```
**********************************************************
   21837 Thu Nov  6 17:28:17 2014
new/usr/src/uts/common/inet/tcp.h
5295 remove maxburst logic from TCP's send algorithm Reviewed by: Dan McDonald <
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2011, Joyent, Inc. All rights reserved.
 * Copyright (c) 2011 Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2014 by Delphix. All rights reserved.
 */
/* Copyright (c) 1990 Mentat Inc. */

#ifndef _INET_TCP_H
#define _INET_TCP_H

#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/inttypes.h>
#include <netinet/ip6.h>
#include <netinet/tcp.h>
#include <sys/socket.h>
#include <sys/socket_proto.h>
#include <sys/md5.h>
#include <inet/common.h>
#include <inet/ip.h>
#include <inet/ip6.h>
#include <inet/mi.h>
#include <inet/mib2.h>
#include <inet/tcp_stack.h>
#include <inet/tcp_sack.h>

/* TCP states */
#define TCPS_CLOSED             -6
#define TCPS_IDLE               -5      /* idle (opened, but not bound) */
#define TCPS_BOUND              -4      /* bound, ready to connect or accept */
#define TCPS_LISTEN             -3      /* listening for connection */
#define TCPS_SYN_SENT           -2      /* active, have sent syn */
#define TCPS_SYN_RCVD           -1      /* have received syn (and sent ours) */
/* states < TCPS_ESTABLISHED are those where connections not established */
#define TCPS_ESTABLISHED        0       /* established */
#define TCPS_CLOSE_WAIT         1       /* rcvd fin, waiting for close */
/* states > TCPS_CLOSE_WAIT are those where user has closed */
#define TCPS_FIN_WAIT_1         2       /* have closed and sent fin */
```



```
#define TCPS_CLOSING            3       /* closed, xchd FIN, await FIN ACK */
#define TCPS_LAST_ACK           4       /* had fin and close; await FIN ACK */
/* states > TCPS_CLOSE_WAIT && < TCPS_FIN_WAIT_2 await ACK of FIN */
#define TCPS_FIN_WAIT_2         5       /* have closed, fin is acked */
#define TCPS_TIME_WAIT          6       /* in 2*msl quiet wait after close */

/*
 * Internal flags used in conjunction with the packet header flags.
 * Used in tcp_input_data to keep track of what needs to be done.
 */
#define TH_LIMIT_XMIT           0x0400  /* Limited xmit is needed */
#define TH_XMIT_NEEDED          0x0800  /* Window opened - send queued data */
#define TH_REXMIT_NEEDED        0x1000  /* Time expired for unacked data */
#define TH_ACK_NEEDED           0x2000  /* Send an ack now. */
#define TH_NEED_SACK_REXMIT     0x4000  /* Use SACK info to retransmission */
#define TH_ACK_TIMER_NEEDED     0x8000  /* Start the delayed ACK timer */
#define TH_ORDREL_NEEDED        0x10000 /* Generate an ordrel indication */
#define TH_MARKNEXT_NEEDED      0x20000 /* Data should have MSGMARKNEXT */
#define TH_SEND_URP_MARK        0x40000 /* Send up tcp_urp_mark_mp */

/*
 * TCP sequence numbers are 32 bit integers operated
 * on with modular arithmetic.  These macros can be
 * used to compare such integers.
 */
#define SEQ_LT(a, b)    ((int32_t)((a)-(b)) < 0)
#define SEQ_LEQ(a, b)   ((int32_t)((a)-(b)) <= 0)
#define SEQ_GT(a, b)    ((int32_t)((a)-(b)) > 0)
#define SEQ_GEQ(a, b)   ((int32_t)((a)-(b)) >= 0)

/* TCP Protocol header */
typedef struct tcphdr_s {
        uint8_t         th_lport[2];    /* Source port */
        uint8_t         th_fport[2];    /* Destination port */
        uint8_t         th_seq[4];      /* Sequence number */
        uint8_t         th_ack[4];      /* Acknowledgement number */
        uint8_t         th_offset_and_rsrvd[1]; /* Offset to the packet data */
        uint8_t         th_flags[1];
        uint8_t         th_win[2];      /* Allocation number */
        uint8_t         th_sum[2];      /* TCP checksum */
        uint8_t         th_urp[2];      /* Urgent pointer */
} tcph_t;
```

*unchanged_portion_omitted_*

```
struct conn_s;
struct tcp_listen_cnt_s;

/*
 * Control structure for each open TCP stream,
 * defined only within the kernel or for a kmem user.
 * NOTE: tcp_reinit_values MUST have a line for each field in this structure!
