Linux Cross Reference

Free Electrons

Embedded Linux Experts

Linux/net/ipv4/tcp_input.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 /*
22  * Changes:
23  *              Pedro Roque     :       Fast Retransmit/Recovery.
24  *                                      Two receive queues.
25  *                                      Retransmit queue handled by TCP.
26  *                                      Better retransmit timer handling.
27  *                                      New congestion avoidance.
28  *                                      Header prediction.
29  *                                      Variable renaming.
30  *
31  *              Eric            :       Fast Retransmit.
32  *              Randy Scott     :       MSS option defines.
33  *              Eric Schenk     :       Fixes to slow start algorithm.
34  *              Eric Schenk     :       Yet another double ACK bug.
35  *              Eric Schenk     :       Delayed ACK bug fixes.
36  *              Eric Schenk     :       Floyd style fast retrans war avoidance.
37  *              David S. Miller :       Don't allow zero congestion window.
38  *              Eric Schenk     :       Fix retransmitter so that it sends
39  *                                      next packet on ack of previous packet.
40  *              Andi Kleen      :       Moved open_request checking here
41  *                                      and process RSTs for open_requests.
42  *              Andi Kleen      :       Better prune_queue, and other fixes.
43  *              Andrey Savochkin:       Fix RTT measurements in the presence of
44  *                                      timestamps.
45  *              Andrey Savochkin:       Check sequence numbers correctly when
46  *                                      removing SACKs due to in sequence incoming
47  *                                      data segments.
48  *              Andi Kleen:             Make sure we never ack data there is not
49  *                                      enough room for. Also make this condition
50  *                                      a fatal error if it might still happen.
51  *              Andi Kleen:             Add tcp_measure_rcv_mss to make
52  *                                      connections with MSS<min(MTU,ann. MSS)
53  *                                      work without delayed acks.
54  *              Andi Kleen:             Process packets with PSH set in the
55  *                                      fast path.
56  *              J Hadi Salim:           ECN support
57  *              Andrei Gurtov,
58  *              Pasi Sarolahti,
59  *              Panu Kuhlberg:          Experimental audit of TCP (re)transmission
60  *                                      engine. Lots of bugs are found.
61  *              Pasi Sarolahti:         F-RTO for dealing with spurious RTOs
62  */
63
64 #define pr_fmt(fmt) "TCP: " fmt
65
66 #include <linux/mm.h>
67 #include <linux/slab.h>
68 #include <linux/module.h>
```

```
 69 #include <linux/sysctl.h>
 70 #include <linux/kernel.h>
 71 #include <net/dst.h>
 72 #include <net/tcp.h>
 73 #include <net/inet_common.h>
 74 #include <linux/ipsec.h>
 75 #include <asm/unaligned.h>
 76 #include <net/netdma.h>
 77 #include <linux/errqueue.h>
 78
 79 int sysctl_tcp_timestamps __read_mostly = 1;
 80 int sysctl_tcp_window_scaling __read_mostly = 1;
 81 int sysctl_tcp_sack __read_mostly = 1;
 82 int sysctl_tcp_fack __read_mostly = 1;
 83 int sysctl_tcp_reordering __read_mostly = TCP_FASTRETRANS_THRESH;
 84 EXPORT_SYMBOL(sysctl_tcp_reordering);
 85 int sysctl_tcp_dsack __read_mostly = 1;
 86 int sysctl_tcp_app_win __read_mostly = 31;
 87 int sysctl_tcp_adv_win_scale __read_mostly = 1;
 88 EXPORT_SYMBOL(sysctl_tcp_adv_win_scale);
 89
 90 /* rfc5961 challenge ack rate limiting */
 91 int sysctl_tcp_challenge_ack_limit = 100;
 92
 93 int sysctl_tcp_stdurg __read_mostly;
 94 int sysctl_tcp_rfc1337 __read_mostly;
 95 int sysctl_tcp_max_orphans __read_mostly = NR_FILE;
 96 int sysctl_tcp_frto __read_mostly = 2;
 97
 98 int sysctl_tcp_thin_dupack __read_mostly;
 99
100 int sysctl_tcp_moderate_rcvbuf __read_mostly = 1;
101 int sysctl_tcp_early_retrans __read_mostly = 3;
102
103 #define FLAG_DATA               0x01 /* Incoming frame contained data.          */
104 #define FLAG_WIN_UPDATE         0x02 /* Incoming ACK was a window update.       */
105 #define FLAG_DATA_ACKED         0x04 /* This ACK acknowledged new data.         */
106 #define FLAG_RETRANS_DATA_ACKED 0x08 /* "" "" some of which was retransmitted.  */
107 #define FLAG_SYN_ACKED          0x10 /* This ACK acknowledged SYN.              */
108 #define FLAG_DATA_SACKED        0x20 /* New SACK.                               */
109 #define FLAG_ECE                0x40 /* ECE in this ACK                         */
110 #define FLAG_SLOWPATH           0x100 /* Do not skip RFC checks for window update.*/
111 #define FLAG_ORIG_SACK_ACKED    0x200 /* Never retransmitted data are (s)acked  */
112 #define FLAG_SND_UNA_ADVANCED   0x400 /* Snd_una was changed (!= FLAG_DATA_ACKED) */
113 #define FLAG_DSACKING_ACK       0x800 /* SACK blocks contained D-SACK info */
114 #define FLAG_SACK_RENEGING      0x2000 /* snd_una advanced to a sacked seq */
115 #define FLAG_UPDATE_TS_RECENT   0x4000 /* tcp_replace_ts_recent() */
116
117 #define FLAG_ACKED              (FLAG_DATA_ACKED|FLAG_SYN_ACKED)
118 #define FLAG_NOT_DUP            (FLAG_DATA|FLAG_WIN_UPDATE|FLAG_ACKED)
119 #define FLAG_CA_ALERT           (FLAG_DATA_SACKED|FLAG_ECE)
120 #define FLAG_FORWARD_PROGRESS   (FLAG_ACKED|FLAG_DATA_SACKED)
121
122 #define TCP_REMNANT (TCP_FLAG_FIN|TCP_FLAG_URG|TCP_FLAG_SYN|TCP_FLAG_PSH)
123 #define TCP_HP_BITS (~(TCP_RESERVED_BITS|TCP_FLAG_PSH))
124
125 /* Adapt the MSS value used to make delayed ack decision to the
126  * real world.
127  */
128 static void tcp_measure_rcv_mss(struct sock *sk, const struct sk_buff *skb)
129 {
130         struct inet_connection_sock *icsk = inet_csk(sk);
131         const unsigned int lss = icsk->icsk_ack.last_seg_size;
132         unsigned int len;
133
134         icsk->icsk_ack.last_seg_size = 0;
135
136         /* skb->len may jitter because of SACKs, even if peer
137          * sends good full-sized frames.
138          */
139         len = skb_shinfo(skb)->gso_size ? : skb->len;
140         if (len >= icsk->icsk_ack.rcv_mss) {
141                 icsk->icsk_ack.rcv_mss = len;
142         } else {
143                 /* Otherwise, we make more careful check taking into account,
144                  * that SACKs block is variable.
145                  *
146                  * "len" is invariant segment length, including TCP header.
147                  */
148                 len += skb->data - skb_transport_header(skb);
149                 if (len >= TCP_MSS_DEFAULT + sizeof(struct tcphdr) ||
150                     /* If PSH is not set, packet should be
151                      * full sized, provided peer TCP is not badly broken.
152                      * This observation (if it is correct 8)) allows
153                      * to handle super-low mtu links fairly.
154                      */
155                     (len >= TCP_MIN_MSS + sizeof(struct tcphdr) &&
156                      !(tcp_flag_word(tcp_hdr(skb)) & TCP_REMNANT))) {
```

```
157                         /* Subtract also invariant (if peer is RFC compliant),
158                          * tcp header plus fixed timestamp option length.
159                          * Resulting "len" is MSS free of SACK jitter.
160                          */
161                         len -= tcp_sk(sk)->tcp_header_len;
162                         icsk->icsk_ack.last_seg_size = len;
163                         if (len == lss) {
164                                 icsk->icsk_ack.rcv_mss = len;
165                                 return;
166                         }
167                 }
168                 if (icsk->icsk_ack.pending & ICSK_ACK_PUSHED)
169                         icsk->icsk_ack.pending |= ICSK_ACK_PUSHED2;
170                 icsk->icsk_ack.pending |= ICSK_ACK_PUSHED;
171         }
172 }
173
174 static void tcp_incr_quickack(struct sock *sk)
175 {
176         struct inet_connection_sock *icsk = inet_csk(sk);
177         unsigned int quickacks = tcp_sk(sk)->rcv_wnd / (2 * icsk->icsk_ack.rcv_mss);
178
179         if (quickacks == 0)
180                 quickacks = 2;
181         if (quickacks > icsk->icsk_ack.quick)
182                 icsk->icsk_ack.quick = min(quickacks, TCP_MAX_QUICKACKS);
183 }
184
185 static void tcp_enter_quickack_mode(struct sock *sk)
186 {
187         struct inet_connection_sock *icsk = inet_csk(sk);
188         tcp_incr_quickack(sk);
189         icsk->icsk_ack.pingpong = 0;
190         icsk->icsk_ack.ato = TCP_ATO_MIN;
191 }
192
193 /* Send ACKs quickly, if "quick" count is not exhausted
194  * and the session is not interactive.
195  */
196
197 static inline bool tcp_in_quickack_mode(const struct sock *sk)
198 {
199         const struct inet_connection_sock *icsk = inet_csk(sk);
200
201         return icsk->icsk_ack.quick && !icsk->icsk_ack.pingpong;
202 }
203
204 static inline void TCP_ECN_queue_cwr(struct tcp_sock *tp)
205 {
206         if (tp->ecn_flags & TCP_ECN_OK)
207                 tp->ecn_flags |= TCP_ECN_QUEUE_CWR;
208 }
209
210 static inline void TCP_ECN_accept_cwr(struct tcp_sock *tp, const struct sk_buff *skb)
211 {
212         if (tcp_hdr(skb)->cwr)
213                 tp->ecn_flags &= ~TCP_ECN_DEMAND_CWR;
214 }
215
216 static inline void TCP_ECN_withdraw_cwr(struct tcp_sock *tp)
217 {
218         tp->ecn_flags &= ~TCP_ECN_DEMAND_CWR;
219 }
220
221 static inline void TCP_ECN_check_ce(struct tcp_sock *tp, const struct sk_buff *skb)
222 {
223         if (!(tp->ecn_flags & TCP_ECN_OK))
224                 return;
225
226         switch (TCP_SKB_CB(skb)->ip_dsfield & INET_ECN_MASK) {
227         case INET_ECN_NOT_ECT:
228                 /* Funny extension: if ECT is not set on a segment,
229                  * and we already seen ECT on a previous segment,
230                  * it is probably a retransmit.
231                  */
232                 if (tp->ecn_flags & TCP_ECN_SEEN)
233                         tcp_enter_quickack_mode((struct sock *)tp);
234                 break;
235         case INET_ECN_CE:
236                 if (!(tp->ecn_flags & TCP_ECN_DEMAND_CWR)) {
237                         /* Better not delay acks, sender can have a very low cwnd */
238                         tcp_enter_quickack_mode((struct sock *)tp);
239                         tp->ecn_flags |= TCP_ECN_DEMAND_CWR;
240                 }
241                 /* fallinto */
242         default:
243                 tp->ecn_flags |= TCP_ECN_SEEN;
244         }
```

```
245 }
246
247 static inline void TCP_ECN_rcv_synack(struct tcp_sock *tp, const struct tcphdr *th)
248 {
249         if ((tp->ecn_flags & TCP_ECN_OK) && (!th->ece || th->cwr))
250                 tp->ecn_flags &= ~TCP_ECN_OK;
251 }
252
253 static inline void TCP_ECN_rcv_syn(struct tcp_sock *tp, const struct tcphdr *th)
254 {
255         if ((tp->ecn_flags & TCP_ECN_OK) && (!th->ece || !th->cwr))
256                 tp->ecn_flags &= ~TCP_ECN_OK;
257 }
258
259 static bool TCP_ECN_rcv_ecn_echo(const struct tcp_sock *tp, const struct tcphdr *th)
260 {
261         if (th->ece && !th->syn && (tp->ecn_flags & TCP_ECN_OK))
262                 return true;
263         return false;
264 }
265
266 /* Buffer size and advertised window tuning.
267  *
268  * 1. Tuning sk->sk_sndbuf, when connection enters established state.
269  */
270
271 static void tcp_sndbuf_expand(struct sock *sk)
272 {
273         const struct tcp_sock *tp = tcp_sk(sk);
274         int sndmem, per_mss;
275         u32 nr_segs;
276
277         /* Worst case is non GSO/TSO : each frame consumes one skb
278          * and skb->head is kmalloced using power of two area of memory
279          */
280         per_mss = max_t(u32, tp->rx_opt.mss_clamp, tp->mss_cache) +
281                   MAX_TCP_HEADER +
282                   SKB_DATA_ALIGN(sizeof(struct skb_shared_info));
283
284         per_mss = roundup_pow_of_two(per_mss) +
285                   SKB_DATA_ALIGN(sizeof(struct sk_buff));
286
287         nr_segs = max_t(u32, TCP_INIT_CWND, tp->snd_cwnd);
288         nr_segs = max_t(u32, nr_segs, tp->reordering + 1);
289
290         /* Fast Recovery (RFC 5681 3.2) :
291          * Cubic needs 1.7 factor, rounded to 2 to include
292          * extra cushion (application might react slowly to POLLOUT)
293          */
294         sndmem = 2 * nr_segs * per_mss;
295
296         if (sk->sk_sndbuf < sndmem)
297                 sk->sk_sndbuf = min(sndmem, sysctl_tcp_wmem[2]);
298 }
299
300 /* 2. Tuning advertised window (window_clamp, rcv_ssthresh)
301  *
302  * All tcp_full_space() is split to two parts: "network" buffer, allocated
303  * forward and advertised in receiver window (tp->rcv_wnd) and
304  * "application buffer", required to isolate scheduling/application
305  * latencies from network.
306  * window_clamp is maximal advertised window. It can be less than
307  * tcp_full_space(), in this case tcp_full_space() - window_clamp
308  * is reserved for "application" buffer. The less window_clamp is
309  * the smoother our behaviour from viewpoint of network, but the lower
310  * throughput and the higher sensitivity of the connection to losses. 8)
311  *
312  * rcv_ssthresh is more strict window_clamp used at "slow start"
313  * phase to predict further behaviour of this connection.
314  * It is used for two goals:
315  * - to enforce header prediction at sender, even when application
316  *   requires some significant "application buffer". It is check #1.
317  * - to prevent pruning of receive queue because of misprediction
318  *   of receiver window. Check #2.
319  *
320  * The scheme does not work when sender sends good segments opening
321  * window and then starts to feed us spaghetti. But it should work
322  * in common situations. Otherwise, we have to rely on queue collapsing.
323  */
324
325 /* Slow part of check#2. */
326 static int __tcp_grow_window(const struct sock *sk, const struct sk_buff *skb)
327 {
328         struct tcp_sock *tp = tcp_sk(sk);
329         /* Optimize this! */
330         int truesize = tcp_win_from_space(skb->truesize) >> 1;
331         int window = tcp_win_from_space(sysctl_tcp_rmem[2]) >> 1;
332
```

```
333         while (tp->rcv_ssthresh <= window) {
334                 if (truesize <= skb->len)
335                         return 2 * inet_csk(sk)->icsk_ack.rcv_mss;
336
337                 truesize >>= 1;
338                 window >>= 1;
339         }
340         return 0;
341 }
342
343 static void tcp_grow_window(struct sock *sk, const struct sk_buff *skb)
344 {
345         struct tcp_sock *tp = tcp_sk(sk);
346
347         /* Check #1 */
348         if (tp->rcv_ssthresh < tp->window_clamp &&
349             (int)tp->rcv_ssthresh < tcp_space(sk) &&
350             !sk_under_memory_pressure(sk)) {
351                 int incr;
352
353                 /* Check #2. Increase window, if skb with such overhead
354                  * will fit to rcvbuf in future.
355                  */
356                 if (tcp_win_from_space(skb->truesize) <= skb->len)
357                         incr = 2 * tp->advmss;
358                 else
359                         incr = __tcp_grow_window(sk, skb);
360
361                 if (incr) {
362                         incr = max_t(int, incr, 2 * skb->len);
363                         tp->rcv_ssthresh = min(tp->rcv_ssthresh + incr,
364                                                tp->window_clamp);
365                         inet_csk(sk)->icsk_ack.quick |= 1;
366                 }
367         }
368 }
369
370 /* 3. Tuning rcvbuf, when connection enters established state. */
371 static void tcp_fixup_rcvbuf(struct sock *sk)
372 {
373         u32 mss = tcp_sk(sk)->advmss;
374         int rcvmem;
375
376         rcvmem = 2 * SKB_TRUESIZE(mss + MAX_TCP_HEADER) *
377                  tcp_default_init_rwnd(mss);
378
379         /* Dynamic Right Sizing (DRS) has 2 to 3 RTT latency
380          * Allow enough cushion so that sender is not limited by our window
381          */
382         if (sysctl_tcp_moderate_rcvbuf)
383                 rcvmem <<= 2;
384
385         if (sk->sk_rcvbuf < rcvmem)
386                 sk->sk_rcvbuf = min(rcvmem, sysctl_tcp_rmem[2]);
387 }
388
389 /* 4. Try to fixup all. It is made immediately after connection enters
390  *    established state.
391  */
392 void tcp_init_buffer_space(struct sock *sk)
393 {
394         struct tcp_sock *tp = tcp_sk(sk);
395         int maxwin;
396
397         if (!(sk->sk_userlocks & SOCK_RCVBUF_LOCK))
398                 tcp_fixup_rcvbuf(sk);
399         if (!(sk->sk_userlocks & SOCK_SNDBUF_LOCK))
400                 tcp_sndbuf_expand(sk);
401
402         tp->rcvq_space.space = tp->rcv_wnd;
403         tp->rcvq_space.time = tcp_time_stamp;
404         tp->rcvq_space.seq = tp->copied_seq;
405
406         maxwin = tcp_full_space(sk);
407
408         if (tp->window_clamp >= maxwin) {
409                 tp->window_clamp = maxwin;
410
411                 if (sysctl_tcp_app_win && maxwin > 4 * tp->advmss)
412                         tp->window_clamp = max(maxwin -
413                                                (maxwin >> sysctl_tcp_app_win),
414                                                4 * tp->advmss);
415         }
416
417         /* Force reservation of one segment. */
418         if (sysctl_tcp_app_win &&
419             tp->window_clamp > 2 * tp->advmss &&
420             tp->window_clamp + tp->advmss > maxwin)
```

```
421                 tp->window_clamp = max(2 * tp->advmss, maxwin - tp->advmss);
422
423         tp->rcv_ssthresh = min(tp->rcv_ssthresh, tp->window_clamp);
424         tp->snd_cwnd_stamp = tcp_time_stamp;
425 }
426
427 /* 5. Recalculate window clamp after socket hit its memory bounds. */
428 static void tcp_clamp_window(struct sock *sk)
429 {
430         struct tcp_sock *tp = tcp_sk(sk);
431         struct inet_connection_sock *icsk = inet_csk(sk);
432
433         icsk->icsk_ack.quick = 0;
434
435         if (sk->sk_rcvbuf < sysctl_tcp_rmem[2] &&
436             !(sk->sk_userlocks & SOCK_RCVBUF_LOCK) &&
437             !sk_under_memory_pressure(sk) &&
438             sk_memory_allocated(sk) < sk_prot_mem_limits(sk, 0)) {
439                 sk->sk_rcvbuf = min(atomic_read(&sk->sk_rmem_alloc),
440                                     sysctl_tcp_rmem[2]);
441         }
442         if (atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf)
443                 tp->rcv_ssthresh = min(tp->window_clamp, 2U * tp->advmss);
444 }
445
446 /* Initialize RCV_MSS value.
447  * RCV_MSS is an our guess about MSS used by the peer.
448  * We haven't any direct information about the MSS.
449  * It's better to underestimate the RCV_MSS rather than overestimate.
450  * Overestimations make us ACKing less frequently than needed.
451  * Underestimations are more easy to detect and fix by tcp_measure_rcv_mss().
452  */
453 void tcp_initialize_rcv_mss(struct sock *sk)
454 {
455         const struct tcp_sock *tp = tcp_sk(sk);
456         unsigned int hint = min_t(unsigned int, tp->advmss, tp->mss_cache);
457
458         hint = min(hint, tp->rcv_wnd / 2);
459         hint = min(hint, TCP_MSS_DEFAULT);
460         hint = max(hint, TCP_MIN_MSS);
461
462         inet_csk(sk)->icsk_ack.rcv_mss = hint;
463 }
464 EXPORT_SYMBOL(tcp_initialize_rcv_mss);
465
466 /* Receiver "autotuning" code.
467  *
468  * The algorithm for RTT estimation w/o timestamps is based on
469  * Dynamic Right-Sizing (DRS) by Wu Feng and Mike Fisk of LANL.
470  * <http://public.lanl.gov/radiant/pubs.html#DRS>
471  *
472  * More detail on this code can be found at
473  * <http://staff.psc.edu/jheffner/>,
474  * though this reference is out of date.  A new paper
475  * is pending.
476  */
477 static void tcp_rcv_rtt_update(struct tcp_sock *tp, u32 sample, int win_dep)
478 {
479         u32 new_sample = tp->rcv_rtt_est.rtt;
480         long m = sample;
481
482         if (m == 0)
483                 m = 1;
484
485         if (new_sample != 0) {
486                 /* If we sample in larger samples in the non-timestamp
487                  * case, we could grossly overestimate the RTT especially
488                  * with chatty applications or bulk transfer apps which
489                  * are stalled on filesystem I/O.
490                  *
491                  * Also, since we are only going for a minimum in the
492                  * non-timestamp case, we do not smooth things out
493                  * else with timestamps disabled convergence takes too
494                  * long.
495                  */
496                 if (!win_dep) {
497                         m -= (new_sample >> 3);
498                         new_sample += m;
499                 } else {
500                         m <<= 3;
501                         if (m < new_sample)
502                                 new_sample = m;
503                 }
504         } else {
505                 /* No previous measure. */
506                 new_sample = m << 3;
507         }
508
```

```
509         if (tp->rcv_rtt_est.rtt != new_sample)
510                 tp->rcv_rtt_est.rtt = new_sample;
511 }
512
513 static inline void tcp_rcv_rtt_measure(struct tcp_sock *tp)
514 {
515         if (tp->rcv_rtt_est.time == 0)
516                 goto new_measure;
517         if (before(tp->rcv_nxt, tp->rcv_rtt_est.seq))
518                 return;
519         tcp_rcv_rtt_update(tp, tcp_time_stamp - tp->rcv_rtt_est.time, 1);
520
521 new_measure:
522         tp->rcv_rtt_est.seq = tp->rcv_nxt + tp->rcv_wnd;
523         tp->rcv_rtt_est.time = tcp_time_stamp;
524 }
525
526 static inline void tcp_rcv_rtt_measure_ts(struct sock *sk,
527                                           const struct sk_buff *skb)
528 {
529         struct tcp_sock *tp = tcp_sk(sk);
530         if (tp->rx_opt.rcv_tsecr &&
531             (TCP_SKB_CB(skb)->end_seq -
532              TCP_SKB_CB(skb)->seq >= inet_csk(sk)->icsk_ack.rcv_mss))
533                 tcp_rcv_rtt_update(tp, tcp_time_stamp - tp->rx_opt.rcv_tsecr, 0);
534 }
535
536 /*
537  * This function should be called every time data is copied to user space.
538  * It calculates the appropriate TCP receive buffer space.
539  */
540 void tcp_rcv_space_adjust(struct sock *sk)
541 {
542         struct tcp_sock *tp = tcp_sk(sk);
543         int time;
544         int copied;
545
546         time = tcp_time_stamp - tp->rcvq_space.time;
547         if (time < (tp->rcv_rtt_est.rtt >> 3) || tp->rcv_rtt_est.rtt == 0)
548                 return;
549
550         /* Number of bytes copied to user in last RTT */
551         copied = tp->copied_seq - tp->rcvq_space.seq;
552         if (copied <= tp->rcvq_space.space)
553                 goto new_measure;
554
555         /* A bit of theory :
556          * copied = bytes received in previous RTT, our base window
557          * To cope with packet losses, we need a 2x factor
558          * To cope with slow start, and sender growing its cwin by 100 %
559          * every RTT, we need a 4x factor, because the ACK we are sending
560          * now is for the next RTT, not the current one :
561          * <prev RTT . ><current RTT .. ><next RTT .... >
562          */
563
564         if (sysctl_tcp_moderate_rcvbuf &&
565             !(sk->sk_userlocks & SOCK_RCVBUF_LOCK)) {
566                 int rcvwin, rcvmem, rcvbuf;
567
568                 /* minimal window to cope with packet losses, assuming
569                  * steady state. Add some cushion because of small variations.
570                  */
571                 rcvwin = (copied << 1) + 16 * tp->advmss;
572
573                 /* If rate increased by 25%,
574                  *      assume slow start, rcvwin = 3 * copied
575                  * If rate increased by 50%,
576                  *      assume sender can use 2x growth, rcvwin = 4 * copied
577                  */
578                 if (copied >=
579                     tp->rcvq_space.space + (tp->rcvq_space.space >> 2)) {
580                         if (copied >=
581                             tp->rcvq_space.space + (tp->rcvq_space.space >> 1))
582                                 rcvwin <<= 1;
583                         else
584                                 rcvwin += (rcvwin >> 1);
585                 }
586
587                 rcvmem = SKB_TRUESIZE(tp->advmss + MAX_TCP_HEADER);
588                 while (tcp_win_from_space(rcvmem) < tp->advmss)
589                         rcvmem += 128;
590
591                 rcvbuf = min(rcvwin / tp->advmss * rcvmem, sysctl_tcp_rmem[2]);
592                 if (rcvbuf > sk->sk_rcvbuf) {
593                         sk->sk_rcvbuf = rcvbuf;
594
595                         /* Make the window clamp follow along.  */
596                         tp->window_clamp = rcvwin;
```

```
597                 }
598         }
599         tp->rcvq_space.space = copied;
600
601 new_measure:
602         tp->rcvq_space.seq = tp->copied_seq;
603         tp->rcvq_space.time = tcp_time_stamp;
604 }
605
606 /* There is something which you must keep in mind when you analyze the
607  * behavior of the tp->ato delayed ack timeout interval.  When a
608  * connection starts up, we want to ack as quickly as possible.  The
609  * problem is that "good" TCP's do slow start at the beginning of data
610  * transmission.  The means that until we send the first few ACK's the
611  * sender will sit on his end and only queue most of his data, because
612  * he can only send snd_cwnd unacked packets at any given time.  For
613  * each ACK we send, he increments snd_cwnd and transmits more of his
614  * queue.  -DaveM
615  */
616 static void tcp_event_data_recv(struct sock *sk, struct sk_buff *skb)
617 {
618         struct tcp_sock *tp = tcp_sk(sk);
619         struct inet_connection_sock *icsk = inet_csk(sk);
620         u32 now;
621
622         inet_csk_schedule_ack(sk);
623
624         tcp_measure_rcv_mss(sk, skb);
625
626         tcp_rcv_rtt_measure(tp);
627
628         now = tcp_time_stamp;
629
630         if (!icsk->icsk_ack.ato) {
631                 /* The _first_ data packet received, initialize
632                  * delayed ACK engine.
633                  */
634                 tcp_incr_quickack(sk);
635                 icsk->icsk_ack.ato = TCP_ATO_MIN;
636         } else {
637                 int m = now - icsk->icsk_ack.lrcvtime;
638
639                 if (m <= TCP_ATO_MIN / 2) {
640                         /* The fastest case is the first. */
641                         icsk->icsk_ack.ato = (icsk->icsk_ack.ato >> 1) + TCP_ATO_MIN / 2;
642                 } else if (m < icsk->icsk_ack.ato) {
643                         icsk->icsk_ack.ato = (icsk->icsk_ack.ato >> 1) + m;
644                         if (icsk->icsk_ack.ato > icsk->icsk_rto)
645                                 icsk->icsk_ack.ato = icsk->icsk_rto;
646                 } else if (m > icsk->icsk_rto) {
647                         /* Too long gap. Apparently sender failed to
648                          * restart window, so that we send ACKs quickly.
649                          */
650                         tcp_incr_quickack(sk);
651                         sk_mem_reclaim(sk);
652                 }
653         }
654         icsk->icsk_ack.lrcvtime = now;
655
656         TCP_ECN_check_ce(tp, skb);
657
658         if (skb->len >= 128)
659                 tcp_grow_window(sk, skb);
660 }
661
662 /* Called to compute a smoothed rtt estimate. The data fed to this
663  * routine either comes from timestamps, or from segments that were
664  * known _not_ to have been retransmitted [see Karn/Partridge
665  * Proceedings SIGCOMM 87]. The algorithm is from the SIGCOMM 88
666  * piece by Van Jacobson.
667  * NOTE: the next three routines used to be one big routine.
668  * To save cycles in the RFC 1323 implementation it was better to break
669  * it up into three procedures. -- erics
670  */
671 static void tcp_rtt_estimator(struct sock *sk, long mrtt_us)
672 {
673         struct tcp_sock *tp = tcp_sk(sk);
674         long m = mrtt_us; /* RTT */
675         u32 srtt = tp->srtt_us;
676
677         /*      The following amusing code comes from Jacobson's
678          *      article in SIGCOMM '88.  Note that rtt and mdev
679          *      are scaled versions of rtt and mean deviation.
680          *      This is designed to be as fast as possible
681          *      m stands for "measurement".
682          *
683          *      On a 1990 paper the rto value is changed to:
684          *      RTO = rtt + 4 * mdev
```

