Linux Cross Reference

Free Electrons

Embedded Linux Experts

Linux/net/ipv4/tcp_output.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:     Pedro Roque     :       Retransmit queue handled by TCP.
23  *                              :       Fragmentation on mtu decrease
24  *                              :       Segment collapse on retransmit
25  *                              :       AF independence
26  *
27  *              Linus Torvalds  :       send_delayed_ack
28  *              David S. Miller :       Charge memory using the right skb
29  *                                      during syn/ack processing.
30  *              David S. Miller :       Output engine completely rewritten.
31  *              Andrea Arcangeli:       SYNACK carry ts_recent in tsecr.
32  *              Cacophonix Gaul :       draft-minshall-nagle-01
33  *              J Hadi Salim    :       ECN support
34  *
35  */
36
37 #define pr_fmt(fmt) "TCP: " fmt
38
39 #include <net/tcp.h>
40
41 #include <linux/compiler.h>
42 #include <linux/gfp.h>
43 #include <linux/module.h>
44
45 /* People can turn this off for buggy TCP's found in printers etc. */
46 int sysctl_tcp_retrans_collapse __read_mostly = 1;
47
48 /* People can turn this on to work with those rare, broken TCPs that
49  * interpret the window field as a signed quantity.
50  */
51 int sysctl_tcp_workaround_signed_windows __read_mostly = 0;
52
53 /* Default TSQ limit of two TSO segments */
54 int sysctl_tcp_limit_output_bytes __read_mostly = 131072;
55
56 /* This limits the percentage of the congestion window which we
57  * will allow a single TSO frame to consume.  Building TSO frames
58  * which are too large can cause TCP streams to be bursty.
59  */
60 int sysctl_tcp_tso_win_divisor __read_mostly = 3;
61
62 int sysctl_tcp_mtu_probing __read_mostly = 0;
63 int sysctl_tcp_base_mss __read_mostly = TCP_BASE_MSS;
64
65 /* By default, RFC2861 behavior.  */
66 int sysctl_tcp_slow_start_after_idle __read_mostly = 1;
67
68 unsigned int sysctl_tcp_notsent_lowat __read_mostly = UINT_MAX;
69 EXPORT_SYMBOL(sysctl_tcp_notsent_lowat);
70
```

```
 71 static bool tcp_write_xmit(struct sock *sk, unsigned int mss_now, int nonagle,
 72                            int push_one, gfp_t gfp);
 73
 74 /* Account for new data that has been sent to the network. */
 75 static void tcp_event_new_data_sent(struct sock *sk, const struct sk_buff *skb)
 76 {
 77         struct inet_connection_sock *icsk = inet_csk(sk);
 78         struct tcp_sock *tp = tcp_sk(sk);
 79         unsigned int prior_packets = tp->packets_out;
 80
 81         tcp_advance_send_head(sk, skb);
 82         tp->snd_nxt = TCP_SKB_CB(skb)->end_seq;
 83
 84         tp->packets_out += tcp_skb_pcount(skb);
 85         if (!prior_packets || icsk->icsk_pending == ICSK_TIME_EARLY_RETRANS ||
 86             icsk->icsk_pending == ICSK_TIME_LOSS_PROBE) {
 87                 tcp_rearm_rto(sk);
 88         }
 89
 90         NET_ADD_STATS(sock_net(sk), LINUX_MIB_TCPORIGDATASENT,
 91                       tcp_skb_pcount(skb));
 92 }
 93
 94 /* SND.NXT, if window was not shrunk.
 95  * If window has been shrunk, what should we make? It is not clear at all.
 96  * Using SND.UNA we will fail to open window, SND.NXT is out of window. :-(
 97  * Anything in between SND.UNA...SND.UNA+SND.WND also can be already
 98  * invalid. OK, let's make this for now:
 99  */
100 static inline __u32 tcp_acceptable_seq(const struct sock *sk)
101 {
102         const struct tcp_sock *tp = tcp_sk(sk);
103
104         if (!before(tcp_wnd_end(tp), tp->snd_nxt))
105                 return tp->snd_nxt;
106         else
107                 return tcp_wnd_end(tp);
108 }
109
110 /* Calculate mss to advertise in SYN segment.
111  * RFC1122, RFC1063, draft-ietf-tcpimpl-pmtud-01 state that:
112  *
113  * 1. It is independent of path mtu.
114  * 2. Ideally, it is maximal possible segment size i.e. 65535-40.
115  * 3. For IPv4 it is reasonable to calculate it from maximal MTU of
116  *    attached devices, because some buggy hosts are confused by
117  *    large MSS.
118  * 4. We do not make 3, we advertise MSS, calculated from first
119  *    hop device mtu, but allow to raise it to ip_rt_min_advmss.
120  *    This may be overridden via information stored in routing table.
121  * 5. Value 65535 for MSS is valid in IPv6 and means "as large as possible,
122  *    probably even Jumbo".
123  */
124 static __u16 tcp_advertise_mss(struct sock *sk)
125 {
126         struct tcp_sock *tp = tcp_sk(sk);
127         const struct dst_entry *dst = __sk_dst_get(sk);
128         int mss = tp->advmss;
129
130         if (dst) {
131                 unsigned int metric = dst_metric_advmss(dst);
132
133                 if (metric < mss) {
134                         mss = metric;
135                         tp->advmss = mss;
136                 }
137         }
138
139         return (__u16)mss;
140 }
141
142 /* RFC2861. Reset CWND after idle period longer RTO to "restart window".
143  * This is the first part of cwnd validation mechanism. */
144 static void tcp_cwnd_restart(struct sock *sk, const struct dst_entry *dst)
145 {
146         struct tcp_sock *tp = tcp_sk(sk);
147         s32 delta = tcp_time_stamp - tp->lsndtime;
148         u32 restart_cwnd = tcp_init_cwnd(tp, dst);
149         u32 cwnd = tp->snd_cwnd;
150
151         tcp_ca_event(sk, CA_EVENT_CWND_RESTART);
152
153         tp->snd_ssthresh = tcp_current_ssthresh(sk);
154         restart_cwnd = min(restart_cwnd, cwnd);
155
156         while ((delta -= inet_csk(sk)->icsk_rto) > 0 && cwnd > restart_cwnd)
157                 cwnd >>= 1;
158         tp->snd_cwnd = max(cwnd, restart_cwnd);
159         tp->snd_cwnd_stamp = tcp_time_stamp;
160         tp->snd_cwnd_used = 0;
```

```
161 }
162
163 /* Congestion state accounting after a packet has been sent. */
164 static void tcp_event_data_sent(struct tcp_sock *tp,
165                                 struct sock *sk)
166 {
167         struct inet_connection_sock *icsk = inet_csk(sk);
168         const u32 now = tcp_time_stamp;
169         const struct dst_entry *dst = __sk_dst_get(sk);
170
171         if (sysctl_tcp_slow_start_after_idle &&
172             (!tp->packets_out && (s32)(now - tp->lsndtime) > icsk->icsk_rto))
173                 tcp_cwnd_restart(sk, __sk_dst_get(sk));
174
175         tp->lsndtime = now;
176
177         /* If it is a reply for ato after last received
178          * packet, enter pingpong mode.
179          */
180         if ((u32)(now - icsk->icsk_ack.lrcvtime) < icsk->icsk_ack.ato &&
181             (!dst || !dst_metric(dst, RTAX_QUICKACK)))
182                         icsk->icsk_ack.pingpong = 1;
183 }
184
185 /* Account for an ACK we sent. */
186 static inline void tcp_event_ack_sent(struct sock *sk, unsigned int pkts)
187 {
188         tcp_dec_quickack_mode(sk, pkts);
189         inet_csk_clear_xmit_timer(sk, ICSK_TIME_DACK);
190 }
191
192
193 u32 tcp_default_init_rwnd(u32 mss)
194 {
195         /* Initial receive window should be twice of TCP_INIT_CWND to
196          * enable proper sending of new unsent data during fast recovery
197          * (RFC 3517, Section 4, NextSeg() rule (2)). Further place a
198          * limit when mss is larger than 1460.
199          */
200         u32 init_rwnd = TCP_INIT_CWND * 2;
201
202         if (mss > 1460)
203                 init_rwnd = max((1460 * init_rwnd) / mss, 2U);
204         return init_rwnd;
205 }
206
207 /* Determine a window scaling and initial window to offer.
208  * Based on the assumption that the given amount of space
209  * will be offered. Store the results in the tp structure.
210  * NOTE: for smooth operation initial space offering should
211  * be a multiple of mss if possible. We assume here that mss >= 1.
212  * This MUST be enforced by all callers.
213  */
214 void tcp_select_initial_window(int __space, __u32 mss,
215                                __u32 *rcv_wnd, __u32 *window_clamp,
216                                int wscale_ok, __u8 *rcv_wscale,
217                                __u32 init_rcv_wnd)
218 {
219         unsigned int space = (__space < 0 ? 0 : __space);
220
221         /* If no clamp set the clamp to the max possible scaled window */
222         if (*window_clamp == 0)
223                 (*window_clamp) = (65535 << 14);
224         space = min(*window_clamp, space);
225
226         /* Quantize space offering to a multiple of mss if possible. */
227         if (space > mss)
228                 space = (space / mss) * mss;
229
230         /* NOTE: offering an initial window larger than 32767
231          * will break some buggy TCP stacks. If the admin tells us
232          * it is likely we could be speaking with such a buggy stack
233          * we will truncate our initial window offering to 32K-1
234          * unless the remote has sent us a window scaling option,
235          * which we interpret as a sign the remote TCP is not
236          * misinterpreting the window field as a signed quantity.
237          */
238         if (sysctl_tcp_workaround_signed_windows)
239                 (*rcv_wnd) = min(space, MAX_TCP_WINDOW);
240         else
241                 (*rcv_wnd) = space;
242
243         (*rcv_wscale) = 0;
244         if (wscale_ok) {
245                 /* Set window scaling on max possible window
246                  * See RFC1323 for an explanation of the limit to 14
247                  */
248                 space = max_t(u32, sysctl_tcp_rmem[2], sysctl_rmem_max);
249                 space = min_t(u32, space, *window_clamp);
250                 while (space > 65535 && (*rcv_wscale) < 14) {
```

```
251                         space >>= 1;
252                         (*rcv_wscale)++;
253                 }
254         }
255
256         if (mss > (1 << *rcv_wscale)) {
257                 if (!init_rcv_wnd) /* Use default unless specified otherwise */
258                         init_rcv_wnd = tcp_default_init_rwnd(mss);
259                 *rcv_wnd = min(*rcv_wnd, init_rcv_wnd * mss);
260         }
261
262         /* Set the clamp no higher than max representable value */
263         (*window_clamp) = min(65535U << (*rcv_wscale), *window_clamp);
264 }
265 EXPORT_SYMBOL(tcp_select_initial_window);
266
267 /* Chose a new window to advertise, update state in tcp_sock for the
268  * socket, and return result with RFC1323 scaling applied.  The return
269  * value can be stuffed directly into th->window for an outgoing
270  * frame.
271  */
272 static u16 tcp_select_window(struct sock *sk)
273 {
274         struct tcp_sock *tp = tcp_sk(sk);
275         u32 old_win = tp->rcv_wnd;
276         u32 cur_win = tcp_receive_window(tp);
277         u32 new_win = __tcp_select_window(sk);
278
279         /* Never shrink the offered window */
280         if (new_win < cur_win) {
281                 /* Danger Will Robinson!
282                  * Don't update rcv_wup/rcv_wnd here or else
283                  * we will not be able to advertise a zero
284                  * window in time.  --DaveM
285                  *
286                  * Relax Will Robinson.
287                  */
288                 if (new_win == 0)
289                         NET_INC_STATS(sock_net(sk),
290                                       LINUX_MIB_TCPWANTZEROWINDOWADV);
291                 new_win = ALIGN(cur_win, 1 << tp->rx_opt.rcv_wscale);
292         }
293         tp->rcv_wnd = new_win;
294         tp->rcv_wup = tp->rcv_nxt;
295
296         /* Make sure we do not exceed the maximum possible
297          * scaled window.
298          */
299         if (!tp->rx_opt.rcv_wscale && sysctl_tcp_workaround_signed_windows)
300                 new_win = min(new_win, MAX_TCP_WINDOW);
301         else
302                 new_win = min(new_win, (65535U << tp->rx_opt.rcv_wscale));
303
304         /* RFC1323 scaling applied */
305         new_win >>= tp->rx_opt.rcv_wscale;
306
307         /* If we advertise zero window, disable fast path. */
308         if (new_win == 0) {
309                 tp->pred_flags = 0;
310                 if (old_win)
311                         NET_INC_STATS(sock_net(sk),
312                                       LINUX_MIB_TCPTOZEROWINDOWADV);
313         } else if (old_win == 0) {
314                 NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPFROMZEROWINDOWADV);
315         }
316
317         return new_win;
318 }
319
320 /* Packet ECN state for a SYN-ACK */
321 static inline void TCP_ECN_send_synack(const struct tcp_sock *tp, struct sk_buff *skb)
322 {
323         TCP_SKB_CB(skb)->tcp_flags &= ~TCPHDR_CWR;
324         if (!(tp->ecn_flags & TCP_ECN_OK))
325                 TCP_SKB_CB(skb)->tcp_flags &= ~TCPHDR_ECE;
326 }
327
328 /* Packet ECN state for a SYN.  */
329 static inline void TCP_ECN_send_syn(struct sock *sk, struct sk_buff *skb)
330 {
331         struct tcp_sock *tp = tcp_sk(sk);
332
333         tp->ecn_flags = 0;
334         if (sock_net(sk)->ipv4.sysctl_tcp_ecn == 1) {
335                 TCP_SKB_CB(skb)->tcp_flags |= TCPHDR_ECE | TCPHDR_CWR;
336                 tp->ecn_flags = TCP_ECN_OK;
337         }
338 }
339
340 static __inline__ void
```

```
341 TCP_ECN_make_synack(const struct request_sock *req, struct tcphdr *th)
342 {
343         if (inet_rsk(req)->ecn_ok)
344                 th->ece = 1;
345 }
346
347 /* Set up ECN state for a packet on a ESTABLISHED socket that is about to
348  * be sent.
349  */
350 static inline void TCP_ECN_send(struct sock *sk, struct sk_buff *skb,
351                                 int tcp_header_len)
352 {
353         struct tcp_sock *tp = tcp_sk(sk);
354
355         if (tp->ecn_flags & TCP_ECN_OK) {
356                 /* Not-retransmitted data segment: set ECT and inject CWR. */
357                 if (skb->len != tcp_header_len &&
358                     !before(TCP_SKB_CB(skb)->seq, tp->snd_nxt)) {
359                         INET_ECN_xmit(sk);
360                         if (tp->ecn_flags & TCP_ECN_QUEUE_CWR) {
361                                 tp->ecn_flags &= ~TCP_ECN_QUEUE_CWR;
362                                 tcp_hdr(skb)->cwr = 1;
363                                 skb_shinfo(skb)->gso_type |= SKB_GSO_TCP_ECN;
364                         }
365                 } else {
366                         /* ACK or retransmitted segment: clear ECT|CE */
367                         INET_ECN_dontxmit(sk);
368                 }
369                 if (tp->ecn_flags & TCP_ECN_DEMAND_CWR)
370                         tcp_hdr(skb)->ece = 1;
371         }
372 }
373
374 /* Constructs common control bits of non-data skb. If SYN/FIN is present,
375  * auto increment end seqno.
376  */
377 static void tcp_init_nondata_skb(struct sk_buff *skb, u32 seq, u8 flags)
378 {
379         struct skb_shared_info *shinfo = skb_shinfo(skb);
380
381         skb->ip_summed = CHECKSUM_PARTIAL;
382         skb->csum = 0;
383
384         TCP_SKB_CB(skb)->tcp_flags = flags;
385         TCP_SKB_CB(skb)->sacked = 0;
386
387         shinfo->gso_segs = 1;
388         shinfo->gso_size = 0;
389         shinfo->gso_type = 0;
390
391         TCP_SKB_CB(skb)->seq = seq;
392         if (flags & (TCPHDR_SYN | TCPHDR_FIN))
393                 seq++;
394         TCP_SKB_CB(skb)->end_seq = seq;
395 }
396
397 static inline bool tcp_urg_mode(const struct tcp_sock *tp)
398 {
399         return tp->snd_una != tp->snd_up;
400 }
401
402 #define OPTION_SACK_ADVERTISE   (1 << 0)
403 #define OPTION_TS               (1 << 1)
404 #define OPTION_MD5              (1 << 2)
405 #define OPTION_WSCALE           (1 << 3)
406 #define OPTION_FAST_OPEN_COOKIE (1 << 8)
407
408 struct tcp_out_options {
409         u16 options;            /* bit field of OPTION_* */
410         u16 mss;                /* 0 to disable */
411         u8 ws;                  /* window scale, 0 to disable */
412         u8 num_sack_blocks;     /* number of SACK blocks to include */
413         u8 hash_size;           /* bytes in hash_location */
414         __u8 *hash_location;    /* temporary pointer, overloaded */
415         __u32 tsval, tsecr;     /* need to include OPTION_TS */
416         struct tcp_fastopen_cookie *fastopen_cookie;    /* Fast open cookie */
417 };
418
419 /* Write previously computed TCP options to the packet.
420  *
421  * Beware: Something in the Internet is very sensitive to the ordering of
422  * TCP options, we learned this through the hard way, so be careful here.
423  * Luckily we can at least blame others for their non-compliance but from
424  * inter-operability perspective it seems that we're somewhat stuck with
425  * the ordering which we have been using if we want to keep working with
426  * those broken things (not that it currently hurts anybody as there isn't
427  * particular reason why the ordering would need to be changed).
428  *
429  * At least SACK_PERM as the first option is known to lead to a disaster
430  * (but it may well be that other scenarios fail similarly).
```