 */
#if (defined(_KERNEL) || defined(_KMEMUSER))

typedef struct tcp_s {
        struct tcp_s    *tcp_time_wait_next;
                                /* Pointer to next T/W block */
        struct tcp_s    *tcp_time_wait_prev;
                                /* Pointer to previous T/W next */
        int64_t         tcp_time_wait_expire;

        struct conn_s   *tcp_connp;     /* back pointer to conn_t */
        tcp_stack_t     *tcp_tcps;      /* back pointer to tcp_stack_t */

        int32_t tcp_state;
        int32_t tcp_rcv_ws;             /* My window scale power */
```

```
157         int32_t tcp_snd_ws;             /* Sender's window scale power */
158         uint32_t tcp_ts_recent;         /* Timestamp of earliest unacked */
159                                         /*  data segment */
160         clock_t tcp_rto;                /* Round trip timeout */
161         int64_t tcp_last_rcv_lbolt;
162                                 /* lbolt on last packet, used for PAWS */
163         uint32_t tcp_rto_initial;       /* Initial RTO */
164         uint32_t tcp_rto_min;           /* Minimum RTO */
165         uint32_t tcp_rto_max;           /* Maximum RTO */

167         uint32_t tcp_snxt;              /* Senders next seq num */
168         uint32_t tcp_swnd;              /* Senders window (relative to suna) */
169         uint32_t tcp_mss;               /* Max segment size */
170         uint32_t tcp_iss;               /* Initial send seq num */
171         uint32_t tcp_rnxt;              /* Seq we expect to recv next */
172         uint32_t tcp_rwnd;

174         /* Fields arranged in approximate access order along main paths */
175         mblk_t  *tcp_xmit_head;         /* Head of xmit/rexmit list */
176         mblk_t  *tcp_xmit_last;         /* Last valid data seen by tcp_wput */
177         mblk_t  *tcp_xmit_tail;         /* Last data sent */
178         uint32_t tcp_unsent;            /* # of bytes in hand that are unsent */
179         uint32_t tcp_xmit_tail_unsent;  /* # of unsent bytes in xmit_tail */

181         uint32_t tcp_suna;              /* Sender unacknowledged */
182         uint32_t tcp_rexmit_nxt;        /* Next rexmit seq num */
183         uint32_t tcp_rexmit_max;        /* Max retran seq num */
183         int32_t tcp_snd_burst;          /* Send burst factor */
184         uint32_t tcp_cwnd;              /* Congestion window */
185         int32_t tcp_cwnd_cnt;           /* cwnd cnt in congestion avoidance */

187         uint32_t tcp_ibsegs;            /* Inbound segments on this stream */
188         uint32_t tcp_obsegs;            /* Outbound segments on this stream */

190         uint32_t tcp_naglim;            /* Tunable nagle limit */
191         uint32_t        tcp_valid_bits;
192 #define TCP_ISS_VALID   0x1     /* Is the tcp_iss seq num active? */
193 #define TCP_FSS_VALID   0x2     /* Is the tcp_fss seq num active? */
194 #define TCP_URG_VALID   0x4     /* Is the tcp_urg seq num active? */
195 #define TCP_OFO_FIN_VALID 0x8   /* Has TCP received an out of order FIN? */



199         timeout_id_t    tcp_timer_tid;  /* Control block for timer service */
200         uchar_t tcp_timer_backoff;      /* Backoff shift count. */
201         int64_t tcp_last_recv_time;     /* Last time we receive a segment. */
202         uint32_t tcp_init_cwnd;         /* Initial cwnd (start/restart) */

204         /* Following manipulated by TCP under squeue protection */
205         uint32_t
206                 tcp_urp_last_valid : 1, /* Is tcp_urp_last valid? */
207                 tcp_hard_binding : 1,   /* TCP_DETACHED_NONEAGER */
208                 tcp_fin_acked : 1,      /* Has our FIN been acked? */
209                 tcp_fin_rcvd : 1,       /* Have we seen a FIN? */

211                 tcp_fin_sent : 1,       /* Have we sent our FIN yet? */
212                 tcp_ordrel_done : 1,    /* Have we sent the ord_rel upstream? */
213                 tcp_detached : 1,       /* If we're detached from a stream */
214                 tcp_zero_win_probe: 1,  /* Zero win probing is in progress */

216                 tcp_loopback: 1,        /* src and dst are the same machine */
217                 tcp_localnet: 1,        /* src and dst are on the same subnet */
218                 tcp_syn_defense: 1,     /* For defense against SYN attack */
219 #define tcp_dontdrop    tcp_syn_defense
220                 tcp_set_timer : 1,
```



```
222                 tcp_active_open: 1,     /* This is a active open */
223                 tcp_rexmit : 1,         /* TCP is retransmitting */
224                 tcp_snd_sack_ok : 1,    /* Can use SACK for this connection */
225                 tcp_hwcksum : 1,        /* The NIC is capable of hwcksum */

227                 tcp_ip_forward_progress : 1,
228                 tcp_ecn_ok : 1,         /* Can use ECN for this connection */
229                 tcp_ecn_echo_on : 1,    /* Need to do ECN echo */
230                 tcp_ecn_cwr_sent : 1,   /* ECN_CWR has been sent */

232                 tcp_cwr : 1,            /* Cwnd has reduced recently */

234                 tcp_pad_to_bit31 : 11;

236         /* Following manipulated by TCP under squeue protection */
237         uint32_t
238                 tcp_snd_ts_ok  : 1,
239                 tcp_snd_ws_ok  : 1,
240                 tcp_reserved_port : 1,
241                 tcp_in_free_list : 1,