```
685          *
686          * Funny. This algorithm seems to be very broken.
687          * These formulae increase RTO, when it should be decreased, increase
688          * too slowly, when it should be increased quickly, decrease too quickly
689          * etc. I guess in BSD RTO takes ONE value, so that it is absolutely
690          * does not matter how to _calculate_ it. Seems, it was trap
691          * that VJ failed to avoid. 8)
692          */
693         if (srtt != 0) {
694                 m -= (srtt >> 3);       /* m is now error in rtt est */
695                 srtt += m;              /* rtt = 7/8 rtt + 1/8 new */
696                 if (m < 0) {
697                         m = -m;         /* m is now abs(error) */
698                         m -= (tp->mdev_us >> 2);   /* similar update on mdev */
699                         /* This is similar to one of Eifel findings.
700                          * Eifel blocks mdev updates when rtt decreases.
701                          * This solution is a bit different: we use finer gain
702                          * for mdev in this case (alpha*beta).
703                          * Like Eifel it also prevents growth of rto,
704                          * but also it limits too fast rto decreases,
705                          * happening in pure Eifel.
706                          */
707                         if (m > 0)
708                                 m >>= 3;
709                 } else {
710                         m -= (tp->mdev_us >> 2);   /* similar update on mdev */
711                 }
712                 tp->mdev_us += m;               /* mdev = 3/4 mdev + 1/4 new */
713                 if (tp->mdev_us > tp->mdev_max_us) {
714                         tp->mdev_max_us = tp->mdev_us;
715                         if (tp->mdev_max_us > tp->rttvar_us)
716                                 tp->rttvar_us = tp->mdev_max_us;
717                 }
718                 if (after(tp->snd_una, tp->rtt_seq)) {
719                         if (tp->mdev_max_us < tp->rttvar_us)
720                                 tp->rttvar_us -= (tp->rttvar_us - tp->mdev_max_us) >> 2;
721                         tp->rtt_seq = tp->snd_nxt;
722                         tp->mdev_max_us = tcp_rto_min_us(sk);
723                 }
724         } else {
725                 /* no previous measure. */
726                 srtt = m << 3;          /* take the measured time to be rtt */
727                 tp->mdev_us = m << 1;   /* make sure rto = 3*rtt */
728                 tp->rttvar_us = max(tp->mdev_us, tcp_rto_min_us(sk));
729                 tp->mdev_max_us = tp->rttvar_us;
730                 tp->rtt_seq = tp->snd_nxt;
731         }
732         tp->srtt_us = max(1U, srtt);
733 }
734
735 /* Set the sk_pacing_rate to allow proper sizing of TSO packets.
736  * Note: TCP stack does not yet implement pacing.
737  * FQ packet scheduler can be used to implement cheap but effective
738  * TCP pacing, to smooth the burst on large writes when packets
739  * in flight is significantly lower than cwnd (or rwin)
740  */
741 static void tcp_update_pacing_rate(struct sock *sk)
742 {
743         const struct tcp_sock *tp = tcp_sk(sk);
744         u64 rate;
745
746         /* set sk_pacing_rate to 200 % of current rate (mss * cwnd / srtt) */
747         rate = (u64)tp->mss_cache * 2 * (USEC_PER_SEC << 3);
748
749         rate *= max(tp->snd_cwnd, tp->packets_out);
750
751         if (likely(tp->srtt_us))
752                 do_div(rate, tp->srtt_us);
753
754         /* ACCESS_ONCE() is needed because sch_fq fetches sk_pacing_rate
755          * without any lock. We want to make sure compiler wont store
756          * intermediate values in this location.
757          */
758         ACCESS_ONCE(sk->sk_pacing_rate) = min_t(u64, rate,
759                                                 sk->sk_max_pacing_rate);
760 }
761
762 /* Calculate rto without backoff.  This is the second half of Van Jacobson's
763  * routine referred to above.
764  */
765 static void tcp_set_rto(struct sock *sk)
766 {
767         const struct tcp_sock *tp = tcp_sk(sk);
768         /* Old crap is replaced with new one. 8)
769          *
770          * More seriously:
771          * 1. If rtt variance happened to be less 50msec, it is hallucination.
772          *    It cannot be less due to utterly erratic ACK generation made
```

```
773          *    at least by solaris and freebsd. "Erratic ACKs" has _nothing_
774          *    to do with delayed acks, because at cwnd>2 true delack timeout
775          *    is invisible. Actually, Linux-2.4 also generates erratic
776          *    ACKs in some circumstances.
777          */
778         inet_csk(sk)->icsk_rto = __tcp_set_rto(tp);
779
780         /* 2. Fixups made earlier cannot be right.
781          *    If we do not estimate RTO correctly without them,
782          *    all the algo is pure shit and should be replaced
783          *    with correct one. It is exactly, which we pretend to do.
784          */
785
786         /* NOTE: clamping at TCP_RTO_MIN is not required, current algo
787          * guarantees that rto is higher.
788          */
789         tcp_bound_rto(sk);
790 }
791
792 __u32 tcp_init_cwnd(const struct tcp_sock *tp, const struct dst_entry *dst)
793 {
794         __u32 cwnd = (dst ? dst_metric(dst, RTAX_INITCWND) : 0);
795
796         if (!cwnd)
797                 cwnd = TCP_INIT_CWND;
798         return min_t(__u32, cwnd, tp->snd_cwnd_clamp);
799 }
800
801 /*
802  * Packet counting of FACK is based on in-order assumptions, therefore TCP
803  * disables it when reordering is detected
804  */
805 void tcp_disable_fack(struct tcp_sock *tp)
806 {
807         /* RFC3517 uses different metric in lost marker => reset on change */
808         if (tcp_is_fack(tp))
809                 tp->lost_skb_hint = NULL;
810         tp->rx_opt.sack_ok &= ~TCP_FACK_ENABLED;
811 }
812
813 /* Take a notice that peer is sending D-SACKs */
814 static void tcp_dsack_seen(struct tcp_sock *tp)
815 {
816         tp->rx_opt.sack_ok |= TCP_DSACK_SEEN;
817 }
818
819 static void tcp_update_reordering(struct sock *sk, const int metric,
820                                   const int ts)
821 {
822         struct tcp_sock *tp = tcp_sk(sk);
823         if (metric > tp->reordering) {
824                 int mib_idx;
825
826                 tp->reordering = min(TCP_MAX_REORDERING, metric);
827
828                 /* This exciting event is worth to be remembered. 8) */
829                 if (ts)
830                         mib_idx = LINUX_MIB_TCPTSREORDER;
831                 else if (tcp_is_reno(tp))
832                         mib_idx = LINUX_MIB_TCPRENOREORDER;
833                 else if (tcp_is_fack(tp))
834                         mib_idx = LINUX_MIB_TCPFACKREORDER;
835                 else
836                         mib_idx = LINUX_MIB_TCPSACKREORDER;
837
838                 NET_INC_STATS_BH(sock_net(sk), mib_idx);
839 #if FASTRETRANS_DEBUG > 1
840                 pr_debug("Disorder%d %d %u f%u s%u rr%d\n",
841                          tp->rx_opt.sack_ok, inet_csk(sk)->icsk_ca_state,
842                          tp->reordering,
843                          tp->fackets_out,
844                          tp->sacked_out,
845                          tp->undo_marker ? tp->undo_retrans : 0);
846 #endif
847                 tcp_disable_fack(tp);
848         }
849
850         if (metric > 0)
851                 tcp_disable_early_retrans(tp);
852 }
853
854 /* This must be called before lost_out is incremented */
855 static void tcp_verify_retransmit_hint(struct tcp_sock *tp, struct sk_buff *skb)
856 {
857         if ((tp->retransmit_skb_hint == NULL) ||
858             before(TCP_SKB_CB(skb)->seq,
859                    TCP_SKB_CB(tp->retransmit_skb_hint)->seq))
860                 tp->retransmit_skb_hint = skb;
```

```
861
862         if (!tp->lost_out ||
863             after(TCP_SKB_CB(skb)->end_seq, tp->retransmit_high))
864                 tp->retransmit_high = TCP_SKB_CB(skb)->end_seq;
865 }
866
867 static void tcp_skb_mark_lost(struct tcp_sock *tp, struct sk_buff *skb)
868 {
869         if (!(TCP_SKB_CB(skb)->sacked & (TCPCB_LOST|TCPCB_SACKED_ACKED))) {
870                 tcp_verify_retransmit_hint(tp, skb);
871
872                 tp->lost_out += tcp_skb_pcount(skb);
873                 TCP_SKB_CB(skb)->sacked |= TCPCB_LOST;
874         }
875 }
876
877 static void tcp_skb_mark_lost_uncond_verify(struct tcp_sock *tp,
878                                             struct sk_buff *skb)
879 {
880         tcp_verify_retransmit_hint(tp, skb);
881
882         if (!(TCP_SKB_CB(skb)->sacked & (TCPCB_LOST|TCPCB_SACKED_ACKED))) {
883                 tp->lost_out += tcp_skb_pcount(skb);
884                 TCP_SKB_CB(skb)->sacked |= TCPCB_LOST;
885         }
886 }
887
888 /* This procedure tags the retransmission queue when SACKs arrive.
889  *
890  * We have three tag bits: SACKED(S), RETRANS(R) and LOST(L).
891  * Packets in queue with these bits set are counted in variables
892  * sacked_out, retrans_out and lost_out, correspondingly.
893  *
894  * Valid combinations are:
895  * Tag  InFlight        Description
896  * 0    1               - orig segment is in flight.
897  * S    0               - nothing flies, orig reached receiver.
898  * L    0               - nothing flies, orig lost by net.
899  * R    2               - both orig and retransmit are in flight.
900  * L|R  1               - orig is lost, retransmit is in flight.
901  * S|R  1               - orig reached receiver, retrans is still in flight.
902  * (L|S|R is logically valid, it could occur when L|R is sacked,
903  *  but it is equivalent to plain S and code short-curcuits it to S.
904  *  L|S is logically invalid, it would mean -1 packet in flight 8))
905  *
906  * These 6 states form finite state machine, controlled by the following events:
907  * 1. New ACK (+SACK) arrives. (tcp_sacktag_write_queue())
908  * 2. Retransmission. (tcp_retransmit_skb(), tcp_xmit_retransmit_queue())
909  * 3. Loss detection event of two flavors:
910  *      A. Scoreboard estimator decided the packet is lost.
911  *         A'. Reno "three dupacks" marks head of queue lost.
912  *         A''. Its FACK modification, head until snd.fack is lost.
913  *      B. SACK arrives sacking SND.NXT at the moment, when the
914  *         segment was retransmitted.
915  * 4. D-SACK added new rule: D-SACK changes any tag to S.
916  *
917  * It is pleasant to note, that state diagram turns out to be commutative,
918  * so that we are allowed not to be bothered by order of our actions,
919  * when multiple events arrive simultaneously. (see the function below).
920  *
921  * Reordering detection.
922  * --------------------
923  * Reordering metric is maximal distance, which a packet can be displaced
924  * in packet stream. With SACKs we can estimate it:
925  *
926  * 1. SACK fills old hole and the corresponding segment was not
927  *    ever retransmitted -> reordering. Alas, we cannot use it
928  *    when segment was retransmitted.
929  * 2. The last flaw is solved with D-SACK. D-SACK arrives
930  *    for retransmitted and already SACKed segment -> reordering..
931  * Both of these heuristics are not used in Loss state, when we cannot
932  * account for retransmits accurately.
933  *
934  * SACK block validation.
935  * ----------------------
936  *
937  * SACK block range validation checks that the received SACK block fits to
938  * the expected sequence limits, i.e., it is between SND.UNA and SND.NXT.
939  * Note that SND.UNA is not included to the range though being valid because
940  * it means that the receiver is rather inconsistent with itself reporting
941  * SACK reneging when it should advance SND.UNA. Such SACK block this is
942  * perfectly valid, however, in light of RFC2018 which explicitly states
943  * that "SACK block MUST reflect the newest segment.  Even if the newest
944  * segment is going to be discarded ...", not that it looks very clever
945  * in case of head skb. Due to potentional receiver driven attacks, we
946  * choose to avoid immediate execution of a walk in write queue due to
947  * reneging and defer head skb's loss recovery to standard loss recovery
948  * procedure that will eventually trigger (nothing forbids us doing this).
```

```
949  *
950  * Implements also blockage to start_seq wrap-around. Problem lies in the
951  * fact that though start_seq (s) is before end_seq (i.e., not reversed),
952  * there's no guarantee that it will be before snd_nxt (n). The problem
953  * happens when start_seq resides between end_seq wrap (e_w) and snd_nxt
954  * wrap (s_w):
955  *
956  *         <- outs wnd ->                          <- wrapzone ->
957  *         u     e      n                         u_w   e_w  s n_w
958  *         |     |      |                          |     |   |  |
959  * |<------------+------+----- TCP seqno space --------------+---------->|
960  * ...-- <2^31 ->|                                           |<--------...
961  * ...---- >2^31 ------>|                                    |<--------...
962  *
963  * Current code wouldn't be vulnerable but it's better still to discard such
964  * crazy SACK blocks. Doing this check for start_seq alone closes somewhat
965  * similar case (end_seq after snd_nxt wrap) as earlier reversed check in
966  * snd_nxt wrap -> snd_una region will then become "well defined", i.e.,
967  * equal to the ideal case (infinite seqno space without wrap caused issues).
968  *
969  * With D-SACK the lower bound is extended to cover sequence space below
970  * SND.UNA down to undo_marker, which is the last point of interest. Yet
971  * again, D-SACK block must not to go across snd_una (for the same reason as
972  * for the normal SACK blocks, explained above). But there all simplicity
973  * ends, TCP might receive valid D-SACKs below that. As long as they reside
974  * fully below undo_marker they do not affect behavior in anyway and can
975  * therefore be safely ignored. In rare cases (which are more or less
976  * theoretical ones), the D-SACK will nicely cross that boundary due to skb
977  * fragmentation and packet reordering past skb's retransmission. To consider
978  * them correctly, the acceptable range must be extended even more though
979  * the exact amount is rather hard to quantify. However, tp->max_window can
980  * be used as an exaggerated estimate.
981  */
982 static bool tcp_is_sackblock_valid(struct tcp_sock *tp, bool is_dsack,
983                                    u32 start_seq, u32 end_seq)
984 {
985         /* Too far in future, or reversed (interpretation is ambiguous) */
986         if (after(end_seq, tp->snd_nxt) || !before(start_seq, end_seq))
987                 return false;
988
989         /* Nasty start_seq wrap-around check (see comments above) */
990         if (!before(start_seq, tp->snd_nxt))
991                 return false;
992
993         /* In outstanding window? ...This is valid exit for D-SACKs too.
994          * start_seq == snd_una is non-sensical (see comments above)
995          */
996         if (after(start_seq, tp->snd_una))
997                 return true;
998
999         if (!is_dsack || !tp->undo_marker)
1000                 return false;
1001
1002         /* ...Then it's D-SACK, and must reside below snd_una completely */
1003         if (after(end_seq, tp->snd_una))
1004                 return false;
1005
1006         if (!before(start_seq, tp->undo_marker))
1007                 return true;
1008
1009         /* Too old */
1010         if (!after(end_seq, tp->undo_marker))
1011                 return false;
1012
1013         /* Undo_marker boundary crossing (overestimates a lot). Known already:
1014          *   start_seq < undo_marker and end_seq >= undo_marker.
1015          */
1016         return !before(start_seq, end_seq - tp->max_window);
1017 }
1018
1019 /* Check for lost retransmit. This superb idea is borrowed from "ratehalving".
1020  * Event "B". Later note: FACK people cheated me again 8), we have to account
1021  * for reordering! Ugly, but should help.
1022  *
1023  * Search retransmitted skbs from write_queue that were sent when snd_nxt was
1024  * less than what is now known to be received by the other end (derived from
1025  * highest SACK block). Also calculate the lowest snd_nxt among the remaining
1026  * retransmitted skbs to avoid some costly processing per ACKs.
1027  */
1028 static void tcp_mark_lost_retrans(struct sock *sk)
1029 {
1030         const struct inet_connection_sock *icsk = inet_csk(sk);
1031         struct tcp_sock *tp = tcp_sk(sk);
1032         struct sk_buff *skb;
1033         int cnt = 0;
1034         u32 new_low_seq = tp->snd_nxt;
1035         u32 received_upto = tcp_highest_sack_seq(tp);
1036
```

```
1037         if (!tcp_is_fack(tp) || !tp->retrans_out ||
1038             !after(received_upto, tp->lost_retrans_low) ||
1039             icsk->icsk_ca_state != TCP_CA_Recovery)
1040                 return;
1041
1042         tcp_for_write_queue(skb, sk) {
1043                 u32 ack_seq = TCP_SKB_CB(skb)->ack_seq;
1044
1045                 if (skb == tcp_send_head(sk))
1046                         break;
1047                 if (cnt == tp->retrans_out)
1048                         break;
1049                 if (!after(TCP_SKB_CB(skb)->end_seq, tp->snd_una))
1050                         continue;
1051
1052                 if (!(TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_RETRANS))
1053                         continue;
1054
1055                 /* TODO: We would like to get rid of tcp_is_fack(tp) only
1056                  * constraint here (see above) but figuring out that at
1057                  * least tp->reordering SACK blocks reside between ack_seq
1058                  * and received_upto is not easy task to do cheaply with
1059                  * the available datastructures.
1060                  *
1061                  * Whether FACK should check here for tp->reordering segs
1062                  * in-between one could argue for either way (it would be
1063                  * rather simple to implement as we could count fack_count
1064                  * during the walk and do tp->fackets_out - fack_count).
1065                  */
1066                 if (after(received_upto, ack_seq)) {
1067                         TCP_SKB_CB(skb)->sacked &= ~TCPCB_SACKED_RETRANS;
1068                         tp->retrans_out -= tcp_skb_pcount(skb);
1069
1070                         tcp_skb_mark_lost_uncond_verify(tp, skb);
1071                         NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPLOSTRETRANSMIT);
1072                 } else {
1073                         if (before(ack_seq, new_low_seq))
1074                                 new_low_seq = ack_seq;
1075                         cnt += tcp_skb_pcount(skb);
1076                 }
1077         }
1078
1079         if (tp->retrans_out)
1080                 tp->lost_retrans_low = new_low_seq;
1081 }
1082
1083 static bool tcp_check_dsack(struct sock *sk, const struct sk_buff *ack_skb,
1084                             struct tcp_sack_block_wire *sp, int num_sacks,
1085                             u32 prior_snd_una)
1086 {
1087         struct tcp_sock *tp = tcp_sk(sk);
1088         u32 start_seq_0 = get_unaligned_be32(&sp[0].start_seq);
1089         u32 end_seq_0 = get_unaligned_be32(&sp[0].end_seq);
1090         bool dup_sack = false;
1091
1092         if (before(start_seq_0, TCP_SKB_CB(ack_skb)->ack_seq)) {
1093                 dup_sack = true;
1094                 tcp_dsack_seen(tp);
1095                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPDSACKRECV);
1096         } else if (num_sacks > 1) {
1097                 u32 end_seq_1 = get_unaligned_be32(&sp[1].end_seq);
1098                 u32 start_seq_1 = get_unaligned_be32(&sp[1].start_seq);
1099
1100                 if (!after(end_seq_0, end_seq_1) &&
1101                     !before(start_seq_0, start_seq_1)) {
1102                         dup_sack = true;
1103                         tcp_dsack_seen(tp);
1104                         NET_INC_STATS_BH(sock_net(sk),
1105                                         LINUX_MIB_TCPDSACKOFORECV);
1106                 }
1107         }
1108
1109         /* D-SACK for already forgotten data... Do dumb counting. */
1110         if (dup_sack && tp->undo_marker && tp->undo_retrans > 0 &&
1111             !after(end_seq_0, prior_snd_una) &&
1112             after(end_seq_0, tp->undo_marker))
1113                 tp->undo_retrans--;
1114
1115         return dup_sack;
1116 }
1117
1118 struct tcp_sacktag_state {
1119         int     reord;
1120         int     fack_count;
1121         long    rtt_us; /* RTT measured by SACKing never-retransmitted data */
1122         int     flag;
1123 };
1124
```

```
1125 /* Check if skb is fully within the SACK block. In presence of GSO skbs,
1126  * the incoming SACK may not exactly match but we can find smaller MSS
1127  * aligned portion of it that matches. Therefore we might need to fragment
1128  * which may fail and creates some hassle (caller must handle error case
1129  * returns).
1130  *
1131  * FIXME: this could be merged to shift decision code
1132  */
1133 static int tcp_match_skb_to_sack(struct sock *sk, struct sk_buff *skb,
1134                                  u32 start_seq, u32 end_seq)
1135 {
1136         int err;
1137         bool in_sack;
1138         unsigned int pkt_len;
1139         unsigned int mss;
1140
1141         in_sack = !after(start_seq, TCP_SKB_CB(skb)->seq) &&
1142                   !before(end_seq, TCP_SKB_CB(skb)->end_seq);
1143
1144         if (tcp_skb_pcount(skb) > 1 && !in_sack &&
1145             after(TCP_SKB_CB(skb)->end_seq, start_seq)) {
1146                 mss = tcp_skb_mss(skb);
1147                 in_sack = !after(start_seq, TCP_SKB_CB(skb)->seq);
1148
1149                 if (!in_sack) {
1150                         pkt_len = start_seq - TCP_SKB_CB(skb)->seq;
1151                         if (pkt_len < mss)
1152                                 pkt_len = mss;
1153                 } else {
1154                         pkt_len = end_seq - TCP_SKB_CB(skb)->seq;
1155                         if (pkt_len < mss)
1156                                 return -EINVAL;
1157                 }
1158
1159                 /* Round if necessary so that SACKs cover only full MSSes
1160                  * and/or the remaining small portion (if present)
1161                  */
1162                 if (pkt_len > mss) {
1163                         unsigned int new_len = (pkt_len / mss) * mss;
1164                         if (!in_sack && new_len < pkt_len) {
1165                                 new_len += mss;
1166                                 if (new_len >= skb->len)
1167                                         return 0;
1168                         }
1169                         pkt_len = new_len;
1170                 }
1171                 err = tcp_fragment(sk, skb, pkt_len, mss, GFP_ATOMIC);
1172                 if (err < 0)
1173                         return err;
1174         }
1175
1176         return in_sack;
1177 }
1178
1179 /* Mark the given newly-SACKed range as such, adjusting counters and hints. */
1180 static u8 tcp_sacktag_one(struct sock *sk,
1181                           struct tcp_sacktag_state *state, u8 sacked,
1182                           u32 start_seq, u32 end_seq,
1183                           int dup_sack, int pcount,
1184                           const struct skb_mstamp *xmit_time)
1185 {
1186         struct tcp_sock *tp = tcp_sk(sk);
1187         int fack_count = state->fack_count;
1188
1189         /* Account D-SACK for retransmitted packet. */
1190         if (dup_sack && (sacked & TCPCB_RETRANS)) {
1191                 if (tp->undo_marker && tp->undo_retrans > 0 &&
1192                     after(end_seq, tp->undo_marker))
1193                         tp->undo_retrans--;
1194                 if (sacked & TCPCB_SACKED_ACKED)
1195                         state->reord = min(fack_count, state->reord);
1196         }
1197
1198         /* Nothing to do; acked frame is about to be dropped (was ACKed). */
1199         if (!after(end_seq, tp->snd_una))
1200                 return sacked;
1201
1202         if (!(sacked & TCPCB_SACKED_ACKED)) {
1203                 if (sacked & TCPCB_SACKED_RETRANS) {
1204                         /* If the segment is not tagged as lost,
1205                          * we do not clear RETRANS, believing
1206                          * that retransmission is still in flight.
1207                          */
1208                         if (sacked & TCPCB_LOST) {
1209                                 sacked &= ~(TCPCB_LOST|TCPCB_SACKED_RETRANS);
1210                                 tp->lost_out -= pcount;
1211                                 tp->retrans_out -= pcount;
1212                         }
```

```
1213                } else {
1214                        if (!(sacked & TCPCB_RETRANS)) {
1215                                /* New sack for not retransmitted frame,
1216                                 * which was in hole. It is reordering.
1217                                 */
1218                                if (before(start_seq,
1219                                           tcp_highest_sack_seq(tp)))
1220                                        state->reord = min(fack_count,
1221                                                           state->reord);
1222                                if (!after(end_seq, tp->high_seq))
1223                                        state->flag |= FLAG_ORIG_SACK_ACKED;
1224                                /* Pick the earliest sequence sacked for RTT */
1225                                if (state->rtt_us < 0) {
1226                                        struct skb_mstamp now;
1227
1228                                        skb_mstamp_get(&now);
1229                                        state->rtt_us = skb_mstamp_us_delta(&now,
1230                                                                           xmit_time);
1231                                }
1232                        }
1233
1234                        if (sacked & TCPCB_LOST) {
1235                                sacked &= ~TCPCB_LOST;
1236                                tp->lost_out -= pcount;
1237                        }
1238                }
1239
1240                sacked |= TCPCB_SACKED_ACKED;
1241                state->flag |= FLAG_DATA_SACKED;
1242                tp->sacked_out += pcount;
1243
1244                fack_count += pcount;
1245
1246                /* Lost marker hint past SACKed? Tweak RFC3517 cnt */
1247                if (!tcp_is_fack(tp) && (tp->lost_skb_hint != NULL) &&
1248                    before(start_seq, TCP_SKB_CB(tp->lost_skb_hint)->seq))
1249                        tp->lost_cnt_hint += pcount;
1250
1251                if (fack_count > tp->fackets_out)
1252                        tp->fackets_out = fack_count;
1253        }
1254
1255        /* D-SACK. We can detect redundant retransmission in S|R and plain R
1256         * frames and clear it. undo_retrans is decreased above, L|R frames
1257         * are accounted above as well.
1258         */
1259        if (dup_sack && (sacked & TCPCB_SACKED_RETRANS)) {
1260                sacked &= ~TCPCB_SACKED_RETRANS;
1261                tp->retrans_out -= pcount;
1262        }
1263
1264        return sacked;
1265 }
1266
1267 /* Shift newly-SACKed bytes from this skb to the immediately previous
1268  * already-SACKed sk_buff. Mark the newly-SACKed bytes as such.
1269  */
1270 static bool tcp_shifted_skb(struct sock *sk, struct sk_buff *skb,
1271                             struct tcp_sacktag_state *state,
1272                             unsigned int pcount, int shifted, int mss,
1273                             bool dup_sack)
1274 {
1275        struct tcp_sock *tp = tcp_sk(sk);
1276        struct sk_buff *prev = tcp_write_queue_prev(sk, skb);
1277        u32 start_seq = TCP_SKB_CB(skb)->seq;   /* start of newly-SACKed */
1278        u32 end_seq = start_seq + shifted;      /* end of newly-SACKed */
1279
1280        BUG_ON(!pcount);
1281
1282        /* Adjust counters and hints for the newly sacked sequence
1283         * range but discard the return value since prev is already
1284         * marked. We must tag the range first because the seq
1285         * advancement below implicitly advances
1286         * tcp_highest_sack_seq() when skb is highest_sack.
1287         */
1288        tcp_sacktag_one(sk, state, TCP_SKB_CB(skb)->sacked,
1289                        start_seq, end_seq, dup_sack, pcount,
1290                        &skb->skb_mstamp);
1291
1292        if (skb == tp->lost_skb_hint)
1293                tp->lost_cnt_hint += pcount;
1294
1295        TCP_SKB_CB(prev)->end_seq += shifted;
1296        TCP_SKB_CB(skb)->seq += shifted;
1297
1298        skb_shinfo(prev)->gso_segs += pcount;
1299        BUG_ON(skb_shinfo(skb)->gso_segs < pcount);
1300        skb_shinfo(skb)->gso_segs -= pcount;
```

```
1301
1302            /* When we're adding to gso_segs == 1, gso_size will be zero,
1303             * in theory this shouldn't be necessary but as long as DSACK
1304             * code can come after this skb later on it's better to keep
1305             * setting gso_size to something.
1306             */
1307            if (!skb_shinfo(prev)->gso_size) {
1308                    skb_shinfo(prev)->gso_size = mss;
1309                    skb_shinfo(prev)->gso_type = sk->sk_gso_type;
1310            }
1311
1312            /* CHECKME: To clear or not to clear? Mimics normal skb currently */
1313            if (skb_shinfo(skb)->gso_segs <= 1) {
1314                    skb_shinfo(skb)->gso_size = 0;
1315                    skb_shinfo(skb)->gso_type = 0;
1316            }
1317
1318            /* Difference in this won't matter, both ACKed by the same cumul. ACK */
1319            TCP_SKB_CB(prev)->sacked |= (TCP_SKB_CB(skb)->sacked & TCPCB_EVER_RETRANS);
1320
1321            if (skb->len > 0) {
1322                    BUG_ON(!tcp_skb_pcount(skb));
1323                    NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_SACKSHIFTED);
1324                    return false;
1325            }
1326
1327            /* Whole SKB was eaten :-) */
1328
1329            if (skb == tp->retransmit_skb_hint)
1330                    tp->retransmit_skb_hint = prev;
1331            if (skb == tp->lost_skb_hint) {
1332                    tp->lost_skb_hint = prev;
1333                    tp->lost_cnt_hint -= tcp_skb_pcount(prev);
1334            }
1335
1336            TCP_SKB_CB(prev)->tcp_flags |= TCP_SKB_CB(skb)->tcp_flags;
1337            if (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_FIN)
1338                    TCP_SKB_CB(prev)->end_seq++;
1339
1340            if (skb == tcp_highest_sack(sk))
1341                    tcp_advance_highest_sack(sk, skb);
1342
1343            tcp_unlink_write_queue(skb, sk);
1344            sk_wmem_free_skb(sk, skb);
1345
1346            NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_SACKMERGED);
1347
1348            return true;
1349    }
1350
1351    /* I wish gso_size would have a bit more sane initialization than
1352     * something-or-zero which complicates things
1353     */
1354    static int tcp_skb_seglen(const struct sk_buff *skb)
1355    {
1356            return tcp_skb_pcount(skb) == 1 ? skb->len : tcp_skb_mss(skb);
1357    }
1358
1359    /* Shifting pages past head area doesn't work */
1360    static int skb_can_shift(const struct sk_buff *skb)
1361    {
1362            return !skb_headlen(skb) && skb_is_nonlinear(skb);
1363    }
1364
1365    /* Try collapsing SACK blocks spanning across multiple skbs to a single
1366     * skb.
1367     */
1368    static struct sk_buff *tcp_shift_skb_data(struct sock *sk, struct sk_buff *skb,
1369                                              struct tcp_sacktag_state *state,
1370                                              u32 start_seq, u32 end_seq,
1371                                              bool dup_sack)
1372    {
1373            struct tcp_sock *tp = tcp_sk(sk);
1374            struct sk_buff *prev;
1375            int mss;
1376            int pcount = 0;
1377            int len;
1378            int in_sack;
1379
1380            if (!sk_can_gso(sk))
1381                    goto fallback;
1382
1383            /* Normally R but no L won't result in plain S */
1384            if (!dup_sack &&
1385                (TCP_SKB_CB(skb)->sacked & (TCPCB_LOST|TCPCB_SACKED_RETRANS)) == TCPCB_SACKED_RETRANS)
1386                    goto fallback;
1387            if (!skb_can_shift(skb))
1388                    goto fallback;
```

```
1389         /* This frame is about to be dropped (was ACKed). */
1390         if (!after(TCP_SKB_CB(skb)->end_seq, tp->snd_una))
1391                 goto fallback;
1392
1393         /* Can only happen with delayed DSACK + discard craziness */
1394         if (unlikely(skb == tcp_write_queue_head(sk)))
1395                 goto fallback;
1396         prev = tcp_write_queue_prev(sk, skb);
1397
1398         if ((TCP_SKB_CB(prev)->sacked & TCPCB_TAGBITS) != TCPCB_SACKED_ACKED)
1399                 goto fallback;
1400
1401         in_sack = !after(start_seq, TCP_SKB_CB(skb)->seq) &&
1402                   !before(end_seq, TCP_SKB_CB(skb)->end_seq);
1403
1404         if (in_sack) {
1405                 len = skb->len;
1406                 pcount = tcp_skb_pcount(skb);
1407                 mss = tcp_skb_seglen(skb);
1408
1409                 /* TODO: Fix DSACKs to not fragment already SACKed and we can
1410                  * drop this restriction as unnecessary
1411                  */
1412                 if (mss != tcp_skb_seglen(prev))
1413                         goto fallback;
1414         } else {
1415                 if (!after(TCP_SKB_CB(skb)->end_seq, start_seq))
1416                         goto noop;
1417                 /* CHECKME: This is non-MSS split case only?, this will
1418                  * cause skipped skbs due to advancing loop btw, original
1419                  * has that feature too
1420                  */
1421                 if (tcp_skb_pcount(skb) <= 1)
1422                         goto noop;
1423
1424                 in_sack = !after(start_seq, TCP_SKB_CB(skb)->seq);
1425                 if (!in_sack) {
1426                         /* TODO: head merge to next could be attempted here
1427                          * if (!after(TCP_SKB_CB(skb)->end_seq, end_seq)),
1428                          * though it might not be worth of the additional hassle
1429                          *
1430                          * ...we can probably just fallback to what was done
1431                          * previously. We could try merging non-SACKed ones
1432                          * as well but it probably isn't going to buy off
1433                          * because later SACKs might again split them, and
1434                          * it would make skb timestamp tracking considerably
1435                          * harder problem.
1436                          */
1437                         goto fallback;
1438                 }
1439
1440                 len = end_seq - TCP_SKB_CB(skb)->seq;
1441                 BUG_ON(len < 0);
1442                 BUG_ON(len > skb->len);
1443
1444                 /* MSS boundaries should be honoured or else pcount will
1445                  * severely break even though it makes things bit trickier.
1446                  * Optimize common case to avoid most of the divides
1447                  */
1448                 mss = tcp_skb_mss(skb);
1449
1450                 /* TODO: Fix DSACKs to not fragment already SACKed and we can
1451                  * drop this restriction as unnecessary
1452                  */
1453                 if (mss != tcp_skb_seglen(prev))
1454                         goto fallback;
1455
1456                 if (len == mss) {
1457                         pcount = 1;
1458                 } else if (len < mss) {
1459                         goto noop;
1460                 } else {
1461                         pcount = len / mss;
1462                         len = pcount * mss;
1463                 }
1464         }
1465
1466         /* tcp_sacktag_one() won't SACK-tag ranges below snd_una */
1467         if (!after(TCP_SKB_CB(skb)->seq + len, tp->snd_una))
1468                 goto fallback;
1469
1470         if (!skb_shift(prev, skb, len))
1471                 goto fallback;
1472         if (!tcp_shifted_skb(sk, skb, state, pcount, len, mss, dup_sack))
1473                 goto out;
1474
1475         /* Hole filled allows collapsing with the next as well, this is very
1476          * useful when hole on every nth skb pattern happens
```

