Linux Cross Reference

Free Electrons

Embedded Linux Experts

Linux/net/ipv4/tcp_cubic.c

```
 1 /*
 2  * TCP CUBIC: Binary Increase Congestion control for TCP v2.3
 3  * Home page:
 4  *      http://netsrv.csc.ncsu.edu/twiki/bin/view/Main/BIC
 5  * This is from the implementation of CUBIC TCP in
 6  * Sangtae Ha, Injong Rhee and Lisong Xu,
 7  *  "CUBIC: A New TCP-Friendly High-Speed TCP Variant"
 8  *  in ACM SIGOPS Operating System Review, July 2008.
 9  * Available from:
10  *  http://netsrv.csc.ncsu.edu/export/cubic_a_new_tcp_2008.pdf
11  *
12  * CUBIC integrates a new slow start algorithm, called HyStart.
13  * The details of HyStart are presented in
14  *  Sangtae Ha and Injong Rhee,
15  *  "Taming the Elephants: New TCP Slow Start", NCSU TechReport 2008.
16  * Available from:
17  *  http://netsrv.csc.ncsu.edu/export/hystart_techreport_2008.pdf
18  *
19  * All testing results are available from:
20  * http://netsrv.csc.ncsu.edu/wiki/index.php/TCP_Testing
21  *
22  * Unless CUBIC is enabled and congestion window is large
23  * this behaves the same as the original Reno.
24  */
25
26 #include <linux/mm.h>
27 #include <linux/module.h>
28 #include <linux/math64.h>
29 #include <net/tcp.h>
30
31 #define BICTCP_BETA_SCALE    1024       /* Scale factor beta calculation
32                                          * max_cwnd = snd_cwnd * beta
33                                          */
34 #define BICTCP_HZ               10      /* BIC HZ 2^10 = 1024 */
35
36 /* Two methods of hybrid slow start */
37 #define HYSTART_ACK_TRAIN       0x1
38 #define HYSTART_DELAY           0x2
39
40 /* Number of delay samples for detecting the increase of delay */
41 #define HYSTART_MIN_SAMPLES     8
42 #define HYSTART_DELAY_MIN       (4U<<3)
43 #define HYSTART_DELAY_MAX       (16U<<3)
44 #define HYSTART_DELAY_THRESH(x) clamp(x, HYSTART_DELAY_MIN, HYSTART_DELAY_MAX)
45
46 static int fast_convergence __read_mostly = 1;
47 static int beta __read_mostly = 717;    /* = 717/1024 (BICTCP_BETA_SCALE) */
48 static int initial_ssthresh __read_mostly;
49 static int bic_scale __read_mostly = 41;
50 static int tcp_friendliness __read_mostly = 1;
51
```

```
 52 static int hystart __read_mostly = 1;
 53 static int hystart_detect __read_mostly = HYSTART_ACK_TRAIN | HYSTART_DELAY;
 54 static int hystart_low_window __read_mostly = 16;
 55 static int hystart_ack_delta __read_mostly = 2;
 56
 57 static u32 cube_rtt_scale __read_mostly;
 58 static u32 beta_scale __read_mostly;
 59 static u64 cube_factor __read_mostly;
 60
 61 /* Note parameters that are used for precomputing scale factors are read-only */
 62 module_param(fast_convergence, int, 0644);
 63 MODULE_PARM_DESC(fast_convergence, "turn on/off fast convergence");
 64 module_param(beta, int, 0644);
 65 MODULE_PARM_DESC(beta, "beta for multiplicative increase");
 66 module_param(initial_ssthresh, int, 0644);
 67 MODULE_PARM_DESC(initial_ssthresh, "initial value of slow start threshold");
 68 module_param(bic_scale, int, 0444);
 69 MODULE_PARM_DESC(bic_scale, "scale (scaled by 1024) value for bic function (bic_scale/1024)");
 70 module_param(tcp_friendliness, int, 0644);
 71 MODULE_PARM_DESC(tcp_friendliness, "turn on/off tcp friendliness");
 72 module_param(hystart, int, 0644);
 73 MODULE_PARM_DESC(hystart, "turn on/off hybrid slow start algorithm");
 74 module_param(hystart_detect, int, 0644);
 75 MODULE_PARM_DESC(hystart_detect, "hyrbrid slow start detection mechanisms"
