Linux Cross Reference

Free Electrons

Embedded Linux Experts

• *source navigation* • diff markup • identifier search • freetext search •

Version: 2.0.40 2.2.26 2.4.37 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 3.10 3.11 3.12 3.13 3.14 3.15 3.16 *3.17*

Linux/net/ipv4/tcp_output.c

```
/*
 * INET         An implementation of the TCP/IP protocol suite for the LINUX
 *              operating system.  INET is implemented using the  BSD Socket
 *              interface as the means of communication with the user level.
 *
 *              Implementation of the Transmission Control Protocol(TCP).
 *
 * Authors:     Ross Biro
 *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *              Mark Evans, <evansmp@uhura.aston.ac.uk>
 *              Corey Minyard <wf-rch!minyard@relay.EU.net>
 *              Florian La Roche, <flla@stud.uni-sb.de>
 *              Charles Hedrick, <hedrick@klinzhai.rutgers.edu>
 *              Linus Torvalds, <torvalds@cs.helsinki.fi>
 *              Alan Cox, <gw4pts@gw4pts.ampr.org>
 *              Matthew Dillon, <dillon@apollo.west.oic.com>
 *              Arnt Gulbrandsen, <agulbra@nvg.unit.no>
 *              Jorge Cwik, <jorge@laser.satlink.net>
 */

/*
 * Changes:     Pedro Roque     :       Retransmit queue handled by TCP.
 *                              :       Fragmentation on mtu decrease
 *                              :       Segment collapse on retransmit
 *                              :       AF independence
 *
 *              Linus Torvalds  :       send_delayed_ack
 *              David S. Miller :       Charge memory using the right skb
 *                                      during syn/ack processing.
 *              David S. Miller :       Output engine completely rewritten.
 *              Andrea Arcangeli:       SYNACK carry ts_recent in tsecr.
 *              Cacophonix Gaul :       draft-minshall-nagle-01
 *              J Hadi Salim    :       ECN support
 *
 */

#define pr_fmt(fmt) "TCP: " fmt

#include <net/tcp.h>

#include <linux/compiler.h>
#include <linux/gfp.h>
#include <linux/module.h>

/* People can turn this off for buggy TCP's found in printers etc. */
int sysctl_tcp_retrans_collapse __read_mostly = 1;

/* People can turn this on to work with those rare, broken TCPs that
 * interpret the window field as a signed quantity.
 */
int sysctl_tcp_workaround_signed_windows __read_mostly = 0;

/* Default TSQ limit of two TSO segments */
int sysctl_tcp_limit_output_bytes __read_mostly = 131072;

/* This limits the percentage of the congestion window which we
 * will allow a single TSO frame to consume.  Building TSO frames
 * which are too large can cause TCP streams to be bursty.
 */
int sysctl_tcp_tso_win_divisor __read_mostly = 3;

int sysctl_tcp_mtu_probing __read_mostly = 0;
int sysctl_tcp_base_mss __read_mostly = TCP_BASE_MSS;

/* By default, RFC2861 behavior.  */
int sysctl_tcp_slow_start_after_idle __read_mostly = 1;

unsigned int sysctl_tcp_notsent_lowat __read_mostly = UINT_MAX;
EXPORT_SYMBOL(sysctl_tcp_notsent_lowat);

```

```
static bool tcp_write_xmit(struct sock *sk, unsigned int mss_now, int nonagle,
                           int push_one, gfp_t gfp);

/* Account for new data that has been sent to the network. */
static void tcp_event_new_data_sent(struct sock *sk, const struct sk_buff *skb)
{
        struct inet_connection_sock *icsk = inet_csk(sk);
        struct tcp_sock *tp = tcp_sk(sk);
        unsigned int prior_packets = tp->packets_out;

        tcp_advance_send_head(sk, skb);
        tp->snd_nxt = TCP_SKB_CB(skb)->end_seq;

        tp->packets_out += tcp_skb_pcount(skb);
        if (!prior_packets || icsk->icsk_pending == ICSK_TIME_EARLY_RETRANS ||
            icsk->icsk_pending == ICSK_TIME_LOSS_PROBE) {
                tcp_rearm_rto(sk);
        }

        NET_ADD_STATS(sock_net(sk), LINUX_MIB_TCPORIGDATASENT,
                      tcp_skb_pcount(skb));
}

/* SND.NXT, if window was not shrunk.
 * If window has been shrunk, what should we make? It is not clear at all.
 * Using SND.UNA we will fail to open window, SND.NXT is out of window. :-(
 * Anything in between SND.UNA...SND.UNA+SND.WND also can be already
 * invalid. OK, let's make this for now:
 */
static inline __u32 tcp_acceptable_seq(const struct sock *sk)
{
        const struct tcp_sock *tp = tcp_sk(sk);

        if (!before(tcp_wnd_end(tp), tp->snd_nxt))
                return tp->snd_nxt;
        else
                return tcp_wnd_end(tp);
}

/* Calculate mss to advertise in SYN segment.
 * RFC1122, RFC1063, draft-ietf-tcpimpl-pmtud-01 state that:
 *
 * 1. It is independent of path mtu.
 * 2. Ideally, it is maximal possible segment size i.e. 65535-40.
 * 3. For IPv4 it is reasonable to calculate it from maximal MTU of
 *    attached devices, because some buggy hosts are confused by
 *    large MSS.
 * 4. We do not make 3, we advertise MSS, calculated from first
 *    hop device mtu, but allow to raise it to ip_rt_min_advmss.
 *    This may be overridden via information stored in routing table.
 * 5. Value 65535 for MSS is valid in IPv6 and means "as large as possible,
 *    probably even Jumbo".
 */
static __u16 tcp_advertise_mss(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);
        const struct dst_entry *dst = __sk_dst_get(sk);
        int mss = tp->advmss;

        if (dst) {
                unsigned int metric = dst_metric_advmss(dst);

                if (metric < mss) {
                        mss = metric;
                        tp->advmss = mss;
                }
        }

        return (__u16)mss;
}

/* RFC2861. Reset CWND after idle period longer RTO to "restart window".
 * This is the first part of cwnd validation mechanism. */
static void tcp_cwnd_restart(struct sock *sk, const struct dst_entry *dst)
{
        struct tcp_sock *tp = tcp_sk(sk);
        s32 delta = tcp_time_stamp - tp->lsndtime;
        u32 restart_cwnd = tcp_init_cwnd(tp, dst);
        u32 cwnd = tp->snd_cwnd;

        tcp_ca_event(sk, CA_EVENT_CWND_RESTART);

        tp->snd_ssthresh = tcp_current_ssthresh(sk);
        restart_cwnd = min(restart_cwnd, cwnd);

        while ((delta -= inet_csk(sk)->icsk_rto) > 0 && cwnd > restart_cwnd)
                cwnd >>= 1;
        tp->snd_cwnd = max(cwnd, restart_cwnd);
        tp->snd_cwnd_stamp = tcp_time_stamp;
        tp->snd_cwnd_used = 0;
```

```
}

/* Congestion state accounting after a packet has been sent. */
static void tcp_event_data_sent(struct tcp_sock *tp,
                                struct sock *sk)
{
        struct inet_connection_sock *icsk = inet_csk(sk);
        const u32 now = tcp_time_stamp;
        const struct dst_entry *dst = __sk_dst_get(sk);

        if (sysctl_tcp_slow_start_after_idle &&
            (!tp->packets_out && (s32)(now - tp->lsndtime) > icsk->icsk_rto))
                tcp_cwnd_restart(sk, __sk_dst_get(sk));

        tp->lsndtime = now;

        /* If it is a reply for ato after last received
         * packet, enter pingpong mode.
         */
        if ((u32)(now - icsk->icsk_ack.lrcvtime) < icsk->icsk_ack.ato &&
            (!dst || !dst_metric(dst, RTAX_QUICKACK)))
                        icsk->icsk_ack.pingpong = 1;
}

/* Account for an ACK we sent. */
static inline void tcp_event_ack_sent(struct sock *sk, unsigned int pkts)
{
        tcp_dec_quickack_mode(sk, pkts);
        inet_csk_clear_xmit_timer(sk, ICSK_TIME_DACK);
}


u32 tcp_default_init_rwnd(u32 mss)
{
        /* Initial receive window should be twice of TCP_INIT_CWND to
         * enable proper sending of new unsent data during fast recovery
         * (RFC 3517, Section 4, NextSeg() rule (2)). Further place a
         * limit when mss is larger than 1460.
         */
        u32 init_rwnd = TCP_INIT_CWND * 2;

        if (mss > 1460)
                init_rwnd = max((1460 * init_rwnd) / mss, 2U);
        return init_rwnd;
}

/* Determine a window scaling and initial window to offer.
 * Based on the assumption that the given amount of space
 * will be offered. Store the results in the tp structure.
 * NOTE: for smooth operation initial space offering should
 * be a multiple of mss if possible. We assume here that mss >= 1.
 * This MUST be enforced by all callers.
 */
void tcp_select_initial_window(int __space, __u32 mss,
                               __u32 *rcv_wnd, __u32 *window_clamp,
                               int wscale_ok, __u8 *rcv_wscale,
                               __u32 init_rcv_wnd)
{
        unsigned int space = (__space < 0 ? 0 : __space);

        /* If no clamp set the clamp to the max possible scaled window */
        if (*window_clamp == 0)
                (*window_clamp) = (65535 << 14);
        space = min(*window_clamp, space);

        /* Quantize space offering to a multiple of mss if possible. */
        if (space > mss)
                space = (space / mss) * mss;

        /* NOTE: offering an initial window larger than 32767
         * will break some buggy TCP stacks. If the admin tells us
         * it is likely we could be speaking with such a buggy stack
         * we will truncate our initial window offering to 32K-1
         * unless the remote has sent us a window scaling option,
         * which we interpret as a sign the remote TCP is not
         * misinterpreting the window field as a signed quantity.
         */
        if (sysctl_tcp_workaround_signed_windows)
                (*rcv_wnd) = min(space, MAX_TCP_WINDOW);
        else
                (*rcv_wnd) = space;

        (*rcv_wscale) = 0;
        if (wscale_ok) {
                /* Set window scaling on max possible window
                 * See RFC1323 for an explanation of the limit to 14
                 */
                space = max_t(u32, sysctl_tcp_rmem[2], sysctl_rmem_max);
                space = min_t(u32, space, *window_clamp);
                while (space > 65535 && (*rcv_wscale) < 14) {
                        space >>= 1;
                        (*rcv_wscale)++;
                }
        }

        if (mss > (1 << *rcv_wscale)) {
                if (!init_rcv_wnd) /* Use default unless specified otherwise */
                        init_rcv_wnd = tcp_default_init_rwnd(mss);
                *rcv_wnd = min(*rcv_wnd, init_rcv_wnd * mss);
        }

        /* Set the clamp no higher than max representable value */
        (*window_clamp) = min(65535U << (*rcv_wscale), *window_clamp);
}
EXPORT_SYMBOL(tcp_select_initial_window);

/* Chose a new window to advertise, update state in tcp_sock for the
 * socket, and return result with RFC1323 scaling applied.  The return
 * value can be stuffed directly into th->window for an outgoing
 * frame.
 */
static u16 tcp_select_window(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);
        u32 old_win = tp->rcv_wnd;
        u32 cur_win = tcp_receive_window(tp);
        u32 new_win = __tcp_select_window(sk);

        /* Never shrink the offered window */
        if (new_win < cur_win) {
                /* Danger Will Robinson!
                 * Don't update rcv_wup/rcv_wnd here or else
                 * we will not be able to advertise a zero
                 * window in time.  --DaveM
                 *
                 * Relax Will Robinson.
                 */
                if (new_win == 0)
                        NET_INC_STATS(sock_net(sk),
                                      LINUX_MIB_TCPWANTZEROWINDOWADV);
                new_win = ALIGN(cur_win, 1 << tp->rx_opt.rcv_wscale);
        }
        tp->rcv_wnd = new_win;
        tp->rcv_wup = tp->rcv_nxt;

        /* Make sure we do not exceed the maximum possible
         * scaled window.
         */
        if (!tp->rx_opt.rcv_wscale && sysctl_tcp_workaround_signed_windows)
                new_win = min(new_win, MAX_TCP_WINDOW);
        else
                new_win = min(new_win, (65535U << tp->rx_opt.rcv_wscale));

        /* RFC1323 scaling applied */
        new_win >>= tp->rx_opt.rcv_wscale;

        /* If we advertise zero window, disable fast path. */
        if (new_win == 0) {
                tp->pred_flags = 0;
                if (old_win)
                        NET_INC_STATS(sock_net(sk),
                                      LINUX_MIB_TCPTOZEROWINDOWADV);
        } else if (old_win == 0) {
                NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPFROMZEROWINDOWADV);
        }

        return new_win;
}

/* Packet ECN state for a SYN-ACK */
static inline void TCP_ECN_send_synack(const struct tcp_sock *tp, struct sk_buff *skb)
{
        TCP_SKB_CB(skb)->tcp_flags &= ~TCPHDR_CWR;
        if (!(tp->ecn_flags & TCP_ECN_OK))
                TCP_SKB_CB(skb)->tcp_flags &= ~TCPHDR_ECE;
}

/* Packet ECN state for a SYN.  */
static inline void TCP_ECN_send_syn(struct sock *sk, struct sk_buff *skb)
{
        struct tcp_sock *tp = tcp_sk(sk);

        tp->ecn_flags = 0;
        if (sock_net(sk)->ipv4.sysctl_tcp_ecn == 1) {
                TCP_SKB_CB(skb)->tcp_flags |= TCPHDR_ECE | TCPHDR_CWR;
                tp->ecn_flags = TCP_ECN_OK;
        }
}

static __inline__ void
```

```
TCP_ECN_make_synack(const struct request_sock *req, struct tcphdr *th)
{
        if (inet_rsk(req)->ecn_ok)
                th->ece = 1;
}

/* Set up ECN state for a packet on a ESTABLISHED socket that is about to
 * be sent.
 */
static inline void TCP_ECN_send(struct sock *sk, struct sk_buff *skb,
                                int tcp_header_len)
{
        struct tcp_sock *tp = tcp_sk(sk);

        if (tp->ecn_flags & TCP_ECN_OK) {
                /* Not-retransmitted data segment: set ECT and inject CWR. */
                if (skb->len != tcp_header_len &&
                    !before(TCP_SKB_CB(skb)->seq, tp->snd_nxt)) {
                        INET_ECN_xmit(sk);
                        if (tp->ecn_flags & TCP_ECN_QUEUE_CWR) {
                                tp->ecn_flags &= ~TCP_ECN_QUEUE_CWR;
                                tcp_hdr(skb)->cwr = 1;
                                skb_shinfo(skb)->gso_type |= SKB_GSO_TCP_ECN;
                        }
                } else {
                        /* ACK or retransmitted segment: clear ECT|CE */
                        INET_ECN_dontxmit(sk);
                }
                if (tp->ecn_flags & TCP_ECN_DEMAND_CWR)
                        tcp_hdr(skb)->ece = 1;
        }
}

/* Constructs common control bits of non-data skb. If SYN/FIN is present,
 * auto increment end seqno.
