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_cong.c

```
 1 /*
 2  * Plugable TCP congestion control support and newReno
 3  * congestion control.
 4  * Based on ideas from I/O scheduler support and Web100.
 5  *
 6  * Copyright (C) 2005 Stephen Hemminger <shemminger@osdl.org>
 7  */
 8
 9 #define pr_fmt(fmt) "TCP: " fmt
10
11 #include <linux/module.h>
12 #include <linux/mm.h>
13 #include <linux/types.h>
14 #include <linux/list.h>
15 #include <linux/gfp.h>
16 #include <net/tcp.h>
17
18 static DEFINE_SPINLOCK(tcp_cong_list_lock);
19 static LIST_HEAD(tcp_cong_list);
20
21 /* Simple linear search, don't expect many entries! */
22 static struct tcp_congestion_ops *tcp_ca_find(const char *name)
23 {
24         struct tcp_congestion_ops *e;
25
26         list_for_each_entry_rcu(e, &tcp_cong_list, list) {
27                 if (strcmp(e->name, name) == 0)
28                         return e;
29         }
30
31         return NULL;
32 }
33
34 /*
35  * Attach new congestion control algorithm to the list
36  * of available options.
37  */
38 int tcp_register_congestion_control(struct tcp_congestion_ops *ca)
39 {
40         int ret = 0;
41
42         /* all algorithms must implement ssthresh and cong_avoid ops */
43         if (!ca->ssthresh || !ca->cong_avoid) {
```

```
44                 pr_err("%s does not implement required ops\n", ca->name);
45                 return -EINVAL;
46         }
47
48         spin_lock(&tcp_cong_list_lock);
49         if (tcp_ca_find(ca->name)) {
50                 pr_notice("%s already registered\n", ca->name);
51                 ret = -EEXIST;
52         } else {
53                 list_add_tail_rcu(&ca->list, &tcp_cong_list);
54                 pr_info("%s registered\n", ca->name);
55         }
56         spin_unlock(&tcp_cong_list_lock);
57
58         return ret;
59 }
60 EXPORT_SYMBOL_GPL(tcp_register_congestion_control);
61
62 /*
63  * Remove congestion control algorithm, called from
64  * the module's remove function.  Module ref counts are used
65  * to ensure that this can't be done till all sockets using
66  * that method are closed.
67  */
68 void tcp_unregister_congestion_control(struct tcp_congestion_ops *ca)
69 {
70         spin_lock(&tcp_cong_list_lock);
71         list_del_rcu(&ca->list);
72         spin_unlock(&tcp_cong_list_lock);
73 }
74 EXPORT_SYMBOL_GPL(tcp_unregister_congestion_control);
75
76 /* Assign choice of congestion control. */
77 void tcp_init_congestion_control(struct sock *sk)
78 {
79         struct inet_connection_sock *icsk = inet_csk(sk);
80         struct tcp_congestion_ops *ca;
81
82         /* if no choice made yet assign the current value set as default */
83         if (icsk->icsk_ca_ops == &tcp_init_congestion_ops) {
84                 rcu_read_lock();
85                 list_for_each_entry_rcu(ca, &tcp_cong_list, list) {
86                         if (try_module_get(ca->owner)) {
87                                 icsk->icsk_ca_ops = ca;
88                                 break;
89                         }
90
91                         /* fallback to next available */
92                 }
93                 rcu_read_unlock();
94         }
95
96         if (icsk->icsk_ca_ops->init)
97                 icsk->icsk_ca_ops->init(sk);
98 }
99
100 /* Manage refcounts on socket close. */
101 void tcp_cleanup_congestion_control(struct sock *sk)
102 {
103         struct inet_connection_sock *icsk = inet_csk(sk);
104
105         if (icsk->icsk_ca_ops->release)
106                 icsk->icsk_ca_ops->release(sk);
107         module_put(icsk->icsk_ca_ops->owner);
108 }
```

```
109
110 /* Used by sysctl to change default congestion control */
111 int tcp_set_default_congestion_control(const char *name)
112 {
113         struct tcp_congestion_ops *ca;
114         int ret = -ENOENT;
115
116         spin_lock(&tcp_cong_list_lock);
117         ca = tcp_ca_find(name);
118 #ifdef CONFIG_MODULES
119         if (!ca && capable(CAP_NET_ADMIN)) {
120                 spin_unlock(&tcp_cong_list_lock);
121
122                 request_module("tcp_%s", name);
123                 spin_lock(&tcp_cong_list_lock);
124                 ca = tcp_ca_find(name);
125         }
126 #endif
127
128         if (ca) {
129                 ca->flags |= TCP_CONG_NON_RESTRICTED;   /* default is always allowed */
130                 list_move(&ca->list, &tcp_cong_list);
131                 ret = 0;
132         }
133         spin_unlock(&tcp_cong_list_lock);
134
135         return ret;
136 }
137
138 /* Set default value from kernel configuration at bootup */
139 static int __init tcp_congestion_default(void)
140 {
141         return tcp_set_default_congestion_control(CONFIG_DEFAULT_TCP_CONG);