 76                  " 1: packet-train 2: delay 3: both packet-train and delay");
 77 module_param(hystart_low_window, int, 0644);
 78 MODULE_PARM_DESC(hystart_low_window, "lower bound cwnd for hybrid slow start");
 79 module_param(hystart_ack_delta, int, 0644);
 80 MODULE_PARM_DESC(hystart_ack_delta, "spacing between ack's indicating train (msecs)");
 81
 82 /* BIC TCP Parameters */
 83 struct bictcp {
 84         u32     cnt;            /* increase cwnd by 1 after ACKs */
 85         u32     last_max_cwnd;  /* last maximum snd_cwnd */
 86         u32     loss_cwnd;      /* congestion window at last loss */
 87         u32     last_cwnd;      /* the last snd_cwnd */
 88         u32     last_time;      /* time when updated last_cwnd */
 89         u32     bic_origin_point;/* origin point of bic function */
 90         u32     bic_K;          /* time to origin point from the beginning of the current epoch */
 91         u32     delay_min;      /* min delay (msec << 3) */
 92         u32     epoch_start;    /* beginning of an epoch */
 93         u32     ack_cnt;        /* number of acks */
 94         u32     tcp_cwnd;       /* estimated tcp cwnd */
 95 #define ACK_RATIO_SHIFT 4
 96 #define ACK_RATIO_LIMIT (32u << ACK_RATIO_SHIFT)
 97         u16     delayed_ack;    /* estimate the ratio of Packets/ACKs << 4 */
 98         u8      sample_cnt;     /* number of samples to decide curr_rtt */
 99         u8      found;          /* the exit point is found? */
100         u32     round_start;    /* beginning of each round */
101         u32     end_seq;        /* end_seq of the round */
102         u32     last_ack;       /* last time when the ACK spacing is close */
103         u32     curr_rtt;       /* the minimum rtt of current round */
104 };
105
106 static inline void bictcp_reset(struct bictcp *ca)
107 {
108         ca->cnt = 0;
109         ca->last_max_cwnd = 0;
110         ca->last_cwnd = 0;
111         ca->last_time = 0;
112         ca->bic_origin_point = 0;
113         ca->bic_K = 0;
114         ca->delay_min = 0;
115         ca->epoch_start = 0;
116         ca->delayed_ack = 2 << ACK_RATIO_SHIFT;
117         ca->ack_cnt = 0;
118         ca->tcp_cwnd = 0;
119         ca->found = 0;
120 }
121
122 static inline u32 bictcp_clock(void)
```

```
123 {
124 #if HZ < 1000
125         return ktime_to_ms(ktime_get_real());
126 #else
127         return jiffies_to_msecs(jiffies);
128 #endif
129 }
130
131 static inline void bictcp_hystart_reset(struct sock *sk)
132 {
133         struct tcp_sock *tp = tcp_sk(sk);
134         struct bictcp *ca = inet_csk_ca(sk);
135
136         ca->round_start = ca->last_ack = bictcp_clock();
137         ca->end_seq = tp->snd_nxt;
138         ca->curr_rtt = 0;
139         ca->sample_cnt = 0;
140 }
141
142 static void bictcp_init(struct sock *sk)
143 {
144         struct bictcp *ca = inet_csk_ca(sk);
145
146         bictcp_reset(ca);
147         ca->loss_cwnd = 0;
148
149         if (hystart)
150                 bictcp_hystart_reset(sk);
151
152         if (!hystart && initial_ssthresh)
153                 tcp_sk(sk)->snd_ssthresh = initial_ssthresh;
154 }
155
156 /* calculate the cubic root of x using a table lookup followed by one
157  * Newton-Raphson iteration.
158  * Avg err ~= 0.195%
159  */
160 static u32 cubic_root(u64 a)
161 {
162         u32 x, b, shift;
163         /*
164          * cbrt(x) MSB values for x MSB values in [0..63].
