[1] Definition

1. **IncomingFrame**: a packet frame which arrives at a congestion node or at its destination.
2. **IncomingFrame.flowid**: an incoming frame can be tagged with the field of its flow id.
3. **IncomingFrame.DE**: an incoming frame is assumed to be tagged with a Discard Eligible (DE) bit which is initialized to 0; intermediate congestion points in the path of this frame can modify the field.
4. **RL[*]**: a set of rate limiters.
5. **RL[i].state:** state of the rate limiter *i*: active or inactive.
6. **RL[i].flowid:** the flow id that is associated with the rate limiter *i*.
7. **RL[i].crate:** the current rate of the rate limiter *i*.
8. **RL[i].trate:** the target rate of the rate limiter *i*.
9. **RL[i].tx_bcount:** number of bytes sent since the last negative feedback frame (Fb < 0).
10. **RL[i].si_count:** the stage of the byte counter that the rate limiter,*i*, is in.
11. **RL[i].timer:** the timer of the rate limiter
12. **RL[i].timer_scount:** the stage of the timer that the rate limiter,*i*, is in.
13. **RL[i].qlen:** the queue length of the rate limiter queue
14. **rlidx**: index of a rate limiter.
15. **FBFrame:** a feedback control frame which sends the congestion information, Fb, back to the traffic source; this packet frame can be sent either from any intermediate reflection point.
16. **FBFrame.SA:** the source MAC address of the feedback control frame.
17. **FBFrame.DA:** the destination MAC address of the feedback control frame.
18. **FBFrame.flowid:** the flow id of the feedback control frame.
19. **FBFrame.fb:** the congestion control information, Fb, of the feedback control frame.
20. **min_dec_factor:** the minimum decrease factor, a single step of decrease should not exceed this value.
21. **qlen**: current queue length (in pages). incremented upon packet arrivals and decremented upon packet departures.
22. **qlen_old**: queue length (in pages) at last sample.
23. **Fb**: feedback value which indicates the level of congestion.
24. **qntz_Fb**: quantized negative Fb (-Fb) value.

