Linux Cross Reference

Free Electrons

Embedded Linux Experts

Linux/net/ipv4/tcp_yeah.c

```
 1 /*
 2  *
 3  *   YeAH TCP
 4  *
 5  * For further details look at:
 6  *   https://web.archive.org/web/20080316215752/http://wil.cs.caltech.edu/pfldnet2007/paper/YeAH_TCP.pdf
 7  *
 8  */
 9 #include <linux/mm.h>
10 #include <linux/module.h>
11 #include <linux/skbuff.h>
12 #include <linux/inet_diag.h>
13
14 #include <net/tcp.h>
15
16 #include "tcp_vegas.h"
17
18 #define TCP_YEAH_ALPHA       80 /* number of packets queued at the bottleneck */
19 #define TCP_YEAH_GAMMA        1 /* fraction of queue to be removed per rtt */
20 #define TCP_YEAH_DELTA        3 /* log minimum fraction of cwnd to be removed on loss */
21 #define TCP_YEAH_EPSILON      1 /* log maximum fraction to be removed on early decongestion */
22 #define TCP_YEAH_PHY          8 /* maximum delta from base */
23 #define TCP_YEAH_RHO         16 /* minimum number of consecutive rtt to consider competition on loss */
24 #define TCP_YEAH_ZETA        50 /* minimum number of state switches to reset reno_count */
25
26 #define TCP_SCALABLE_AI_CNT      100U
27
28 /* YeAH variables */
29 struct yeah {
30         struct vegas vegas;     /* must be first */
31
32         /* YeAH */
33         u32 lastQ;
34         u32 doing_reno_now;
35
36         u32 reno_count;
37         u32 fast_count;
38
39         u32 pkts_acked;
40 };
41
42 static void tcp_yeah_init(struct sock *sk)
43 {
44         struct tcp_sock *tp = tcp_sk(sk);
45         struct yeah *yeah = inet_csk_ca(sk);
46
47         tcp_vegas_init(sk);
48
49         yeah->doing_reno_now = 0;
50         yeah->lastQ = 0;
51
52         yeah->reno_count = 2;
53
54         /* Ensure the MD arithmetic works.  This is somewhat pedantic,
55          * since I don't think we will see a cwnd this large. :) */
```

```
56          tp->snd_cwnd_clamp = min_t(u32, tp->snd_cwnd_clamp, 0xffffffff/128);
57
58  }
59
60
61  static void tcp_yeah_pkts_acked(struct sock *sk, u32 pkts_acked, s32 rtt_us)
62  {
63          const struct inet_connection_sock *icsk = inet_csk(sk);
64          struct yeah *yeah = inet_csk_ca(sk);
65
66          if (icsk->icsk_ca_state == TCP_CA_Open)
67                  yeah->pkts_acked = pkts_acked;
68
69          tcp_vegas_pkts_acked(sk, pkts_acked, rtt_us);
70  }
71
72  static void tcp_yeah_cong_avoid(struct sock *sk, u32 ack, u32 acked)
73  {
74          struct tcp_sock *tp = tcp_sk(sk);
75          struct yeah *yeah = inet_csk_ca(sk);
76
77          if (!tcp_is_cwnd_limited(sk))
78                  return;
79
80          if (tp->snd_cwnd <= tp->snd_ssthresh)
81                  tcp_slow_start(tp, acked);
82
83          else if (!yeah->doing_reno_now) {
84                  /* Scalable */
85
86                  tp->snd_cwnd_cnt += yeah->pkts_acked;
87                  if (tp->snd_cwnd_cnt > min(tp->snd_cwnd, TCP_SCALABLE_AI_CNT)){
88                          if (tp->snd_cwnd < tp->snd_cwnd_clamp)
89                                  tp->snd_cwnd++;
90                          tp->snd_cwnd_cnt = 0;
91                  }
92
93                  yeah->pkts_acked = 1;
94
95          } else {
96                  /* Reno */
97                  tcp_cong_avoid_ai(tp, tp->snd_cwnd);
98          }
99
100         /* The key players are v_vegas.beg_snd_una and v_beg_snd_nxt.
101          *
102          * These are so named because they represent the approximate values
103          * of snd_una and snd_nxt at the beginning of the current RTT. More
104          * precisely, they represent the amount of data sent during the RTT.
105          * At the end of the RTT, when we receive an ACK for v_beg_snd_nxt,
106          * we will calculate that (v_beg_snd_nxt - v_vegas.beg_snd_una) outstanding
107          * bytes of data have been ACKed during the course of the RTT, giving
108          * an "actual" rate of:
109          *
110          *     (v_beg_snd_nxt - v_vegas.beg_snd_una) / (rtt duration)
111          *
112          * Unfortunately, v_vegas.beg_snd_una is not exactly equal to snd_una,
113          * because delayed ACKs can cover more than one segment, so they
114          * don't line up yeahly with the boundaries of RTTs.
115          *
116          * Another unfortunate fact of life is that delayed ACKs delay the
117          * advance of the left edge of our send window, so that the number
118          * of bytes we send in an RTT is often less than our cwnd will allow.
119          * So we keep track of our cwnd separately, in v_beg_snd_cwnd.
120          */
121
122         if (after(ack, yeah->vegas.beg_snd_nxt)) {
123
124                 /* We do the Vegas calculations only if we got enough RTT
125                  * samples that we can be reasonably sure that we got
126                  * at least one RTT sample that wasn't from a delayed ACK.
127                  * If we only had 2 samples total,
128                  * then that means we're getting only 1 ACK per RTT, which
129                  * means they're almost certainly delayed ACKs.
130                  * If  we have 3 samples, we should be OK.
```

```
131                 */
132
133                if (yeah->vegas.cntRTT > 2) {