243                 tcp_snd_zcopy_on : 1,   /* xmit zero-copy enabled */
244                 tcp_snd_zcopy_aware : 1, /* client is zero-copy aware */
245                 tcp_xmit_zc_clean : 1,  /* the xmit list is free of zc-mblk */
246                 tcp_wait_for_eagers : 1, /* Wait for eagers to disappear */

248                 tcp_accept_error : 1,   /* Error during TLI accept */
249                 tcp_send_discon_ind : 1, /* TLI accept err, send discon ind */
250                 tcp_cork : 1,           /* tcp_cork option */
251                 tcp_tconnind_started : 1, /* conn_ind message is being sent */

253                 tcp_lso :1,             /* Lower layer is capable of LSO */
254                 tcp_is_wnd_shrnk : 1,   /* Window has shrunk */

256                 tcp_pad_to_bit_31 : 18;

258         uint32_t        tcp_initial_pmtu; /* Initial outgoing Path MTU. */

260         mblk_t  *tcp_reass_head;        /* Out of order reassembly list head */
261         mblk_t  *tcp_reass_tail;        /* Out of order reassembly list tail */

263         /* SACK related info */
264         tcp_sack_info_t tcp_sack_info;

266 #define tcp_pipe                tcp_sack_info.tcp_pipe
267 #define tcp_fack                tcp_sack_info.tcp_fack
268 #define tcp_sack_snxt           tcp_sack_info.tcp_sack_snxt
269 #define tcp_max_sack_blk        tcp_sack_info.tcp_max_sack_blk
270 #define tcp_num_sack_blk        tcp_sack_info.tcp_num_sack_blk
271 #define tcp_sack_list           tcp_sack_info.tcp_sack_list
272 #define tcp_num_notsack_blk     tcp_sack_info.tcp_num_notsack_blk
273 #define tcp_cnt_notsack_list    tcp_sack_info.tcp_cnt_notsack_list
274 #define tcp_notsack_list        tcp_sack_info.tcp_notsack_list

276         mblk_t  *tcp_rcv_list;          /* Queued until push, urgent data, */
277         mblk_t  *tcp_rcv_last_head;     /* optdata, or the count exceeds */
278         mblk_t  *tcp_rcv_last_tail;     /* tcp_rcv_push_wait. */
279         uint32_t tcp_rcv_cnt;           /* tcp_rcv_list is b_next chain. */

281         uint32_t tcp_cwnd_ssthresh;     /* Congestion window */
282         uint32_t tcp_cwnd_max;
283         uint32_t tcp_csuna;             /* Clear (no rexmits in window) suna */

285         clock_t tcp_rtt_sa;             /* Round trip smoothed average */
286         clock_t tcp_rtt_sd;             /* Round trip smoothed deviation */
287         clock_t tcp_rtt_update;         /* Round trip update(s) */
```

```
288         clock_t tcp_ms_we_have_waited;  /* Total retrans time */

290         uint32_t tcp_swl1;              /* These help us avoid using stale */
291         uint32_t tcp_swl2;              /*  packets to update state */

293         uint32_t tcp_rack;              /* Seq # we have acked */
294         uint32_t tcp_rack_cnt;          /* # of segs we have deferred ack */
295         uint32_t tcp_rack_cur_max;      /* # of segs we may defer ack for now */
296         uint32_t tcp_rack_abs_max;      /* # of segs we may defer ack ever */
297         timeout_id_t    tcp_ack_tid;    /* Delayed ACK timer ID */
298         timeout_id_t    tcp_push_tid;   /* Push timer ID */

300         uint32_t tcp_max_swnd;          /* Maximum swnd we have seen */

302         struct tcp_s *tcp_listener;     /* Our listener */

304         uint32_t tcp_irs;               /* Initial recv seq num */
305         uint32_t tcp_fss;               /* Final/fin send seq num */
306         uint32_t tcp_urg;               /* Urgent data seq num */

308         clock_t tcp_first_timer_threshold;  /* When to prod IP */
309         clock_t tcp_second_timer_threshold; /* When to give up completely */
310         clock_t tcp_first_ctimer_threshold; /* 1st threshold while connecting */
311         clock_t tcp_second_ctimer_threshold; /* 2nd ... while connecting */

313         uint32_t tcp_urp_last;          /* Last urp for which signal sent */
314         mblk_t  *tcp_urp_mp;            /* T_EXDATA_IND for urgent byte */
315         mblk_t  *tcp_urp_mark_mp;       /* zero-length marked/unmarked msg */

317         int tcp_conn_req_cnt_q0;        /* # of conn reqs in SYN_RCVD */
318         int tcp_conn_req_cnt_q; /* # of conn reqs in ESTABLISHED */
319         int tcp_conn_req_max;   /* # of ESTABLISHED conn reqs allowed */
320         t_scalar_t tcp_conn_req_seqnum; /* Incrementing pending conn req ID */
321 #define tcp_ip_addr_cache       tcp_reass_tail
322                                 /* Cache ip addresses that */
323                                 /* complete the 3-way handshake */
324         kmutex_t  tcp_eager_lock;
325         struct tcp_s *tcp_eager_next_q; /* next eager in ESTABLISHED state */
326         struct tcp_s *tcp_eager_last_q; /* last eager in ESTABLISHED state */
327         struct tcp_s *tcp_eager_next_q0; /* next eager in SYN_RCVD state */
328         struct tcp_s *tcp_eager_prev_q0; /* prev eager in SYN_RCVD state */
329                                 /* all eagers form a circular list */
330         boolean_t tcp_conn_def_q0;      /* move from q0 to q deferred */

332         union {
333             mblk_t *tcp_eager_conn_ind; /* T_CONN_IND waiting for 3rd ack. */
334             mblk_t *tcp_opts_conn_req; /* T_CONN_REQ w/ options processed */
335         } tcp_conn;
336         uint32_t tcp_syn_rcvd_timeout;  /* How many SYN_RCVD timeout in q0 */

338         /*
339          * TCP Keepalive Timer members.