142 }
143 late_initcall(tcp_congestion_default);
144
145
146 /* Build string with list of available congestion control values */
147 void tcp_get_available_congestion_control(char *buf, size_t maxlen)
148 {
149         struct tcp_congestion_ops *ca;
150         size_t offs = 0;
151
152         rcu_read_lock();
153         list_for_each_entry_rcu(ca, &tcp_cong_list, list) {
154                 offs += snprintf(buf + offs, maxlen - offs,
155                                  "%s%s",
156                                  offs == 0 ? "" : " ", ca->name);
157
158         }
159         rcu_read_unlock();
160 }
161
162 /* Get current default congestion control */
163 void tcp_get_default_congestion_control(char *name)
164 {
165         struct tcp_congestion_ops *ca;
166         /* We will always have reno... */
167         BUG_ON(list_empty(&tcp_cong_list));
168
169         rcu_read_lock();
170         ca = list_entry(tcp_cong_list.next, struct tcp_congestion_ops, list);
171         strncpy(name, ca->name, TCP_CA_NAME_MAX);
172         rcu_read_unlock();
173 }
```

```
174
175 /* Built list of non-restricted congestion control values */
176 void tcp_get_allowed_congestion_control(char *buf, size_t maxlen)
177 {
178         struct tcp_congestion_ops *ca;
179         size_t offs = 0;
180
181         *buf = '\0';
182         rcu_read_lock();
183         list_for_each_entry_rcu(ca, &tcp_cong_list, list) {
184                 if (!(ca->flags & TCP_CONG_NON_RESTRICTED))
185                         continue;
186                 offs += snprintf(buf + offs, maxlen - offs,
187                                  "%s%s",
188                                  offs == 0 ? "" : " ", ca->name);
189
190         }
191         rcu_read_unlock();
192 }
193
194 /* Change list of non-restricted congestion control */
195 int tcp_set_allowed_congestion_control(char *val)
196 {
197         struct tcp_congestion_ops *ca;
198         char *saved_clone, *clone, *name;
199         int ret = 0;
200
201         saved_clone = clone = kstrdup(val, GFP_USER);
202         if (!clone)
203                 return -ENOMEM;
204
205         spin_lock(&tcp_cong_list_lock);
206         /* pass 1 check for bad entries */
207         while ((name = strsep(&clone, " ")) && *name) {
208                 ca = tcp_ca_find(name);
209                 if (!ca) {
210                         ret = -ENOENT;
211                         goto out;
212                 }
213         }
214
215         /* pass 2 clear old values */
216         list_for_each_entry_rcu(ca, &tcp_cong_list, list)
217                 ca->flags &= ~TCP_CONG_NON_RESTRICTED;
218
219         /* pass 3 mark as allowed */
220         while ((name = strsep(&val, " ")) && *name) {
221                 ca = tcp_ca_find(name);
222                 WARN_ON(!ca);
223                 if (ca)
224                         ca->flags |= TCP_CONG_NON_RESTRICTED;
225         }
226 out:
227         spin_unlock(&tcp_cong_list_lock);
228         kfree(saved_clone);
229
230         return ret;
231 }
232
233
234 /* Change congestion control for socket */
235 int tcp_set_congestion_control(struct sock *sk, const char *name)
236 {
237         struct inet_connection_sock *icsk = inet_csk(sk);
238         struct tcp_congestion_ops *ca;
```

```
239         int err = 0;
240
241         rcu_read_lock();
242         ca = tcp_ca_find(name);
243
244         /* no change asking for existing value */
245         if (ca == icsk->icsk_ca_ops)
246                 goto out;
247
248 #ifdef CONFIG_MODULES
249         /* not found attempt to autoload module */
250         if (!ca && capable(CAP_NET_ADMIN)) {
251                 rcu_read_unlock();
252                 request_module("tcp_%s", name);
253                 rcu_read_lock();
254                 ca = tcp_ca_find(name);
255         }
256 #endif
257         if (!ca)
258                 err = -ENOENT;
259
260         else if (!((ca->flags & TCP_CONG_NON_RESTRICTED) ||
261                    ns_capable(sock_net(sk)->user_ns, CAP_NET_ADMIN)))
262                 err = -EPERM;
263
264         else if (!try_module_get(ca->owner))
265                 err = -EBUSY;
266
267         else {
268                 tcp_cleanup_congestion_control(sk);
269                 icsk->icsk_ca_ops = ca;
270
271                 if (sk->sk_state != TCP_CLOSE && icsk->icsk_ca_ops->init)
272                         icsk->icsk_ca_ops->init(sk);
273         }
274  out:
275         rcu_read_unlock();
276         return err;
277 }
278
279 /* Slow start is used when congestion window is no greater than the slow start
280  * threshold. We base on RFC2581 and also handle stretch ACKs properly.