134                        u32 rtt, queue;
135                        u64 bw;
136
137                        /* We have enough RTT samples, so, using the Vegas
138                         * algorithm, we determine if we should increase or
139                         * decrease cwnd, and by how much.
140                         */
141
142                        /* Pluck out the RTT we are using for the Vegas
143                         * calculations. This is the min RTT seen during the
144                         * last RTT. Taking the min filters out the effects
145                         * of delayed ACKs, at the cost of noticing congestion
146                         * a bit later.
147                         */
148                        rtt = yeah->vegas.minRTT;
149
150                        /* Compute excess number of packets above bandwidth
151                         * Avoid doing full 64 bit divide.
152                         */
153                        bw = tp->snd_cwnd;
154                        bw *= rtt - yeah->vegas.baseRTT;
155                        do_div(bw, rtt);
156                        queue = bw;
157
158                        if (queue > TCP_YEAH_ALPHA ||
159                            rtt - yeah->vegas.baseRTT > (yeah->vegas.baseRTT / TCP_YEAH_PHY)) {
160                                if (queue > TCP_YEAH_ALPHA &&
161                                    tp->snd_cwnd > yeah->reno_count) {
162                                        u32 reduction = min(queue / TCP_YEAH_GAMMA ,
163                                                            tp->snd_cwnd >> TCP_YEAH_EPSILON);
164
165                                        tp->snd_cwnd -= reduction;
166
167                                        tp->snd_cwnd = max(tp->snd_cwnd,
168                                                           yeah->reno_count);
169
170                                        tp->snd_ssthresh = tp->snd_cwnd;
171                                }
172
173                                if (yeah->reno_count <= 2)
174                                        yeah->reno_count = max(tp->snd_cwnd>>1, 2U);
175                                else
176                                        yeah->reno_count++;
177
178                                yeah->doing_reno_now = min(yeah->doing_reno_now + 1,
179                                                           0xffffffU);
180                        } else {
181                                yeah->fast_count++;
182
183                                if (yeah->fast_count > TCP_YEAH_ZETA) {
184                                        yeah->reno_count = 2;
185                                        yeah->fast_count = 0;
186                                }
187
188                                yeah->doing_reno_now = 0;
189                        }
190
191                        yeah->lastQ = queue;
192
193                }
194
195                /* Save the extent of the current window so we can use this
196                 * at the end of the next RTT.
197                 */
198                yeah->vegas.beg_snd_una  = yeah->vegas.beg_snd_nxt;
199                yeah->vegas.beg_snd_nxt  = tp->snd_nxt;
200                yeah->vegas.beg_snd_cwnd = tp->snd_cwnd;
201
202                /* Wipe the slate clean for the next RTT. */
203                yeah->vegas.cntRTT = 0;
204                yeah->vegas.minRTT = 0x7fffffff;
205        }
```

```
206 }
207
208 static u32 tcp_yeah_ssthresh(struct sock *sk) {
209         const struct tcp_sock *tp = tcp_sk(sk);
210         struct yeah *yeah = inet_csk_ca(sk);
211         u32 reduction;
212
213         if (yeah->doing_reno_now < TCP_YEAH_RHO) {
214                 reduction = yeah->lastQ;
215
216                 reduction = min(reduction, max(tp->snd_cwnd>>1, 2U));
217
218                 reduction = max(reduction, tp->snd_cwnd >> TCP_YEAH_DELTA);
219         } else
220                 reduction = max(tp->snd_cwnd>>1, 2U);
221
222         yeah->fast_count = 0;
223         yeah->reno_count = max(yeah->reno_count>>1, 2U);
224
225         return tp->snd_cwnd - reduction;
226 }
227
228 static struct tcp_congestion_ops tcp_yeah __read_mostly = {
229         .init           = tcp_yeah_init,
230         .ssthresh       = tcp_yeah_ssthresh,
231         .cong_avoid     = tcp_yeah_cong_avoid,
232         .set_state      = tcp_vegas_state,
233         .cwnd_event     = tcp_vegas_cwnd_event,
234         .get_info       = tcp_vegas_get_info,
235         .pkts_acked     = tcp_yeah_pkts_acked,
236
237         .owner          = THIS_MODULE,
238         .name           = "yeah",
239 };
240
241 static int __init tcp_yeah_register(void)
242 {
243         BUG_ON(sizeof(struct yeah) > ICSK_CA_PRIV_SIZE);
244         tcp_register_congestion_control(&tcp_yeah);
245         return 0;
246 }
247
248 static void __exit tcp_yeah_unregister(void)
249 {
250         tcp_unregister_congestion_control(&tcp_yeah);
251 }
252
253 module_init(tcp_yeah_register);
254 module_exit(tcp_yeah_unregister);
255
256 MODULE_AUTHOR("Angelo P. Castellani");
257 MODULE_LICENSE("GPL");
258 MODULE_DESCRIPTION("YeAH TCP");
259
```