```
431  */
432 static void tcp_options_write(__be32 *ptr, struct tcp_sock *tp,
433                               struct tcp_out_options *opts)
434 {
435         u16 options = opts->options;    /* mungable copy */
436
437         if (unlikely(OPTION_MD5 & options)) {
438                 *ptr++ = htonl((TCPOPT_NOP << 24) | (TCPOPT_NOP << 16) |
439                                (TCPOPT_MD5SIG << 8) | TCPOLEN_MD5SIG);
440                 /* overload cookie hash location */
441                 opts->hash_location = (__u8 *)ptr;
442                 ptr += 4;
443         }
444
445         if (unlikely(opts->mss)) {
446                 *ptr++ = htonl((TCPOPT_MSS << 24) |
447                                (TCPOLEN_MSS << 16) |
448                                opts->mss);
449         }
450
451         if (likely(OPTION_TS & options)) {
452                 if (unlikely(OPTION_SACK_ADVERTISE & options)) {
453                         *ptr++ = htonl((TCPOPT_SACK_PERM << 24) |
454                                        (TCPOLEN_SACK_PERM << 16) |
455                                        (TCPOPT_TIMESTAMP << 8) |
456                                        TCPOLEN_TIMESTAMP);
457                         options &= ~OPTION_SACK_ADVERTISE;
458                 } else {
459                         *ptr++ = htonl((TCPOPT_NOP << 24) |
460                                        (TCPOPT_NOP << 16) |
461                                        (TCPOPT_TIMESTAMP << 8) |
462                                        TCPOLEN_TIMESTAMP);
463                 }
464                 *ptr++ = htonl(opts->tsval);
465                 *ptr++ = htonl(opts->tsecr);
466         }
467
468         if (unlikely(OPTION_SACK_ADVERTISE & options)) {
469                 *ptr++ = htonl((TCPOPT_NOP << 24) |
470                                (TCPOPT_NOP << 16) |
471                                (TCPOPT_SACK_PERM << 8) |
472                                TCPOLEN_SACK_PERM);
473         }
474
475         if (unlikely(OPTION_WSCALE & options)) {
476                 *ptr++ = htonl((TCPOPT_NOP << 24) |
477                                (TCPOPT_WINDOW << 16) |
478                                (TCPOLEN_WINDOW << 8) |
479                                opts->ws);
480         }
481
482         if (unlikely(opts->num_sack_blocks)) {
483                 struct tcp_sack_block *sp = tp->rx_opt.dsack ?
484                         tp->duplicate_sack : tp->selective_acks;
485                 int this_sack;
486
487                 *ptr++ = htonl((TCPOPT_NOP  << 24) |
488                                (TCPOPT_NOP  << 16) |
489                                (TCPOPT_SACK <<  8) |
490                                (TCPOLEN_SACK_BASE + (opts->num_sack_blocks *
491                                                      TCPOLEN_SACK_PERBLOCK)));
492
493                 for (this_sack = 0; this_sack < opts->num_sack_blocks;
494                      ++this_sack) {
495                         *ptr++ = htonl(sp[this_sack].start_seq);
496                         *ptr++ = htonl(sp[this_sack].end_seq);
497                 }
498
499                 tp->rx_opt.dsack = 0;
500         }
501
502         if (unlikely(OPTION_FAST_OPEN_COOKIE & options)) {
503                 struct tcp_fastopen_cookie *foc = opts->fastopen_cookie;
504
505                 *ptr++ = htonl((TCPOPT_EXP << 24) |
506                                ((TCPOLEN_EXP_FASTOPEN_BASE + foc->len) << 16) |
507                                TCPOPT_FASTOPEN_MAGIC);
508
509                 memcpy(ptr, foc->val, foc->len);
510                 if ((foc->len & 3) == 2) {
511                         u8 *align = ((u8 *)ptr) + foc->len;
512                         align[0] = align[1] = TCPOPT_NOP;
513                 }
514                 ptr += (foc->len + 3) >> 2;
515         }
516 }
517
518 /* Compute TCP options for SYN packets. This is not the final
519  * network wire format yet.
520  */
```

```
521 static unsigned int tcp_syn_options(struct sock *sk, struct sk_buff *skb,
522                                      struct tcp_out_options *opts,
523                                      struct tcp_md5sig_key **md5)
524 {
525         struct tcp_sock *tp = tcp_sk(sk);
526         unsigned int remaining = MAX_TCP_OPTION_SPACE;
527         struct tcp_fastopen_request *fastopen = tp->fastopen_req;
528
529 #ifdef CONFIG_TCP_MD5SIG
530         *md5 = tp->af_specific->md5_lookup(sk, sk);
531         if (*md5) {
532                 opts->options |= OPTION_MD5;
533                 remaining -= TCPOLEN_MD5SIG_ALIGNED;
534         }
535 #else
536         *md5 = NULL;
537 #endif
538
539         /* We always get an MSS option.  The option bytes which will be seen in
540          * normal data packets should timestamps be used, must be in the MSS
541          * advertised.  But we subtract them from tp->mss_cache so that
542          * calculations in tcp_sendmsg are simpler etc.  So account for this
543          * fact here if necessary.  If we don't do this correctly, as a
544          * receiver we won't recognize data packets as being full sized when we
545          * should, and thus we won't abide by the delayed ACK rules correctly.
546          * SACKs don't matter, we never delay an ACK when we have any of those
547          * going out.  */
548         opts->mss = tcp_advertise_mss(sk);
549         remaining -= TCPOLEN_MSS_ALIGNED;
550
551         if (likely(sysctl_tcp_timestamps && *md5 == NULL)) {
552                 opts->options |= OPTION_TS;
553                 opts->tsval = TCP_SKB_CB(skb)->when + tp->tsoffset;
554                 opts->tsecr = tp->rx_opt.ts_recent;
555                 remaining -= TCPOLEN_TSTAMP_ALIGNED;
556         }
557         if (likely(sysctl_tcp_window_scaling)) {
558                 opts->ws = tp->rx_opt.rcv_wscale;
559                 opts->options |= OPTION_WSCALE;
560                 remaining -= TCPOLEN_WSCALE_ALIGNED;
561         }
562         if (likely(sysctl_tcp_sack)) {
563                 opts->options |= OPTION_SACK_ADVERTISE;
564                 if (unlikely(!(OPTION_TS & opts->options)))
565                         remaining -= TCPOLEN_SACKPERM_ALIGNED;
566         }
567
568         if (fastopen && fastopen->cookie.len >= 0) {
569                 u32 need = TCPOLEN_EXP_FASTOPEN_BASE + fastopen->cookie.len;
570                 need = (need + 3) & ~3U;  /* Align to 32 bits */
571                 if (remaining >= need) {
572                         opts->options |= OPTION_FAST_OPEN_COOKIE;
573                         opts->fastopen_cookie = &fastopen->cookie;
574                         remaining -= need;
575                         tp->syn_fastopen = 1;
576                 }
577         }
578
579         return MAX_TCP_OPTION_SPACE - remaining;
580 }
581
582 /* Set up TCP options for SYN-ACKs. */
583 static unsigned int tcp_synack_options(struct sock *sk,
584                                        struct request_sock *req,
585                                        unsigned int mss, struct sk_buff *skb,
586                                        struct tcp_out_options *opts,
587                                        struct tcp_md5sig_key **md5,
588                                        struct tcp_fastopen_cookie *foc)
589 {
590         struct inet_request_sock *ireq = inet_rsk(req);
591         unsigned int remaining = MAX_TCP_OPTION_SPACE;
592
593 #ifdef CONFIG_TCP_MD5SIG
594         *md5 = tcp_rsk(req)->af_specific->md5_lookup(sk, req);
595         if (*md5) {
596                 opts->options |= OPTION_MD5;
597                 remaining -= TCPOLEN_MD5SIG_ALIGNED;
598
599                 /* We can't fit any SACK blocks in a packet with MD5 + TS
600                  * options. There was discussion about disabling SACK
601                  * rather than TS in order to fit in better with old,
602                  * buggy kernels, but that was deemed to be unnecessary.
603                  */
604                 ireq->tstamp_ok &= !ireq->sack_ok;
605         }
606 #else
607         *md5 = NULL;
608 #endif
609
610         /* We always send an MSS option. */
```

```
611         opts->mss = mss;
612         remaining -= TCPOLEN_MSS_ALIGNED;
613
614         if (likely(ireq->wscale_ok)) {
615                 opts->ws = ireq->rcv_wscale;
616                 opts->options |= OPTION_WSCALE;
617                 remaining -= TCPOLEN_WSCALE_ALIGNED;
618         }
619         if (likely(ireq->tstamp_ok)) {
620                 opts->options |= OPTION_TS;
621                 opts->tsval = TCP_SKB_CB(skb)->when;
622                 opts->tsecr = req->ts_recent;
623                 remaining -= TCPOLEN_TSTAMP_ALIGNED;
624         }
625         if (likely(ireq->sack_ok)) {
626                 opts->options |= OPTION_SACK_ADVERTISE;
627                 if (unlikely(!ireq->tstamp_ok))
628                         remaining -= TCPOLEN_SACKPERM_ALIGNED;
629         }
630         if (foc != NULL && foc->len >= 0) {
631                 u32 need = TCPOLEN_EXP_FASTOPEN_BASE + foc->len;
632                 need = (need + 3) & ~3U;  /* Align to 32 bits */
633                 if (remaining >= need) {
634                         opts->options |= OPTION_FAST_OPEN_COOKIE;
635                         opts->fastopen_cookie = foc;
636                         remaining -= need;
637                 }
638         }
639
640         return MAX_TCP_OPTION_SPACE - remaining;
641 }
642
643 /* Compute TCP options for ESTABLISHED sockets. This is not the
644  * final wire format yet.
645  */
646 static unsigned int tcp_established_options(struct sock *sk, struct sk_buff *skb,
647                                             struct tcp_out_options *opts,
648                                             struct tcp_md5sig_key **md5)
649 {
650         struct tcp_skb_cb *tcb = skb ? TCP_SKB_CB(skb) : NULL;
651         struct tcp_sock *tp = tcp_sk(sk);
652         unsigned int size = 0;
653         unsigned int eff_sacks;
654
655         opts->options = 0;
656
657 #ifdef CONFIG_TCP_MD5SIG
658         *md5 = tp->af_specific->md5_lookup(sk, sk);
659         if (unlikely(*md5)) {
660                 opts->options |= OPTION_MD5;
661                 size += TCPOLEN_MD5SIG_ALIGNED;
662         }
663 #else
664         *md5 = NULL;
665 #endif
666
667         if (likely(tp->rx_opt.tstamp_ok)) {
668                 opts->options |= OPTION_TS;
669                 opts->tsval = tcb ? tcb->when + tp->tsoffset : 0;
670                 opts->tsecr = tp->rx_opt.ts_recent;
671                 size += TCPOLEN_TSTAMP_ALIGNED;
672         }
673
674         eff_sacks = tp->rx_opt.num_sacks + tp->rx_opt.dsack;
675         if (unlikely(eff_sacks)) {
676                 const unsigned int remaining = MAX_TCP_OPTION_SPACE - size;
677                 opts->num_sack_blocks =
678                         min_t(unsigned int, eff_sacks,
679                               (remaining - TCPOLEN_SACK_BASE_ALIGNED) /
680                               TCPOLEN_SACK_PERBLOCK);
681                 size += TCPOLEN_SACK_BASE_ALIGNED +
682                         opts->num_sack_blocks * TCPOLEN_SACK_PERBLOCK;
683         }
684
685         return size;
686 }
687
688
689 /* TCP SMALL QUEUES (TSQ)
690  *
691  * TSQ goal is to keep small amount of skbs per tcp flow in tx queues (qdisc+dev)
692  * to reduce RTT and bufferbloat.
693  * We do this using a special skb destructor (tcp_wfree).
694  *
695  * Its important tcp_wfree() can be replaced by sock_wfree() in the event skb
696  * needs to be reallocated in a driver.
697  * The invariant being skb->truesize subtracted from sk->sk_wmem_alloc
698  *
699  * Since transmit from skb destructor is forbidden, we use a tasklet
700  * to process all sockets that eventually need to send more skbs.
```

```
701  * We use one tasklet per cpu, with its own queue of sockets.
702  */
703 struct tsq_tasklet {
704         struct tasklet_struct   tasklet;
705         struct list_head        head; /* queue of tcp sockets */
706 };
707 static DEFINE_PER_CPU(struct tsq_tasklet, tsq_tasklet);
708
709 static void tcp_tsq_handler(struct sock *sk)
710 {
711         if ((1 << sk->sk_state) &
712             (TCPF_ESTABLISHED | TCPF_FIN_WAIT1 | TCPF_CLOSING |
713              TCPF_CLOSE_WAIT  | TCPF_LAST_ACK))
714                 tcp_write_xmit(sk, tcp_current_mss(sk), tcp_sk(sk)->nonagle,
715                                0, GFP_ATOMIC);
716 }
717 /*
718  * One tasklet per cpu tries to send more skbs.
719  * We run in tasklet context but need to disable irqs when
720  * transferring tsq->head because tcp_wfree() might
721  * interrupt us (non NAPI drivers)
722  */
723 static void tcp_tasklet_func(unsigned long data)
724 {
725         struct tsq_tasklet *tsq = (struct tsq_tasklet *)data;
726         LIST_HEAD(list);
727         unsigned long flags;
728         struct list_head *q, *n;
729         struct tcp_sock *tp;
730         struct sock *sk;
731
732         local_irq_save(flags);
733         list_splice_init(&tsq->head, &list);
734         local_irq_restore(flags);
735
736         list_for_each_safe(q, n, &list) {
737                 tp = list_entry(q, struct tcp_sock, tsq_node);
738                 list_del(&tp->tsq_node);
739
740                 sk = (struct sock *)tp;
741                 bh_lock_sock(sk);
742
743                 if (!sock_owned_by_user(sk)) {
744                         tcp_tsq_handler(sk);
745                 } else {
746                         /* defer the work to tcp_release_cb() */
747                         set_bit(TCP_TSQ_DEFERRED, &tp->tsq_flags);
748                 }
749                 bh_unlock_sock(sk);
750
751                 clear_bit(TSQ_QUEUED, &tp->tsq_flags);
752                 sk_free(sk);
753         }
754 }
755
756 #define TCP_DEFERRED_ALL ((1UL << TCP_TSQ_DEFERRED) |           \
757                           (1UL << TCP_WRITE_TIMER_DEFERRED) |   \
758                           (1UL << TCP_DELACK_TIMER_DEFERRED) |  \
759                           (1UL << TCP_MTU_REDUCED_DEFERRED))
760 /**
761  * tcp_release_cb - tcp release_sock() callback
762  * @sk: socket
763  *
764  * called from release_sock() to perform protocol dependent
765  * actions before socket release.
766  */
767 void tcp_release_cb(struct sock *sk)
768 {
769         struct tcp_sock *tp = tcp_sk(sk);
770         unsigned long flags, nflags;
771
772         /* perform an atomic operation only if at least one flag is set */
773         do {
774                 flags = tp->tsq_flags;
775                 if (!(flags & TCP_DEFERRED_ALL))
776                         return;
777                 nflags = flags & ~TCP_DEFERRED_ALL;
778         } while (cmpxchg(&tp->tsq_flags, flags, nflags) != flags);
779
780         if (flags & (1UL << TCP_TSQ_DEFERRED))
781                 tcp_tsq_handler(sk);
782
783         /* Here begins the tricky part :
784          * We are called from release_sock() with :
785          * 1) BH disabled
786          * 2) sk_lock.slock spinlock held
787          * 3) socket owned by us (sk->sk_lock.owned == 1)
788          *
789          * But following code is meant to be called from BH handlers,
790          * so we should keep BH disabled, but early release socket ownership
```

```
791          */
792         sock_release_ownership(sk);
793
794         if (flags & (1UL << TCP_WRITE_TIMER_DEFERRED)) {
795                 tcp_write_timer_handler(sk);
796                 __sock_put(sk);
797         }
798         if (flags & (1UL << TCP_DELACK_TIMER_DEFERRED)) {
799                 tcp_delack_timer_handler(sk);
800                 __sock_put(sk);
801         }
802         if (flags & (1UL << TCP_MTU_REDUCED_DEFERRED)) {
803                 inet_csk(sk)->icsk_af_ops->mtu_reduced(sk);
804                 __sock_put(sk);
805         }
806 }
807 EXPORT_SYMBOL(tcp_release_cb);
808
809 void __init tcp_tasklet_init(void)
810 {
811         int i;
812
813         for_each_possible_cpu(i) {
814                 struct tsq_tasklet *tsq = &per_cpu(tsq_tasklet, i);
815
816                 INIT_LIST_HEAD(&tsq->head);
817                 tasklet_init(&tsq->tasklet,
818                              tcp_tasklet_func,
819                              (unsigned long)tsq);
820         }
821 }
822
823 /*
824  * Write buffer destructor automatically called from kfree_skb.
825  * We can't xmit new skbs from this context, as we might already
826  * hold qdisc lock.
827  */
828 void tcp_wfree(struct sk_buff *skb)
829 {
830         struct sock *sk = skb->sk;
831         struct tcp_sock *tp = tcp_sk(sk);
832
833         if (test_and_clear_bit(TSQ_THROTTLED, &tp->tsq_flags) &&
834             !test_and_set_bit(TSQ_QUEUED, &tp->tsq_flags)) {
835                 unsigned long flags;
836                 struct tsq_tasklet *tsq;
837
838                 /* Keep a ref on socket.
839                  * This last ref will be released in tcp_tasklet_func()
840                  */
841                 atomic_sub(skb->truesize - 1, &sk->sk_wmem_alloc);
842
843                 /* queue this socket to tasklet queue */
844                 local_irq_save(flags);
845                 tsq = &__get_cpu_var(tsq_tasklet);
846                 list_add(&tp->tsq_node, &tsq->head);
847                 tasklet_schedule(&tsq->tasklet);
848                 local_irq_restore(flags);
849         } else {
850                 sock_wfree(skb);
851         }
852 }
853
854 /* This routine actually transmits TCP packets queued in by
855  * tcp_do_sendmsg().  This is used by both the initial
856  * transmission and possible later retransmissions.
857  * All SKB's seen here are completely headerless.  It is our
858  * job to build the TCP header, and pass the packet down to
859  * IP so it can do the same plus pass the packet off to the
860  * device.
861  *
862  * We are working here with either a clone of the original
863  * SKB, or a fresh unique copy made by the retransmit engine.
864  */
865 static int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb, int clone_it,
866                             gfp_t gfp_mask)
867 {
868         const struct inet_connection_sock *icsk = inet_csk(sk);
869         struct inet_sock *inet;
870         struct tcp_sock *tp;
871         struct tcp_skb_cb *tcb;
872         struct tcp_out_options opts;
873         unsigned int tcp_options_size, tcp_header_size;
874         struct tcp_md5sig_key *md5;
875         struct tcphdr *th;
876         int err;
877
878         BUG_ON(!skb || !tcp_skb_pcount(skb));
879
880         if (clone_it) {
```

