QCN-SP Pseudo-code (Revision 1.0)

Extensions and modifications required to support QCN-SP (compared to QCN) are identified in **bold**.

Definitions and variables:

CP:	Congestion Point
RP:	Reaction Point
Frame:	a packet frame
Frame.length:	packet frame length
Frame.flowid:	a frame can be tagged with the field of its flow id.
RL[*]:	a set of rate limiters.
RL[i].state:	state of the rate limiter i: active or inactive.
RL[i].flowid:	the flow id that is associated with the rate limiter i.
RL[i].rate:	the speed of the rate limiter i.
RL[i].rd:	the amount of rate decrease in response to the last negative feedback frame (Fb < 0).
RL[i].tx_bcount:	number of bytes sent since the last negative feedback frame (Fb < 0).
RL[i].si_count:	the stage of self increase that the rate limiter,i, is in.
RL[i].timer:	the drift timer of the rate limiter
RL[i].qlen:	the queue length of the rate limiter queue
RL[i].CPID:	**CPID of CP associated with this RL.**
RL[i].cpMACAddress:	**MAC address of CP associated with this RL.**
rlidx:	index of a rate limiter.
FBFrame:	a feedback control frame which sends the congestion information, Fb, back to the traffic source
FBFrame.SA:	the source MAC address of the feedback control frame.
FBFrame.DA:	the destination MAC address of the feedback control frame.
FBFrame.flowid:	the flow id of the feedback control frame.
FBFrame.fb:	the congestion control information, Fb, of the feedback control frame.
FBFrame.CPID:	**the Congestion Point Identifier associated with a given CP**
FBFrame.opcode:	**PROBE: Probe packet from RP to CP; REQUEST: FB request packet from CP to RP**
qLen:	current queue length (in pages). incremented upon packet arrivals and decremented upon packet departures.
qLenOld:	queue length (in pages) at last sample.
sampleByteAcc:	Accumulated number of bytes sent on a queue since last check for sending an FB message was performed in the CP.
Fb:	feedback value which indicates the level of congestion.
qntz_Fb:	quantized negative Fb (-Fb) value.
cmDomainEdge:	Per-CP flag indicating if this node (queue) is at the edge of the CM domain. Used to drop CM packets at the edge of the domain

Constants and parameters:

C	line rate
QCN_MAX_INCREASE:	Maximum rate increase as fraction of line rate C

QCN_MAX_FB:	Maximum value for FB. Set to 64.
A:	Basic rate increase
MIN_DEC_FACTOR:	the minimum decrease factor, a single step of decrease should not exceed this value.
GD:	Rate decrease factor constant
MIN_RATE:	Minimum data rate
TIMER_PERIOD:	Drift timer period
DRIFT_FACTOR:	Drift factor
FAST_RECOVERY_TH:	Fast Recovery Threshold
MAC_ADDRESS:	The CP or RP MAC address
EXTRA_FAST_RECOVERY:	Flag to enable extra fast recovery
SUB_PATH_PROBE_ENABLED:	**Flag to enable Sub-path probes**
EFR_MAX:	Maximum value for EFR
HYPERACTIVE_INCREASE:	Flag to enable Hyperactive Increase
QEQ:	Qeq
W:	Weight for queue offset
SAMPLING_INTERVAL:	CP message sampling interval in bytes