281  * We do not implement RFC3465 Appropriate Byte Counting (ABC) per se but
282  * something better;) a packet is only considered (s)acked in its entirety to
283  * defend the ACK attacks described in the RFC. Slow start processes a stretch
284  * ACK of degree N as if N acks of degree 1 are received back to back except
285  * ABC caps N to 2. Slow start exits when cwnd grows over ssthresh and
286  * returns the leftover acks to adjust cwnd in congestion avoidance mode.
287  */
288 int tcp_slow_start(struct tcp_sock *tp, u32 acked)
289 {
290         u32 cwnd = tp->snd_cwnd + acked;
291
292         if (cwnd > tp->snd_ssthresh)
293                 cwnd = tp->snd_ssthresh + 1;
294         acked -= cwnd - tp->snd_cwnd;
295         tp->snd_cwnd = min(cwnd, tp->snd_cwnd_clamp);
296         return acked;
297 }
298 EXPORT_SYMBOL_GPL(tcp_slow_start);
299
300 /* In theory this is tp->snd_cwnd += 1 / tp->snd_cwnd (or alternative w) */
301 void tcp_cong_avoid_ai(struct tcp_sock *tp, u32 w)
302 {
303         if (tp->snd_cwnd_cnt >= w) {
```

```
304                 if (tp->snd_cwnd < tp->snd_cwnd_clamp)
305                         tp->snd_cwnd++;
306                 tp->snd_cwnd_cnt = 0;
307         } else {
308                 tp->snd_cwnd_cnt++;
309         }
310 }
311 EXPORT_SYMBOL_GPL(tcp_cong_avoid_ai);
312
313 /*
314  * TCP Reno congestion control
315  * This is special case used for fallback as well.
316  */
317 /* This is Jacobson's slow start and congestion avoidance.
318  * SIGCOMM '88, p. 328.
319  */
320 void tcp_reno_cong_avoid(struct sock *sk, u32 ack, u32 acked)
321 {
322         struct tcp_sock *tp = tcp_sk(sk);
323
324         if (!tcp_is_cwnd_limited(sk))
325                 return;
326
327         /* In "safe" area, increase. */
328         if (tp->snd_cwnd <= tp->snd_ssthresh)
329                 tcp_slow_start(tp, acked);
330         /* In dangerous area, increase slowly. */
331         else
332                 tcp_cong_avoid_ai(tp, tp->snd_cwnd);
333 }
334 EXPORT_SYMBOL_GPL(tcp_reno_cong_avoid);
335
336 /* Slow start threshold is half the congestion window (min 2) */
337 u32 tcp_reno_ssthresh(struct sock *sk)
338 {
339         const struct tcp_sock *tp = tcp_sk(sk);
340         return max(tp->snd_cwnd >> 1U, 2U);
341 }
342 EXPORT_SYMBOL_GPL(tcp_reno_ssthresh);
343
344 struct tcp_congestion_ops tcp_reno = {
345         .flags          = TCP_CONG_NON_RESTRICTED,
346         .name           = "reno",
347         .owner          = THIS_MODULE,
348         .ssthresh       = tcp_reno_ssthresh,
349         .cong_avoid     = tcp_reno_cong_avoid,
350 };
351
352 /* Initial congestion control used (until SYN)
353  * really reno under another name so we can tell difference
354  * during tcp_set_default_congestion_control
355  */
356 struct tcp_congestion_ops tcp_init_congestion_ops  = {
357         .name           = "",
358         .owner          = THIS_MODULE,
359         .ssthresh       = tcp_reno_ssthresh,
360         .cong_avoid     = tcp_reno_cong_avoid,
361 };
362 EXPORT_SYMBOL_GPL(tcp_init_congestion_ops);
363
```