340          * All keepalive timer intervals are in milliseconds.
341          */
342         int32_t tcp_ka_last_intrvl;     /* Last probe interval */
343         timeout_id_t tcp_ka_tid;        /* Keepalive timer ID */
344         uint32_t tcp_ka_interval;       /* Keepalive interval */

346         /*
347          * TCP connection is terminated if we don't hear back from the peer
348          * for tcp_ka_abort_thres milliseconds after the first keepalive probe.
349          * tcp_ka_rinterval is the interval in milliseconds between successive
350          * keepalive probes. tcp_ka_cnt is the number of keepalive probes to
351          * be sent before terminating the connection, if we don't hear back from
352          * peer.
353          * tcp_ka_abort_thres = tcp_ka_rinterval * tcp_ka_cnt
```



```
354          */
355         uint32_t tcp_ka_rinterval;      /* keepalive retransmit interval */
356         uint32_t tcp_ka_abort_thres;    /* Keepalive abort threshold */
357         uint32_t tcp_ka_cnt;            /* count of keepalive probes */

359         int32_t tcp_client_errno;       /* How the client screwed up */

361         /*
362          * The header template lives in conn_ht_iphc allocated by tcp_build_hdrs
363          * We maintain three pointers into conn_ht_iphc.
364          */
365         ipha_t  *tcp_ipha;              /* IPv4 header in conn_ht_iphc */
366         ip6_t   *tcp_ip6h;              /* IPv6 header in conn_ht_iphc */
367         tcpha_t *tcp_tcpha;             /* TCP header in conn_ht_iphc */

369         uint16_t tcp_last_sent_len;     /* Record length for nagle */
370         uint16_t tcp_last_recv_len;     /* Used by DTrace */
371         uint16_t tcp_dupack_cnt;        /* # of consequtive duplicate acks */

373         kmutex_t        *tcp_acceptor_lockp;    /* Ptr to tf_lock */

375         mblk_t          *tcp_ordrel_mp;         /* T_ordrel_ind mblk */
376         t_uscalar_t     tcp_acceptor_id;        /* ACCEPTOR_id */

378         int             tcp_ipsec_overhead;

380         uint_t          tcp_recvifindex; /* Last received IPV6_RCVPKTINFO */
381         uint_t          tcp_recvhops;   /* Last received IPV6_RECVHOPLIMIT */
382         uint_t          tcp_recvtclass; /* Last received IPV6_RECVTCLASS */
383         ip6_hbh_t       *tcp_hopopts;   /* Last received IPV6_RECVHOPOPTS */
384         ip6_dest_t      *tcp_dstopts;   /* Last received IPV6_RECVDSTOPTS */
385         ip6_dest_t      *tcp_rthdrdstopts; /* Last recv IPV6_RECVRTHDRDSTOPTS */
386         ip6_rthdr_t     *tcp_rthdr;     /* Last received IPV6_RECVRTHDR */
387         uint_t          tcp_hopoptslen;
388         uint_t          tcp_dstoptslen;
389         uint_t          tcp_rthdrdstoptslen;
390         uint_t          tcp_rthdrlen;

392         mblk_t          *tcp_timercache;

394         kmutex_t        tcp_closelock;
395         kcondvar_t      tcp_closecv;
396         uint8_t         tcp_closed;
397         uint8_t         tcp_closeflags;
398         mblk_t          tcp_closemp;
399         timeout_id_t    tcp_linger_tid; /* Linger timer ID */

401         struct tcp_s *tcp_acceptor_hash; /* Acceptor hash chain */
402         struct tcp_s **tcp_ptpahn; /* Pointer to previous accept hash next. */
403         struct tcp_s *tcp_bind_hash; /* Bind hash chain */
404         struct tcp_s *tcp_bind_hash_port; /* tcp_t's bound to the same lport */
405         struct tcp_s **tcp_ptpbhn;

407         uint_t          tcp_maxpsz_multiplier;

409         uint32_t        tcp_lso_max; /* maximum LSO payload */

411         uint32_t        tcp_ofo_fin_seq; /* Recv out of order FIN seq num */
412         uint32_t        tcp_cwr_snd_max;

414         struct tcp_s *tcp_saved_listener;       /* saved value of listener */

416         uint32_t        tcp_in_ack_unsent;      /* ACK for unsent data cnt. */

418         /*
419          * All fusion-related fields are protected by squeue.