```
881                 skb_mstamp_get(&skb->skb_mstamp);
882
883                 if (unlikely(skb_cloned(skb)))
884                         skb = pskb_copy(skb, gfp_mask);
885                 else
886                         skb = skb_clone(skb, gfp_mask);
887                 if (unlikely(!skb))
888                         return -ENOBUFS;
889                 /* Our usage of tstamp should remain private */
890                 skb->tstamp.tv64 = 0;
891         }
892
893         inet = inet_sk(sk);
894         tp = tcp_sk(sk);
895         tcb = TCP_SKB_CB(skb);
896         memset(&opts, 0, sizeof(opts));
897
898         if (unlikely(tcb->tcp_flags & TCPHDR_SYN))
899                 tcp_options_size = tcp_syn_options(sk, skb, &opts, &md5);
900         else
901                 tcp_options_size = tcp_established_options(sk, skb, &opts,
902                                                            &md5);
903         tcp_header_size = tcp_options_size + sizeof(struct tcphdr);
904
905         if (tcp_packets_in_flight(tp) == 0)
906                 tcp_ca_event(sk, CA_EVENT_TX_START);
907
908         /* if no packet is in qdisc/device queue, then allow XPS to select
909          * another queue.
910          */
911         skb->ooo_okay = sk_wmem_alloc_get(sk) == 0;
912
913         skb_push(skb, tcp_header_size);
914         skb_reset_transport_header(skb);
915
916         skb_orphan(skb);
917         skb->sk = sk;
918         skb->destructor = tcp_wfree;
919         skb_set_hash_from_sk(skb, sk);
920         atomic_add(skb->truesize, &sk->sk_wmem_alloc);
921
922         /* Build TCP header and checksum it. */
923         th = tcp_hdr(skb);
924         th->source              = inet->inet_sport;
925         th->dest                = inet->inet_dport;
926         th->seq                 = htonl(tcb->seq);
927         th->ack_seq             = htonl(tp->rcv_nxt);
928         *(((__be16 *)th) + 6)   = htons(((tcp_header_size >> 2) << 12) |
929                                         tcb->tcp_flags);
930
931         if (unlikely(tcb->tcp_flags & TCPHDR_SYN)) {
932                 /* RFC1323: The window in SYN & SYN/ACK segments
933                  * is never scaled.
934                  */
935                 th->window      = htons(min(tp->rcv_wnd, 65535U));
936         } else {
937                 th->window      = htons(tcp_select_window(sk));
938         }
939         th->check               = 0;
940         th->urg_ptr             = 0;
941
942         /* The urg_mode check is necessary during a below snd_una win probe */
943         if (unlikely(tcp_urg_mode(tp) && before(tcb->seq, tp->snd_up))) {
944                 if (before(tp->snd_up, tcb->seq + 0x10000)) {
945                         th->urg_ptr = htons(tp->snd_up - tcb->seq);
946                         th->urg = 1;
947                 } else if (after(tcb->seq + 0xFFFF, tp->snd_nxt)) {
948                         th->urg_ptr = htons(0xFFFF);
949                         th->urg = 1;
950                 }
951         }
952
953         tcp_options_write((__be32 *)(th + 1), tp, &opts);
954         if (likely((tcb->tcp_flags & TCPHDR_SYN) == 0))
955                 TCP_ECN_send(sk, skb, tcp_header_size);
956
957 #ifdef CONFIG_TCP_MD5SIG
958         /* Calculate the MD5 hash, as we have all we need now */
959         if (md5) {
960                 sk_nocaps_add(sk, NETIF_F_GSO_MASK);
961                 tp->af_specific->calc_md5_hash(opts.hash_location,
962                                                md5, sk, NULL, skb);
963         }
964 #endif
965
966         icsk->icsk_af_ops->send_check(sk, skb);
967
968         if (likely(tcb->tcp_flags & TCPHDR_ACK))
969                 tcp_event_ack_sent(sk, tcp_skb_pcount(skb));
970
```

```
971         if (skb->len != tcp_header_size)
972                 tcp_event_data_sent(tp, sk);
973
974         if (after(tcb->end_seq, tp->snd_nxt) || tcb->seq == tcb->end_seq)
975                 TCP_ADD_STATS(sock_net(sk), TCP_MIB_OUTSEGS,
976                               tcp_skb_pcount(skb));
977
978         err = icsk->icsk_af_ops->queue_xmit(sk, skb, &inet->cork.fl);
979         if (likely(err <= 0))
980                 return err;
981
982         tcp_enter_cwr(sk);
983
984         return net_xmit_eval(err);
985 }
986
987 /* This routine just queues the buffer for sending.
988  *
989  * NOTE: probe0 timer is not checked, do not forget tcp_push_pending_frames,
990  * otherwise socket can stall.
991  */
992 static void tcp_queue_skb(struct sock *sk, struct sk_buff *skb)
993 {
994         struct tcp_sock *tp = tcp_sk(sk);
995
996         /* Advance write_seq and place onto the write_queue. */
997         tp->write_seq = TCP_SKB_CB(skb)->end_seq;
998         skb_header_release(skb);
999         tcp_add_write_queue_tail(sk, skb);
1000         sk->sk_wmem_queued += skb->truesize;
1001         sk_mem_charge(sk, skb->truesize);
1002 }
1003
1004 /* Initialize TSO segments for a packet. */
1005 static void tcp_set_skb_tso_segs(const struct sock *sk, struct sk_buff *skb,
1006                                  unsigned int mss_now)
1007 {
1008         struct skb_shared_info *shinfo = skb_shinfo(skb);
1009
1010         /* Make sure we own this skb before messing gso_size/gso_segs */
1011         WARN_ON_ONCE(skb_cloned(skb));
1012
1013         if (skb->len <= mss_now || skb->ip_summed == CHECKSUM_NONE) {
1014                 /* Avoid the costly divide in the normal
1015                  * non-TSO case.
1016                  */
1017                 shinfo->gso_segs = 1;
1018                 shinfo->gso_size = 0;
1019                 shinfo->gso_type = 0;
1020         } else {
1021                 shinfo->gso_segs = DIV_ROUND_UP(skb->len, mss_now);
1022                 shinfo->gso_size = mss_now;
1023                 shinfo->gso_type = sk->sk_gso_type;
1024         }
1025 }
1026
1027 /* When a modification to fackets out becomes necessary, we need to check
1028  * skb is counted to fackets_out or not.
1029  */
1030 static void tcp_adjust_fackets_out(struct sock *sk, const struct sk_buff *skb,
1031                                    int decr)
1032 {
1033         struct tcp_sock *tp = tcp_sk(sk);
1034
1035         if (!tp->sacked_out || tcp_is_reno(tp))
1036                 return;
1037
1038         if (after(tcp_highest_sack_seq(tp), TCP_SKB_CB(skb)->seq))
1039                 tp->fackets_out -= decr;
1040 }
1041
1042 /* Pcount in the middle of the write queue got changed, we need to do various
1043  * tweaks to fix counters
1044  */
1045 static void tcp_adjust_pcount(struct sock *sk, const struct sk_buff *skb, int decr)
1046 {
1047         struct tcp_sock *tp = tcp_sk(sk);
1048
1049         tp->packets_out -= decr;
1050
1051         if (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED)
1052                 tp->sacked_out -= decr;
1053         if (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_RETRANS)
1054                 tp->retrans_out -= decr;
1055         if (TCP_SKB_CB(skb)->sacked & TCPCB_LOST)
1056                 tp->lost_out -= decr;
1057
1058         /* Reno case is special. Sigh... */
1059         if (tcp_is_reno(tp) && decr > 0)
1060                 tp->sacked_out -= min_t(u32, tp->sacked_out, decr);
```

```
1061
1062        tcp_adjust_fackets_out(sk, skb, decr);
1063
1064        if (tp->lost_skb_hint &&
1065            before(TCP_SKB_CB(skb)->seq, TCP_SKB_CB(tp->lost_skb_hint)->seq) &&
1066            (tcp_is_fack(tp) || (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED)))
1067                tp->lost_cnt_hint -= decr;
1068
1069        tcp_verify_left_out(tp);
1070 }
1071
1072 static void tcp_fragment_tstamp(struct sk_buff *skb, struct sk_buff *skb2)
1073 {
1074        struct skb_shared_info *shinfo = skb_shinfo(skb);
1075
1076        if (unlikely(shinfo->tx_flags & SKBTX_ANY_TSTAMP) &&
1077            !before(shinfo->tskey, TCP_SKB_CB(skb2)->seq)) {
1078                struct skb_shared_info *shinfo2 = skb_shinfo(skb2);
1079                u8 tsflags = shinfo->tx_flags & SKBTX_ANY_TSTAMP;
1080
1081                shinfo->tx_flags &= ~tsflags;
1082                shinfo2->tx_flags |= tsflags;
1083                swap(shinfo->tskey, shinfo2->tskey);
1084        }
1085 }
1086
1087 /* Function to create two new TCP segments.  Shrinks the given segment
1088  * to the specified size and appends a new segment with the rest of the
1089  * packet to the list.  This won't be called frequently, I hope.
1090  * Remember, these are still headerless SKBs at this point.
1091  */
1092 int tcp_fragment(struct sock *sk, struct sk_buff *skb, u32 len,
1093                  unsigned int mss_now, gfp_t gfp)
1094 {
1095        struct tcp_sock *tp = tcp_sk(sk);
1096        struct sk_buff *buff;
1097        int nsize, old_factor;
1098        int nlen;
1099        u8 flags;
1100
1101        if (WARN_ON(len > skb->len))
1102                return -EINVAL;
1103
1104        nsize = skb_headlen(skb) - len;
1105        if (nsize < 0)
1106                nsize = 0;
1107
1108        if (skb_unclone(skb, gfp))
1109                return -ENOMEM;
1110
1111        /* Get a new skb... force flag on. */
1112        buff = sk_stream_alloc_skb(sk, nsize, gfp);
1113        if (buff == NULL)
1114                return -ENOMEM; /* We'll just try again later. */
1115
1116        sk->sk_wmem_queued += buff->truesize;
1117        sk_mem_charge(sk, buff->truesize);
1118        nlen = skb->len - len - nsize;
1119        buff->truesize += nlen;
1120        skb->truesize -= nlen;
1121
1122        /* Correct the sequence numbers. */
1123        TCP_SKB_CB(buff)->seq = TCP_SKB_CB(skb)->seq + len;
1124        TCP_SKB_CB(buff)->end_seq = TCP_SKB_CB(skb)->end_seq;
1125        TCP_SKB_CB(skb)->end_seq = TCP_SKB_CB(buff)->seq;
1126
1127        /* PSH and FIN should only be set in the second packet. */
1128        flags = TCP_SKB_CB(skb)->tcp_flags;
1129        TCP_SKB_CB(skb)->tcp_flags = flags & ~(TCPHDR_FIN | TCPHDR_PSH);
1130        TCP_SKB_CB(buff)->tcp_flags = flags;
1131        TCP_SKB_CB(buff)->sacked = TCP_SKB_CB(skb)->sacked;
1132
1133        if (!skb_shinfo(skb)->nr_frags && skb->ip_summed != CHECKSUM_PARTIAL) {
1134                /* Copy and checksum data tail into the new buffer. */
1135                buff->csum = csum_partial_copy_nocheck(skb->data + len,
1136                                                       skb_put(buff, nsize),
1137                                                       nsize, 0);
1138
1139                skb_trim(skb, len);
1140
1141                skb->csum = csum_block_sub(skb->csum, buff->csum, len);
1142        } else {
1143                skb->ip_summed = CHECKSUM_PARTIAL;
1144                skb_split(skb, buff, len);
1145        }
1146
1147        buff->ip_summed = skb->ip_summed;
1148
1149        /* Looks stupid, but our code really uses when of
1150         * skbs, which it never sent before. --ANK
```

```
1151          */
1152         TCP_SKB_CB(buff)->when = TCP_SKB_CB(skb)->when;
1153         buff->tstamp = skb->tstamp;
1154         tcp_fragment_tstamp(skb, buff);
1155
1156         old_factor = tcp_skb_pcount(skb);
1157
1158         /* Fix up tso_factor for both original and new SKB.  */
1159         tcp_set_skb_tso_segs(sk, skb, mss_now);
1160         tcp_set_skb_tso_segs(sk, buff, mss_now);
1161
1162         /* If this packet has been sent out already, we must
1163          * adjust the various packet counters.
1164          */
1165         if (!before(tp->snd_nxt, TCP_SKB_CB(buff)->end_seq)) {
1166                 int diff = old_factor - tcp_skb_pcount(skb) -
1167                         tcp_skb_pcount(buff);
1168
1169                 if (diff)
1170                         tcp_adjust_pcount(sk, skb, diff);
1171         }
1172
1173         /* Link BUFF into the send queue. */
1174         skb_header_release(buff);
1175         tcp_insert_write_queue_after(skb, buff, sk);
1176
1177         return 0;
1178 }
1179
1180 /* This is similar to __pskb_pull_head() (it will go to core/skbuff.c
1181  * eventually). The difference is that pulled data not copied, but
1182  * immediately discarded.
1183  */
1184 static void __pskb_trim_head(struct sk_buff *skb, int len)
1185 {
1186         struct skb_shared_info *shinfo;
1187         int i, k, eat;
1188
1189         eat = min_t(int, len, skb_headlen(skb));
1190         if (eat) {
1191                 __skb_pull(skb, eat);
1192                 len -= eat;
1193                 if (!len)
1194                         return;
1195         }
1196         eat = len;
1197         k = 0;
1198         shinfo = skb_shinfo(skb);
1199         for (i = 0; i < shinfo->nr_frags; i++) {
1200                 int size = skb_frag_size(&shinfo->frags[i]);
1201
1202                 if (size <= eat) {
1203                         skb_frag_unref(skb, i);
1204                         eat -= size;
1205                 } else {
1206                         shinfo->frags[k] = shinfo->frags[i];
1207                         if (eat) {
1208                                 shinfo->frags[k].page_offset += eat;
1209                                 skb_frag_size_sub(&shinfo->frags[k], eat);
1210                                 eat = 0;
1211                         }
1212                         k++;
1213                 }
1214         }
1215         shinfo->nr_frags = k;
1216
1217         skb_reset_tail_pointer(skb);
1218         skb->data_len -= len;
1219         skb->len = skb->data_len;
1220 }
1221
1222 /* Remove acked data from a packet in the transmit queue. */
1223 int tcp_trim_head(struct sock *sk, struct sk_buff *skb, u32 len)
1224 {
1225         if (skb_unclone(skb, GFP_ATOMIC))
1226                 return -ENOMEM;
1227
1228         __pskb_trim_head(skb, len);
1229
1230         TCP_SKB_CB(skb)->seq += len;
1231         skb->ip_summed = CHECKSUM_PARTIAL;
1232
1233         skb->truesize        -= len;
1234         sk->sk_wmem_queued   -= len;
1235         sk_mem_uncharge(sk, len);
1236         sock_set_flag(sk, SOCK_QUEUE_SHRUNK);
1237
1238         /* Any change of skb->len requires recalculation of tso factor. */
1239         if (tcp_skb_pcount(skb) > 1)
1240                 tcp_set_skb_tso_segs(sk, skb, tcp_skb_mss(skb));
```