Reaction Point:

```
void RP::initialize()
{
    FB_FACTOR = C * QCN_MAX_INCREASE / (A * QCN_MAX_FB);

    RL[*].state = INACTIVE;
    RL[*].flowid = -1;
    RL[*].rate = C;
    RL[*].Rd = 0;
    RL[*].tx_bcount = 0;
    RL[*].si_count = -1;
}

void RP::processQcnMessage(FBFrame)
{
    rlidx = getRateLimiterIndex(FBFrame);

    if (RL[rlidx].state == INACTIVE) {
          if (FBFrame.fb > 0) {
            RL[rlidx].state = ACTIVE;
            RL[rlidx].rate = C;
            RL[rlidx].efr_count = 0;
            RL[rlidx].si_count = -1;
            RL[rlidx].Rd = 0;
            RL[rlidx].tx_bcount = 0;
            RL[rlidx].flowid = FBFrame.flowid;
            RL[rlidx].cpMACAddress = FBFrame.SA;
            RL[rlidx].CPID = FBFrame.CPID;
          } else {
            return;
          }
    }

    if (FBFrame.fb > 0) {     // Negative feedback (request to reduce rate)
```

```
      if (RL[rlidx].CPID != FBFrame.CPID) {
          RL[rlidx].CPID = FBFrame.CPID;
          RL[rlidx].cpMACAddress = FBFrame.SA;
      }
      RL[rlidx].flowid = FBFrame.flowid;
      // multiplicative decrease
      dec_factor = (1.0 - GD * FBFrame.fb);
      if (dec_factor < MIN_DEC_FACTOR) {
          dec_factor = MIN_DEC_FACTOR;
      }
      oldrate = RL[rlidx].rate;
      RL[rlidx].rate = RL[rlidx].rate * dec_factor;
      if (RL[rlidx].rate < MIN_RATE) {
          RL[rlidx].rate = MIN_RATE;
      }
      // store rate decrease
      Rd = oldrate - RL[rlidx].rate;

      if (EXTRA_FAST_RECOVERY) {
          // store cumulative rate decrease
          if (RL[rlidx].si_count) {
            // beginning of new cycle
            RL[rlidx].Rd = Rd;
            RL[rlidx].efr_count = 0;
          } else if (RL[rlidx].efr_count < EFR_MAX) {
            RL[rlidx].Rd = RL[rlidx].Rd + Rd;
            RL[rlidx].efr_count++;
          }
      } else {
          RL[rlidx].Rd = Rd;
      }
      RL[rlidx].si_count = 0;

      // Do not reset tx_bcount with probing
      // if (!EXTRA_FAST_RECOVERY || RL[rlidx].si_count != 0) {
      //    RL[rlidx].tx_bcount = 0;
      // }
      // reset drift timer
      setDriftTimer(rlidx, TIMER_PERIOD);
    } else {
      if (RL[rlidx].CPID == FBFrame.CPID) {
          // positive reinforcement (probe response)
          RL[rlidx].si_count++;
          // Since this is the response to a probe, don't reset tx_bcount.
          // RL[rlidx].tx_bcount = 0;
          self_increase(rlidx, -FBFrame.fb);
      }
    }
}

void RP::self_increase(rlidx, Fb)
{
    Ri = 0;
    to_count = RL[rlidx].si_count;

    if (RL[rlidx].si_count + Fb <= FAST_RECOVERY_TH) {
      // fast recovery
```

```
      // Ri = RL[rlidx].Rd / (2 ^ to_count);
      if (to_count >= 0)
          Ri = RL[rlidx].Rd / (1 << to_count);
      else
          Ri = RL[rlidx].Rd * 2;    // 2^-1 = 1/2
    } else {
      // active increase
      if (HYPERACTIVE_INCREASE) {
          Ri = qcnA * (Fb * FB_FACTOR + (to_count - FAST_RECOVERY_TH));
      } else {
          Ri = qcnA * (Fb * FB_FACTOR + 1);
      }
    }
    // Limit rate increase
    if (Ri > C * QCN_MAX_INCREASE)
      Ri = C * QCN_MAX_INCREASE;
    RL[rlidx].rate = RL[rlidx].rate + Ri;
    // saturate rate at C
    if (RL[rlidx].rate > C) {
      RL[rlidx].rate = C;
    }
}

void RP::transmit(Frame, rlidx)
{
    if (RL[rlidx].qlen == 0 && RL[rlidx].rate >= C) {
      RL[rlidx].state = INACTIVE;
      RL[rlidx].flowid = -1;
      RL[rlidx].rate = C;
      RL[rlidx].tx_bcount = 0;
      RL[rlidx].si_count = -1;
      RL[rlidx].CPID = 0;
    } else {
        RL[rlidx].tx_bcount += Frame.length;
      // If a negative FBFrame has not been received after transmitting
      // TO_THRESH bytes, send probe and trigger self_increase
        if (RL[rlidx].tx_bcount > TO_THRESH) {
          sendProbe(rlidx);
          RL[rlidx].si_count++;
          RL[rlidx].tx_bcount = 0;
          self_increase(rlidx, 0);
      }
    }
    send(Frame);
}

void RP::sendProbe(rlidx)
{
    FBFrame.opcode = PROBE;
    FBFrame.CPID = RL[rlidx].CPID;
    FBFrame.fb = -MAX_FB;
    FBFrame.DA = RL[rlidx].cpMACAddress;
    FBFrame.SA = MAC_ADDRESS;
    FBFrame.flowid = RL[rlidx].flowid;

    send(FBFrame);
}
```

```
void RP::drift_timer_expired(rlidx)
{
    if (RL[rlidx].state == ACTIVE) {
      RL[rlidx].rate = RL[rlidx].rate * DRIFT_FACTOR;
      if (RL[rlidx].rate > C) {
          RL[rlidx].rate = C;
      }
      setDriftTimer(rlidx, TIMER_PERIOD);
    }
}
```

