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_timer.c

```
 1 /*
 2  * INET         An implementation of the TCP/IP protocol suite for the LINUX
 3  *              operating system.  INET is implemented using the  BSD Socket
 4  *              interface as the means of communication with the user level.
 5  *
 6  *              Implementation of the Transmission Control Protocol(TCP).
 7  *
 8  * Authors:     Ross Biro
 9  *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
10  *              Mark Evans, <evansmp@uhura.aston.ac.uk>
11  *              Corey Minyard <wf-rch!minyard@relay.EU.net>
12  *              Florian La Roche, <flla@stud.uni-sb.de>
13  *              Charles Hedrick, <hedrick@klinzhai.rutgers.edu>
14  *              Linus Torvalds, <torvalds@cs.helsinki.fi>
15  *              Alan Cox, <gw4pts@gw4pts.ampr.org>
16  *              Matthew Dillon, <dillon@apollo.west.oic.com>
17  *              Arnt Gulbrandsen, <agulbra@nvg.unit.no>
18  *              Jorge Cwik, <jorge@laser.satlink.net>
19  */
20
21 #include <linux/module.h>
22 #include <linux/gfp.h>
23 #include <net/tcp.h>
24
25 int sysctl_tcp_syn_retries __read_mostly = TCP_SYN_RETRIES;
26 int sysctl_tcp_synack_retries __read_mostly = TCP_SYNACK_RETRIES;
27 int sysctl_tcp_keepalive_time __read_mostly = TCP_KEEPALIVE_TIME;
28 int sysctl_tcp_keepalive_probes __read_mostly = TCP_KEEPALIVE_PROBES;
29 int sysctl_tcp_keepalive_intvl __read_mostly = TCP_KEEPALIVE_INTVL;
30 int sysctl_tcp_retries1 __read_mostly = TCP_RETR1;
31 int sysctl_tcp_retries2 __read_mostly = TCP_RETR2;
32 int sysctl_tcp_orphan_retries __read_mostly;
33 int sysctl_tcp_thin_linear_timeouts __read_mostly;
34
35 static void tcp_write_err(struct sock *sk)
36 {
37         sk->sk_err = sk->sk_err_soft ? : ETIMEDOUT;
38         sk->sk_error_report(sk);
39
40         tcp_done(sk);
41         NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPABORTONTIMEOUT);
42 }
43
44 /* Do not allow orphaned sockets to eat all our resources.
45  * This is direct violation of TCP specs, but it is required
46  * to prevent DoS attacks. It is called when a retransmission timeout
47  * or zero probe timeout occurs on orphaned socket.
48  *
49  * Criteria is still not confirmed experimentally and may change.
50  * We kill the socket, if:
51  * 1. If number of orphaned sockets exceeds an administratively configured
52  *    limit.
53  * 2. If we have strong memory pressure.
54  */
55 static int tcp_out_of_resources(struct sock *sk, int do_reset)
56 {
57         struct tcp_sock *tp = tcp_sk(sk);
58         int shift = 0;
59
60         /* If peer does not open window for long time, or did not transmit
61          * anything for long time, penalize it. */
62         if ((s32)(tcp_time_stamp - tp->lsndtime) > 2*TCP_RTO_MAX || !do_reset)
63                 shift++;
64
65         /* If some dubious ICMP arrived, penalize even more. */
66         if (sk->sk_err_soft)
67                 shift++;
```

```
68
69          if (tcp_check_oom(sk, shift)) {
70                  /* Catch exceptional cases, when connection requires reset.
