Linux Cross Reference

Free Electrons

Embedded Linux Experts

Linux/net/ipv4/tcp_westwood.c

```
 1 /*
 2  * TCP Westwood+: end-to-end bandwidth estimation for TCP
 3  *
 4  *      Angelo Dell'Aera: author of the first version of TCP Westwood+ in Linux 2.4
 5  *
 6  * Support at http://c3lab.poliba.it/index.php/Westwood
 7  * Main references in literature:
 8  *
 9  * - Mascolo S, Casetti, M. Gerla et al.
10  *   "TCP Westwood: bandwidth estimation for TCP" Proc. ACM Mobicom 2001
11  *
12  * - A. Grieco, s. Mascolo
13  *   "Performance evaluation of New Reno, Vegas, Westwood+ TCP" ACM Computer
14  *     Comm. Review, 2004
15  *
16  * - A. Dell'Aera, L. Grieco, S. Mascolo.
17  *   "Linux 2.4 Implementation of Westwood+ TCP with Rate-Halving :
18  *    A Performance Evaluation Over the Internet" (ICC 2004), Paris, June 2004
19  *
20  * Westwood+ employs end-to-end bandwidth measurement to set cwnd and
21  * ssthresh after packet loss. The probing phase is as the original Reno.
22  */
23
24 #include <linux/mm.h>
25 #include <linux/module.h>
26 #include <linux/skbuff.h>
27 #include <linux/inet_diag.h>
28 #include <net/tcp.h>
29
30 /* TCP Westwood structure */
31 struct westwood {
32         u32     bw_ns_est;        /* first bandwidth estimation..not too smoothed 8) */
33         u32     bw_est;           /* bandwidth estimate */
34         u32     rtt_win_sx;       /* here starts a new evaluation... */
35         u32     bk;
36         u32     snd_una;          /* used for evaluating the number of acked bytes */
37         u32     cumul_ack;
38         u32     accounted;
39         u32     rtt;
40         u32     rtt_min;          /* minimum observed RTT */
41         u8      first_ack;        /* flag which infers that this is the first ack */
42         u8      reset_rtt_min;    /* Reset RTT min to next RTT sample*/
43 };
```

```
 44
 45
 46 /* TCP Westwood functions and constants */
 47 #define TCP_WESTWOOD_RTT_MIN   (HZ/20)  /* 50ms */
 48 #define TCP_WESTWOOD_INIT_RTT  (20*HZ)  /* maybe too conservative?! */
 49
 50 /*
 51  * @tcp_westwood_create
 52  * This function initializes fields used in TCP Westwood+,
 53  * it is called after the initial SYN, so the sequence numbers
 54  * are correct but new passive connections we have no
 55  * information about RTTmin at this time so we simply set it to
 56  * TCP_WESTWOOD_INIT_RTT. This value was chosen to be too conservative
 57  * since in this way we're sure it will be updated in a consistent
 58  * way as soon as possible. It will reasonably happen within the first
 59  * RTT period of the connection lifetime.
 60  */
 61 static void tcp_westwood_init(struct sock *sk)
 62 {
 63         struct westwood *w = inet_csk_ca(sk);
 64
 65         w->bk = 0;
 66         w->bw_ns_est = 0;
 67         w->bw_est = 0;
 68         w->accounted = 0;
 69         w->cumul_ack = 0;
 70         w->reset_rtt_min = 1;
 71         w->rtt_min = w->rtt = TCP_WESTWOOD_INIT_RTT;
 72         w->rtt_win_sx = tcp_time_stamp;
 73         w->snd_una = tcp_sk(sk)->snd_una;
 74         w->first_ack = 1;
 75 }
 76
 77 /*
 78  * @westwood_do_filter
 79  * Low-pass filter. Implemented using constant coefficients.