```
1477          */
1478         if (prev == tcp_write_queue_tail(sk))
1479                 goto out;
1480         skb = tcp_write_queue_next(sk, prev);
1481
1482         if (!skb_can_shift(skb) ||
1483             (skb == tcp_send_head(sk)) ||
1484             ((TCP_SKB_CB(skb)->sacked & TCPCB_TAGBITS) != TCPCB_SACKED_ACKED) ||
1485             (mss != tcp_skb_seglen(skb)))
1486                 goto out;
1487
1488         len = skb->len;
1489         if (skb_shift(prev, skb, len)) {
1490                 pcount += tcp_skb_pcount(skb);
1491                 tcp_shifted_skb(sk, skb, state, tcp_skb_pcount(skb), len, mss, 0);
1492         }
1493
1494 out:
1495         state->fack_count += pcount;
1496         return prev;
1497
1498 noop:
1499         return skb;
1500
1501 fallback:
1502         NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_SACKSHIFTFALLBACK);
1503         return NULL;
1504 }
1505
1506 static struct sk_buff *tcp_sacktag_walk(struct sk_buff *skb, struct sock *sk,
1507                                         struct tcp_sack_block *next_dup,
1508                                         struct tcp_sacktag_state *state,
1509                                         u32 start_seq, u32 end_seq,
1510                                         bool dup_sack_in)
1511 {
1512         struct tcp_sock *tp = tcp_sk(sk);
1513         struct sk_buff *tmp;
1514
1515         tcp_for_write_queue_from(skb, sk) {
1516                 int in_sack = 0;
1517                 bool dup_sack = dup_sack_in;
1518
1519                 if (skb == tcp_send_head(sk))
1520                         break;
1521
1522                 /* queue is in-order => we can short-circuit the walk early */
1523                 if (!before(TCP_SKB_CB(skb)->seq, end_seq))
1524                         break;
1525
1526                 if ((next_dup != NULL) &&
1527                     before(TCP_SKB_CB(skb)->seq, next_dup->end_seq)) {
1528                         in_sack = tcp_match_skb_to_sack(sk, skb,
1529                                                         next_dup->start_seq,
1530                                                         next_dup->end_seq);
1531                         if (in_sack > 0)
1532                                 dup_sack = true;
1533                 }
1534
1535                 /* skb reference here is a bit tricky to get right, since
1536                  * shifting can eat and free both this skb and the next,
1537                  * so not even _safe variant of the loop is enough.
1538                  */
1539                 if (in_sack <= 0) {
1540                         tmp = tcp_shift_skb_data(sk, skb, state,
1541                                                  start_seq, end_seq, dup_sack);
1542                         if (tmp != NULL) {
1543                                 if (tmp != skb) {
1544                                         skb = tmp;
1545                                         continue;
1546                                 }
1547
1548                                 in_sack = 0;
1549                         } else {
1550                                 in_sack = tcp_match_skb_to_sack(sk, skb,
1551                                                                 start_seq,
1552                                                                 end_seq);
1553                         }
1554                 }
1555
1556                 if (unlikely(in_sack < 0))
1557                         break;
1558
1559                 if (in_sack) {
1560                         TCP_SKB_CB(skb)->sacked =
1561                                 tcp_sacktag_one(sk,
1562                                                 state,
1563                                                 TCP_SKB_CB(skb)->sacked,
1564                                                 TCP_SKB_CB(skb)->seq,
```

```
1565                                                TCP_SKB_CB(skb)->end_seq,
1566                                                dup_sack,
1567                                                tcp_skb_pcount(skb),
1568                                                &skb->skb_mstamp);
1569
1570                        if (!before(TCP_SKB_CB(skb)->seq,
1571                                    tcp_highest_sack_seq(tp)))
1572                                tcp_advance_highest_sack(sk, skb);
1573                }
1574
1575                state->fack_count += tcp_skb_pcount(skb);
1576        }
1577        return skb;
1578 }
1579
1580 /* Avoid all extra work that is being done by sacktag while walking in
1581  * a normal way
1582  */
1583 static struct sk_buff *tcp_sacktag_skip(struct sk_buff *skb, struct sock *sk,
1584                                         struct tcp_sacktag_state *state,
1585                                         u32 skip_to_seq)
1586 {
1587        tcp_for_write_queue_from(skb, sk) {
1588                if (skb == tcp_send_head(sk))
1589                        break;
1590
1591                if (after(TCP_SKB_CB(skb)->end_seq, skip_to_seq))
1592                        break;
1593
1594                state->fack_count += tcp_skb_pcount(skb);
1595        }
1596        return skb;
1597 }
1598
1599 static struct sk_buff *tcp_maybe_skipping_dsack(struct sk_buff *skb,
1600                                                 struct sock *sk,
1601                                                 struct tcp_sack_block *next_dup,
1602                                                 struct tcp_sacktag_state *state,
1603                                                 u32 skip_to_seq)
1604 {
1605        if (next_dup == NULL)
1606                return skb;
1607
1608        if (before(next_dup->start_seq, skip_to_seq)) {
1609                skb = tcp_sacktag_skip(skb, sk, state, next_dup->start_seq);
1610                skb = tcp_sacktag_walk(skb, sk, NULL, state,
1611                                       next_dup->start_seq, next_dup->end_seq,
1612                                       1);
1613        }
1614
1615        return skb;
1616 }
1617
1618 static int tcp_sack_cache_ok(const struct tcp_sock *tp, const struct tcp_sack_block *cache)
1619 {
1620        return cache < tp->recv_sack_cache + ARRAY_SIZE(tp->recv_sack_cache);
1621 }
1622
1623 static int
1624 tcp_sacktag_write_queue(struct sock *sk, const struct sk_buff *ack_skb,
1625                         u32 prior_snd_una, long *sack_rtt_us)
1626 {
1627        struct tcp_sock *tp = tcp_sk(sk);
1628        const unsigned char *ptr = (skb_transport_header(ack_skb) +
1629                                    TCP_SKB_CB(ack_skb)->sacked);
1630        struct tcp_sack_block_wire *sp_wire = (struct tcp_sack_block_wire *)(ptr+2);
1631        struct tcp_sack_block sp[TCP_NUM_SACKS];
1632        struct tcp_sack_block *cache;
1633        struct tcp_sacktag_state state;
1634        struct sk_buff *skb;
1635        int num_sacks = min(TCP_NUM_SACKS, (ptr[1] - TCPOLEN_SACK_BASE) >> 3);
1636        int used_sacks;
1637        bool found_dup_sack = false;
1638        int i, j;
1639        int first_sack_index;
1640
1641        state.flag = 0;
1642        state.reord = tp->packets_out;
1643        state.rtt_us = -1L;
1644
1645        if (!tp->sacked_out) {
1646                if (WARN_ON(tp->fackets_out))
1647                        tp->fackets_out = 0;
1648                tcp_highest_sack_reset(sk);
1649        }
1650
1651        found_dup_sack = tcp_check_dsack(sk, ack_skb, sp_wire,
1652                                         num_sacks, prior_snd_una);
```

```
1653        if (found_dup_sack)
1654                state.flag |= FLAG_DSACKING_ACK;
1655
1656        /* Eliminate too old ACKs, but take into
1657         * account more or less fresh ones, they can
1658         * contain valid SACK info.
1659         */
1660        if (before(TCP_SKB_CB(ack_skb)->ack_seq, prior_snd_una - tp->max_window))
1661                return 0;
1662
1663        if (!tp->packets_out)
1664                goto out;
1665
1666        used_sacks = 0;
1667        first_sack_index = 0;
1668        for (i = 0; i < num_sacks; i++) {
1669                bool dup_sack = !i && found_dup_sack;
1670
1671                sp[used_sacks].start_seq = get_unaligned_be32(&sp_wire[i].start_seq);
1672                sp[used_sacks].end_seq = get_unaligned_be32(&sp_wire[i].end_seq);
1673
1674                if (!tcp_is_sackblock_valid(tp, dup_sack,
1675                                            sp[used_sacks].start_seq,
1676                                            sp[used_sacks].end_seq)) {
1677                        int mib_idx;
1678
1679                        if (dup_sack) {
1680                                if (!tp->undo_marker)
1681                                        mib_idx = LINUX_MIB_TCPDSACKIGNOREDNOUNDO;
1682                                else
1683                                        mib_idx = LINUX_MIB_TCPDSACKIGNOREDOLD;
1684                        } else {
1685                                /* Don't count olds caused by ACK reordering */
1686                                if ((TCP_SKB_CB(ack_skb)->ack_seq != tp->snd_una) &&
1687                                    !after(sp[used_sacks].end_seq, tp->snd_una))
1688                                        continue;
1689                                mib_idx = LINUX_MIB_TCPSACKDISCARD;
1690                        }
1691
1692                        NET_INC_STATS_BH(sock_net(sk), mib_idx);
1693                        if (i == 0)
1694                                first_sack_index = -1;
1695                        continue;
1696                }
1697
1698                /* Ignore very old stuff early */
1699                if (!after(sp[used_sacks].end_seq, prior_snd_una))
1700                        continue;
1701
1702                used_sacks++;
1703        }
1704
1705        /* order SACK blocks to allow in order walk of the retrans queue */
1706        for (i = used_sacks - 1; i > 0; i--) {
1707                for (j = 0; j < i; j++) {
1708                        if (after(sp[j].start_seq, sp[j + 1].start_seq)) {
1709                                swap(sp[j], sp[j + 1]);
1710
1711                                /* Track where the first SACK block goes to */
1712                                if (j == first_sack_index)
1713                                        first_sack_index = j + 1;
1714                        }
1715                }
1716        }
1717
1718        skb = tcp_write_queue_head(sk);
1719        state.fack_count = 0;
1720        i = 0;
1721
1722        if (!tp->sacked_out) {
1723                /* It's already past, so skip checking against it */
1724                cache = tp->recv_sack_cache + ARRAY_SIZE(tp->recv_sack_cache);
1725        } else {
1726                cache = tp->recv_sack_cache;
1727                /* Skip empty blocks in at head of the cache */
1728                while (tcp_sack_cache_ok(tp, cache) && !cache->start_seq &&
1729                       !cache->end_seq)
1730                        cache++;
1731        }
1732
1733        while (i < used_sacks) {
1734                u32 start_seq = sp[i].start_seq;
1735                u32 end_seq = sp[i].end_seq;
1736                bool dup_sack = (found_dup_sack && (i == first_sack_index));
1737                struct tcp_sack_block *next_dup = NULL;
1738
1739                if (found_dup_sack && ((i + 1) == first_sack_index))
1740                        next_dup = &sp[i + 1];
```

```
1741
1742                /* Skip too early cached blocks */
1743                while (tcp_sack_cache_ok(tp, cache) &&
1744                       !before(start_seq, cache->end_seq))
1745                        cache++;
1746
1747                /* Can skip some work by looking recv_sack_cache? */
1748                if (tcp_sack_cache_ok(tp, cache) && !dup_sack &&
1749                    after(end_seq, cache->start_seq)) {
1750
1751                        /* Head todo? */
1752                        if (before(start_seq, cache->start_seq)) {
1753                                skb = tcp_sacktag_skip(skb, sk, &state,
1754                                                       start_seq);
1755                                skb = tcp_sacktag_walk(skb, sk, next_dup,
1756                                                       &state,
1757                                                       start_seq,
1758                                                       cache->start_seq,
1759                                                       dup_sack);
1760                        }
1761
1762                        /* Rest of the block already fully processed? */
1763                        if (!after(end_seq, cache->end_seq))
1764                                goto advance_sp;
1765
1766                        skb = tcp_maybe_skipping_dsack(skb, sk, next_dup,
1767                                                       &state,
1768                                                       cache->end_seq);
1769
1770                        /* ...tail remains todo... */
1771                        if (tcp_highest_sack_seq(tp) == cache->end_seq) {
1772                                /* ...but better entrypoint exists! */
1773                                skb = tcp_highest_sack(sk);
1774                                if (skb == NULL)
1775                                        break;
1776                                state.fack_count = tp->fackets_out;
1777                                cache++;
1778                                goto walk;
1779                        }
1780
1781                        skb = tcp_sacktag_skip(skb, sk, &state, cache->end_seq);
1782                        /* Check overlap against next cached too (past this one already) */
1783                        cache++;
1784                        continue;
1785                }
1786
1787                if (!before(start_seq, tcp_highest_sack_seq(tp))) {
1788                        skb = tcp_highest_sack(sk);
1789                        if (skb == NULL)
1790                                break;
1791                        state.fack_count = tp->fackets_out;
1792                }
1793                skb = tcp_sacktag_skip(skb, sk, &state, start_seq);
1794
1795 walk:
1796                skb = tcp_sacktag_walk(skb, sk, next_dup, &state,
1797                                       start_seq, end_seq, dup_sack);
1798
1799 advance_sp:
1800                i++;
1801        }
1802
1803        /* Clear the head of the cache sack blocks so we can skip it next time */
1804        for (i = 0; i < ARRAY_SIZE(tp->recv_sack_cache) - used_sacks; i++) {
1805                tp->recv_sack_cache[i].start_seq = 0;
1806                tp->recv_sack_cache[i].end_seq = 0;
1807        }
1808        for (j = 0; j < used_sacks; j++)
1809                tp->recv_sack_cache[i++] = sp[j];
1810
1811        tcp_mark_lost_retrans(sk);
1812
1813        tcp_verify_left_out(tp);
1814
1815        if ((state.reord < tp->fackets_out) &&
1816            ((inet_csk(sk)->icsk_ca_state != TCP_CA_Loss) || tp->undo_marker))
1817                tcp_update_reordering(sk, tp->fackets_out - state.reord, 0);
1818
1819 out:
1820
1821 #if FASTRETRANS_DEBUG > 0
1822        WARN_ON((int)tp->sacked_out < 0);
1823        WARN_ON((int)tp->lost_out < 0);
1824        WARN_ON((int)tp->retrans_out < 0);
1825        WARN_ON((int)tcp_packets_in_flight(tp) < 0);
1826 #endif
1827        *sack_rtt_us = state.rtt_us;
1828        return state.flag;
```

```
1829 }
1830
1831 /* Limits sacked_out so that sum with lost_out isn't ever larger than
1832  * packets_out. Returns false if sacked_out adjustement wasn't necessary.
1833  */
1834 static bool tcp_limit_reno_sacked(struct tcp_sock *tp)
1835 {
1836         u32 holes;
1837
1838         holes = max(tp->lost_out, 1U);
1839         holes = min(holes, tp->packets_out);
1840
1841         if ((tp->sacked_out + holes) > tp->packets_out) {
1842                 tp->sacked_out = tp->packets_out - holes;
1843                 return true;
1844         }
1845         return false;
1846 }
1847
1848 /* If we receive more dupacks than we expected counting segments
1849  * in assumption of absent reordering, interpret this as reordering.
1850  * The only another reason could be bug in receiver TCP.
1851  */
1852 static void tcp_check_reno_reordering(struct sock *sk, const int addend)
1853 {
1854         struct tcp_sock *tp = tcp_sk(sk);
1855         if (tcp_limit_reno_sacked(tp))
1856                 tcp_update_reordering(sk, tp->packets_out + addend, 0);
1857 }
1858
1859 /* Emulate SACKs for SACKless connection: account for a new dupack. */
1860
1861 static void tcp_add_reno_sack(struct sock *sk)
1862 {
1863         struct tcp_sock *tp = tcp_sk(sk);
1864         tp->sacked_out++;
1865         tcp_check_reno_reordering(sk, 0);
1866         tcp_verify_left_out(tp);
1867 }
1868
1869 /* Account for ACK, ACKing some data in Reno Recovery phase. */
1870
1871 static void tcp_remove_reno_sacks(struct sock *sk, int acked)
1872 {
1873         struct tcp_sock *tp = tcp_sk(sk);
1874
1875         if (acked > 0) {
1876                 /* One ACK acked hole. The rest eat duplicate ACKs. */
1877                 if (acked - 1 >= tp->sacked_out)
1878                         tp->sacked_out = 0;
1879                 else
1880                         tp->sacked_out -= acked - 1;
1881         }
1882         tcp_check_reno_reordering(sk, acked);
1883         tcp_verify_left_out(tp);
1884 }
1885
1886 static inline void tcp_reset_reno_sack(struct tcp_sock *tp)
1887 {
1888         tp->sacked_out = 0;
1889 }
1890
1891 static void tcp_clear_retrans_partial(struct tcp_sock *tp)
1892 {
1893         tp->retrans_out = 0;
1894         tp->lost_out = 0;
1895
1896         tp->undo_marker = 0;
1897         tp->undo_retrans = -1;
1898 }
1899
1900 void tcp_clear_retrans(struct tcp_sock *tp)
1901 {
1902         tcp_clear_retrans_partial(tp);
1903
1904         tp->fackets_out = 0;
1905         tp->sacked_out = 0;
1906 }
1907
1908 /* Enter Loss state. If we detect SACK reneging, forget all SACK information
1909  * and reset tags completely, otherwise preserve SACKs. If receiver
1910  * dropped its ofo queue, we will know this due to reneging detection.
1911  */
1912 void tcp_enter_loss(struct sock *sk)
1913 {
1914         const struct inet_connection_sock *icsk = inet_csk(sk);
1915         struct tcp_sock *tp = tcp_sk(sk);
1916         struct sk_buff *skb;
```

```
1917         bool new_recovery = false;
1918         bool is_reneg;                  /* is receiver reneging on SACKs? */
1919
1920         /* Reduce ssthresh if it has not yet been made inside this window. */
1921         if (icsk->icsk_ca_state <= TCP_CA_Disorder ||
1922             !after(tp->high_seq, tp->snd_una) ||
1923             (icsk->icsk_ca_state == TCP_CA_Loss && !icsk->icsk_retransmits)) {
1924                 new_recovery = true;
1925                 tp->prior_ssthresh = tcp_current_ssthresh(sk);
1926                 tp->snd_ssthresh = icsk->icsk_ca_ops->ssthresh(sk);
1927                 tcp_ca_event(sk, CA_EVENT_LOSS);
1928         }
1929         tp->snd_cwnd       = 1;
1930         tp->snd_cwnd_cnt   = 0;
1931         tp->snd_cwnd_stamp = tcp_time_stamp;
1932
1933         tcp_clear_retrans_partial(tp);
1934
1935         if (tcp_is_reno(tp))
1936                 tcp_reset_reno_sack(tp);
1937
1938         tp->undo_marker = tp->snd_una;
1939
1940         skb = tcp_write_queue_head(sk);
1941         is_reneg = skb && (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED);
1942         if (is_reneg) {
1943                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPSACKRENEGING);
1944                 tp->sacked_out = 0;
1945                 tp->fackets_out = 0;
1946         }
1947         tcp_clear_all_retrans_hints(tp);
1948
1949         tcp_for_write_queue(skb, sk) {
1950                 if (skb == tcp_send_head(sk))
1951                         break;
1952
1953                 if (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_RETRANS)
1954                         tp->undo_marker = 0;
1955
1956                 TCP_SKB_CB(skb)->sacked &= (~TCPCB_TAGBITS)|TCPCB_SACKED_ACKED;
1957                 if (!(TCP_SKB_CB(skb)->sacked&TCPCB_SACKED_ACKED) || is_reneg) {
1958                         TCP_SKB_CB(skb)->sacked &= ~TCPCB_SACKED_ACKED;
1959                         TCP_SKB_CB(skb)->sacked |= TCPCB_LOST;
1960                         tp->lost_out += tcp_skb_pcount(skb);
1961                         tp->retransmit_high = TCP_SKB_CB(skb)->end_seq;
1962                 }
1963         }
1964         tcp_verify_left_out(tp);
1965
1966         /* Timeout in disordered state after receiving substantial DUPACKs
1967          * suggests that the degree of reordering is over-estimated.
1968          */
1969         if (icsk->icsk_ca_state <= TCP_CA_Disorder &&
1970             tp->sacked_out >= sysctl_tcp_reordering)
1971                 tp->reordering = min_t(unsigned int, tp->reordering,
1972                                        sysctl_tcp_reordering);
1973         tcp_set_ca_state(sk, TCP_CA_Loss);
1974         tp->high_seq = tp->snd_nxt;
1975         TCP_ECN_queue_cwr(tp);
1976
1977         /* F-RTO RFC5682 sec 3.1 step 1: retransmit SND.UNA if no previous
1978          * loss recovery is underway except recurring timeout(s) on
1979          * the same SND.UNA (sec 3.2). Disable F-RTO on path MTU probing
1980          */
1981         tp->frto = sysctl_tcp_frto &&
1982                    (new_recovery || icsk->icsk_retransmits) &&
1983                    !inet_csk(sk)->icsk_mtup.probe_size;
1984 }
1985
1986 /* If ACK arrived pointing to a remembered SACK, it means that our
1987  * remembered SACKs do not reflect real state of receiver i.e.
1988  * receiver _host_ is heavily congested (or buggy).
1989  *
1990  * To avoid big spurious retransmission bursts due to transient SACK
1991  * scoreboard oddities that look like reneging, we give the receiver a
1992  * little time (max(RTT/2, 10ms)) to send us some more ACKs that will
1993  * restore sanity to the SACK scoreboard. If the apparent reneging
1994  * persists until this RTO then we'll clear the SACK scoreboard.
1995  */
1996 static bool tcp_check_sack_reneging(struct sock *sk, int flag)
1997 {
1998         if (flag & FLAG_SACK_RENEGING) {
1999                 struct tcp_sock *tp = tcp_sk(sk);
2000                 unsigned long delay = max(usecs_to_jiffies(tp->srtt_us >> 4),
2001                                           msecs_to_jiffies(10));
2002
2003                 inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
2004                                           delay, TCP_RTO_MAX);
```

```
2005                return true;
2006        }
2007        return false;
2008 }
2009
2010 static inline int tcp_fackets_out(const struct tcp_sock *tp)
2011 {
2012        return tcp_is_reno(tp) ? tp->sacked_out + 1 : tp->fackets_out;
2013 }
2014
2015 /* Heurestics to calculate number of duplicate ACKs. There's no dupACKs
2016  * counter when SACK is enabled (without SACK, sacked_out is used for
2017  * that purpose).
2018  *
2019  * Instead, with FACK TCP uses fackets_out that includes both SACKed
2020  * segments up to the highest received SACK block so far and holes in
2021  * between them.
2022  *
2023  * With reordering, holes may still be in flight, so RFC3517 recovery
2024  * uses pure sacked_out (total number of SACKed segments) even though
2025  * it violates the RFC that uses duplicate ACKs, often these are equal
2026  * but when e.g. out-of-window ACKs or packet duplication occurs,
2027  * they differ. Since neither occurs due to loss, TCP should really
2028  * ignore them.
2029  */
2030 static inline int tcp_dupack_heuristics(const struct tcp_sock *tp)
2031 {
2032        return tcp_is_fack(tp) ? tp->fackets_out : tp->sacked_out + 1;
2033 }
2034
2035 static bool tcp_pause_early_retransmit(struct sock *sk, int flag)
2036 {
2037        struct tcp_sock *tp = tcp_sk(sk);
2038        unsigned long delay;
2039
2040        /* Delay early retransmit and entering fast recovery for
2041         * max(RTT/4, 2msec) unless ack has ECE mark, no RTT samples
2042         * available, or RTO is scheduled to fire first.
2043         */
2044        if (sysctl_tcp_early_retrans < 2 || sysctl_tcp_early_retrans > 3 ||
2045            (flag & FLAG_ECE) || !tp->srtt_us)
2046                return false;
2047
2048        delay = max(usecs_to_jiffies(tp->srtt_us >> 5),
2049                    msecs_to_jiffies(2));
2050
2051        if (!time_after(inet_csk(sk)->icsk_timeout, (jiffies + delay)))
2052                return false;
2053
2054        inet_csk_reset_xmit_timer(sk, ICSK_TIME_EARLY_RETRANS, delay,
2055                                  TCP_RTO_MAX);
2056        return true;
2057 }
2058
2059 /* Linux NewReno/SACK/FACK/ECN state machine.
2060  * --------------------------------------
2061  *
2062  * "Open"       Normal state, no dubious events, fast path.
2063  * "Disorder"   In all the respects it is "Open",
2064  *              but requires a bit more attention. It is entered when
2065  *              we see some SACKs or dupacks. It is split of "Open"
2066  *              mainly to move some processing from fast path to slow one.
2067  * "CWR"        CWND was reduced due to some Congestion Notification event.
2068  *              It can be ECN, ICMP source quench, local device congestion.
2069  * "Recovery"   CWND was reduced, we are fast-retransmitting.
2070  * "Loss"       CWND was reduced due to RTO timeout or SACK reneging.
2071  *
2072  * tcp_fastretrans_alert() is entered:
2073  * - each incoming ACK, if state is not "Open"
2074  * - when arrived ACK is unusual, namely:
2075  *      * SACK
2076  *      * Duplicate ACK.
2077  *      * ECN ECE.
2078  *
2079  * Counting packets in flight is pretty simple.
2080  *
2081  *      in_flight = packets_out - left_out + retrans_out
2082  *
2083  *      packets_out is SND.NXT-SND.UNA counted in packets.
2084  *
2085  *      retrans_out is number of retransmitted segments.
2086  *
2087  *      left_out is number of segments left network, but not ACKed yet.
2088  *
2089  *              left_out = sacked_out + lost_out
2090  *
2091  *     sacked_out: Packets, which arrived to receiver out of order
2092  *                 and hence not ACKed. With SACKs this number is simply
```

```
2093  *                amount of SACKed data. Even without SACKs
2094  *                it is easy to give pretty reliable estimate of this number,
2095  *                counting duplicate ACKs.
2096  *
2097  *       lost_out: Packets lost by network. TCP has no explicit
2098  *                "loss notification" feedback from network (for now).
2099  *                It means that this number can be only _guessed_.
2100  *                Actually, it is the heuristics to predict lossage that
2101  *                distinguishes different algorithms.
2102  *
2103  *      F.e. after RTO, when all the queue is considered as lost,
2104  *      lost_out = packets_out and in_flight = retrans_out.
2105  *
2106  *              Essentially, we have now two algorithms counting
2107  *              lost packets.
2108  *
2109  *              FACK: It is the simplest heuristics. As soon as we decided
2110  *              that something is lost, we decide that _all_ not SACKed
2111  *              packets until the most forward SACK are lost. I.e.
2112  *              lost_out = fackets_out - sacked_out and left_out = fackets_out.
2113  *              It is absolutely correct estimate, if network does not reorder
2114  *              packets. And it loses any connection to reality when reordering
2115  *              takes place. We use FACK by default until reordering
2116  *              is suspected on the path to this destination.
2117  *
2118  *              NewReno: when Recovery is entered, we assume that one segment
2119  *              is lost (classic Reno). While we are in Recovery and
2120  *              a partial ACK arrives, we assume that one more packet
2121  *              is lost (NewReno). This heuristics are the same in NewReno
2122  *              and SACK.
2123  *
2124  *  Imagine, that's all! Forget about all this shamanism about CWND inflation
2125  *  deflation etc. CWND is real congestion window, never inflated, changes
2126  *  only according to classic VJ rules.
2127  *
2128  * Really tricky (and requiring careful tuning) part of algorithm
2129  * is hidden in functions tcp_time_to_recover() and tcp_xmit_retransmit_queue().
2130  * The first determines the moment _when_ we should reduce CWND and,
2131  * hence, slow down forward transmission. In fact, it determines the moment
2132  * when we decide that hole is caused by loss, rather than by a reorder.
2133  *
2134  * tcp_xmit_retransmit_queue() decides, _what_ we should retransmit to fill
2135  * holes, caused by lost packets.
2136  *
2137  * And the most logically complicated part of algorithm is undo
2138  * heuristics. We detect false retransmits due to both too early
2139  * fast retransmit (reordering) and underestimated RTO, analyzing
2140  * timestamps and D-SACKs. When we detect that some segments were
2141  * retransmitted by mistake and CWND reduction was wrong, we undo
2142  * window reduction and abort recovery phase. This logic is hidden
2143  * inside several functions named tcp_try_undo_<something>.
2144  */
2145
2146 /* This function decides, when we should leave Disordered state
2147  * and enter Recovery phase, reducing congestion window.
2148  *
2149  * Main question: may we further continue forward transmission
2150  * with the same cwnd?
2151  */
2152 static bool tcp_time_to_recover(struct sock *sk, int flag)
2153 {
2154         struct tcp_sock *tp = tcp_sk(sk);
2155         __u32 packets_out;
2156
2157         /* Trick#1: The loss is proven. */
2158         if (tp->lost_out)
2159                 return true;
2160
2161         /* Not-A-Trick#2 : Classic rule... */
2162         if (tcp_dupack_heuristics(tp) > tp->reordering)
2163                 return true;
2164
2165         /* Trick#4: It is still not OK... But will it be useful to delay
2166          * recovery more?
2167          */
2168         packets_out = tp->packets_out;
2169         if (packets_out <= tp->reordering &&
2170             tp->sacked_out >= max_t(__u32, packets_out/2, sysctl_tcp_reordering) &&
2171             !tcp_may_send_now(sk)) {
2172                 /* We have nothing to send. This connection is limited
2173                  * either by receiver window or by application.
2174                  */
2175                 return true;
2176         }
2177
2178         /* If a thin stream is detected, retransmit after first
2179          * received dupack. Employ only if SACK is supported in order
2180          * to avoid possible corner-case series of spurious retransmissions
```

