little data in the network),
    // so we speed up.
    segCwnd++;
    tcb->m_cWnd = segCwnd * tcb->m_segmentSize;
  }
else
  {
    // We are going at the right speed
  }
```

TCP NewReno Implementation

TcpNewReno :: IncreaseWindow

TCPVegas:IncreaseWindow()

```
if (!m_doingVegasNow)
  {
    // If Vegas is not on, we follow NewReno algorithm
    NS_LOG_LOGIC ("Vegas is not turned on, we follow NewReno algorithm.");
    TcpNewReno::IncreaseWindow (tcb, segmentsAcked);
    return;
  }
```

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

  if (tcb->m_cWnd < tcb->m_ssThresh)
    {
      segmentsAcked = SlowStart (tcb, segmentsAcked);
    }

  if (tcb->m_cWnd >= tcb->m_ssThresh)
    {
      CongestionAvoidance (tcb, segmentsAcked);
    }
}
```

TcpNewReno :: SlowStart

```
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;
    }

  return 0;
}
```

TcpNewReno :: CongestionAvoidance

```
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);
    }
}
```

Thank You