Linux Cross Reference

Free Electrons

Embedded Linux Experts

Linux/net/ipv4/tcp_illinois.c

```
 1 /*
 2  * TCP Illinois congestion control.
 3  * Home page:
 4  *      http://www.ews.uiuc.edu/~shaoliu/tcpillinois/index.html
 5  *
 6  * The algorithm is described in:
 7  * "TCP-Illinois: A Loss and Delay-Based Congestion Control Algorithm
 8  *  for High-Speed Networks"
 9  * http://www.ifp.illinois.edu/~srikant/Papers/liubassri06perf.pdf
10  *
11  * Implemented from description in paper and ns-2 simulation.
12  * Copyright (C) 2007 Stephen Hemminger <shemminger@linux-foundation.org>
13  */
14
15 #include <linux/module.h>
16 #include <linux/skbuff.h>
17 #include <linux/inet_diag.h>
18 #include <asm/div64.h>
19 #include <net/tcp.h>
20
21 #define ALPHA_SHIFT     7
22 #define ALPHA_SCALE     (1u<<ALPHA_SHIFT)
23 #define ALPHA_MIN       ((3*ALPHA_SCALE)/10)    /* ~0.3 */
24 #define ALPHA_MAX       (10*ALPHA_SCALE)        /* 10.0 */
25 #define ALPHA_BASE      ALPHA_SCALE             /* 1.0 */
26 #define RTT_MAX         (U32_MAX / ALPHA_MAX)   /* 3.3 secs */
27
28 #define BETA_SHIFT      6
29 #define BETA_SCALE      (1u<<BETA_SHIFT)
30 #define BETA_MIN        (BETA_SCALE/8)          /* 0.125 */
31 #define BETA_MAX        (BETA_SCALE/2)          /* 0.5 */
32 #define BETA_BASE       BETA_MAX
33
34 static int win_thresh __read_mostly = 15;
35 module_param(win_thresh, int, 0);
36 MODULE_PARM_DESC(win_thresh, "Window threshold for starting adaptive sizing");
37
38 static int theta __read_mostly = 5;
39 module_param(theta, int, 0);
40 MODULE_PARM_DESC(theta, "# of fast RTT's before full growth");
41
42 /* TCP Illinois Parameters */
43 struct illinois {
```

```
 44         u64     sum_rtt;        /* sum of rtt's measured within last rtt */
 45         u16     cnt_rtt;        /* # of rtts measured within last rtt */
 46         u32     base_rtt;       /* min of all rtt in usec */
 47         u32     max_rtt;        /* max of all rtt in usec */
 48         u32     end_seq;        /* right edge of current RTT */
 49         u32     alpha;          /* Additive increase */
 50         u32     beta;           /* Muliplicative decrease */
 51         u16     acked;          /* # packets acked by current ACK */
 52         u8      rtt_above;      /* average rtt has gone above threshold */
 53         u8      rtt_low;        /* # of rtts measurements below threshold */
 54 };
 55
 56 static void rtt_reset(struct sock *sk)
 57 {
 58         struct tcp_sock *tp = tcp_sk(sk);
 59         struct illinois *ca = inet_csk_ca(sk);
 60
 61         ca->end_seq = tp->snd_nxt;
 62         ca->cnt_rtt = 0;
 63         ca->sum_rtt = 0;
 64
 65         /* TODO: age max_rtt? */
 66 }
 67
 68 static void tcp_illinois_init(struct sock *sk)
 69 {
 70         struct illinois *ca = inet_csk_ca(sk);
 71
 72         ca->alpha = ALPHA_MAX;
 73         ca->beta = BETA_BASE;
 74         ca->base_rtt = 0x7fffffff;
 75         ca->max_rtt = 0;
 76
 77         ca->acked = 0;
 78         ca->rtt_low = 0;
 79         ca->rtt_above = 0;
 80
 81         rtt_reset(sk);
 82 }
 83
 84 /* Measure RTT for each ack. */
 85 static void tcp_illinois_acked(struct sock *sk, u32 pkts_acked, s32 rtt)
 86 {
 87         struct illinois *ca = inet_csk_ca(sk);
 88
 89         ca->acked = pkts_acked;
 90
 91         /* dup ack, no rtt sample */
 92         if (rtt < 0)
 93                 return;
 94
 95         /* ignore bogus values, this prevents wraparound in alpha math */
 96         if (rtt > RTT_MAX)
 97                 rtt = RTT_MAX;
 98
 99         /* keep track of minimum RTT seen so far */
100         if (ca->base_rtt > rtt)
101                 ca->base_rtt = rtt;
102
103         /* and max */
104         if (ca->max_rtt < rtt)
105                 ca->max_rtt = rtt;
106
107         ++ca->cnt_rtt;
108         ca->sum_rtt += rtt;
```

```
109 }
110
111 /* Maximum queuing delay */
112 static inline u32 max_delay(const struct illinois *ca)
113 {
114         return ca->max_rtt - ca->base_rtt;
115 }
116
117 /* Average queuing delay */
118 static inline u32 avg_delay(const struct illinois *ca)
119 {
120         u64 t = ca->sum_rtt;
121
122         do_div(t, ca->cnt_rtt);
123         return t - ca->base_rtt;
124 }
125
126 /*
127  * Compute value of alpha used for additive increase.