 */
static void tcp_init_nondata_skb(struct sk_buff *skb, u32 seq, u8 flags)
{
        struct skb_shared_info *shinfo = skb_shinfo(skb);

        skb->ip_summed = CHECKSUM_PARTIAL;
        skb->csum = 0;

        TCP_SKB_CB(skb)->tcp_flags = flags;
        TCP_SKB_CB(skb)->sacked = 0;

        shinfo->gso_segs = 1;
        shinfo->gso_size = 0;
        shinfo->gso_type = 0;

        TCP_SKB_CB(skb)->seq = seq;
        if (flags & (TCPHDR_SYN | TCPHDR_FIN))
                seq++;
        TCP_SKB_CB(skb)->end_seq = seq;
}

static inline bool tcp_urg_mode(const struct tcp_sock *tp)
{
        return tp->snd_una != tp->snd_up;
}

#define OPTION_SACK_ADVERTISE   (1 << 0)
#define OPTION_TS               (1 << 1)
#define OPTION_MD5              (1 << 2)
#define OPTION_WSCALE           (1 << 3)
#define OPTION_FAST_OPEN_COOKIE (1 << 8)

struct tcp_out_options {
        u16 options;            /* bit field of OPTION_* */
        u16 mss;                /* 0 to disable */
        u8 ws;                  /* window scale, 0 to disable */
        u8 num_sack_blocks;     /* number of SACK blocks to include */
        u8 hash_size;           /* bytes in hash_location */
        __u8 *hash_location;    /* temporary pointer, overloaded */
        __u32 tsval, tsecr;     /* need to include OPTION_TS */
        struct tcp_fastopen_cookie *fastopen_cookie;    /* Fast open cookie */
};

/* Write previously computed TCP options to the packet.
 *
 * Beware: Something in the Internet is very sensitive to the ordering of
 * TCP options, we learned this through the hard way, so be careful here.
 * Luckily we can at least blame others for their non-compliance but from
 * inter-operability perspective it seems that we're somewhat stuck with
 * the ordering which we have been using if we want to keep working with
 * those broken things (not that it currently hurts anybody as there isn't
 * particular reason why the ordering would need to be changed).
 *
 * At least SACK_PERM as the first option is known to lead to a disaster
 * (but it may well be that other scenarios fail similarly).
 */
static void tcp_options_write(__be32 *ptr, struct tcp_sock *tp,
                              struct tcp_out_options *opts)
{
        u16 options = opts->options;    /* mungable copy */

        if (unlikely(OPTION_MD5 & options)) {
                *ptr++ = htonl((TCPOPT_NOP << 24) | (TCPOPT_NOP << 16) |
                               (TCPOPT_MD5SIG << 8) | TCPOLEN_MD5SIG);
                /* overload cookie hash location */
                opts->hash_location = (__u8 *)ptr;
                ptr += 4;
        }

        if (unlikely(opts->mss)) {
                *ptr++ = htonl((TCPOPT_MSS << 24) |
                               (TCPOLEN_MSS << 16) |
                               opts->mss);
        }

        if (likely(OPTION_TS & options)) {
                if (unlikely(OPTION_SACK_ADVERTISE & options)) {
                        *ptr++ = htonl((TCPOPT_SACK_PERM << 24) |
                                       (TCPOLEN_SACK_PERM << 16) |
                                       (TCPOPT_TIMESTAMP << 8) |
                                       TCPOLEN_TIMESTAMP);
                        options &= ~OPTION_SACK_ADVERTISE;
                } else {
                        *ptr++ = htonl((TCPOPT_NOP << 24) |
                                       (TCPOPT_NOP << 16) |
                                       (TCPOPT_TIMESTAMP << 8) |
                                       TCPOLEN_TIMESTAMP);
                }
                *ptr++ = htonl(opts->tsval);
                *ptr++ = htonl(opts->tsecr);
        }

        if (unlikely(OPTION_SACK_ADVERTISE & options)) {
                *ptr++ = htonl((TCPOPT_NOP << 24) |
                               (TCPOPT_NOP << 16) |
                               (TCPOPT_SACK_PERM << 8) |
                               TCPOLEN_SACK_PERM);
        }

        if (unlikely(OPTION_WSCALE & options)) {
                *ptr++ = htonl((TCPOPT_NOP << 24) |
                               (TCPOPT_WINDOW << 16) |
                               (TCPOLEN_WINDOW << 8) |
                               opts->ws);
        }

        if (unlikely(opts->num_sack_blocks)) {
                struct tcp_sack_block *sp = tp->rx_opt.dsack ?
                        tp->duplicate_sack : tp->selective_acks;
                int this_sack;

                *ptr++ = htonl((TCPOPT_NOP  << 24) |
                               (TCPOPT_NOP  << 16) |
                               (TCPOPT_SACK <<  8) |
                               (TCPOLEN_SACK_BASE + (opts->num_sack_blocks *
                                                     TCPOLEN_SACK_PERBLOCK)));

                for (this_sack = 0; this_sack < opts->num_sack_blocks;
                     ++this_sack) {
                        *ptr++ = htonl(sp[this_sack].start_seq);
                        *ptr++ = htonl(sp[this_sack].end_seq);
                }

                tp->rx_opt.dsack = 0;
        }

        if (unlikely(OPTION_FAST_OPEN_COOKIE & options)) {
                struct tcp_fastopen_cookie *foc = opts->fastopen_cookie;

                *ptr++ = htonl((TCPOPT_EXP << 24) |
                               ((TCPOLEN_EXP_FASTOPEN_BASE + foc->len) << 16) |
                               TCPOPT_FASTOPEN_MAGIC);

                memcpy(ptr, foc->val, foc->len);
                if ((foc->len & 3) == 2) {
                        u8 *align = ((u8 *)ptr) + foc->len;
                        align[0] = align[1] = TCPOPT_NOP;
                }
                ptr += (foc->len + 3) >> 2;
        }
}

/* Compute TCP options for SYN packets. This is not the final
 * network wire format yet.
 */
```

```
static unsigned int tcp_syn_options(struct sock *sk, struct sk_buff *skb,
                                    struct tcp_out_options *opts,
                                    struct tcp_md5sig_key **md5)
{
        struct tcp_sock *tp = tcp_sk(sk);
        unsigned int remaining = MAX_TCP_OPTION_SPACE;
        struct tcp_fastopen_request *fastopen = tp->fastopen_req;

#ifdef CONFIG_TCP_MD5SIG
        *md5 = tp->af_specific->md5_lookup(sk, sk);
        if (*md5) {
                opts->options |= OPTION_MD5;
                remaining -= TCPOLEN_MD5SIG_ALIGNED;
        }
#else
        *md5 = NULL;
#endif

        /* We always get an MSS option.  The option bytes which will be seen in
         * normal data packets should timestamps be used, must be in the MSS
         * advertised.  But we subtract them from tp->mss_cache so that
         * calculations in tcp_sendmsg are simpler etc.  So account for this
         * fact here if necessary.  If we don't do this correctly, as a
         * receiver we won't recognize data packets as being full sized when we
         * should, and thus we won't abide by the delayed ACK rules correctly.
         * SACKs don't matter, we never delay an ACK when we have any of those
         * going out.  */
        opts->mss = tcp_advertise_mss(sk);
        remaining -= TCPOLEN_MSS_ALIGNED;

        if (likely(sysctl_tcp_timestamps && *md5 == NULL)) {
                opts->options |= OPTION_TS;
                opts->tsval = TCP_SKB_CB(skb)->when + tp->tsoffset;
                opts->tsecr = tp->rx_opt.ts_recent;
                remaining -= TCPOLEN_TSTAMP_ALIGNED;
        }
        if (likely(sysctl_tcp_window_scaling)) {
                opts->ws = tp->rx_opt.rcv_wscale;
                opts->options |= OPTION_WSCALE;
                remaining -= TCPOLEN_WSCALE_ALIGNED;
        }
        if (likely(sysctl_tcp_sack)) {
                opts->options |= OPTION_SACK_ADVERTISE;
                if (unlikely(!(OPTION_TS & opts->options)))
                        remaining -= TCPOLEN_SACKPERM_ALIGNED;
        }

        if (fastopen && fastopen->cookie.len >= 0) {
                u32 need = TCPOLEN_EXP_FASTOPEN_BASE + fastopen->cookie.len;
                need = (need + 3) & ~3U;  /* Align to 32 bits */
                if (remaining >= need) {
                        opts->options |= OPTION_FAST_OPEN_COOKIE;
                        opts->fastopen_cookie = &fastopen->cookie;
                        remaining -= need;
                        tp->syn_fastopen = 1;
                }
        }

        return MAX_TCP_OPTION_SPACE - remaining;
}

/* Set up TCP options for SYN-ACKs. */
static unsigned int tcp_synack_options(struct sock *sk,
                                       struct request_sock *req,
                                       unsigned int mss, struct sk_buff *skb,
                                       struct tcp_out_options *opts,
                                       struct tcp_md5sig_key **md5,
                                       struct tcp_fastopen_cookie *foc)
{
        struct inet_request_sock *ireq = inet_rsk(req);
        unsigned int remaining = MAX_TCP_OPTION_SPACE;

#ifdef CONFIG_TCP_MD5SIG
        *md5 = tcp_rsk(req)->af_specific->md5_lookup(sk, req);
        if (*md5) {
                opts->options |= OPTION_MD5;
                remaining -= TCPOLEN_MD5SIG_ALIGNED;

                /* We can't fit any SACK blocks in a packet with MD5 + TS
                 * options. There was discussion about disabling SACK
                 * rather than TS in order to fit in better with old,
                 * buggy kernels, but that was deemed to be unnecessary.
                 */
                ireq->tstamp_ok &= !ireq->sack_ok;
        }
#else
        *md5 = NULL;
#endif

        /* We always send an MSS option. */
        opts->mss = mss;
        remaining -= TCPOLEN_MSS_ALIGNED;

        if (likely(ireq->wscale_ok)) {
                opts->ws = ireq->rcv_wscale;
                opts->options |= OPTION_WSCALE;
                remaining -= TCPOLEN_WSCALE_ALIGNED;
        }
        if (likely(ireq->tstamp_ok)) {
                opts->options |= OPTION_TS;
                opts->tsval = TCP_SKB_CB(skb)->when;
                opts->tsecr = req->ts_recent;
                remaining -= TCPOLEN_TSTAMP_ALIGNED;
        }
        if (likely(ireq->sack_ok)) {
                opts->options |= OPTION_SACK_ADVERTISE;
                if (unlikely(!ireq->tstamp_ok))
                        remaining -= TCPOLEN_SACKPERM_ALIGNED;
        }
        if (foc != NULL && foc->len >= 0) {
                u32 need = TCPOLEN_EXP_FASTOPEN_BASE + foc->len;
                need = (need + 3) & ~3U;  /* Align to 32 bits */
                if (remaining >= need) {
                        opts->options |= OPTION_FAST_OPEN_COOKIE;
                        opts->fastopen_cookie = foc;
                        remaining -= need;
                }
        }

        return MAX_TCP_OPTION_SPACE - remaining;
}

/* Compute TCP options for ESTABLISHED sockets. This is not the
 * final wire format yet.
 */
static unsigned int tcp_established_options(struct sock *sk, struct sk_buff *skb,
                                        struct tcp_out_options *opts,
                                        struct tcp_md5sig_key **md5)
{
        struct tcp_skb_cb *tcb = skb ? TCP_SKB_CB(skb) : NULL;
        struct tcp_sock *tp = tcp_sk(sk);
        unsigned int size = 0;
        unsigned int eff_sacks;

        opts->options = 0;

#ifdef CONFIG_TCP_MD5SIG
        *md5 = tp->af_specific->md5_lookup(sk, sk);
        if (unlikely(*md5)) {
                opts->options |= OPTION_MD5;
                size += TCPOLEN_MD5SIG_ALIGNED;
        }
#else
        *md5 = NULL;
#endif

        if (likely(tp->rx_opt.tstamp_ok)) {
                opts->options |= OPTION_TS;
                opts->tsval = tcb ? tcb->when + tp->tsoffset : 0;
                opts->tsecr = tp->rx_opt.ts_recent;
                size += TCPOLEN_TSTAMP_ALIGNED;
        }

        eff_sacks = tp->rx_opt.num_sacks + tp->rx_opt.dsack;
        if (unlikely(eff_sacks)) {
                const unsigned int remaining = MAX_TCP_OPTION_SPACE - size;
                opts->num_sack_blocks =
                        min_t(unsigned int, eff_sacks,
                              (remaining - TCPOLEN_SACK_BASE_ALIGNED) /
                              TCPOLEN_SACK_PERBLOCK);
                size += TCPOLEN_SACK_BASE_ALIGNED +
                        opts->num_sack_blocks * TCPOLEN_SACK_PERBLOCK;
        }

        return size;
}


/* TCP SMALL QUEUES (TSQ)
 *
 * TSQ goal is to keep small amount of skbs per tcp flow in tx queues (qdisc+dev)
 * to reduce RTT and bufferbloat.
 * We do this using a special skb destructor (tcp_wfree).
 *
 * Its important tcp_wfree() can be replaced by sock_wfree() in the event skb
 * needs to be reallocated in a driver.
 * The invariant being skb->truesize subtracted from sk->sk_wmem_alloc
 *
 * Since transmit from skb destructor is forbidden, we use a tasklet
 * to process all sockets that eventually need to send more skbs.
 * We use one tasklet per cpu, with its own queue of sockets.
 */
struct tsq_tasklet {
        struct tasklet_struct   tasklet;
        struct list_head        head; /* queue of tcp sockets */
};
static DEFINE_PER_CPU(struct tsq_tasklet, tsq_tasklet);

static void tcp_tsq_handler(struct sock *sk)
{
        if ((1 << sk->sk_state) &
            (TCPF_ESTABLISHED | TCPF_FIN_WAIT1 | TCPF_CLOSING |
             TCPF_CLOSE_WAIT  | TCPF_LAST_ACK))
                tcp_write_xmit(sk, tcp_current_mss(sk), tcp_sk(sk)->nonagle,
                               0, GFP_ATOMIC);
}
/*
 * One tasklet per cpu tries to send more skbs.
 * We run in tasklet context but need to disable irqs when
 * transferring tsq->head because tcp_wfree() might
 * interrupt us (non NAPI drivers)
 */
static void tcp_tasklet_func(unsigned long data)
{
        struct tsq_tasklet *tsq = (struct tsq_tasklet *)data;
        LIST_HEAD(list);
        unsigned long flags;
        struct list_head *q, *n;
        struct tcp_sock *tp;
        struct sock *sk;

        local_irq_save(flags);
        list_splice_init(&tsq->head, &list);
        local_irq_restore(flags);

        list_for_each_safe(q, n, &list) {
                tp = list_entry(q, struct tcp_sock, tsq_node);
                list_del(&tp->tsq_node);

                sk = (struct sock *)tp;
                bh_lock_sock(sk);

                if (!sock_owned_by_user(sk)) {
                        tcp_tsq_handler(sk);
                } else {
                        /* defer the work to tcp_release_cb() */
                        set_bit(TCP_TSQ_DEFERRED, &tp->tsq_flags);
                }
                bh_unlock_sock(sk);

                clear_bit(TSQ_QUEUED, &tp->tsq_flags);
                sk_free(sk);
        }
}

#define TCP_DEFERRED_ALL ((1UL << TCP_TSQ_DEFERRED) |           \
                          (1UL << TCP_WRITE_TIMER_DEFERRED) |   \
                          (1UL << TCP_DELACK_TIMER_DEFERRED) |  \
                          (1UL << TCP_MTU_REDUCED_DEFERRED))
/**
 * tcp_release_cb - tcp release_sock() callback
 * @sk: socket
 *
 * called from release_sock() to perform protocol dependent
 * actions before socket release.