 80  */
 81 static inline u32 westwood_do_filter(u32 a, u32 b)
 82 {
 83         return ((7 * a) + b) >> 3;
 84 }
 85
 86 static void westwood_filter(struct westwood *w, u32 delta)
 87 {
 88         /* If the filter is empty fill it with the first sample of bandwidth  */
 89         if (w->bw_ns_est == 0 && w->bw_est == 0) {
 90                 w->bw_ns_est = w->bk / delta;
 91                 w->bw_est = w->bw_ns_est;
 92         } else {
 93                 w->bw_ns_est = westwood_do_filter(w->bw_ns_est, w->bk / delta);
 94                 w->bw_est = westwood_do_filter(w->bw_est, w->bw_ns_est);
 95         }
 96 }
 97
 98 /*
 99  * @westwood_pkts_acked
100  * Called after processing group of packets.
101  * but all westwood needs is the last sample of srtt.
102  */
103 static void tcp_westwood_pkts_acked(struct sock *sk, u32 cnt, s32 rtt)
104 {
105         struct westwood *w = inet_csk_ca(sk);
106
107         if (rtt > 0)
108                 w->rtt = usecs_to_jiffies(rtt);
```

```
109 }
110
111 /*
112  * @westwood_update_window
113  * It updates RTT evaluation window if it is the right moment to do
114  * it. If so it calls filter for evaluating bandwidth.
115  */
116 static void westwood_update_window(struct sock *sk)
117 {
118         struct westwood *w = inet_csk_ca(sk);
119         s32 delta = tcp_time_stamp - w->rtt_win_sx;
120
121         /* Initialize w->snd_una with the first acked sequence number in order
122          * to fix mismatch between tp->snd_una and w->snd_una for the first
123          * bandwidth sample
124          */
125         if (w->first_ack) {
126                 w->snd_una = tcp_sk(sk)->snd_una;
127                 w->first_ack = 0;
128         }
129
130         /*
131          * See if a RTT-window has passed.
132          * Be careful since if RTT is less than
133          * 50ms we don't filter but we continue 'building the sample'.
134          * This minimum limit was chosen since an estimation on small
135          * time intervals is better to avoid...
136          * Obviously on a LAN we reasonably will always have
137          * right_bound = left_bound + WESTWOOD_RTT_MIN
138          */
139         if (w->rtt && delta > max_t(u32, w->rtt, TCP_WESTWOOD_RTT_MIN)) {
140                 westwood_filter(w, delta);
141
142                 w->bk = 0;
143                 w->rtt_win_sx = tcp_time_stamp;
144         }
145 }
146
147 static inline void update_rtt_min(struct westwood *w)
148 {
149         if (w->reset_rtt_min) {
150                 w->rtt_min = w->rtt;
151                 w->reset_rtt_min = 0;
152         } else
153                 w->rtt_min = min(w->rtt, w->rtt_min);
154 }
155
156
157 /*
158  * @westwood_fast_bw
159  * It is called when we are in fast path. In particular it is called when
160  * header prediction is successful. In such case in fact update is
161  * straight forward and doesn't need any particular care.
162  */
163 static inline void westwood_fast_bw(struct sock *sk)
164 {
165         const struct tcp_sock *tp = tcp_sk(sk);
166         struct westwood *w = inet_csk_ca(sk);
167
168         westwood_update_window(sk);
169
170         w->bk += tp->snd_una - w->snd_una;
171         w->snd_una = tp->snd_una;
172         update_rtt_min(w);
173 }
```

```
174
175 /*
176  * @westwood_acked_count
177  * This function evaluates cumul_ack for evaluating bk in case of
178  * delayed or partial acks.
179  */
180 static inline u32 westwood_acked_count(struct sock *sk)
181 {
182         const struct tcp_sock *tp = tcp_sk(sk);
183         struct westwood *w = inet_csk_ca(sk);
184
185         w->cumul_ack = tp->snd_una - w->snd_una;
186
187         /* If cumul_ack is 0 this is a dupack since it's not moving
188          * tp->snd_una.