```
1241
1242        return 0;
1243 }
1244
1245 /* Calculate MSS not accounting any TCP options.  */
1246 static inline int __tcp_mtu_to_mss(struct sock *sk, int pmtu)
1247 {
1248        const struct tcp_sock *tp = tcp_sk(sk);
1249        const struct inet_connection_sock *icsk = inet_csk(sk);
1250        int mss_now;
1251
1252        /* Calculate base mss without TCP options:
1253           It is MMS_S - sizeof(tcphdr) of rfc1122
1254         */
1255        mss_now = pmtu - icsk->icsk_af_ops->net_header_len - sizeof(struct tcphdr);
1256
1257        /* IPv6 adds a frag_hdr in case RTAX_FEATURE_ALLFRAG is set */
1258        if (icsk->icsk_af_ops->net_frag_header_len) {
1259                const struct dst_entry *dst = __sk_dst_get(sk);
1260
1261                if (dst && dst_allfrag(dst))
1262                        mss_now -= icsk->icsk_af_ops->net_frag_header_len;
1263        }
1264
1265        /* Clamp it (mss_clamp does not include tcp options) */
1266        if (mss_now > tp->rx_opt.mss_clamp)
1267                mss_now = tp->rx_opt.mss_clamp;
1268
1269        /* Now subtract optional transport overhead */
1270        mss_now -= icsk->icsk_ext_hdr_len;
1271
1272        /* Then reserve room for full set of TCP options and 8 bytes of data */
1273        if (mss_now < 48)
1274                mss_now = 48;
1275        return mss_now;
1276 }
1277
1278 /* Calculate MSS. Not accounting for SACKs here.  */
1279 int tcp_mtu_to_mss(struct sock *sk, int pmtu)
1280 {
1281        /* Subtract TCP options size, not including SACKs */
1282        return __tcp_mtu_to_mss(sk, pmtu) -
1283               (tcp_sk(sk)->tcp_header_len - sizeof(struct tcphdr));
1284 }
1285
1286 /* Inverse of above */
1287 int tcp_mss_to_mtu(struct sock *sk, int mss)
1288 {
1289        const struct tcp_sock *tp = tcp_sk(sk);
1290        const struct inet_connection_sock *icsk = inet_csk(sk);
1291        int mtu;
1292
1293        mtu = mss +
1294              tp->tcp_header_len +
1295              icsk->icsk_ext_hdr_len +
1296              icsk->icsk_af_ops->net_header_len;
1297
1298        /* IPv6 adds a frag_hdr in case RTAX_FEATURE_ALLFRAG is set */
1299        if (icsk->icsk_af_ops->net_frag_header_len) {
1300                const struct dst_entry *dst = __sk_dst_get(sk);
1301
1302                if (dst && dst_allfrag(dst))
1303                        mtu += icsk->icsk_af_ops->net_frag_header_len;
1304        }
1305        return mtu;
1306 }
1307
1308 /* MTU probing init per socket */
1309 void tcp_mtup_init(struct sock *sk)
1310 {
1311        struct tcp_sock *tp = tcp_sk(sk);
1312        struct inet_connection_sock *icsk = inet_csk(sk);
1313
1314        icsk->icsk_mtup.enabled = sysctl_tcp_mtu_probing > 1;
1315        icsk->icsk_mtup.search_high = tp->rx_opt.mss_clamp + sizeof(struct tcphdr) +
1316                                      icsk->icsk_af_ops->net_header_len;
1317        icsk->icsk_mtup.search_low = tcp_mss_to_mtu(sk, sysctl_tcp_base_mss);
1318        icsk->icsk_mtup.probe_size = 0;
1319 }
1320 EXPORT_SYMBOL(tcp_mtup_init);
1321
1322 /* This function synchronize snd mss to current pmtu/exthdr set.
1323
1324    tp->rx_opt.user_mss is mss set by user by TCP_MAXSEG. It does NOT counts
1325    for TCP options, but includes only bare TCP header.
1326
1327    tp->rx_opt.mss_clamp is mss negotiated at connection setup.
1328    It is minimum of user_mss and mss received with SYN.
1329    It also does not include TCP options.
1330
```

```
1331    inet_csk(sk)->icsk_pmtu_cookie is last pmtu, seen by this function.
1332
1333    tp->mss_cache is current effective sending mss, including
1334    all tcp options except for SACKs. It is evaluated,
1335    taking into account current pmtu, but never exceeds
1336    tp->rx_opt.mss_clamp.
1337
1338    NOTE1. rfc1122 clearly states that advertised MSS
1339    DOES NOT include either tcp or ip options.
1340
1341    NOTE2. inet_csk(sk)->icsk_pmtu_cookie and tp->mss_cache
1342    are READ ONLY outside this function.         --ANK (980731)
1343  */
1344 unsigned int tcp_sync_mss(struct sock *sk, u32 pmtu)
1345 {
1346         struct tcp_sock *tp = tcp_sk(sk);
1347         struct inet_connection_sock *icsk = inet_csk(sk);
1348         int mss_now;
1349
1350         if (icsk->icsk_mtup.search_high > pmtu)
1351                 icsk->icsk_mtup.search_high = pmtu;
1352
1353         mss_now = tcp_mtu_to_mss(sk, pmtu);
1354         mss_now = tcp_bound_to_half_wnd(tp, mss_now);
1355
1356         /* And store cached results */
1357         icsk->icsk_pmtu_cookie = pmtu;
1358         if (icsk->icsk_mtup.enabled)
1359                 mss_now = min(mss_now, tcp_mtu_to_mss(sk, icsk->icsk_mtup.search_low));
1360         tp->mss_cache = mss_now;
1361
1362         return mss_now;
1363 }
1364 EXPORT_SYMBOL(tcp_sync_mss);
1365
1366 /* Compute the current effective MSS, taking SACKs and IP options,
1367  * and even PMTU discovery events into account.
1368  */
1369 unsigned int tcp_current_mss(struct sock *sk)
1370 {
1371         const struct tcp_sock *tp = tcp_sk(sk);
1372         const struct dst_entry *dst = __sk_dst_get(sk);
1373         u32 mss_now;
1374         unsigned int header_len;
1375         struct tcp_out_options opts;
1376         struct tcp_md5sig_key *md5;
1377
1378         mss_now = tp->mss_cache;
1379
1380         if (dst) {
1381                 u32 mtu = dst_mtu(dst);
1382                 if (mtu != inet_csk(sk)->icsk_pmtu_cookie)
1383                         mss_now = tcp_sync_mss(sk, mtu);
1384         }
1385
1386         header_len = tcp_established_options(sk, NULL, &opts, &md5) +
1387                      sizeof(struct tcphdr);
1388         /* The mss_cache is sized based on tp->tcp_header_len, which assumes
1389          * some common options. If this is an odd packet (because we have SACK
1390          * blocks etc) then our calculated header_len will be different, and
1391          * we have to adjust mss_now correspondingly */
1392         if (header_len != tp->tcp_header_len) {
1393                 int delta = (int) header_len - tp->tcp_header_len;
1394                 mss_now -= delta;
1395         }
1396
1397         return mss_now;
1398 }
1399
1400 /* RFC2861, slow part. Adjust cwnd, after it was not full during one rto.
1401  * As additional protections, we do not touch cwnd in retransmission phases,
1402  * and if application hit its sndbuf limit recently.
1403  */
1404 static void tcp_cwnd_application_limited(struct sock *sk)
1405 {
1406         struct tcp_sock *tp = tcp_sk(sk);
1407
1408         if (inet_csk(sk)->icsk_ca_state == TCP_CA_Open &&
1409             sk->sk_socket && !test_bit(SOCK_NOSPACE, &sk->sk_socket->flags)) {
1410                 /* Limited by application or receiver window. */
1411                 u32 init_win = tcp_init_cwnd(tp, __sk_dst_get(sk));
1412                 u32 win_used = max(tp->snd_cwnd_used, init_win);
1413                 if (win_used < tp->snd_cwnd) {
1414                         tp->snd_ssthresh = tcp_current_ssthresh(sk);
1415                         tp->snd_cwnd = (tp->snd_cwnd + win_used) >> 1;
1416                 }
1417                 tp->snd_cwnd_used = 0;
1418         }
1419         tp->snd_cwnd_stamp = tcp_time_stamp;
1420 }
```

```
1421
1422 static void tcp_cwnd_validate(struct sock *sk, bool is_cwnd_limited)
1423 {
1424         struct tcp_sock *tp = tcp_sk(sk);
1425
1426         /* Track the maximum number of outstanding packets in each
1427          * window, and remember whether we were cwnd-limited then.
1428          */
1429         if (!before(tp->snd_una, tp->max_packets_seq) ||
1430             tp->packets_out > tp->max_packets_out) {
1431                 tp->max_packets_out = tp->packets_out;
1432                 tp->max_packets_seq = tp->snd_nxt;
1433                 tp->is_cwnd_limited = is_cwnd_limited;
1434         }
1435
1436         if (tcp_is_cwnd_limited(sk)) {
1437                 /* Network is feed fully. */
1438                 tp->snd_cwnd_used = 0;
1439                 tp->snd_cwnd_stamp = tcp_time_stamp;
1440         } else {
1441                 /* Network starves. */
1442                 if (tp->packets_out > tp->snd_cwnd_used)
1443                         tp->snd_cwnd_used = tp->packets_out;
1444
1445                 if (sysctl_tcp_slow_start_after_idle &&
1446                     (s32)(tcp_time_stamp - tp->snd_cwnd_stamp) >= inet_csk(sk)->icsk_rto)
1447                         tcp_cwnd_application_limited(sk);
1448         }
1449 }
1450
1451 /* Minshall's variant of the Nagle send check. */
1452 static bool tcp_minshall_check(const struct tcp_sock *tp)
1453 {
1454         return after(tp->snd_sml, tp->snd_una) &&
1455                 !after(tp->snd_sml, tp->snd_nxt);
1456 }
1457
1458 /* Update snd_sml if this skb is under mss
1459  * Note that a TSO packet might end with a sub-mss segment
1460  * The test is really :
1461  * if ((skb->len % mss) != 0)
1462  *        tp->snd_sml = TCP_SKB_CB(skb)->end_seq;
1463  * But we can avoid doing the divide again given we already have
1464  *  skb_pcount = skb->len / mss_now
1465  */
1466 static void tcp_minshall_update(struct tcp_sock *tp, unsigned int mss_now,
1467                                 const struct sk_buff *skb)
1468 {
1469         if (skb->len < tcp_skb_pcount(skb) * mss_now)
1470                 tp->snd_sml = TCP_SKB_CB(skb)->end_seq;
1471 }
1472
1473 /* Return false, if packet can be sent now without violation Nagle's rules:
1474  * 1. It is full sized. (provided by caller in %partial bool)
1475  * 2. Or it contains FIN. (already checked by caller)
1476  * 3. Or TCP_CORK is not set, and TCP_NODELAY is set.
1477  * 4. Or TCP_CORK is not set, and all sent packets are ACKed.
1478  *    With Minshall's modification: all sent small packets are ACKed.
1479  */
1480 static bool tcp_nagle_check(bool partial, const struct tcp_sock *tp,
1481                             int nonagle)
1482 {
1483         return partial &&
1484                 ((nonagle & TCP_NAGLE_CORK) ||
1485                  (!nonagle && tp->packets_out && tcp_minshall_check(tp)));
1486 }
1487 /* Returns the portion of skb which can be sent right away */
1488 static unsigned int tcp_mss_split_point(const struct sock *sk,
1489                                         const struct sk_buff *skb,
1490                                         unsigned int mss_now,
1491                                         unsigned int max_segs,
1492                                         int nonagle)
1493 {
1494         const struct tcp_sock *tp = tcp_sk(sk);
1495         u32 partial, needed, window, max_len;
1496
1497         window = tcp_wnd_end(tp) - TCP_SKB_CB(skb)->seq;
1498         max_len = mss_now * max_segs;
1499
1500         if (likely(max_len <= window && skb != tcp_write_queue_tail(sk)))
1501                 return max_len;
1502
1503         needed = min(skb->len, window);
1504
1505         if (max_len <= needed)
1506                 return max_len;
1507
1508         partial = needed % mss_now;
1509         /* If last segment is not a full MSS, check if Nagle rules allow us
1510          * to include this last segment in this skb.
```

```
1511          * Otherwise, we'll split the skb at last MSS boundary
1512          */
1513         if (tcp_nagle_check(partial != 0, tp, nonagle))
1514                 return needed - partial;
1515
1516         return needed;
1517 }
1518
1519 /* Can at least one segment of SKB be sent right now, according to the
1520  * congestion window rules?  If so, return how many segments are allowed.
1521  */
1522 static inline unsigned int tcp_cwnd_test(const struct tcp_sock *tp,
1523                                          const struct sk_buff *skb)
1524 {
1525         u32 in_flight, cwnd;
1526
1527         /* Don't be strict about the congestion window for the final FIN.  */
1528         if ((TCP_SKB_CB(skb)->tcp_flags & TCPHDR_FIN) &&
1529             tcp_skb_pcount(skb) == 1)
1530                 return 1;
1531
1532         in_flight = tcp_packets_in_flight(tp);
1533         cwnd = tp->snd_cwnd;
1534         if (in_flight < cwnd)
1535                 return (cwnd - in_flight);
1536
1537         return 0;
1538 }
1539
1540 /* Initialize TSO state of a skb.
1541  * This must be invoked the first time we consider transmitting
1542  * SKB onto the wire.
1543  */
1544 static int tcp_init_tso_segs(const struct sock *sk, struct sk_buff *skb,
1545                              unsigned int mss_now)
1546 {
1547         int tso_segs = tcp_skb_pcount(skb);
1548
1549         if (!tso_segs || (tso_segs > 1 && tcp_skb_mss(skb) != mss_now)) {
1550                 tcp_set_skb_tso_segs(sk, skb, mss_now);
1551                 tso_segs = tcp_skb_pcount(skb);
1552         }
1553         return tso_segs;
1554 }
1555
1556
1557 /* Return true if the Nagle test allows this packet to be
1558  * sent now.
1559  */
1560 static inline bool tcp_nagle_test(const struct tcp_sock *tp, const struct sk_buff *skb,
1561                                   unsigned int cur_mss, int nonagle)
1562 {
1563         /* Nagle rule does not apply to frames, which sit in the middle of the
1564          * write_queue (they have no chances to get new data).
1565          *
1566          * This is implemented in the callers, where they modify the 'nonagle'
1567          * argument based upon the location of SKB in the send queue.
1568          */
1569         if (nonagle & TCP_NAGLE_PUSH)
1570                 return true;
1571
1572         /* Don't use the nagle rule for urgent data (or for the final FIN). */
1573         if (tcp_urg_mode(tp) || (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_FIN))
1574                 return true;
1575
1576         if (!tcp_nagle_check(skb->len < cur_mss, tp, nonagle))
1577                 return true;
1578
1579         return false;
1580 }
1581
1582 /* Does at least the first segment of SKB fit into the send window? */
1583 static bool tcp_snd_wnd_test(const struct tcp_sock *tp,
1584                              const struct sk_buff *skb,
1585                              unsigned int cur_mss)
1586 {
1587         u32 end_seq = TCP_SKB_CB(skb)->end_seq;
1588
1589         if (skb->len > cur_mss)
1590                 end_seq = TCP_SKB_CB(skb)->seq + cur_mss;
1591
1592         return !after(end_seq, tcp_wnd_end(tp));
1593 }
1594
1595 /* This checks if the data bearing packet SKB (usually tcp_send_head(sk))
1596  * should be put on the wire right now.  If so, it returns the number of
1597  * packets allowed by the congestion window.
1598  */
1599 static unsigned int tcp_snd_test(const struct sock *sk, struct sk_buff *skb,
1600                                  unsigned int cur_mss, int nonagle)
```

```
1601 {
1602         const struct tcp_sock *tp = tcp_sk(sk);
1603         unsigned int cwnd_quota;
1604
1605         tcp_init_tso_segs(sk, skb, cur_mss);
1606
1607         if (!tcp_nagle_test(tp, skb, cur_mss, nonagle))
1608                 return 0;
1609
1610         cwnd_quota = tcp_cwnd_test(tp, skb);
1611         if (cwnd_quota && !tcp_snd_wnd_test(tp, skb, cur_mss))
1612                 cwnd_quota = 0;
1613
1614         return cwnd_quota;
1615 }
1616
1617 /* Test if sending is allowed right now. */
1618 bool tcp_may_send_now(struct sock *sk)
1619 {
1620         const struct tcp_sock *tp = tcp_sk(sk);
1621         struct sk_buff *skb = tcp_send_head(sk);
1622
1623         return skb &&
1624                 tcp_snd_test(sk, skb, tcp_current_mss(sk),
1625                              (tcp_skb_is_last(sk, skb) ?
1626                               tp->nonagle : TCP_NAGLE_PUSH));
1627 }
1628
1629 /* Trim TSO SKB to LEN bytes, put the remaining data into a new packet
1630  * which is put after SKB on the list.  It is very much like
1631  * tcp_fragment() except that it may make several kinds of assumptions
1632  * in order to speed up the splitting operation.  In particular, we
1633  * know that all the data is in scatter-gather pages, and that the
1634  * packet has never been sent out before (and thus is not cloned).
1635  */
1636 static int tso_fragment(struct sock *sk, struct sk_buff *skb, unsigned int len,
1637                         unsigned int mss_now, gfp_t gfp)
1638 {
1639         struct sk_buff *buff;
1640         int nlen = skb->len - len;
1641         u8 flags;
1642
1643         /* All of a TSO frame must be composed of paged data.  */
1644         if (skb->len != skb->data_len)
1645                 return tcp_fragment(sk, skb, len, mss_now, gfp);
1646
1647         buff = sk_stream_alloc_skb(sk, 0, gfp);
1648         if (unlikely(buff == NULL))
1649                 return -ENOMEM;
1650
1651         sk->sk_wmem_queued += buff->truesize;
1652         sk_mem_charge(sk, buff->truesize);
1653         buff->truesize += nlen;
1654         skb->truesize -= nlen;
1655
1656         /* Correct the sequence numbers. */
1657         TCP_SKB_CB(buff)->seq = TCP_SKB_CB(skb)->seq + len;
1658         TCP_SKB_CB(buff)->end_seq = TCP_SKB_CB(skb)->end_seq;
1659         TCP_SKB_CB(skb)->end_seq = TCP_SKB_CB(buff)->seq;
1660
1661         /* PSH and FIN should only be set in the second packet. */
1662         flags = TCP_SKB_CB(skb)->tcp_flags;
1663         TCP_SKB_CB(skb)->tcp_flags = flags & ~(TCPHDR_FIN | TCPHDR_PSH);
1664         TCP_SKB_CB(buff)->tcp_flags = flags;
1665
1666         /* This packet was never sent out yet, so no SACK bits. */
1667         TCP_SKB_CB(buff)->sacked = 0;
1668
1669         buff->ip_summed = skb->ip_summed = CHECKSUM_PARTIAL;
1670         skb_split(skb, buff, len);
1671         tcp_fragment_tstamp(skb, buff);
1672
1673         /* Fix up tso_factor for both original and new SKB.  */
1674         tcp_set_skb_tso_segs(sk, skb, mss_now);
1675         tcp_set_skb_tso_segs(sk, buff, mss_now);
1676
1677         /* Link BUFF into the send queue. */
1678         skb_header_release(buff);
1679         tcp_insert_write_queue_after(skb, buff, sk);
1680
1681         return 0;
1682 }
1683
1684 /* Try to defer sending, if possible, in order to minimize the amount
1685  * of TSO splitting we do.  View it as a kind of TSO Nagle test.
1686  *
1687  * This algorithm is from John Heffner.
1688  */
1689 static bool tcp_tso_should_defer(struct sock *sk, struct sk_buff *skb,
1690                                  bool *is_cwnd_limited)
```