```

```
420          */
421         struct tcp_s *tcp_loopback_peer;        /* peer tcp for loopback */
422         mblk_t  *tcp_fused_sigurg_mp;           /* M_PCSIG mblk for SIGURG */

424         uint32_t
425                 tcp_fused : 1,          /* loopback tcp in fusion mode */
426                 tcp_unfusable : 1,      /* fusion not allowed on endpoint */
427                 tcp_fused_sigurg : 1,   /* send SIGURG upon draining */

429                 tcp_fuse_to_bit_31 : 29;

431         kmutex_t tcp_non_sq_lock;

433         /*
434          * This variable is accessed without any lock protection
435          * and therefore must not be declared as a bit field along
436          * with the rest which require such condition.
437          */
438         boolean_t       tcp_issocket;   /* this is a socket tcp */

440         /* protected by the tcp_non_sq_lock lock */
441         uint32_t        tcp_squeue_bytes;

443         /*
444          * tcp_closemp_used is protected by listener's tcp_eager_lock
445          * when used for eagers. When used for a tcp in TIME_WAIT state
446          * or in tcp_close(), it is not protected by any lock as we
447          * do not expect any other thread to use it concurrently.
448          * We do allow re-use of tcp_closemp in tcp_time_wait_collector()
449          * and tcp_close() but not concurrently.
450          */
451         boolean_t tcp_closemp_used;

453         /*
454          * previous and next eagers in the list of droppable eagers. See
455          * the comments before MAKE_DROPPABLE(). These pointers are
456          * protected by listener's tcp_eager_lock.
457          */
458         struct tcp_s    *tcp_eager_prev_drop_q0;
459         struct tcp_s    *tcp_eager_next_drop_q0;

461         /*
462          * Have we flow controlled xmitter?
463          * This variable can be modified outside the squeue and hence must
464          * not be declared as a bit field along with the rest that are
465          * modified only within the squeue.
466          * protected by the tcp_non_sq_lock lock.
467          */
468         boolean_t       tcp_flow_stopped;

470         /*
471          * Sender's next sequence number at the time the window was shrunk.
472          */
473         uint32_t        tcp_snxt_shrunk;

475         /*
476          * Socket generation number which is bumped when a connection attempt
477          * is initiated. Its main purpose is to ensure that the socket does not
478          * miss the asynchronous connected/disconnected notification.
479          */
480         sock_connid_t   tcp_connid;

482         /* mblk_t used to enter TCP's squeue from the service routine. */
483         mblk_t          *tcp_rsrv_mp;
484         /* Mutex for accessing tcp_rsrv_mp */
485         kmutex_t        tcp_rsrv_mp_lock;
```



```
487         /* For connection counting. */
488         struct tcp_listen_cnt_s *tcp_listen_cnt;

490         /* Segment reassembly timer. */
491         timeout_id_t            tcp_reass_tid;

493         /* FIN-WAIT-2 flush timeout */
494         uint32_t                tcp_fin_wait_2_flush_interval;

496 #ifdef DEBUG
497         pc_t                    tcmp_stk[15];
498 #endif
499 } tcp_t;
```

unchanged_portion_omitted_

```
**********************************************************
  130083 Thu Nov  6 17:28:18 2014
new/usr/src/uts/common/inet/tcp/tcp.c
5295 remove maxburst logic from TCP's send algorithm Reviewed by: Dan McDonald <
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2011, Joyent Inc. All rights reserved.
 * Copyright (c) 2011 Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2013,2014 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright 2014, OmniTI Computer Consulting, Inc. All rights reserved.