```
2181          * Use only if there are no unsent data.
2182          */
2183         if ((tp->thin_dupack || sysctl_tcp_thin_dupack) &&
2184             tcp_stream_is_thin(tp) && tcp_dupack_heuristics(tp) > 1 &&
2185             tcp_is_sack(tp) && !tcp_send_head(sk))
2186                 return true;
2187
2188         /* Trick#6: TCP early retransmit, per RFC5827.  To avoid spurious
2189          * retransmissions due to small network reorderings, we implement
2190          * Mitigation A.3 in the RFC and delay the retransmission for a short
2191          * interval if appropriate.
2192          */
2193         if (tp->do_early_retrans && !tp->retrans_out && tp->sacked_out &&
2194             (tp->packets_out >= (tp->sacked_out + 1) && tp->packets_out < 4) &&
2195             !tcp_may_send_now(sk))
2196                 return !tcp_pause_early_retransmit(sk, flag);
2197
2198         return false;
2199 }
2200
2201 /* Detect loss in event "A" above by marking head of queue up as lost.
2202  * For FACK or non-SACK(Reno) senders, the first "packets" number of segments
2203  * are considered lost. For RFC3517 SACK, a segment is considered lost if it
2204  * has at least tp->reordering SACKed seqments above it; "packets" refers to
2205  * the maximum SACKed segments to pass before reaching this limit.
2206  */
2207 static void tcp_mark_head_lost(struct sock *sk, int packets, int mark_head)
2208 {
2209         struct tcp_sock *tp = tcp_sk(sk);
2210         struct sk_buff *skb;
2211         int cnt, oldcnt;
2212         int err;
2213         unsigned int mss;
2214         /* Use SACK to deduce losses of new sequences sent during recovery */
2215         const u32 loss_high = tcp_is_sack(tp) ?  tp->snd_nxt : tp->high_seq;
2216
2217         WARN_ON(packets > tp->packets_out);
2218         if (tp->lost_skb_hint) {
2219                 skb = tp->lost_skb_hint;
2220                 cnt = tp->lost_cnt_hint;
2221                 /* Head already handled? */
2222                 if (mark_head && skb != tcp_write_queue_head(sk))
2223                         return;
2224         } else {
2225                 skb = tcp_write_queue_head(sk);
2226                 cnt = 0;
2227         }
2228
2229         tcp_for_write_queue_from(skb, sk) {
2230                 if (skb == tcp_send_head(sk))
2231                         break;
2232                 /* TODO: do this better */
2233                 /* this is not the most efficient way to do this... */
2234                 tp->lost_skb_hint = skb;
2235                 tp->lost_cnt_hint = cnt;
2236
2237                 if (after(TCP_SKB_CB(skb)->end_seq, loss_high))
2238                         break;
2239
2240                 oldcnt = cnt;
2241                 if (tcp_is_fack(tp) || tcp_is_reno(tp) ||
2242                     (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED))
2243                         cnt += tcp_skb_pcount(skb);
2244
2245                 if (cnt > packets) {
2246                         if ((tcp_is_sack(tp) && !tcp_is_fack(tp)) ||
2247                             (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED) ||
2248                             (oldcnt >= packets))
2249                                 break;
2250
2251                         mss = skb_shinfo(skb)->gso_size;
2252                         err = tcp_fragment(sk, skb, (packets - oldcnt) * mss,
2253                                            mss, GFP_ATOMIC);
2254                         if (err < 0)
2255                                 break;
2256                         cnt = packets;
2257                 }
2258
2259                 tcp_skb_mark_lost(tp, skb);
2260
2261                 if (mark_head)
2262                         break;
2263         }
2264         tcp_verify_left_out(tp);
2265 }
2266
2267 /* Account newly detected lost packet(s) */
2268
```

```
2269 static void tcp_update_scoreboard(struct sock *sk, int fast_rexmit)
2270 {
2271         struct tcp_sock *tp = tcp_sk(sk);
2272
2273         if (tcp_is_reno(tp)) {
2274                 tcp_mark_head_lost(sk, 1, 1);
2275         } else if (tcp_is_fack(tp)) {
2276                 int lost = tp->fackets_out - tp->reordering;
2277                 if (lost <= 0)
2278                         lost = 1;
2279                 tcp_mark_head_lost(sk, lost, 0);
2280         } else {
2281                 int sacked_upto = tp->sacked_out - tp->reordering;
2282                 if (sacked_upto >= 0)
2283                         tcp_mark_head_lost(sk, sacked_upto, 0);
2284                 else if (fast_rexmit)
2285                         tcp_mark_head_lost(sk, 1, 1);
2286         }
2287 }
2288
2289 /* CWND moderation, preventing bursts due to too big ACKs
2290  * in dubious situations.
2291  */
2292 static inline void tcp_moderate_cwnd(struct tcp_sock *tp)
2293 {
2294         tp->snd_cwnd = min(tp->snd_cwnd,
2295                            tcp_packets_in_flight(tp) + tcp_max_burst(tp));
2296         tp->snd_cwnd_stamp = tcp_time_stamp;
2297 }
2298
2299 /* Nothing was retransmitted or returned timestamp is less
2300  * than timestamp of the first retransmission.
2301  */
2302 static inline bool tcp_packet_delayed(const struct tcp_sock *tp)
2303 {
2304         return !tp->retrans_stamp ||
2305                 (tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr &&
2306                  before(tp->rx_opt.rcv_tsecr, tp->retrans_stamp));
2307 }
2308
2309 /* Undo procedures. */
2310
2311 #if FASTRETRANS_DEBUG > 1
2312 static void DBGUNDO(struct sock *sk, const char *msg)
2313 {
2314         struct tcp_sock *tp = tcp_sk(sk);
2315         struct inet_sock *inet = inet_sk(sk);
2316
2317         if (sk->sk_family == AF_INET) {
2318                 pr_debug("Undo %s %pI4/%u c%u l%u ss%u/%u p%u\n",
2319                          msg,
2320                          &inet->inet_daddr, ntohs(inet->inet_dport),
2321                          tp->snd_cwnd, tcp_left_out(tp),
2322                          tp->snd_ssthresh, tp->prior_ssthresh,
2323                          tp->packets_out);
2324         }
2325 #if IS_ENABLED(CONFIG_IPV6)
2326         else if (sk->sk_family == AF_INET6) {
2327                 struct ipv6_pinfo *np = inet6_sk(sk);
2328                 pr_debug("Undo %s %pI6/%u c%u l%u ss%u/%u p%u\n",
2329                          msg,
2330                          &np->daddr, ntohs(inet->inet_dport),
2331                          tp->snd_cwnd, tcp_left_out(tp),
2332                          tp->snd_ssthresh, tp->prior_ssthresh,
2333                          tp->packets_out);
2334         }
2335 #endif
2336 }
2337 #else
2338 #define DBGUNDO(x...) do { } while (0)
2339 #endif
2340
2341 static void tcp_undo_cwnd_reduction(struct sock *sk, bool unmark_loss)
2342 {
2343         struct tcp_sock *tp = tcp_sk(sk);
2344
2345         if (unmark_loss) {
2346                 struct sk_buff *skb;
2347
2348                 tcp_for_write_queue(skb, sk) {
2349                         if (skb == tcp_send_head(sk))
2350                                 break;
2351                         TCP_SKB_CB(skb)->sacked &= ~TCPCB_LOST;
2352                 }
2353                 tp->lost_out = 0;
2354                 tcp_clear_all_retrans_hints(tp);
2355         }
2356
```

```
2357         if (tp->prior_ssthresh) {
2358                 const struct inet_connection_sock *icsk = inet_csk(sk);
2359
2360                 if (icsk->icsk_ca_ops->undo_cwnd)
2361                         tp->snd_cwnd = icsk->icsk_ca_ops->undo_cwnd(sk);
2362                 else
2363                         tp->snd_cwnd = max(tp->snd_cwnd, tp->snd_ssthresh << 1);
2364
2365                 if (tp->prior_ssthresh > tp->snd_ssthresh) {
2366                         tp->snd_ssthresh = tp->prior_ssthresh;
2367                         TCP_ECN_withdraw_cwr(tp);
2368                 }
2369         } else {
2370                 tp->snd_cwnd = max(tp->snd_cwnd, tp->snd_ssthresh);
2371         }
2372         tp->snd_cwnd_stamp = tcp_time_stamp;
2373         tp->undo_marker = 0;
2374 }
2375
2376 static inline bool tcp_may_undo(const struct tcp_sock *tp)
2377 {
2378         return tp->undo_marker && (!tp->undo_retrans || tcp_packet_delayed(tp));
2379 }
2380
2381 /* People celebrate: "We love our President!" */
2382 static bool tcp_try_undo_recovery(struct sock *sk)
2383 {
2384         struct tcp_sock *tp = tcp_sk(sk);
2385
2386         if (tcp_may_undo(tp)) {
2387                 int mib_idx;
2388
2389                 /* Happy end! We did not retransmit anything
2390                  * or our original transmission succeeded.
2391                  */
2392                 DBGUNDO(sk, inet_csk(sk)->icsk_ca_state == TCP_CA_Loss ? "loss" : "retrans");
2393                 tcp_undo_cwnd_reduction(sk, false);
2394                 if (inet_csk(sk)->icsk_ca_state == TCP_CA_Loss)
2395                         mib_idx = LINUX_MIB_TCPLOSSUNDO;
2396                 else
2397                         mib_idx = LINUX_MIB_TCPFULLUNDO;
2398
2399                 NET_INC_STATS_BH(sock_net(sk), mib_idx);
2400         }
2401         if (tp->snd_una == tp->high_seq && tcp_is_reno(tp)) {
2402                 /* Hold old state until something *above* high_seq
2403                  * is ACKed. For Reno it is MUST to prevent false
2404                  * fast retransmits (RFC2582). SACK TCP is safe. */
2405                 tcp_moderate_cwnd(tp);
2406                 return true;
2407         }
2408         tcp_set_ca_state(sk, TCP_CA_Open);
2409         return false;
2410 }
2411
2412 /* Try to undo cwnd reduction, because D-SACKs acked all retransmitted data */
2413 static bool tcp_try_undo_dsack(struct sock *sk)
2414 {
2415         struct tcp_sock *tp = tcp_sk(sk);
2416
2417         if (tp->undo_marker && !tp->undo_retrans) {
2418                 DBGUNDO(sk, "D-SACK");
2419                 tcp_undo_cwnd_reduction(sk, false);
2420                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPDSACKUNDO);
2421                 return true;
2422         }
2423         return false;
2424 }
2425
2426 /* We can clear retrans_stamp when there are no retransmissions in the
2427  * window. It would seem that it is trivially available for us in
2428  * tp->retrans_out, however, that kind of assumptions doesn't consider
2429  * what will happen if errors occur when sending retransmission for the
2430  * second time. ...It could the that such segment has only
2431  * TCPCB_EVER_RETRANS set at the present time. It seems that checking
2432  * the head skb is enough except for some reneging corner cases that
2433  * are not worth the effort.
2434  *
2435  * Main reason for all this complexity is the fact that connection dying
2436  * time now depends on the validity of the retrans_stamp, in particular,
2437  * that successive retransmissions of a segment must not advance
2438  * retrans_stamp under any conditions.
2439  */
2440 static bool tcp_any_retrans_done(const struct sock *sk)
2441 {
2442         const struct tcp_sock *tp = tcp_sk(sk);
2443         struct sk_buff *skb;
2444
```

```
2445         if (tp->retrans_out)
2446                 return true;
2447
2448         skb = tcp_write_queue_head(sk);
2449         if (unlikely(skb && TCP_SKB_CB(skb)->sacked & TCPCB_EVER_RETRANS))
2450                 return true;
2451
2452         return false;
2453 }
2454
2455 /* Undo during loss recovery after partial ACK or using F-RTO. */
2456 static bool tcp_try_undo_loss(struct sock *sk, bool frto_undo)
2457 {
2458         struct tcp_sock *tp = tcp_sk(sk);
2459
2460         if (frto_undo || tcp_may_undo(tp)) {
2461                 tcp_undo_cwnd_reduction(sk, true);
2462
2463                 DBGUNDO(sk, "partial loss");
2464                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPLOSSUNDO);
2465                 if (frto_undo)
2466                         NET_INC_STATS_BH(sock_net(sk),
2467                                          LINUX_MIB_TCPSPURIOUSRTOS);
2468                 inet_csk(sk)->icsk_retransmits = 0;
2469                 if (frto_undo || tcp_is_sack(tp))
2470                         tcp_set_ca_state(sk, TCP_CA_Open);
2471                 return true;
2472         }
2473         return false;
2474 }
2475
2476 /* The cwnd reduction in CWR and Recovery use the PRR algorithm
2477  * https://datatracker.ietf.org/doc/draft-ietf-tcpm-proportional-rate-reduction/
2478  * It computes the number of packets to send (sndcnt) based on packets newly
2479  * delivered:
2480  *   1) If the packets in flight is larger than ssthresh, PRR spreads the
2481  *      cwnd reductions across a full RTT.
2482  *   2) If packets in flight is lower than ssthresh (such as due to excess
2483  *      losses and/or application stalls), do not perform any further cwnd
2484  *      reductions, but instead slow start up to ssthresh.
2485  */
2486 static void tcp_init_cwnd_reduction(struct sock *sk)
2487 {
2488         struct tcp_sock *tp = tcp_sk(sk);
2489
2490         tp->high_seq = tp->snd_nxt;
2491         tp->tlp_high_seq = 0;
2492         tp->snd_cwnd_cnt = 0;
2493         tp->prior_cwnd = tp->snd_cwnd;
2494         tp->prr_delivered = 0;
2495         tp->prr_out = 0;
2496         tp->snd_ssthresh = inet_csk(sk)->icsk_ca_ops->ssthresh(sk);
2497         TCP_ECN_queue_cwr(tp);
2498 }
2499
2500 static void tcp_cwnd_reduction(struct sock *sk, const int prior_unsacked,
2501                                int fast_rexmit)
2502 {
2503         struct tcp_sock *tp = tcp_sk(sk);
2504         int sndcnt = 0;
2505         int delta = tp->snd_ssthresh - tcp_packets_in_flight(tp);
2506         int newly_acked_sacked = prior_unsacked -
2507                                  (tp->packets_out - tp->sacked_out);
2508
2509         tp->prr_delivered += newly_acked_sacked;
2510         if (tcp_packets_in_flight(tp) > tp->snd_ssthresh) {
2511                 u64 dividend = (u64)tp->snd_ssthresh * tp->prr_delivered +
2512                                tp->prior_cwnd - 1;
2513                 sndcnt = div_u64(dividend, tp->prior_cwnd) - tp->prr_out;
2514         } else {
2515                 sndcnt = min_t(int, delta,
2516                                max_t(int, tp->prr_delivered - tp->prr_out,
2517                                      newly_acked_sacked) + 1);
2518         }
2519
2520         sndcnt = max(sndcnt, (fast_rexmit ? 1 : 0));
2521         tp->snd_cwnd = tcp_packets_in_flight(tp) + sndcnt;
2522 }
2523
2524 static inline void tcp_end_cwnd_reduction(struct sock *sk)
2525 {
2526         struct tcp_sock *tp = tcp_sk(sk);
2527
2528         /* Reset cwnd to ssthresh in CWR or Recovery (unless it's undone) */
2529         if (inet_csk(sk)->icsk_ca_state == TCP_CA_CWR ||
2530             (tp->undo_marker && tp->snd_ssthresh < TCP_INFINITE_SSTHRESH)) {
2531                 tp->snd_cwnd = tp->snd_ssthresh;
2532                 tp->snd_cwnd_stamp = tcp_time_stamp;
```

```
2533         }
2534         tcp_ca_event(sk, CA_EVENT_COMPLETE_CWR);
2535 }
2536
2537 /* Enter CWR state. Disable cwnd undo since congestion is proven with ECN */
2538 void tcp_enter_cwr(struct sock *sk)
2539 {
2540         struct tcp_sock *tp = tcp_sk(sk);
2541
2542         tp->prior_ssthresh = 0;
2543         if (inet_csk(sk)->icsk_ca_state < TCP_CA_CWR) {
2544                 tp->undo_marker = 0;
2545                 tcp_init_cwnd_reduction(sk);
2546                 tcp_set_ca_state(sk, TCP_CA_CWR);
2547         }
2548 }
2549
2550 static void tcp_try_keep_open(struct sock *sk)
2551 {
2552         struct tcp_sock *tp = tcp_sk(sk);
2553         int state = TCP_CA_Open;
2554
2555         if (tcp_left_out(tp) || tcp_any_retrans_done(sk))
2556                 state = TCP_CA_Disorder;
2557
2558         if (inet_csk(sk)->icsk_ca_state != state) {
2559                 tcp_set_ca_state(sk, state);
2560                 tp->high_seq = tp->snd_nxt;
2561         }
2562 }
2563
2564 static void tcp_try_to_open(struct sock *sk, int flag, const int prior_unsacked)
2565 {
2566         struct tcp_sock *tp = tcp_sk(sk);
2567
2568         tcp_verify_left_out(tp);
2569
2570         if (!tcp_any_retrans_done(sk))
2571                 tp->retrans_stamp = 0;
2572
2573         if (flag & FLAG_ECE)
2574                 tcp_enter_cwr(sk);
2575
2576         if (inet_csk(sk)->icsk_ca_state != TCP_CA_CWR) {
2577                 tcp_try_keep_open(sk);
2578         } else {
2579                 tcp_cwnd_reduction(sk, prior_unsacked, 0);
2580         }
2581 }
2582
2583 static void tcp_mtup_probe_failed(struct sock *sk)
2584 {
2585         struct inet_connection_sock *icsk = inet_csk(sk);
2586
2587         icsk->icsk_mtup.search_high = icsk->icsk_mtup.probe_size - 1;
2588         icsk->icsk_mtup.probe_size = 0;
2589 }
2590
2591 static void tcp_mtup_probe_success(struct sock *sk)
2592 {
2593         struct tcp_sock *tp = tcp_sk(sk);
2594         struct inet_connection_sock *icsk = inet_csk(sk);
2595
2596         /* FIXME: breaks with very large cwnd */
2597         tp->prior_ssthresh = tcp_current_ssthresh(sk);
2598         tp->snd_cwnd = tp->snd_cwnd *
2599                        tcp_mss_to_mtu(sk, tp->mss_cache) /
2600                        icsk->icsk_mtup.probe_size;
2601         tp->snd_cwnd_cnt = 0;
2602         tp->snd_cwnd_stamp = tcp_time_stamp;
2603         tp->snd_ssthresh = tcp_current_ssthresh(sk);
2604
2605         icsk->icsk_mtup.search_low = icsk->icsk_mtup.probe_size;
2606         icsk->icsk_mtup.probe_size = 0;
2607         tcp_sync_mss(sk, icsk->icsk_pmtu_cookie);
2608 }
2609
2610 /* Do a simple retransmit without using the backoff mechanisms in
2611  * tcp_timer. This is used for path mtu discovery.
2612  * The socket is already locked here.
2613  */
2614 void tcp_simple_retransmit(struct sock *sk)
2615 {
2616         const struct inet_connection_sock *icsk = inet_csk(sk);
2617         struct tcp_sock *tp = tcp_sk(sk);
2618         struct sk_buff *skb;
2619         unsigned int mss = tcp_current_mss(sk);
2620         u32 prior_lost = tp->lost_out;
```

```
2621
2622            tcp_for_write_queue(skb, sk) {
2623                    if (skb == tcp_send_head(sk))
2624                            break;
2625                    if (tcp_skb_seglen(skb) > mss &&
2626                        !(TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED)) {
2627                            if (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_RETRANS) {
2628                                    TCP_SKB_CB(skb)->sacked &= ~TCPCB_SACKED_RETRANS;
2629                                    tp->retrans_out -= tcp_skb_pcount(skb);
2630                            }
2631                            tcp_skb_mark_lost_uncond_verify(tp, skb);
2632                    }
2633            }
2634
2635            tcp_clear_retrans_hints_partial(tp);
2636
2637            if (prior_lost == tp->lost_out)
2638                    return;
2639
2640            if (tcp_is_reno(tp))
2641                    tcp_limit_reno_sacked(tp);
2642
2643            tcp_verify_left_out(tp);
2644
2645            /* Don't muck with the congestion window here.
2646             * Reason is that we do not increase amount of _data_
2647             * in network, but units changed and effective
2648             * cwnd/ssthresh really reduced now.
2649             */
2650            if (icsk->icsk_ca_state != TCP_CA_Loss) {
2651                    tp->high_seq = tp->snd_nxt;
2652                    tp->snd_ssthresh = tcp_current_ssthresh(sk);
2653                    tp->prior_ssthresh = 0;
2654                    tp->undo_marker = 0;
2655                    tcp_set_ca_state(sk, TCP_CA_Loss);
2656            }
2657            tcp_xmit_retransmit_queue(sk);
2658    }
2659    EXPORT_SYMBOL(tcp_simple_retransmit);
2660
2661    static void tcp_enter_recovery(struct sock *sk, bool ece_ack)
2662    {
2663            struct tcp_sock *tp = tcp_sk(sk);
2664            int mib_idx;
2665
2666            if (tcp_is_reno(tp))
2667                    mib_idx = LINUX_MIB_TCPRENORECOVERY;
2668            else
2669                    mib_idx = LINUX_MIB_TCPSACKRECOVERY;
2670
2671            NET_INC_STATS_BH(sock_net(sk), mib_idx);
2672
2673            tp->prior_ssthresh = 0;
2674            tp->undo_marker = tp->snd_una;
2675            tp->undo_retrans = tp->retrans_out ? : -1;
2676
2677            if (inet_csk(sk)->icsk_ca_state < TCP_CA_CWR) {
2678                    if (!ece_ack)
2679                            tp->prior_ssthresh = tcp_current_ssthresh(sk);
2680                    tcp_init_cwnd_reduction(sk);
2681            }
2682            tcp_set_ca_state(sk, TCP_CA_Recovery);
2683    }
2684
2685    /* Process an ACK in CA_Loss state. Move to CA_Open if lost data are
2686     * recovered or spurious. Otherwise retransmits more on partial ACKs.
2687     */
2688    static void tcp_process_loss(struct sock *sk, int flag, bool is_dupack)
2689    {
2690            struct tcp_sock *tp = tcp_sk(sk);
2691            bool recovered = !before(tp->snd_una, tp->high_seq);
2692
2693            if (tp->frto) { /* F-RTO RFC5682 sec 3.1 (sack enhanced version). */
2694                    /* Step 3.b. A timeout is spurious if not all data are
2695                     * lost, i.e., never-retransmitted data are (s)acked.
2696                     */
2697                    if (tcp_try_undo_loss(sk, flag & FLAG_ORIG_SACK_ACKED))
2698                            return;
2699
2700                    if (after(tp->snd_nxt, tp->high_seq) &&
2701                        (flag & FLAG_DATA_SACKED || is_dupack)) {
2702                            tp->frto = 0; /* Loss was real: 2nd part of step 3.a */
2703                    } else if (flag & FLAG_SND_UNA_ADVANCED && !recovered) {
2704                            tp->high_seq = tp->snd_nxt;
2705                            __tcp_push_pending_frames(sk, tcp_current_mss(sk),
2706                                                      TCP_NAGLE_OFF);
2707                            if (after(tp->snd_nxt, tp->high_seq))
2708                                    return; /* Step 2.b */
```

```
2709                        tp->frto = 0;
2710                }
2711        }
2712
2713        if (recovered) {
2714                /* F-RTO RFC5682 sec 3.1 step 2.a and 1st part of step 3.a */
2715                tcp_try_undo_recovery(sk);
2716                return;
2717        }
2718        if (tcp_is_reno(tp)) {
2719                /* A Reno DUPACK means new data in F-RTO step 2.b above are
2720                 * delivered. Lower inflight to clock out (re)tranmissions.
2721                 */
2722                if (after(tp->snd_nxt, tp->high_seq) && is_dupack)
2723                        tcp_add_reno_sack(sk);
2724                else if (flag & FLAG_SND_UNA_ADVANCED)
2725                        tcp_reset_reno_sack(tp);
2726        }
2727        if (tcp_try_undo_loss(sk, false))
2728                return;
2729        tcp_xmit_retransmit_queue(sk);
2730 }
2731
2732 /* Undo during fast recovery after partial ACK. */
2733 static bool tcp_try_undo_partial(struct sock *sk, const int acked,
2734                                  const int prior_unsacked)
2735 {
2736        struct tcp_sock *tp = tcp_sk(sk);
2737
2738        if (tp->undo_marker && tcp_packet_delayed(tp)) {
2739                /* Plain luck! Hole if filled with delayed
2740                 * packet, rather than with a retransmit.
2741                 */
2742                tcp_update_reordering(sk, tcp_fackets_out(tp) + acked, 1);
2743
2744                /* We are getting evidence that the reordering degree is higher
2745                 * than we realized. If there are no retransmits out then we
2746                 * can undo. Otherwise we clock out new packets but do not
2747                 * mark more packets lost or retransmit more.
2748                 */
2749                if (tp->retrans_out) {
2750                        tcp_cwnd_reduction(sk, prior_unsacked, 0);
2751                        return true;
2752                }
2753
2754                if (!tcp_any_retrans_done(sk))
2755                        tp->retrans_stamp = 0;
2756
2757                DBGUNDO(sk, "partial recovery");
2758                tcp_undo_cwnd_reduction(sk, true);
2759                NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPPARTIALUNDO);
2760                tcp_try_keep_open(sk);
2761                return true;
2762        }
2763        return false;
2764 }
2765
2766 /* Process an event, which can update packets-in-flight not trivially.
2767  * Main goal of this function is to calculate new estimate for left_out,
2768  * taking into account both packets sitting in receiver's buffer and
2769  * packets lost by network.
2770  *
2771  * Besides that it does CWND reduction, when packet loss is detected
2772  * and changes state of machine.
2773  *
2774  * It does _not_ decide what to send, it is made in function
2775  * tcp_xmit_retransmit_queue().
2776  */
2777 static void tcp_fastretrans_alert(struct sock *sk, const int acked,
2778                                   const int prior_unsacked,
2779                                   bool is_dupack, int flag)
2780 {
2781        struct inet_connection_sock *icsk = inet_csk(sk);
2782        struct tcp_sock *tp = tcp_sk(sk);
2783        bool do_lost = is_dupack || ((flag & FLAG_DATA_SACKED) &&
2784                                     (tcp_fackets_out(tp) > tp->reordering));
2785        int fast_rexmit = 0;
2786
2787        if (WARN_ON(!tp->packets_out && tp->sacked_out))
2788                tp->sacked_out = 0;
2789        if (WARN_ON(!tp->sacked_out && tp->fackets_out))
2790                tp->fackets_out = 0;
2791
2792        /* Now state machine starts.
2793         * A. ECE, hence prohibit cwnd undoing, the reduction is required. */
2794        if (flag & FLAG_ECE)
2795                tp->prior_ssthresh = 0;
2796
```

```
2797        /* B. In all the states check for reneging SACKs. */
2798        if (tcp_check_sack_reneging(sk, flag))
2799                return;
2800
2801        /* C. Check consistency of the current state. */
2802        tcp_verify_left_out(tp);
2803
2804        /* D. Check state exit conditions. State can be terminated
2805         *    when high_seq is ACKed. */
2806        if (icsk->icsk_ca_state == TCP_CA_Open) {
2807                WARN_ON(tp->retrans_out != 0);
2808                tp->retrans_stamp = 0;
2809        } else if (!before(tp->snd_una, tp->high_seq)) {
2810                switch (icsk->icsk_ca_state) {
2811                case TCP_CA_CWR:
2812                        /* CWR is to be held something *above* high_seq
2813                         * is ACKed for CWR bit to reach receiver. */
2814                        if (tp->snd_una != tp->high_seq) {
2815                                tcp_end_cwnd_reduction(sk);
2816                                tcp_set_ca_state(sk, TCP_CA_Open);
2817                        }
2818                        break;
2819
2820                case TCP_CA_Recovery:
2821                        if (tcp_is_reno(tp))
2822                                tcp_reset_reno_sack(tp);
2823                        if (tcp_try_undo_recovery(sk))
2824                                return;
2825                        tcp_end_cwnd_reduction(sk);
2826                        break;
2827                }
2828        }
2829
2830        /* E. Process state. */
2831        switch (icsk->icsk_ca_state) {
2832        case TCP_CA_Recovery:
2833                if (!(flag & FLAG_SND_UNA_ADVANCED)) {
2834                        if (tcp_is_reno(tp) && is_dupack)
2835                                tcp_add_reno_sack(sk);
2836                } else {
2837                        if (tcp_try_undo_partial(sk, acked, prior_unsacked))
2838                                return;
2839                        /* Partial ACK arrived. Force fast retransmit. */
2840                        do_lost = tcp_is_reno(tp) ||
2841                                  tcp_fackets_out(tp) > tp->reordering;
2842                }
2843                if (tcp_try_undo_dsack(sk)) {
2844                        tcp_try_keep_open(sk);
2845                        return;
2846                }
2847                break;
2848        case TCP_CA_Loss:
2849                tcp_process_loss(sk, flag, is_dupack);
2850                if (icsk->icsk_ca_state != TCP_CA_Open)
2851                        return;
2852                /* Fall through to processing in Open state. */
2853        default:
2854                if (tcp_is_reno(tp)) {
2855                        if (flag & FLAG_SND_UNA_ADVANCED)
2856                                tcp_reset_reno_sack(tp);
2857                        if (is_dupack)
2858                                tcp_add_reno_sack(sk);
2859                }
2860
2861                if (icsk->icsk_ca_state <= TCP_CA_Disorder)
2862                        tcp_try_undo_dsack(sk);
2863
2864                if (!tcp_time_to_recover(sk, flag)) {
2865                        tcp_try_to_open(sk, flag, prior_unsacked);
2866                        return;
2867                }
2868
2869                /* MTU probe failure: don't reduce cwnd */
2870                if (icsk->icsk_ca_state < TCP_CA_CWR &&
2871                    icsk->icsk_mtup.probe_size &&
2872                    tp->snd_una == tp->mtu_probe.probe_seq_start) {
2873                        tcp_mtup_probe_failed(sk);
2874                        /* Restores the reduction we did in tcp_mtup_probe() */
2875                        tp->snd_cwnd++;
2876                        tcp_simple_retransmit(sk);
2877                        return;
2878                }
2879
2880                /* Otherwise enter Recovery state */
2881                tcp_enter_recovery(sk, (flag & FLAG_ECE));
2882                fast_rexmit = 1;
2883        }
2884
```

```
2885         if (do_lost)
2886                 tcp_update_scoreboard(sk, fast_rexmit);
2887         tcp_cwnd_reduction(sk, prior_unsacked, fast_rexmit);
2888         tcp_xmit_retransmit_queue(sk);
2889 }
2890
2891 static inline bool tcp_ack_update_rtt(struct sock *sk, const int flag,
2892                                       long seq_rtt_us, long sack_rtt_us)
2893 {
2894         const struct tcp_sock *tp = tcp_sk(sk);
2895
2896         /* Prefer RTT measured from ACK's timing to TS-ECR. This is because
2897          * broken middle-boxes or peers may corrupt TS-ECR fields. But
2898          * Karn's algorithm forbids taking RTT if some retransmitted data
2899          * is acked (RFC6298).
2900          */
2901         if (flag & FLAG_RETRANS_DATA_ACKED)
2902                 seq_rtt_us = -1L;
2903
2904         if (seq_rtt_us < 0)
2905                 seq_rtt_us = sack_rtt_us;
2906
2907         /* RTTM Rule: A TSecr value received in a segment is used to
2908          * update the averaged RTT measurement only if the segment
2909          * acknowledges some new data, i.e., only if it advances the
2910          * left edge of the send window.
2911          * See draft-ietf-tcplw-high-performance-00, section 3.3.
2912          */
2913         if (seq_rtt_us < 0 && tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr &&
2914             flag & FLAG_ACKED)
2915                 seq_rtt_us = jiffies_to_usecs(tcp_time_stamp - tp->rx_opt.rcv_tsecr);
2916
2917         if (seq_rtt_us < 0)
2918                 return false;
2919
2920         tcp_rtt_estimator(sk, seq_rtt_us);
2921         tcp_set_rto(sk);
2922
2923         /* RFC6298: only reset backoff on valid RTT measurement. */
2924         inet_csk(sk)->icsk_backoff = 0;
2925         return true;
2926 }
2927
2928 /* Compute time elapsed between (last) SYNACK and the ACK completing 3WHS. */
2929 static void tcp_synack_rtt_meas(struct sock *sk, const u32 synack_stamp)
2930 {
2931         struct tcp_sock *tp = tcp_sk(sk);
2932         long seq_rtt_us = -1L;
2933
2934         if (synack_stamp && !tp->total_retrans)
2935                 seq_rtt_us = jiffies_to_usecs(tcp_time_stamp - synack_stamp);
2936
2937         /* If the ACK acks both the SYNACK and the (Fast Open'd) data packets
2938          * sent in SYN_RECV, SYNACK RTT is the smooth RTT computed in tcp_ack()
2939          */
2940         if (!tp->srtt_us)
2941                 tcp_ack_update_rtt(sk, FLAG_SYN_ACKED, seq_rtt_us, -1L);
2942 }
2943
2944 static void tcp_cong_avoid(struct sock *sk, u32 ack, u32 acked)
2945 {
2946         const struct inet_connection_sock *icsk = inet_csk(sk);
2947
2948         icsk->icsk_ca_ops->cong_avoid(sk, ack, acked);
2949         tcp_sk(sk)->snd_cwnd_stamp = tcp_time_stamp;
2950 }
2951
2952 /* Restart timer after forward progress on connection.
2953  * RFC2988 recommends to restart timer to now+rto.
2954  */
2955 void tcp_rearm_rto(struct sock *sk)
2956 {
2957         const struct inet_connection_sock *icsk = inet_csk(sk);
2958         struct tcp_sock *tp = tcp_sk(sk);
2959
2960         /* If the retrans timer is currently being used by Fast Open
2961          * for SYN-ACK retrans purpose, stay put.
2962          */
2963         if (tp->fastopen_rsk)
2964                 return;
2965
2966         if (!tp->packets_out) {
2967                 inet_csk_clear_xmit_timer(sk, ICSK_TIME_RETRANS);
2968         } else {
2969                 u32 rto = inet_csk(sk)->icsk_rto;
2970                 /* Offset the time elapsed after installing regular RTO */
2971                 if (icsk->icsk_pending == ICSK_TIME_EARLY_RETRANS ||
2972                     icsk->icsk_pending == ICSK_TIME_LOSS_PROBE) {
```