71                   *      1. Last segment was sent recently. */
72                  if ((s32)(tcp_time_stamp - tp->lsndtime) <= TCP_TIMEWAIT_LEN ||
73                      /*  2. Window is closed. */
74                      (!tp->snd_wnd && !tp->packets_out))
75                          do_reset = 1;
76                  if (do_reset)
77                          tcp_send_active_reset(sk, GFP_ATOMIC);
78                  tcp_done(sk);
79                  NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPABORTONMEMORY);
80                  return 1;
81          }
82          return 0;
83  }
84
85  /* Calculate maximal number or retries on an orphaned socket. */
86  static int tcp_orphan_retries(struct sock *sk, int alive)
87  {
88          int retries = sysctl_tcp_orphan_retries; /* May be zero. */
89
90          /* We know from an ICMP that something is wrong. */
91          if (sk->sk_err_soft && !alive)
92                  retries = 0;
93
94          /* However, if socket sent something recently, select some safe
95           * number of retries. 8 corresponds to >100 seconds with minimal
96           * RTO of 200msec. */
97          if (retries == 0 && alive)
98                  retries = 8;
99          return retries;
100 }
101
102 static void tcp_mtu_probing(struct inet_connection_sock *icsk, struct sock *sk)
103 {
104         /* Black hole detection */
105         if (sysctl_tcp_mtu_probing) {
106                 if (!icsk->icsk_mtup.enabled) {
107                         icsk->icsk_mtup.enabled = 1;
108                         tcp_sync_mss(sk, icsk->icsk_pmtu_cookie);
109                 } else {
110                         struct tcp_sock *tp = tcp_sk(sk);
111                         int mss;
112
113                         mss = tcp_mtu_to_mss(sk, icsk->icsk_mtup.search_low) >> 1;
114                         mss = min(sysctl_tcp_base_mss, mss);
115                         mss = max(mss, 68 - tp->tcp_header_len);
116                         icsk->icsk_mtup.search_low = tcp_mss_to_mtu(sk, mss);
117                         tcp_sync_mss(sk, icsk->icsk_pmtu_cookie);
118                 }
119         }
120 }
121
122 /* This function calculates a "timeout" which is equivalent to the timeout of a
123  * TCP connection after "boundary" unsuccessful, exponentially backed-off
124  * retransmissions with an initial RTO of TCP_RTO_MIN or TCP_TIMEOUT_INIT if
125  * syn_set flag is set.
126  */
127 static bool retransmits_timed_out(struct sock *sk,
128                                   unsigned int boundary,
129                                   unsigned int timeout,
130                                   bool syn_set)
131 {
132         unsigned int linear_backoff_thresh, start_ts;
133         unsigned int rto_base = syn_set ? TCP_TIMEOUT_INIT : TCP_RTO_MIN;
134
135         if (!inet_csk(sk)->icsk_retransmits)
136                 return false;
137
138         if (unlikely(!tcp_sk(sk)->retrans_stamp))
139                 start_ts = TCP_SKB_CB(tcp_write_queue_head(sk))->when;
140         else
141                 start_ts = tcp_sk(sk)->retrans_stamp;
142
143         if (likely(timeout == 0)) {
144                 linear_backoff_thresh = ilog2(TCP_RTO_MAX/rto_base);
145
146                 if (boundary <= linear_backoff_thresh)
147                         timeout = ((2 << boundary) - 1) * rto_base;
148                 else
149                         timeout = ((2 << linear_backoff_thresh) - 1) * rto_base +
150                                 (boundary - linear_backoff_thresh) * TCP_RTO_MAX;
151         }
152         return (tcp_time_stamp - start_ts) >= timeout;
153 }
154
155 /* A write timeout has occurred. Process the after effects. */
```

```
156 static int tcp_write_timeout(struct sock *sk)
157 {
158         struct inet_connection_sock *icsk = inet_csk(sk);
159         struct tcp_sock *tp = tcp_sk(sk);
160         int retry_until;
161         bool do_reset, syn_set = false;
162
163         if ((1 << sk->sk_state) & (TCPF_SYN_SENT | TCPF_SYN_RECV)) {
164                 if (icsk->icsk_retransmits) {
165                         dst_negative_advice(sk);
166                         if (tp->syn_fastopen || tp->syn_data)
