Linux/net/sched/sch_red.c

```
 1 /*
 2  * net/sched/sch_red.c  Random Early Detection queue.
 3  *
 4  *              This program is free software; you can redistribute it and/or
 5  *              modify it under the terms of the GNU General Public License
 6  *              as published by the Free Software Foundation; either version
 7  *              2 of the License, or (at your option) any later version.
 8  *
 9  * Authors:     Alexey Kuznetsov, <kuznet@ms2.inr.ac.ru>
10  *
11  * Changes:
12  * J Hadi Salim 980914: computation fixes
13  * Alexey Makarenko <makar@phoenix.kharkov.ua> 990814: qave on idle link was calculated incorrectly.
14  * J Hadi Salim 980816:  ECN support
15  */
16
17 #include <linux/module.h>
18 #include <linux/types.h>
19 #include <linux/kernel.h>
20 #include <linux/skbuff.h>
21 #include <net/pkt_sched.h>
22 #include <net/inet_ecn.h>
23 #include <net/red.h>
24
25
26 /*      Parameters, settable by user:
27         -----------------------------
28
29         limit           - bytes (must be > qth_max + burst)
30
31         Hard limit on queue length, should be chosen >qth_max
32         to allow packet bursts. This parameter does not
33         affect the algorithms behaviour and can be chosen
34         arbitrarily high (well, less than ram size)
35         Really, this limit will never be reached
36         if RED works correctly.
37  */
38
39 struct red_sched_data {
40         u32                     limit;          /* HARD maximal queue length */
41         unsigned char           flags;
42         struct timer_list       adapt_timer;
43         struct red_parms        parms;
44         struct red_vars         vars;
45         struct red_stats        stats;
46         struct Qdisc            *qdisc;
47 };
48
49 static inline int red_use_ecn(struct red_sched_data *q)
50 {
51         return q->flags & TC_RED_ECN;
52 }
53
54 static inline int red_use_harddrop(struct red_sched_data *q)
55 {
56         return q->flags & TC_RED_HARDDROP;
57 }
58
59 static int red_enqueue(struct sk_buff *skb, struct Qdisc *sch,
60                        struct sk_buff **to_free)
61 {
62         struct red_sched_data *q = qdisc_priv(sch);
63         struct Qdisc *child = q->qdisc;
64         int ret;
65
66         q->vars.qavg = red_calc_qavg(&q->parms,
67                                      &q->vars,
68                                      child->qstats.backlog);
69
70         if (red_is_idling(&q->vars))
71                 red_end_of_idle_period(&q->vars);
72
73         switch (red_action(&q->parms, &q->vars, q->vars.qavg)) {
74         case RED_DONT_MARK:
75                 break;
76
```

```
77          case RED_PROB_MARK:
78                  qdisc_qstats_overlimit(sch);
79                  if (!red_use_ecn(q) || !INET_ECN_set_ce(skb)) {
80                          q->stats.prob_drop++;
81                          goto congestion_drop;
82                  }
83
84                  q->stats.prob_mark++;
85                  break;
86
87          case RED_HARD_MARK:
88                  qdisc_qstats_overlimit(sch);
89                  if (red_use_harddrop(q) || !red_use_ecn(q) ||
90                      !INET_ECN_set_ce(skb)) {
91                          q->stats.forced_drop++;
92                          goto congestion_drop;
93                  }
94
95                  q->stats.forced_mark++;
96                  break;
97          }
98
99          ret = qdisc_enqueue(skb, child, to_free);
100         if (likely(ret == NET_XMIT_SUCCESS)) {
101                 qdisc_qstats_backlog_inc(sch, skb);
102                 sch->q.qlen++;
103         } else if (net_xmit_drop_count(ret)) {
104                 q->stats.pdrop++;
105                 qdisc_qstats_drop(sch);
106         }
107         return ret;
108
109 congestion_drop:
110         qdisc_drop(skb, sch, to_free);
111         return NET_XMIT_CN;
112 }
113
114 static struct sk_buff *red_dequeue(struct Qdisc *sch)
115 {
116         struct sk_buff *skb;
117         struct red_sched_data *q = qdisc_priv(sch);
118         struct Qdisc *child = q->qdisc;
119
120         skb = child->dequeue(child);
121         if (skb) {
122                 qdisc_bstats_update(sch, skb);