 */
void tcp_release_cb(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);
        unsigned long flags, nflags;

        /* perform an atomic operation only if at least one flag is set */
        do {
                flags = tp->tsq_flags;
                if (!(flags & TCP_DEFERRED_ALL))
                        return;
                nflags = flags & ~TCP_DEFERRED_ALL;
        } while (cmpxchg(&tp->tsq_flags, flags, nflags) != flags);

        if (flags & (1UL << TCP_TSQ_DEFERRED))
                tcp_tsq_handler(sk);

        /* Here begins the tricky part :
         * We are called from release_sock() with :
         * 1) BH disabled
         * 2) sk_lock.slock spinlock held
         * 3) socket owned by us (sk->sk_lock.owned == 1)
         *
         * But following code is meant to be called from BH handlers,
         * so we should keep BH disabled, but early release socket ownership
         */
        sock_release_ownership(sk);

        if (flags & (1UL << TCP_WRITE_TIMER_DEFERRED)) {
                tcp_write_timer_handler(sk);
                __sock_put(sk);
        }
        if (flags & (1UL << TCP_DELACK_TIMER_DEFERRED)) {
                tcp_delack_timer_handler(sk);
                __sock_put(sk);
        }
        if (flags & (1UL << TCP_MTU_REDUCED_DEFERRED)) {
                inet_csk(sk)->icsk_af_ops->mtu_reduced(sk);
                __sock_put(sk);
        }
}
EXPORT_SYMBOL(tcp_release_cb);

void __init tcp_tasklet_init(void)
{
        int i;

        for_each_possible_cpu(i) {
                struct tsq_tasklet *tsq = &per_cpu(tsq_tasklet, i);

                INIT_LIST_HEAD(&tsq->head);
                tasklet_init(&tsq->tasklet,
                             tcp_tasklet_func,
                             (unsigned long)tsq);
        }
}

/*
 * Write buffer destructor automatically called from kfree_skb.
 * We can't xmit new skbs from this context, as we might already
 * hold qdisc lock.
 */
void tcp_wfree(struct sk_buff *skb)
{
        struct sock *sk = skb->sk;
        struct tcp_sock *tp = tcp_sk(sk);

        if (test_and_clear_bit(TSQ_THROTTLED, &tp->tsq_flags) &&
            !test_and_set_bit(TSQ_QUEUED, &tp->tsq_flags)) {
                unsigned long flags;
                struct tsq_tasklet *tsq;

                /* Keep a ref on socket.
                 * This last ref will be released in tcp_tasklet_func()
                 */
                atomic_sub(skb->truesize - 1, &sk->sk_wmem_alloc);

                /* queue this socket to tasklet queue */
                local_irq_save(flags);
                tsq = &__get_cpu_var(tsq_tasklet);
                list_add(&tp->tsq_node, &tsq->head);
                tasklet_schedule(&tsq->tasklet);
                local_irq_restore(flags);
        } else {
                sock_wfree(skb);
        }
}

/* This routine actually transmits TCP packets queued in by
 * tcp_do_sendmsg().  This is used by both the initial
 * transmission and possible later retransmissions.
 * All SKB's seen here are completely headerless.  It is our
 * job to build the TCP header, and pass the packet down to
 * IP so it can do the same plus pass the packet off to the
 * device.
 *
 * We are working here with either a clone of the original
 * SKB, or a fresh unique copy made by the retransmit engine.
 */
static int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb, int clone_it,
                            gfp_t gfp_mask)
{
        const struct inet_connection_sock *icsk = inet_csk(sk);
        struct inet_sock *inet;
        struct tcp_sock *tp;
        struct tcp_skb_cb *tcb;
        struct tcp_out_options opts;
        unsigned int tcp_options_size, tcp_header_size;
        struct tcp_md5sig_key *md5;
        struct tcphdr *th;
        int err;

        BUG_ON(!skb || !tcp_skb_pcount(skb));

        if (clone_it) {
                skb_mstamp_get(&skb->skb_mstamp);

                if (unlikely(skb_cloned(skb)))
                        skb = pskb_copy(skb, gfp_mask);
                else
                        skb = skb_clone(skb, gfp_mask);
                if (unlikely(!skb))
                        return -ENOBUFS;
                /* Our usage of tstamp should remain private */
                skb->tstamp.tv64 = 0;
        }

        inet = inet_sk(sk);
        tp = tcp_sk(sk);
        tcb = TCP_SKB_CB(skb);
        memset(&opts, 0, sizeof(opts));

        if (unlikely(tcb->tcp_flags & TCPHDR_SYN))
                tcp_options_size = tcp_syn_options(sk, skb, &opts, &md5);
        else
                tcp_options_size = tcp_established_options(sk, skb, &opts,
                                                           &md5);
        tcp_header_size = tcp_options_size + sizeof(struct tcphdr);

        if (tcp_packets_in_flight(tp) == 0)
                tcp_ca_event(sk, CA_EVENT_TX_START);

        /* if no packet is in qdisc/device queue, then allow XPS to select
         * another queue.
         */
        skb->ooo_okay = sk_wmem_alloc_get(sk) == 0;

        skb_push(skb, tcp_header_size);
        skb_reset_transport_header(skb);

        skb_orphan(skb);
        skb->sk = sk;
        skb->destructor = tcp_wfree;
        skb_set_hash_from_sk(skb, sk);
        atomic_add(skb->truesize, &sk->sk_wmem_alloc);

        /* Build TCP header and checksum it. */
        th = tcp_hdr(skb);
        th->source              = inet->inet_sport;
        th->dest                = inet->inet_dport;
        th->seq                 = htonl(tcb->seq);
        th->ack_seq             = htonl(tp->rcv_nxt);
        *(((__be16 *)th) + 6)   = htons(((tcp_header_size >> 2) << 12) |
                                        tcb->tcp_flags);

        if (unlikely(tcb->tcp_flags & TCPHDR_SYN)) {
                /* RFC1323: The window in SYN & SYN/ACK segments
                 * is never scaled.
                 */
                th->window      = htons(min(tp->rcv_wnd, 65535U));
        } else {
                th->window      = htons(tcp_select_window(sk));
        }
        th->check               = 0;
        th->urg_ptr             = 0;

        /* The urg_mode check is necessary during a below snd_una win probe */
        if (unlikely(tcp_urg_mode(tp) && before(tcb->seq, tp->snd_up))) {
                if (before(tp->snd_up, tcb->seq + 0x10000)) {
                        th->urg_ptr = htons(tp->snd_up - tcb->seq);
                        th->urg = 1;
                } else if (after(tcb->seq + 0xFFFF, tp->snd_nxt)) {
                        th->urg_ptr = htons(0xFFFF);
                        th->urg = 1;
                }
        }

        tcp_options_write((__be32 *)(th + 1), tp, &opts);
        if (likely((tcb->tcp_flags & TCPHDR_SYN) == 0))
                TCP_ECN_send(sk, skb, tcp_header_size);

#ifdef CONFIG_TCP_MD5SIG
        /* Calculate the MD5 hash, as we have all we need now */
        if (md5) {
                sk_nocaps_add(sk, NETIF_F_GSO_MASK);
                tp->af_specific->calc_md5_hash(opts.hash_location,
                                               md5, sk, NULL, skb);
        }
#endif

        icsk->icsk_af_ops->send_check(sk, skb);

        if (likely(tcb->tcp_flags & TCPHDR_ACK))
                tcp_event_ack_sent(sk, tcp_skb_pcount(skb));

        if (skb->len != tcp_header_size)
                tcp_event_data_sent(tp, sk);

        if (after(tcb->end_seq, tp->snd_nxt) || tcb->seq == tcb->end_seq)
                TCP_ADD_STATS(sock_net(sk), TCP_MIB_OUTSEGS,
                              tcp_skb_pcount(skb));

        err = icsk->icsk_af_ops->queue_xmit(sk, skb, &inet->cork.fl);
        if (likely(err <= 0))
                return err;

        tcp_enter_cwr(sk);

        return net_xmit_eval(err);
}

/* This routine just queues the buffer for sending.
 *
 * NOTE: probe0 timer is not checked, do not forget tcp_push_pending_frames,
 * otherwise socket can stall.
 */
static void tcp_queue_skb(struct sock *sk, struct sk_buff *skb)
{
        struct tcp_sock *tp = tcp_sk(sk);

        /* Advance write_seq and place onto the write_queue. */
        tp->write_seq = TCP_SKB_CB(skb)->end_seq;
        skb_header_release(skb);
        tcp_add_write_queue_tail(sk, skb);
        sk->sk_wmem_queued += skb->truesize;
        sk_mem_charge(sk, skb->truesize);
}

/* Initialize TSO segments for a packet. */
static void tcp_set_skb_tso_segs(const struct sock *sk, struct sk_buff *skb,
                                 unsigned int mss_now)
{
        struct skb_shared_info *shinfo = skb_shinfo(skb);

        /* Make sure we own this skb before messing gso_size/gso_segs */
        WARN_ON_ONCE(skb_cloned(skb));

        if (skb->len <= mss_now || skb->ip_summed == CHECKSUM_NONE) {
                /* Avoid the costly divide in the normal
                 * non-TSO case.
                 */
                shinfo->gso_segs = 1;
                shinfo->gso_size = 0;
                shinfo->gso_type = 0;
        } else {
                shinfo->gso_segs = DIV_ROUND_UP(skb->len, mss_now);
                shinfo->gso_size = mss_now;
                shinfo->gso_type = sk->sk_gso_type;
        }
}

/* When a modification to fackets out becomes necessary, we need to check
 * skb is counted to fackets_out or not.
 */
static void tcp_adjust_fackets_out(struct sock *sk, const struct sk_buff *skb,
                                   int decr)
{
        struct tcp_sock *tp = tcp_sk(sk);

        if (!tp->sacked_out || tcp_is_reno(tp))
                return;

        if (after(tcp_highest_sack_seq(tp), TCP_SKB_CB(skb)->seq))
                tp->fackets_out -= decr;
}

/* Pcount in the middle of the write queue got changed, we need to do various
 * tweaks to fix counters
 */
static void tcp_adjust_pcount(struct sock *sk, const struct sk_buff *skb, int decr)
{
        struct tcp_sock *tp = tcp_sk(sk);

        tp->packets_out -= decr;

        if (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED)
                tp->sacked_out -= decr;
        if (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_RETRANS)
                tp->retrans_out -= decr;
        if (TCP_SKB_CB(skb)->sacked & TCPCB_LOST)
                tp->lost_out -= decr;

        /* Reno case is special. Sigh... */
        if (tcp_is_reno(tp) && decr > 0)
                tp->sacked_out -= min_t(u32, tp->sacked_out, decr);

        tcp_adjust_fackets_out(sk, skb, decr);

        if (tp->lost_skb_hint &&
            before(TCP_SKB_CB(skb)->seq, TCP_SKB_CB(tp->lost_skb_hint)->seq) &&
            (tcp_is_fack(tp) || (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED)))
                tp->lost_cnt_hint -= decr;

        tcp_verify_left_out(tp);
}

static void tcp_fragment_tstamp(struct sk_buff *skb, struct sk_buff *skb2)
{
        struct skb_shared_info *shinfo = skb_shinfo(skb);

        if (unlikely(shinfo->tx_flags & SKBTX_ANY_TSTAMP) &&
            !before(shinfo->tskey, TCP_SKB_CB(skb2)->seq)) {
                struct skb_shared_info *shinfo2 = skb_shinfo(skb2);
                u8 tsflags = shinfo->tx_flags & SKBTX_ANY_TSTAMP;

                shinfo->tx_flags &= ~tsflags;
                shinfo2->tx_flags |= tsflags;
                swap(shinfo->tskey, shinfo2->tskey);
        }
}

/* Function to create two new TCP segments.  Shrinks the given segment
 * to the specified size and appends a new segment with the rest of the
 * packet to the list.  This won't be called frequently, I hope.
 * Remember, these are still headerless SKBs at this point.
 */
int tcp_fragment(struct sock *sk, struct sk_buff *skb, u32 len,
                 unsigned int mss_now, gfp_t gfp)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct sk_buff *buff;
        int nsize, old_factor;
        int nlen;
        u8 flags;

        if (WARN_ON(len > skb->len))
                return -EINVAL;

        nsize = skb_headlen(skb) - len;
        if (nsize < 0)
                nsize = 0;

        if (skb_unclone(skb, gfp))
                return -ENOMEM;

        /* Get a new skb... force flag on. */
        buff = sk_stream_alloc_skb(sk, nsize, gfp);
        if (buff == NULL)
                return -ENOMEM; /* We'll just try again later. */

        sk->sk_wmem_queued += buff->truesize;
        sk_mem_charge(sk, buff->truesize);
        nlen = skb->len - len - nsize;
        buff->truesize += nlen;
        skb->truesize -= nlen;

        /* Correct the sequence numbers. */
        TCP_SKB_CB(buff)->seq = TCP_SKB_CB(skb)->seq + len;
        TCP_SKB_CB(buff)->end_seq = TCP_SKB_CB(skb)->end_seq;
        TCP_SKB_CB(skb)->end_seq = TCP_SKB_CB(buff)->seq;

        /* PSH and FIN should only be set in the second packet. */
        flags = TCP_SKB_CB(skb)->tcp_flags;
        TCP_SKB_CB(skb)->tcp_flags = flags & ~(TCPHDR_FIN | TCPHDR_PSH);
        TCP_SKB_CB(buff)->tcp_flags = flags;
        TCP_SKB_CB(buff)->sacked = TCP_SKB_CB(skb)->sacked;

        if (!skb_shinfo(skb)->nr_frags && skb->ip_summed != CHECKSUM_PARTIAL) {
                /* Copy and checksum data tail into the new buffer. */
                buff->csum = csum_partial_copy_nocheck(skb->data + len,
                                                       skb_put(buff, nsize),
                                                       nsize, 0);

                skb_trim(skb, len);

                skb->csum = csum_block_sub(skb->csum, buff->csum, len);
        } else {
                skb->ip_summed = CHECKSUM_PARTIAL;
                skb_split(skb, buff, len);
        }

        buff->ip_summed = skb->ip_summed;

        /* Looks stupid, but our code really uses when of
         * skbs, which it never sent before. --ANK
         */
        TCP_SKB_CB(buff)->when = TCP_SKB_CB(skb)->when;
        buff->tstamp = skb->tstamp;
        tcp_fragment_tstamp(skb, buff);

        old_factor = tcp_skb_pcount(skb);

        /* Fix up tso_factor for both original and new SKB.  */
        tcp_set_skb_tso_segs(sk, skb, mss_now);
        tcp_set_skb_tso_segs(sk, buff, mss_now);

        /* If this packet has been sent out already, we must
         * adjust the various packet counters.
         */
        if (!before(tp->snd_nxt, TCP_SKB_CB(buff)->end_seq)) {
                int diff = old_factor - tcp_skb_pcount(skb) -
                        tcp_skb_pcount(buff);

                if (diff)
                        tcp_adjust_pcount(sk, skb, diff);
        }

        /* Link BUFF into the send queue. */
        skb_header_release(buff);
        tcp_insert_write_queue_after(skb, buff, sk);

        return 0;
}

/* This is similar to __pskb_pull_head() (it will go to core/skbuff.c
 * eventually). The difference is that pulled data not copied, but
 * immediately discarded.