```
1691 {
1692         struct tcp_sock *tp = tcp_sk(sk);
1693         const struct inet_connection_sock *icsk = inet_csk(sk);
1694         u32 send_win, cong_win, limit, in_flight;
1695         int win_divisor;
1696
1697         if (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_FIN)
1698                 goto send_now;
1699
1700         if (icsk->icsk_ca_state != TCP_CA_Open)
1701                 goto send_now;
1702
1703         /* Defer for less than two clock ticks. */
1704         if (tp->tso_deferred &&
1705             (((u32)jiffies << 1) >> 1) - (tp->tso_deferred >> 1) > 1)
1706                 goto send_now;
1707
1708         in_flight = tcp_packets_in_flight(tp);
1709
1710         BUG_ON(tcp_skb_pcount(skb) <= 1 || (tp->snd_cwnd <= in_flight));
1711
1712         send_win = tcp_wnd_end(tp) - TCP_SKB_CB(skb)->seq;
1713
1714         /* From in_flight test above, we know that cwnd > in_flight.  */
1715         cong_win = (tp->snd_cwnd - in_flight) * tp->mss_cache;
1716
1717         limit = min(send_win, cong_win);
1718
1719         /* If a full-sized TSO skb can be sent, do it. */
1720         if (limit >= min_t(unsigned int, sk->sk_gso_max_size,
1721                            tp->xmit_size_goal_segs * tp->mss_cache))
1722                 goto send_now;
1723
1724         /* Middle in queue won't get any more data, full sendable already? */
1725         if ((skb != tcp_write_queue_tail(sk)) && (limit >= skb->len))
1726                 goto send_now;
1727
1728         win_divisor = ACCESS_ONCE(sysctl_tcp_tso_win_divisor);
1729         if (win_divisor) {
1730                 u32 chunk = min(tp->snd_wnd, tp->snd_cwnd * tp->mss_cache);
1731
1732                 /* If at least some fraction of a window is available,
1733                  * just use it.
1734                  */
1735                 chunk /= win_divisor;
1736                 if (limit >= chunk)
1737                         goto send_now;
1738         } else {
1739                 /* Different approach, try not to defer past a single
1740                  * ACK.  Receiver should ACK every other full sized
1741                  * frame, so if we have space for more than 3 frames
1742                  * then send now.
1743                  */
1744                 if (limit > tcp_max_tso_deferred_mss(tp) * tp->mss_cache)
1745                         goto send_now;
1746         }
1747
1748         /* Ok, it looks like it is advisable to defer.
1749          * Do not rearm the timer if already set to not break TCP ACK clocking.
1750          */
1751         if (!tp->tso_deferred)
1752                 tp->tso_deferred = 1 | (jiffies << 1);
1753
1754         if (cong_win < send_win && cong_win < skb->len)
1755                 *is_cwnd_limited = true;
1756
1757         return true;
1758
1759 send_now:
1760         tp->tso_deferred = 0;
1761         return false;
1762 }
1763
1764 /* Create a new MTU probe if we are ready.
1765  * MTU probe is regularly attempting to increase the path MTU by
1766  * deliberately sending larger packets.  This discovers routing
1767  * changes resulting in larger path MTUs.
1768  *
1769  * Returns 0 if we should wait to probe (no cwnd available),
1770  *         1 if a probe was sent,
1771  *         -1 otherwise
1772  */
1773 static int tcp_mtu_probe(struct sock *sk)
1774 {
1775         struct tcp_sock *tp = tcp_sk(sk);
1776         struct inet_connection_sock *icsk = inet_csk(sk);
1777         struct sk_buff *skb, *nskb, *next;
1778         int len;
1779         int probe_size;
1780         int size_needed;
```

```
1781        int copy;
1782        int mss_now;
1783
1784        /* Not currently probing/verifying,
1785         * not in recovery,
1786         * have enough cwnd, and
1787         * not SACKing (the variable headers throw things off) */
1788        if (!icsk->icsk_mtup.enabled ||
1789            icsk->icsk_mtup.probe_size ||
1790            inet_csk(sk)->icsk_ca_state != TCP_CA_Open ||
1791            tp->snd_cwnd < 11 ||
1792            tp->rx_opt.num_sacks || tp->rx_opt.dsack)
1793                return -1;
1794
1795        /* Very simple search strategy: just double the MSS. */
1796        mss_now = tcp_current_mss(sk);
1797        probe_size = 2 * tp->mss_cache;
1798        size_needed = probe_size + (tp->reordering + 1) * tp->mss_cache;
1799        if (probe_size > tcp_mtu_to_mss(sk, icsk->icsk_mtup.search_high)) {
1800                /* TODO: set timer for probe_converge_event */
1801                return -1;
1802        }
1803
1804        /* Have enough data in the send queue to probe? */
1805        if (tp->write_seq - tp->snd_nxt < size_needed)
1806                return -1;
1807
1808        if (tp->snd_wnd < size_needed)
1809                return -1;
1810        if (after(tp->snd_nxt + size_needed, tcp_wnd_end(tp)))
1811                return 0;
1812
1813        /* Do we need to wait to drain cwnd? With none in flight, don't stall */
1814        if (tcp_packets_in_flight(tp) + 2 > tp->snd_cwnd) {
1815                if (!tcp_packets_in_flight(tp))
1816                        return -1;
1817                else
1818                        return 0;
1819        }
1820
1821        /* We're allowed to probe.  Build it now. */
1822        if ((nskb = sk_stream_alloc_skb(sk, probe_size, GFP_ATOMIC)) == NULL)
1823                return -1;
1824        sk->sk_wmem_queued += nskb->truesize;
1825        sk_mem_charge(sk, nskb->truesize);
1826
1827        skb = tcp_send_head(sk);
1828
1829        TCP_SKB_CB(nskb)->seq = TCP_SKB_CB(skb)->seq;
1830        TCP_SKB_CB(nskb)->end_seq = TCP_SKB_CB(skb)->seq + probe_size;
1831        TCP_SKB_CB(nskb)->tcp_flags = TCPHDR_ACK;
1832        TCP_SKB_CB(nskb)->sacked = 0;
1833        nskb->csum = 0;
1834        nskb->ip_summed = skb->ip_summed;
1835
1836        tcp_insert_write_queue_before(nskb, skb, sk);
1837
1838        len = 0;
1839        tcp_for_write_queue_from_safe(skb, next, sk) {
1840                copy = min_t(int, skb->len, probe_size - len);
1841                if (nskb->ip_summed)
1842                        skb_copy_bits(skb, 0, skb_put(nskb, copy), copy);
1843                else
1844                        nskb->csum = skb_copy_and_csum_bits(skb, 0,
1845                                                            skb_put(nskb, copy),
1846                                                            copy, nskb->csum);
1847
1848                if (skb->len <= copy) {
1849                        /* We've eaten all the data from this skb.
1850                         * Throw it away. */
1851                        TCP_SKB_CB(nskb)->tcp_flags |= TCP_SKB_CB(skb)->tcp_flags;
1852                        tcp_unlink_write_queue(skb, sk);
1853                        sk_wmem_free_skb(sk, skb);
1854                } else {
1855                        TCP_SKB_CB(nskb)->tcp_flags |= TCP_SKB_CB(skb)->tcp_flags &
1856                                                       ~(TCPHDR_FIN|TCPHDR_PSH);
1857                        if (!skb_shinfo(skb)->nr_frags) {
1858                                skb_pull(skb, copy);
1859                                if (skb->ip_summed != CHECKSUM_PARTIAL)
1860                                        skb->csum = csum_partial(skb->data,
1861                                                                 skb->len, 0);
1862                        } else {
1863                                __pskb_trim_head(skb, copy);
1864                                tcp_set_skb_tso_segs(sk, skb, mss_now);
1865                        }
1866                        TCP_SKB_CB(skb)->seq += copy;
1867                }
1868
1869                len += copy;
1870
```

```
1871                if (len >= probe_size)
1872                        break;
1873        }
1874        tcp_init_tso_segs(sk, nskb, nskb->len);
1875
1876        /* We're ready to send.  If this fails, the probe will
1877         * be resegmented into mss-sized pieces by tcp_write_xmit(). */
1878        TCP_SKB_CB(nskb)->when = tcp_time_stamp;
1879        if (!tcp_transmit_skb(sk, nskb, 1, GFP_ATOMIC)) {
1880                /* Decrement cwnd here because we are sending
1881                 * effectively two packets. */
1882                tp->snd_cwnd--;
1883                tcp_event_new_data_sent(sk, nskb);
1884
1885                icsk->icsk_mtup.probe_size = tcp_mss_to_mtu(sk, nskb->len);
1886                tp->mtu_probe.probe_seq_start = TCP_SKB_CB(nskb)->seq;
1887                tp->mtu_probe.probe_seq_end = TCP_SKB_CB(nskb)->end_seq;
1888
1889                return 1;
1890        }
1891
1892        return -1;
1893 }
1894
1895 /* This routine writes packets to the network.  It advances the
1896  * send_head.  This happens as incoming acks open up the remote
1897  * window for us.
1898  *
1899  * LARGESEND note: !tcp_urg_mode is overkill, only frames between
1900  * snd_up-64k-mss .. snd_up cannot be large. However, taking into
1901  * account rare use of URG, this is not a big flaw.
1902  *
1903  * Send at most one packet when push_one > 0. Temporarily ignore
1904  * cwnd limit to force at most one packet out when push_one == 2.
1905
1906  * Returns true, if no segments are in flight and we have queued segments,
1907  * but cannot send anything now because of SWS or another problem.
1908  */
1909 static bool tcp_write_xmit(struct sock *sk, unsigned int mss_now, int nonagle,
1910                            int push_one, gfp_t gfp)
1911 {
1912        struct tcp_sock *tp = tcp_sk(sk);
1913        struct sk_buff *skb;
1914        unsigned int tso_segs, sent_pkts;
1915        int cwnd_quota;
1916        int result;
1917        bool is_cwnd_limited = false;
1918
1919        sent_pkts = 0;
1920
1921        if (!push_one) {
1922                /* Do MTU probing. */
1923                result = tcp_mtu_probe(sk);
1924                if (!result) {
1925                        return false;
1926                } else if (result > 0) {
1927                        sent_pkts = 1;
1928                }
1929        }
1930
1931        while ((skb = tcp_send_head(sk))) {
1932                unsigned int limit;
1933
1934                tso_segs = tcp_init_tso_segs(sk, skb, mss_now);
1935                BUG_ON(!tso_segs);
1936
1937                if (unlikely(tp->repair) && tp->repair_queue == TCP_SEND_QUEUE) {
1938                        /* "when" is used as a start point for the retransmit timer */
1939                        TCP_SKB_CB(skb)->when = tcp_time_stamp;
1940                        goto repair; /* Skip network transmission */
1941                }
1942
1943                cwnd_quota = tcp_cwnd_test(tp, skb);
1944                if (!cwnd_quota) {
1945                        is_cwnd_limited = true;
1946                        if (push_one == 2)
1947                                /* Force out a loss probe pkt. */
1948                                cwnd_quota = 1;
1949                        else
1950                                break;
1951                }
1952
1953                if (unlikely(!tcp_snd_wnd_test(tp, skb, mss_now)))
1954                        break;
1955
1956                if (tso_segs == 1) {
1957                        if (unlikely(!tcp_nagle_test(tp, skb, mss_now,
1958                                                     (tcp_skb_is_last(sk, skb) ?
1959                                                      nonagle : TCP_NAGLE_PUSH))))
1960                                break;
```

```
1961                } else {
1962                        if (!push_one &&
1963                            tcp_tso_should_defer(sk, skb, &is_cwnd_limited))
1964                                break;
1965                }
1966
1967                /* TCP Small Queues :
1968                 * Control number of packets in qdisc/devices to two packets / or ~1 ms.
1969                 * This allows for :
1970                 *  - better RTT estimation and ACK scheduling
1971                 *  - faster recovery
1972                 *  - high rates
1973                 * Alas, some drivers / subsystems require a fair amount
1974                 * of queued bytes to ensure line rate.
1975                 * One example is wifi aggregation (802.11 AMPDU)
1976                 */
1977                limit = max_t(unsigned int, sysctl_tcp_limit_output_bytes,
1978                              sk->sk_pacing_rate >> 10);
1979
1980                if (atomic_read(&sk->sk_wmem_alloc) > limit) {
1981                        set_bit(TSQ_THROTTLED, &tp->tsq_flags);
1982                        /* It is possible TX completion already happened
1983                         * before we set TSQ_THROTTLED, so we must
1984                         * test again the condition.
1985                         */
1986                        smp_mb__after_atomic();
1987                        if (atomic_read(&sk->sk_wmem_alloc) > limit)
1988                                break;
1989                }
1990
1991                limit = mss_now;
1992                if (tso_segs > 1 && !tcp_urg_mode(tp))
1993                        limit = tcp_mss_split_point(sk, skb, mss_now,
1994                                                    min_t(unsigned int,
1995                                                          cwnd_quota,
1996                                                          sk->sk_gso_max_segs),
1997                                                    nonagle);
1998
1999                if (skb->len > limit &&
2000                    unlikely(tso_fragment(sk, skb, limit, mss_now, gfp)))
2001                        break;
2002
2003                TCP_SKB_CB(skb)->when = tcp_time_stamp;
2004
2005                if (unlikely(tcp_transmit_skb(sk, skb, 1, gfp)))
2006                        break;
2007
2008 repair:
2009                /* Advance the send_head.  This one is sent out.
2010                 * This call will increment packets_out.
2011                 */
2012                tcp_event_new_data_sent(sk, skb);
2013
2014                tcp_minshall_update(tp, mss_now, skb);
2015                sent_pkts += tcp_skb_pcount(skb);
2016
2017                if (push_one)
2018                        break;
2019        }
2020
2021        if (likely(sent_pkts)) {
2022                if (tcp_in_cwnd_reduction(sk))
2023                        tp->prr_out += sent_pkts;
2024
2025                /* Send one loss probe per tail loss episode. */
2026                if (push_one != 2)
2027                        tcp_schedule_loss_probe(sk);
2028                tcp_cwnd_validate(sk, is_cwnd_limited);
2029                return false;
2030        }
2031        return (push_one == 2) || (!tp->packets_out && tcp_send_head(sk));
2032 }
2033
2034 bool tcp_schedule_loss_probe(struct sock *sk)
2035 {
2036        struct inet_connection_sock *icsk = inet_csk(sk);
2037        struct tcp_sock *tp = tcp_sk(sk);
2038        u32 timeout, tlp_time_stamp, rto_time_stamp;
2039        u32 rtt = usecs_to_jiffies(tp->srtt_us >> 3);
2040
2041        if (WARN_ON(icsk->icsk_pending == ICSK_TIME_EARLY_RETRANS))
2042                return false;
2043        /* No consecutive loss probes. */
2044        if (WARN_ON(icsk->icsk_pending == ICSK_TIME_LOSS_PROBE)) {
2045                tcp_rearm_rto(sk);
2046                return false;
2047        }
2048        /* Don't do any loss probe on a Fast Open connection before 3WHS
2049         * finishes.
2050         */
```

```
2051         if (sk->sk_state == TCP_SYN_RECV)
2052                 return false;
2053
2054         /* TLP is only scheduled when next timer event is RTO. */
2055         if (icsk->icsk_pending != ICSK_TIME_RETRANS)
2056                 return false;
2057
2058         /* Schedule a loss probe in 2*RTT for SACK capable connections
2059          * in Open state, that are either limited by cwnd or application.
2060          */
2061         if (sysctl_tcp_early_retrans < 3 || !tp->srtt_us || !tp->packets_out ||
2062             !tcp_is_sack(tp) || inet_csk(sk)->icsk_ca_state != TCP_CA_Open)
2063                 return false;
2064
2065         if ((tp->snd_cwnd > tcp_packets_in_flight(tp)) &&
2066              tcp_send_head(sk))
2067                 return false;
2068
2069         /* Probe timeout is at least 1.5*rtt + TCP_DELACK_MAX to account
2070          * for delayed ack when there's one outstanding packet.
2071          */
2072         timeout = rtt << 1;
2073         if (tp->packets_out == 1)
2074                 timeout = max_t(u32, timeout,
2075                                 (rtt + (rtt >> 1) + TCP_DELACK_MAX));
2076         timeout = max_t(u32, timeout, msecs_to_jiffies(10));
2077
2078         /* If RTO is shorter, just schedule TLP in its place. */
2079         tlp_time_stamp = tcp_time_stamp + timeout;
2080         rto_time_stamp = (u32)inet_csk(sk)->icsk_timeout;
2081         if ((s32)(tlp_time_stamp - rto_time_stamp) > 0) {
2082                 s32 delta = rto_time_stamp - tcp_time_stamp;
2083                 if (delta > 0)
2084                         timeout = delta;
2085         }
2086
2087         inet_csk_reset_xmit_timer(sk, ICSK_TIME_LOSS_PROBE, timeout,
2088                                   TCP_RTO_MAX);
2089         return true;
2090 }
2091
2092 /* Thanks to skb fast clones, we can detect if a prior transmit of
2093  * a packet is still in a qdisc or driver queue.
2094  * In this case, there is very little point doing a retransmit !
2095  * Note: This is called from BH context only.
2096  */
2097 static bool skb_still_in_host_queue(const struct sock *sk,
2098                                     const struct sk_buff *skb)
2099 {
2100         const struct sk_buff *fclone = skb + 1;
2101
2102         if (unlikely(skb->fclone == SKB_FCLONE_ORIG &&
2103                      fclone->fclone == SKB_FCLONE_CLONE)) {
2104                 NET_INC_STATS_BH(sock_net(sk),
2105                                  LINUX_MIB_TCPSPURIOUS_RTX_HOSTQUEUES);
2106                 return true;
2107         }
2108         return false;
2109 }
2110
2111 /* When probe timeout (PTO) fires, send a new segment if one exists, else
2112  * retransmit the last segment.
2113  */
2114 void tcp_send_loss_probe(struct sock *sk)
2115 {
2116         struct tcp_sock *tp = tcp_sk(sk);
2117         struct sk_buff *skb;
2118         int pcount;
2119         int mss = tcp_current_mss(sk);
2120         int err = -1;
2121
2122         if (tcp_send_head(sk) != NULL) {
2123                 err = tcp_write_xmit(sk, mss, TCP_NAGLE_OFF, 2, GFP_ATOMIC);
2124                 goto rearm_timer;
2125         }
2126
2127         /* At most one outstanding TLP retransmission. */
2128         if (tp->tlp_high_seq)
2129                 goto rearm_timer;
2130
2131         /* Retransmit last segment. */
2132         skb = tcp_write_queue_tail(sk);
2133         if (WARN_ON(!skb))
2134                 goto rearm_timer;
2135
2136         if (skb_still_in_host_queue(sk, skb))
2137                 goto rearm_timer;
2138
2139         pcount = tcp_skb_pcount(skb);
2140         if (WARN_ON(!pcount))
```