123                 qdisc_qstats_backlog_dec(sch, skb);
124                 sch->q.qlen--;
125         } else {
126                 if (!red_is_idling(&q->vars))
127                         red_start_of_idle_period(&q->vars);
128         }
129         return skb;
130 }
131
132 static struct sk_buff *red_peek(struct Qdisc *sch)
133 {
134         struct red_sched_data *q = qdisc_priv(sch);
135         struct Qdisc *child = q->qdisc;
136
137         return child->ops->peek(child);
138 }
139
140 static void red_reset(struct Qdisc *sch)
141 {
142         struct red_sched_data *q = qdisc_priv(sch);
143
144         qdisc_reset(q->qdisc);
145         sch->qstats.backlog = 0;
146         sch->q.qlen = 0;
147         red_restart(&q->vars);
148 }
149
150 static void red_destroy(struct Qdisc *sch)
151 {
152         struct red_sched_data *q = qdisc_priv(sch);
153
154         del_timer_sync(&q->adapt_timer);
155         qdisc_destroy(q->qdisc);
156 }
157
158 static const struct nla_policy red_policy[TCA_RED_MAX + 1] = {
159         [TCA_RED_PARMS] = { .len = sizeof(struct tc_red_qopt) },
160         [TCA_RED_STAB]  = { .len = RED_STAB_SIZE },
161         [TCA_RED_MAX_P] = { .type = NLA_U32 },
162 };
163
164 static int red_change(struct Qdisc *sch, struct nlattr *opt)
165 {
166         struct red_sched_data *q = qdisc_priv(sch);
167         struct nlattr *tb[TCA_RED_MAX + 1];
168         struct tc_red_qopt *ctl;
169         struct Qdisc *child = NULL;
170         int err;
171         u32 max_P;
```

```
172
173         if (opt == NULL)
174                 return -EINVAL;
175
176         err = nla_parse_nested(tb, TCA_RED_MAX, opt, red_policy);
177         if (err < 0)
178                 return err;
179
180         if (tb[TCA_RED_PARMS] == NULL ||
181             tb[TCA_RED_STAB] == NULL)
182                 return -EINVAL;
183
184         max_P = tb[TCA_RED_MAX_P] ? nla_get_u32(tb[TCA_RED_MAX_P]) : 0;
185
186         ctl = nla_data(tb[TCA_RED_PARMS]);
187
188         if (ctl->limit > 0) {
189                 child = fifo_create_dflt(sch, &bfifo_qdisc_ops, ctl->limit);
190                 if (IS_ERR(child))
191                         return PTR_ERR(child);
192         }
193
194         sch_tree_lock(sch);
195         q->flags = ctl->flags;
196         q->limit = ctl->limit;
197         if (child) {
198                 qdisc_tree_reduce_backlog(q->qdisc, q->qdisc->q.qlen,
199                                           q->qdisc->qstats.backlog);
200                 qdisc_destroy(q->qdisc);
201                 q->qdisc = child;
202         }
203
204         red_set_parms(&q->parms,
205                       ctl->qth_min, ctl->qth_max, ctl->Wlog,
206                       ctl->Plog, ctl->Scell_log,
207                       nla_data(tb[TCA_RED_STAB]),
208                       max_P);
209         red_set_vars(&q->vars);
210
211         del_timer(&q->adapt_timer);
212         if (ctl->flags & TC_RED_ADAPTATIVE)
213                 mod_timer(&q->adapt_timer, jiffies + HZ/2);
214
215         if (!q->qdisc->q.qlen)
216                 red_start_of_idle_period(&q->vars);
217
218         sch_tree_unlock(sch);
219         return 0;
220 }
221
222 static inline void red_adaptative_timer(unsigned long arg)
223 {
224         struct Qdisc *sch = (struct Qdisc *)arg;
225         struct red_sched_data *q = qdisc_priv(sch);
226         spinlock_t *root_lock = qdisc_lock(qdisc_root_sleeping(sch));
227
228         spin_lock(root_lock);
229         red_adaptative_algo(&q->parms, &q->vars);
230         mod_timer(&q->adapt_timer, jiffies + HZ/2);
231         spin_unlock(root_lock);
232 }
233
234 static int red_init(struct Qdisc *sch, struct nlattr *opt)
235 {
236         struct red_sched_data *q = qdisc_priv(sch);
237
238         q->qdisc = &noop_qdisc;
239         setup_timer(&q->adapt_timer, red_adaptative_timer, (unsigned long)sch);
240         return red_change(sch, opt);
241 }
242
243 static int red_dump(struct Qdisc *sch, struct sk_buff *skb)
244 {
245         struct red_sched_data *q = qdisc_priv(sch);
246         struct nlattr *opts = NULL;
247         struct tc_red_qopt opt = {
248                 .limit          = q->limit,
249                 .flags          = q->flags,