167                                 tcp_fastopen_cache_set(sk, 0, NULL, true);
168                         if (tp->syn_data)
169                                 NET_INC_STATS_BH(sock_net(sk),
170                                                  LINUX_MIB_TCPFASTOPENACTIVEFAIL);
171                 }
172                 retry_until = icsk->icsk_syn_retries ? : sysctl_tcp_syn_retries;
173                 syn_set = true;
174         } else {
175                 if (retransmits_timed_out(sk, sysctl_tcp_retries1, 0, 0)) {
176                         /* Black hole detection */
177                         tcp_mtu_probing(icsk, sk);
178
179                         dst_negative_advice(sk);
180                 }
181
182                 retry_until = sysctl_tcp_retries2;
183                 if (sock_flag(sk, SOCK_DEAD)) {
184                         const int alive = (icsk->icsk_rto < TCP_RTO_MAX);
185
186                         retry_until = tcp_orphan_retries(sk, alive);
187                         do_reset = alive ||
188                                 !retransmits_timed_out(sk, retry_until, 0, 0);
189
190                         if (tcp_out_of_resources(sk, do_reset))
191                                 return 1;
192                 }
193         }
194
195         if (retransmits_timed_out(sk, retry_until,
196                                   syn_set ? 0 : icsk->icsk_user_timeout, syn_set)) {
197                 /* Has it gone just too far? */
198                 tcp_write_err(sk);
199                 return 1;
200         }
201         return 0;
202 }
203
204 void tcp_delack_timer_handler(struct sock *sk)
205 {
206         struct tcp_sock *tp = tcp_sk(sk);
207         struct inet_connection_sock *icsk = inet_csk(sk);
208
209         sk_mem_reclaim_partial(sk);
210
211         if (sk->sk_state == TCP_CLOSE || !(icsk->icsk_ack.pending & ICSK_ACK_TIMER))
212                 goto out;
213
214         if (time_after(icsk->icsk_ack.timeout, jiffies)) {
215                 sk_reset_timer(sk, &icsk->icsk_delack_timer, icsk->icsk_ack.timeout);
216                 goto out;
217         }
218         icsk->icsk_ack.pending &= ~ICSK_ACK_TIMER;
219
220         if (!skb_queue_empty(&tp->ucopy.prequeue)) {
221                 struct sk_buff *skb;
222
223                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPSCHEDULERFAILED);
224
225                 while ((skb = __skb_dequeue(&tp->ucopy.prequeue)) != NULL)
226                         sk_backlog_rcv(sk, skb);
227
228                 tp->ucopy.memory = 0;
229         }
230
231         if (inet_csk_ack_scheduled(sk)) {
232                 if (!icsk->icsk_ack.pingpong) {
233                         /* Delayed ACK missed: inflate ATO. */
234                         icsk->icsk_ack.ato = min(icsk->icsk_ack.ato << 1, icsk->icsk_rto);
235                 } else {
236                         /* Delayed ACK missed: leave pingpong mode and
237                          * deflate ATO.
238                          */
239                         icsk->icsk_ack.pingpong = 0;
240                         icsk->icsk_ack.ato      = TCP_ATO_MIN;
241                 }
242                 tcp_send_ack(sk);
243                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_DELAYEDACKS);
```

```
244         }
245
246 out:
247         if (sk_under_memory_pressure(sk))
248                 sk_mem_reclaim(sk);
249 }
250
251 static void tcp_delack_timer(unsigned long data)
252 {
253         struct sock *sk = (struct sock *)data;
254
255         bh_lock_sock(sk);
256         if (!sock_owned_by_user(sk)) {
257                 tcp_delack_timer_handler(sk);
258         } else {
259                 inet_csk(sk)->icsk_ack.blocked = 1;
260                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_DELAYEDACKLOCKED);
261                 /* deleguate our work to tcp_release_cb() */
262                 if (!test_and_set_bit(TCP_DELACK_TIMER_DEFERRED, &tcp_sk(sk)->tsq_flags))
263                         sock_hold(sk);
264         }
265         bh_unlock_sock(sk);
266         sock_put(sk);
267 }
268
269 static void tcp_probe_timer(struct sock *sk)
270 {
271         struct inet_connection_sock *icsk = inet_csk(sk);
272         struct tcp_sock *tp = tcp_sk(sk);