Congestion Point:

```
void CP::initialize()
{
    qLen = 0;
    qLenOld = 0;
    sampleByteAcc = 0;
    samplingInterval = SAMPLING_INTERVAL;
}

int CP::calculateFb(isProbe)
{
    if (qLen || qLenOld)
        Fb = (QEQ - qLen) - W * (qLen - qLenOld);
    else                // Create maximim positive Fb if cQlen == QlenOld == 0
        Fb = QEQ * (2*W + 1);

    if (Fb < -QEQ * (2*W + 1)) {
      Fb = -QEQ * (2*W + 1);
    } else if (Fb > 0) {
      if (isProbe) {
          // If this is a probe, also calculate positive feedback.
          Fb = min(Fb, QEQ * (2*W + 1));
      } else {
          Fb = 0;
      }
    }
    return -Fb * ((2<<5)-1) / ((2*W+1)*QEQ);    // Quantization to +/- 64
}

// Probe message handler. This is for probes directed to the CP only.
// Assumption: probe is already associated with a given queue.
// Probe will be dropped or replied to.
// Note:
//    Sub-path probes are handled by handleMessage().
//
void CP::handleProbe(FBFrame)
{
    qntzFb = calculateFb(true);
    if (qntzFb < 0) {   // Only react to directed probe if feedback is positive
      FBFrame.fb = qntz_Fb;
      FBFrame.DA = FBFrame.SA;       // back to sender
      FBFrame.SA = MAC_ADDRESS;
      FBFrame.opcode = REQUEST;
```

```
      send(FBFrame);
    } else {
      drop(FBFrame);
    }
}

// Packet message handler.
// Assumes message is classified and associated with a given queue.
//
void CP::handleMessage(Frame)
{
    isCmPacket = (Frame.opcode == REQUEST || Frame.opcode == PROBE);

    if (isCmPacket) {
      if (Frame.opcode == PROBE) {
        if (Frame.CPID == CPID) {
          handleProbe(Frame);
          return;
        }
        if (SUB_PATH_PROBE_ENABLED) {
          qntz_Fb = calculateFb(true);

          if (qntz_Fb >= 0) {
            drop(Frame);
            return;
          } else if (qntz_Fb > Frame.fb) {
            // Update Fb in probe packet:
            // set minimal positive increase
            Frame.fb = qntz_Fb;
            }
          }
        }
      }
      // Drop CM packets at CM domain edge.
      if (cmDomainEdge)
        drop(Frame);
        return;
    }

    // Do not use the QCN style probability approach to generate QCN packets,
    // but use a dynamic sampling rate based approach instead.
    //
    // We do not create a CM packet as response to a QCN packet,
    // but count the packet into the interval.
    //
    generateQcnFrame = false;
    sampleByteAcc += Frame.length;
    if (sampleByteAcc >= samplingInterval && !isCmPacket) {
      sampleByteAcc = 0;

      qntz_Fb = calculateFb(false);

      // Generate new sampling interval base on current load.
      samplingInterval = SAMPLING_INTERVAL * 7 / (7 + qntz_Fb);
            // (7 / (7 + qntz_Fb)) creates a range of 7/70 .. 7/7
            // ie 1/10 to 1, meaning the sampling interval is reduced
            // up to 10-fold with the highest possible level of congestion.
```

```
            // There are up to ten times as many samples in that case.
            // This is similar to the probability calculation of 1..10%
            // of packets with a packet size of 1,500 bytes, but does
            // not depend on packet size, nor require a calculation
            // with each packet. Re-calculation instead occurs with
            // each sample.
            // A real implementation would probably use a table based
            // approach to reduce HW complexity.
            //
            // Note that the sampling interval should be randomized
            // (e.g., use +/- 10-20% of the calculated interval),
            // but we omit that here for simplification.
      if (qntz_Fb) {
          generateQcnFrame = true;
      }
      qLenOld = qLen;
    }

    if (generateQcnFrame) {
      FBFrame.CPID = CPID;                // CPID is the queue's CPID
      FBFrame.fb = qntz_Fb;
      FBFrame.DA = Frame.SA;
      FBFrame.SA = MAC_ADDRESS;
      FBFrame.flowid = Frame.flowid;
      FBFrame.opcode = REQUEST;
      send(FBFrame);
    }
    send(Frame);
}
```