 */
/* Copyright (c) 1990 Mentat Inc. */

#include <sys/types.h>
#include <sys/stream.h>
#include <sys/strsun.h>
#include <sys/strsubr.h>
#include <sys/stropts.h>
#include <sys/strlog.h>
#define _SUN_TPI_VERSION 2
#include <sys/tihdr.h>
#include <sys/timod.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/suntpi.h>
#include <sys/xti_inet.h>
#include <sys/cmn_err.h>
#include <sys/debug.h>
#include <sys/sdt.h>
#include <sys/vtrace.h>
#include <sys/kmem.h>
#include <sys/ethernet.h>
#include <sys/cpuvar.h>
#include <sys/dlpi.h>
#include <sys/pattr.h>
#include <sys/policy.h>
#include <sys/priv.h>
#include <sys/zone.h>
#include <sys/sunldi.h>

#include <sys/errno.h>
#include <sys/signal.h>
#include <sys/socket.h>
```



```
#include <sys/socketvar.h>
#include <sys/sockio.h>
#include <sys/isa_defs.h>
#include <sys/md5.h>
#include <sys/random.h>
#include <sys/uio.h>
#include <sys/systm.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <netinet/ip6.h>
#include <netinet/icmp6.h>
#include <net/if.h>
#include <net/route.h>
#include <inet/ipsec_impl.h>

#include <inet/common.h>
#include <inet/ip.h>
#include <inet/ip_impl.h>
#include <inet/ip6.h>
#include <inet/ip_ndp.h>
#include <inet/proto_set.h>
#include <inet/mib2.h>
#include <inet/optcom.h>
#include <inet/snmpcom.h>
#include <inet/kstatcom.h>
#include <inet/tcp.h>
#include <inet/tcp_impl.h>
#include <inet/tcp_cluster.h>
#include <inet/udp_impl.h>
#include <net/pfkeyv2.h>
#include <inet/ipdrop.h>

#include <inet/ipclassifier.h>
#include <inet/ip_ire.h>
#include <inet/ip_ftable.h>
#include <inet/ip_if.h>
#include <inet/ipp_common.h>
#include <inet/ip_rts.h>
#include <inet/ip_netinfo.h>
#include <sys/squeue_impl.h>
#include <sys/squeue.h>
#include <sys/tsol/label.h>
#include <sys/tsol/tnet.h>
#include <rpc/pmap_prot.h>
#include <sys/callo.h>

/*
 * TCP Notes: aka FireEngine Phase I (PSARC 2002/433)
 *
 * (Read the detailed design doc in PSARC case directory)
 *
 * The entire tcp state is contained in tcp_t and conn_t structure
 * which are allocated in tandem using ipcl_conn_create() and passing
 * IPCL_TCPCONN as a flag. We use 'conn_ref' and 'conn_lock' to protect
 * the references on the tcp_t. The tcp_t structure is never compressed
 * and packets always land on the correct TCP perimeter from the time
 * eager is created till the time tcp_t dies (as such the old mentat
 * TCP global queue is not used for detached state and no IPSEC checking
 * is required). The global queue is still allocated to send out resets
 * for connection which have no listeners and IP directly calls
 * tcp_xmit_listeners_reset() which does any policy check.
 *
 * Protection and Synchronisation mechanism:
 *
 * The tcp data structure does not use any kind of lock for protecting
 * its state but instead uses 'squeues' for mutual exclusion from various
```

```
 127  * read and write side threads. To access a tcp member, the thread should
 128  * always be behind squeue (via squeue_enter with flags as SQ_FILL, SQ_PROCESS,
 129  * or SQ_NODRAIN). Since the squeues allow a direct function call, caller
 130  * can pass any tcp function having prototype of edesc_t as argument
 131  * (different from traditional STREAMs model where packets come in only
 132  * designated entry points). The list of functions that can be directly
 133  * called via squeue are listed before the usual function prototype.
 134  *
 135  * Referencing:
 136  *
 137  * TCP is MT-Hot and we use a reference based scheme to make sure that the
 138  * tcp structure doesn't disappear when its needed. When the application
 139  * creates an outgoing connection or accepts an incoming connection, we
 140  * start out with 2 references on 'conn_ref'. One for TCP and one for IP.
 141  * The IP reference is just a symbolic reference since ip_tcpclose()
 142  * looks at tcp structure after tcp_close_output() returns which could
 143  * have dropped the last TCP reference. So as long as the connection is
 144  * in attached state i.e. !TCP_IS_DETACHED, we have 2 references on the
 145  * conn_t. The classifier puts its own reference when the connection is
 146  * inserted in listen or connected hash. Anytime a thread needs to enter
 147  * the tcp connection perimeter, it retrieves the conn/tcp from q->ptr
 148  * on write side or by doing a classify on read side and then puts a
 149  * reference on the conn before doing squeue_enter/tryenter/fill. For
 150  * read side, the classifier itself puts the reference under fanout lock
 151  * to make sure that tcp can't disappear before it gets processed. The
 152  * squeue will drop this reference automatically so the called function
 153  * doesn't have to do a DEC_REF.
 154  *
 155  * Opening a new connection:
 156  *
 157  * The outgoing connection open is pretty simple. tcp_open() does the
 158  * work in creating the conn/tcp structure and initializing it. The
 159  * squeue assignment is done based on the CPU the application
 160  * is running on. So for outbound connections, processing is always done
 161  * on application CPU which might be different from the incoming CPU
 162  * being interrupted by the NIC. An optimal way would be to figure out
 163  * the NIC <-> CPU binding at listen time, and assign the outgoing
 164  * connection to the squeue attached to the CPU that will be interrupted
 165  * for incoming packets (we know the NIC based on the bind IP address).
 166  * This might seem like a problem if more data is going out but the
 167  * fact is that in most cases the transmit is ACK driven transmit where
 168  * the outgoing data normally sits on TCP's xmit queue waiting to be
 169  * transmitted.
 170  *
 171  * Accepting a connection:
 172  *
 173  * This is a more interesting case because of various races involved in
 174  * establishing a eager in its own perimeter. Read the meta comment on
 175  * top of tcp_input_listener(). But briefly, the squeue is picked by
 176  * ip_fanout based on the ring or the sender (if loopback).