128  * If small window then use 1.0, equivalent to Reno.
129  *
130  * For larger windows, adjust based on average delay.
131  * A. If average delay is at minimum (we are uncongested),
132  *    then use large alpha (10.0) to increase faster.
133  * B. If average delay is at maximum (getting congested)
134  *    then use small alpha (0.3)
135  *
136  * The result is a convex window growth curve.
137  */
138 static u32 alpha(struct illinois *ca, u32 da, u32 dm)
139 {
140         u32 d1 = dm / 100;      /* Low threshold */
141
142         if (da <= d1) {
143                 /* If never got out of low delay zone, then use max */
144                 if (!ca->rtt_above)
145                         return ALPHA_MAX;
146
147                 /* Wait for 5 good RTT's before allowing alpha to go alpha max.
148                  * This prevents one good RTT from causing sudden window increase.
149                  */
150                 if (++ca->rtt_low < theta)
151                         return ca->alpha;
152
153                 ca->rtt_low = 0;
154                 ca->rtt_above = 0;
155                 return ALPHA_MAX;
156         }
157
158         ca->rtt_above = 1;
159
160         /*
161          * Based on:
162          *
163          *      (dm - d1) amin amax
164          * k1 = -------------------
165          *         amax - amin
166          *
167          *       (dm - d1) amin
168          * k2 = ----------------  - d1
169          *        amax - amin
170          *
171          *             k1
172          * alpha = ----------
173          *          k2 + da
```

```
174          */
175
176         dm -= d1;
177         da -= d1;
178         return (dm * ALPHA_MAX) /
179                 (dm + (da  * (ALPHA_MAX - ALPHA_MIN)) / ALPHA_MIN);
180 }
181
182 /*
183  * Beta used for multiplicative decrease.
184  * For small window sizes returns same value as Reno (0.5)
185  *
186  * If delay is small (10% of max) then beta = 1/8
187  * If delay is up to 80% of max then beta = 1/2
188  * In between is a linear function
189  */
190 static u32 beta(u32 da, u32 dm)
191 {
192         u32 d2, d3;
193
194         d2 = dm / 10;
195         if (da <= d2)
196                 return BETA_MIN;
197
198         d3 = (8 * dm) / 10;
199         if (da >= d3 || d3 <= d2)
200                 return BETA_MAX;
201
202         /*
203          * Based on:
204          *
205          *       bmin d3 - bmax d2
206          * k3 = -------------------
207          *         d3 - d2
208          *
209          *       bmax - bmin
210          * k4 = -------------
211          *         d3 - d2
212          *
213          * b = k3 + k4 da
214          */
215         return (BETA_MIN * d3 - BETA_MAX * d2 + (BETA_MAX - BETA_MIN) * da)
216                 / (d3 - d2);
217 }
218
219 /* Update alpha and beta values once per RTT */
220 static void update_params(struct sock *sk)
221 {
222         struct tcp_sock *tp = tcp_sk(sk);
223         struct illinois *ca = inet_csk_ca(sk);
224
225         if (tp->snd_cwnd < win_thresh) {
226                 ca->alpha = ALPHA_BASE;
227                 ca->beta = BETA_BASE;
228         } else if (ca->cnt_rtt > 0) {
229                 u32 dm = max_delay(ca);
230                 u32 da = avg_delay(ca);
231
232                 ca->alpha = alpha(ca, da, dm);
233                 ca->beta = beta(da, dm);
234         }
235
236         rtt_reset(sk);
237 }
238
```

```
239 /*
240  * In case of loss, reset to default values
241  */
242 static void tcp_illinois_state(struct sock *sk, u8 new_state)
243 {
244         struct illinois *ca = inet_csk_ca(sk);
245
246         if (new_state == TCP_CA_Loss) {
247                 ca->alpha = ALPHA_BASE;
248                 ca->beta = BETA_BASE;
249                 ca->rtt_low = 0;
250                 ca->rtt_above = 0;
251                 rtt_reset(sk);
252         }
253 }
254
255 /*
256  * Increase window in response to successful acknowledgment.