 */
static void __pskb_trim_head(struct sk_buff *skb, int len)
{
        struct skb_shared_info *shinfo;
        int i, k, eat;

        eat = min_t(int, len, skb_headlen(skb));
        if (eat) {
                __skb_pull(skb, eat);
                len -= eat;
                if (!len)
                        return;
        }
        eat = len;
        k = 0;
        shinfo = skb_shinfo(skb);
        for (i = 0; i < shinfo->nr_frags; i++) {
                int size = skb_frag_size(&shinfo->frags[i]);

                if (size <= eat) {
                        skb_frag_unref(skb, i);
                        eat -= size;
                } else {
                        shinfo->frags[k] = shinfo->frags[i];
                        if (eat) {
                                shinfo->frags[k].page_offset += eat;
                                skb_frag_size_sub(&shinfo->frags[k], eat);
                                eat = 0;
                        }
                        k++;
                }
        }
        shinfo->nr_frags = k;

        skb_reset_tail_pointer(skb);
        skb->data_len -= len;
        skb->len = skb->data_len;
}

/* Remove acked data from a packet in the transmit queue. */
int tcp_trim_head(struct sock *sk, struct sk_buff *skb, u32 len)
{
        if (skb_unclone(skb, GFP_ATOMIC))
                return -ENOMEM;

        __pskb_trim_head(skb, len);

        TCP_SKB_CB(skb)->seq += len;
        skb->ip_summed = CHECKSUM_PARTIAL;

        skb->truesize        -= len;
        sk->sk_wmem_queued   -= len;
        sk_mem_uncharge(sk, len);
        sock_set_flag(sk, SOCK_QUEUE_SHRUNK);

        /* Any change of skb->len requires recalculation of tso factor. */
        if (tcp_skb_pcount(skb) > 1)
                tcp_set_skb_tso_segs(sk, skb, tcp_skb_mss(skb));

        return 0;
}

/* Calculate MSS not accounting any TCP options.  */
static inline int __tcp_mtu_to_mss(struct sock *sk, int pmtu)
{
        const struct tcp_sock *tp = tcp_sk(sk);
        const struct inet_connection_sock *icsk = inet_csk(sk);
        int mss_now;

        /* Calculate base mss without TCP options:
           It is MMS_S - sizeof(tcphdr) of rfc1122
         */
        mss_now = pmtu - icsk->icsk_af_ops->net_header_len - sizeof(struct tcphdr);

        /* IPv6 adds a frag_hdr in case RTAX_FEATURE_ALLFRAG is set */
        if (icsk->icsk_af_ops->net_frag_header_len) {
                const struct dst_entry *dst = __sk_dst_get(sk);

                if (dst && dst_allfrag(dst))
                        mss_now -= icsk->icsk_af_ops->net_frag_header_len;
        }

        /* Clamp it (mss_clamp does not include tcp options) */
        if (mss_now > tp->rx_opt.mss_clamp)
                mss_now = tp->rx_opt.mss_clamp;

        /* Now subtract optional transport overhead */
        mss_now -= icsk->icsk_ext_hdr_len;

        /* Then reserve room for full set of TCP options and 8 bytes of data */
        if (mss_now < 48)
                mss_now = 48;
        return mss_now;
}

/* Calculate MSS. Not accounting for SACKs here.  */
int tcp_mtu_to_mss(struct sock *sk, int pmtu)
{
        /* Subtract TCP options size, not including SACKs */
        return __tcp_mtu_to_mss(sk, pmtu) -
               (tcp_sk(sk)->tcp_header_len - sizeof(struct tcphdr));
}

/* Inverse of above */
int tcp_mss_to_mtu(struct sock *sk, int mss)
{
        const struct tcp_sock *tp = tcp_sk(sk);
        const struct inet_connection_sock *icsk = inet_csk(sk);
        int mtu;

        mtu = mss +
              tp->tcp_header_len +
              icsk->icsk_ext_hdr_len +
              icsk->icsk_af_ops->net_header_len;

        /* IPv6 adds a frag_hdr in case RTAX_FEATURE_ALLFRAG is set */
        if (icsk->icsk_af_ops->net_frag_header_len) {
                const struct dst_entry *dst = __sk_dst_get(sk);

                if (dst && dst_allfrag(dst))
                        mtu += icsk->icsk_af_ops->net_frag_header_len;
        }
        return mtu;
}

/* MTU probing init per socket */
void tcp_mtup_init(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct inet_connection_sock *icsk = inet_csk(sk);

        icsk->icsk_mtup.enabled = sysctl_tcp_mtu_probing > 1;
        icsk->icsk_mtup.search_high = tp->rx_opt.mss_clamp + sizeof(struct tcphdr) +
                               icsk->icsk_af_ops->net_header_len;
        icsk->icsk_mtup.search_low = tcp_mss_to_mtu(sk, sysctl_tcp_base_mss);
        icsk->icsk_mtup.probe_size = 0;
}
EXPORT_SYMBOL(tcp_mtup_init);

/* This function synchronize snd mss to current pmtu/exthdr set.

   tp->rx_opt.user_mss is mss set by user by TCP_MAXSEG. It does NOT counts
   for TCP options, but includes only bare TCP header.

   tp->rx_opt.mss_clamp is mss negotiated at connection setup.
   It is minimum of user_mss and mss received with SYN.
   It also does not include TCP options.

   inet_csk(sk)->icsk_pmtu_cookie is last pmtu, seen by this function.

   tp->mss_cache is current effective sending mss, including
   all tcp options except for SACKs. It is evaluated,
   taking into account current pmtu, but never exceeds
   tp->rx_opt.mss_clamp.

   NOTE1. rfc1122 clearly states that advertised MSS
   DOES NOT include either tcp or ip options.

   NOTE2. inet_csk(sk)->icsk_pmtu_cookie and tp->mss_cache
   are READ ONLY outside this function.         --ANK (980731)
 */
unsigned int tcp_sync_mss(struct sock *sk, u32 pmtu)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct inet_connection_sock *icsk = inet_csk(sk);
        int mss_now;

        if (icsk->icsk_mtup.search_high > pmtu)
                icsk->icsk_mtup.search_high = pmtu;

        mss_now = tcp_mtu_to_mss(sk, pmtu);
        mss_now = tcp_bound_to_half_wnd(tp, mss_now);

        /* And store cached results */
        icsk->icsk_pmtu_cookie = pmtu;
        if (icsk->icsk_mtup.enabled)
                mss_now = min(mss_now, tcp_mtu_to_mss(sk, icsk->icsk_mtup.search_low));
        tp->mss_cache = mss_now;

        return mss_now;
}
EXPORT_SYMBOL(tcp_sync_mss);

/* Compute the current effective MSS, taking SACKs and IP options,
 * and even PMTU discovery events into account.
 */
unsigned int tcp_current_mss(struct sock *sk)
{
        const struct tcp_sock *tp = tcp_sk(sk);
        const struct dst_entry *dst = __sk_dst_get(sk);
        u32 mss_now;
        unsigned int header_len;
        struct tcp_out_options opts;
        struct tcp_md5sig_key *md5;

        mss_now = tp->mss_cache;

        if (dst) {
                u32 mtu = dst_mtu(dst);
                if (mtu != inet_csk(sk)->icsk_pmtu_cookie)
                        mss_now = tcp_sync_mss(sk, mtu);
        }

        header_len = tcp_established_options(sk, NULL, &opts, &md5) +
                     sizeof(struct tcphdr);
        /* The mss_cache is sized based on tp->tcp_header_len, which assumes
         * some common options. If this is an odd packet (because we have SACK
         * blocks etc) then our calculated header_len will be different, and
         * we have to adjust mss_now correspondingly */
        if (header_len != tp->tcp_header_len) {
                int delta = (int) header_len - tp->tcp_header_len;
                mss_now -= delta;
        }

        return mss_now;
}

/* RFC2861, slow part. Adjust cwnd, after it was not full during one rto.
 * As additional protections, we do not touch cwnd in retransmission phases,
 * and if application hit its sndbuf limit recently.
 */
static void tcp_cwnd_application_limited(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);

        if (inet_csk(sk)->icsk_ca_state == TCP_CA_Open &&
            sk->sk_socket && !test_bit(SOCK_NOSPACE, &sk->sk_socket->flags)) {
                /* Limited by application or receiver window. */
                u32 init_win = tcp_init_cwnd(tp, __sk_dst_get(sk));
                u32 win_used = max(tp->snd_cwnd_used, init_win);
                if (win_used < tp->snd_cwnd) {
                        tp->snd_ssthresh = tcp_current_ssthresh(sk);
                        tp->snd_cwnd = (tp->snd_cwnd + win_used) >> 1;
                }
                tp->snd_cwnd_used = 0;
        }
        tp->snd_cwnd_stamp = tcp_time_stamp;
}
```

```

static void tcp_cwnd_validate(struct sock *sk, bool is_cwnd_limited)
{
        struct tcp_sock *tp = tcp_sk(sk);

        /* Track the maximum number of outstanding packets in each
         * window, and remember whether we were cwnd-limited then.
         */
        if (!before(tp->snd_una, tp->max_packets_seq) ||
            tp->packets_out > tp->max_packets_out) {
                tp->max_packets_out = tp->packets_out;
                tp->max_packets_seq = tp->snd_nxt;
                tp->is_cwnd_limited = is_cwnd_limited;
        }

        if (tcp_is_cwnd_limited(sk)) {
                /* Network is feed fully. */
                tp->snd_cwnd_used = 0;
                tp->snd_cwnd_stamp = tcp_time_stamp;
        } else {
                /* Network starves. */
                if (tp->packets_out > tp->snd_cwnd_used)
                        tp->snd_cwnd_used = tp->packets_out;

                if (sysctl_tcp_slow_start_after_idle &&
                    (s32)(tcp_time_stamp - tp->snd_cwnd_stamp) >= inet_csk(sk)->icsk_rto)
                        tcp_cwnd_application_limited(sk);
        }
}

/* Minshall's variant of the Nagle send check. */
static bool tcp_minshall_check(const struct tcp_sock *tp)
{
        return after(tp->snd_sml, tp->snd_una) &&
                !after(tp->snd_sml, tp->snd_nxt);
}

/* Update snd_sml if this skb is under mss
 * Note that a TSO packet might end with a sub-mss segment
 * The test is really :
 * if ((skb->len % mss) != 0)
 *        tp->snd_sml = TCP_SKB_CB(skb)->end_seq;
 * But we can avoid doing the divide again given we already have
 *  skb_pcount = skb->len / mss_now
 */
static void tcp_minshall_update(struct tcp_sock *tp, unsigned int mss_now,
                                const struct sk_buff *skb)
{
        if (skb->len < tcp_skb_pcount(skb) * mss_now)
                tp->snd_sml = TCP_SKB_CB(skb)->end_seq;
}

/* Return false, if packet can be sent now without violation Nagle's rules:
 * 1. It is full sized. (provided by caller in %partial bool)
 * 2. Or it contains FIN. (already checked by caller)
 * 3. Or TCP_CORK is not set, and TCP_NODELAY is set.
 * 4. Or TCP_CORK is not set, and all sent packets are ACKed.
 *    With Minshall's modification: all sent small packets are ACKed.
 */
static bool tcp_nagle_check(bool partial, const struct tcp_sock *tp,
                            int nonagle)
{
        return partial &&
                ((nonagle & TCP_NAGLE_CORK) ||
                 (!nonagle && tp->packets_out && tcp_minshall_check(tp)));
}
/* Returns the portion of skb which can be sent right away */
static unsigned int tcp_mss_split_point(const struct sock *sk,
                                        const struct sk_buff *skb,
                                        unsigned int mss_now,
                                        unsigned int max_segs,
                                        int nonagle)
{
        const struct tcp_sock *tp = tcp_sk(sk);
        u32 partial, needed, window, max_len;

        window = tcp_wnd_end(tp) - TCP_SKB_CB(skb)->seq;
        max_len = mss_now * max_segs;

        if (likely(max_len <= window && skb != tcp_write_queue_tail(sk)))
                return max_len;

        needed = min(skb->len, window);

        if (max_len <= needed)
                return max_len;

        partial = needed % mss_now;
        /* If last segment is not a full MSS, check if Nagle rules allow us
         * to include this last segment in this skb.
         * Otherwise, we'll split the skb at last MSS boundary
         */
        if (tcp_nagle_check(partial != 0, tp, nonagle))
                return needed - partial;

        return needed;
}

/* Can at least one segment of SKB be sent right now, according to the
 * congestion window rules?  If so, return how many segments are allowed.
 */
static inline unsigned int tcp_cwnd_test(const struct tcp_sock *tp,
                                         const struct sk_buff *skb)
{
        u32 in_flight, cwnd;

        /* Don't be strict about the congestion window for the final FIN.  */
        if ((TCP_SKB_CB(skb)->tcp_flags & TCPHDR_FIN) &&
            tcp_skb_pcount(skb) == 1)
                return 1;

        in_flight = tcp_packets_in_flight(tp);
        cwnd = tp->snd_cwnd;
        if (in_flight < cwnd)
                return (cwnd - in_flight);

        return 0;
}

/* Initialize TSO state of a skb.
 * This must be invoked the first time we consider transmitting
 * SKB onto the wire.
 */
static int tcp_init_tso_segs(const struct sock *sk, struct sk_buff *skb,
                             unsigned int mss_now)
{
        int tso_segs = tcp_skb_pcount(skb);

        if (!tso_segs || (tso_segs > 1 && tcp_skb_mss(skb) != mss_now)) {
                tcp_set_skb_tso_segs(sk, skb, mss_now);
                tso_segs = tcp_skb_pcount(skb);
        }
        return tso_segs;
}


/* Return true if the Nagle test allows this packet to be
 * sent now.
 */
static inline bool tcp_nagle_test(const struct tcp_sock *tp, const struct sk_buff *skb,
                                  unsigned int cur_mss, int nonagle)
{
        /* Nagle rule does not apply to frames, which sit in the middle of the
         * write_queue (they have no chances to get new data).
         *
         * This is implemented in the callers, where they modify the 'nonagle'
         * argument based upon the location of SKB in the send queue.
         */
        if (nonagle & TCP_NAGLE_PUSH)
                return true;

        /* Don't use the nagle rule for urgent data (or for the final FIN). */
        if (tcp_urg_mode(tp) || (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_FIN))
                return true;

        if (!tcp_nagle_check(skb->len < cur_mss, tp, nonagle))
                return true;

        return false;
}

/* Does at least the first segment of SKB fit into the send window? */
static bool tcp_snd_wnd_test(const struct tcp_sock *tp,
                             const struct sk_buff *skb,
                             unsigned int cur_mss)
{
        u32 end_seq = TCP_SKB_CB(skb)->end_seq;

        if (skb->len > cur_mss)
                end_seq = TCP_SKB_CB(skb)->seq + cur_mss;

        return !after(end_seq, tcp_wnd_end(tp));
}

/* This checks if the data bearing packet SKB (usually tcp_send_head(sk))
 * should be put on the wire right now.  If so, it returns the number of
 * packets allowed by the congestion window.
 */
static unsigned int tcp_snd_test(const struct sock *sk, struct sk_buff *skb,
                                 unsigned int cur_mss, int nonagle)
```

```
{
        const struct tcp_sock *tp = tcp_sk(sk);
        unsigned int cwnd_quota;

        tcp_init_tso_segs(sk, skb, cur_mss);

        if (!tcp_nagle_test(tp, skb, cur_mss, nonagle))
                return 0;

        cwnd_quota = tcp_cwnd_test(tp, skb);
        if (cwnd_quota && !tcp_snd_wnd_test(tp, skb, cur_mss))
                cwnd_quota = 0;

        return cwnd_quota;
}

/* Test if sending is allowed right now. */
bool tcp_may_send_now(struct sock *sk)
{
        const struct tcp_sock *tp = tcp_sk(sk);
        struct sk_buff *skb = tcp_send_head(sk);

        return skb &&
                tcp_snd_test(sk, skb, tcp_current_mss(sk),
                             (tcp_skb_is_last(sk, skb) ?
                              tp->nonagle : TCP_NAGLE_PUSH));
}

/* Trim TSO SKB to LEN bytes, put the remaining data into a new packet
 * which is put after SKB on the list.  It is very much like
 * tcp_fragment() except that it may make several kinds of assumptions
 * in order to speed up the splitting operation.  In particular, we
 * know that all the data is in scatter-gather pages, and that the
 * packet has never been sent out before (and thus is not cloned).