273         int max_probes;
274
275         if (tp->packets_out || !tcp_send_head(sk)) {
276                 icsk->icsk_probes_out = 0;
277                 return;
278         }
279
280         /* *WARNING* RFC 1122 forbids this
281          *
282          * It doesn't AFAIK, because we kill the retransmit timer -AK
283          *
284          * FIXME: We ought not to do it, Solaris 2.5 actually has fixing
285          * this behaviour in Solaris down as a bug fix. [AC]
286          *
287          * Let me to explain. icsk_probes_out is zeroed by incoming ACKs
288          * even if they advertise zero window. Hence, connection is killed only
289          * if we received no ACKs for normal connection timeout. It is not killed
290          * only because window stays zero for some time, window may be zero
291          * until armageddon and even later. We are in full accordance
292          * with RFCs, only probe timer combines both retransmission timeout
293          * and probe timeout in one bottle.                              --ANK
294          */
295         max_probes = sysctl_tcp_retries2;
296
297         if (sock_flag(sk, SOCK_DEAD)) {
298                 const int alive = ((icsk->icsk_rto << icsk->icsk_backoff) < TCP_RTO_MAX);
299
300                 max_probes = tcp_orphan_retries(sk, alive);
301
302                 if (tcp_out_of_resources(sk, alive || icsk->icsk_probes_out <= max_probes))
303                         return;
304         }
305
306         if (icsk->icsk_probes_out > max_probes) {
307                 tcp_write_err(sk);
308         } else {
309                 /* Only send another probe if we didn't close things up. */
310                 tcp_send_probe0(sk);
311         }
312 }
313
314 /*
315  *      Timer for Fast Open socket to retransmit SYNACK. Note that the
316  *      sk here is the child socket, not the parent (listener) socket.
317  */
318 static void tcp_fastopen_synack_timer(struct sock *sk)
319 {
320         struct inet_connection_sock *icsk = inet_csk(sk);
321         int max_retries = icsk->icsk_syn_retries ? :
322             sysctl_tcp_synack_retries + 1; /* add one more retry for fastopen */
323         struct request_sock *req;
324
325         req = tcp_sk(sk)->fastopen_rsk;
326         req->rsk_ops->syn_ack_timeout(sk, req);
327
328         if (req->num_timeout >= max_retries) {
329                 tcp_write_err(sk);
330                 return;
331         }
```

```
332         /* XXX (TFO) - Unlike regular SYN-ACK retransmit, we ignore error
333          * returned from rtx_syn_ack() to make it more persistent like
334          * regular retransmit because if the child socket has been accepted
335          * it's not good to give up too easily.
336          */
337         inet_rtx_syn_ack(sk, req);
338         req->num_timeout++;
339         inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
340                           TCP_TIMEOUT_INIT << req->num_timeout, TCP_RTO_MAX);
341 }
342
343 /*
344  *      The TCP retransmit timer.
345  */
346
347 void tcp_retransmit_timer(struct sock *sk)
348 {
349         struct tcp_sock *tp = tcp_sk(sk);
350         struct inet_connection_sock *icsk = inet_csk(sk);
351
352         if (tp->fastopen_rsk) {
353                 WARN_ON_ONCE(sk->sk_state != TCP_SYN_RECV &&
354                              sk->sk_state != TCP_FIN_WAIT1);
355                 tcp_fastopen_synack_timer(sk);
356                 /* Before we receive ACK to our SYN-ACK don't retransmit
357                  * anything else (e.g., data or FIN segments).
358                  */
359                 return;
360         }
361         if (!tp->packets_out)
362                 goto out;
363
364         WARN_ON(tcp_write_queue_empty(sk));
365
366         tp->tlp_high_seq = 0;
367
368         if (!tp->snd_wnd && !sock_flag(sk, SOCK_DEAD) &&
369             !((1 << sk->sk_state) & (TCPF_SYN_SENT | TCPF_SYN_RECV))) {
370                 /* Receiver dastardly shrinks window. Our retransmits
371                  * become zero probes, but we should not timeout this
372                  * connection. If the socket is an orphan, time it out,
373                  * we cannot allow such beasts to hang infinitely.