```
2973                        struct sk_buff *skb = tcp_write_queue_head(sk);
2974                        const u32 rto_time_stamp = TCP_SKB_CB(skb)->when + rto;
2975                        s32 delta = (s32)(rto_time_stamp - tcp_time_stamp);
2976                        /* delta may not be positive if the socket is locked
2977                         * when the retrans timer fires and is rescheduled.
2978                         */
2979                        if (delta > 0)
2980                                rto = delta;
2981                }
2982                inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS, rto,
2983                                          TCP_RTO_MAX);
2984        }
2985 }
2986
2987 /* This function is called when the delayed ER timer fires. TCP enters
2988  * fast recovery and performs fast-retransmit.
2989  */
2990 void tcp_resume_early_retransmit(struct sock *sk)
2991 {
2992        struct tcp_sock *tp = tcp_sk(sk);
2993
2994        tcp_rearm_rto(sk);
2995
2996        /* Stop if ER is disabled after the delayed ER timer is scheduled */
2997        if (!tp->do_early_retrans)
2998                return;
2999
3000        tcp_enter_recovery(sk, false);
3001        tcp_update_scoreboard(sk, 1);
3002        tcp_xmit_retransmit_queue(sk);
3003 }
3004
3005 /* If we get here, the whole TSO packet has not been acked. */
3006 static u32 tcp_tso_acked(struct sock *sk, struct sk_buff *skb)
3007 {
3008        struct tcp_sock *tp = tcp_sk(sk);
3009        u32 packets_acked;
3010
3011        BUG_ON(!after(TCP_SKB_CB(skb)->end_seq, tp->snd_una));
3012
3013        packets_acked = tcp_skb_pcount(skb);
3014        if (tcp_trim_head(sk, skb, tp->snd_una - TCP_SKB_CB(skb)->seq))
3015                return 0;
3016        packets_acked -= tcp_skb_pcount(skb);
3017
3018        if (packets_acked) {
3019                BUG_ON(tcp_skb_pcount(skb) == 0);
3020                BUG_ON(!before(TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq));
3021        }
3022
3023        return packets_acked;
3024 }
3025
3026 /* Remove acknowledged frames from the retransmission queue. If our packet
3027  * is before the ack sequence we can discard it as it's confirmed to have
3028  * arrived at the other end.
3029  */
3030 static int tcp_clean_rtx_queue(struct sock *sk, int prior_fackets,
3031                                u32 prior_snd_una, long sack_rtt_us)
3032 {
3033        const struct inet_connection_sock *icsk = inet_csk(sk);
3034        struct skb_mstamp first_ackt, last_ackt, now;
3035        struct tcp_sock *tp = tcp_sk(sk);
3036        u32 prior_sacked = tp->sacked_out;
3037        u32 reord = tp->packets_out;
3038        bool fully_acked = true;
3039        long ca_seq_rtt_us = -1L;
3040        long seq_rtt_us = -1L;
3041        struct sk_buff *skb;
3042        u32 pkts_acked = 0;
3043        bool rtt_update;
3044        int flag = 0;
3045
3046        first_ackt.v64 = 0;
3047
3048        while ((skb = tcp_write_queue_head(sk)) && skb != tcp_send_head(sk)) {
3049                struct skb_shared_info *shinfo = skb_shinfo(skb);
3050                struct tcp_skb_cb *scb = TCP_SKB_CB(skb);
3051                u8 sacked = scb->sacked;
3052                u32 acked_pcount;
3053
3054                if (unlikely(shinfo->tx_flags & SKBTX_ACK_TSTAMP) &&
3055                    between(shinfo->tskey, prior_snd_una, tp->snd_una - 1))
3056                        __skb_tstamp_tx(skb, NULL, sk, SCM_TSTAMP_ACK);
3057
3058                /* Determine how many packets and what bytes were acked, tso and else */
3059                if (after(scb->end_seq, tp->snd_una)) {
3060                        if (tcp_skb_pcount(skb) == 1 ||
```

```
3061                            !after(tp->snd_una, scb->seq))
3062                                break;
3063
3064                        acked_pcount = tcp_tso_acked(sk, skb);
3065                        if (!acked_pcount)
3066                                break;
3067
3068                        fully_acked = false;
3069                } else {
3070                        acked_pcount = tcp_skb_pcount(skb);
3071                }
3072
3073                if (sacked & TCPCB_RETRANS) {
3074                        if (sacked & TCPCB_SACKED_RETRANS)
3075                                tp->retrans_out -= acked_pcount;
3076                        flag |= FLAG_RETRANS_DATA_ACKED;
3077                } else {
3078                        last_ackt = skb->skb_mstamp;
3079                        WARN_ON_ONCE(last_ackt.v64 == 0);
3080                        if (!first_ackt.v64)
3081                                first_ackt = last_ackt;
3082
3083                        if (!(sacked & TCPCB_SACKED_ACKED))
3084                                reord = min(pkts_acked, reord);
3085                        if (!after(scb->end_seq, tp->high_seq))
3086                                flag |= FLAG_ORIG_SACK_ACKED;
3087                }
3088
3089                if (sacked & TCPCB_SACKED_ACKED)
3090                        tp->sacked_out -= acked_pcount;
3091                if (sacked & TCPCB_LOST)
3092                        tp->lost_out -= acked_pcount;
3093
3094                tp->packets_out -= acked_pcount;
3095                pkts_acked += acked_pcount;
3096
3097                /* Initial outgoing SYN's get put onto the write_queue
3098                 * just like anything else we transmit.  It is not
3099                 * true data, and if we misinform our callers that
3100                 * this ACK acks real data, we will erroneously exit
3101                 * connection startup slow start one packet too
3102                 * quickly.  This is severely frowned upon behavior.
3103                 */
3104                if (!(scb->tcp_flags & TCPHDR_SYN)) {
3105                        flag |= FLAG_DATA_ACKED;
3106                } else {
3107                        flag |= FLAG_SYN_ACKED;
3108                        tp->retrans_stamp = 0;
3109                }
3110
3111                if (!fully_acked)
3112                        break;
3113
3114                tcp_unlink_write_queue(skb, sk);
3115                sk_wmem_free_skb(sk, skb);
3116                if (skb == tp->retransmit_skb_hint)
3117                        tp->retransmit_skb_hint = NULL;
3118                if (skb == tp->lost_skb_hint)
3119                        tp->lost_skb_hint = NULL;
3120        }
3121
3122        if (likely(between(tp->snd_up, prior_snd_una, tp->snd_una)))
3123                tp->snd_up = tp->snd_una;
3124
3125        if (skb && (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED))
3126                flag |= FLAG_SACK_RENEGING;
3127
3128        skb_mstamp_get(&now);
3129        if (first_ackt.v64) {
3130                seq_rtt_us = skb_mstamp_us_delta(&now, &first_ackt);
3131                ca_seq_rtt_us = skb_mstamp_us_delta(&now, &last_ackt);
3132        }
3133
3134        rtt_update = tcp_ack_update_rtt(sk, flag, seq_rtt_us, sack_rtt_us);
3135
3136        if (flag & FLAG_ACKED) {
3137                const struct tcp_congestion_ops *ca_ops
3138                        = inet_csk(sk)->icsk_ca_ops;
3139
3140                tcp_rearm_rto(sk);
3141                if (unlikely(icsk->icsk_mtup.probe_size &&
3142                             !after(tp->mtu_probe.probe_seq_end, tp->snd_una))) {
3143                        tcp_mtup_probe_success(sk);
3144                }
3145
3146                if (tcp_is_reno(tp)) {
3147                        tcp_remove_reno_sacks(sk, pkts_acked);
3148                } else {
```

```
3149                        int delta;
3150
3151                        /* Non-retransmitted hole got filled? That's reordering */
3152                        if (reord < prior_fackets)
3153                                tcp_update_reordering(sk, tp->fackets_out - reord, 0);
3154
3155                        delta = tcp_is_fack(tp) ? pkts_acked :
3156                                                  prior_sacked - tp->sacked_out;
3157                        tp->lost_cnt_hint -= min(tp->lost_cnt_hint, delta);
3158                }
3159
3160                tp->fackets_out -= min(pkts_acked, tp->fackets_out);
3161
3162                if (ca_ops->pkts_acked)
3163                        ca_ops->pkts_acked(sk, pkts_acked, ca_seq_rtt_us);
3164
3165        } else if (skb && rtt_update && sack_rtt_us >= 0 &&
3166                   sack_rtt_us > skb_mstamp_us_delta(&now, &skb->skb_mstamp)) {
3167                /* Do not re-arm RTO if the sack RTT is measured from data sent
3168                 * after when the head was last (re)transmitted. Otherwise the
3169                 * timeout may continue to extend in loss recovery.
3170                 */
3171                tcp_rearm_rto(sk);
3172        }
3173
3174 #if FASTRETRANS_DEBUG > 0
3175        WARN_ON((int)tp->sacked_out < 0);
3176        WARN_ON((int)tp->lost_out < 0);
3177        WARN_ON((int)tp->retrans_out < 0);
3178        if (!tp->packets_out && tcp_is_sack(tp)) {
3179                icsk = inet_csk(sk);
3180                if (tp->lost_out) {
3181                        pr_debug("Leak l=%u %d\n",
3182                                 tp->lost_out, icsk->icsk_ca_state);
3183                        tp->lost_out = 0;
3184                }
3185                if (tp->sacked_out) {
3186                        pr_debug("Leak s=%u %d\n",
3187                                 tp->sacked_out, icsk->icsk_ca_state);
3188                        tp->sacked_out = 0;
3189                }
3190                if (tp->retrans_out) {
3191                        pr_debug("Leak r=%u %d\n",
3192                                 tp->retrans_out, icsk->icsk_ca_state);
3193                        tp->retrans_out = 0;
3194                }
3195        }
3196 #endif
3197        return flag;
3198 }
3199
3200 static void tcp_ack_probe(struct sock *sk)
3201 {
3202        const struct tcp_sock *tp = tcp_sk(sk);
3203        struct inet_connection_sock *icsk = inet_csk(sk);
3204
3205        /* Was it a usable window open? */
3206
3207        if (!after(TCP_SKB_CB(tcp_send_head(sk))->end_seq, tcp_wnd_end(tp))) {
3208                icsk->icsk_backoff = 0;
3209                inet_csk_clear_xmit_timer(sk, ICSK_TIME_PROBE0);
3210                /* Socket must be waked up by subsequent tcp_data_snd_check().
3211                 * This function is not for random using!
3212                 */
3213        } else {
3214                inet_csk_reset_xmit_timer(sk, ICSK_TIME_PROBE0,
3215                                          min(icsk->icsk_rto << icsk->icsk_backoff, TCP_RTO_MAX),
3216                                          TCP_RTO_MAX);
3217        }
3218 }
3219
3220 static inline bool tcp_ack_is_dubious(const struct sock *sk, const int flag)
3221 {
3222        return !(flag & FLAG_NOT_DUP) || (flag & FLAG_CA_ALERT) ||
3223                inet_csk(sk)->icsk_ca_state != TCP_CA_Open;
3224 }
3225
3226 /* Decide wheather to run the increase function of congestion control. */
3227 static inline bool tcp_may_raise_cwnd(const struct sock *sk, const int flag)
3228 {
3229        if (tcp_in_cwnd_reduction(sk))
3230                return false;
3231
3232        /* If reordering is high then always grow cwnd whenever data is
3233         * delivered regardless of its ordering. Otherwise stay conservative
3234         * and only grow cwnd on in-order delivery (RFC5681). A stretched ACK w/
3235         * new SACK or ECE mark may first advance cwnd here and later reduce
3236         * cwnd in tcp_fastretrans_alert() based on more states.
```

```
3237          */
3238         if (tcp_sk(sk)->reordering > sysctl_tcp_reordering)
3239                 return flag & FLAG_FORWARD_PROGRESS;
3240
3241         return flag & FLAG_DATA_ACKED;
3242 }
3243
3244 /* Check that window update is acceptable.
3245  * The function assumes that snd_una<=ack<=snd_next.
3246  */
3247 static inline bool tcp_may_update_window(const struct tcp_sock *tp,
3248                                          const u32 ack, const u32 ack_seq,
3249                                          const u32 nwin)
3250 {
3251         return  after(ack, tp->snd_una) ||
3252                 after(ack_seq, tp->snd_wl1) ||
3253                 (ack_seq == tp->snd_wl1 && nwin > tp->snd_wnd);
3254 }
3255
3256 /* Update our send window.
3257  *
3258  * Window update algorithm, described in RFC793/RFC1122 (used in linux-2.2
3259  * and in FreeBSD. NetBSD's one is even worse.) is wrong.
3260  */
3261 static int tcp_ack_update_window(struct sock *sk, const struct sk_buff *skb, u32 ack,
3262                                  u32 ack_seq)
3263 {
3264         struct tcp_sock *tp = tcp_sk(sk);
3265         int flag = 0;
3266         u32 nwin = ntohs(tcp_hdr(skb)->window);
3267
3268         if (likely(!tcp_hdr(skb)->syn))
3269                 nwin <<= tp->rx_opt.snd_wscale;
3270
3271         if (tcp_may_update_window(tp, ack, ack_seq, nwin)) {
3272                 flag |= FLAG_WIN_UPDATE;
3273                 tcp_update_wl(tp, ack_seq);
3274
3275                 if (tp->snd_wnd != nwin) {
3276                         tp->snd_wnd = nwin;
3277
3278                         /* Note, it is the only place, where
3279                          * fast path is recovered for sending TCP.
3280                          */
3281                         tp->pred_flags = 0;
3282                         tcp_fast_path_check(sk);
3283
3284                         if (nwin > tp->max_window) {
3285                                 tp->max_window = nwin;
3286                                 tcp_sync_mss(sk, inet_csk(sk)->icsk_pmtu_cookie);
3287                         }
3288                 }
3289         }
3290
3291         tp->snd_una = ack;
3292
3293         return flag;
3294 }
3295
3296 /* RFC 5961 7 [ACK Throttling] */
3297 static void tcp_send_challenge_ack(struct sock *sk)
3298 {
3299         /* unprotected vars, we dont care of overwrites */
3300         static u32 challenge_timestamp;
3301         static unsigned int challenge_count;
3302         u32 now = jiffies / HZ;
3303
3304         if (now != challenge_timestamp) {
3305                 challenge_timestamp = now;
3306                 challenge_count = 0;
3307         }
3308         if (++challenge_count <= sysctl_tcp_challenge_ack_limit) {
3309                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPCHALLENGEACK);
3310                 tcp_send_ack(sk);
3311         }
3312 }
3313
3314 static void tcp_store_ts_recent(struct tcp_sock *tp)
3315 {
3316         tp->rx_opt.ts_recent = tp->rx_opt.rcv_tsval;
3317         tp->rx_opt.ts_recent_stamp = get_seconds();
3318 }
3319
3320 static void tcp_replace_ts_recent(struct tcp_sock *tp, u32 seq)
3321 {
3322         if (tp->rx_opt.saw_tstamp && !after(seq, tp->rcv_wup)) {
3323                 /* PAWS bug workaround wrt. ACK frames, the PAWS discard
3324                  * extra check below makes sure this can only happen
```

```
3325                 * for pure ACK frames.  -DaveM
3326                 *
3327                 * Not only, also it occurs for expired timestamps.
3328                 */
3329
3330                if (tcp_paws_check(&tp->rx_opt, 0))
3331                        tcp_store_ts_recent(tp);
3332        }
3333 }
3334
3335 /* This routine deals with acks during a TLP episode.
3336  * Ref: loss detection algorithm in draft-dukkipati-tcpm-tcp-loss-probe.
3337  */
3338 static void tcp_process_tlp_ack(struct sock *sk, u32 ack, int flag)
3339 {
3340        struct tcp_sock *tp = tcp_sk(sk);
3341        bool is_tlp_dupack = (ack == tp->tlp_high_seq) &&
3342                             !(flag & (FLAG_SND_UNA_ADVANCED |
3343                                       FLAG_NOT_DUP | FLAG_DATA_SACKED));
3344
3345        /* Mark the end of TLP episode on receiving TLP dupack or when
3346         * ack is after tlp_high_seq.
3347         */
3348        if (is_tlp_dupack) {
3349                tp->tlp_high_seq = 0;
3350                return;
3351        }
3352
3353        if (after(ack, tp->tlp_high_seq)) {
3354                tp->tlp_high_seq = 0;
3355                /* Don't reduce cwnd if DSACK arrives for TLP retrans. */
3356                if (!(flag & FLAG_DSACKING_ACK)) {
3357                        tcp_init_cwnd_reduction(sk);
3358                        tcp_set_ca_state(sk, TCP_CA_CWR);
3359                        tcp_end_cwnd_reduction(sk);
3360                        tcp_try_keep_open(sk);
3361                        NET_INC_STATS_BH(sock_net(sk),
3362                                         LINUX_MIB_TCPLOSSPROBERECOVERY);
3363                }
3364        }
3365 }
3366
3367 /* This routine deals with incoming acks, but not outgoing ones. */
3368 static int tcp_ack(struct sock *sk, const struct sk_buff *skb, int flag)
3369 {
3370        struct inet_connection_sock *icsk = inet_csk(sk);
3371        struct tcp_sock *tp = tcp_sk(sk);
3372        u32 prior_snd_una = tp->snd_una;
3373        u32 ack_seq = TCP_SKB_CB(skb)->seq;
3374        u32 ack = TCP_SKB_CB(skb)->ack_seq;
3375        bool is_dupack = false;
3376        u32 prior_fackets;
3377        int prior_packets = tp->packets_out;
3378        const int prior_unsacked = tp->packets_out - tp->sacked_out;
3379        int acked = 0; /* Number of packets newly acked */
3380        long sack_rtt_us = -1L;
3381
3382        /* If the ack is older than previous acks
3383         * then we can probably ignore it.
3384         */
3385        if (before(ack, prior_snd_una)) {
3386                /* RFC 5961 5.2 [Blind Data Injection Attack].[Mitigation] */
3387                if (before(ack, prior_snd_una - tp->max_window)) {
3388                        tcp_send_challenge_ack(sk);
3389                        return -1;
3390                }
3391                goto old_ack;
3392        }
3393
3394        /* If the ack includes data we haven't sent yet, discard
3395         * this segment (RFC793 Section 3.9).
3396         */
3397        if (after(ack, tp->snd_nxt))
3398                goto invalid_ack;
3399
3400        if (icsk->icsk_pending == ICSK_TIME_EARLY_RETRANS ||
3401            icsk->icsk_pending == ICSK_TIME_LOSS_PROBE)
3402                tcp_rearm_rto(sk);
3403
3404        if (after(ack, prior_snd_una)) {
3405                flag |= FLAG_SND_UNA_ADVANCED;
3406                icsk->icsk_retransmits = 0;
3407        }
3408
3409        prior_fackets = tp->fackets_out;
3410
3411        /* ts_recent update must be made after we are sure that the packet
3412         * is in window.
```

```
3413          */
3414         if (flag & FLAG_UPDATE_TS_RECENT)
3415                 tcp_replace_ts_recent(tp, TCP_SKB_CB(skb)->seq);
3416
3417         if (!(flag & FLAG_SLOWPATH) && after(ack, prior_snd_una)) {
3418                 /* Window is constant, pure forward advance.
3419                  * No more checks are required.
3420                  * Note, we use the fact that SND.UNA>=SND.WL2.
3421                  */
3422                 tcp_update_wl(tp, ack_seq);
3423                 tp->snd_una = ack;
3424                 flag |= FLAG_WIN_UPDATE;
3425
3426                 tcp_ca_event(sk, CA_EVENT_FAST_ACK);
3427
3428                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPHPACKS);
3429         } else {
3430                 if (ack_seq != TCP_SKB_CB(skb)->end_seq)
3431                         flag |= FLAG_DATA;
3432                 else
3433                         NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPPUREACKS);
3434
3435                 flag |= tcp_ack_update_window(sk, skb, ack, ack_seq);
3436
3437                 if (TCP_SKB_CB(skb)->sacked)
3438                         flag |= tcp_sacktag_write_queue(sk, skb, prior_snd_una,
3439                                                         &sack_rtt_us);
3440
3441                 if (TCP_ECN_rcv_ecn_echo(tp, tcp_hdr(skb)))
3442                         flag |= FLAG_ECE;
3443
3444                 tcp_ca_event(sk, CA_EVENT_SLOW_ACK);
3445         }
3446
3447         /* We passed data and got it acked, remove any soft error
3448          * log. Something worked...
3449          */
3450         sk->sk_err_soft = 0;
3451         icsk->icsk_probes_out = 0;
3452         tp->rcv_tstamp = tcp_time_stamp;
3453         if (!prior_packets)
3454                 goto no_queue;
3455
3456         /* See if we can take anything off of the retransmit queue. */
3457         acked = tp->packets_out;
3458         flag |= tcp_clean_rtx_queue(sk, prior_fackets, prior_snd_una,
3459                                     sack_rtt_us);
3460         acked -= tp->packets_out;
3461
3462         /* Advance cwnd if state allows */
3463         if (tcp_may_raise_cwnd(sk, flag))
3464                 tcp_cong_avoid(sk, ack, acked);
3465
3466         if (tcp_ack_is_dubious(sk, flag)) {
3467                 is_dupack = !(flag & (FLAG_SND_UNA_ADVANCED | FLAG_NOT_DUP));
3468                 tcp_fastretrans_alert(sk, acked, prior_unsacked,
3469                                       is_dupack, flag);
3470         }
3471         if (tp->tlp_high_seq)
3472                 tcp_process_tlp_ack(sk, ack, flag);
3473
3474         if ((flag & FLAG_FORWARD_PROGRESS) || !(flag & FLAG_NOT_DUP)) {
3475                 struct dst_entry *dst = __sk_dst_get(sk);
3476                 if (dst)
3477                         dst_confirm(dst);
3478         }
3479
3480         if (icsk->icsk_pending == ICSK_TIME_RETRANS)
3481                 tcp_schedule_loss_probe(sk);
3482         tcp_update_pacing_rate(sk);
3483         return 1;
3484
3485 no_queue:
3486         /* If data was DSACKed, see if we can undo a cwnd reduction. */
3487         if (flag & FLAG_DSACKING_ACK)
3488                 tcp_fastretrans_alert(sk, acked, prior_unsacked,
3489                                       is_dupack, flag);
3490         /* If this ack opens up a zero window, clear backoff.  It was
3491          * being used to time the probes, and is probably far higher than
3492          * it needs to be for normal retransmission.
3493          */
3494         if (tcp_send_head(sk))
3495                 tcp_ack_probe(sk);
3496
3497         if (tp->tlp_high_seq)
3498                 tcp_process_tlp_ack(sk, ack, flag);
3499         return 1;
3500
```

```
3501 invalid_ack:
3502         SOCK_DEBUG(sk, "Ack %u after %u:%u\n", ack, tp->snd_una, tp->snd_nxt);
3503         return -1;
3504
3505 old_ack:
3506         /* If data was SACKed, tag it and see if we should send more data.
3507          * If data was DSACKed, see if we can undo a cwnd reduction.
3508          */
3509         if (TCP_SKB_CB(skb)->sacked) {
3510                 flag |= tcp_sacktag_write_queue(sk, skb, prior_snd_una,
3511                                                 &sack_rtt_us);
3512                 tcp_fastretrans_alert(sk, acked, prior_unsacked,
3513                                       is_dupack, flag);
3514         }
3515
3516         SOCK_DEBUG(sk, "Ack %u before %u:%u\n", ack, tp->snd_una, tp->snd_nxt);
3517         return 0;
3518 }
3519
3520 /* Look for tcp options. Normally only called on SYN and SYNACK packets.
3521  * But, this can also be called on packets in the established flow when
3522  * the fast version below fails.
3523  */
3524 void tcp_parse_options(const struct sk_buff *skb,
3525                        struct tcp_options_received *opt_rx, int estab,
3526                        struct tcp_fastopen_cookie *foc)
3527 {
3528         const unsigned char *ptr;
3529         const struct tcphdr *th = tcp_hdr(skb);
3530         int length = (th->doff * 4) - sizeof(struct tcphdr);
3531
3532         ptr = (const unsigned char *)(th + 1);
3533         opt_rx->saw_tstamp = 0;
3534
3535         while (length > 0) {
3536                 int opcode = *ptr++;
3537                 int opsize;
3538
3539                 switch (opcode) {
3540                 case TCPOPT_EOL:
3541                         return;
3542                 case TCPOPT_NOP:        /* Ref: RFC 793 section 3.1 */
3543                         length--;
3544                         continue;
3545                 default:
3546                         opsize = *ptr++;
3547                         if (opsize < 2) /* "silly options" */
3548                                 return;
3549                         if (opsize > length)
3550                                 return; /* don't parse partial options */
3551                         switch (opcode) {
3552                         case TCPOPT_MSS:
3553                                 if (opsize == TCPOLEN_MSS && th->syn && !estab) {
3554                                         u16 in_mss = get_unaligned_be16(ptr);
3555                                         if (in_mss) {
3556                                                 if (opt_rx->user_mss &&
3557                                                     opt_rx->user_mss < in_mss)
3558                                                         in_mss = opt_rx->user_mss;
3559                                                 opt_rx->mss_clamp = in_mss;
3560                                         }
3561                                 }
3562                                 break;
3563                         case TCPOPT_WINDOW:
3564                                 if (opsize == TCPOLEN_WINDOW && th->syn &&
3565                                     !estab && sysctl_tcp_window_scaling) {
3566                                         __u8 snd_wscale = *(__u8 *)ptr;
3567                                         opt_rx->wscale_ok = 1;
3568                                         if (snd_wscale > 14) {
3569                                                 net_info_ratelimited("%s: Illegal window scaling value %d >14 received\n",
3570                                                                      __func__,
3571                                                                      snd_wscale);
3572                                                 snd_wscale = 14;
3573                                         }
3574                                         opt_rx->snd_wscale = snd_wscale;
3575                                 }
3576                                 break;
3577                         case TCPOPT_TIMESTAMP:
3578                                 if ((opsize == TCPOLEN_TIMESTAMP) &&
3579                                     ((estab && opt_rx->tstamp_ok) ||
3580                                      (!estab && sysctl_tcp_timestamps))) {
3581                                         opt_rx->saw_tstamp = 1;
3582                                         opt_rx->rcv_tsval = get_unaligned_be32(ptr);
3583                                         opt_rx->rcv_tsecr = get_unaligned_be32(ptr + 4);
3584                                 }
3585                                 break;
3586                         case TCPOPT_SACK_PERM:
3587                                 if (opsize == TCPOLEN_SACK_PERM && th->syn &&
3588                                     !estab && sysctl_tcp_sack) {
```

```
3589                                    opt_rx->sack_ok = TCP_SACK_SEEN;
3590                                    tcp_sack_reset(opt_rx);
3591                            }
3592                            break;
3593
3594                    case TCPOPT_SACK:
3595                            if ((opsize >= (TCPOLEN_SACK_BASE + TCPOLEN_SACK_PERBLOCK)) &&
3596                               !((opsize - TCPOLEN_SACK_BASE) % TCPOLEN_SACK_PERBLOCK) &&
3597                               opt_rx->sack_ok) {
3598                                    TCP_SKB_CB(skb)->sacked = (ptr - 2) - (unsigned char *)th;
3599                            }
3600                            break;
3601 #ifdef CONFIG_TCP_MD5SIG
3602                    case TCPOPT_MD5SIG:
3603                            /*
3604                             * The MD5 Hash has already been
3605                             * checked (see tcp_v{4,6}_do_rcv()).
3606                             */
3607                            break;
3608 #endif
3609                    case TCPOPT_EXP:
3610                            /* Fast Open option shares code 254 using a
3611                             * 16 bits magic number. It's valid only in
3612                             * SYN or SYN-ACK with an even size.
3613                             */
3614                            if (opsize < TCPOLEN_EXP_FASTOPEN_BASE ||
3615                                get_unaligned_be16(ptr) != TCPOPT_FASTOPEN_MAGIC ||
3616                                foc == NULL || !th->syn || (opsize & 1))
3617                                    break;
3618                            foc->len = opsize - TCPOLEN_EXP_FASTOPEN_BASE;
3619                            if (foc->len >= TCP_FASTOPEN_COOKIE_MIN &&
3620                                foc->len <= TCP_FASTOPEN_COOKIE_MAX)
3621                                    memcpy(foc->val, ptr + 2, foc->len);
3622                            else if (foc->len != 0)
3623                                    foc->len = -1;
3624                            break;
3625
3626                    }
3627                    ptr += opsize-2;
3628                    length -= opsize;
3629            }
3630    }
3631 }
3632 EXPORT_SYMBOL(tcp_parse_options);
3633
3634 static bool tcp_parse_aligned_timestamp(struct tcp_sock *tp, const struct tcphdr *th)
3635 {
3636         const __be32 *ptr = (const __be32 *)(th + 1);
3637
3638         if (*ptr == htonl((TCPOPT_NOP << 24) | (TCPOPT_NOP << 16)
3639                           | (TCPOPT_TIMESTAMP << 8) | TCPOLEN_TIMESTAMP)) {
3640                 tp->rx_opt.saw_tstamp = 1;
3641                 ++ptr;
3642                 tp->rx_opt.rcv_tsval = ntohl(*ptr);
3643                 ++ptr;
3644                 if (*ptr)
3645                         tp->rx_opt.rcv_tsecr = ntohl(*ptr) - tp->tsoffset;
3646                 else
3647                         tp->rx_opt.rcv_tsecr = 0;
3648                 return true;
3649         }
3650         return false;
3651 }
3652
3653 /* Fast parse options. This hopes to only see timestamps.
3654  * If it is wrong it falls back on tcp_parse_options().
3655  */
3656 static bool tcp_fast_parse_options(const struct sk_buff *skb,
3657                                    const struct tcphdr *th, struct tcp_sock *tp)
3658 {
3659         /* In the spirit of fast parsing, compare doff directly to constant
3660          * values.  Because equality is used, short doff can be ignored here.
3661          */
3662         if (th->doff == (sizeof(*th) / 4)) {
3663                 tp->rx_opt.saw_tstamp = 0;
3664                 return false;
3665         } else if (tp->rx_opt.tstamp_ok &&
3666                    th->doff == ((sizeof(*th) + TCPOLEN_TSTAMP_ALIGNED) / 4)) {
3667                 if (tcp_parse_aligned_timestamp(tp, th))
3668                         return true;
3669         }
3670
3671         tcp_parse_options(skb, &tp->rx_opt, 1, NULL);
3672         if (tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr)
3673                 tp->rx_opt.rcv_tsecr -= tp->tsoffset;
3674
3675         return true;
3676 }
```