```
2141                goto rearm_timer;
2142
2143        if ((pcount > 1) && (skb->len > (pcount - 1) * mss)) {
2144                if (unlikely(tcp_fragment(sk, skb, (pcount - 1) * mss, mss,
2145                                          GFP_ATOMIC)))
2146                        goto rearm_timer;
2147                skb = tcp_write_queue_tail(sk);
2148        }
2149
2150        if (WARN_ON(!skb || !tcp_skb_pcount(skb)))
2151                goto rearm_timer;
2152
2153        err = __tcp_retransmit_skb(sk, skb);
2154
2155        /* Record snd_nxt for loss detection. */
2156        if (likely(!err))
2157                tp->tlp_high_seq = tp->snd_nxt;
2158
2159 rearm_timer:
2160        inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
2161                                  inet_csk(sk)->icsk_rto,
2162                                  TCP_RTO_MAX);
2163
2164        if (likely(!err))
2165                NET_INC_STATS_BH(sock_net(sk),
2166                                 LINUX_MIB_TCPLOSSPROBES);
2167 }
2168
2169 /* Push out any pending frames which were held back due to
2170  * TCP_CORK or attempt at coalescing tiny packets.
2171  * The socket must be locked by the caller.
2172  */
2173 void __tcp_push_pending_frames(struct sock *sk, unsigned int cur_mss,
2174                                int nonagle)
2175 {
2176        /* If we are closed, the bytes will have to remain here.
2177         * In time closedown will finish, we empty the write queue and
2178         * all will be happy.
2179         */
2180        if (unlikely(sk->sk_state == TCP_CLOSE))
2181                return;
2182
2183        if (tcp_write_xmit(sk, cur_mss, nonagle, 0,
2184                           sk_gfp_atomic(sk, GFP_ATOMIC)))
2185                tcp_check_probe_timer(sk);
2186 }
2187
2188 /* Send _single_ skb sitting at the send head. This function requires
2189  * true push pending frames to setup probe timer etc.
2190  */
2191 void tcp_push_one(struct sock *sk, unsigned int mss_now)
2192 {
2193        struct sk_buff *skb = tcp_send_head(sk);
2194
2195        BUG_ON(!skb || skb->len < mss_now);
2196
2197        tcp_write_xmit(sk, mss_now, TCP_NAGLE_PUSH, 1, sk->sk_allocation);
2198 }
2199
2200 /* This function returns the amount that we can raise the
2201  * usable window based on the following constraints
2202  *
2203  * 1. The window can never be shrunk once it is offered (RFC 793)
2204  * 2. We limit memory per socket
2205  *
2206  * RFC 1122:
2207  * "the suggested [SWS] avoidance algorithm for the receiver is to keep
2208  *  RECV.NEXT + RCV.WIN fixed until:
2209  *  RCV.BUFF - RCV.USER - RCV.WINDOW >= min(1/2 RCV.BUFF, MSS)"
2210  *
2211  * i.e. don't raise the right edge of the window until you can raise
2212  * it at least MSS bytes.
2213  *
2214  * Unfortunately, the recommended algorithm breaks header prediction,
2215  * since header prediction assumes th->window stays fixed.
2216  *
2217  * Strictly speaking, keeping th->window fixed violates the receiver
2218  * side SWS prevention criteria. The problem is that under this rule
2219  * a stream of single byte packets will cause the right side of the
2220  * window to always advance by a single byte.
2221  *
2222  * Of course, if the sender implements sender side SWS prevention
2223  * then this will not be a problem.
2224  *
2225  * BSD seems to make the following compromise:
2226  *
2227  *      If the free space is less than the 1/4 of the maximum
2228  *      space available and the free space is less than 1/2 mss,
2229  *      then set the window to 0.
2230  *      [ Actually, bsd uses MSS and 1/4 of maximal _window_ ]
```

```
2231  *      Otherwise, just prevent the window from shrinking
2232  *      and from being larger than the largest representable value.
2233  *
2234  * This prevents incremental opening of the window in the regime
2235  * where TCP is limited by the speed of the reader side taking
2236  * data out of the TCP receive queue. It does nothing about
2237  * those cases where the window is constrained on the sender side
2238  * because the pipeline is full.
2239  *
2240  * BSD also seems to "accidentally" limit itself to windows that are a
2241  * multiple of MSS, at least until the free space gets quite small.
2242  * This would appear to be a side effect of the mbuf implementation.
2243  * Combining these two algorithms results in the observed behavior
2244  * of having a fixed window size at almost all times.
2245  *
2246  * Below we obtain similar behavior by forcing the offered window to
2247  * a multiple of the mss when it is feasible to do so.
2248  *
2249  * Note, we don't "adjust" for TIMESTAMP or SACK option bytes.
2250  * Regular options like TIMESTAMP are taken into account.
2251  */
2252 u32 __tcp_select_window(struct sock *sk)
2253 {
2254         struct inet_connection_sock *icsk = inet_csk(sk);
2255         struct tcp_sock *tp = tcp_sk(sk);
2256         /* MSS for the peer's data.  Previous versions used mss_clamp
2257          * here.  I don't know if the value based on our guesses
2258          * of peer's MSS is better for the performance.  It's more correct
2259          * but may be worse for the performance because of rcv_mss
2260          * fluctuations.  --SAW  1998/11/1
2261          */
2262         int mss = icsk->icsk_ack.rcv_mss;
2263         int free_space = tcp_space(sk);
2264         int allowed_space = tcp_full_space(sk);
2265         int full_space = min_t(int, tp->window_clamp, allowed_space);
2266         int window;
2267
2268         if (mss > full_space)
2269                 mss = full_space;
2270
2271         if (free_space < (full_space >> 1)) {
2272                 icsk->icsk_ack.quick = 0;
2273
2274                 if (sk_under_memory_pressure(sk))
2275                         tp->rcv_ssthresh = min(tp->rcv_ssthresh,
2276                                                4U * tp->advmss);
2277
2278                 /* free_space might become our new window, make sure we don't
2279                  * increase it due to wscale.
2280                  */
2281                 free_space = round_down(free_space, 1 << tp->rx_opt.rcv_wscale);
2282
2283                 /* if free space is less than mss estimate, or is below 1/16th
2284                  * of the maximum allowed, try to move to zero-window, else
2285                  * tcp_clamp_window() will grow rcv buf up to tcp_rmem[2], and
2286                  * new incoming data is dropped due to memory limits.
2287                  * With large window, mss test triggers way too late in order
2288                  * to announce zero window in time before rmem limit kicks in.
2289                  */
2290                 if (free_space < (allowed_space >> 4) || free_space < mss)
2291                         return 0;
2292         }
2293
2294         if (free_space > tp->rcv_ssthresh)
2295                 free_space = tp->rcv_ssthresh;
2296
2297         /* Don't do rounding if we are using window scaling, since the
2298          * scaled window will not line up with the MSS boundary anyway.
2299          */
2300         window = tp->rcv_wnd;
2301         if (tp->rx_opt.rcv_wscale) {
2302                 window = free_space;
2303
2304                 /* Advertise enough space so that it won't get scaled away.
2305                  * Import case: prevent zero window announcement if
2306                  * 1<<rcv_wscale > mss.
2307                  */
2308                 if (((window >> tp->rx_opt.rcv_wscale) << tp->rx_opt.rcv_wscale) != window)
2309                         window = (((window >> tp->rx_opt.rcv_wscale) + 1)
2310                                   << tp->rx_opt.rcv_wscale);
2311         } else {
2312                 /* Get the largest window that is a nice multiple of mss.
2313                  * Window clamp already applied above.
2314                  * If our current window offering is within 1 mss of the
2315                  * free space we just keep it. This prevents the divide
2316                  * and multiply from happening most of the time.
2317                  * We also don't do any window rounding when the free space
2318                  * is too small.
2319                  */
2320                 if (window <= free_space - mss || window > free_space)
```

```
2321                        window = (free_space / mss) * mss;
2322                else if (mss == full_space &&
2323                         free_space > window + (full_space >> 1))
2324                        window = free_space;
2325        }
2326
2327        return window;
2328 }
2329
2330 /* Collapses two adjacent SKB's during retransmission. */
2331 static void tcp_collapse_retrans(struct sock *sk, struct sk_buff *skb)
2332 {
2333        struct tcp_sock *tp = tcp_sk(sk);
2334        struct sk_buff *next_skb = tcp_write_queue_next(sk, skb);
2335        int skb_size, next_skb_size;
2336
2337        skb_size = skb->len;
2338        next_skb_size = next_skb->len;
2339
2340        BUG_ON(tcp_skb_pcount(skb) != 1 || tcp_skb_pcount(next_skb) != 1);
2341
2342        tcp_highest_sack_combine(sk, next_skb, skb);
2343
2344        tcp_unlink_write_queue(next_skb, sk);
2345
2346        skb_copy_from_linear_data(next_skb, skb_put(skb, next_skb_size),
2347                                  next_skb_size);
2348
2349        if (next_skb->ip_summed == CHECKSUM_PARTIAL)
2350                skb->ip_summed = CHECKSUM_PARTIAL;
2351
2352        if (skb->ip_summed != CHECKSUM_PARTIAL)
2353                skb->csum = csum_block_add(skb->csum, next_skb->csum, skb_size);
2354
2355        /* Update sequence range on original skb. */
2356        TCP_SKB_CB(skb)->end_seq = TCP_SKB_CB(next_skb)->end_seq;
2357
2358        /* Merge over control information. This moves PSH/FIN etc. over */
2359        TCP_SKB_CB(skb)->tcp_flags |= TCP_SKB_CB(next_skb)->tcp_flags;
2360
2361        /* All done, get rid of second SKB and account for it so
2362         * packet counting does not break.
2363         */
2364        TCP_SKB_CB(skb)->sacked |= TCP_SKB_CB(next_skb)->sacked & TCPCB_EVER_RETRANS;
2365
2366        /* changed transmit queue under us so clear hints */
2367        tcp_clear_retrans_hints_partial(tp);
2368        if (next_skb == tp->retransmit_skb_hint)
2369                tp->retransmit_skb_hint = skb;
2370
2371        tcp_adjust_pcount(sk, next_skb, tcp_skb_pcount(next_skb));
2372
2373        sk_wmem_free_skb(sk, next_skb);
2374 }
2375
2376 /* Check if coalescing SKBs is legal. */
2377 static bool tcp_can_collapse(const struct sock *sk, const struct sk_buff *skb)
2378 {
2379        if (tcp_skb_pcount(skb) > 1)
2380                return false;
2381        /* TODO: SACK collapsing could be used to remove this condition */
2382        if (skb_shinfo(skb)->nr_frags != 0)
2383                return false;
2384        if (skb_cloned(skb))
2385                return false;
2386        if (skb == tcp_send_head(sk))
2387                return false;
2388        /* Some heurestics for collapsing over SACK'd could be invented */
2389        if (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED)
2390                return false;
2391
2392        return true;
2393 }
2394
2395 /* Collapse packets in the retransmit queue to make to create
2396  * less packets on the wire. This is only done on retransmission.
2397  */
2398 static void tcp_retrans_try_collapse(struct sock *sk, struct sk_buff *to,
2399                                      int space)
2400 {
2401        struct tcp_sock *tp = tcp_sk(sk);
2402        struct sk_buff *skb = to, *tmp;
2403        bool first = true;
2404
2405        if (!sysctl_tcp_retrans_collapse)
2406                return;
2407        if (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_SYN)
2408                return;
2409
2410        tcp_for_write_queue_from_safe(skb, tmp, sk) {
```

```
2411                if (!tcp_can_collapse(sk, skb))
2412                        break;
2413
2414                space -= skb->len;
2415
2416                if (first) {
2417                        first = false;
2418                        continue;
2419                }
2420
2421                if (space < 0)
2422                        break;
2423                /* Punt if not enough space exists in the first SKB for
2424                 * the data in the second
2425                 */
2426                if (skb->len > skb_availroom(to))
2427                        break;
2428
2429                if (after(TCP_SKB_CB(skb)->end_seq, tcp_wnd_end(tp)))
2430                        break;
2431
2432                tcp_collapse_retrans(sk, to);
2433        }
2434 }
2435
2436 /* This retransmits one SKB.  Policy decisions and retransmit queue
2437  * state updates are done by the caller.  Returns non-zero if an
2438  * error occurred which prevented the send.
2439  */
2440 int __tcp_retransmit_skb(struct sock *sk, struct sk_buff *skb)
2441 {
2442        struct tcp_sock *tp = tcp_sk(sk);
2443        struct inet_connection_sock *icsk = inet_csk(sk);
2444        unsigned int cur_mss;
2445        int err;
2446
2447        /* Inconslusive MTU probe */
2448        if (icsk->icsk_mtup.probe_size) {
2449                icsk->icsk_mtup.probe_size = 0;
2450        }
2451
2452        /* Do not sent more than we queued. 1/4 is reserved for possible
2453         * copying overhead: fragmentation, tunneling, mangling etc.
2454         */
2455        if (atomic_read(&sk->sk_wmem_alloc) >
2456            min(sk->sk_wmem_queued + (sk->sk_wmem_queued >> 2), sk->sk_sndbuf))
2457                return -EAGAIN;
2458
2459        if (skb_still_in_host_queue(sk, skb))
2460                return -EBUSY;
2461
2462        if (before(TCP_SKB_CB(skb)->seq, tp->snd_una)) {
2463                if (before(TCP_SKB_CB(skb)->end_seq, tp->snd_una))
2464                        BUG();
2465                if (tcp_trim_head(sk, skb, tp->snd_una - TCP_SKB_CB(skb)->seq))
2466                        return -ENOMEM;
2467        }
2468
2469        if (inet_csk(sk)->icsk_af_ops->rebuild_header(sk))
2470                return -EHOSTUNREACH; /* Routing failure or similar. */
2471
2472        cur_mss = tcp_current_mss(sk);
2473
2474        /* If receiver has shrunk his window, and skb is out of
2475         * new window, do not retransmit it. The exception is the
2476         * case, when window is shrunk to zero. In this case
2477         * our retransmit serves as a zero window probe.
2478         */
2479        if (!before(TCP_SKB_CB(skb)->seq, tcp_wnd_end(tp)) &&
2480            TCP_SKB_CB(skb)->seq != tp->snd_una)
2481                return -EAGAIN;
2482
2483        if (skb->len > cur_mss) {
2484                if (tcp_fragment(sk, skb, cur_mss, cur_mss, GFP_ATOMIC))
2485                        return -ENOMEM; /* We'll try again later. */
2486        } else {
2487                int oldpcount = tcp_skb_pcount(skb);
2488
2489                if (unlikely(oldpcount > 1)) {
2490                        if (skb_unclone(skb, GFP_ATOMIC))
2491                                return -ENOMEM;
2492                        tcp_init_tso_segs(sk, skb, cur_mss);
2493                        tcp_adjust_pcount(sk, skb, oldpcount - tcp_skb_pcount(skb));
2494                }
2495        }
2496
2497        tcp_retrans_try_collapse(sk, skb, cur_mss);
2498
2499        /* Make a copy, if the first transmission SKB clone we made
2500         * is still in somebody's hands, else make a clone.
```

```
2501          */
2502         TCP_SKB_CB(skb)->when = tcp_time_stamp;
2503
2504         /* make sure skb->data is aligned on arches that require it
2505          * and check if ack-trimming & collapsing extended the headroom
2506          * beyond what csum_start can cover.
2507          */
2508         if (unlikely((NET_IP_ALIGN && ((unsigned long)skb->data & 3)) ||
2509                      skb_headroom(skb) >= 0xFFFF)) {
2510                 struct sk_buff *nskb = __pskb_copy(skb, MAX_TCP_HEADER,
2511                                                    GFP_ATOMIC);
2512                 err = nskb ? tcp_transmit_skb(sk, nskb, 0, GFP_ATOMIC) :
2513                              -ENOBUFS;
2514         } else {
2515                 err = tcp_transmit_skb(sk, skb, 1, GFP_ATOMIC);
2516         }
2517
2518         if (likely(!err)) {
2519                 TCP_SKB_CB(skb)->sacked |= TCPCB_EVER_RETRANS;
2520                 /* Update global TCP statistics. */
2521                 TCP_INC_STATS(sock_net(sk), TCP_MIB_RETRANSSEGS);
2522                 if (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_SYN)
2523                         NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPSYNRETRANS);
2524                 tp->total_retrans++;
2525         }
2526         return err;
2527 }
2528
2529 int tcp_retransmit_skb(struct sock *sk, struct sk_buff *skb)
2530 {
2531         struct tcp_sock *tp = tcp_sk(sk);
2532         int err = __tcp_retransmit_skb(sk, skb);
2533
2534         if (err == 0) {
2535 #if FASTRETRANS_DEBUG > 0
2536                 if (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_RETRANS) {
2537                         net_dbg_ratelimited("retrans_out leaked\n");
2538                 }
2539 #endif
2540                 if (!tp->retrans_out)
2541                         tp->lost_retrans_low = tp->snd_nxt;
2542                 TCP_SKB_CB(skb)->sacked |= TCPCB_RETRANS;
2543                 tp->retrans_out += tcp_skb_pcount(skb);
2544
2545                 /* Save stamp of the first retransmit. */
2546                 if (!tp->retrans_stamp)
2547                         tp->retrans_stamp = TCP_SKB_CB(skb)->when;
2548
2549                 /* snd_nxt is stored to detect loss of retransmitted segment,
2550                  * see tcp_input.c tcp_sacktag_write_queue().
2551                  */
2552                 TCP_SKB_CB(skb)->ack_seq = tp->snd_nxt;
2553         } else if (err != -EBUSY) {
2554                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPRETRANSFAIL);
2555         }
2556
2557         if (tp->undo_retrans < 0)
2558                 tp->undo_retrans = 0;
2559         tp->undo_retrans += tcp_skb_pcount(skb);
2560         return err;
2561 }
2562
2563 /* Check if we forward retransmits are possible in the current
2564  * window/congestion state.
2565  */
2566 static bool tcp_can_forward_retransmit(struct sock *sk)
2567 {
2568         const struct inet_connection_sock *icsk = inet_csk(sk);
2569         const struct tcp_sock *tp = tcp_sk(sk);
2570
2571         /* Forward retransmissions are possible only during Recovery. */
2572         if (icsk->icsk_ca_state != TCP_CA_Recovery)
2573                 return false;
2574
2575         /* No forward retransmissions in Reno are possible. */
2576         if (tcp_is_reno(tp))
2577                 return false;
2578
2579         /* Yeah, we have to make difficult choice between forward transmission
2580          * and retransmission... Both ways have their merits...
2581          *
2582          * For now we do not retransmit anything, while we have some new
2583          * segments to send. In the other cases, follow rule 3 for
2584          * NextSeg() specified in RFC3517.
2585          */
2586
2587         if (tcp_may_send_now(sk))
2588                 return false;
2589
2590         return true;
```