 177  *
 178  * Closing a connection:
 179  *
 180  * The close is fairly straight forward. tcp_close() calls tcp_close_output()
 181  * via squeue to do the close and mark the tcp as detached if the connection
 182  * was in state TCPS_ESTABLISHED or greater. In the later case, TCP keep its
 183  * reference but tcp_close() drop IP's reference always. So if tcp was
 184  * not killed, it is sitting in time_wait list with 2 reference - 1 for TCP
 185  * and 1 because it is in classifier's connected hash. This is the condition
 186  * we use to determine that its OK to clean up the tcp outside of squeue
 187  * when time wait expires (check the ref under fanout and conn_lock and
 188  * if it is 2, remove it from fanout hash and kill it).
 189  *
 190  * Although close just drops the necessary references and marks the
 191  * tcp_detached state, tcp_close needs to know the tcp_detached has been
 192  * set (under squeue) before letting the STREAM go away (because a
```



```
 193  * inbound packet might attempt to go up the STREAM while the close
 194  * has happened and tcp_detached is not set). So a special lock and
 195  * flag is used along with a condition variable (tcp_closelock, tcp_closed,
 196  * and tcp_closecv) to signal tcp_close that tcp_close_out() has marked
 197  * tcp_detached.
 198  *
 199  * Special provisions and fast paths:
 200  *
 201  * We make special provisions for sockfs by marking tcp_issocket
 202  * whenever we have only sockfs on top of TCP. This allows us to skip
 203  * putting the tcp in acceptor hash since a sockfs listener can never
 204  * become acceptor and also avoid allocating a tcp_t for acceptor STREAM
 205  * since eager has already been allocated and the accept now happens
 206  * on acceptor STREAM. There is a big blob of comment on top of
 207  * tcp_input_listener explaining the new accept. When socket is POP'd,
 208  * sockfs sends us an ioctl to mark the fact and we go back to old
 209  * behaviour. Once tcp_issocket is unset, its never set for the
 210  * life of that connection.
 211  *
 212  * IPsec notes :
 213  *
 214  * Since a packet is always executed on the correct TCP perimeter
 215  * all IPsec processing is defered to IP including checking new
 216  * connections and setting IPSEC policies for new connection. The
 217  * only exception is tcp_xmit_listeners_reset() which is called
 218  * directly from IP and needs to policy check to see if TH_RST
 219  * can be sent out.
 220  */

 222 /*
 223  * Values for squeue switch:
 224  * 1: SQ_NODRAIN
 225  * 2: SQ_PROCESS
 226  * 3: SQ_FILL
 227  */
 228 int tcp_squeue_wput = 2;        /* /etc/systems */
 229 int tcp_squeue_flag;

 231 /*
 232  * To prevent memory hog, limit the number of entries in tcp_free_list
 233  * to 1% of available memory / number of cpus
 234  */
 235 uint_t tcp_free_list_max_cnt = 0;

 237 #define TIDUSZ  4096    /* transport interface data unit size */

 239 /*
 240  * Size of acceptor hash list.  It has to be a power of 2 for hashing.
 241  */
 242 #define TCP_ACCEPTOR_FANOUT_SIZE                512

 244 #ifdef  _ILP32
 245 #define TCP_ACCEPTOR_HASH(accid)                                        \
 246                 (((uint_t)(accid) >> 8) & (TCP_ACCEPTOR_FANOUT_SIZE - 1))
 247 #else
 248 #define TCP_ACCEPTOR_HASH(accid)                                        \
 249                 ((uint_t)(accid) & (TCP_ACCEPTOR_FANOUT_SIZE - 1))
 250 #endif  /* _ILP32 */

 252 /*
 253  * Minimum number of connections which can be created per listener.  Used
 254  * when the listener connection count is in effect.
 255  */
 256 static uint32_t tcp_min_conn_listener = 2;

 258 uint32_t tcp_early_abort = 30;
```

```
 260 /* TCP Timer control structure */
 261 typedef struct tcpt_s {
 262         pfv_t   tcpt_pfv;       /* The routine we are to call */
 263         tcp_t   *tcpt_tcp;      /* The parameter we are to pass in */
 264 } tcpt_t;
______unchanged_portion_omitted_

2329 /*
2330  * Initialize the various fields in tcp_t.  If parent (the listener) is non
2331  * NULL, certain values will be inheritted from it.