257  */
258 static void tcp_illinois_cong_avoid(struct sock *sk, u32 ack, u32 acked)
259 {
260         struct tcp_sock *tp = tcp_sk(sk);
261         struct illinois *ca = inet_csk_ca(sk);
262
263         if (after(ack, ca->end_seq))
264                 update_params(sk);
265
266         /* RFC2861 only increase cwnd if fully utilized */
267         if (!tcp_is_cwnd_limited(sk))
268                 return;
269
270         /* In slow start */
271         if (tp->snd_cwnd <= tp->snd_ssthresh)
272                 tcp_slow_start(tp, acked);
273
274         else {
275                 u32 delta;
276
277                 /* snd_cwnd_cnt is # of packets since last cwnd increment */
278                 tp->snd_cwnd_cnt += ca->acked;
279                 ca->acked = 1;
280
281                 /* This is close approximation of:
282                  * tp->snd_cwnd += alpha/tp->snd_cwnd
283                 */
284                 delta = (tp->snd_cwnd_cnt * ca->alpha) >> ALPHA_SHIFT;
285                 if (delta >= tp->snd_cwnd) {
286                         tp->snd_cwnd = min(tp->snd_cwnd + delta / tp->snd_cwnd,
287                                            (u32) tp->snd_cwnd_clamp);
288                         tp->snd_cwnd_cnt = 0;
289                 }
290         }
291 }
292
293 static u32 tcp_illinois_ssthresh(struct sock *sk)
294 {
295         struct tcp_sock *tp = tcp_sk(sk);
296         struct illinois *ca = inet_csk_ca(sk);
297
298         /* Multiplicative decrease */
299         return max(tp->snd_cwnd - ((tp->snd_cwnd * ca->beta) >> BETA_SHIFT), 2U);
300 }
301
302
303 /* Extract info for Tcp socket info provided via netlink. */
```

```
304 static void tcp_illinois_info(struct sock *sk, u32 ext,
305                                struct sk_buff *skb)
306 {
307         const struct illinois *ca = inet_csk_ca(sk);
308
309         if (ext & (1 << (INET_DIAG_VEGASINFO - 1))) {
310                 struct tcpvegas_info info = {
311                         .tcpv_enabled = 1,
312                         .tcpv_rttcnt = ca->cnt_rtt,
313                         .tcpv_minrtt = ca->base_rtt,
314                 };
315
316                 if (info.tcpv_rttcnt > 0) {
317                         u64 t = ca->sum_rtt;
318
319                         do_div(t, info.tcpv_rttcnt);
320                         info.tcpv_rtt = t;
321                 }
322                 nla_put(skb, INET_DIAG_VEGASINFO, sizeof(info), &info);
323         }
324 }
325
326 static struct tcp_congestion_ops tcp_illinois __read_mostly = {
327         .init           = tcp_illinois_init,
328         .ssthresh       = tcp_illinois_ssthresh,
329         .cong_avoid     = tcp_illinois_cong_avoid,
330         .set_state      = tcp_illinois_state,
331         .get_info       = tcp_illinois_info,
332         .pkts_acked     = tcp_illinois_acked,
333
334         .owner          = THIS_MODULE,
335         .name           = "illinois",
336 };
337
338 static int __init tcp_illinois_register(void)
339 {
340         BUILD_BUG_ON(sizeof(struct illinois) > ICSK_CA_PRIV_SIZE);
341         return tcp_register_congestion_control(&tcp_illinois);
342 }
343
344 static void __exit tcp_illinois_unregister(void)
345 {
346         tcp_unregister_congestion_control(&tcp_illinois);
347 }
348
349 module_init(tcp_illinois_register);
350 module_exit(tcp_illinois_unregister);
351
352 MODULE_AUTHOR("Stephen Hemminger, Shao Liu");
353 MODULE_LICENSE("GPL");
354 MODULE_DESCRIPTION("TCP Illinois");
355 MODULE_VERSION("1.0");
356
```