 */
static int tso_fragment(struct sock *sk, struct sk_buff *skb, unsigned int len,
                        unsigned int mss_now, gfp_t gfp)
{
        struct sk_buff *buff;
        int nlen = skb->len - len;
        u8 flags;

        /* All of a TSO frame must be composed of paged data.  */
        if (skb->len != skb->data_len)
                return tcp_fragment(sk, skb, len, mss_now, gfp);

        buff = sk_stream_alloc_skb(sk, 0, gfp);
        if (unlikely(buff == NULL))
                return -ENOMEM;

        sk->sk_wmem_queued += buff->truesize;
        sk_mem_charge(sk, buff->truesize);
        buff->truesize += nlen;
        skb->truesize -= nlen;

        /* Correct the sequence numbers. */
        TCP_SKB_CB(buff)->seq = TCP_SKB_CB(skb)->seq + len;
        TCP_SKB_CB(buff)->end_seq = TCP_SKB_CB(skb)->end_seq;
        TCP_SKB_CB(skb)->end_seq = TCP_SKB_CB(buff)->seq;

        /* PSH and FIN should only be set in the second packet. */
        flags = TCP_SKB_CB(skb)->tcp_flags;
        TCP_SKB_CB(skb)->tcp_flags = flags & ~(TCPHDR_FIN | TCPHDR_PSH);
        TCP_SKB_CB(buff)->tcp_flags = flags;

        /* This packet was never sent out yet, so no SACK bits. */
        TCP_SKB_CB(buff)->sacked = 0;

        buff->ip_summed = skb->ip_summed = CHECKSUM_PARTIAL;
        skb_split(skb, buff, len);
        tcp_fragment_tstamp(skb, buff);

        /* Fix up tso_factor for both original and new SKB.  */
        tcp_set_skb_tso_segs(sk, skb, mss_now);
        tcp_set_skb_tso_segs(sk, buff, mss_now);

        /* Link BUFF into the send queue. */
        skb_header_release(buff);
        tcp_insert_write_queue_after(skb, buff, sk);

        return 0;
}

/* Try to defer sending, if possible, in order to minimize the amount
 * of TSO splitting we do.  View it as a kind of TSO Nagle test.
 *
 * This algorithm is from John Heffner.
 */
static bool tcp_tso_should_defer(struct sock *sk, struct sk_buff *skb,
                                 bool *is_cwnd_limited)
```

```
{
        struct tcp_sock *tp = tcp_sk(sk);
        const struct inet_connection_sock *icsk = inet_csk(sk);
        u32 send_win, cong_win, limit, in_flight;
        int win_divisor;

        if (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_FIN)
                goto send_now;

        if (icsk->icsk_ca_state != TCP_CA_Open)
                goto send_now;

        /* Defer for less than two clock ticks. */
        if (tp->tso_deferred &&
            (((u32)jiffies << 1) >> 1) - (tp->tso_deferred >> 1) > 1)
                goto send_now;

        in_flight = tcp_packets_in_flight(tp);

        BUG_ON(tcp_skb_pcount(skb) <= 1 || (tp->snd_cwnd <= in_flight));

        send_win = tcp_wnd_end(tp) - TCP_SKB_CB(skb)->seq;

        /* From in_flight test above, we know that cwnd > in_flight.  */
        cong_win = (tp->snd_cwnd - in_flight) * tp->mss_cache;

        limit = min(send_win, cong_win);

        /* If a full-sized TSO skb can be sent, do it. */
        if (limit >= min_t(unsigned int, sk->sk_gso_max_size,
                           tp->xmit_size_goal_segs * tp->mss_cache))
                goto send_now;

        /* Middle in queue won't get any more data, full sendable already? */
        if ((skb != tcp_write_queue_tail(sk)) && (limit >= skb->len))
                goto send_now;

        win_divisor = ACCESS_ONCE(sysctl_tcp_tso_win_divisor);
        if (win_divisor) {
                u32 chunk = min(tp->snd_wnd, tp->snd_cwnd * tp->mss_cache);

                /* If at least some fraction of a window is available,
                 * just use it.
                 */
                chunk /= win_divisor;
                if (limit >= chunk)
                        goto send_now;
        } else {
                /* Different approach, try not to defer past a single
                 * ACK.  Receiver should ACK every other full sized
                 * frame, so if we have space for more than 3 frames
                 * then send now.
                 */
                if (limit > tcp_max_tso_deferred_mss(tp) * tp->mss_cache)
                        goto send_now;
        }

        /* Ok, it looks like it is advisable to defer.
         * Do not rearm the timer if already set to not break TCP ACK clocking.
         */
        if (!tp->tso_deferred)
                tp->tso_deferred = 1 | (jiffies << 1);

        if (cong_win < send_win && cong_win < skb->len)
                *is_cwnd_limited = true;

        return true;

send_now:
        tp->tso_deferred = 0;
        return false;
}

/* Create a new MTU probe if we are ready.
 * MTU probe is regularly attempting to increase the path MTU by
 * deliberately sending larger packets.  This discovers routing
 * changes resulting in larger path MTUs.
 *
 * Returns 0 if we should wait to probe (no cwnd available),
 *         1 if a probe was sent,
 *         -1 otherwise
 */
static int tcp_mtu_probe(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct inet_connection_sock *icsk = inet_csk(sk);
        struct sk_buff *skb, *nskb, *next;
        int len;
        int probe_size;
        int size_needed;
        int copy;
        int mss_now;

        /* Not currently probing/verifying,
         * not in recovery,
         * have enough cwnd, and
         * not SACKing (the variable headers throw things off) */
        if (!icsk->icsk_mtup.enabled ||
            icsk->icsk_mtup.probe_size ||
            inet_csk(sk)->icsk_ca_state != TCP_CA_Open ||
            tp->snd_cwnd < 11 ||
            tp->rx_opt.num_sacks || tp->rx_opt.dsack)
                return -1;

        /* Very simple search strategy: just double the MSS. */
        mss_now = tcp_current_mss(sk);
        probe_size = 2 * tp->mss_cache;
        size_needed = probe_size + (tp->reordering + 1) * tp->mss_cache;
        if (probe_size > tcp_mtu_to_mss(sk, icsk->icsk_mtup.search_high)) {
                /* TODO: set timer for probe_converge_event */
                return -1;
        }

        /* Have enough data in the send queue to probe? */
        if (tp->write_seq - tp->snd_nxt < size_needed)
                return -1;

        if (tp->snd_wnd < size_needed)
                return -1;
        if (after(tp->snd_nxt + size_needed, tcp_wnd_end(tp)))
                return 0;

        /* Do we need to wait to drain cwnd? With none in flight, don't stall */
        if (tcp_packets_in_flight(tp) + 2 > tp->snd_cwnd) {
                if (!tcp_packets_in_flight(tp))
                        return -1;
                else
                        return 0;
        }

        /* We're allowed to probe.  Build it now. */
        if ((nskb = sk_stream_alloc_skb(sk, probe_size, GFP_ATOMIC)) == NULL)
                return -1;
        sk->sk_wmem_queued += nskb->truesize;
        sk_mem_charge(sk, nskb->truesize);

        skb = tcp_send_head(sk);

        TCP_SKB_CB(nskb)->seq = TCP_SKB_CB(skb)->seq;
        TCP_SKB_CB(nskb)->end_seq = TCP_SKB_CB(skb)->seq + probe_size;
        TCP_SKB_CB(nskb)->tcp_flags = TCPHDR_ACK;
        TCP_SKB_CB(nskb)->sacked = 0;
        nskb->csum = 0;
        nskb->ip_summed = skb->ip_summed;

        tcp_insert_write_queue_before(nskb, skb, sk);

        len = 0;
        tcp_for_write_queue_from_safe(skb, next, sk) {
                copy = min_t(int, skb->len, probe_size - len);
                if (nskb->ip_summed)
                        skb_copy_bits(skb, 0, skb_put(nskb, copy), copy);
                else
                        nskb->csum = skb_copy_and_csum_bits(skb, 0,
                                                            skb_put(nskb, copy),
                                                            copy, nskb->csum);

                if (skb->len <= copy) {
                        /* We've eaten all the data from this skb.
                         * Throw it away. */
                        TCP_SKB_CB(nskb)->tcp_flags |= TCP_SKB_CB(skb)->tcp_flags;
                        tcp_unlink_write_queue(skb, sk);
                        sk_wmem_free_skb(sk, skb);
                } else {
                        TCP_SKB_CB(nskb)->tcp_flags |= TCP_SKB_CB(skb)->tcp_flags &
                                                   ~(TCPHDR_FIN|TCPHDR_PSH);
                        if (!skb_shinfo(skb)->nr_frags) {
                                skb_pull(skb, copy);
                                if (skb->ip_summed != CHECKSUM_PARTIAL)
                                        skb->csum = csum_partial(skb->data,
                                                                 skb->len, 0);
                        } else {
                                __pskb_trim_head(skb, copy);
                                tcp_set_skb_tso_segs(sk, skb, mss_now);
                        }
                        TCP_SKB_CB(skb)->seq += copy;
                }

                len += copy;

                if (len >= probe_size)
                        break;
        }
        tcp_init_tso_segs(sk, nskb, nskb->len);

        /* We're ready to send.  If this fails, the probe will
         * be resegmented into mss-sized pieces by tcp_write_xmit(). */
        TCP_SKB_CB(nskb)->when = tcp_time_stamp;
        if (!tcp_transmit_skb(sk, nskb, 1, GFP_ATOMIC)) {
                /* Decrement cwnd here because we are sending
                 * effectively two packets. */
                tp->snd_cwnd--;
                tcp_event_new_data_sent(sk, nskb);

                icsk->icsk_mtup.probe_size = tcp_mss_to_mtu(sk, nskb->len);
                tp->mtu_probe.probe_seq_start = TCP_SKB_CB(nskb)->seq;
                tp->mtu_probe.probe_seq_end = TCP_SKB_CB(nskb)->end_seq;

                return 1;
        }

        return -1;
}

/* This routine writes packets to the network.  It advances the
 * send_head.  This happens as incoming acks open up the remote
 * window for us.
 *
 * LARGESEND note: !tcp_urg_mode is overkill, only frames between
 * snd_up-64k-mss .. snd_up cannot be large. However, taking into
 * account rare use of URG, this is not a big flaw.
 *
 * Send at most one packet when push_one > 0. Temporarily ignore
 * cwnd limit to force at most one packet out when push_one == 2.

 * Returns true, if no segments are in flight and we have queued segments,
 * but cannot send anything now because of SWS or another problem.
 */
static bool tcp_write_xmit(struct sock *sk, unsigned int mss_now, int nonagle,
                           int push_one, gfp_t gfp)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct sk_buff *skb;
        unsigned int tso_segs, sent_pkts;
        int cwnd_quota;
        int result;
        bool is_cwnd_limited = false;

        sent_pkts = 0;

        if (!push_one) {
                /* Do MTU probing. */
                result = tcp_mtu_probe(sk);
                if (!result) {
                        return false;
                } else if (result > 0) {
                        sent_pkts = 1;
                }
        }

        while ((skb = tcp_send_head(sk))) {
                unsigned int limit;

                tso_segs = tcp_init_tso_segs(sk, skb, mss_now);
                BUG_ON(!tso_segs);

                if (unlikely(tp->repair) && tp->repair_queue == TCP_SEND_QUEUE) {
                        /* "when" is used as a start point for the retransmit timer */
                        TCP_SKB_CB(skb)->when = tcp_time_stamp;
                        goto repair; /* Skip network transmission */
                }

                cwnd_quota = tcp_cwnd_test(tp, skb);
                if (!cwnd_quota) {
                        is_cwnd_limited = true;
                        if (push_one == 2)
                                /* Force out a loss probe pkt. */
                                cwnd_quota = 1;
                        else
                                break;
                }

                if (unlikely(!tcp_snd_wnd_test(tp, skb, mss_now)))
                        break;

                if (tso_segs == 1) {
                        if (unlikely(!tcp_nagle_test(tp, skb, mss_now,
                                                     (tcp_skb_is_last(sk, skb) ?
                                                      nonagle : TCP_NAGLE_PUSH))))
                                break;
                } else {
                        if (!push_one &&
                            tcp_tso_should_defer(sk, skb, &is_cwnd_limited))
                                break;
                }

                /* TCP Small Queues :
                 * Control number of packets in qdisc/devices to two packets / or ~1 ms.
                 * This allows for :
                 *  - better RTT estimation and ACK scheduling
                 *  - faster recovery
                 *  - high rates
                 * Alas, some drivers / subsystems require a fair amount
                 * of queued bytes to ensure line rate.
                 * One example is wifi aggregation (802.11 AMPDU)
                 */
                limit = max_t(unsigned int, sysctl_tcp_limit_output_bytes,
                              sk->sk_pacing_rate >> 10);

                if (atomic_read(&sk->sk_wmem_alloc) > limit) {
                        set_bit(TSQ_THROTTLED, &tp->tsq_flags);
                        /* It is possible TX completion already happened
                         * before we set TSQ_THROTTLED, so we must
                         * test again the condition.
                         */
                        smp_mb__after_atomic();
                        if (atomic_read(&sk->sk_wmem_alloc) > limit)
                                break;
                }

                limit = mss_now;
                if (tso_segs > 1 && !tcp_urg_mode(tp))
                        limit = tcp_mss_split_point(sk, skb, mss_now,
                                                    min_t(unsigned int,
                                                          cwnd_quota,
                                                          sk->sk_gso_max_segs),
                                                    nonagle);

                if (skb->len > limit &&
                    unlikely(tso_fragment(sk, skb, limit, mss_now, gfp)))
                        break;

                TCP_SKB_CB(skb)->when = tcp_time_stamp;

                if (unlikely(tcp_transmit_skb(sk, skb, 1, gfp)))
                        break;

repair:
                /* Advance the send_head.  This one is sent out.
                 * This call will increment packets_out.
                 */
                tcp_event_new_data_sent(sk, skb);

                tcp_minshall_update(tp, mss_now, skb);
                sent_pkts += tcp_skb_pcount(skb);

                if (push_one)
                        break;
        }

        if (likely(sent_pkts)) {
                if (tcp_in_cwnd_reduction(sk))
                        tp->prr_out += sent_pkts;

                /* Send one loss probe per tail loss episode. */
                if (push_one != 2)
                        tcp_schedule_loss_probe(sk);
                tcp_cwnd_validate(sk, is_cwnd_limited);
                return false;
        }
        return (push_one == 2) || (!tp->packets_out && tcp_send_head(sk));
}

bool tcp_schedule_loss_probe(struct sock *sk)
{
        struct inet_connection_sock *icsk = inet_csk(sk);
        struct tcp_sock *tp = tcp_sk(sk);
        u32 timeout, tlp_time_stamp, rto_time_stamp;
        u32 rtt = usecs_to_jiffies(tp->srtt_us >> 3);

        if (WARN_ON(icsk->icsk_pending == ICSK_TIME_EARLY_RETRANS))
                return false;
        /* No consecutive loss probes. */
        if (WARN_ON(icsk->icsk_pending == ICSK_TIME_LOSS_PROBE)) {
                tcp_rearm_rto(sk);
                return false;
        }
        /* Don't do any loss probe on a Fast Open connection before 3WHS
         * finishes.
         */
        if (sk->sk_state == TCP_SYN_RECV)
                return false;

        /* TLP is only scheduled when next timer event is RTO. */
        if (icsk->icsk_pending != ICSK_TIME_RETRANS)
                return false;

        /* Schedule a loss probe in 2*RTT for SACK capable connections
         * in Open state, that are either limited by cwnd or application.
         */
        if (sysctl_tcp_early_retrans < 3 || !tp->srtt_us || !tp->packets_out ||
            !tcp_is_sack(tp) || inet_csk(sk)->icsk_ca_state != TCP_CA_Open)
                return false;

        if ((tp->snd_cwnd > tcp_packets_in_flight(tp)) &&
             tcp_send_head(sk))
                return false;

        /* Probe timeout is at least 1.5*rtt + TCP_DELACK_MAX to account
         * for delayed ack when there's one outstanding packet.
