Linux Cross Reference

Free Electrons

Embedded Linux Experts

• *source navigation* • diff markup • identifier search • freetext search •

Version:
2.0.40 2.2.26 2.4.37 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 3.10 3.11 3.12 3.13 3.14 3.15 3.16 ***3.17***

Linux/net/ipv4/tcp_htcp.c

```
 1 /*
 2  * H-TCP congestion control. The algorithm is detailed in:
 3  * R.N.Shorten, D.J.Leith:
 4  *   "H-TCP: TCP for high-speed and long-distance networks"
 5  *   Proc. PFLDnet, Argonne, 2004.
 6  * http://www.hamilton.ie/net/htcp3.pdf
 7  */
 8
 9 #include <linux/mm.h>
10 #include <linux/module.h>
11 #include <net/tcp.h>
12
13 #define ALPHA_BASE      (1<<7)  /* 1.0 with shift << 7 */
14 #define BETA_MIN        (1<<6)  /* 0.5 with shift << 7 */
15 #define BETA_MAX        102     /* 0.8 with shift << 7 */
16
17 static int use_rtt_scaling __read_mostly = 1;
18 module_param(use_rtt_scaling, int, 0644);
19 MODULE_PARM_DESC(use_rtt_scaling, "turn on/off RTT scaling");
20
21 static int use_bandwidth_switch __read_mostly = 1;
22 module_param(use_bandwidth_switch, int, 0644);
23 MODULE_PARM_DESC(use_bandwidth_switch, "turn on/off bandwidth switcher");
24
25 struct htcp {
26         u32     alpha;          /* Fixed point arith, << 7 */
27         u8      beta;           /* Fixed point arith, << 7 */
28         u8      modeswitch;     /* Delay modeswitch
29                                    until we had at least one congestion event */
30         u16     pkts_acked;
31         u32     packetcount;
32         u32     minRTT;
33         u32     maxRTT;
34         u32     last_cong;      /* Time since last congestion event end */
35         u32     undo_last_cong;
36
37         u32     undo_maxRTT;
38         u32     undo_old_maxB;
39
40         /* Bandwidth estimation */
41         u32     minB;
42         u32     maxB;
43         u32     old_maxB;
```

```
44          u32     Bi;
45          u32     lasttime;
46 };
47
48 static inline u32 htcp_cong_time(const struct htcp *ca)
49 {
50          return jiffies - ca->last_cong;
51 }
52
53 static inline u32 htcp_ccount(const struct htcp *ca)
54 {
55          return htcp_cong_time(ca) / ca->minRTT;
56 }
57
58 static inline void htcp_reset(struct htcp *ca)
59 {
60          ca->undo_last_cong = ca->last_cong;
61          ca->undo_maxRTT = ca->maxRTT;
62          ca->undo_old_maxB = ca->old_maxB;
63
64          ca->last_cong = jiffies;
65 }
66
67 static u32 htcp_cwnd_undo(struct sock *sk)
68 {
69          const struct tcp_sock *tp = tcp_sk(sk);
70          struct htcp *ca = inet_csk_ca(sk);
71
72          if (ca->undo_last_cong) {
73                  ca->last_cong = ca->undo_last_cong;
74                  ca->maxRTT = ca->undo_maxRTT;
75                  ca->old_maxB = ca->undo_old_maxB;
76                  ca->undo_last_cong = 0;
77          }
78
79          return max(tp->snd_cwnd, (tp->snd_ssthresh << 7) / ca->beta);
80 }
81
82 static inline void measure_rtt(struct sock *sk, u32 srtt)
83 {
84          const struct inet_connection_sock *icsk = inet_csk(sk);
85          struct htcp *ca = inet_csk_ca(sk);
86
87          /* keep track of minimum RTT seen so far, minRTT is zero at first */
88          if (ca->minRTT > srtt || !ca->minRTT)
89                  ca->minRTT = srtt;
90
91          /* max RTT */
92          if (icsk->icsk_ca_state == TCP_CA_Open) {
93                  if (ca->maxRTT < ca->minRTT)
94                          ca->maxRTT = ca->minRTT;
95                  if (ca->maxRTT < srtt &&
96                      srtt <= ca->maxRTT + msecs_to_jiffies(20))
97                          ca->maxRTT = srtt;
98          }
99 }
100
101 static void measure_achieved_throughput(struct sock *sk, u32 pkts_acked, s32 rtt)
102 {