374                  */
375                 struct inet_sock *inet = inet_sk(sk);
376                 if (sk->sk_family == AF_INET) {
377                         LIMIT_NETDEBUG(KERN_DEBUG pr_fmt("Peer %pI4:%u/%u unexpectedly shrunk window %u:%u (repaired)\n"),
378                                        &inet->inet_daddr,
379                                        ntohs(inet->inet_dport), inet->inet_num,
380                                        tp->snd_una, tp->snd_nxt);
381                 }
382 #if IS_ENABLED(CONFIG_IPV6)
383                 else if (sk->sk_family == AF_INET6) {
384                         LIMIT_NETDEBUG(KERN_DEBUG pr_fmt("Peer %pI6:%u/%u unexpectedly shrunk window %u:%u (repaired)\n"),
385                                        &sk->sk_v6_daddr,
386                                        ntohs(inet->inet_dport), inet->inet_num,
387                                        tp->snd_una, tp->snd_nxt);
388                 }
389 #endif
390                 if (tcp_time_stamp - tp->rcv_tstamp > TCP_RTO_MAX) {
391                         tcp_write_err(sk);
392                         goto out;
393                 }
394                 tcp_enter_loss(sk);
395                 tcp_retransmit_skb(sk, tcp_write_queue_head(sk));
396                 __sk_dst_reset(sk);
397                 goto out_reset_timer;
398         }
399
400         if (tcp_write_timeout(sk))
401                 goto out;
402
403         if (icsk->icsk_retransmits == 0) {
404                 int mib_idx;
405
406                 if (icsk->icsk_ca_state == TCP_CA_Recovery) {
407                         if (tcp_is_sack(tp))
408                                 mib_idx = LINUX_MIB_TCPSACKRECOVERYFAIL;
409                         else
410                                 mib_idx = LINUX_MIB_TCPRENORECOVERYFAIL;
411                 } else if (icsk->icsk_ca_state == TCP_CA_Loss) {
412                         mib_idx = LINUX_MIB_TCPLOSSFAILURES;
413                 } else if ((icsk->icsk_ca_state == TCP_CA_Disorder) ||
414                            tp->sacked_out) {
415                         if (tcp_is_sack(tp))
416                                 mib_idx = LINUX_MIB_TCPSACKFAILURES;
417                         else
418                                 mib_idx = LINUX_MIB_TCPRENOFAILURES;
419                 } else {
```

```
420                         mib_idx = LINUX_MIB_TCPTIMEOUTS;
421                 }
422                 NET_INC_STATS_BH(sock_net(sk), mib_idx);
423         }
424
425         tcp_enter_loss(sk);
426
427         if (tcp_retransmit_skb(sk, tcp_write_queue_head(sk)) > 0) {
428                 /* Retransmission failed because of local congestion,
429                  * do not backoff.
430                  */
431                 if (!icsk->icsk_retransmits)
432                         icsk->icsk_retransmits = 1;
433                 inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
434                                           min(icsk->icsk_rto, TCP_RESOURCE_PROBE_INTERVAL),
435                                           TCP_RTO_MAX);
436                 goto out;
437         }
438
439         /* Increase the timeout each time we retransmit.  Note that
440          * we do not increase the rtt estimate.  rto is initialized
441          * from rtt, but increases here.  Jacobson (SIGCOMM 88) suggests
442          * that doubling rto each time is the least we can get away with.
443          * In KA9Q, Karn uses this for the first few times, and then
444          * goes to quadratic.  netBSD doubles, but only goes up to *64,
445          * and clamps at 1 to 64 sec afterwards.  Note that 120 sec is
446          * defined in the protocol as the maximum possible RTT.  I guess
447          * we'll have to use something other than TCP to talk to the
448          * University of Mars.