         */
        timeout = rtt << 1;
        if (tp->packets_out == 1)
                timeout = max_t(u32, timeout,
                                (rtt + (rtt >> 1) + TCP_DELACK_MAX));
        timeout = max_t(u32, timeout, msecs_to_jiffies(10));

        /* If RTO is shorter, just schedule TLP in its place. */
        tlp_time_stamp = tcp_time_stamp + timeout;
        rto_time_stamp = (u32)inet_csk(sk)->icsk_timeout;
        if ((s32)(tlp_time_stamp - rto_time_stamp) > 0) {
                s32 delta = rto_time_stamp - tcp_time_stamp;
                if (delta > 0)
                        timeout = delta;
        }

        inet_csk_reset_xmit_timer(sk, ICSK_TIME_LOSS_PROBE, timeout,
                                  TCP_RTO_MAX);
        return true;
}

/* Thanks to skb fast clones, we can detect if a prior transmit of
 * a packet is still in a qdisc or driver queue.
 * In this case, there is very little point doing a retransmit !
 * Note: This is called from BH context only.
 */
static bool skb_still_in_host_queue(const struct sock *sk,
                                    const struct sk_buff *skb)
{
        const struct sk_buff *fclone = skb + 1;

        if (unlikely(skb->fclone == SKB_FCLONE_ORIG &&
                     fclone->fclone == SKB_FCLONE_CLONE)) {
                NET_INC_STATS_BH(sock_net(sk),
                                 LINUX_MIB_TCPSPURIOUS_RTX_HOSTQUEUES);
                return true;
        }
        return false;
}

/* When probe timeout (PTO) fires, send a new segment if one exists, else
 * retransmit the last segment.
 */
void tcp_send_loss_probe(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct sk_buff *skb;
        int pcount;
        int mss = tcp_current_mss(sk);
        int err = -1;

        if (tcp_send_head(sk) != NULL) {
                err = tcp_write_xmit(sk, mss, TCP_NAGLE_OFF, 2, GFP_ATOMIC);
                goto rearm_timer;
        }

        /* At most one outstanding TLP retransmission. */
        if (tp->tlp_high_seq)
                goto rearm_timer;

        /* Retransmit last segment. */
        skb = tcp_write_queue_tail(sk);
        if (WARN_ON(!skb))
                goto rearm_timer;

        if (skb_still_in_host_queue(sk, skb))
                goto rearm_timer;

        pcount = tcp_skb_pcount(skb);
        if (WARN_ON(!pcount))
```

```
                goto rearm_timer;

        if ((pcount > 1) && (skb->len > (pcount - 1) * mss)) {
                if (unlikely(tcp_fragment(sk, skb, (pcount - 1) * mss, mss,
                                          GFP_ATOMIC)))
                        goto rearm_timer;
                skb = tcp_write_queue_tail(sk);
        }

        if (WARN_ON(!skb || !tcp_skb_pcount(skb)))
                goto rearm_timer;

        err = __tcp_retransmit_skb(sk, skb);

        /* Record snd_nxt for loss detection. */
        if (likely(!err))
                tp->tlp_high_seq = tp->snd_nxt;

rearm_timer:
        inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
                                  inet_csk(sk)->icsk_rto,
                                  TCP_RTO_MAX);

        if (likely(!err))
                NET_INC_STATS_BH(sock_net(sk),
                                 LINUX_MIB_TCPLOSSPROBES);
}

/* Push out any pending frames which were held back due to
 * TCP_CORK or attempt at coalescing tiny packets.
 * The socket must be locked by the caller.
 */
void __tcp_push_pending_frames(struct sock *sk, unsigned int cur_mss,
                               int nonagle)
{
        /* If we are closed, the bytes will have to remain here.
         * In time closedown will finish, we empty the write queue and
         * all will be happy.
         */
        if (unlikely(sk->sk_state == TCP_CLOSE))
                return;

        if (tcp_write_xmit(sk, cur_mss, nonagle, 0,
                           sk_gfp_atomic(sk, GFP_ATOMIC)))
                tcp_check_probe_timer(sk);
}

/* Send _single_ skb sitting at the send head. This function requires
 * true push pending frames to setup probe timer etc.
 */
void tcp_push_one(struct sock *sk, unsigned int mss_now)
{
        struct sk_buff *skb = tcp_send_head(sk);

        BUG_ON(!skb || skb->len < mss_now);

        tcp_write_xmit(sk, mss_now, TCP_NAGLE_PUSH, 1, sk->sk_allocation);
}

/* This function returns the amount that we can raise the
 * usable window based on the following constraints
 *
 * 1. The window can never be shrunk once it is offered (RFC 793)
 * 2. We limit memory per socket
 *
 * RFC 1122:
 * "the suggested [SWS] avoidance algorithm for the receiver is to keep
 *  RECV.NEXT + RCV.WIN fixed until:
 *  RCV.BUFF - RCV.USER - RCV.WINDOW >= min(1/2 RCV.BUFF, MSS)"
 *
 * i.e. don't raise the right edge of the window until you can raise
 * it at least MSS bytes.
 *
 * Unfortunately, the recommended algorithm breaks header prediction,
 * since header prediction assumes th->window stays fixed.
 *
 * Strictly speaking, keeping th->window fixed violates the receiver
 * side SWS prevention criteria. The problem is that under this rule
 * a stream of single byte packets will cause the right side of the
 * window to always advance by a single byte.
 *
 * Of course, if the sender implements sender side SWS prevention
 * then this will not be a problem.
 *
 * BSD seems to make the following compromise:
 *
 *      If the free space is less than the 1/4 of the maximum
 *      space available and the free space is less than 1/2 mss,
 *      then set the window to 0.
 *      [ Actually, bsd uses MSS and 1/4 of maximal _window_ ]
 *      Otherwise, just prevent the window from shrinking
 *      and from being larger than the largest representable value.
 *
 * This prevents incremental opening of the window in the regime
 * where TCP is limited by the speed of the reader side taking
 * data out of the TCP receive queue. It does nothing about
 * those cases where the window is constrained on the sender side
 * because the pipeline is full.
 *
 * BSD also seems to "accidentally" limit itself to windows that are a
 * multiple of MSS, at least until the free space gets quite small.
 * This would appear to be a side effect of the mbuf implementation.
 * Combining these two algorithms results in the observed behavior
 * of having a fixed window size at almost all times.
 *
 * Below we obtain similar behavior by forcing the offered window to
 * a multiple of the mss when it is feasible to do so.
 *
 * Note, we don't "adjust" for TIMESTAMP or SACK option bytes.
 * Regular options like TIMESTAMP are taken into account.
 */
u32 __tcp_select_window(struct sock *sk)
{
        struct inet_connection_sock *icsk = inet_csk(sk);
        struct tcp_sock *tp = tcp_sk(sk);
        /* MSS for the peer's data.  Previous versions used mss_clamp
         * here.  I don't know if the value based on our guesses
         * of peer's MSS is better for the performance.  It's more correct
         * but may be worse for the performance because of rcv_mss
         * fluctuations.  --SAW  1998/11/1
         */
        int mss = icsk->icsk_ack.rcv_mss;
        int free_space = tcp_space(sk);
        int allowed_space = tcp_full_space(sk);
        int full_space = min_t(int, tp->window_clamp, allowed_space);
        int window;

        if (mss > full_space)
                mss = full_space;

        if (free_space < (full_space >> 1)) {
                icsk->icsk_ack.quick = 0;

                if (sk_under_memory_pressure(sk))
                        tp->rcv_ssthresh = min(tp->rcv_ssthresh,
                                               4U * tp->advmss);

                /* free_space might become our new window, make sure we don't
                 * increase it due to wscale.
                 */
                free_space = round_down(free_space, 1 << tp->rx_opt.rcv_wscale);

                /* if free space is less than mss estimate, or is below 1/16th
                 * of the maximum allowed, try to move to zero-window, else
                 * tcp_clamp_window() will grow rcv buf up to tcp_rmem[2], and
                 * new incoming data is dropped due to memory limits.
                 * With large window, mss test triggers way too late in order
                 * to announce zero window in time before rmem limit kicks in.
                 */
                if (free_space < (allowed_space >> 4) || free_space < mss)
                        return 0;
        }

        if (free_space > tp->rcv_ssthresh)
                free_space = tp->rcv_ssthresh;

        /* Don't do rounding if we are using window scaling, since the
         * scaled window will not line up with the MSS boundary anyway.
         */
        window = tp->rcv_wnd;
        if (tp->rx_opt.rcv_wscale) {
                window = free_space;

                /* Advertise enough space so that it won't get scaled away.
                 * Import case: prevent zero window announcement if
                 * 1<<rcv_wscale > mss.
                 */
                if (((window >> tp->rx_opt.rcv_wscale) << tp->rx_opt.rcv_wscale) != window)
                        window = (((window >> tp->rx_opt.rcv_wscale) + 1)
                                  << tp->rx_opt.rcv_wscale);
        } else {
                /* Get the largest window that is a nice multiple of mss.
                 * Window clamp already applied above.
                 * If our current window offering is within 1 mss of the
                 * free space we just keep it. This prevents the divide
                 * and multiply from happening most of the time.
                 * We also don't do any window rounding when the free space
                 * is too small.
                 */
                if (window <= free_space - mss || window > free_space)
```

```
                        window = (free_space / mss) * mss;
                else if (mss == full_space &&
                         free_space > window + (full_space >> 1))
                        window = free_space;
        }

        return window;
}

/* Collapses two adjacent SKB's during retransmission. */
static void tcp_collapse_retrans(struct sock *sk, struct sk_buff *skb)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct sk_buff *next_skb = tcp_write_queue_next(sk, skb);
        int skb_size, next_skb_size;

        skb_size = skb->len;
        next_skb_size = next_skb->len;

        BUG_ON(tcp_skb_pcount(skb) != 1 || tcp_skb_pcount(next_skb) != 1);

        tcp_highest_sack_combine(sk, next_skb, skb);

        tcp_unlink_write_queue(next_skb, sk);

        skb_copy_from_linear_data(next_skb, skb_put(skb, next_skb_size),
                                  next_skb_size);

        if (next_skb->ip_summed == CHECKSUM_PARTIAL)
                skb->ip_summed = CHECKSUM_PARTIAL;

        if (skb->ip_summed != CHECKSUM_PARTIAL)
                skb->csum = csum_block_add(skb->csum, next_skb->csum, skb_size);

        /* Update sequence range on original skb. */
        TCP_SKB_CB(skb)->end_seq = TCP_SKB_CB(next_skb)->end_seq;

        /* Merge over control information. This moves PSH/FIN etc. over */
        TCP_SKB_CB(skb)->tcp_flags |= TCP_SKB_CB(next_skb)->tcp_flags;

        /* All done, get rid of second SKB and account for it so
         * packet counting does not break.
         */
        TCP_SKB_CB(skb)->sacked |= TCP_SKB_CB(next_skb)->sacked & TCPCB_EVER_RETRANS;

        /* changed transmit queue under us so clear hints */
        tcp_clear_retrans_hints_partial(tp);
        if (next_skb == tp->retransmit_skb_hint)
                tp->retransmit_skb_hint = skb;

        tcp_adjust_pcount(sk, next_skb, tcp_skb_pcount(next_skb));

        sk_wmem_free_skb(sk, next_skb);
}

/* Check if coalescing SKBs is legal. */
static bool tcp_can_collapse(const struct sock *sk, const struct sk_buff *skb)
{
        if (tcp_skb_pcount(skb) > 1)
                return false;
        /* TODO: SACK collapsing could be used to remove this condition */
        if (skb_shinfo(skb)->nr_frags != 0)
                return false;
        if (skb_cloned(skb))
                return false;
        if (skb == tcp_send_head(sk))
                return false;
        /* Some heurestics for collapsing over SACK'd could be invented */
        if (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED)
                return false;

        return true;
}

/* Collapse packets in the retransmit queue to make to create
 * less packets on the wire. This is only done on retransmission.
 */
static void tcp_retrans_try_collapse(struct sock *sk, struct sk_buff *to,
                                     int space)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct sk_buff *skb = to, *tmp;
        bool first = true;

        if (!sysctl_tcp_retrans_collapse)
                return;
        if (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_SYN)
                return;

        tcp_for_write_queue_from_safe(skb, tmp, sk) {
                if (!tcp_can_collapse(sk, skb))
                        break;

                space -= skb->len;

                if (first) {
                        first = false;
                        continue;
                }

                if (space < 0)
                        break;
                /* Punt if not enough space exists in the first SKB for
                 * the data in the second
                 */
                if (skb->len > skb_availroom(to))
                        break;

                if (after(TCP_SKB_CB(skb)->end_seq, tcp_wnd_end(tp)))
                        break;

                tcp_collapse_retrans(sk, to);
        }
}

/* This retransmits one SKB.  Policy decisions and retransmit queue
 * state updates are done by the caller.  Returns non-zero if an
 * error occurred which prevented the send.
 */
int __tcp_retransmit_skb(struct sock *sk, struct sk_buff *skb)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct inet_connection_sock *icsk = inet_csk(sk);
        unsigned int cur_mss;
        int err;

        /* Inconslusive MTU probe */
        if (icsk->icsk_mtup.probe_size) {
                icsk->icsk_mtup.probe_size = 0;
        }

        /* Do not sent more than we queued. 1/4 is reserved for possible
         * copying overhead: fragmentation, tunneling, mangling etc.
         */
        if (atomic_read(&sk->sk_wmem_alloc) >
            min(sk->sk_wmem_queued + (sk->sk_wmem_queued >> 2), sk->sk_sndbuf))
                return -EAGAIN;

        if (skb_still_in_host_queue(sk, skb))
                return -EBUSY;

        if (before(TCP_SKB_CB(skb)->seq, tp->snd_una)) {
                if (before(TCP_SKB_CB(skb)->end_seq, tp->snd_una))
                        BUG();
                if (tcp_trim_head(sk, skb, tp->snd_una - TCP_SKB_CB(skb)->seq))
                        return -ENOMEM;
        }

        if (inet_csk(sk)->icsk_af_ops->rebuild_header(sk))
                return -EHOSTUNREACH; /* Routing failure or similar. */

        cur_mss = tcp_current_mss(sk);

        /* If receiver has shrunk his window, and skb is out of
         * new window, do not retransmit it. The exception is the
         * case, when window is shrunk to zero. In this case
         * our retransmit serves as a zero window probe.
         */
        if (!before(TCP_SKB_CB(skb)->seq, tcp_wnd_end(tp)) &&
            TCP_SKB_CB(skb)->seq != tp->snd_una)
                return -EAGAIN;

        if (skb->len > cur_mss) {
                if (tcp_fragment(sk, skb, cur_mss, cur_mss, GFP_ATOMIC))
                        return -ENOMEM; /* We'll try again later. */
        } else {
                int oldpcount = tcp_skb_pcount(skb);

                if (unlikely(oldpcount > 1)) {
                        if (skb_unclone(skb, GFP_ATOMIC))
                                return -ENOMEM;
                        tcp_init_tso_segs(sk, skb, cur_mss);
                        tcp_adjust_pcount(sk, skb, oldpcount - tcp_skb_pcount(skb));
                }
        }

        tcp_retrans_try_collapse(sk, skb, cur_mss);

        /* Make a copy, if the first transmission SKB clone we made
         * is still in somebody's hands, else make a clone.
         */
        TCP_SKB_CB(skb)->when = tcp_time_stamp;

        /* make sure skb->data is aligned on arches that require it
         * and check if ack-trimming & collapsing extended the headroom
         * beyond what csum_start can cover.