250                 .qth_min        = q->parms.qth_min >> q->parms.Wlog,
251                 .qth_max        = q->parms.qth_max >> q->parms.Wlog,
252                 .Wlog           = q->parms.Wlog,
253                 .Plog           = q->parms.Plog,
254                 .Scell_log      = q->parms.Scell_log,
255         };
256
257         sch->qstats.backlog = q->qdisc->qstats.backlog;
258         opts = nla_nest_start(skb, TCA_OPTIONS);
259         if (opts == NULL)
260                 goto nla_put_failure;
261         if (nla_put(skb, TCA_RED_PARMS, sizeof(opt), &opt) ||
262             nla_put_u32(skb, TCA_RED_MAX_P, q->parms.max_P))
263                 goto nla_put_failure;
264         return nla_nest_end(skb, opts);
265
266 nla_put_failure:
```

```
267         nla_nest_cancel(skb, opts);
268         return -EMSGSIZE;
269 }
270
271 static int red_dump_stats(struct Qdisc *sch, struct gnet_dump *d)
272 {
273         struct red_sched_data *q = qdisc_priv(sch);
274         struct tc_red_xstats st = {
275                 .early  = q->stats.prob_drop + q->stats.forced_drop,
276                 .pdrop  = q->stats.pdrop,
277                 .other  = q->stats.other,
278                 .marked = q->stats.prob_mark + q->stats.forced_mark,
279         };
280
281         return gnet_stats_copy_app(d, &st, sizeof(st));
282 }
283
284 static int red_dump_class(struct Qdisc *sch, unsigned long cl,
285                           struct sk_buff *skb, struct tcmsg *tcm)
286 {
287         struct red_sched_data *q = qdisc_priv(sch);
288
289         tcm->tcm_handle |= TC_H_MIN(1);
290         tcm->tcm_info = q->qdisc->handle;
291         return 0;
292 }
293
294 static int red_graft(struct Qdisc *sch, unsigned long arg, struct Qdisc *new,
295                      struct Qdisc **old)
296 {
297         struct red_sched_data *q = qdisc_priv(sch);
298
299         if (new == NULL)
300                 new = &noop_qdisc;
301
302         *old = qdisc_replace(sch, new, &q->qdisc);
303         return 0;
304 }
305
306 static struct Qdisc *red_leaf(struct Qdisc *sch, unsigned long arg)
307 {
308         struct red_sched_data *q = qdisc_priv(sch);
309         return q->qdisc;
310 }
311
312 static unsigned long red_get(struct Qdisc *sch, u32 classid)
313 {
314         return 1;
315 }
316
317 static void red_put(struct Qdisc *sch, unsigned long arg)
318 {
319 }
320
321 static void red_walk(struct Qdisc *sch, struct qdisc_walker *walker)
322 {
323         if (!walker->stop) {
324                 if (walker->count >= walker->skip)
325                         if (walker->fn(sch, 1, walker) < 0) {
326                                 walker->stop = 1;
327                                 return;
328                         }
329                 walker->count++;
330         }
331 }
332
333 static const struct Qdisc_class_ops red_class_ops = {
334         .graft          =       red_graft,
335         .leaf           =       red_leaf,
336         .get            =       red_get,
337         .put            =       red_put,
338         .walk           =       red_walk,
339         .dump           =       red_dump_class,
340 };
341
342 static struct Qdisc_ops red_qdisc_ops __read_mostly = {
343         .id             =       "red",
344         .priv_size      =       sizeof(struct red_sched_data),
345         .cl_ops         =       &red_class_ops,
346         .enqueue        =       red_enqueue,
347         .dequeue        =       red_dequeue,
348         .peek           =       red_peek,
349         .init           =       red_init,
350         .reset          =       red_reset,
351         .destroy        =       red_destroy,
352         .change         =       red_change,
353         .dump           =       red_dump,
354         .dump_stats     =       red_dump_stats,
355         .owner          =       THIS_MODULE,
356 };
357
358 static int __init red_module_init(void)
359 {
360         return register_qdisc(&red_qdisc_ops);
361 }
```

```
362
363 static void __exit red_module_exit(void)
364 {
365         unregister_qdisc(&red_qdisc_ops);
366 }
367
368 module_init(red_module_init)
369 module_exit(red_module_exit)
370
371 MODULE_LICENSE("GPL");
372
```