449          *
450          * PAWS allows us longer timeouts and large windows, so once
451          * implemented ftp to mars will work nicely. We will have to fix
452          * the 120 second clamps though!
453          */
454         icsk->icsk_backoff++;
455         icsk->icsk_retransmits++;
456
457 out_reset_timer:
458         /* If stream is thin, use linear timeouts. Since 'icsk_backoff' is
459          * used to reset timer, set to 0. Recalculate 'icsk_rto' as this
460          * might be increased if the stream oscillates between thin and thick,
461          * thus the old value might already be too high compared to the value
462          * set by 'tcp_set_rto' in tcp_input.c which resets the rto without
463          * backoff. Limit to TCP_THIN_LINEAR_RETRIES before initiating
464          * exponential backoff behaviour to avoid continue hammering
465          * linear-timeout retransmissions into a black hole
466          */
467         if (sk->sk_state == TCP_ESTABLISHED &&
468             (tp->thin_lto || sysctl_tcp_thin_linear_timeouts) &&
469             tcp_stream_is_thin(tp) &&
470             icsk->icsk_retransmits <= TCP_THIN_LINEAR_RETRIES) {
471                 icsk->icsk_backoff = 0;
472                 icsk->icsk_rto = min(__tcp_set_rto(tp), TCP_RTO_MAX);
473         } else {
474                 /* Use normal (exponential) backoff */
475                 icsk->icsk_rto = min(icsk->icsk_rto << 1, TCP_RTO_MAX);
476         }
477         inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS, icsk->icsk_rto, TCP_RTO_MAX);
478         if (retransmits_timed_out(sk, sysctl_tcp_retries1 + 1, 0, 0))
479                 __sk_dst_reset(sk);
480
481 out:;
482 }
483
484 void tcp_write_timer_handler(struct sock *sk)
485 {
486         struct inet_connection_sock *icsk = inet_csk(sk);
487         int event;
488
489         if (sk->sk_state == TCP_CLOSE || !icsk->icsk_pending)
490                 goto out;
491
492         if (time_after(icsk->icsk_timeout, jiffies)) {
493                 sk_reset_timer(sk, &icsk->icsk_retransmit_timer, icsk->icsk_timeout);
494                 goto out;
495         }
496
497         event = icsk->icsk_pending;
498
499         switch (event) {
500         case ICSK_TIME_EARLY_RETRANS:
501                 tcp_resume_early_retransmit(sk);
502                 break;
503         case ICSK_TIME_LOSS_PROBE:
504                 tcp_send_loss_probe(sk);
505                 break;
506         case ICSK_TIME_RETRANS:
507                 icsk->icsk_pending = 0;
```

```
508                 tcp_retransmit_timer(sk);
509                 break;
510         case ICSK_TIME_PROBE0:
511                 icsk->icsk_pending = 0;
512                 tcp_probe_timer(sk);
513                 break;
514         }
515
516 out:
517         sk_mem_reclaim(sk);
518 }
519
520 static void tcp_write_timer(unsigned long data)
521 {
522         struct sock *sk = (struct sock *)data;
523
524         bh_lock_sock(sk);
525         if (!sock_owned_by_user(sk)) {
526                 tcp_write_timer_handler(sk);
527         } else {
528                 /* deleguate our work to tcp_release_cb() */
529                 if (!test_and_set_bit(TCP_WRITE_TIMER_DEFERRED, &tcp_sk(sk)->tsq_flags))
530                         sock_hold(sk);
531         }
532         bh_unlock_sock(sk);
533         sock_put(sk);
534 }
535
536 /*
537  *      Timer for listening sockets
538  */
539
540 static void tcp_synack_timer(struct sock *sk)
541 {
542         inet_csk_reqsk_queue_prune(sk, TCP_SYNQ_INTERVAL,
543                                    TCP_TIMEOUT_INIT, TCP_RTO_MAX);
544 }
545
546 void tcp_syn_ack_timeout(struct sock *sk, struct request_sock *req)
547 {
548         NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPTIMEOUTS);
549 }