         */
        if (unlikely((NET_IP_ALIGN && ((unsigned long)skb->data & 3)) ||
                     skb_headroom(skb) >= 0xFFFF)) {
                struct sk_buff *nskb = __pskb_copy(skb, MAX_TCP_HEADER,
                                                   GFP_ATOMIC);
                err = nskb ? tcp_transmit_skb(sk, nskb, 0, GFP_ATOMIC) :
                             -ENOBUFS;
        } else {
                err = tcp_transmit_skb(sk, skb, 1, GFP_ATOMIC);
        }

        if (likely(!err)) {
                TCP_SKB_CB(skb)->sacked |= TCPCB_EVER_RETRANS;
                /* Update global TCP statistics. */
                TCP_INC_STATS(sock_net(sk), TCP_MIB_RETRANSSEGS);
                if (TCP_SKB_CB(skb)->tcp_flags & TCPHDR_SYN)
                        NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPSYNRETRANS);
                tp->total_retrans++;
        }
        return err;
}

int tcp_retransmit_skb(struct sock *sk, struct sk_buff *skb)
{
        struct tcp_sock *tp = tcp_sk(sk);
        int err = __tcp_retransmit_skb(sk, skb);

        if (err == 0) {
#if FASTRETRANS_DEBUG > 0
                if (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_RETRANS) {
                        net_dbg_ratelimited("retrans_out leaked\n");
                }
#endif
                if (!tp->retrans_out)
                        tp->lost_retrans_low = tp->snd_nxt;
                TCP_SKB_CB(skb)->sacked |= TCPCB_RETRANS;
                tp->retrans_out += tcp_skb_pcount(skb);

                /* Save stamp of the first retransmit. */
                if (!tp->retrans_stamp)
                        tp->retrans_stamp = TCP_SKB_CB(skb)->when;

                /* snd_nxt is stored to detect loss of retransmitted segment,
                 * see tcp_input.c tcp_sacktag_write_queue().
                 */
                TCP_SKB_CB(skb)->ack_seq = tp->snd_nxt;
        } else if (err != -EBUSY) {
                NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPRETRANSFAIL);
        }

        if (tp->undo_retrans < 0)
                tp->undo_retrans = 0;
        tp->undo_retrans += tcp_skb_pcount(skb);
        return err;
}

/* Check if we forward retransmits are possible in the current
 * window/congestion state.
 */
static bool tcp_can_forward_retransmit(struct sock *sk)
{
        const struct inet_connection_sock *icsk = inet_csk(sk);
        const struct tcp_sock *tp = tcp_sk(sk);

        /* Forward retransmissions are possible only during Recovery. */
        if (icsk->icsk_ca_state != TCP_CA_Recovery)
                return false;

        /* No forward retransmissions in Reno are possible. */
        if (tcp_is_reno(tp))
                return false;

        /* Yeah, we have to make difficult choice between forward transmission
         * and retransmission... Both ways have their merits...
         *
         * For now we do not retransmit anything, while we have some new
         * segments to send. In the other cases, follow rule 3 for
         * NextSeg() specified in RFC3517.
         */

        if (tcp_may_send_now(sk))
                return false;

        return true;
```

```
}

/* This gets called after a retransmit timeout, and the initially
 * retransmitted data is acknowledged.  It tries to continue
 * resending the rest of the retransmit queue, until either
 * we've sent it all or the congestion window limit is reached.
 * If doing SACK, the first ACK which comes back for a timeout
 * based retransmit packet might feed us FACK information again.
 * If so, we use it to avoid unnecessarily retransmissions.
 */
void tcp_xmit_retransmit_queue(struct sock *sk)
{
        const struct inet_connection_sock *icsk = inet_csk(sk);
        struct tcp_sock *tp = tcp_sk(sk);
        struct sk_buff *skb;
        struct sk_buff *hole = NULL;
        u32 last_lost;
        int mib_idx;
        int fwd_rexmitting = 0;

        if (!tp->packets_out)
                return;

        if (!tp->lost_out)
                tp->retransmit_high = tp->snd_una;

        if (tp->retransmit_skb_hint) {
                skb = tp->retransmit_skb_hint;
                last_lost = TCP_SKB_CB(skb)->end_seq;
                if (after(last_lost, tp->retransmit_high))
                        last_lost = tp->retransmit_high;
        } else {
                skb = tcp_write_queue_head(sk);
                last_lost = tp->snd_una;
        }

        tcp_for_write_queue_from(skb, sk) {
                __u8 sacked = TCP_SKB_CB(skb)->sacked;

                if (skb == tcp_send_head(sk))
                        break;
                /* we could do better than to assign each time */
                if (hole == NULL)
                        tp->retransmit_skb_hint = skb;

                /* Assume this retransmit will generate
                 * only one packet for congestion window
                 * calculation purposes.  This works because
                 * tcp_retransmit_skb() will chop up the
                 * packet to be MSS sized and all the
                 * packet counting works out.
                 */
                if (tcp_packets_in_flight(tp) >= tp->snd_cwnd)
                        return;

                if (fwd_rexmitting) {
begin_fwd:
                        if (!before(TCP_SKB_CB(skb)->seq, tcp_highest_sack_seq(tp)))
                                break;
                        mib_idx = LINUX_MIB_TCPFORWARDRETRANS;

                } else if (!before(TCP_SKB_CB(skb)->seq, tp->retransmit_high)) {
                        tp->retransmit_high = last_lost;
                        if (!tcp_can_forward_retransmit(sk))
                                break;
                        /* Backtrack if necessary to non-L'ed skb */
                        if (hole != NULL) {
                                skb = hole;
                                hole = NULL;
                        }
                        fwd_rexmitting = 1;
                        goto begin_fwd;

                } else if (!(sacked & TCPCB_LOST)) {
                        if (hole == NULL && !(sacked & (TCPCB_SACKED_RETRANS|TCPCB_SACKED_ACKED)))
                                hole = skb;
                        continue;

                } else {
                        last_lost = TCP_SKB_CB(skb)->end_seq;
                        if (icsk->icsk_ca_state != TCP_CA_Loss)
                                mib_idx = LINUX_MIB_TCPFASTRETRANS;
                        else
                                mib_idx = LINUX_MIB_TCPSLOWSTARTRETRANS;
                }

                if (sacked & (TCPCB_SACKED_ACKED|TCPCB_SACKED_RETRANS))
                        continue;

                if (tcp_retransmit_skb(sk, skb))
```

```
                        return;

                NET_INC_STATS_BH(sock_net(sk), mib_idx);

                if (tcp_in_cwnd_reduction(sk))
                        tp->prr_out += tcp_skb_pcount(skb);

                if (skb == tcp_write_queue_head(sk))
                        inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
                                                  inet_csk(sk)->icsk_rto,
                                                  TCP_RTO_MAX);
        }
}

/* Send a fin.  The caller locks the socket for us.  This cannot be
 * allowed to fail queueing a FIN frame under any circumstances.
 */
void tcp_send_fin(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct sk_buff *skb = tcp_write_queue_tail(sk);
        int mss_now;

        /* Optimization, tack on the FIN if we have a queue of
         * unsent frames.  But be careful about outgoing SACKS
         * and IP options.
         */
        mss_now = tcp_current_mss(sk);

        if (tcp_send_head(sk) != NULL) {
                TCP_SKB_CB(skb)->tcp_flags |= TCPHDR_FIN;
                TCP_SKB_CB(skb)->end_seq++;
                tp->write_seq++;
        } else {
                /* Socket is locked, keep trying until memory is available. */
                for (;;) {
                        skb = alloc_skb_fclone(MAX_TCP_HEADER,
                                               sk->sk_allocation);
                        if (skb)
                                break;
                        yield();
                }

                /* Reserve space for headers and prepare control bits. */
                skb_reserve(skb, MAX_TCP_HEADER);
                /* FIN eats a sequence byte, write_seq advanced by tcp_queue_skb(). */
                tcp_init_nondata_skb(skb, tp->write_seq,
                                     TCPHDR_ACK | TCPHDR_FIN);
                tcp_queue_skb(sk, skb);
        }
        __tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_OFF);
}

/* We get here when a process closes a file descriptor (either due to
 * an explicit close() or as a byproduct of exit()'ing) and there
 * was unread data in the receive queue.  This behavior is recommended
 * by RFC 2525, section 2.17.  -DaveM
 */
void tcp_send_active_reset(struct sock *sk, gfp_t priority)
{
        struct sk_buff *skb;

        /* NOTE: No TCP options attached and we never retransmit this. */
        skb = alloc_skb(MAX_TCP_HEADER, priority);
        if (!skb) {
                NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPABORTFAILED);
                return;
        }

        /* Reserve space for headers and prepare control bits. */
        skb_reserve(skb, MAX_TCP_HEADER);
        tcp_init_nondata_skb(skb, tcp_acceptable_seq(sk),
                             TCPHDR_ACK | TCPHDR_RST);
        /* Send it off. */
        TCP_SKB_CB(skb)->when = tcp_time_stamp;
        if (tcp_transmit_skb(sk, skb, 0, priority))
                NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPABORTFAILED);

        TCP_INC_STATS(sock_net(sk), TCP_MIB_OUTRSTS);
}

/* Send a crossed SYN-ACK during socket establishment.
 * WARNING: This routine must only be called when we have already sent
 * a SYN packet that crossed the incoming SYN that caused this routine
 * to get called. If this assumption fails then the initial rcv_wnd
 * and rcv_wscale values will not be correct.
 */
int tcp_send_synack(struct sock *sk)
{
        struct sk_buff *skb;
        skb = tcp_write_queue_head(sk);
        if (skb == NULL || !(TCP_SKB_CB(skb)->tcp_flags & TCPHDR_SYN)) {
                pr_debug("%s: wrong queue state\n", __func__);
                return -EFAULT;
        }
        if (!(TCP_SKB_CB(skb)->tcp_flags & TCPHDR_ACK)) {
                if (skb_cloned(skb)) {
                        struct sk_buff *nskb = skb_copy(skb, GFP_ATOMIC);
                        if (nskb == NULL)
                                return -ENOMEM;
                        tcp_unlink_write_queue(skb, sk);
                        skb_header_release(nskb);
                        __tcp_add_write_queue_head(sk, nskb);
                        sk_wmem_free_skb(sk, skb);
                        sk->sk_wmem_queued += nskb->truesize;
                        sk_mem_charge(sk, nskb->truesize);
                        skb = nskb;
                }

                TCP_SKB_CB(skb)->tcp_flags |= TCPHDR_ACK;
                TCP_ECN_send_synack(tcp_sk(sk), skb);
        }
        TCP_SKB_CB(skb)->when = tcp_time_stamp;
        return tcp_transmit_skb(sk, skb, 1, GFP_ATOMIC);
}

/**
 * tcp_make_synack - Prepare a SYN-ACK.
 * sk: listener socket
 * dst: dst entry attached to the SYNACK
 * req: request_sock pointer
 *
 * Allocate one skb and build a SYNACK packet.
 * @dst is consumed : Caller should not use it again.
 */
struct sk_buff *tcp_make_synack(struct sock *sk, struct dst_entry *dst,
                                struct request_sock *req,
                                struct tcp_fastopen_cookie *foc)
{
        struct tcp_out_options opts;
        struct inet_request_sock *ireq = inet_rsk(req);
        struct tcp_sock *tp = tcp_sk(sk);
        struct tcphdr *th;
        struct sk_buff *skb;
        struct tcp_md5sig_key *md5;
        int tcp_header_size;
        int mss;

        skb = sock_wmalloc(sk, MAX_TCP_HEADER, 1, GFP_ATOMIC);
        if (unlikely(!skb)) {
                dst_release(dst);
                return NULL;
        }
        /* Reserve space for headers. */
        skb_reserve(skb, MAX_TCP_HEADER);

        skb_dst_set(skb, dst);
        security_skb_owned_by(skb, sk);

        mss = dst_metric_advmss(dst);
        if (tp->rx_opt.user_mss && tp->rx_opt.user_mss < mss)
                mss = tp->rx_opt.user_mss;

        memset(&opts, 0, sizeof(opts));
#ifdef CONFIG_SYN_COOKIES
        if (unlikely(req->cookie_ts))
                TCP_SKB_CB(skb)->when = cookie_init_timestamp(req);
        else
#endif
        TCP_SKB_CB(skb)->when = tcp_time_stamp;
        tcp_header_size = tcp_synack_options(sk, req, mss, skb, &opts, &md5,
                                             foc) + sizeof(*th);

        skb_push(skb, tcp_header_size);
        skb_reset_transport_header(skb);

        th = tcp_hdr(skb);
        memset(th, 0, sizeof(struct tcphdr));
        th->syn = 1;
        th->ack = 1;
        TCP_ECN_make_synack(req, th);
        th->source = htons(ireq->ir_num);
        th->dest = ireq->ir_rmt_port;
        /* Setting of flags are superfluous here for callers (and ECE is
         * not even correctly set)
         */
        tcp_init_nondata_skb(skb, tcp_rsk(req)->snt_isn,
                             TCPHDR_SYN | TCPHDR_ACK);
        th->seq = htonl(TCP_SKB_CB(skb)->seq);
        /* XXX data is queued and acked as is. No buffer/window check */
        th->ack_seq = htonl(tcp_rsk(req)->rcv_nxt);

        /* RFC1323: The window in SYN & SYN/ACK segments is never scaled. */
        th->window = htons(min(req->rcv_wnd, 65535U));
        tcp_options_write((__be32 *)(th + 1), tp, &opts);
        th->doff = (tcp_header_size >> 2);
        TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_OUTSEGS);

#ifdef CONFIG_TCP_MD5SIG
        /* Okay, we have all we need - do the md5 hash if needed */
        if (md5) {
                tcp_rsk(req)->af_specific->calc_md5_hash(opts.hash_location,
                                               md5, NULL, req, skb);
        }
#endif

        return skb;
}
EXPORT_SYMBOL(tcp_make_synack);

/* Do all connect socket setups that can be done AF independent. */
static void tcp_connect_init(struct sock *sk)
{
        const struct dst_entry *dst = __sk_dst_get(sk);
        struct tcp_sock *tp = tcp_sk(sk);
        __u8 rcv_wscale;

        /* We'll fix this up when we get a response from the other end.
         * See tcp_input.c:tcp_rcv_state_process case TCP_SYN_SENT.
         */
        tp->tcp_header_len = sizeof(struct tcphdr) +
                (sysctl_tcp_timestamps ? TCPOLEN_TSTAMP_ALIGNED : 0);

#ifdef CONFIG_TCP_MD5SIG
        if (tp->af_specific->md5_lookup(sk, sk) != NULL)
                tp->tcp_header_len += TCPOLEN_MD5SIG_ALIGNED;
#endif

        /* If user gave his TCP_MAXSEG, record it to clamp */
        if (tp->rx_opt.user_mss)
                tp->rx_opt.mss_clamp = tp->rx_opt.user_mss;
        tp->max_window = 0;
        tcp_mtup_init(sk);
        tcp_sync_mss(sk, dst_mtu(dst));

        if (!tp->window_clamp)
                tp->window_clamp = dst_metric(dst, RTAX_WINDOW);
        tp->advmss = dst_metric_advmss(dst);
        if (tp->rx_opt.user_mss && tp->rx_opt.user_mss < tp->advmss)
                tp->advmss = tp->rx_opt.user_mss;

        tcp_initialize_rcv_mss(sk);

        /* limit the window selection if the user enforce a smaller rx buffer */
        if (sk->sk_userlocks & SOCK_RCVBUF_LOCK &&
            (tp->window_clamp > tcp_full_space(sk) || tp->window_clamp == 0))
                tp->window_clamp = tcp_full_space(sk);

        tcp_select_initial_window(tcp_full_space(sk),
                                  tp->advmss - (tp->rx_opt.ts_recent_stamp ? tp->tcp_header_len - sizeof(struct tcphdr) : 0),
                                  &tp->rcv_wnd,
                                  &tp->window_clamp,
                                  sysctl_tcp_window_scaling,
                                  &rcv_wscale,
                                  dst_metric(dst, RTAX_INITRWND));

        tp->rx_opt.rcv_wscale = rcv_wscale;
        tp->rcv_ssthresh = tp->rcv_wnd;

        sk->sk_err = 0;
        sock_reset_flag(sk, SOCK_DONE);
        tp->snd_wnd = 0;
        tcp_init_wl(tp, 0);
        tp->snd_una = tp->write_seq;
        tp->snd_sml = tp->write_seq;
        tp->snd_up = tp->write_seq;
        tp->snd_nxt = tp->write_seq;

        if (likely(!tp->repair))
                tp->rcv_nxt = 0;
        else
                tp->rcv_tstamp = tcp_time_stamp;
        tp->rcv_wup = tp->rcv_nxt;
        tp->copied_seq = tp->rcv_nxt;

        inet_csk(sk)->icsk_rto = TCP_TIMEOUT_INIT;
        inet_csk(sk)->icsk_retransmits = 0;
        tcp_clear_retrans(tp);
```

```
}

static void tcp_connect_queue_skb(struct sock *sk, struct sk_buff *skb)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct tcp_skb_cb *tcb = TCP_SKB_CB(skb);

        tcb->end_seq += skb->len;
        skb_header_release(skb);
        __tcp_add_write_queue_tail(sk, skb);
        sk->sk_wmem_queued += skb->truesize;
        sk_mem_charge(sk, skb->truesize);
        tp->write_seq = tcb->end_seq;
        tp->packets_out += tcp_skb_pcount(skb);
}

/* Build and send a SYN with data and (cached) Fast Open cookie. However,
 * queue a data-only packet after the regular SYN, such that regular SYNs
 * are retransmitted on timeouts. Also if the remote SYN-ACK acknowledges
 * only the SYN sequence, the data are retransmitted in the first ACK.