```
2591 }
2592
2593 /* This gets called after a retransmit timeout, and the initially
2594  * retransmitted data is acknowledged.  It tries to continue
2595  * resending the rest of the retransmit queue, until either
2596  * we've sent it all or the congestion window limit is reached.
2597  * If doing SACK, the first ACK which comes back for a timeout
2598  * based retransmit packet might feed us FACK information again.
2599  * If so, we use it to avoid unnecessarily retransmissions.
2600  */
2601 void tcp_xmit_retransmit_queue(struct sock *sk)
2602 {
2603         const struct inet_connection_sock *icsk = inet_csk(sk);
2604         struct tcp_sock *tp = tcp_sk(sk);
2605         struct sk_buff *skb;
2606         struct sk_buff *hole = NULL;
2607         u32 last_lost;
2608         int mib_idx;
2609         int fwd_rexmitting = 0;
2610
2611         if (!tp->packets_out)
2612                 return;
2613
2614         if (!tp->lost_out)
2615                 tp->retransmit_high = tp->snd_una;
2616
2617         if (tp->retransmit_skb_hint) {
2618                 skb = tp->retransmit_skb_hint;
2619                 last_lost = TCP_SKB_CB(skb)->end_seq;
2620                 if (after(last_lost, tp->retransmit_high))
2621                         last_lost = tp->retransmit_high;
2622         } else {
2623                 skb = tcp_write_queue_head(sk);
2624                 last_lost = tp->snd_una;
2625         }
2626
2627         tcp_for_write_queue_from(skb, sk) {
2628                 __u8 sacked = TCP_SKB_CB(skb)->sacked;
2629
2630                 if (skb == tcp_send_head(sk))
2631                         break;
2632                 /* we could do better than to assign each time */
2633                 if (hole == NULL)
2634                         tp->retransmit_skb_hint = skb;
2635
2636                 /* Assume this retransmit will generate
2637                  * only one packet for congestion window
2638                  * calculation purposes.  This works because
2639                  * tcp_retransmit_skb() will chop up the
2640                  * packet to be MSS sized and all the
2641                  * packet counting works out.
2642                  */
2643                 if (tcp_packets_in_flight(tp) >= tp->snd_cwnd)
2644                         return;
2645
2646                 if (fwd_rexmitting) {
2647 begin_fwd:
2648                         if (!before(TCP_SKB_CB(skb)->seq, tcp_highest_sack_seq(tp)))
2649                                 break;
2650                         mib_idx = LINUX_MIB_TCPFORWARDRETRANS;
2651
2652                 } else if (!before(TCP_SKB_CB(skb)->seq, tp->retransmit_high)) {
2653                         tp->retransmit_high = last_lost;
2654                         if (!tcp_can_forward_retransmit(sk))
2655                                 break;
2656                         /* Backtrack if necessary to non-L'ed skb */
2657                         if (hole != NULL) {
2658                                 skb = hole;
2659                                 hole = NULL;
2660                         }
2661                         fwd_rexmitting = 1;
2662                         goto begin_fwd;
2663
2664                 } else if (!(sacked & TCPCB_LOST)) {
2665                         if (hole == NULL && !(sacked & (TCPCB_SACKED_RETRANS|TCPCB_SACKED_ACKED)))
2666                                 hole = skb;
2667                         continue;
2668
2669                 } else {
2670                         last_lost = TCP_SKB_CB(skb)->end_seq;
2671                         if (icsk->icsk_ca_state != TCP_CA_Loss)
2672                                 mib_idx = LINUX_MIB_TCPFASTRETRANS;
2673                         else
2674                                 mib_idx = LINUX_MIB_TCPSLOWSTARTRETRANS;
2675                 }
2676
2677                 if (sacked & (TCPCB_SACKED_ACKED|TCPCB_SACKED_RETRANS))
2678                         continue;
2679
2680                 if (tcp_retransmit_skb(sk, skb))
```

```
2681                        return;
2682
2683                NET_INC_STATS_BH(sock_net(sk), mib_idx);
2684
2685                if (tcp_in_cwnd_reduction(sk))
2686                        tp->prr_out += tcp_skb_pcount(skb);
2687
2688                if (skb == tcp_write_queue_head(sk))
2689                        inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
2690                                                  inet_csk(sk)->icsk_rto,
2691                                                  TCP_RTO_MAX);
2692        }
2693 }
2694
2695 /* Send a fin.  The caller locks the socket for us.  This cannot be
2696  * allowed to fail queueing a FIN frame under any circumstances.
2697  */
2698 void tcp_send_fin(struct sock *sk)
2699 {
2700        struct tcp_sock *tp = tcp_sk(sk);
2701        struct sk_buff *skb = tcp_write_queue_tail(sk);
2702        int mss_now;
2703
2704        /* Optimization, tack on the FIN if we have a queue of
2705         * unsent frames.  But be careful about outgoing SACKS
2706         * and IP options.
2707         */
2708        mss_now = tcp_current_mss(sk);
2709
2710        if (tcp_send_head(sk) != NULL) {
2711                TCP_SKB_CB(skb)->tcp_flags |= TCPHDR_FIN;
2712                TCP_SKB_CB(skb)->end_seq++;
2713                tp->write_seq++;
2714        } else {
2715                /* Socket is locked, keep trying until memory is available. */
2716                for (;;) {
2717                        skb = alloc_skb_fclone(MAX_TCP_HEADER,
2718                                               sk->sk_allocation);
2719                        if (skb)
2720                                break;
2721                        yield();
2722                }
2723
2724                /* Reserve space for headers and prepare control bits. */
2725                skb_reserve(skb, MAX_TCP_HEADER);
2726                /* FIN eats a sequence byte, write_seq advanced by tcp_queue_skb(). */
2727                tcp_init_nondata_skb(skb, tp->write_seq,
2728                                     TCPHDR_ACK | TCPHDR_FIN);
2729                tcp_queue_skb(sk, skb);
2730        }
2731        __tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_OFF);
2732 }
2733
2734 /* We get here when a process closes a file descriptor (either due to
2735  * an explicit close() or as a byproduct of exit()'ing) and there
2736  * was unread data in the receive queue.  This behavior is recommended
2737  * by RFC 2525, section 2.17.  -DaveM
2738  */
2739 void tcp_send_active_reset(struct sock *sk, gfp_t priority)
2740 {
2741        struct sk_buff *skb;
2742
2743        /* NOTE: No TCP options attached and we never retransmit this. */
2744        skb = alloc_skb(MAX_TCP_HEADER, priority);
2745        if (!skb) {
2746                NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPABORTFAILED);
2747                return;
2748        }
2749
2750        /* Reserve space for headers and prepare control bits. */
2751        skb_reserve(skb, MAX_TCP_HEADER);
2752        tcp_init_nondata_skb(skb, tcp_acceptable_seq(sk),
2753                             TCPHDR_ACK | TCPHDR_RST);
2754        /* Send it off. */
2755        TCP_SKB_CB(skb)->when = tcp_time_stamp;
2756        if (tcp_transmit_skb(sk, skb, 0, priority))
2757                NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPABORTFAILED);
2758
2759        TCP_INC_STATS(sock_net(sk), TCP_MIB_OUTRSTS);
2760 }
2761
2762 /* Send a crossed SYN-ACK during socket establishment.
2763  * WARNING: This routine must only be called when we have already sent
2764  * a SYN packet that crossed the incoming SYN that caused this routine
2765  * to get called. If this assumption fails then the initial rcv_wnd
2766  * and rcv_wscale values will not be correct.
2767  */
2768 int tcp_send_synack(struct sock *sk)
2769 {
2770        struct sk_buff *skb;
```

```
2771
2772         skb = tcp_write_queue_head(sk);
2773         if (skb == NULL || !(TCP_SKB_CB(skb)->tcp_flags & TCPHDR_SYN)) {
2774                 pr_debug("%s: wrong queue state\n", __func__);
2775                 return -EFAULT;
2776         }
2777         if (!(TCP_SKB_CB(skb)->tcp_flags & TCPHDR_ACK)) {
2778                 if (skb_cloned(skb)) {
2779                         struct sk_buff *nskb = skb_copy(skb, GFP_ATOMIC);
2780                         if (nskb == NULL)
2781                                 return -ENOMEM;
2782                         tcp_unlink_write_queue(skb, sk);
2783                         skb_header_release(nskb);
2784                         __tcp_add_write_queue_head(sk, nskb);
2785                         sk_wmem_free_skb(sk, skb);
2786                         sk->sk_wmem_queued += nskb->truesize;
2787                         sk_mem_charge(sk, nskb->truesize);
2788                         skb = nskb;
2789                 }
2790
2791                 TCP_SKB_CB(skb)->tcp_flags |= TCPHDR_ACK;
2792                 TCP_ECN_send_synack(tcp_sk(sk), skb);
2793         }
2794         TCP_SKB_CB(skb)->when = tcp_time_stamp;
2795         return tcp_transmit_skb(sk, skb, 1, GFP_ATOMIC);
2796 }
2797
2798 /**
2799  * tcp_make_synack - Prepare a SYN-ACK.
2800  * sk: listener socket
2801  * dst: dst entry attached to the SYNACK
2802  * req: request_sock pointer
2803  *
2804  * Allocate one skb and build a SYNACK packet.
2805  * @dst is consumed : Caller should not use it again.
2806  */
2807 struct sk_buff *tcp_make_synack(struct sock *sk, struct dst_entry *dst,
2808                                 struct request_sock *req,
2809                                 struct tcp_fastopen_cookie *foc)
2810 {
2811         struct tcp_out_options opts;
2812         struct inet_request_sock *ireq = inet_rsk(req);
2813         struct tcp_sock *tp = tcp_sk(sk);
2814         struct tcphdr *th;
2815         struct sk_buff *skb;
2816         struct tcp_md5sig_key *md5;
2817         int tcp_header_size;
2818         int mss;
2819
2820         skb = sock_wmalloc(sk, MAX_TCP_HEADER, 1, GFP_ATOMIC);
2821         if (unlikely(!skb)) {
2822                 dst_release(dst);
2823                 return NULL;
2824         }
2825         /* Reserve space for headers. */
2826         skb_reserve(skb, MAX_TCP_HEADER);
2827
2828         skb_dst_set(skb, dst);
2829         security_skb_owned_by(skb, sk);
2830
2831         mss = dst_metric_advmss(dst);
2832         if (tp->rx_opt.user_mss && tp->rx_opt.user_mss < mss)
2833                 mss = tp->rx_opt.user_mss;
2834
2835         memset(&opts, 0, sizeof(opts));
2836 #ifdef CONFIG_SYN_COOKIES
2837         if (unlikely(req->cookie_ts))
2838                 TCP_SKB_CB(skb)->when = cookie_init_timestamp(req);
2839         else
2840 #endif
2841         TCP_SKB_CB(skb)->when = tcp_time_stamp;
2842         tcp_header_size = tcp_synack_options(sk, req, mss, skb, &opts, &md5,
2843                                              foc) + sizeof(*th);
2844
2845         skb_push(skb, tcp_header_size);
2846         skb_reset_transport_header(skb);
2847
2848         th = tcp_hdr(skb);
2849         memset(th, 0, sizeof(struct tcphdr));
2850         th->syn = 1;
2851         th->ack = 1;
2852         TCP_ECN_make_synack(req, th);
2853         th->source = htons(ireq->ir_num);
2854         th->dest = ireq->ir_rmt_port;
2855         /* Setting of flags are superfluous here for callers (and ECE is
2856          * not even correctly set)
2857          */
2858         tcp_init_nondata_skb(skb, tcp_rsk(req)->snt_isn,
2859                              TCPHDR_SYN | TCPHDR_ACK);
2860
```

```
2861         th->seq = htonl(TCP_SKB_CB(skb)->seq);
2862         /* XXX data is queued and acked as is. No buffer/window check */
2863         th->ack_seq = htonl(tcp_rsk(req)->rcv_nxt);
2864
2865         /* RFC1323: The window in SYN & SYN/ACK segments is never scaled. */
2866         th->window = htons(min(req->rcv_wnd, 65535U));
2867         tcp_options_write((__be32 *)(th + 1), tp, &opts);
2868         th->doff = (tcp_header_size >> 2);
2869         TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_OUTSEGS);
2870
2871 #ifdef CONFIG_TCP_MD5SIG
2872         /* Okay, we have all we need - do the md5 hash if needed */
2873         if (md5) {
2874                 tcp_rsk(req)->af_specific->calc_md5_hash(opts.hash_location,
2875                                                  md5, NULL, req, skb);
2876         }
2877 #endif
2878
2879         return skb;
2880 }
2881 EXPORT_SYMBOL(tcp_make_synack);
2882
2883 /* Do all connect socket setups that can be done AF independent. */
2884 static void tcp_connect_init(struct sock *sk)
2885 {
2886         const struct dst_entry *dst = __sk_dst_get(sk);
2887         struct tcp_sock *tp = tcp_sk(sk);
2888         __u8 rcv_wscale;
2889
2890         /* We'll fix this up when we get a response from the other end.
2891          * See tcp_input.c:tcp_rcv_state_process case TCP_SYN_SENT.
2892          */
2893         tp->tcp_header_len = sizeof(struct tcphdr) +
2894                 (sysctl_tcp_timestamps ? TCPOLEN_TSTAMP_ALIGNED : 0);
2895
2896 #ifdef CONFIG_TCP_MD5SIG
2897         if (tp->af_specific->md5_lookup(sk, sk) != NULL)
2898                 tp->tcp_header_len += TCPOLEN_MD5SIG_ALIGNED;
2899 #endif
2900
2901         /* If user gave his TCP_MAXSEG, record it to clamp */
2902         if (tp->rx_opt.user_mss)
2903                 tp->rx_opt.mss_clamp = tp->rx_opt.user_mss;
2904         tp->max_window = 0;
2905         tcp_mtup_init(sk);
2906         tcp_sync_mss(sk, dst_mtu(dst));
2907
2908         if (!tp->window_clamp)
2909                 tp->window_clamp = dst_metric(dst, RTAX_WINDOW);
2910         tp->advmss = dst_metric_advmss(dst);
2911         if (tp->rx_opt.user_mss && tp->rx_opt.user_mss < tp->advmss)
2912                 tp->advmss = tp->rx_opt.user_mss;
2913
2914         tcp_initialize_rcv_mss(sk);
2915
2916         /* limit the window selection if the user enforce a smaller rx buffer */
2917         if (sk->sk_userlocks & SOCK_RCVBUF_LOCK &&
2918             (tp->window_clamp > tcp_full_space(sk) || tp->window_clamp == 0))
2919                 tp->window_clamp = tcp_full_space(sk);
2920
2921         tcp_select_initial_window(tcp_full_space(sk),
2922                                   tp->advmss - (tp->rx_opt.ts_recent_stamp ? tp->tcp_header_len - sizeof(struct tcphdr) : 0),
2923                                   &tp->rcv_wnd,
2924                                   &tp->window_clamp,
2925                                   sysctl_tcp_window_scaling,
2926                                   &rcv_wscale,
2927                                   dst_metric(dst, RTAX_INITRWND));
2928
2929         tp->rx_opt.rcv_wscale = rcv_wscale;
2930         tp->rcv_ssthresh = tp->rcv_wnd;
2931
2932         sk->sk_err = 0;
2933         sock_reset_flag(sk, SOCK_DONE);
2934         tp->snd_wnd = 0;
2935         tcp_init_wl(tp, 0);
2936         tp->snd_una = tp->write_seq;
2937         tp->snd_sml = tp->write_seq;
2938         tp->snd_up = tp->write_seq;
2939         tp->snd_nxt = tp->write_seq;
2940
2941         if (likely(!tp->repair))
2942                 tp->rcv_nxt = 0;
2943         else
2944                 tp->rcv_tstamp = tcp_time_stamp;
2945         tp->rcv_wup = tp->rcv_nxt;
2946         tp->copied_seq = tp->rcv_nxt;
2947
2948         inet_csk(sk)->icsk_rto = TCP_TIMEOUT_INIT;
2949         inet_csk(sk)->icsk_retransmits = 0;
2950         tcp_clear_retrans(tp);
```

