[1] Definition - Variables

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. **RL[*]**: a set of rate limiters.
4. **RL[i].state:** state of the rate limiter *i*: active or inactive.
5. **RL[i].flowid:** the flow id that is associated with the rate limiter *i*.
6. **RL[i].crate:** the current rate of the rate limiter *i*.
7. **RL[i].trate:** the target rate of the rate limiter *i*.
8. **RL[i].tx_bcount:** number of bytes left before increasing the stage of the byte counter.
9. **RL[i].si_count:** the stage of the byte counter that the rate limiter, *i*, is in.
10. **RL[i].timer:** the timer of the rate limiter
11. **RL[i].timer_scount:** the stage of the timer that the rate limiter, *i*, is in.
12. **RL[i].qlen:** the queue length of the rate limiter queue
13. **rlidx**: index of a rate limiter.
14. **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.
15. **FBFrame.SA:** the source MAC address of the feedback control frame.
16. **FBFrame.DA:** the destination MAC address of the feedback control frame.
17. **FBFrame.flowid:** the flow id of the feedback control frame.
18. **FBFrame.fb:** the congestion control information, Fb, of the feedback control frame.
19. **qlen**: current queue length (in pages). incremented upon packet arrivals and decremented upon packet departures.
20. **qlen_old**: queue length (in pages) at last sample.
21. **Fb**: feedback value which indicates the level of congestion.
22. **qntz_Fb**: quantized negative Fb (-Fb) value.
23. **time_to_mark**: number of bytes left before the next sample will be taken

[1] EDCS-608482

Definition – Parameters

24. **Q_EQ**: the reference point of a queue. QCN aims to keep the queue occupancy at this reference level under congestion.
25. **W:** the control parameter in calculating the congestion level variable Fb.
26. **GD:** the control gain parameter which determines the level of rate decrease given a Fb < 0 signals.
27. **BC_LIMIT**: the parameter which determines the byte-counter time-out threshold.
28. **TIMER_PERIOD**: the parameter which determines the timer time-out threshold.
29. **R_AI:** the parameter which determines the rate increase amount in AI stage.
30. **R_HAI:** the parameter which determines the rate increase amount in HAI stage.
31. **FAST_RECOVERY_TH:** the threshold which determines when a RL will exit fast recovery (FR) stage, set to 5.
32. **MIN_RATE:** the minimum rate of a rate limiter, set to 10Mbps.
33. **MIN_DEC_FACTOR:** the minimum rate decrease factor, set to 0.5.
34. **C:** the speed of a link where a rate limiter is installed
35. **SWITCH_MAC_ADDRESS:** the congestion point MAC address which is used as SA in the feedback frame

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_bcount = BC_LIMIT;
9.        RL[*].si_count = 0;
10.       RL[*].timer_scount = 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.       rlidx = get_rate_limiter_index(FBFrame.flowid);

18.       if (RL[rlidx].state = = INACTIVE) then
19.           if (FBFrame.fb != 0) then
20.               //initialize new rate limiter
21.               RL[rlidx].state = ACTIVE;
22.               RL[rlidx].flowid = FBFrame.flowid;
23.               RL[rlidx].crate = C;
24.               RL[rlidx].trate = C;
25.               RL[*].tx_bcount = BC_LIMIT;
26.               RL[rlidx].si_count = 0;
27.           else
28.               //ignore FBFrame
29.               return;
30.           endif
31.       endif
32.
```

```
33.          if (FBFrame.fb != 0) then
34.
35.              // use the current rate as the next target rate.
36.              // in the first cycle of fast recovery,
37.              // the Fb < 0  signal would not reset the target rate.
38.              if (RL[rlidx].si_count ! = 0) then
39.                   RL[rlidx].trate = RL[rlidx].crate;
40.                   RL[rlidx].tx_bcount = BC_LIMIT;
41.              endif
42.
43.              // set the stage counter
44.              RL[rlidx].si_count = 0;
45.              RL[rlidx].timer_scount = 0;
46.
47.
48.              // update the current rate, multiplicative decrease
49.               dec_factor = (1 − GD * FBFrame.fb);
50.               if (dec_factor < MIN_DEC_FACTOR) then
51.                    dec_factor = MIN_DEC_FACTOR;
52.               endif
53.               RL[rlidx].crate = RL[rlidx].crate * dec_factor;
54.               if (RL[rlidx].crate < MIN_RATE) then
55.                    RL[rlidx].crate = MIN_RATE;
56.               endif
57.
58.               //reset the timer
59.               set_timer(rlidx, TIMER_PERIOD);
60.          endif
61.     }

62.     self_increase(rlidx)
63.     {
64.          to_count = minimum(RL[rlidx].si_count, RL[rlidx].timer_scount);
65.
66.          // if in the active probing stages, increase the target rate
67.          if (RL[rlidx].si_count > FAST_RECOVERY_TH ||
68.              RL[rlidx].timer_scount > FAST_RECOVERY_TH) then
69.               if (RL[rlidx].si_count > FAST_RECOVERY_TH &&
70.                   RL[rlidx].timer_scount > FAST_RECOVERY_TH) then
71.                    //hyperactive increase
72.                    Ri = R_HAI * (to_count − FAST_RECOVERY_TH);
73.               else
74.                    //active increase
75.                    Ri = R_AI;
76.               endif
77.          else
78.                    Ri = 0;
79.          endif
```