 * If cookie is not cached or other error occurs, falls back to send a
 * regular SYN with Fast Open cookie request option.
 */
static int tcp_send_syn_data(struct sock *sk, struct sk_buff *syn)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct tcp_fastopen_request *fo = tp->fastopen_req;
        int syn_loss = 0, space, i, err = 0, iovlen = fo->data->msg_iovlen;
        struct sk_buff *syn_data = NULL, *data;
        unsigned long last_syn_loss = 0;

        tp->rx_opt.mss_clamp = tp->advmss;  /* If MSS is not cached */
        tcp_fastopen_cache_get(sk, &tp->rx_opt.mss_clamp, &fo->cookie,
                               &syn_loss, &last_syn_loss);
        /* Recurring FO SYN losses: revert to regular handshake temporarily */
        if (syn_loss > 1 &&
            time_before(jiffies, last_syn_loss + (60*HZ << syn_loss))) {
                fo->cookie.len = -1;
                goto fallback;
        }

        if (sysctl_tcp_fastopen & TFO_CLIENT_NO_COOKIE)
                fo->cookie.len = -1;
        else if (fo->cookie.len <= 0)
                goto fallback;

        /* MSS for SYN-data is based on cached MSS and bounded by PMTU and
         * user-MSS. Reserve maximum option space for middleboxes that add
         * private TCP options. The cost is reduced data space in SYN :(
         */
        if (tp->rx_opt.user_mss && tp->rx_opt.user_mss < tp->rx_opt.mss_clamp)
                tp->rx_opt.mss_clamp = tp->rx_opt.user_mss;
        space = __tcp_mtu_to_mss(sk, inet_csk(sk)->icsk_pmtu_cookie) -
                MAX_TCP_OPTION_SPACE;

        space = min_t(size_t, space, fo->size);

        /* limit to order-0 allocations */
        space = min_t(size_t, space, SKB_MAX_HEAD(MAX_TCP_HEADER));

        syn_data = skb_copy_expand(syn, MAX_TCP_HEADER, space,
                                   sk->sk_allocation);
        if (syn_data == NULL)
                goto fallback;

        for (i = 0; i < iovlen && syn_data->len < space; ++i) {
                struct iovec *iov = &fo->data->msg_iov[i];
                unsigned char __user *from = iov->iov_base;
                int len = iov->iov_len;

                if (syn_data->len + len > space)
                        len = space - syn_data->len;
                else if (i + 1 == iovlen)
                        /* No more data pending in inet_wait_for_connect() */
                        fo->data = NULL;

                if (skb_add_data(syn_data, from, len))
                        goto fallback;
        }

        /* Queue a data-only packet after the regular SYN for retransmission */
        data = pskb_copy(syn_data, sk->sk_allocation);
        if (data == NULL)
                goto fallback;
        TCP_SKB_CB(data)->seq++;
        TCP_SKB_CB(data)->tcp_flags &= ~TCPHDR_SYN;
        TCP_SKB_CB(data)->tcp_flags = (TCPHDR_ACK|TCPHDR_PSH);
        tcp_connect_queue_skb(sk, data);
        fo->copied = data->len;

        /* syn_data is about to be sent, we need to take current time stamps
         * for the packets that are in write queue : SYN packet and DATA
         */
        skb_mstamp_get(&syn->skb_mstamp);
        data->skb_mstamp = syn->skb_mstamp;

        if (tcp_transmit_skb(sk, syn_data, 0, sk->sk_allocation) == 0) {
                tp->syn_data = (fo->copied > 0);
                NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPORIGDATASENT);
                goto done;
        }
        syn_data = NULL;

fallback:
        /* Send a regular SYN with Fast Open cookie request option */
        if (fo->cookie.len > 0)
                fo->cookie.len = 0;
        err = tcp_transmit_skb(sk, syn, 1, sk->sk_allocation);
        if (err)
                tp->syn_fastopen = 0;
        kfree_skb(syn_data);
done:
        fo->cookie.len = -1;  /* Exclude Fast Open option for SYN retries */
        return err;
}

/* Build a SYN and send it off. */
int tcp_connect(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct sk_buff *buff;
        int err;

        tcp_connect_init(sk);

        if (unlikely(tp->repair)) {
                tcp_finish_connect(sk, NULL);
                return 0;
        }

        buff = alloc_skb_fclone(MAX_TCP_HEADER + 15, sk->sk_allocation);
        if (unlikely(buff == NULL))
                return -ENOBUFS;

        /* Reserve space for headers. */
        skb_reserve(buff, MAX_TCP_HEADER);

        tcp_init_nondata_skb(buff, tp->write_seq++, TCPHDR_SYN);
        tp->retrans_stamp = TCP_SKB_CB(buff)->when = tcp_time_stamp;
        tcp_connect_queue_skb(sk, buff);
        TCP_ECN_send_syn(sk, buff);

        /* Send off SYN; include data in Fast Open. */
        err = tp->fastopen_req ? tcp_send_syn_data(sk, buff) :
              tcp_transmit_skb(sk, buff, 1, sk->sk_allocation);
        if (err == -ECONNREFUSED)
                return err;

        /* We change tp->snd_nxt after the tcp_transmit_skb() call
         * in order to make this packet get counted in tcpOutSegs.
         */
        tp->snd_nxt = tp->write_seq;
        tp->pushed_seq = tp->write_seq;
        TCP_INC_STATS(sock_net(sk), TCP_MIB_ACTIVEOPENS);

        /* Timer for repeating the SYN until an answer. */
        inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
                                  inet_csk(sk)->icsk_rto, TCP_RTO_MAX);
        return 0;
}
EXPORT_SYMBOL(tcp_connect);

/* Send out a delayed ack, the caller does the policy checking
 * to see if we should even be here.  See tcp_input.c:tcp_ack_snd_check()
 * for details.
 */
void tcp_send_delayed_ack(struct sock *sk)
{
        struct inet_connection_sock *icsk = inet_csk(sk);
        int ato = icsk->icsk_ack.ato;
        unsigned long timeout;

        if (ato > TCP_DELACK_MIN) {
                const struct tcp_sock *tp = tcp_sk(sk);
                int max_ato = HZ / 2;

                if (icsk->icsk_ack.pingpong ||
                    (icsk->icsk_ack.pending & ICSK_ACK_PUSHED))
                        max_ato = TCP_DELACK_MAX;

                /* Slow path, intersegment interval is "high". */

                /* If some rtt estimate is known, use it to bound delayed ack.
                 * Do not use inet_csk(sk)->icsk_rto here, use results of rtt measurements
                 * directly.
                 */
                if (tp->srtt_us) {
                        int rtt = max_t(int, usecs_to_jiffies(tp->srtt_us >> 3),
                                        TCP_DELACK_MIN);

                        if (rtt < max_ato)
                                max_ato = rtt;
                }

                ato = min(ato, max_ato);
        }

        /* Stay within the limit we were given */
        timeout = jiffies + ato;

        /* Use new timeout only if there wasn't a older one earlier. */
        if (icsk->icsk_ack.pending & ICSK_ACK_TIMER) {
                /* If delack timer was blocked or is about to expire,
                 * send ACK now.
                 */
                if (icsk->icsk_ack.blocked ||
                    time_before_eq(icsk->icsk_ack.timeout, jiffies + (ato >> 2))) {
                        tcp_send_ack(sk);
                        return;
                }

                if (!time_before(timeout, icsk->icsk_ack.timeout))
                        timeout = icsk->icsk_ack.timeout;
        }
        icsk->icsk_ack.pending |= ICSK_ACK_SCHED | ICSK_ACK_TIMER;
        icsk->icsk_ack.timeout = timeout;
        sk_reset_timer(sk, &icsk->icsk_delack_timer, timeout);
}

/* This routine sends an ack and also updates the window. */
void tcp_send_ack(struct sock *sk)
{
        struct sk_buff *buff;

        /* If we have been reset, we may not send again. */
        if (sk->sk_state == TCP_CLOSE)
                return;

        /* We are not putting this on the write queue, so
         * tcp_transmit_skb() will set the ownership to this
         * sock.
         */
        buff = alloc_skb(MAX_TCP_HEADER, sk_gfp_atomic(sk, GFP_ATOMIC));
        if (buff == NULL) {
                inet_csk_schedule_ack(sk);
                inet_csk(sk)->icsk_ack.ato = TCP_ATO_MIN;
                inet_csk_reset_xmit_timer(sk, ICSK_TIME_DACK,
                                          TCP_DELACK_MAX, TCP_RTO_MAX);
                return;
        }

        /* Reserve space for headers and prepare control bits. */
        skb_reserve(buff, MAX_TCP_HEADER);
        tcp_init_nondata_skb(buff, tcp_acceptable_seq(sk), TCPHDR_ACK);

        /* Send it off, this clears delayed acks for us. */
        TCP_SKB_CB(buff)->when = tcp_time_stamp;
        tcp_transmit_skb(sk, buff, 0, sk_gfp_atomic(sk, GFP_ATOMIC));
}

/* This routine sends a packet with an out of date sequence
 * number. It assumes the other end will try to ack it.
 *
 * Question: what should we make while urgent mode?
 * 4.4BSD forces sending single byte of data. We cannot send
 * out of window data, because we have SND.NXT==SND.MAX...
 *
 * Current solution: to send TWO zero-length segments in urgent mode:
 * one is with SEG.SEQ=SND.UNA to deliver urgent pointer, another is
 * out-of-date with SND.UNA-1 to probe window.
 */
static int tcp_xmit_probe_skb(struct sock *sk, int urgent)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct sk_buff *skb;

        /* We don't queue it, tcp_transmit_skb() sets ownership. */
        skb = alloc_skb(MAX_TCP_HEADER, sk_gfp_atomic(sk, GFP_ATOMIC));
        if (skb == NULL)
                return -1;

        /* Reserve space for headers and set control bits. */
        skb_reserve(skb, MAX_TCP_HEADER);
        /* Use a previous sequence.  This should cause the other
         * end to send an ack.  Don't queue or clone SKB, just
         * send it.
         */
        tcp_init_nondata_skb(skb, tp->snd_una - !urgent, TCPHDR_ACK);
        TCP_SKB_CB(skb)->when = tcp_time_stamp;
        return tcp_transmit_skb(sk, skb, 0, GFP_ATOMIC);
}

void tcp_send_window_probe(struct sock *sk)
{
        if (sk->sk_state == TCP_ESTABLISHED) {
                tcp_sk(sk)->snd_wl1 = tcp_sk(sk)->rcv_nxt - 1;
                tcp_xmit_probe_skb(sk, 0);
        }
}

/* Initiate keepalive or window probe from timer. */
int tcp_write_wakeup(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct sk_buff *skb;

        if (sk->sk_state == TCP_CLOSE)
                return -1;

        if ((skb = tcp_send_head(sk)) != NULL &&
            before(TCP_SKB_CB(skb)->seq, tcp_wnd_end(tp))) {
                int err;
                unsigned int mss = tcp_current_mss(sk);
                unsigned int seg_size = tcp_wnd_end(tp) - TCP_SKB_CB(skb)->seq;

                if (before(tp->pushed_seq, TCP_SKB_CB(skb)->end_seq))
                        tp->pushed_seq = TCP_SKB_CB(skb)->end_seq;

                /* We are probing the opening of a window
                 * but the window size is != 0
                 * must have been a result SWS avoidance ( sender )
                 */
                if (seg_size < TCP_SKB_CB(skb)->end_seq - TCP_SKB_CB(skb)->seq ||
                    skb->len > mss) {
                        seg_size = min(seg_size, mss);
                        TCP_SKB_CB(skb)->tcp_flags |= TCPHDR_PSH;
                        if (tcp_fragment(sk, skb, seg_size, mss, GFP_ATOMIC))
                                return -1;
                } else if (!tcp_skb_pcount(skb))
                        tcp_set_skb_tso_segs(sk, skb, mss);

                TCP_SKB_CB(skb)->tcp_flags |= TCPHDR_PSH;
                TCP_SKB_CB(skb)->when = tcp_time_stamp;
                err = tcp_transmit_skb(sk, skb, 1, GFP_ATOMIC);
                if (!err)
                        tcp_event_new_data_sent(sk, skb);
                return err;
        } else {
                if (between(tp->snd_up, tp->snd_una + 1, tp->snd_una + 0xFFFF))
                        tcp_xmit_probe_skb(sk, 1);
                return tcp_xmit_probe_skb(sk, 0);
        }
}

/* A window probe timeout has occurred.  If window is not closed send
 * a partial packet else a zero probe.
 */
void tcp_send_probe0(struct sock *sk)
{
        struct inet_connection_sock *icsk = inet_csk(sk);
        struct tcp_sock *tp = tcp_sk(sk);
        int err;

        err = tcp_write_wakeup(sk);

        if (tp->packets_out || !tcp_send_head(sk)) {
                /* Cancel probe timer, if it is not required. */
                icsk->icsk_probes_out = 0;
                icsk->icsk_backoff = 0;
                return;
        }

        if (err <= 0) {
                if (icsk->icsk_backoff < sysctl_tcp_retries2)
                        icsk->icsk_backoff++;
                icsk->icsk_probes_out++;
                inet_csk_reset_xmit_timer(sk, ICSK_TIME_PROBE0,
                                          min(icsk->icsk_rto << icsk->icsk_backoff, TCP_RTO_MAX),
                                          TCP_RTO_MAX);
        } else {
                /* If packet was not sent due to local congestion,
                 * do not backoff and do not remember icsk_probes_out.
                 * Let local senders to fight for local resources.
                 *
                 * Use accumulated backoff yet.
                 */
                if (!icsk->icsk_probes_out)
                        icsk->icsk_probes_out = 1;
                inet_csk_reset_xmit_timer(sk, ICSK_TIME_PROBE0,
                                          min(icsk->icsk_rto << icsk->icsk_backoff,
                                              TCP_RESOURCE_PROBE_INTERVAL),
                                          TCP_RTO_MAX);
        }
}

int tcp_rtx_synack(struct sock *sk, struct request_sock *req)
{
        const struct tcp_request_sock_ops *af_ops = tcp_rsk(req)->af_specific;
        struct flowi fl;
        int res;

        res = af_ops->send_synack(sk, NULL, &fl, req, 0, NULL);
        if (!res) {
                TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_RETRANSSEGS);
                NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPSYNRETRANS);
        }
        return res;
}
EXPORT_SYMBOL(tcp_rtx_synack);

```

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