189          */
190         if (!w->cumul_ack) {
191                 w->accounted += tp->mss_cache;
192                 w->cumul_ack = tp->mss_cache;
193         }
194
195         if (w->cumul_ack > tp->mss_cache) {
196                 /* Partial or delayed ack */
197                 if (w->accounted >= w->cumul_ack) {
198                         w->accounted -= w->cumul_ack;
199                         w->cumul_ack = tp->mss_cache;
200                 } else {
201                         w->cumul_ack -= w->accounted;
202                         w->accounted = 0;
203                 }
204         }
205
206         w->snd_una = tp->snd_una;
207
208         return w->cumul_ack;
209 }
210
211
212 /*
213  * TCP Westwood
214  * Here limit is evaluated as Bw estimation*RTTmin (for obtaining it
215  * in packets we use mss_cache). Rttmin is guaranteed to be >= 2
216  * so avoids ever returning 0.
217  */
218 static u32 tcp_westwood_bw_rttmin(const struct sock *sk)
219 {
220         const struct tcp_sock *tp = tcp_sk(sk);
221         const struct westwood *w = inet_csk_ca(sk);
222         return max_t(u32, (w->bw_est * w->rtt_min) / tp->mss_cache, 2);
223 }
224
225 static void tcp_westwood_event(struct sock *sk, enum tcp_ca_event event)
226 {
227         struct tcp_sock *tp = tcp_sk(sk);
228         struct westwood *w = inet_csk_ca(sk);
229
230         switch (event) {
231         case CA_EVENT_FAST_ACK:
232                 westwood_fast_bw(sk);
233                 break;
234
235         case CA_EVENT_COMPLETE_CWR:
236                 tp->snd_cwnd = tp->snd_ssthresh = tcp_westwood_bw_rttmin(sk);
237                 break;
238
```

```
239         case CA_EVENT_LOSS:
240                 tp->snd_ssthresh = tcp_westwood_bw_rttmin(sk);
241                 /* Update RTT_min when next ack arrives */
242                 w->reset_rtt_min = 1;
243                 break;
244
245         case CA_EVENT_SLOW_ACK:
246                 westwood_update_window(sk);
247                 w->bk += westwood_acked_count(sk);
248                 update_rtt_min(w);
249                 break;
250
251         default:
252                 /* don't care */
253                 break;
254         }
255 }
256
257
258 /* Extract info for Tcp socket info provided via netlink. */
259 static void tcp_westwood_info(struct sock *sk, u32 ext,
260                               struct sk_buff *skb)
261 {
262         const struct westwood *ca = inet_csk_ca(sk);
263         if (ext & (1 << (INET_DIAG_VEGASINFO - 1))) {
264                 struct tcpvegas_info info = {
265                         .tcpv_enabled = 1,
266                         .tcpv_rtt = jiffies_to_usecs(ca->rtt),
267                         .tcpv_minrtt = jiffies_to_usecs(ca->rtt_min),
268                 };
269
270                 nla_put(skb, INET_DIAG_VEGASINFO, sizeof(info), &info);
271         }
272 }
273
274
275 static struct tcp_congestion_ops tcp_westwood __read_mostly = {
276         .init           = tcp_westwood_init,
277         .ssthresh       = tcp_reno_ssthresh,
278         .cong_avoid     = tcp_reno_cong_avoid,
279         .cwnd_event     = tcp_westwood_event,
280         .get_info       = tcp_westwood_info,
281         .pkts_acked     = tcp_westwood_pkts_acked,
282
283         .owner          = THIS_MODULE,
284         .name           = "westwood"
285 };
286
287 static int __init tcp_westwood_register(void)
288 {
289         BUILD_BUG_ON(sizeof(struct westwood) > ICSK_CA_PRIV_SIZE);
290         return tcp_register_congestion_control(&tcp_westwood);
291 }
292
293 static void __exit tcp_westwood_unregister(void)
294 {
295         tcp_unregister_congestion_control(&tcp_westwood);
296 }
297
298 module_init(tcp_westwood_register);
299 module_exit(tcp_westwood_unregister);
300
301 MODULE_AUTHOR("Stephen Hemminger, Angelo Dell'Aera");
302 MODULE_LICENSE("GPL");
303 MODULE_DESCRIPTION("TCP Westwood+");
```

304