```
3677
3678 #ifdef CONFIG_TCP_MD5SIG
3679 /*
3680  * Parse MD5 Signature option
3681  */
3682 const u8 *tcp_parse_md5sig_option(const struct tcphdr *th)
3683 {
3684         int length = (th->doff << 2) - sizeof(*th);
3685         const u8 *ptr = (const u8 *)(th + 1);
3686
3687         /* If the TCP option is too short, we can short cut */
3688         if (length < TCPOLEN_MD5SIG)
3689                 return NULL;
3690
3691         while (length > 0) {
3692                 int opcode = *ptr++;
3693                 int opsize;
3694
3695                 switch (opcode) {
3696                 case TCPOPT_EOL:
3697                         return NULL;
3698                 case TCPOPT_NOP:
3699                         length--;
3700                         continue;
3701                 default:
3702                         opsize = *ptr++;
3703                         if (opsize < 2 || opsize > length)
3704                                 return NULL;
3705                         if (opcode == TCPOPT_MD5SIG)
3706                                 return opsize == TCPOLEN_MD5SIG ? ptr : NULL;
3707                 }
3708                 ptr += opsize - 2;
3709                 length -= opsize;
3710         }
3711         return NULL;
3712 }
3713 EXPORT_SYMBOL(tcp_parse_md5sig_option);
3714 #endif
3715
3716 /* Sorry, PAWS as specified is broken wrt. pure-ACKs -DaveM
3717  *
3718  * It is not fatal. If this ACK does _not_ change critical state (seqs, window)
3719  * it can pass through stack. So, the following predicate verifies that
3720  * this segment is not used for anything but congestion avoidance or
3721  * fast retransmit. Moreover, we even are able to eliminate most of such
3722  * second order effects, if we apply some small "replay" window (~RTO)
3723  * to timestamp space.
3724  *
3725  * All these measures still do not guarantee that we reject wrapped ACKs
3726  * on networks with high bandwidth, when sequence space is recycled fastly,
3727  * but it guarantees that such events will be very rare and do not affect
3728  * connection seriously. This doesn't look nice, but alas, PAWS is really
3729  * buggy extension.
3730  *
3731  * [ Later note. Even worse! It is buggy for segments _with_ data. RFC
3732  * states that events when retransmit arrives after original data are rare.
3733  * It is a blatant lie. VJ forgot about fast retransmit! 8)8) It is
3734  * the biggest problem on large power networks even with minor reordering.
3735  * OK, let's give it small replay window. If peer clock is even 1hz, it is safe
3736  * up to bandwidth of 18Gigabit/sec. 8) ]
3737  */
3738
3739 static int tcp_disordered_ack(const struct sock *sk, const struct sk_buff *skb)
3740 {
3741         const struct tcp_sock *tp = tcp_sk(sk);
3742         const struct tcphdr *th = tcp_hdr(skb);
3743         u32 seq = TCP_SKB_CB(skb)->seq;
3744         u32 ack = TCP_SKB_CB(skb)->ack_seq;
3745
3746         return (/* 1. Pure ACK with correct sequence number. */
3747                 (th->ack && seq == TCP_SKB_CB(skb)->end_seq && seq == tp->rcv_nxt) &&
3748
3749                 /* 2. ... and duplicate ACK. */
3750                 ack == tp->snd_una &&
3751
3752                 /* 3. ... and does not update window. */
3753                 !tcp_may_update_window(tp, ack, seq, ntohs(th->window) << tp->rx_opt.snd_wscale) &&
3754
3755                 /* 4. ... and sits in replay window. */
3756                 (s32)(tp->rx_opt.ts_recent - tp->rx_opt.rcv_tsval) <= (inet_csk(sk)->icsk_rto * 1024) / HZ);
3757 }
3758
3759 static inline bool tcp_paws_discard(const struct sock *sk,
3760                                     const struct sk_buff *skb)
3761 {
3762         const struct tcp_sock *tp = tcp_sk(sk);
3763
3764         return !tcp_paws_check(&tp->rx_opt, TCP_PAWS_WINDOW) &&
```

```
3765                !tcp_disordered_ack(sk, skb);
3766 }
3767
3768 /* Check segment sequence number for validity.
3769  *
3770  * Segment controls are considered valid, if the segment
3771  * fits to the window after truncation to the window. Acceptability
3772  * of data (and SYN, FIN, of course) is checked separately.
3773  * See tcp_data_queue(), for example.
3774  *
3775  * Also, controls (RST is main one) are accepted using RCV.WUP instead
3776  * of RCV.NXT. Peer still did not advance his SND.UNA when we
3777  * delayed ACK, so that hisSND.UNA<=ourRCV.WUP.
3778  * (borrowed from freebsd)
3779  */
3780
3781 static inline bool tcp_sequence(const struct tcp_sock *tp, u32 seq, u32 end_seq)
3782 {
3783         return  !before(end_seq, tp->rcv_wup) &&
3784                 !after(seq, tp->rcv_nxt + tcp_receive_window(tp));
3785 }
3786
3787 /* When we get a reset we do this. */
3788 void tcp_reset(struct sock *sk)
3789 {
3790         /* We want the right error as BSD sees it (and indeed as we do). */
3791         switch (sk->sk_state) {
3792         case TCP_SYN_SENT:
3793                 sk->sk_err = ECONNREFUSED;
3794                 break;
3795         case TCP_CLOSE_WAIT:
3796                 sk->sk_err = EPIPE;
3797                 break;
3798         case TCP_CLOSE:
3799                 return;
3800         default:
3801                 sk->sk_err = ECONNRESET;
3802         }
3803         /* This barrier is coupled with smp_rmb() in tcp_poll() */
3804         smp_wmb();
3805
3806         if (!sock_flag(sk, SOCK_DEAD))
3807                 sk->sk_error_report(sk);
3808
3809         tcp_done(sk);
3810 }
3811
3812 /*
3813  *      Process the FIN bit. This now behaves as it is supposed to work
3814  *      and the FIN takes effect when it is validly part of sequence
3815  *      space. Not before when we get holes.
3816  *
3817  *      If we are ESTABLISHED, a received fin moves us to CLOSE-WAIT
3818  *      (and thence onto LAST-ACK and finally, CLOSE, we never enter
3819  *      TIME-WAIT)
3820  *
3821  *      If we are in FINWAIT-1, a received FIN indicates simultaneous
3822  *      close and we go into CLOSING (and later onto TIME-WAIT)
3823  *
3824  *      If we are in FINWAIT-2, a received FIN moves us to TIME-WAIT.
3825  */
3826 static void tcp_fin(struct sock *sk)
3827 {
3828         struct tcp_sock *tp = tcp_sk(sk);
3829         const struct dst_entry *dst;
3830
3831         inet_csk_schedule_ack(sk);
3832
3833         sk->sk_shutdown |= RCV_SHUTDOWN;
3834         sock_set_flag(sk, SOCK_DONE);
3835
3836         switch (sk->sk_state) {
3837         case TCP_SYN_RECV:
3838         case TCP_ESTABLISHED:
3839                 /* Move to CLOSE_WAIT */
3840                 tcp_set_state(sk, TCP_CLOSE_WAIT);
3841                 dst = __sk_dst_get(sk);
3842                 if (!dst || !dst_metric(dst, RTAX_QUICKACK))
3843                         inet_csk(sk)->icsk_ack.pingpong = 1;
3844                 break;
3845
3846         case TCP_CLOSE_WAIT:
3847         case TCP_CLOSING:
3848                 /* Received a retransmission of the FIN, do
3849                  * nothing.
3850                  */
3851                 break;
3852         case TCP_LAST_ACK:
```

```
3853                /* RFC793: Remain in the LAST-ACK state. */
3854                break;
3855
3856        case TCP_FIN_WAIT1:
3857                /* This case occurs when a simultaneous close
3858                 * happens, we must ack the received FIN and
3859                 * enter the CLOSING state.
3860                 */
3861                tcp_send_ack(sk);
3862                tcp_set_state(sk, TCP_CLOSING);
3863                break;
3864        case TCP_FIN_WAIT2:
3865                /* Received a FIN -- send ACK and enter TIME_WAIT. */
3866                tcp_send_ack(sk);
3867                tcp_time_wait(sk, TCP_TIME_WAIT, 0);
3868                break;
3869        default:
3870                /* Only TCP_LISTEN and TCP_CLOSE are left, in these
3871                 * cases we should never reach this piece of code.
3872                 */
3873                pr_err("%s: Impossible, sk->sk_state=%d\n",
3874                       __func__, sk->sk_state);
3875                break;
3876        }
3877
3878        /* It _is_ possible, that we have something out-of-order _after_ FIN.
3879         * Probably, we should reset in this case. For now drop them.
3880         */
3881        __skb_queue_purge(&tp->out_of_order_queue);
3882        if (tcp_is_sack(tp))
3883                tcp_sack_reset(&tp->rx_opt);
3884        sk_mem_reclaim(sk);
3885
3886        if (!sock_flag(sk, SOCK_DEAD)) {
3887                sk->sk_state_change(sk);
3888
3889                /* Do not send POLL_HUP for half duplex close. */
3890                if (sk->sk_shutdown == SHUTDOWN_MASK ||
3891                    sk->sk_state == TCP_CLOSE)
3892                        sk_wake_async(sk, SOCK_WAKE_WAITD, POLL_HUP);
3893                else
3894                        sk_wake_async(sk, SOCK_WAKE_WAITD, POLL_IN);
3895        }
3896 }
3897
3898 static inline bool tcp_sack_extend(struct tcp_sack_block *sp, u32 seq,
3899                                    u32 end_seq)
3900 {
3901        if (!after(seq, sp->end_seq) && !after(sp->start_seq, end_seq)) {
3902                if (before(seq, sp->start_seq))
3903                        sp->start_seq = seq;
3904                if (after(end_seq, sp->end_seq))
3905                        sp->end_seq = end_seq;
3906                return true;
3907        }
3908        return false;
3909 }
3910
3911 static void tcp_dsack_set(struct sock *sk, u32 seq, u32 end_seq)
3912 {
3913        struct tcp_sock *tp = tcp_sk(sk);
3914
3915        if (tcp_is_sack(tp) && sysctl_tcp_dsack) {
3916                int mib_idx;
3917
3918                if (before(seq, tp->rcv_nxt))
3919                        mib_idx = LINUX_MIB_TCPDSACKOLDSENT;
3920                else
3921                        mib_idx = LINUX_MIB_TCPDSACKOFOSENT;
3922
3923                NET_INC_STATS_BH(sock_net(sk), mib_idx);
3924
3925                tp->rx_opt.dsack = 1;
3926                tp->duplicate_sack[0].start_seq = seq;
3927                tp->duplicate_sack[0].end_seq = end_seq;
3928        }
3929 }
3930
3931 static void tcp_dsack_extend(struct sock *sk, u32 seq, u32 end_seq)
3932 {
3933        struct tcp_sock *tp = tcp_sk(sk);
3934
3935        if (!tp->rx_opt.dsack)
3936                tcp_dsack_set(sk, seq, end_seq);
3937        else
3938                tcp_sack_extend(tp->duplicate_sack, seq, end_seq);
3939 }
3940
```

```
3941 static void tcp_send_dupack(struct sock *sk, const struct sk_buff *skb)
3942 {
3943         struct tcp_sock *tp = tcp_sk(sk);
3944
3945         if (TCP_SKB_CB(skb)->end_seq != TCP_SKB_CB(skb)->seq &&
3946             before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt)) {
3947                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_DELAYEDACKLOST);
3948                 tcp_enter_quickack_mode(sk);
3949
3950                 if (tcp_is_sack(tp) && sysctl_tcp_dsack) {
3951                         u32 end_seq = TCP_SKB_CB(skb)->end_seq;
3952
3953                         if (after(TCP_SKB_CB(skb)->end_seq, tp->rcv_nxt))
3954                                 end_seq = tp->rcv_nxt;
3955                         tcp_dsack_set(sk, TCP_SKB_CB(skb)->seq, end_seq);
3956                 }
3957         }
3958
3959         tcp_send_ack(sk);
3960 }
3961
3962 /* These routines update the SACK block as out-of-order packets arrive or
3963  * in-order packets close up the sequence space.
3964  */
3965 static void tcp_sack_maybe_coalesce(struct tcp_sock *tp)
3966 {
3967         int this_sack;
3968         struct tcp_sack_block *sp = &tp->selective_acks[0];
3969         struct tcp_sack_block *swalk = sp + 1;
3970
3971         /* See if the recent change to the first SACK eats into
3972          * or hits the sequence space of other SACK blocks, if so coalesce.
3973          */
3974         for (this_sack = 1; this_sack < tp->rx_opt.num_sacks;) {
3975                 if (tcp_sack_extend(sp, swalk->start_seq, swalk->end_seq)) {
3976                         int i;
3977
3978                         /* Zap SWALK, by moving every further SACK up by one slot.
3979                          * Decrease num_sacks.
3980                          */
3981                         tp->rx_opt.num_sacks--;
3982                         for (i = this_sack; i < tp->rx_opt.num_sacks; i++)
3983                                 sp[i] = sp[i + 1];
3984                         continue;
3985                 }
3986                 this_sack++, swalk++;
3987         }
3988 }
3989
3990 static void tcp_sack_new_ofo_skb(struct sock *sk, u32 seq, u32 end_seq)
3991 {
3992         struct tcp_sock *tp = tcp_sk(sk);
3993         struct tcp_sack_block *sp = &tp->selective_acks[0];
3994         int cur_sacks = tp->rx_opt.num_sacks;
3995         int this_sack;
3996
3997         if (!cur_sacks)
3998                 goto new_sack;
3999
4000         for (this_sack = 0; this_sack < cur_sacks; this_sack++, sp++) {
4001                 if (tcp_sack_extend(sp, seq, end_seq)) {
4002                         /* Rotate this_sack to the first one. */
4003                         for (; this_sack > 0; this_sack--, sp--)
4004                                 swap(*sp, *(sp - 1));
4005                         if (cur_sacks > 1)
4006                                 tcp_sack_maybe_coalesce(tp);
4007                         return;
4008                 }
4009         }
4010
4011         /* Could not find an adjacent existing SACK, build a new one,
4012          * put it at the front, and shift everyone else down.  We
4013          * always know there is at least one SACK present already here.
4014          *
4015          * If the sack array is full, forget about the last one.
4016          */
4017         if (this_sack >= TCP_NUM_SACKS) {
4018                 this_sack--;
4019                 tp->rx_opt.num_sacks--;
4020                 sp--;
4021         }
4022         for (; this_sack > 0; this_sack--, sp--)
4023                 *sp = *(sp - 1);
4024
4025 new_sack:
4026         /* Build the new head SACK, and we're done. */
4027         sp->start_seq = seq;
4028         sp->end_seq = end_seq;
```

```
4029         tp->rx_opt.num_sacks++;
4030 }
4031
4032 /* RCV.NXT advances, some SACKs should be eaten. */
4033
4034 static void tcp_sack_remove(struct tcp_sock *tp)
4035 {
4036         struct tcp_sack_block *sp = &tp->selective_acks[0];
4037         int num_sacks = tp->rx_opt.num_sacks;
4038         int this_sack;
4039
4040         /* Empty ofo queue, hence, all the SACKs are eaten. Clear. */
4041         if (skb_queue_empty(&tp->out_of_order_queue)) {
4042                 tp->rx_opt.num_sacks = 0;
4043                 return;
4044         }
4045
4046         for (this_sack = 0; this_sack < num_sacks;) {
4047                 /* Check if the start of the sack is covered by RCV.NXT. */
4048                 if (!before(tp->rcv_nxt, sp->start_seq)) {
4049                         int i;
4050
4051                         /* RCV.NXT must cover all the block! */
4052                         WARN_ON(before(tp->rcv_nxt, sp->end_seq));
4053
4054                         /* Zap this SACK, by moving forward any other SACKS. */
4055                         for (i = this_sack+1; i < num_sacks; i++)
4056                                 tp->selective_acks[i-1] = tp->selective_acks[i];
4057                         num_sacks--;
4058                         continue;
4059                 }
4060                 this_sack++;
4061                 sp++;
4062         }
4063         tp->rx_opt.num_sacks = num_sacks;
4064 }
4065
4066 /* This one checks to see if we can put data from the
4067  * out_of_order queue into the receive_queue.
4068  */
4069 static void tcp_ofo_queue(struct sock *sk)
4070 {
4071         struct tcp_sock *tp = tcp_sk(sk);
4072         __u32 dsack_high = tp->rcv_nxt;
4073         struct sk_buff *skb;
4074
4075         while ((skb = skb_peek(&tp->out_of_order_queue)) != NULL) {
4076                 if (after(TCP_SKB_CB(skb)->seq, tp->rcv_nxt))
4077                         break;
4078
4079                 if (before(TCP_SKB_CB(skb)->seq, dsack_high)) {
4080                         __u32 dsack = dsack_high;
4081                         if (before(TCP_SKB_CB(skb)->end_seq, dsack_high))
4082                                 dsack_high = TCP_SKB_CB(skb)->end_seq;
4083                         tcp_dsack_extend(sk, TCP_SKB_CB(skb)->seq, dsack);
4084                 }
4085
4086                 if (!after(TCP_SKB_CB(skb)->end_seq, tp->rcv_nxt)) {
4087                         SOCK_DEBUG(sk, "ofo packet was already received\n");
4088                         __skb_unlink(skb, &tp->out_of_order_queue);
4089                         __kfree_skb(skb);
4090                         continue;
4091                 }
4092                 SOCK_DEBUG(sk, "ofo requeuing : rcv_next %X seq %X - %X\n",
4093                            tp->rcv_nxt, TCP_SKB_CB(skb)->seq,
4094                            TCP_SKB_CB(skb)->end_seq);
4095
4096                 __skb_unlink(skb, &tp->out_of_order_queue);
4097                 __skb_queue_tail(&sk->sk_receive_queue, skb);
4098                 tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;
4099                 if (tcp_hdr(skb)->fin)
4100                         tcp_fin(sk);
4101         }
4102 }
4103
4104 static bool tcp_prune_ofo_queue(struct sock *sk);
4105 static int tcp_prune_queue(struct sock *sk);
4106
4107 static int tcp_try_rmem_schedule(struct sock *sk, struct sk_buff *skb,
4108                                  unsigned int size)
4109 {
4110         if (atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf ||
4111             !sk_rmem_schedule(sk, skb, size)) {
4112
4113                 if (tcp_prune_queue(sk) < 0)
4114                         return -1;
4115
4116                 if (!sk_rmem_schedule(sk, skb, size)) {
```

```
4117                        if (!tcp_prune_ofo_queue(sk))
4118                                return -1;
4119
4120                        if (!sk_rmem_schedule(sk, skb, size))
4121                                return -1;
4122                }
4123        }
4124        return 0;
4125 }
4126
4127 /**
4128  * tcp_try_coalesce - try to merge skb to prior one
4129  * @sk: socket
4130  * @to: prior buffer
4131  * @from: buffer to add in queue
4132  * @fragstolen: pointer to boolean
4133  *
4134  * Before queueing skb @from after @to, try to merge them
4135  * to reduce overall memory use and queue lengths, if cost is small.
4136  * Packets in ofo or receive queues can stay a long time.
4137  * Better try to coalesce them right now to avoid future collapses.
4138  * Returns true if caller should free @from instead of queueing it
4139  */
4140 static bool tcp_try_coalesce(struct sock *sk,
4141                              struct sk_buff *to,
4142                              struct sk_buff *from,
4143                              bool *fragstolen)
4144 {
4145        int delta;
4146
4147        *fragstolen = false;
4148
4149        if (tcp_hdr(from)->fin)
4150                return false;
4151
4152        /* Its possible this segment overlaps with prior segment in queue */
4153        if (TCP_SKB_CB(from)->seq != TCP_SKB_CB(to)->end_seq)
4154                return false;
4155
4156        if (!skb_try_coalesce(to, from, fragstolen, &delta))
4157                return false;
4158
4159        atomic_add(delta, &sk->sk_rmem_alloc);
4160        sk_mem_charge(sk, delta);
4161        NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPRCVCOALESCE);
4162        TCP_SKB_CB(to)->end_seq = TCP_SKB_CB(from)->end_seq;
4163        TCP_SKB_CB(to)->ack_seq = TCP_SKB_CB(from)->ack_seq;
4164        return true;
4165 }
4166
4167 static void tcp_data_queue_ofo(struct sock *sk, struct sk_buff *skb)
4168 {
4169        struct tcp_sock *tp = tcp_sk(sk);
4170        struct sk_buff *skb1;
4171        u32 seq, end_seq;
4172
4173        TCP_ECN_check_ce(tp, skb);
4174
4175        if (unlikely(tcp_try_rmem_schedule(sk, skb, skb->truesize))) {
4176                NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPOFODROP);
4177                __kfree_skb(skb);
4178                return;
4179        }
4180
4181        /* Disable header prediction. */
4182        tp->pred_flags = 0;
4183        inet_csk_schedule_ack(sk);
4184
4185        NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPOFOQUEUE);
4186        SOCK_DEBUG(sk, "out of order segment: rcv_next %X seq %X - %X\n",
4187                   tp->rcv_nxt, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq);
4188
4189        skb1 = skb_peek_tail(&tp->out_of_order_queue);
4190        if (!skb1) {
4191                /* Initial out of order segment, build 1 SACK. */
4192                if (tcp_is_sack(tp)) {
4193                        tp->rx_opt.num_sacks = 1;
4194                        tp->selective_acks[0].start_seq = TCP_SKB_CB(skb)->seq;
4195                        tp->selective_acks[0].end_seq =
4196                                                TCP_SKB_CB(skb)->end_seq;
4197                }
4198                __skb_queue_head(&tp->out_of_order_queue, skb);
4199                goto end;
4200        }
4201
4202        seq = TCP_SKB_CB(skb)->seq;
4203        end_seq = TCP_SKB_CB(skb)->end_seq;
4204
```

```
4205         if (seq == TCP_SKB_CB(skb1)->end_seq) {
4206                 bool fragstolen;
4207
4208                 if (!tcp_try_coalesce(sk, skb1, skb, &fragstolen)) {
4209                         __skb_queue_after(&tp->out_of_order_queue, skb1, skb);
4210                 } else {
4211                         tcp_grow_window(sk, skb);
4212                         kfree_skb_partial(skb, fragstolen);
4213                         skb = NULL;
4214                 }
4215
4216                 if (!tp->rx_opt.num_sacks ||
4217                     tp->selective_acks[0].end_seq != seq)
4218                         goto add_sack;
4219
4220                 /* Common case: data arrive in order after hole. */
4221                 tp->selective_acks[0].end_seq = end_seq;
4222                 goto end;
4223         }
4224
4225         /* Find place to insert this segment. */
4226         while (1) {
4227                 if (!after(TCP_SKB_CB(skb1)->seq, seq))
4228                         break;
4229                 if (skb_queue_is_first(&tp->out_of_order_queue, skb1)) {
4230                         skb1 = NULL;
4231                         break;
4232                 }
4233                 skb1 = skb_queue_prev(&tp->out_of_order_queue, skb1);
4234         }
4235
4236         /* Do skb overlap to previous one? */
4237         if (skb1 && before(seq, TCP_SKB_CB(skb1)->end_seq)) {
4238                 if (!after(end_seq, TCP_SKB_CB(skb1)->end_seq)) {
4239                         /* All the bits are present. Drop. */
4240                         NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPOFOMERGE);
4241                         __kfree_skb(skb);
4242                         skb = NULL;
4243                         tcp_dsack_set(sk, seq, end_seq);
4244                         goto add_sack;
4245                 }
4246                 if (after(seq, TCP_SKB_CB(skb1)->seq)) {
4247                         /* Partial overlap. */
4248                         tcp_dsack_set(sk, seq,
4249                                       TCP_SKB_CB(skb1)->end_seq);
4250                 } else {
4251                         if (skb_queue_is_first(&tp->out_of_order_queue,
4252                                                skb1))
4253                                 skb1 = NULL;
4254                         else
4255                                 skb1 = skb_queue_prev(
4256                                         &tp->out_of_order_queue,
4257                                         skb1);
4258                 }
4259         }
4260         if (!skb1)
4261                 __skb_queue_head(&tp->out_of_order_queue, skb);
4262         else
4263                 __skb_queue_after(&tp->out_of_order_queue, skb1, skb);
4264
4265         /* And clean segments covered by new one as whole. */
4266         while (!skb_queue_is_last(&tp->out_of_order_queue, skb)) {
4267                 skb1 = skb_queue_next(&tp->out_of_order_queue, skb);
4268
4269                 if (!after(end_seq, TCP_SKB_CB(skb1)->seq))
4270                         break;
4271                 if (before(end_seq, TCP_SKB_CB(skb1)->end_seq)) {
4272                         tcp_dsack_extend(sk, TCP_SKB_CB(skb1)->seq,
4273                                          end_seq);
4274                         break;
4275                 }
4276                 __skb_unlink(skb1, &tp->out_of_order_queue);
4277                 tcp_dsack_extend(sk, TCP_SKB_CB(skb1)->seq,
4278                                  TCP_SKB_CB(skb1)->end_seq);
4279                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPOFOMERGE);
4280                 __kfree_skb(skb1);
4281         }
4282
4283 add_sack:
4284         if (tcp_is_sack(tp))
4285                 tcp_sack_new_ofo_skb(sk, seq, end_seq);
4286 end:
4287         if (skb) {
4288                 tcp_grow_window(sk, skb);
4289                 skb_set_owner_r(skb, sk);
4290         }
4291 }
4292
```

```
4293 static int __must_check tcp_queue_rcv(struct sock *sk, struct sk_buff *skb, int hdrlen,
4294                   bool *fragstolen)
4295 {
4296         int eaten;
4297         struct sk_buff *tail = skb_peek_tail(&sk->sk_receive_queue);
4298
4299         __skb_pull(skb, hdrlen);
4300         eaten = (tail &&
4301                  tcp_try_coalesce(sk, tail, skb, fragstolen)) ? 1 : 0;
4302         tcp_sk(sk)->rcv_nxt = TCP_SKB_CB(skb)->end_seq;
4303         if (!eaten) {
4304                 __skb_queue_tail(&sk->sk_receive_queue, skb);
4305                 skb_set_owner_r(skb, sk);
4306         }
4307         return eaten;
4308 }
4309
4310 int tcp_send_rcvq(struct sock *sk, struct msghdr *msg, size_t size)
4311 {
4312         struct sk_buff *skb = NULL;
4313         struct tcphdr *th;
4314         bool fragstolen;
4315
4316         if (size == 0)
4317                 return 0;
4318
4319         skb = alloc_skb(size + sizeof(*th), sk->sk_allocation);
4320         if (!skb)
4321                 goto err;
4322
4323         if (tcp_try_rmem_schedule(sk, skb, size + sizeof(*th)))
4324                 goto err_free;
4325
4326         th = (struct tcphdr *)skb_put(skb, sizeof(*th));
4327         skb_reset_transport_header(skb);
4328         memset(th, 0, sizeof(*th));
4329
4330         if (memcpy_fromiovec(skb_put(skb, size), msg->msg_iov, size))
4331                 goto err_free;
4332
4333         TCP_SKB_CB(skb)->seq = tcp_sk(sk)->rcv_nxt;
4334         TCP_SKB_CB(skb)->end_seq = TCP_SKB_CB(skb)->seq + size;
4335         TCP_SKB_CB(skb)->ack_seq = tcp_sk(sk)->snd_una - 1;
4336
4337         if (tcp_queue_rcv(sk, skb, sizeof(*th), &fragstolen)) {
4338                 WARN_ON_ONCE(fragstolen); /* should not happen */
4339                 __kfree_skb(skb);
4340         }
4341         return size;
4342
4343 err_free:
4344         kfree_skb(skb);
4345 err:
4346         return -ENOMEM;
4347 }
4348
4349 static void tcp_data_queue(struct sock *sk, struct sk_buff *skb)
4350 {
4351         const struct tcphdr *th = tcp_hdr(skb);
4352         struct tcp_sock *tp = tcp_sk(sk);
4353         int eaten = -1;
4354         bool fragstolen = false;
4355
4356         if (TCP_SKB_CB(skb)->seq == TCP_SKB_CB(skb)->end_seq)
4357                 goto drop;
4358
4359         skb_dst_drop(skb);
4360         __skb_pull(skb, th->doff * 4);
4361
4362         TCP_ECN_accept_cwr(tp, skb);
4363
4364         tp->rx_opt.dsack = 0;
4365
4366         /*  Queue data for delivery to the user.
4367          *  Packets in sequence go to the receive queue.
4368          *  Out of sequence packets to the out_of_order_queue.
4369          */
4370         if (TCP_SKB_CB(skb)->seq == tp->rcv_nxt) {
4371                 if (tcp_receive_window(tp) == 0)
4372                         goto out_of_window;
4373
4374                 /* Ok. In sequence. In window. */
4375                 if (tp->ucopy.task == current &&
4376                     tp->copied_seq == tp->rcv_nxt && tp->ucopy.len &&
4377                     sock_owned_by_user(sk) && !tp->urg_data) {
4378                         int chunk = min_t(unsigned int, skb->len,
4379                                           tp->ucopy.len);
4380
```

```
4381                        __set_current_state(TASK_RUNNING);
4382
4383                        local_bh_enable();
4384                        if (!skb_copy_datagram_iovec(skb, 0, tp->ucopy.iov, chunk)) {
4385                                tp->ucopy.len -= chunk;
4386                                tp->copied_seq += chunk;
4387                                eaten = (chunk == skb->len);
4388                                tcp_rcv_space_adjust(sk);
4389                        }
4390                        local_bh_disable();
4391                }
4392
4393                if (eaten <= 0) {
4394 queue_and_out:
4395                        if (eaten < 0 &&
4396                            tcp_try_rmem_schedule(sk, skb, skb->truesize))
4397                                goto drop;
4398
4399                        eaten = tcp_queue_rcv(sk, skb, 0, &fragstolen);
4400                }
4401                tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;
4402                if (skb->len)
4403                        tcp_event_data_recv(sk, skb);
4404                if (th->fin)
4405                        tcp_fin(sk);
4406
4407                if (!skb_queue_empty(&tp->out_of_order_queue)) {
4408                        tcp_ofo_queue(sk);
4409
4410                        /* RFC2581. 4.2. SHOULD send immediate ACK, when
4411                         * gap in queue is filled.
4412                         */
4413                        if (skb_queue_empty(&tp->out_of_order_queue))
4414                                inet_csk(sk)->icsk_ack.pingpong = 0;
4415                }
4416
4417                if (tp->rx_opt.num_sacks)
4418                        tcp_sack_remove(tp);
4419
4420                tcp_fast_path_check(sk);
4421
4422                if (eaten > 0)
4423                        kfree_skb_partial(skb, fragstolen);
4424                if (!sock_flag(sk, SOCK_DEAD))
4425                        sk->sk_data_ready(sk);
4426                return;
4427        }
4428
4429        if (!after(TCP_SKB_CB(skb)->end_seq, tp->rcv_nxt)) {
4430                /* A retransmit, 2nd most common case.  Force an immediate ack. */
4431                NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_DELAYEDACKLOST);
4432                tcp_dsack_set(sk, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq);
4433
4434 out_of_window:
4435                tcp_enter_quickack_mode(sk);
4436                inet_csk_schedule_ack(sk);
4437 drop:
4438                __kfree_skb(skb);
4439                return;
4440        }
4441
4442        /* Out of window. F.e. zero window probe. */
4443        if (!before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt + tcp_receive_window(tp)))
4444                goto out_of_window;
4445
4446        tcp_enter_quickack_mode(sk);
4447
4448        if (before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt)) {
4449                /* Partial packet, seq < rcv_next < end_seq */
4450                SOCK_DEBUG(sk, "partial packet: rcv_next %X seq %X - %X\n",
4451                           tp->rcv_nxt, TCP_SKB_CB(skb)->seq,
4452                           TCP_SKB_CB(skb)->end_seq);
4453
4454                tcp_dsack_set(sk, TCP_SKB_CB(skb)->seq, tp->rcv_nxt);
4455
4456                /* If window is closed, drop tail of packet. But after
4457                 * remembering D-SACK for its head made in previous line.
4458                 */
4459                if (!tcp_receive_window(tp))
4460                        goto out_of_window;
4461                goto queue_and_out;
4462        }
4463
4464        tcp_data_queue_ofo(sk, skb);
4465 }
4466
4467 static struct sk_buff *tcp_collapse_one(struct sock *sk, struct sk_buff *skb,
4468                                         struct sk_buff_head *list)
```