165          * Precomputed then refined by hand - Willy Tarreau
166          *
167          * For x in [0..63],
168          *   v = cbrt(x << 18) - 1
169          *   cbrt(x) = (v[x] + 10) >> 6
170          */
171         static const u8 v[] = {
172                 /* 0x00 */    0,   54,   54,   54,  118,  118,  118,  118,
173                 /* 0x08 */  123,  129,  134,  138,  143,  147,  151,  156,
174                 /* 0x10 */  157,  161,  164,  168,  170,  173,  176,  179,
175                 /* 0x18 */  181,  185,  187,  190,  192,  194,  197,  199,
176                 /* 0x20 */  200,  202,  204,  206,  209,  211,  213,  215,
177                 /* 0x28 */  217,  219,  221,  222,  224,  225,  227,  229,
178                 /* 0x30 */  231,  232,  234,  236,  237,  239,  240,  242,
179                 /* 0x38 */  244,  245,  246,  248,  250,  251,  252,  254,
180         };
181
182         b = fls64(a);
183         if (b < 7) {
184                 /* a in [0..63] */
185                 return ((u32)v[(u32)a] + 35) >> 6;
186         }
187
188         b = ((b * 84) >> 8) - 1;
189         shift = (a >> (b * 3));
190
191         x = ((u32)(((u32)v[shift] + 10) << b)) >> 6;
192
193         /*
```

```
194          * Newton-Raphson iteration
195          *                         2
196          * x    = ( 2 * x  +  a / x  ) / 3
197          *  k+1          k         k
198          */
199         x = (2 * x + (u32)div64_u64(a, (u64)x * (u64)(x - 1)));
200         x = ((x * 341) >> 10);
201         return x;
202 }
203
204 /*
205  * Compute congestion window to use.
206  */
207 static inline void bictcp_update(struct bictcp *ca, u32 cwnd)
208 {
209         u32 delta, bic_target, max_cnt;
210         u64 offs, t;
211
212         ca->ack_cnt++;  /* count the number of ACKs */
213
214         if (ca->last_cwnd == cwnd &&
215             (s32)(tcp_time_stamp - ca->last_time) <= HZ / 32)
216                 return;
217
218         ca->last_cwnd = cwnd;
219         ca->last_time = tcp_time_stamp;
220
221         if (ca->epoch_start == 0) {
222                 ca->epoch_start = tcp_time_stamp;       /* record the beginning of an epoch */
223                 ca->ack_cnt = 1;                        /* start counting */
224                 ca->tcp_cwnd = cwnd;                    /* syn with cubic */
225
226                 if (ca->last_max_cwnd <= cwnd) {
227                         ca->bic_K = 0;
228                         ca->bic_origin_point = cwnd;
229                 } else {
230                         /* Compute new K based on
231                          * (wmax-cwnd) * (srtt>>3 / HZ) / c * 2^(3*bictcp_HZ)
232                          */
233                         ca->bic_K = cubic_root(cube_factor
234                                                * (ca->last_max_cwnd - cwnd));
235                         ca->bic_origin_point = ca->last_max_cwnd;
236                 }
237         }
238
239         /* cubic function - calc*/
240         /* calculate c * time^3 / rtt,
241          *  while considering overflow in calculation of time^3
242          * (so time^3 is done by using 64 bit)
243          * and without the support of division of 64bit numbers
244          * (so all divisions are done by using 32 bit)
245          *  also NOTE the unit of those veriables
246          *        time  = (t - K) / 2^bictcp_HZ
247          *        c = bic_scale >> 10
248          * rtt  = (srtt >> 3) / HZ
249          * !!! The following code does not have overflow problems,
250          * if the cwnd < 1 million packets !!!
251          */
252
253         t = (s32)(tcp_time_stamp - ca->epoch_start);
254         t += msecs_to_jiffies(ca->delay_min >> 3);
255         /* change the unit from HZ to bictcp_HZ */
256         t <<= BICTCP_HZ;
257         do_div(t, HZ);
258
259         if (t < ca->bic_K)              /* t - K */
260                 offs = ca->bic_K - t;
261         else
262                 offs = t - ca->bic_K;
263
264         /* c/rtt * (t-K)^3 */
```

```
265         delta = (cube_rtt_scale * offs * offs * offs) >> (10+3*BICTCP_HZ);
266         if (t < ca->bic_K)                                      /* below origin*/
267                 bic_target = ca->bic_origin_point - delta;
268         else                                                    /* above origin*/
269                 bic_target = ca->bic_origin_point + delta;
270
271         /* cubic function - calc bictcp_cnt*/
272         if (bic_target > cwnd) {
273                 ca->cnt = cwnd / (bic_target - cwnd);
274         } else {
275                 ca->cnt = 100 * cwnd;              /* very small increment*/
276         }
277
278         /*
279          * The initial growth of cubic function may be too conservative
280          * when the available bandwidth is still unknown.