```
80.
81.          //at the end of the first cycle of recovery
82.          if ((RL[rlidx].si_count == 1 || RL[rlidx].timer_scount == 1) &&
83.               RL[rlidx].trate > 10* RL[rlidx].crate) then
84.                    RL[rlidx].trate = RL[rlidx].trate/8;
85.          else
86.                    RL[rlidx].trate = RL[rlidx].trate + Ri;
87.          endif
88.
89.          RL[rlidx].crate = (RL[rlidx].trate + RL[rlidx].crate)/2;
90.
91.          //saturate rate at C
92.          if (RL[rlidx].crate > C) then
93.                RL[rlidx].crate = C;
94.          endif
95.     }
96.
97.     foreach (Transmit Frame))
98.     {
99.          //release the rate limiter when its rate has reached C
100.         //and its associated queue is empty
101.         if ( RL[rlidx].crate = = C && RL[rlidx].qlen = = 0) then
102.             RL[rlidx].state = INACTIVE;
103.             RL[rlidx].flowid = −1;
104.             RL[rlidx].crate = C;
105.             RL[rlidx].trate = C;
106.             RL[rlidx].tx_bcount = BC_LIMIT;
107.             RL[rlidx].si_count = 0;
108.             RL[rlidx].timer = INACTIVE;
109.         else
110.             RL[rlidx].tx_bcount  - = length(Transmit Frame);
111.
112.
113.            if (RL[rlidx].tx_bcount < 0) then
114.                    RL[rlidx].si_count++;
115.                    //if a negative FBframe has not been received after transmitting
116.                    //BC_LIMIT bytes, trigger self_increase; margin of randomness 30%
117.                    if (RL[rlidx].si_count < FAST_RECOVERY_TH) then
118.                      expire_thresh = random_number_betwen(0.85,1.15)*BC_LIMIT;
119.                    else
120.                     expire_thresh = random_number_betwen(0.85,1.15)*BC_LIMIT/2;
121.                    endif
122.
123.                    RL[rlidx].tx_bcount = expire_thresh;
124.                    self_increase(rlidx);
125.            endif
126.         endif
127.    }
```

```
128.  /*  Timers */
129.  timer_expired(rlidx)
130.  {
131.        if (RL[rlidx].state = = ACTIVE ) then
132.              RL[rlidx].timer_scount++;;
133.              self_increase(rlidx);
134.
135.              //reset the  timer
136.              //margin of randomness 30%
137.              if (RL[rlidx].timer_scount < FAST_RECOVERY_TH) then
138.                    expire_period = random_number_betwen(0.85,1.15)*TIMER_PERIOD;
139.              else
140.                    expire_period = random_number_betwen(0.85,1.15)*TIMER_PERIOD /2;
141.              endif
142.              set _timer(rlidx, expire_period);
143.
144.        endif
145.  }
```

QCN Congestion Point:

```
146.  initialize()
147.  {
148.        qlen = 0;
149.        qlen_old = 0;
150.        time_to_mark = Mark_Table(0);
151.  }
152.
153.  foreach (IncomingFrame)
154.  {
155.        //calculate Fb value
156.       Fb = (Q_EQ − qlen) − W * (qlen − qlen_old);
157.       if (Fb < −Q_EQ * (2 * W + 1)) then
158.            Fb = −Q_EQ * (2 * W + 1);
159.       elseif (Fb > 0) then
160.            Fb = 0;
161.       endif
162.
163.       //the maximum value of −Fb determines the number of bits that Fb uses.
164.       //uniform quantization of −Fb, qntz_Fb, uses most significant bits of –Fb.
165.       //note that now qntz_Fb has positive values.
166.       qntz_Fb = −Fb(most significant bits);
167.
168.       //sampling probability is a function of Fb
169.       generate_fb_frame = 0;
170.
171.       time_to_mark - = length(IncomingFrame);
172.       if (time_to_mark < 0) then
173.            //generate a feedback frame if Fb is negative
174.            if (Fb < 0) then
175.                 generate_fb_frame = 1;
176.            endif
177.            qlen_old = qlen;
178.            //Mark Table is described below. Margin of randomness 30%
179.            next_period  =  Mark_Table(qntz_Fb);
180.            time_to_mark = random_number_between(0.85,1.15)*next_period;
181.       endif
182.
183.       if (generate_fb_frame) then
184.            FBFrame.DA = IncomingFrame.SA;
185.            FBFrame.SA = SWITCH_MAC_ADDRESS;
186.            FBFrame.flowid = IncomingFrame.flowid;
187.            FBFrame.fb = qntz_Fb;
188.            forward(FBFrame);
189.       endif
```

```
190.    }
191.
192.
193.    //assuming 6 bits of quantization
194.    Mark_Table(qntz_Fb) {
195.
196.          switch (qntz_Fb/8){
197.            case 0: return 150KB;
198.            case 1: return 75KB;
199.            case 2: return 50KB;
200.            case 3: return 37.5KB;
201.            case 4: return 30KB;
202.            case 5: return 25KB;
203.            case 6: return 21.5KB;
204.            case 7: return 18.5KB;
205.        }
206.    }
```