```
2951 }
2952
2953 static void tcp_connect_queue_skb(struct sock *sk, struct sk_buff *skb)
2954 {
2955         struct tcp_sock *tp = tcp_sk(sk);
2956         struct tcp_skb_cb *tcb = TCP_SKB_CB(skb);
2957
2958         tcb->end_seq += skb->len;
2959         skb_header_release(skb);
2960         __tcp_add_write_queue_tail(sk, skb);
2961         sk->sk_wmem_queued += skb->truesize;
2962         sk_mem_charge(sk, skb->truesize);
2963         tp->write_seq = tcb->end_seq;
2964         tp->packets_out += tcp_skb_pcount(skb);
2965 }
2966
2967 /* Build and send a SYN with data and (cached) Fast Open cookie. However,
2968  * queue a data-only packet after the regular SYN, such that regular SYNs
2969  * are retransmitted on timeouts. Also if the remote SYN-ACK acknowledges
2970  * only the SYN sequence, the data are retransmitted in the first ACK.
2971  * If cookie is not cached or other error occurs, falls back to send a
2972  * regular SYN with Fast Open cookie request option.
2973  */
2974 static int tcp_send_syn_data(struct sock *sk, struct sk_buff *syn)
2975 {
2976         struct tcp_sock *tp = tcp_sk(sk);
2977         struct tcp_fastopen_request *fo = tp->fastopen_req;
2978         int syn_loss = 0, space, i, err = 0, iovlen = fo->data->msg_iovlen;
2979         struct sk_buff *syn_data = NULL, *data;
2980         unsigned long last_syn_loss = 0;
2981
2982         tp->rx_opt.mss_clamp = tp->advmss;  /* If MSS is not cached */
2983         tcp_fastopen_cache_get(sk, &tp->rx_opt.mss_clamp, &fo->cookie,
2984                                &syn_loss, &last_syn_loss);
2985         /* Recurring FO SYN losses: revert to regular handshake temporarily */
2986         if (syn_loss > 1 &&
2987             time_before(jiffies, last_syn_loss + (60*HZ << syn_loss))) {
2988                 fo->cookie.len = -1;
2989                 goto fallback;
2990         }
2991
2992         if (sysctl_tcp_fastopen & TFO_CLIENT_NO_COOKIE)
2993                 fo->cookie.len = -1;
2994         else if (fo->cookie.len <= 0)
2995                 goto fallback;
2996
2997         /* MSS for SYN-data is based on cached MSS and bounded by PMTU and
2998          * user-MSS. Reserve maximum option space for middleboxes that add
2999          * private TCP options. The cost is reduced data space in SYN :(
3000          */
3001         if (tp->rx_opt.user_mss && tp->rx_opt.user_mss < tp->rx_opt.mss_clamp)
3002                 tp->rx_opt.mss_clamp = tp->rx_opt.user_mss;
3003         space = __tcp_mtu_to_mss(sk, inet_csk(sk)->icsk_pmtu_cookie) -
3004                 MAX_TCP_OPTION_SPACE;
3005
3006         space = min_t(size_t, space, fo->size);
3007
3008         /* limit to order-0 allocations */
3009         space = min_t(size_t, space, SKB_MAX_HEAD(MAX_TCP_HEADER));
3010
3011         syn_data = skb_copy_expand(syn, MAX_TCP_HEADER, space,
3012                                    sk->sk_allocation);
3013         if (syn_data == NULL)
3014                 goto fallback;
3015
3016         for (i = 0; i < iovlen && syn_data->len < space; ++i) {
3017                 struct iovec *iov = &fo->data->msg_iov[i];
3018                 unsigned char __user *from = iov->iov_base;
3019                 int len = iov->iov_len;
3020
3021                 if (syn_data->len + len > space)
3022                         len = space - syn_data->len;
3023                 else if (i + 1 == iovlen)
3024                         /* No more data pending in inet_wait_for_connect() */
3025                         fo->data = NULL;
3026
3027                 if (skb_add_data(syn_data, from, len))
3028                         goto fallback;
3029         }
3030
3031         /* Queue a data-only packet after the regular SYN for retransmission */
3032         data = pskb_copy(syn_data, sk->sk_allocation);
3033         if (data == NULL)
3034                 goto fallback;
3035         TCP_SKB_CB(data)->seq++;
3036         TCP_SKB_CB(data)->tcp_flags &= ~TCPHDR_SYN;
3037         TCP_SKB_CB(data)->tcp_flags = (TCPHDR_ACK|TCPHDR_PSH);
3038         tcp_connect_queue_skb(sk, data);
3039         fo->copied = data->len;
3040
```

```
3041         /* syn_data is about to be sent, we need to take current time stamps
3042          * for the packets that are in write queue : SYN packet and DATA
3043          */
3044         skb_mstamp_get(&syn->skb_mstamp);
3045         data->skb_mstamp = syn->skb_mstamp;
3046
3047         if (tcp_transmit_skb(sk, syn_data, 0, sk->sk_allocation) == 0) {
3048                 tp->syn_data = (fo->copied > 0);
3049                 NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPORIGDATASENT);
3050                 goto done;
3051         }
3052         syn_data = NULL;
3053
3054 fallback:
3055         /* Send a regular SYN with Fast Open cookie request option */
3056         if (fo->cookie.len > 0)
3057                 fo->cookie.len = 0;
3058         err = tcp_transmit_skb(sk, syn, 1, sk->sk_allocation);
3059         if (err)
3060                 tp->syn_fastopen = 0;
3061         kfree_skb(syn_data);
3062 done:
3063         fo->cookie.len = -1;  /* Exclude Fast Open option for SYN retries */
3064         return err;
3065 }
3066
3067 /* Build a SYN and send it off. */
3068 int tcp_connect(struct sock *sk)
3069 {
3070         struct tcp_sock *tp = tcp_sk(sk);
3071         struct sk_buff *buff;
3072         int err;
3073
3074         tcp_connect_init(sk);
3075
3076         if (unlikely(tp->repair)) {
3077                 tcp_finish_connect(sk, NULL);
3078                 return 0;
3079         }
3080
3081         buff = alloc_skb_fclone(MAX_TCP_HEADER + 15, sk->sk_allocation);
3082         if (unlikely(buff == NULL))
3083                 return -ENOBUFS;
3084
3085         /* Reserve space for headers. */
3086         skb_reserve(buff, MAX_TCP_HEADER);
3087
3088         tcp_init_nondata_skb(buff, tp->write_seq++, TCPHDR_SYN);
3089         tp->retrans_stamp = TCP_SKB_CB(buff)->when = tcp_time_stamp;
3090         tcp_connect_queue_skb(sk, buff);
3091         TCP_ECN_send_syn(sk, buff);
3092
3093         /* Send off SYN; include data in Fast Open. */
3094         err = tp->fastopen_req ? tcp_send_syn_data(sk, buff) :
3095               tcp_transmit_skb(sk, buff, 1, sk->sk_allocation);
3096         if (err == -ECONNREFUSED)
3097                 return err;
3098
3099         /* We change tp->snd_nxt after the tcp_transmit_skb() call
3100          * in order to make this packet get counted in tcpOutSegs.
3101          */
3102         tp->snd_nxt = tp->write_seq;
3103         tp->pushed_seq = tp->write_seq;
3104         TCP_INC_STATS(sock_net(sk), TCP_MIB_ACTIVEOPENS);
3105
3106         /* Timer for repeating the SYN until an answer. */
3107         inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
3108                                   inet_csk(sk)->icsk_rto, TCP_RTO_MAX);
3109         return 0;
3110 }
3111 EXPORT_SYMBOL(tcp_connect);
3112
3113 /* Send out a delayed ack, the caller does the policy checking
3114  * to see if we should even be here.  See tcp_input.c:tcp_ack_snd_check()
3115  * for details.
3116  */
3117 void tcp_send_delayed_ack(struct sock *sk)
3118 {
3119         struct inet_connection_sock *icsk = inet_csk(sk);
3120         int ato = icsk->icsk_ack.ato;
3121         unsigned long timeout;
3122
3123         if (ato > TCP_DELACK_MIN) {
3124                 const struct tcp_sock *tp = tcp_sk(sk);
3125                 int max_ato = HZ / 2;
3126
3127                 if (icsk->icsk_ack.pingpong ||
3128                     (icsk->icsk_ack.pending & ICSK_ACK_PUSHED))
3129                         max_ato = TCP_DELACK_MAX;
3130
```

```
3131                /* Slow path, intersegment interval is "high". */
3132
3133                /* If some rtt estimate is known, use it to bound delayed ack.
3134                 * Do not use inet_csk(sk)->icsk_rto here, use results of rtt measurements
3135                 * directly.
3136                 */
3137                if (tp->srtt_us) {
3138                        int rtt = max_t(int, usecs_to_jiffies(tp->srtt_us >> 3),
3139                                        TCP_DELACK_MIN);
3140
3141                        if (rtt < max_ato)
3142                                max_ato = rtt;
3143                }
3144
3145                ato = min(ato, max_ato);
3146        }
3147
3148        /* Stay within the limit we were given */
3149        timeout = jiffies + ato;
3150
3151        /* Use new timeout only if there wasn't a older one earlier. */
3152        if (icsk->icsk_ack.pending & ICSK_ACK_TIMER) {
3153                /* If delack timer was blocked or is about to expire,
3154                 * send ACK now.
3155                 */
3156                if (icsk->icsk_ack.blocked ||
3157                    time_before_eq(icsk->icsk_ack.timeout, jiffies + (ato >> 2))) {
3158                        tcp_send_ack(sk);
3159                        return;
3160                }
3161
3162                if (!time_before(timeout, icsk->icsk_ack.timeout))
3163                        timeout = icsk->icsk_ack.timeout;
3164        }
3165        icsk->icsk_ack.pending |= ICSK_ACK_SCHED | ICSK_ACK_TIMER;
3166        icsk->icsk_ack.timeout = timeout;
3167        sk_reset_timer(sk, &icsk->icsk_delack_timer, timeout);
3168 }
3169
3170 /* This routine sends an ack and also updates the window. */
3171 void tcp_send_ack(struct sock *sk)
3172 {
3173        struct sk_buff *buff;
3174
3175        /* If we have been reset, we may not send again. */
3176        if (sk->sk_state == TCP_CLOSE)
3177                return;
3178
3179        /* We are not putting this on the write queue, so
3180         * tcp_transmit_skb() will set the ownership to this
3181         * sock.
3182         */
3183        buff = alloc_skb(MAX_TCP_HEADER, sk_gfp_atomic(sk, GFP_ATOMIC));
3184        if (buff == NULL) {
3185                inet_csk_schedule_ack(sk);
3186                inet_csk(sk)->icsk_ack.ato = TCP_ATO_MIN;
3187                inet_csk_reset_xmit_timer(sk, ICSK_TIME_DACK,
3188                                          TCP_DELACK_MAX, TCP_RTO_MAX);
3189                return;
3190        }
3191
3192        /* Reserve space for headers and prepare control bits. */
3193        skb_reserve(buff, MAX_TCP_HEADER);
3194        tcp_init_nondata_skb(buff, tcp_acceptable_seq(sk), TCPHDR_ACK);
3195
3196        /* Send it off, this clears delayed acks for us. */
3197        TCP_SKB_CB(buff)->when = tcp_time_stamp;
3198        tcp_transmit_skb(sk, buff, 0, sk_gfp_atomic(sk, GFP_ATOMIC));
3199 }
3200
3201 /* This routine sends a packet with an out of date sequence
3202  * number. It assumes the other end will try to ack it.
3203  *
3204  * Question: what should we make while urgent mode?
3205  * 4.4BSD forces sending single byte of data. We cannot send
3206  * out of window data, because we have SND.NXT==SND.MAX...
3207  *
3208  * Current solution: to send TWO zero-length segments in urgent mode:
3209  * one is with SEG.SEQ=SND.UNA to deliver urgent pointer, another is
3210  * out-of-date with SND.UNA-1 to probe window.
3211  */
3212 static int tcp_xmit_probe_skb(struct sock *sk, int urgent)
3213 {
3214        struct tcp_sock *tp = tcp_sk(sk);
3215        struct sk_buff *skb;
3216
3217        /* We don't queue it, tcp_transmit_skb() sets ownership. */
3218        skb = alloc_skb(MAX_TCP_HEADER, sk_gfp_atomic(sk, GFP_ATOMIC));
3219        if (skb == NULL)
3220                return -1;
```

```
3221
3222         /* Reserve space for headers and set control bits. */
3223         skb_reserve(skb, MAX_TCP_HEADER);
3224         /* Use a previous sequence.  This should cause the other
3225          * end to send an ack.  Don't queue or clone SKB, just
3226          * send it.
3227          */
3228         tcp_init_nondata_skb(skb, tp->snd_una - !urgent, TCPHDR_ACK);
3229         TCP_SKB_CB(skb)->when = tcp_time_stamp;
3230         return tcp_transmit_skb(sk, skb, 0, GFP_ATOMIC);
3231 }
3232
3233 void tcp_send_window_probe(struct sock *sk)
3234 {
3235         if (sk->sk_state == TCP_ESTABLISHED) {
3236                 tcp_sk(sk)->snd_wl1 = tcp_sk(sk)->rcv_nxt - 1;
3237                 tcp_xmit_probe_skb(sk, 0);
3238         }
3239 }
3240
3241 /* Initiate keepalive or window probe from timer. */
3242 int tcp_write_wakeup(struct sock *sk)
3243 {
3244         struct tcp_sock *tp = tcp_sk(sk);
3245         struct sk_buff *skb;
3246
3247         if (sk->sk_state == TCP_CLOSE)
3248                 return -1;
3249
3250         if ((skb = tcp_send_head(sk)) != NULL &&
3251             before(TCP_SKB_CB(skb)->seq, tcp_wnd_end(tp))) {
3252                 int err;
3253                 unsigned int mss = tcp_current_mss(sk);
3254                 unsigned int seg_size = tcp_wnd_end(tp) - TCP_SKB_CB(skb)->seq;
3255
3256                 if (before(tp->pushed_seq, TCP_SKB_CB(skb)->end_seq))
3257                         tp->pushed_seq = TCP_SKB_CB(skb)->end_seq;
3258
3259                 /* We are probing the opening of a window
3260                  * but the window size is != 0
3261                  * must have been a result SWS avoidance ( sender )
3262                  */
3263                 if (seg_size < TCP_SKB_CB(skb)->end_seq - TCP_SKB_CB(skb)->seq ||
3264                     skb->len > mss) {
3265                         seg_size = min(seg_size, mss);
3266                         TCP_SKB_CB(skb)->tcp_flags |= TCPHDR_PSH;
3267                         if (tcp_fragment(sk, skb, seg_size, mss, GFP_ATOMIC))
3268                                 return -1;
3269                 } else if (!tcp_skb_pcount(skb))
3270                         tcp_set_skb_tso_segs(sk, skb, mss);
3271
3272                 TCP_SKB_CB(skb)->tcp_flags |= TCPHDR_PSH;
3273                 TCP_SKB_CB(skb)->when = tcp_time_stamp;
3274                 err = tcp_transmit_skb(sk, skb, 1, GFP_ATOMIC);
3275                 if (!err)
3276                         tcp_event_new_data_sent(sk, skb);
3277                 return err;
3278         } else {
3279                 if (between(tp->snd_up, tp->snd_una + 1, tp->snd_una + 0xFFFF))
3280                         tcp_xmit_probe_skb(sk, 1);
3281                 return tcp_xmit_probe_skb(sk, 0);
3282         }
3283 }
3284
3285 /* A window probe timeout has occurred.  If window is not closed send
3286  * a partial packet else a zero probe.
3287  */
3288 void tcp_send_probe0(struct sock *sk)
3289 {
3290         struct inet_connection_sock *icsk = inet_csk(sk);
3291         struct tcp_sock *tp = tcp_sk(sk);
3292         int err;
3293
3294         err = tcp_write_wakeup(sk);
3295
3296         if (tp->packets_out || !tcp_send_head(sk)) {
3297                 /* Cancel probe timer, if it is not required. */
3298                 icsk->icsk_probes_out = 0;
3299                 icsk->icsk_backoff = 0;
3300                 return;
3301         }
3302
3303         if (err <= 0) {
3304                 if (icsk->icsk_backoff < sysctl_tcp_retries2)
3305                         icsk->icsk_backoff++;
3306                 icsk->icsk_probes_out++;
3307                 inet_csk_reset_xmit_timer(sk, ICSK_TIME_PROBE0,
3308                                           min(icsk->icsk_rto << icsk->icsk_backoff, TCP_RTO_MAX),
3309                                           TCP_RTO_MAX);
3310         } else {
```

```
3311                /* If packet was not sent due to local congestion,
3312                 * do not backoff and do not remember icsk_probes_out.
3313                 * Let local senders to fight for local resources.
3314                 *
3315                 * Use accumulated backoff yet.
3316                 */
3317                if (!icsk->icsk_probes_out)
3318                        icsk->icsk_probes_out = 1;
3319                inet_csk_reset_xmit_timer(sk, ICSK_TIME_PROBE0,
3320                                          min(icsk->icsk_rto << icsk->icsk_backoff,
3321                                              TCP_RESOURCE_PROBE_INTERVAL),
3322                                          TCP_RTO_MAX);
3323        }
3324 }
3325
3326 int tcp_rtx_synack(struct sock *sk, struct request_sock *req)
3327 {
3328        const struct tcp_request_sock_ops *af_ops = tcp_rsk(req)->af_specific;
3329        struct flowi fl;
3330        int res;
3331
3332        res = af_ops->send_synack(sk, NULL, &fl, req, 0, NULL);
3333        if (!res) {
3334                TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_RETRANSSEGS);
3335                NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPSYNRETRANS);
3336        }
3337        return res;
3338 }
3339 EXPORT_SYMBOL(tcp_rtx_synack);
3340
```

This page was automatically generated by LXR 0.3.1 (source). • Linux is a registered trademark of Linus Torvalds • Contact us

- Home
- Development
- Services
- Training
- Docs
- Community
- Company
- Blog