```
4469 {
4470         struct sk_buff *next = NULL;
4471
4472         if (!skb_queue_is_last(list, skb))
4473                 next = skb_queue_next(list, skb);
4474
4475         __skb_unlink(skb, list);
4476         __kfree_skb(skb);
4477         NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPRCVCOLLAPSED);
4478
4479         return next;
4480 }
4481
4482 /* Collapse contiguous sequence of skbs head..tail with
4483  * sequence numbers start..end.
4484  *
4485  * If tail is NULL, this means until the end of the list.
4486  *
4487  * Segments with FIN/SYN are not collapsed (only because this
4488  * simplifies code)
4489  */
4490 static void
4491 tcp_collapse(struct sock *sk, struct sk_buff_head *list,
4492              struct sk_buff *head, struct sk_buff *tail,
4493              u32 start, u32 end)
4494 {
4495         struct sk_buff *skb, *n;
4496         bool end_of_skbs;
4497
4498         /* First, check that queue is collapsible and find
4499          * the point where collapsing can be useful. */
4500         skb = head;
4501 restart:
4502         end_of_skbs = true;
4503         skb_queue_walk_from_safe(list, skb, n) {
4504                 if (skb == tail)
4505                         break;
4506                 /* No new bits? It is possible on ofo queue. */
4507                 if (!before(start, TCP_SKB_CB(skb)->end_seq)) {
4508                         skb = tcp_collapse_one(sk, skb, list);
4509                         if (!skb)
4510                                 break;
4511                         goto restart;
4512                 }
4513
4514                 /* The first skb to collapse is:
4515                  * - not SYN/FIN and
4516                  * - bloated or contains data before "start" or
4517                  *   overlaps to the next one.
4518                  */
4519                 if (!tcp_hdr(skb)->syn && !tcp_hdr(skb)->fin &&
4520                     (tcp_win_from_space(skb->truesize) > skb->len ||
4521                      before(TCP_SKB_CB(skb)->seq, start))) {
4522                         end_of_skbs = false;
4523                         break;
4524                 }
4525
4526                 if (!skb_queue_is_last(list, skb)) {
4527                         struct sk_buff *next = skb_queue_next(list, skb);
4528                         if (next != tail &&
4529                             TCP_SKB_CB(skb)->end_seq != TCP_SKB_CB(next)->seq) {
4530                                 end_of_skbs = false;
4531                                 break;
4532                         }
4533                 }
4534
4535                 /* Decided to skip this, advance start seq. */
4536                 start = TCP_SKB_CB(skb)->end_seq;
4537         }
4538         if (end_of_skbs || tcp_hdr(skb)->syn || tcp_hdr(skb)->fin)
4539                 return;
4540
4541         while (before(start, end)) {
4542                 struct sk_buff *nskb;
4543                 unsigned int header = skb_headroom(skb);
4544                 int copy = SKB_MAX_ORDER(header, 0);
4545
4546                 /* Too big header? This can happen with IPv6. */
4547                 if (copy < 0)
4548                         return;
4549                 if (end - start < copy)
4550                         copy = end - start;
4551                 nskb = alloc_skb(copy + header, GFP_ATOMIC);
4552                 if (!nskb)
4553                         return;
4554
4555                 skb_set_mac_header(nskb, skb_mac_header(skb) - skb->head);
4556                 skb_set_network_header(nskb, (skb_network_header(skb) -
```

```
4557                                            skb->head));
4558                 skb_set_transport_header(nskb, (skb_transport_header(skb) -
4559                                                 skb->head));
4560                 skb_reserve(nskb, header);
4561                 memcpy(nskb->head, skb->head, header);
4562                 memcpy(nskb->cb, skb->cb, sizeof(skb->cb));
4563                 TCP_SKB_CB(nskb)->seq = TCP_SKB_CB(nskb)->end_seq = start;
4564                 __skb_queue_before(list, skb, nskb);
4565                 skb_set_owner_r(nskb, sk);
4566
4567                 /* Copy data, releasing collapsed skbs. */
4568                 while (copy > 0) {
4569                         int offset = start - TCP_SKB_CB(skb)->seq;
4570                         int size = TCP_SKB_CB(skb)->end_seq - start;
4571
4572                         BUG_ON(offset < 0);
4573                         if (size > 0) {
4574                                 size = min(copy, size);
4575                                 if (skb_copy_bits(skb, offset, skb_put(nskb, size), size))
4576                                         BUG();
4577                                 TCP_SKB_CB(nskb)->end_seq += size;
4578                                 copy -= size;
4579                                 start += size;
4580                         }
4581                         if (!before(start, TCP_SKB_CB(skb)->end_seq)) {
4582                                 skb = tcp_collapse_one(sk, skb, list);
4583                                 if (!skb ||
4584                                     skb == tail ||
4585                                     tcp_hdr(skb)->syn ||
4586                                     tcp_hdr(skb)->fin)
4587                                         return;
4588                         }
4589                 }
4590         }
4591 }
4592
4593 /* Collapse ofo queue. Algorithm: select contiguous sequence of skbs
4594  * and tcp_collapse() them until all the queue is collapsed.
4595  */
4596 static void tcp_collapse_ofo_queue(struct sock *sk)
4597 {
4598         struct tcp_sock *tp = tcp_sk(sk);
4599         struct sk_buff *skb = skb_peek(&tp->out_of_order_queue);
4600         struct sk_buff *head;
4601         u32 start, end;
4602
4603         if (skb == NULL)
4604                 return;
4605
4606         start = TCP_SKB_CB(skb)->seq;
4607         end = TCP_SKB_CB(skb)->end_seq;
4608         head = skb;
4609
4610         for (;;) {
4611                 struct sk_buff *next = NULL;
4612
4613                 if (!skb_queue_is_last(&tp->out_of_order_queue, skb))
4614                         next = skb_queue_next(&tp->out_of_order_queue, skb);
4615                 skb = next;
4616
4617                 /* Segment is terminated when we see gap or when
4618                  * we are at the end of all the queue. */
4619                 if (!skb ||
4620                     after(TCP_SKB_CB(skb)->seq, end) ||
4621                     before(TCP_SKB_CB(skb)->end_seq, start)) {
4622                         tcp_collapse(sk, &tp->out_of_order_queue,
4623                                      head, skb, start, end);
4624                         head = skb;
4625                         if (!skb)
4626                                 break;
4627                         /* Start new segment */
4628                         start = TCP_SKB_CB(skb)->seq;
4629                         end = TCP_SKB_CB(skb)->end_seq;
4630                 } else {
4631                         if (before(TCP_SKB_CB(skb)->seq, start))
4632                                 start = TCP_SKB_CB(skb)->seq;
4633                         if (after(TCP_SKB_CB(skb)->end_seq, end))
4634                                 end = TCP_SKB_CB(skb)->end_seq;
4635                 }
4636         }
4637 }
4638
4639 /*
4640  * Purge the out-of-order queue.
4641  * Return true if queue was pruned.
4642  */
4643 static bool tcp_prune_ofo_queue(struct sock *sk)
4644 {
```

```
4645         struct tcp_sock *tp = tcp_sk(sk);
4646         bool res = false;
4647
4648         if (!skb_queue_empty(&tp->out_of_order_queue)) {
4649                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_OFOPRUNED);
4650                 __skb_queue_purge(&tp->out_of_order_queue);
4651
4652                 /* Reset SACK state.  A conforming SACK implementation will
4653                  * do the same at a timeout based retransmit.  When a connection
4654                  * is in a sad state like this, we care only about integrity
4655                  * of the connection not performance.
4656                  */
4657                 if (tp->rx_opt.sack_ok)
4658                         tcp_sack_reset(&tp->rx_opt);
4659                 sk_mem_reclaim(sk);
4660                 res = true;
4661         }
4662         return res;
4663 }
4664
4665 /* Reduce allocated memory if we can, trying to get
4666  * the socket within its memory limits again.
4667  *
4668  * Return less than zero if we should start dropping frames
4669  * until the socket owning process reads some of the data
4670  * to stabilize the situation.
4671  */
4672 static int tcp_prune_queue(struct sock *sk)
4673 {
4674         struct tcp_sock *tp = tcp_sk(sk);
4675
4676         SOCK_DEBUG(sk, "prune_queue: c=%x\n", tp->copied_seq);
4677
4678         NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_PRUNECALLED);
4679
4680         if (atomic_read(&sk->sk_rmem_alloc) >= sk->sk_rcvbuf)
4681                 tcp_clamp_window(sk);
4682         else if (sk_under_memory_pressure(sk))
4683                 tp->rcv_ssthresh = min(tp->rcv_ssthresh, 4U * tp->advmss);
4684
4685         tcp_collapse_ofo_queue(sk);
4686         if (!skb_queue_empty(&sk->sk_receive_queue))
4687                 tcp_collapse(sk, &sk->sk_receive_queue,
4688                              skb_peek(&sk->sk_receive_queue),
4689                              NULL,
4690                              tp->copied_seq, tp->rcv_nxt);
4691         sk_mem_reclaim(sk);
4692
4693         if (atomic_read(&sk->sk_rmem_alloc) <= sk->sk_rcvbuf)
4694                 return 0;
4695
4696         /* Collapsing did not help, destructive actions follow.
4697          * This must not ever occur. */
4698
4699         tcp_prune_ofo_queue(sk);
4700
4701         if (atomic_read(&sk->sk_rmem_alloc) <= sk->sk_rcvbuf)
4702                 return 0;
4703
4704         /* If we are really being abused, tell the caller to silently
4705          * drop receive data on the floor.  It will get retransmitted
4706          * and hopefully then we'll have sufficient space.
4707          */
4708         NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_RCVPRUNED);
4709
4710         /* Massive buffer overcommit. */
4711         tp->pred_flags = 0;
4712         return -1;
4713 }
4714
4715 static bool tcp_should_expand_sndbuf(const struct sock *sk)
4716 {
4717         const struct tcp_sock *tp = tcp_sk(sk);
4718
4719         /* If the user specified a specific send buffer setting, do
4720          * not modify it.
4721          */
4722         if (sk->sk_userlocks & SOCK_SNDBUF_LOCK)
4723                 return false;
4724
4725         /* If we are under global TCP memory pressure, do not expand.  */
4726         if (sk_under_memory_pressure(sk))
4727                 return false;
4728
4729         /* If we are under soft global TCP memory pressure, do not expand.  */
4730         if (sk_memory_allocated(sk) >= sk_prot_mem_limits(sk, 0))
4731                 return false;
4732
```

```
4733         /* If we filled the congestion window, do not expand.  */
4734         if (tp->packets_out >= tp->snd_cwnd)
4735                 return false;
4736
4737         return true;
4738 }
4739
4740 /* When incoming ACK allowed to free some skb from write_queue,
4741  * we remember this event in flag SOCK_QUEUE_SHRUNK and wake up socket
4742  * on the exit from tcp input handler.
4743  *
4744  * PROBLEM: sndbuf expansion does not work well with largesend.
4745  */
4746 static void tcp_new_space(struct sock *sk)
4747 {
4748         struct tcp_sock *tp = tcp_sk(sk);
4749
4750         if (tcp_should_expand_sndbuf(sk)) {
4751                 tcp_sndbuf_expand(sk);
4752                 tp->snd_cwnd_stamp = tcp_time_stamp;
4753         }
4754
4755         sk->sk_write_space(sk);
4756 }
4757
4758 static void tcp_check_space(struct sock *sk)
4759 {
4760         if (sock_flag(sk, SOCK_QUEUE_SHRUNK)) {
4761                 sock_reset_flag(sk, SOCK_QUEUE_SHRUNK);
4762                 if (sk->sk_socket &&
4763                     test_bit(SOCK_NOSPACE, &sk->sk_socket->flags))
4764                         tcp_new_space(sk);
4765         }
4766 }
4767
4768 static inline void tcp_data_snd_check(struct sock *sk)
4769 {
4770         tcp_push_pending_frames(sk);
4771         tcp_check_space(sk);
4772 }
4773
4774 /*
4775  * Check if sending an ack is needed.
4776  */
4777 static void __tcp_ack_snd_check(struct sock *sk, int ofo_possible)
4778 {
4779         struct tcp_sock *tp = tcp_sk(sk);
4780
4781             /* More than one full frame received... */
4782         if (((tp->rcv_nxt - tp->rcv_wup) > inet_csk(sk)->icsk_ack.rcv_mss &&
4783              /* ... and right edge of window advances far enough.
4784               * (tcp_recvmsg() will send ACK otherwise). Or...
4785               */
4786              __tcp_select_window(sk) >= tp->rcv_wnd) ||
4787             /* We ACK each frame or... */
4788             tcp_in_quickack_mode(sk) ||
4789             /* We have out of order data. */
4790             (ofo_possible && skb_peek(&tp->out_of_order_queue))) {
4791                 /* Then ack it now */
4792                 tcp_send_ack(sk);
4793         } else {
4794                 /* Else, send delayed ack. */
4795                 tcp_send_delayed_ack(sk);
4796         }
4797 }
4798
4799 static inline void tcp_ack_snd_check(struct sock *sk)
4800 {
4801         if (!inet_csk_ack_scheduled(sk)) {
4802                 /* We sent a data segment already. */
4803                 return;
4804         }
4805         __tcp_ack_snd_check(sk, 1);
4806 }
4807
4808 /*
4809  *      This routine is only called when we have urgent data
4810  *      signaled. Its the 'slow' part of tcp_urg. It could be
4811  *      moved inline now as tcp_urg is only called from one
4812  *      place. We handle URGent data wrong. We have to - as
4813  *      BSD still doesn't use the correction from RFC961.
4814  *      For 1003.1g we should support a new option TCP_STDURG to permit
4815  *      either form (or just set the sysctl tcp_stdurg).
4816  */
4817
4818 static void tcp_check_urg(struct sock *sk, const struct tcphdr *th)
4819 {
4820         struct tcp_sock *tp = tcp_sk(sk);
```

```
4821         u32 ptr = ntohs(th->urg_ptr);
4822
4823         if (ptr && !sysctl_tcp_stdurg)
4824                 ptr--;
4825         ptr += ntohl(th->seq);
4826
4827         /* Ignore urgent data that we've already seen and read. */
4828         if (after(tp->copied_seq, ptr))
4829                 return;
4830
4831         /* Do not replay urg ptr.
4832          *
4833          * NOTE: interesting situation not covered by specs.
4834          * Misbehaving sender may send urg ptr, pointing to segment,
4835          * which we already have in ofo queue. We are not able to fetch
4836          * such data and will stay in TCP_URG_NOTYET until will be eaten
4837          * by recvmsg(). Seems, we are not obliged to handle such wicked
4838          * situations. But it is worth to think about possibility of some
4839          * DoSes using some hypothetical application level deadlock.
4840          */
4841         if (before(ptr, tp->rcv_nxt))
4842                 return;
4843
4844         /* Do we already have a newer (or duplicate) urgent pointer? */
4845         if (tp->urg_data && !after(ptr, tp->urg_seq))
4846                 return;
4847
4848         /* Tell the world about our new urgent pointer. */
4849         sk_send_sigurg(sk);
4850
4851         /* We may be adding urgent data when the last byte read was
4852          * urgent. To do this requires some care. We cannot just ignore
4853          * tp->copied_seq since we would read the last urgent byte again
4854          * as data, nor can we alter copied_seq until this data arrives
4855          * or we break the semantics of SIOCATMARK (and thus sockatmark())
4856          *
4857          * NOTE. Double Dutch. Rendering to plain English: author of comment
4858          * above did something sort of  send("A", MSG_OOB); send("B", MSG_OOB);
4859          * and expect that both A and B disappear from stream. This is _wrong_.
4860          * Though this happens in BSD with high probability, this is occasional.
4861          * Any application relying on this is buggy. Note also, that fix "works"
4862          * only in this artificial test. Insert some normal data between A and B and we will
4863          * decline of BSD again. Verdict: it is better to remove to trap
4864          * buggy users.
4865          */
4866         if (tp->urg_seq == tp->copied_seq && tp->urg_data &&
4867             !sock_flag(sk, SOCK_URGINLINE) && tp->copied_seq != tp->rcv_nxt) {
4868                 struct sk_buff *skb = skb_peek(&sk->sk_receive_queue);
4869                 tp->copied_seq++;
4870                 if (skb && !before(tp->copied_seq, TCP_SKB_CB(skb)->end_seq)) {
4871                         __skb_unlink(skb, &sk->sk_receive_queue);
4872                         __kfree_skb(skb);
4873                 }
4874         }
4875
4876         tp->urg_data = TCP_URG_NOTYET;
4877         tp->urg_seq = ptr;
4878
4879         /* Disable header prediction. */
4880         tp->pred_flags = 0;
4881 }
4882
4883 /* This is the 'fast' part of urgent handling. */
4884 static void tcp_urg(struct sock *sk, struct sk_buff *skb, const struct tcphdr *th)
4885 {
4886         struct tcp_sock *tp = tcp_sk(sk);
4887
4888         /* Check if we get a new urgent pointer - normally not. */
4889         if (th->urg)
4890                 tcp_check_urg(sk, th);
4891
4892         /* Do we wait for any urgent data? - normally not... */
4893         if (tp->urg_data == TCP_URG_NOTYET) {
4894                 u32 ptr = tp->urg_seq - ntohl(th->seq) + (th->doff * 4) -
4895                           th->syn;
4896
4897                 /* Is the urgent pointer pointing into this packet? */
4898                 if (ptr < skb->len) {
4899                         u8 tmp;
4900                         if (skb_copy_bits(skb, ptr, &tmp, 1))
4901                                 BUG();
4902                         tp->urg_data = TCP_URG_VALID | tmp;
4903                         if (!sock_flag(sk, SOCK_DEAD))
4904                                 sk->sk_data_ready(sk);
4905                 }
4906         }
4907 }
4908
```

```
4909 static int tcp_copy_to_iovec(struct sock *sk, struct sk_buff *skb, int hlen)
4910 {
4911         struct tcp_sock *tp = tcp_sk(sk);
4912         int chunk = skb->len - hlen;
4913         int err;
4914
4915         local_bh_enable();
4916         if (skb_csum_unnecessary(skb))
4917                 err = skb_copy_datagram_iovec(skb, hlen, tp->ucopy.iov, chunk);
4918         else
4919                 err = skb_copy_and_csum_datagram_iovec(skb, hlen,
4920                                                        tp->ucopy.iov);
4921
4922         if (!err) {
4923                 tp->ucopy.len -= chunk;
4924                 tp->copied_seq += chunk;
4925                 tcp_rcv_space_adjust(sk);
4926         }
4927
4928         local_bh_disable();
4929         return err;
4930 }
4931
4932 static __sum16 __tcp_checksum_complete_user(struct sock *sk,
4933                                             struct sk_buff *skb)
4934 {
4935         __sum16 result;
4936
4937         if (sock_owned_by_user(sk)) {
4938                 local_bh_enable();
4939                 result = __tcp_checksum_complete(skb);
4940                 local_bh_disable();
4941         } else {
4942                 result = __tcp_checksum_complete(skb);
4943         }
4944         return result;
4945 }
4946
4947 static inline bool tcp_checksum_complete_user(struct sock *sk,
4948                                               struct sk_buff *skb)
4949 {
4950         return !skb_csum_unnecessary(skb) &&
4951                __tcp_checksum_complete_user(sk, skb);
4952 }
4953
4954 #ifdef CONFIG_NET_DMA
4955 static bool tcp_dma_try_early_copy(struct sock *sk, struct sk_buff *skb,
4956                                    int hlen)
4957 {
4958         struct tcp_sock *tp = tcp_sk(sk);
4959         int chunk = skb->len - hlen;
4960         int dma_cookie;
4961         bool copied_early = false;
4962
4963         if (tp->ucopy.wakeup)
4964                 return false;
4965
4966         if (!tp->ucopy.dma_chan && tp->ucopy.pinned_list)
4967                 tp->ucopy.dma_chan = net_dma_find_channel();
4968
4969         if (tp->ucopy.dma_chan && skb_csum_unnecessary(skb)) {
4970
4971                 dma_cookie = dma_skb_copy_datagram_iovec(tp->ucopy.dma_chan,
4972                                                          skb, hlen,
4973                                                          tp->ucopy.iov, chunk,
4974                                                          tp->ucopy.pinned_list);
4975
4976                 if (dma_cookie < 0)
4977                         goto out;
4978
4979                 tp->ucopy.dma_cookie = dma_cookie;
4980                 copied_early = true;
4981
4982                 tp->ucopy.len -= chunk;
4983                 tp->copied_seq += chunk;
4984                 tcp_rcv_space_adjust(sk);
4985
4986                 if ((tp->ucopy.len == 0) ||
4987                     (tcp_flag_word(tcp_hdr(skb)) & TCP_FLAG_PSH) ||
4988                     (atomic_read(&sk->sk_rmem_alloc) > (sk->sk_rcvbuf >> 1))) {
4989                         tp->ucopy.wakeup = 1;
4990                         sk->sk_data_ready(sk);
4991                 }
4992         } else if (chunk > 0) {
4993                 tp->ucopy.wakeup = 1;
4994                 sk->sk_data_ready(sk);
4995         }
4996 out:
```

```
4997         return copied_early;
4998 }
4999 #endif /* CONFIG_NET_DMA */
5000
5001 /* Does PAWS and seqno based validation of an incoming segment, flags will
5002  * play significant role here.
5003  */
5004 static bool tcp_validate_incoming(struct sock *sk, struct sk_buff *skb,
5005                                   const struct tcphdr *th, int syn_inerr)
5006 {
5007         struct tcp_sock *tp = tcp_sk(sk);
5008
5009         /* RFC1323: H1. Apply PAWS check first. */
5010         if (tcp_fast_parse_options(skb, th, tp) && tp->rx_opt.saw_tstamp &&
5011             tcp_paws_discard(sk, skb)) {
5012                 if (!th->rst) {
5013                         NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_PAWSESTABREJECTED);
5014                         tcp_send_dupack(sk, skb);
5015                         goto discard;
5016                 }
5017                 /* Reset is accepted even if it did not pass PAWS. */
5018         }
5019
5020         /* Step 1: check sequence number */
5021         if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq)) {
5022                 /* RFC793, page 37: "In all states except SYN-SENT, all reset
5023                  * (RST) segments are validated by checking their SEQ-fields."
5024                  * And page 69: "If an incoming segment is not acceptable,
5025                  * an acknowledgment should be sent in reply (unless the RST
5026                  * bit is set, if so drop the segment and return)".
5027                  */
5028                 if (!th->rst) {
5029                         if (th->syn)
5030                                 goto syn_challenge;
5031                         tcp_send_dupack(sk, skb);
5032                 }
5033                 goto discard;
5034         }
5035
5036         /* Step 2: check RST bit */
5037         if (th->rst) {
5038                 /* RFC 5961 3.2 :
5039                  * If sequence number exactly matches RCV.NXT, then
5040                  *     RESET the connection
5041                  * else
5042                  *     Send a challenge ACK
5043                  */
5044                 if (TCP_SKB_CB(skb)->seq == tp->rcv_nxt)
5045                         tcp_reset(sk);
5046                 else
5047                         tcp_send_challenge_ack(sk);
5048                 goto discard;
5049         }
5050
5051         /* step 3: check security and precedence [ignored] */
5052
5053         /* step 4: Check for a SYN
5054          * RFC 5691 4.2 : Send a challenge ack
5055          */
5056         if (th->syn) {
5057 syn_challenge:
5058                 if (syn_inerr)
5059                         TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_INERRS);
5060                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPSYNCHALLENGE);
5061                 tcp_send_challenge_ack(sk);
5062                 goto discard;
5063         }
5064
5065         return true;
5066
5067 discard:
5068         __kfree_skb(skb);
5069         return false;
5070 }
5071
5072 /*
5073  *      TCP receive function for the ESTABLISHED state.
5074  *
5075  *      It is split into a fast path and a slow path. The fast path is
5076  *      disabled when:
5077  *      - A zero window was announced from us - zero window probing
5078  *        is only handled properly in the slow path.
5079  *      - Out of order segments arrived.
5080  *      - Urgent data is expected.
5081  *      - There is no buffer space left
5082  *      - Unexpected TCP flags/window values/header lengths are received
5083  *        (detected by checking the TCP header against pred_flags)
5084  *      - Data is sent in both directions. Fast path only supports pure senders
```

```
5085  *        or pure receivers (this means either the sequence number or the ack
5086  *        value must stay constant)
5087  *      - Unexpected TCP option.
5088  *
5089  *      When these conditions are not satisfied it drops into a standard
5090  *      receive procedure patterned after RFC793 to handle all cases.
5091  *      The first three cases are guaranteed by proper pred_flags setting,
5092  *      the rest is checked inline. Fast processing is turned on in
5093  *      tcp_data_queue when everything is OK.
5094  */
5095 void tcp_rcv_established(struct sock *sk, struct sk_buff *skb,
5096                          const struct tcphdr *th, unsigned int len)
5097 {
5098         struct tcp_sock *tp = tcp_sk(sk);
5099
5100         if (unlikely(sk->sk_rx_dst == NULL))
5101                 inet_csk(sk)->icsk_af_ops->sk_rx_dst_set(sk, skb);
5102         /*
5103          *      Header prediction.
5104          *      The code loosely follows the one in the famous
5105          *      "30 instruction TCP receive" Van Jacobson mail.
5106          *
5107          *      Van's trick is to deposit buffers into socket queue
5108          *      on a device interrupt, to call tcp_recv function
5109          *      on the receive process context and checksum and copy
5110          *      the buffer to user space. smart...
5111          *
5112          *      Our current scheme is not silly either but we take the
5113          *      extra cost of the net_bh soft interrupt processing...
5114          *      We do checksum and copy also but from device to kernel.
5115          */
5116
5117         tp->rx_opt.saw_tstamp = 0;
5118
5119         /*      pred_flags is 0xS?10 << 16 + snd_wnd
5120          *      if header_prediction is to be made
5121          *      'S' will always be tp->tcp_header_len >> 2
5122          *      '?' will be 0 for the fast path, otherwise pred_flags is 0 to
5123          *  turn it off (when there are holes in the receive
5124          *       space for instance)
5125          *      PSH flag is ignored.
5126          */
5127
5128         if ((tcp_flag_word(th) & TCP_HP_BITS) == tp->pred_flags &&
5129             TCP_SKB_CB(skb)->seq == tp->rcv_nxt &&
5130             !after(TCP_SKB_CB(skb)->ack_seq, tp->snd_nxt)) {
5131                 int tcp_header_len = tp->tcp_header_len;
5132
5133                 /* Timestamp header prediction: tcp_header_len
5134                  * is automatically equal to th->doff*4 due to pred_flags
5135                  * match.
5136                  */
5137
5138                 /* Check timestamp */
5139                 if (tcp_header_len == sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED) {
5140                         /* No? Slow path! */
5141                         if (!tcp_parse_aligned_timestamp(tp, th))
5142                                 goto slow_path;
5143
5144                         /* If PAWS failed, check it more carefully in slow path */
5145                         if ((s32)(tp->rx_opt.rcv_tsval - tp->rx_opt.ts_recent) < 0)
5146                                 goto slow_path;
5147
5148                         /* DO NOT update ts_recent here, if checksum fails
5149                          * and timestamp was corrupted part, it will result
5150                          * in a hung connection since we will drop all
5151                          * future packets due to the PAWS test.
5152                          */
5153                 }
5154
5155                 if (len <= tcp_header_len) {
5156                         /* Bulk data transfer: sender */
5157                         if (len == tcp_header_len) {
5158                                 /* Predicted packet is in window by definition.
5159                                  * seq == rcv_nxt and rcv_wup <= rcv_nxt.
5160                                  * Hence, check seq<=rcv_wup reduces to:
5161                                  */
5162                                 if (tcp_header_len ==
5163                                     (sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED) &&
5164                                     tp->rcv_nxt == tp->rcv_wup)
5165                                         tcp_store_ts_recent(tp);
5166
5167                                 /* We know that such packets are checksummed
5168                                  * on entry.
5169                                  */
5170                                 tcp_ack(sk, skb, 0);
5171                                 __kfree_skb(skb);
5172                                 tcp_data_snd_check(sk);
```

```
5173                            return;
5174                    } else { /* Header too small */
5175                            TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_INERRS);
5176                            goto discard;
5177                    }
5178            } else {
5179                    int eaten = 0;
5180                    int copied_early = 0;
5181                    bool fragstolen = false;
5182
5183                    if (tp->copied_seq == tp->rcv_nxt &&
5184                        len - tcp_header_len <= tp->ucopy.len) {
5185 #ifdef CONFIG_NET_DMA
5186                            if (tp->ucopy.task == current &&
5187                                sock_owned_by_user(sk) &&
5188                                tcp_dma_try_early_copy(sk, skb, tcp_header_len)) {
5189                                    copied_early = 1;
5190                                    eaten = 1;
5191                            }
5192 #endif
5193                            if (tp->ucopy.task == current &&
5194                                sock_owned_by_user(sk) && !copied_early) {
5195                                    __set_current_state(TASK_RUNNING);
5196
5197                                    if (!tcp_copy_to_iovec(sk, skb, tcp_header_len))
5198                                            eaten = 1;
5199                            }
5200                            if (eaten) {
5201                                    /* Predicted packet is in window by definition.
5202                                     * seq == rcv_nxt and rcv_wup <= rcv_nxt.
5203                                     * Hence, check seq<=rcv_wup reduces to:
5204                                     */
5205                                    if (tcp_header_len ==
5206                                        (sizeof(struct tcphdr) +
5207                                         TCPOLEN_TSTAMP_ALIGNED) &&
5208                                        tp->rcv_nxt == tp->rcv_wup)
5209                                            tcp_store_ts_recent(tp);
5210
5211                                    tcp_rcv_rtt_measure_ts(sk, skb);
5212
5213                                    __skb_pull(skb, tcp_header_len);
5214                                    tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;
5215                                    NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPHPHITSTOUSER);
5216                            }
5217                            if (copied_early)
5218                                    tcp_cleanup_rbuf(sk, skb->len);
5219                    }
5220                    if (!eaten) {
5221                            if (tcp_checksum_complete_user(sk, skb))
5222                                    goto csum_error;
5223
5224                            if ((int)skb->truesize > sk->sk_forward_alloc)
5225                                    goto step5;
5226
5227                            /* Predicted packet is in window by definition.
5228                             * seq == rcv_nxt and rcv_wup <= rcv_nxt.
5229                             * Hence, check seq<=rcv_wup reduces to:
5230                             */
5231                            if (tcp_header_len ==
5232                                (sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED) &&
5233                                tp->rcv_nxt == tp->rcv_wup)
5234                                    tcp_store_ts_recent(tp);
5235
5236                            tcp_rcv_rtt_measure_ts(sk, skb);
5237
5238                            NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPHPHITS);
5239
5240                            /* Bulk data transfer: receiver */
5241                            eaten = tcp_queue_rcv(sk, skb, tcp_header_len,
5242                                                  &fragstolen);
5243                    }
5244
5245                    tcp_event_data_recv(sk, skb);
5246
5247                    if (TCP_SKB_CB(skb)->ack_seq != tp->snd_una) {
5248                            /* Well, only one small jumplet in fast path... */
5249                            tcp_ack(sk, skb, FLAG_DATA);
5250                            tcp_data_snd_check(sk);
5251                            if (!inet_csk_ack_scheduled(sk))
5252                                    goto no_ack;
5253                    }
5254
5255                    if (!copied_early || tp->rcv_nxt != tp->rcv_wup)
5256                            __tcp_ack_snd_check(sk, 0);
5257 no_ack:
5258 #ifdef CONFIG_NET_DMA
5259                    if (copied_early)
5260                            __skb_queue_tail(&sk->sk_async_wait_queue, skb);
```