103         const struct inet_connection_sock *icsk = inet_csk(sk);
104         const struct tcp_sock *tp = tcp_sk(sk);
105         struct htcp *ca = inet_csk_ca(sk);
106         u32 now = tcp_time_stamp;
107
108         if (icsk->icsk_ca_state == TCP_CA_Open)
```

```
109                 ca->pkts_acked = pkts_acked;
110
111         if (rtt > 0)
112                 measure_rtt(sk, usecs_to_jiffies(rtt));
113
114         if (!use_bandwidth_switch)
115                 return;
116
117         /* achieved throughput calculations */
118         if (!((1 << icsk->icsk_ca_state) & (TCPF_CA_Open | TCPF_CA_Disorder))) {
119                 ca->packetcount = 0;
120                 ca->lasttime = now;
121                 return;
122         }
123
124         ca->packetcount += pkts_acked;
125
126         if (ca->packetcount >= tp->snd_cwnd - (ca->alpha >> 7 ? : 1) &&
127             now - ca->lasttime >= ca->minRTT &&
128             ca->minRTT > 0) {
129                 __u32 cur_Bi = ca->packetcount * HZ / (now - ca->lasttime);
130
131                 if (htcp_ccount(ca) <= 3) {
132                         /* just after backoff */
133                         ca->minB = ca->maxB = ca->Bi = cur_Bi;
134                 } else {
135                         ca->Bi = (3 * ca->Bi + cur_Bi) / 4;
136                         if (ca->Bi > ca->maxB)
137                                 ca->maxB = ca->Bi;
138                         if (ca->minB > ca->maxB)
139                                 ca->minB = ca->maxB;
140                 }
141                 ca->packetcount = 0;
142                 ca->lasttime = now;
143         }
144 }
145
146 static inline void htcp_beta_update(struct htcp *ca, u32 minRTT, u32 maxRTT)
147 {
148         if (use_bandwidth_switch) {
149                 u32 maxB = ca->maxB;
150                 u32 old_maxB = ca->old_maxB;
151                 ca->old_maxB = ca->maxB;
152
153                 if (!between(5 * maxB, 4 * old_maxB, 6 * old_maxB)) {
154                         ca->beta = BETA_MIN;
155                         ca->modeswitch = 0;
156                         return;
157                 }
158         }
159
160         if (ca->modeswitch && minRTT > msecs_to_jiffies(10) && maxRTT) {
161                 ca->beta = (minRTT << 7) / maxRTT;
162                 if (ca->beta < BETA_MIN)
163                         ca->beta = BETA_MIN;
164                 else if (ca->beta > BETA_MAX)
165                         ca->beta = BETA_MAX;
166         } else {
167                 ca->beta = BETA_MIN;
168                 ca->modeswitch = 1;
169         }
170 }
171
172 static inline void htcp_alpha_update(struct htcp *ca)
173 {
```

```
174         u32 minRTT = ca->minRTT;
175         u32 factor = 1;
176         u32 diff = htcp_cong_time(ca);
177
178         if (diff > HZ) {
179                 diff -= HZ;
180                 factor = 1 + (10 * diff + ((diff / 2) * (diff / 2) / HZ)) / HZ;
181         }
182
183         if (use_rtt_scaling && minRTT) {
184                 u32 scale = (HZ << 3) / (10 * minRTT);
185
186                 /* clamping ratio to interval [0.5,10]<<3 */
187                 scale = min(max(scale, 1U << 2), 10U << 3);
188                 factor = (factor << 3) / scale;
189                 if (!factor)
190                         factor = 1;
191         }
192
193         ca->alpha = 2 * factor * ((1 << 7) - ca->beta);
194         if (!ca->alpha)
195                 ca->alpha = ALPHA_BASE;
196 }
197
198 /*
199  * After we have the rtt data to calculate beta, we'd still prefer to wait one
200  * rtt before we adjust our beta to ensure we are working from a consistent
201  * data.
202  *
203  * This function should be called when we hit a congestion event since only at