[1] EDCS-608482

QCN Reaction Point:

```
initialize()
{
    //* indicates all rate limiters
    RL[*].state = INACTIVE;
    RL[*].flowid = −1;
    RL[*].crate = C;
    RL[*].trate = C;
    RL[*].tx_bcount = 0;
    RL[*].si_count = 0;
    RL[*].timer_count = 0;
}

foreach (FBFrame)
{
    //obtain the rate limiter index that is associated with a flowid
    //if no match, return the index of the next available rate limiter
    rlidx = get_rate_limiter_index(FBFrame.flowid);

    if (RL[rlidx].state = = INACTIVE) then
        if (FBFrame.fb != 0) then
            //initialize new rate limiter
            RL[rlidx].state = ACTIVE;
            RL[rlidx].flowid = FBFrame.flowid;
            RL[rlidx].crate = C;
            RL[rlidx].trate = C;
            RL[rlidx].si_count = 0;
        else
            //ignore FBFrame
            return;
        endif
    endif
```

```
        if (FBFrame.fb != 0) then
            // use the current rate as the next target rate.
            // however under EXTRA_FAST_RECOVERY  mode:
            // in the first cycle of fast recovery,
            // the Fb < 0  signal would not reset the target rate.
            if (EXTRA_FAST_RECOVERY != TRUE
                || RL[rlidx].si_count ! = 0) then
                    RL[rlidx].trate = RL[rlidx].crate;
                    RL[rlidx].tx_bcount = 0;
            endif
            // set the stage counter
            RL[rlidx].si_count = 0;
            RL[rlidx].timer_scount = 0;
            // update the current rate, multiplicative decrease
            dec_factor = (1 − GD * FBFrame.fb);
            if (dec_factor < min_dec_factor) then
                    dec_factor = min_dec_factor;
            endif
            RL[rlidx].crate = RL[rlidx].crate * (1 − dec_factor);
            if (RL[rlidx].crate < MIN_RATE) then
                    RL[rlidx].crate = MIN_RATE;
            endif
            //reset the timer
            set_timer(rlidx, TIMER_PERIOD);
        endif
}

self_increase(rlidx)
{
        to_count = minimum(RL[rlidx].si_count, RL[rlidx].timer_scount);
        // if in the active probing stages, increase the target rate
        if (RL[rlidx].si_count > FAST_RECOVERY_TH ||
            RL[rlidx].timer_scount > FAST_RECOVERY_TH) then
            if (RL[rlidx].si_count > FAST_RECOVERY_TH &&
                RL[rlidx].timer_scount > FAST_RECOVERY_TH) then
                    //hyperactive increase
                    Ri = B * (to_count − FAST_RECOVERY_TH);
            else
                    //active increase
                    Ri = A;
            endif
        else
```

```
                Ri = 0;
        endif



        //at the end of the first cycle of recovery
        if (EXTR_FAST_RECOVERY  && RL[rlidx].si_count == 1 &&
           RL[rlidx].trate > 10* RL[rlidx].crate) then
                RL[rlidx].trate = RL[rlidx].trate/8;
        else
                RL[rlidx].trate = RL[rlidx].trate + Ri;

        RL[rlidx].crate = (RL[rlidx].trate + RL[rlidx].crate)/2;

        //saturate rate at C
        if (RL[rlidx].crate > C) then
              RL[rlidx].crate = C;
        endif
}

foreach (Transmit Frame))
{
        //release the rate limiter when its rate has reached C
        //and its associated queue is empty
        if ( RL[rlidx].rate = = C && RL[rlidx].qlen = = 0) then
            RL[rlidx].state = INACTIVE;
            RL[rlidx].flowid = −1;
            RL[rlidx].crate = C;
            RL[rlidx].trate = C;
            RL[rlidx].tx_bcount = 0;
            RL[rlidx].si_count = 0;
            RL[rlidx].timer = INACTIVE;
        else
              RL[rlidx].tx_bcount += length(Transmit Frame);
              //if a negative FBframe has not been received after transmitting
              //TO_THRESH bytes, trigger self_increase
             if (RL[rlidx].si_bcount < FAST_RECOVERY_TH) then
                   expire_thresh = TO_THRESH;
            else
                   expire_thresh = TO_THRESH/2;
             endif
             if (RL[rlidx].tx_bcount > expire_thresh) then
                   RL[rlidx].si_count++;
                   RL[rlidx].tx_bcount = 0;
                   self_increase(rlidx);
             endif
        endif
}
```

```
/*  Timers */
timer_expired(rlidx)
{
        if (RL[rlidx].state = = ACTIVE ) then
              RL[rlidx].timer_scount++;;
              self_increase(rlidx);

              //reset the  timer

              if (RL[rlidx].timer_scount < FAST_RECOVERY_TH) then
                      expire_period = TO_THRESH;
              else
                      expire_period = TO_THRESH/2;
              endif
              set _timer(rlidx, expire_period);

        endif
}
```

QCN Congestion Point:

```
initialize()
{
        qlen = 0;
        qlen_old = 0;
}

foreach (IncomingFrame)
{
        //calculate Fb value
        Fb = (Q_EQ − qlen) − W * (qlen − qlen_old);
        if (Fb < −Q_EQ * (2 * W + 1)) then
                Fb = −Q_EQ * (2 * W + 1);
        elseif (Fb > 0) then
                Fb = 0;
        endif

        //the maximum value of −Fb determines the number of bits that Fb uses.
        //uniform quantization of −Fb, qntz_Fb, uses most significant bits of –Fb.
        //note that now qntz_Fb has positive values.
        qntz_Fb = −Fb(most significant bits);

        //sampling probability is a function of Fb
        generate_fb_frame = 0;
        if (urand() < (BASE_PROBABILITY + C * qntz_Fb)) then
                //generate a feedback frame if Fb is negative
                if (Fb < 0) then
                        generate_fb_frame = 1;
                endif
                qlen_old = qlen;
        endif

        //set DE bit if Fb is negative
        if (Fb < 0) then
                IncomingFrame.DE = 1;
        endif

        if (generate_fb_frame) then
                FBFrame.DA = IncomingFrame.SA;
                FBFrame.SA = SWITCH_MAC_ADDRESS;
                FBFrame.flowid = IncomingFrame.flowid;
                FBFrame.fb = qntz_Fb;
                forward(FBFrame);
        endif
}
```