281          */
282         if (ca->last_max_cwnd == 0 && ca->cnt > 20)
283                 ca->cnt = 20;   /* increase cwnd 5% per RTT */
284
285         /* TCP Friendly */
286         if (tcp_friendliness) {
287                 u32 scale = beta_scale;
288                 delta = (cwnd * scale) >> 3;
289                 while (ca->ack_cnt > delta) {           /* update tcp cwnd */
290                         ca->ack_cnt -= delta;
291                         ca->tcp_cwnd++;
292                 }
293
294                 if (ca->tcp_cwnd > cwnd){       /* if bic is slower than tcp */
295                         delta = ca->tcp_cwnd - cwnd;
296                         max_cnt = cwnd / delta;
297                         if (ca->cnt > max_cnt)
298                                 ca->cnt = max_cnt;
299                 }
300         }
301
302         ca->cnt = (ca->cnt << ACK_RATIO_SHIFT) / ca->delayed_ack;
303         if (ca->cnt == 0)                       /* cannot be zero */
304                 ca->cnt = 1;
305 }
306
307 static void bictcp_cong_avoid(struct sock *sk, u32 ack, u32 acked)
308 {
309         struct tcp_sock *tp = tcp_sk(sk);
310         struct bictcp *ca = inet_csk_ca(sk);
311
312         if (!tcp_is_cwnd_limited(sk))
313                 return;
314
315         if (tp->snd_cwnd <= tp->snd_ssthresh) {
316                 if (hystart && after(ack, ca->end_seq))
317                         bictcp_hystart_reset(sk);
318                 tcp_slow_start(tp, acked);
319         } else {
320                 bictcp_update(ca, tp->snd_cwnd);
321                 tcp_cong_avoid_ai(tp, ca->cnt);
322         }
323
324 }
325
326 static u32 bictcp_recalc_ssthresh(struct sock *sk)
327 {
328         const struct tcp_sock *tp = tcp_sk(sk);
329         struct bictcp *ca = inet_csk_ca(sk);
330
331         ca->epoch_start = 0;    /* end of epoch */
332
333         /* Wmax and fast convergence */
334         if (tp->snd_cwnd < ca->last_max_cwnd && fast_convergence)
335                 ca->last_max_cwnd = (tp->snd_cwnd * (BICTCP_BETA_SCALE + beta))
```

```
336                         / (2 * BICTCP_BETA_SCALE);
337         else
338                 ca->last_max_cwnd = tp->snd_cwnd;
339
340         ca->loss_cwnd = tp->snd_cwnd;
341
342         return max((tp->snd_cwnd * beta) / BICTCP_BETA_SCALE, 2U);
343 }
344
345 static u32 bictcp_undo_cwnd(struct sock *sk)
346 {
347         struct bictcp *ca = inet_csk_ca(sk);
348
349         return max(tcp_sk(sk)->snd_cwnd, ca->loss_cwnd);
350 }
351
352 static void bictcp_state(struct sock *sk, u8 new_state)
353 {
354         if (new_state == TCP_CA_Loss) {
355                 bictcp_reset(inet_csk_ca(sk));
356                 bictcp_hystart_reset(sk);
357         }
358 }
359
360 static void hystart_update(struct sock *sk, u32 delay)
361 {
362         struct tcp_sock *tp = tcp_sk(sk);
363         struct bictcp *ca = inet_csk_ca(sk);
364
365         if (!(ca->found & hystart_detect)) {
366                 u32 now = bictcp_clock();
367
368                 /* first detection parameter - ack-train detection */
369                 if ((s32)(now - ca->last_ack) <= hystart_ack_delta) {
370                         ca->last_ack = now;
371                         if ((s32)(now - ca->round_start) > ca->delay_min >> 4)
372                                 ca->found |= HYSTART_ACK_TRAIN;
373                 }
374
375                 /* obtain the minimum delay of more than sampling packets */
376                 if (ca->sample_cnt < HYSTART_MIN_SAMPLES) {
377                         if (ca->curr_rtt == 0 || ca->curr_rtt > delay)
378                                 ca->curr_rtt = delay;
379
380                         ca->sample_cnt++;
381                 } else {
382                         if (ca->curr_rtt > ca->delay_min +
383                             HYSTART_DELAY_THRESH(ca->delay_min>>4))
384                                 ca->found |= HYSTART_DELAY;
385                 }
386                 /*
387                  * Either one of two conditions are met,
388                  * we exit from slow start immediately.