204  * that point do we really have a real sense of maxRTT (the queues en route
205  * were getting just too full now).
206  */
207 static void htcp_param_update(struct sock *sk)
208 {
209         struct htcp *ca = inet_csk_ca(sk);
210         u32 minRTT = ca->minRTT;
211         u32 maxRTT = ca->maxRTT;
212
213         htcp_beta_update(ca, minRTT, maxRTT);
214         htcp_alpha_update(ca);
215
216         /* add slowly fading memory for maxRTT to accommodate routing changes */
217         if (minRTT > 0 && maxRTT > minRTT)
218                 ca->maxRTT = minRTT + ((maxRTT - minRTT) * 95) / 100;
219 }
220
221 static u32 htcp_recalc_ssthresh(struct sock *sk)
222 {
223         const struct tcp_sock *tp = tcp_sk(sk);
224         const struct htcp *ca = inet_csk_ca(sk);
225
226         htcp_param_update(sk);
227         return max((tp->snd_cwnd * ca->beta) >> 7, 2U);
228 }
229
230 static void htcp_cong_avoid(struct sock *sk, u32 ack, u32 acked)
231 {
232         struct tcp_sock *tp = tcp_sk(sk);
233         struct htcp *ca = inet_csk_ca(sk);
234
235         if (!tcp_is_cwnd_limited(sk))
236                 return;
237
238         if (tp->snd_cwnd <= tp->snd_ssthresh)
```

```
239                 tcp_slow_start(tp, acked);
240         else {
241                 /* In dangerous area, increase slowly.
242                  * In theory this is tp->snd_cwnd += alpha / tp->snd_cwnd
243                  */
244                 if ((tp->snd_cwnd_cnt * ca->alpha)>>7 >= tp->snd_cwnd) {
245                         if (tp->snd_cwnd < tp->snd_cwnd_clamp)
246                                 tp->snd_cwnd++;
247                         tp->snd_cwnd_cnt = 0;
248                         htcp_alpha_update(ca);
249                 } else
250                         tp->snd_cwnd_cnt += ca->pkts_acked;
251
252                 ca->pkts_acked = 1;
253         }
254 }
255
256 static void htcp_init(struct sock *sk)
257 {
258         struct htcp *ca = inet_csk_ca(sk);
259
260         memset(ca, 0, sizeof(struct htcp));
261         ca->alpha = ALPHA_BASE;
262         ca->beta = BETA_MIN;
263         ca->pkts_acked = 1;
264         ca->last_cong = jiffies;
265 }
266
267 static void htcp_state(struct sock *sk, u8 new_state)
268 {
269         switch (new_state) {
270         case TCP_CA_Open:
271                 {
272                         struct htcp *ca = inet_csk_ca(sk);
273                         if (ca->undo_last_cong) {
274                                 ca->last_cong = jiffies;
275                                 ca->undo_last_cong = 0;
276                         }
277                 }
278                 break;
279         case TCP_CA_CWR:
280         case TCP_CA_Recovery:
281         case TCP_CA_Loss:
282                 htcp_reset(inet_csk_ca(sk));
283                 break;
284         }
285 }
286
287 static struct tcp_congestion_ops htcp __read_mostly = {
288         .init           = htcp_init,
289         .ssthresh       = htcp_recalc_ssthresh,
290         .cong_avoid     = htcp_cong_avoid,
291         .set_state      = htcp_state,
292         .undo_cwnd      = htcp_cwnd_undo,
293         .pkts_acked     = measure_achieved_throughput,
294         .owner          = THIS_MODULE,
295         .name           = "htcp",
296 };
297
298 static int __init htcp_register(void)
299 {
300         BUILD_BUG_ON(sizeof(struct htcp) > ICSK_CA_PRIV_SIZE);
301         BUILD_BUG_ON(BETA_MIN >= BETA_MAX);
302         return tcp_register_congestion_control(&htcp);
303 }
```

```
304
305 static void __exit htcp_unregister(void)
306 {
307         tcp_unregister_congestion_control(&htcp);
308 }
309
310 module_init(htcp_register);
311 module_exit(htcp_unregister);
312
313 MODULE_AUTHOR("Baruch Even");
314 MODULE_LICENSE("GPL");
315 MODULE_DESCRIPTION("H-TCP");
316
```