```
5261                        else
5262 #endif
5263                        if (eaten)
5264                                kfree_skb_partial(skb, fragstolen);
5265                        sk->sk_data_ready(sk);
5266                        return;
5267                }
5268        }
5269
5270 slow_path:
5271        if (len < (th->doff << 2) || tcp_checksum_complete_user(sk, skb))
5272                goto csum_error;
5273
5274        if (!th->ack && !th->rst)
5275                goto discard;
5276
5277        /*
5278         *      Standard slow path.
5279         */
5280
5281        if (!tcp_validate_incoming(sk, skb, th, 1))
5282                return;
5283
5284 step5:
5285        if (tcp_ack(sk, skb, FLAG_SLOWPATH | FLAG_UPDATE_TS_RECENT) < 0)
5286                goto discard;
5287
5288        tcp_rcv_rtt_measure_ts(sk, skb);
5289
5290        /* Process urgent data. */
5291        tcp_urg(sk, skb, th);
5292
5293        /* step 7: process the segment text */
5294        tcp_data_queue(sk, skb);
5295
5296        tcp_data_snd_check(sk);
5297        tcp_ack_snd_check(sk);
5298        return;
5299
5300 csum_error:
5301        TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_CSUMERRORS);
5302        TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_INERRS);
5303
5304 discard:
5305        __kfree_skb(skb);
5306 }
5307 EXPORT_SYMBOL(tcp_rcv_established);
5308
5309 void tcp_finish_connect(struct sock *sk, struct sk_buff *skb)
5310 {
5311        struct tcp_sock *tp = tcp_sk(sk);
5312        struct inet_connection_sock *icsk = inet_csk(sk);
5313
5314        tcp_set_state(sk, TCP_ESTABLISHED);
5315
5316        if (skb != NULL) {
5317                icsk->icsk_af_ops->sk_rx_dst_set(sk, skb);
5318                security_inet_conn_established(sk, skb);
5319        }
5320
5321        /* Make sure socket is routed, for correct metrics.  */
5322        icsk->icsk_af_ops->rebuild_header(sk);
5323
5324        tcp_init_metrics(sk);
5325
5326        tcp_init_congestion_control(sk);
5327
5328        /* Prevent spurious tcp_cwnd_restart() on first data
5329         * packet.
5330         */
5331        tp->lsndtime = tcp_time_stamp;
5332
5333        tcp_init_buffer_space(sk);
5334
5335        if (sock_flag(sk, SOCK_KEEPOPEN))
5336                inet_csk_reset_keepalive_timer(sk, keepalive_time_when(tp));
5337
5338        if (!tp->rx_opt.snd_wscale)
5339                __tcp_fast_path_on(tp, tp->snd_wnd);
5340        else
5341                tp->pred_flags = 0;
5342
5343        if (!sock_flag(sk, SOCK_DEAD)) {
5344                sk->sk_state_change(sk);
5345                sk_wake_async(sk, SOCK_WAKE_IO, POLL_OUT);
5346        }
5347 }
5348
```

```
5349 static bool tcp_rcv_fastopen_synack(struct sock *sk, struct sk_buff *synack,
5350                                       struct tcp_fastopen_cookie *cookie)
5351 {
5352         struct tcp_sock *tp = tcp_sk(sk);
5353         struct sk_buff *data = tp->syn_data ? tcp_write_queue_head(sk) : NULL;
5354         u16 mss = tp->rx_opt.mss_clamp;
5355         bool syn_drop;
5356
5357         if (mss == tp->rx_opt.user_mss) {
5358                 struct tcp_options_received opt;
5359
5360                 /* Get original SYNACK MSS value if user MSS sets mss_clamp */
5361                 tcp_clear_options(&opt);
5362                 opt.user_mss = opt.mss_clamp = 0;
5363                 tcp_parse_options(synack, &opt, 0, NULL);
5364                 mss = opt.mss_clamp;
5365         }
5366
5367         if (!tp->syn_fastopen)  /* Ignore an unsolicited cookie */
5368                 cookie->len = -1;
5369
5370         /* The SYN-ACK neither has cookie nor acknowledges the data. Presumably
5371          * the remote receives only the retransmitted (regular) SYNs: either
5372          * the original SYN-data or the corresponding SYN-ACK is lost.
5373          */
5374         syn_drop = (cookie->len <= 0 && data && tp->total_retrans);
5375
5376         tcp_fastopen_cache_set(sk, mss, cookie, syn_drop);
5377
5378         if (data) { /* Retransmit unacked data in SYN */
5379                 tcp_for_write_queue_from(data, sk) {
5380                         if (data == tcp_send_head(sk) ||
5381                             __tcp_retransmit_skb(sk, data))
5382                                 break;
5383                 }
5384                 tcp_rearm_rto(sk);
5385                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPFASTOPENACTIVEFAIL);
5386                 return true;
5387         }
5388         tp->syn_data_acked = tp->syn_data;
5389         if (tp->syn_data_acked)
5390                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPFASTOPENACTIVE);
5391         return false;
5392 }
5393
5394 static int tcp_rcv_synsent_state_process(struct sock *sk, struct sk_buff *skb,
5395                                          const struct tcphdr *th, unsigned int len)
5396 {
5397         struct inet_connection_sock *icsk = inet_csk(sk);
5398         struct tcp_sock *tp = tcp_sk(sk);
5399         struct tcp_fastopen_cookie foc = { .len = -1 };
5400         int saved_clamp = tp->rx_opt.mss_clamp;
5401
5402         tcp_parse_options(skb, &tp->rx_opt, 0, &foc);
5403         if (tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr)
5404                 tp->rx_opt.rcv_tsecr -= tp->tsoffset;
5405
5406         if (th->ack) {
5407                 /* rfc793:
5408                  * "If the state is SYN-SENT then
5409                  *    first check the ACK bit
5410                  *      If the ACK bit is set
5411                  *        If SEG.ACK =< ISS, or SEG.ACK > SND.NXT, send
5412                  *        a reset (unless the RST bit is set, if so drop
5413                  *        the segment and return)"
5414                  */
5415                 if (!after(TCP_SKB_CB(skb)->ack_seq, tp->snd_una) ||
5416                     after(TCP_SKB_CB(skb)->ack_seq, tp->snd_nxt))
5417                         goto reset_and_undo;
5418
5419                 if (tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr &&
5420                     !between(tp->rx_opt.rcv_tsecr, tp->retrans_stamp,
5421                              tcp_time_stamp)) {
5422                         NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_PAWSACTIVEREJECTED);
5423                         goto reset_and_undo;
5424                 }
5425
5426                 /* Now ACK is acceptable.
5427                  *
5428                  * "If the RST bit is set
5429                  *    If the ACK was acceptable then signal the user "error:
5430                  *    connection reset", drop the segment, enter CLOSED state,
5431                  *    delete TCB, and return."
5432                  */
5433
5434                 if (th->rst) {
5435                         tcp_reset(sk);
5436                         goto discard;
```

```
5437                }
5438
5439                /* rfc793:
5440                 *   "fifth, if neither of the SYN or RST bits is set then
5441                 *    drop the segment and return."
5442                 *
5443                 *    See note below!
5444                 *                                        --ANK(990513)
5445                 */
5446                if (!th->syn)
5447                        goto discard_and_undo;
5448
5449                /* rfc793:
5450                 *   "If the SYN bit is on ...
5451                 *    are acceptable then ...
5452                 *    (our SYN has been ACKed), change the connection
5453                 *    state to ESTABLISHED..."
5454                 */
5455
5456                TCP_ECN_rcv_synack(tp, th);
5457
5458                tcp_init_wl(tp, TCP_SKB_CB(skb)->seq);
5459                tcp_ack(sk, skb, FLAG_SLOWPATH);
5460
5461                /* Ok.. it's good. Set up sequence numbers and
5462                 * move to established.
5463                 */
5464                tp->rcv_nxt = TCP_SKB_CB(skb)->seq + 1;
5465                tp->rcv_wup = TCP_SKB_CB(skb)->seq + 1;
5466
5467                /* RFC1323: The window in SYN & SYN/ACK segments is
5468                 * never scaled.
5469                 */
5470                tp->snd_wnd = ntohs(th->window);
5471
5472                if (!tp->rx_opt.wscale_ok) {
5473                        tp->rx_opt.snd_wscale = tp->rx_opt.rcv_wscale = 0;
5474                        tp->window_clamp = min(tp->window_clamp, 65535U);
5475                }
5476
5477                if (tp->rx_opt.saw_tstamp) {
5478                        tp->rx_opt.tstamp_ok       = 1;
5479                        tp->tcp_header_len =
5480                                sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED;
5481                        tp->advmss          -= TCPOLEN_TSTAMP_ALIGNED;
5482                        tcp_store_ts_recent(tp);
5483                } else {
5484                        tp->tcp_header_len = sizeof(struct tcphdr);
5485                }
5486
5487                if (tcp_is_sack(tp) && sysctl_tcp_fack)
5488                        tcp_enable_fack(tp);
5489
5490                tcp_mtup_init(sk);
5491                tcp_sync_mss(sk, icsk->icsk_pmtu_cookie);
5492                tcp_initialize_rcv_mss(sk);
5493
5494                /* Remember, tcp_poll() does not lock socket!
5495                 * Change state from SYN-SENT only after copied_seq
5496                 * is initialized. */
5497                tp->copied_seq = tp->rcv_nxt;
5498
5499                smp_mb();
5500
5501                tcp_finish_connect(sk, skb);
5502
5503                if ((tp->syn_fastopen || tp->syn_data) &&
5504                    tcp_rcv_fastopen_synack(sk, skb, &foc))
5505                        return -1;
5506
5507                if (sk->sk_write_pending ||
5508                    icsk->icsk_accept_queue.rskq_defer_accept ||
5509                    icsk->icsk_ack.pingpong) {
5510                        /* Save one ACK. Data will be ready after
5511                         * several ticks, if write_pending is set.
5512                         *
5513                         * It may be deleted, but with this feature tcpdumps
5514                         * look so _wonderfully_ clever, that I was not able
5515                         * to stand against the temptation 8)     --ANK
5516                         */
5517                        inet_csk_schedule_ack(sk);
5518                        icsk->icsk_ack.lrcvtime = tcp_time_stamp;
5519                        tcp_enter_quickack_mode(sk);
5520                        inet_csk_reset_xmit_timer(sk, ICSK_TIME_DACK,
5521                                                  TCP_DELACK_MAX, TCP_RTO_MAX);
5522
5523 discard:
5524                        __kfree_skb(skb);
```

```
5525                    return 0;
5526            } else {
5527                    tcp_send_ack(sk);
5528            }
5529            return -1;
5530    }
5531
5532    /* No ACK in the segment */
5533
5534    if (th->rst) {
5535            /* rfc793:
5536             * "If the RST bit is set
5537             *
5538             *      Otherwise (no ACK) drop the segment and return."
5539             */
5540
5541            goto discard_and_undo;
5542    }
5543
5544    /* PAWS check. */
5545    if (tp->rx_opt.ts_recent_stamp && tp->rx_opt.saw_tstamp &&
5546        tcp_paws_reject(&tp->rx_opt, 0))
5547            goto discard_and_undo;
5548
5549    if (th->syn) {
5550            /* We see SYN without ACK. It is attempt of
5551             * simultaneous connect with crossed SYNs.
5552             * Particularly, it can be connect to self.
5553             */
5554            tcp_set_state(sk, TCP_SYN_RECV);
5555
5556            if (tp->rx_opt.saw_tstamp) {
5557                    tp->rx_opt.tstamp_ok = 1;
5558                    tcp_store_ts_recent(tp);
5559                    tp->tcp_header_len =
5560                            sizeof(struct tcphdr) + TCPOLEN_TSTAMP_ALIGNED;
5561            } else {
5562                    tp->tcp_header_len = sizeof(struct tcphdr);
5563            }
5564
5565            tp->rcv_nxt = TCP_SKB_CB(skb)->seq + 1;
5566            tp->rcv_wup = TCP_SKB_CB(skb)->seq + 1;
5567
5568            /* RFC1323: The window in SYN & SYN/ACK segments is
5569             * never scaled.
5570             */
5571            tp->snd_wnd    = ntohs(th->window);
5572            tp->snd_wl1    = TCP_SKB_CB(skb)->seq;
5573            tp->max_window = tp->snd_wnd;
5574
5575            TCP_ECN_rcv_syn(tp, th);
5576
5577            tcp_mtup_init(sk);
5578            tcp_sync_mss(sk, icsk->icsk_pmtu_cookie);
5579            tcp_initialize_rcv_mss(sk);
5580
5581            tcp_send_synack(sk);
5582 #if 0
5583            /* Note, we could accept data and URG from this segment.
5584             * There are no obstacles to make this (except that we must
5585             * either change tcp_recvmsg() to prevent it from returning data
5586             * before 3WHS completes per RFC793, or employ TCP Fast Open).
5587             *
5588             * However, if we ignore data in ACKless segments sometimes,
5589             * we have no reasons to accept it sometimes.
5590             * Also, seems the code doing it in step6 of tcp_rcv_state_process
5591             * is not flawless. So, discard packet for sanity.
5592             * Uncomment this return to process the data.
5593             */
5594            return -1;
5595 #else
5596            goto discard;
5597 #endif
5598    }
5599    /* "fifth, if neither of the SYN or RST bits is set then
5600     * drop the segment and return."
5601     */
5602
5603 discard_and_undo:
5604    tcp_clear_options(&tp->rx_opt);
5605    tp->rx_opt.mss_clamp = saved_clamp;
5606    goto discard;
5607
5608 reset_and_undo:
5609    tcp_clear_options(&tp->rx_opt);
5610    tp->rx_opt.mss_clamp = saved_clamp;
5611    return 1;
5612 }
```

```
5613
5614 /*
5615  *      This function implements the receiving procedure of RFC 793 for
5616  *      all states except ESTABLISHED and TIME_WAIT.
5617  *      It's called from both tcp_v4_rcv and tcp_v6_rcv and should be
5618  *      address independent.
5619  */
5620
5621 int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb,
5622                           const struct tcphdr *th, unsigned int len)
5623 {
5624         struct tcp_sock *tp = tcp_sk(sk);
5625         struct inet_connection_sock *icsk = inet_csk(sk);
5626         struct request_sock *req;
5627         int queued = 0;
5628         bool acceptable;
5629         u32 synack_stamp;
5630
5631         tp->rx_opt.saw_tstamp = 0;
5632
5633         switch (sk->sk_state) {
5634         case TCP_CLOSE:
5635                 goto discard;
5636
5637         case TCP_LISTEN:
5638                 if (th->ack)
5639                         return 1;
5640
5641                 if (th->rst)
5642                         goto discard;
5643
5644                 if (th->syn) {
5645                         if (th->fin)
5646                                 goto discard;
5647                         if (icsk->icsk_af_ops->conn_request(sk, skb) < 0)
5648                                 return 1;
5649
5650                         /* Now we have several options: In theory there is
5651                          * nothing else in the frame. KA9Q has an option to
5652                          * send data with the syn, BSD accepts data with the
5653                          * syn up to the [to be] advertised window and
5654                          * Solaris 2.1 gives you a protocol error. For now
5655                          * we just ignore it, that fits the spec precisely
5656                          * and avoids incompatibilities. It would be nice in
5657                          * future to drop through and process the data.
5658                          *
5659                          * Now that TTCP is starting to be used we ought to
5660                          * queue this data.
5661                          * But, this leaves one open to an easy denial of
5662                          * service attack, and SYN cookies can't defend
5663                          * against this problem. So, we drop the data
5664                          * in the interest of security over speed unless
5665                          * it's still in use.
5666                          */
5667                         kfree_skb(skb);
5668                         return 0;
5669                 }
5670                 goto discard;
5671
5672         case TCP_SYN_SENT:
5673                 queued = tcp_rcv_synsent_state_process(sk, skb, th, len);
5674                 if (queued >= 0)
5675                         return queued;
5676
5677                 /* Do step6 onward by hand. */
5678                 tcp_urg(sk, skb, th);
5679                 __kfree_skb(skb);
5680                 tcp_data_snd_check(sk);
5681                 return 0;
5682         }
5683
5684         req = tp->fastopen_rsk;
5685         if (req != NULL) {
5686                 WARN_ON_ONCE(sk->sk_state != TCP_SYN_RECV &&
5687                     sk->sk_state != TCP_FIN_WAIT1);
5688
5689                 if (tcp_check_req(sk, skb, req, NULL, true) == NULL)
5690                         goto discard;
5691         }
5692
5693         if (!th->ack && !th->rst)
5694                 goto discard;
5695
5696         if (!tcp_validate_incoming(sk, skb, th, 0))
5697                 return 0;
5698
5699         /* step 5: check the ACK field */
5700         acceptable = tcp_ack(sk, skb, FLAG_SLOWPATH |
```

```
5701                                    FLAG_UPDATE_TS_RECENT) > 0;
5702
5703        switch (sk->sk_state) {
5704        case TCP_SYN_RECV:
5705                if (!acceptable)
5706                        return 1;
5707
5708                /* Once we leave TCP_SYN_RECV, we no longer need req
5709                 * so release it.
5710                 */
5711                if (req) {
5712                        synack_stamp = tcp_rsk(req)->snt_synack;
5713                        tp->total_retrans = req->num_retrans;
5714                        reqsk_fastopen_remove(sk, req, false);
5715                } else {
5716                        synack_stamp = tp->lsndtime;
5717                        /* Make sure socket is routed, for correct metrics. */
5718                        icsk->icsk_af_ops->rebuild_header(sk);
5719                        tcp_init_congestion_control(sk);
5720
5721                        tcp_mtup_init(sk);
5722                        tp->copied_seq = tp->rcv_nxt;
5723                        tcp_init_buffer_space(sk);
5724                }
5725                smp_mb();
5726                tcp_set_state(sk, TCP_ESTABLISHED);
5727                sk->sk_state_change(sk);
5728
5729                /* Note, that this wakeup is only for marginal crossed SYN case.
5730                 * Passively open sockets are not waked up, because
5731                 * sk->sk_sleep == NULL and sk->sk_socket == NULL.
5732                 */
5733                if (sk->sk_socket)
5734                        sk_wake_async(sk, SOCK_WAKE_IO, POLL_OUT);
5735
5736                tp->snd_una = TCP_SKB_CB(skb)->ack_seq;
5737                tp->snd_wnd = ntohs(th->window) << tp->rx_opt.snd_wscale;
5738                tcp_init_wl(tp, TCP_SKB_CB(skb)->seq);
5739                tcp_synack_rtt_meas(sk, synack_stamp);
5740
5741                if (tp->rx_opt.tstamp_ok)
5742                        tp->advmss -= TCPOLEN_TSTAMP_ALIGNED;
5743
5744                if (req) {
5745                        /* Re-arm the timer because data may have been sent out.
5746                         * This is similar to the regular data transmission case
5747                         * when new data has just been ack'ed.
5748                         *
5749                         * (TFO) - we could try to be more aggressive and
5750                         * retransmitting any data sooner based on when they
5751                         * are sent out.
5752                         */
5753                        tcp_rearm_rto(sk);
5754                } else
5755                        tcp_init_metrics(sk);
5756
5757                tcp_update_pacing_rate(sk);
5758
5759                /* Prevent spurious tcp_cwnd_restart() on first data packet */
5760                tp->lsndtime = tcp_time_stamp;
5761
5762                tcp_initialize_rcv_mss(sk);
5763                tcp_fast_path_on(tp);
5764                break;
5765
5766        case TCP_FIN_WAIT1: {
5767                struct dst_entry *dst;
5768                int tmo;
5769
5770                /* If we enter the TCP_FIN_WAIT1 state and we are a
5771                 * Fast Open socket and this is the first acceptable
5772                 * ACK we have received, this would have acknowledged
5773                 * our SYNACK so stop the SYNACK timer.
5774                 */
5775                if (req != NULL) {
5776                        /* Return RST if ack_seq is invalid.
5777                         * Note that RFC793 only says to generate a
5778                         * DUPACK for it but for TCP Fast Open it seems
5779                         * better to treat this case like TCP_SYN_RECV
5780                         * above.
5781                         */
5782                        if (!acceptable)
5783                                return 1;
5784                        /* We no longer need the request sock. */
5785                        reqsk_fastopen_remove(sk, req, false);
5786                        tcp_rearm_rto(sk);
5787                }
5788                if (tp->snd_una != tp->write_seq)
```

```
5789                    break;
5790
5791            tcp_set_state(sk, TCP_FIN_WAIT2);
5792            sk->sk_shutdown |= SEND_SHUTDOWN;
5793
5794            dst = __sk_dst_get(sk);
5795            if (dst)
5796                    dst_confirm(dst);
5797
5798            if (!sock_flag(sk, SOCK_DEAD)) {
5799                    /* Wake up lingering close() */
5800                    sk->sk_state_change(sk);
5801                    break;
5802            }
5803
5804            if (tp->linger2 < 0 ||
5805                (TCP_SKB_CB(skb)->end_seq != TCP_SKB_CB(skb)->seq &&
5806                 after(TCP_SKB_CB(skb)->end_seq - th->fin, tp->rcv_nxt))) {
5807                    tcp_done(sk);
5808                    NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPABORTONDATA);
5809                    return 1;
5810            }
5811
5812            tmo = tcp_fin_time(sk);
5813            if (tmo > TCP_TIMEWAIT_LEN) {
5814                    inet_csk_reset_keepalive_timer(sk, tmo - TCP_TIMEWAIT_LEN);
5815            } else if (th->fin || sock_owned_by_user(sk)) {
5816                    /* Bad case. We could lose such FIN otherwise.
5817                     * It is not a big problem, but it looks confusing
5818                     * and not so rare event. We still can lose it now,
5819                     * if it spins in bh_lock_sock(), but it is really
5820                     * marginal case.
5821                     */
5822                    inet_csk_reset_keepalive_timer(sk, tmo);
5823            } else {
5824                    tcp_time_wait(sk, TCP_FIN_WAIT2, tmo);
5825                    goto discard;
5826            }
5827            break;
5828    }
5829
5830    case TCP_CLOSING:
5831            if (tp->snd_una == tp->write_seq) {
5832                    tcp_time_wait(sk, TCP_TIME_WAIT, 0);
5833                    goto discard;
5834            }
5835            break;
5836
5837    case TCP_LAST_ACK:
5838            if (tp->snd_una == tp->write_seq) {
5839                    tcp_update_metrics(sk);
5840                    tcp_done(sk);
5841                    goto discard;
5842            }
5843            break;
5844    }
5845
5846    /* step 6: check the URG bit */
5847    tcp_urg(sk, skb, th);
5848
5849    /* step 7: process the segment text */
5850    switch (sk->sk_state) {
5851    case TCP_CLOSE_WAIT:
5852    case TCP_CLOSING:
5853    case TCP_LAST_ACK:
5854            if (!before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt))
5855                    break;
5856    case TCP_FIN_WAIT1:
5857    case TCP_FIN_WAIT2:
5858            /* RFC 793 says to queue data in these states,
5859             * RFC 1122 says we MUST send a reset.
5860             * BSD 4.4 also does reset.
5861             */
5862            if (sk->sk_shutdown & RCV_SHUTDOWN) {
5863                    if (TCP_SKB_CB(skb)->end_seq != TCP_SKB_CB(skb)->seq &&
5864                        after(TCP_SKB_CB(skb)->end_seq - th->fin, tp->rcv_nxt)) {
5865                            NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPABORTONDATA);
5866                            tcp_reset(sk);
5867                            return 1;
5868                    }
5869            }
5870            /* Fall through */
5871    case TCP_ESTABLISHED:
5872            tcp_data_queue(sk, skb);
5873            queued = 1;
5874            break;
5875    }
5876
```

```
5877         /* tcp_data could move socket to TIME-WAIT */
5878         if (sk->sk_state != TCP_CLOSE) {
5879                 tcp_data_snd_check(sk);
5880                 tcp_ack_snd_check(sk);
5881         }
5882
5883         if (!queued) {
5884 discard:
5885                 __kfree_skb(skb);
5886         }
5887         return 0;
5888 }
5889 EXPORT_SYMBOL(tcp_rcv_state_process);
5890
5891 static inline void pr_drop_req(struct request_sock *req, __u16 port, int family)
5892 {
5893         struct inet_request_sock *ireq = inet_rsk(req);
5894
5895         if (family == AF_INET)
5896                 LIMIT_NETDEBUG(KERN_DEBUG pr_fmt("drop open request from %pI4/%u\n"),
5897                                &ireq->ir_rmt_addr, port);
5898 #if IS_ENABLED(CONFIG_IPV6)
5899         else if (family == AF_INET6)
5900                 LIMIT_NETDEBUG(KERN_DEBUG pr_fmt("drop open request from %pI6/%u\n"),
5901                                &ireq->ir_v6_rmt_addr, port);
5902 #endif
5903 }
5904
5905 int tcp_conn_request(struct request_sock_ops *rsk_ops,
5906                      const struct tcp_request_sock_ops *af_ops,
5907                      struct sock *sk, struct sk_buff *skb)
5908 {
5909         struct tcp_options_received tmp_opt;
5910         struct request_sock *req;
5911         struct tcp_sock *tp = tcp_sk(sk);
5912         struct dst_entry *dst = NULL;
5913         __u32 isn = TCP_SKB_CB(skb)->when;
5914         bool want_cookie = false, fastopen;
5915         struct flowi fl;
5916         struct tcp_fastopen_cookie foc = { .len = -1 };
5917         int err;
5918
5919
5920         /* TW buckets are converted to open requests without
5921          * limitations, they conserve resources and peer is
5922          * evidently real one.
5923          */
5924         if ((sysctl_tcp_syncookies == 2 ||
5925              inet_csk_reqsk_queue_is_full(sk)) && !isn) {
5926                 want_cookie = tcp_syn_flood_action(sk, skb, rsk_ops->slab_name);
5927                 if (!want_cookie)
5928                         goto drop;
5929         }
5930
5931
5932         /* Accept backlog is full. If we have already queued enough
5933          * of warm entries in syn queue, drop request. It is better than
5934          * clogging syn queue with openreqs with exponentially increasing
5935          * timeout.
5936          */
5937         if (sk_acceptq_is_full(sk) && inet_csk_reqsk_queue_young(sk) > 1) {
5938                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENOVERFLOWS);
5939                 goto drop;
5940         }
5941
5942         req = inet_reqsk_alloc(rsk_ops);
5943         if (!req)
5944                 goto drop;
5945
5946         tcp_rsk(req)->af_specific = af_ops;
5947
5948         tcp_clear_options(&tmp_opt);
5949         tmp_opt.mss_clamp = af_ops->mss_clamp;
5950         tmp_opt.user_mss  = tp->rx_opt.user_mss;
5951         tcp_parse_options(skb, &tmp_opt, 0, want_cookie ? NULL : &foc);
5952
5953         if (want_cookie && !tmp_opt.saw_tstamp)
5954                 tcp_clear_options(&tmp_opt);
5955
5956         tmp_opt.tstamp_ok = tmp_opt.saw_tstamp;
5957         tcp_openreq_init(req, &tmp_opt, skb, sk);
5958
5959         af_ops->init_req(req, sk, skb);
5960
5961         if (security_inet_conn_request(sk, skb, req))
5962                 goto drop_and_free;
5963
5964         if (!want_cookie || tmp_opt.tstamp_ok)
```

```
5965                TCP_ECN_create_request(req, skb, sock_net(sk));
5966
5967        if (want_cookie) {
5968                isn = cookie_init_sequence(af_ops, sk, skb, &req->mss);
5969                req->cookie_ts = tmp_opt.tstamp_ok;
5970        } else if (!isn) {
5971                /* VJ's idea. We save last timestamp seen
5972                 * from the destination in peer table, when entering
5973                 * state TIME-WAIT, and check against it before
5974                 * accepting new connection request.
5975                 *
5976                 * If "isn" is not zero, this request hit alive
5977                 * timewait bucket, so that all the necessary checks
5978                 * are made in the function processing timewait state.
5979                 */
5980                if (tcp_death_row.sysctl_tw_recycle) {
5981                        bool strict;
5982
5983                        dst = af_ops->route_req(sk, &fl, req, &strict);
5984
5985                        if (dst && strict &&
5986                            !tcp_peer_is_proven(req, dst, true,
5987                                                tmp_opt.saw_tstamp)) {
5988                                NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_PAWSPASSIVEREJECTED);
5989                                goto drop_and_release;
5990                        }
5991                }
5992                /* Kill the following clause, if you dislike this way. */
5993                else if (!sysctl_tcp_syncookies &&
5994                         (sysctl_max_syn_backlog - inet_csk_reqsk_queue_len(sk) <
5995                          (sysctl_max_syn_backlog >> 2)) &&
5996                         !tcp_peer_is_proven(req, dst, false,
5997                                             tmp_opt.saw_tstamp)) {
5998                        /* Without syncookies last quarter of
5999                         * backlog is filled with destinations,
6000                         * proven to be alive.
6001                         * It means that we continue to communicate
6002                         * to destinations, already remembered
6003                         * to the moment of synflood.
6004                         */
6005                        pr_drop_req(req, ntohs(tcp_hdr(skb)->source),
6006                                    rsk_ops->family);
6007                        goto drop_and_release;
6008                }
6009
6010                isn = af_ops->init_seq(skb);
6011        }
6012        if (!dst) {
6013                dst = af_ops->route_req(sk, &fl, req, NULL);
6014                if (!dst)
6015                        goto drop_and_free;
6016        }
6017
6018        tcp_rsk(req)->snt_isn = isn;
6019        tcp_openreq_init_rwin(req, sk, dst);
6020        fastopen = !want_cookie &&
6021                   tcp_try_fastopen(sk, skb, req, &foc, dst);
6022        err = af_ops->send_synack(sk, dst, &fl, req,
6023                                  skb_get_queue_mapping(skb), &foc);
6024        if (!fastopen) {
6025                if (err || want_cookie)
6026                        goto drop_and_free;
6027
6028                tcp_rsk(req)->listener = NULL;
6029                af_ops->queue_hash_add(sk, req, TCP_TIMEOUT_INIT);
6030        }
6031
6032        return 0;
6033
6034 drop_and_release:
6035        dst_release(dst);
6036 drop_and_free:
6037        reqsk_free(req);
6038 drop:
6039        NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENDROPS);
6040        return 0;
6041 }
6042 EXPORT_SYMBOL(tcp_conn_request);
6043
```