389                  */
390                 if (ca->found & hystart_detect)
391                         tp->snd_ssthresh = tp->snd_cwnd;
392         }
393 }
394
395 /* Track delayed acknowledgment ratio using sliding window
396  * ratio = (15*ratio + sample) / 16
397  */
398 static void bictcp_acked(struct sock *sk, u32 cnt, s32 rtt_us)
399 {
400         const struct inet_connection_sock *icsk = inet_csk(sk);
401         const struct tcp_sock *tp = tcp_sk(sk);
402         struct bictcp *ca = inet_csk_ca(sk);
403         u32 delay;
404
405         if (icsk->icsk_ca_state == TCP_CA_Open) {
406                 u32 ratio = ca->delayed_ack;
```

```
407
408                 ratio -= ca->delayed_ack >> ACK_RATIO_SHIFT;
409                 ratio += cnt;
410
411                 ca->delayed_ack = clamp(ratio, 1U, ACK_RATIO_LIMIT);
412         }
413
414         /* Some calls are for duplicates without timetamps */
415         if (rtt_us < 0)
416                 return;
417
418         /* Discard delay samples right after fast recovery */
419         if (ca->epoch_start && (s32)(tcp_time_stamp - ca->epoch_start) < HZ)
420                 return;
421
422         delay = (rtt_us << 3) / USEC_PER_MSEC;
423         if (delay == 0)
424                 delay = 1;
425
426         /* first time call or link delay decreases */
427         if (ca->delay_min == 0 || ca->delay_min > delay)
428                 ca->delay_min = delay;
429
430         /* hystart triggers when cwnd is larger than some threshold */
431         if (hystart && tp->snd_cwnd <= tp->snd_ssthresh &&
432             tp->snd_cwnd >= hystart_low_window)
433                 hystart_update(sk, delay);
434 }
435
436 static struct tcp_congestion_ops cubictcp __read_mostly = {
437         .init           = bictcp_init,
438         .ssthresh       = bictcp_recalc_ssthresh,
439         .cong_avoid     = bictcp_cong_avoid,
440         .set_state      = bictcp_state,
441         .undo_cwnd      = bictcp_undo_cwnd,
442         .pkts_acked     = bictcp_acked,
443         .owner          = THIS_MODULE,
444         .name           = "cubic",
445 };
446
447 static int __init cubictcp_register(void)
448 {
449         BUILD_BUG_ON(sizeof(struct bictcp) > ICSK_CA_PRIV_SIZE);
450
451         /* Precompute a bunch of the scaling factors that are used per-packet
452          * based on SRTT of 100ms
453          */
454
455         beta_scale = 8*(BICTCP_BETA_SCALE+beta)/ 3 / (BICTCP_BETA_SCALE - beta);
456
457         cube_rtt_scale = (bic_scale * 10);      /* 1024*c/rtt */
458
459         /* calculate the "K" for (wmax-cwnd) = c/rtt * K^3
460          *  so K = cubic_root( (wmax-cwnd)*rtt/c )
461          * the unit of K is bictcp_HZ=2^10, not HZ
462          *
463          *  c = bic_scale >> 10
464          *  rtt = 100ms
465          *
466          * the following code has been designed and tested for
467          * cwnd < 1 million packets
468          * RTT < 100 seconds
469          * HZ < 1,000,00  (corresponding to 10 nano-second)
470          */
471
472         /* 1/c * 2^2*bictcp_HZ * srtt */
473         cube_factor = 1ull << (10+3*BICTCP_HZ); /* 2^40 */
474
475         /* divide by bic_scale and by constant Srtt (100ms) */
476         do_div(cube_factor, bic_scale * 10);
477
```

```
478         return tcp_register_congestion_control(&cubictcp);
479 }
480
481 static void __exit cubictcp_unregister(void)
482 {
483         tcp_unregister_congestion_control(&cubictcp);
484 }
485
486 module_init(cubictcp_register);
487 module_exit(cubictcp_unregister);
488
489 MODULE_AUTHOR("Sangtae Ha, Stephen Hemminger");
490 MODULE_LICENSE("GPL");
491 MODULE_DESCRIPTION("CUBIC TCP");
492 MODULE_VERSION("2.3");
493
```

This page was automatically generated by LXR 0.3.1 (source). • Linux is a registered trademark of Linus Torvalds • Contact us

- Home
- Development
- Services
- Training
- Docs
- Community
- Company
- Blog