550 EXPORT_SYMBOL(tcp_syn_ack_timeout);
551
552 void tcp_set_keepalive(struct sock *sk, int val)
553 {
554         if ((1 << sk->sk_state) & (TCPF_CLOSE | TCPF_LISTEN))
555                 return;
556
557         if (val && !sock_flag(sk, SOCK_KEEPOPEN))
558                 inet_csk_reset_keepalive_timer(sk, keepalive_time_when(tcp_sk(sk)));
559         else if (!val)
560                 inet_csk_delete_keepalive_timer(sk);
561 }
562
563
564 static void tcp_keepalive_timer (unsigned long data)
565 {
566         struct sock *sk = (struct sock *) data;
567         struct inet_connection_sock *icsk = inet_csk(sk);
568         struct tcp_sock *tp = tcp_sk(sk);
569         u32 elapsed;
570
571         /* Only process if socket is not in use. */
572         bh_lock_sock(sk);
573         if (sock_owned_by_user(sk)) {
574                 /* Try again later. */
575                 inet_csk_reset_keepalive_timer (sk, HZ/20);
576                 goto out;
577         }
578
579         if (sk->sk_state == TCP_LISTEN) {
580                 tcp_synack_timer(sk);
581                 goto out;
582         }
583
584         if (sk->sk_state == TCP_FIN_WAIT2 && sock_flag(sk, SOCK_DEAD)) {
585                 if (tp->linger2 >= 0) {
586                         const int tmo = tcp_fin_time(sk) - TCP_TIMEWAIT_LEN;
587
588                         if (tmo > 0) {
589                                 tcp_time_wait(sk, TCP_FIN_WAIT2, tmo);
590                                 goto out;
591                         }
592                 }
593                 tcp_send_active_reset(sk, GFP_ATOMIC);
594                 goto death;
595         }
```

```
596
597         if (!sock_flag(sk, SOCK_KEEPOPEN) || sk->sk_state == TCP_CLOSE)
598                 goto out;
599
600         elapsed = keepalive_time_when(tp);
601
602         /* It is alive without keepalive 8) */
603         if (tp->packets_out || tcp_send_head(sk))
604                 goto resched;
605
606         elapsed = keepalive_time_elapsed(tp);
607
608         if (elapsed >= keepalive_time_when(tp)) {
609                 /* If the TCP_USER_TIMEOUT option is enabled, use that
610                  * to determine when to timeout instead.
611                  */
612                 if ((icsk->icsk_user_timeout != 0 &&
613                     elapsed >= icsk->icsk_user_timeout &&
614                     icsk->icsk_probes_out > 0) ||
615                     (icsk->icsk_user_timeout == 0 &&
616                     icsk->icsk_probes_out >= keepalive_probes(tp))) {
617                         tcp_send_active_reset(sk, GFP_ATOMIC);
618                         tcp_write_err(sk);
619                         goto out;
620                 }
621                 if (tcp_write_wakeup(sk) <= 0) {
622                         icsk->icsk_probes_out++;
623                         elapsed = keepalive_intvl_when(tp);
624                 } else {
625                         /* If keepalive was lost due to local congestion,
626                          * try harder.
627                          */
628                         elapsed = TCP_RESOURCE_PROBE_INTERVAL;
629                 }
630         } else {
631                 /* It is tp->rcv_tstamp + keepalive_time_when(tp) */
632                 elapsed = keepalive_time_when(tp) - elapsed;
633         }
634
635         sk_mem_reclaim(sk);
636
637 resched:
638         inet_csk_reset_keepalive_timer (sk, elapsed);
639         goto out;
640
641 death:
642         tcp_done(sk);
643
644 out:
645         bh_unlock_sock(sk);
646         sock_put(sk);
647 }
648
649 void tcp_init_xmit_timers(struct sock *sk)
650 {
651         inet_csk_init_xmit_timers(sk, &tcp_write_timer, &tcp_delack_timer,
652                                   &tcp_keepalive_timer);
653 }
654 EXPORT_SYMBOL(tcp_init_xmit_timers);
655
```