## 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].rate**: the current rate of the rate limiter \( i \).
8. **RL[i].rate**: the target rate of the rate limiter \( i \).
9. **RL[i].tx_bcount**: number of bytes sent since the last negative feedback frame \((F_b < 0)\).
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_scounr**: 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, \( F_b \), 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, \( F_b \), of the feedback control frame.
20. **min_dec_factor**: the minimum decrease factor, a single step of decrease should not exceed this value.
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 \( F_b \) \((-F_b)\) value.

---

1 EDCS-608482
QCN Reaction Point:

1. initialize()
2. {
3.     /* indicates all rate limiters */
4.     RL[*].state = INACTIVE;
5.     RL[*].flowid = -1;
6.     RL[*].crate = C;
7.     RL[*].trate = C;
8.     RL[*].tx_bcoun = 0;
9.     RL[*].si_count = 0;
10.    RL[*].timer_count = 0;
11. }
12.
13. foreach (FBFrame)
14. {
15.     //obtain the rate limiter index that is associated with a flowid
16.     //if no match, return the index of the next available rate limiter
17.     rldx = get_rate_limiter_index(FBFrame.flowid);
18.     if (RL[rldx].state == INACTIVE) then
19.         if (FBFrame.fb != 0) then
20.             //initialize new rate limiter
21.             RL[rldx].state = ACTIVE;
22.             RL[rldx].flowid = FBFrame.flowid;
23.             RL[rldx].crate = C;
24.             RL[rldx].trate = C;
25.             RL[rldx].si_count = 0;
26.         else
27.             //ignore FBFrame
28.             return;
29.         endif
30.     endif
31. }
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].crate = 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
        // else
    endif
}
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;
endif

// 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_bcoun = 0;
        RL[rlidx].si_count = 0;
        RL[rlidx].timer = INACTIVE;
    else
        RL[rlidx].tx_bcoun += length(Transmit Frame);
        // if a negative FBframe has not been received after transmitting
        // TO_THRESH bytes, trigger self_increase
        if (RL[rlidx].si_bcoun < FAST_RECOVERY_TH) then
            expire_thresh = TO_THRESH;
        else
            expire_thresh = TO_THRESH/2;
        endif
        if (RL[rlidx].tx_bcoun > expire_thresh) then
            RL[rlidx].si_count++;
            RL[rlidx].tx_bcoun = 0;
            self_increase(rlidx);
        endif
    endif
}
127. /* Timers */
128. timer_expired(rlidx)
129. {
130.   if (RL[rlidx].state == ACTIVE) then
131.     RL[rlidx].timer_scourt++;   
132.     self_increase(rlidx);
133.   
134.     //reset the timer
135.   
136.   if (RL[rlidx].timer_scourt < FAST_RECOVERY_TH) then
137.     expire_period = TO_THRESH;
138.   else
139.     expire_period = TO_THRESH/2;
140.   endif
141.   set_timer(rlidx, expire_period);
142.  
143.  endif
144. }
QCN Congestion Point:

```c
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
        FBFram.DA = IncomingFrame.SA;
        FBFram.SA = SWITCH_MAC_ADDRESS;
        FBFram.flowid = IncomingFrame.flowid;
        FBFram.fb = qntz_Fb;
        forward(FBFram);
    endif
}
```