2332  */
2333 void
2334 tcp_init_values(tcp_t *tcp, tcp_t *parent)
2335 {
2336         tcp_stack_t     *tcps = tcp->tcp_tcps;
2337         conn_t          *connp = tcp->tcp_connp;
2338         clock_t         rto;

2340         ASSERT((connp->conn_family == AF_INET &&
2341             connp->conn_ipversion == IPV4_VERSION) ||
2342             (connp->conn_family == AF_INET6 &&
2343             (connp->conn_ipversion == IPV4_VERSION ||
2344             connp->conn_ipversion == IPV6_VERSION)));

2346         if (parent == NULL) {
2347                 tcp->tcp_naglim = tcps->tcps_naglim_def;

2349                 tcp->tcp_rto_initial = tcps->tcps_rexmit_interval_initial;
2350                 tcp->tcp_rto_min = tcps->tcps_rexmit_interval_min;
2351                 tcp->tcp_rto_max = tcps->tcps_rexmit_interval_max;

2353                 tcp->tcp_first_ctimer_threshold =
2354                     tcps->tcps_ip_notify_cinterval;
2355                 tcp->tcp_second_ctimer_threshold =
2356                     tcps->tcps_ip_abort_cinterval;
2357                 tcp->tcp_first_timer_threshold = tcps->tcps_ip_notify_interval;
2358                 tcp->tcp_second_timer_threshold = tcps->tcps_ip_abort_interval;

2360                 tcp->tcp_fin_wait_2_flush_interval =
2361                     tcps->tcps_fin_wait_2_flush_interval;

2363                 tcp->tcp_ka_interval = tcps->tcps_keepalive_interval;
2364                 tcp->tcp_ka_abort_thres = tcps->tcps_keepalive_abort_interval;
2365                 tcp->tcp_ka_cnt = 0;
2366                 tcp->tcp_ka_rinterval = 0;

2368                 /*
2369                  * Default value of tcp_init_cwnd is 0, so no need to set here
2370                  * if parent is NULL.  But we need to inherit it from parent.
2371                  */
2372         } else {
2373                 /* Inherit various TCP parameters from the parent. */
2374                 tcp->tcp_naglim = parent->tcp_naglim;

2376                 tcp->tcp_rto_initial = parent->tcp_rto_initial;
2377                 tcp->tcp_rto_min = parent->tcp_rto_min;
2378                 tcp->tcp_rto_max = parent->tcp_rto_max;

2380                 tcp->tcp_first_ctimer_threshold =
2381                     parent->tcp_first_ctimer_threshold;
2382                 tcp->tcp_second_ctimer_threshold =
2383                     parent->tcp_second_ctimer_threshold;
2384                 tcp->tcp_first_timer_threshold =
2385                     parent->tcp_first_timer_threshold;
2386                 tcp->tcp_second_timer_threshold =
```



```
2387                     parent->tcp_second_timer_threshold;

2389                 tcp->tcp_fin_wait_2_flush_interval =
2390                     parent->tcp_fin_wait_2_flush_interval;

2392                 tcp->tcp_ka_interval = parent->tcp_ka_interval;
2393                 tcp->tcp_ka_abort_thres = parent->tcp_ka_abort_thres;
2394                 tcp->tcp_ka_cnt = parent->tcp_ka_cnt;
2395                 tcp->tcp_ka_rinterval = parent->tcp_ka_rinterval;

2397                 tcp->tcp_init_cwnd = parent->tcp_init_cwnd;
2398         }

2400         /*
2401          * Initialize tcp_rtt_sa and tcp_rtt_sd so that the calculated RTO
2402          * will be close to tcp_rexmit_interval_initial.  By doing this, we
2403          * allow the algorithm to adjust slowly to large fluctuations of RTT
2404          * during first few transmissions of a connection as seen in slow
2405          * links.
2406          */
2407         tcp->tcp_rtt_sa = tcp->tcp_rto_initial << 2;
2408         tcp->tcp_rtt_sd = tcp->tcp_rto_initial >> 1;
2409         rto = (tcp->tcp_rtt_sa >> 3) + tcp->tcp_rtt_sd +
2410             tcps->tcps_rexmit_interval_extra + (tcp->tcp_rtt_sa >> 5) +
2411             tcps->tcps_conn_grace_period;
2412         TCP_SET_RTO(tcp, rto);

2414         tcp->tcp_timer_backoff = 0;
2415         tcp->tcp_ms_we_have_waited = 0;
2416         tcp->tcp_last_recv_time = ddi_get_lbolt();
2417         tcp->tcp_cwnd_max = tcps->tcps_cwnd_max_;
2418         tcp->tcp_cwnd_ssthresh = TCP_MAX_LARGEWIN;
2419         tcp->tcp_snd_burst = TCP_CWND_INFINITE;

2420         tcp->tcp_maxpsz_multiplier = tcps->tcps_maxpsz_multiplier;

2422         /* NOTE:  ISS is now set in tcp_set_destination(). */

2424         /* Reset fusion-related fields */
2425         tcp->tcp_fused = B_FALSE;
2426         tcp->tcp_unfusable = B_FALSE;
2427         tcp->tcp_fused_sigurg = B_FALSE;
2428         tcp->tcp_loopback_peer = NULL;

2430         /* We rebuild the header template on the next connect/conn_request */

2432         connp->conn_mlp_type = mlptSingle;

2434         /*
2435          * Init the window scale to the max so tcp_rwnd_set() won't pare
2436          * down tcp_rwnd. tcp_set_destination() will set the right value later.
2437          */
2438         tcp->tcp_rcv_ws = TCP_MAX_WINSHIFT;
2439         tcp->tcp_rwnd = connp->conn_rcvbuf;

2441         tcp->tcp_cork = B_FALSE;
2442         /*
2443          * Init the tcp_debug option if it wasn't already set.  This value
2444          * determines whether TCP
2445          * calls strlog() to print out debug messages.  Doing this
2446          * initialization here means that this value is not inherited thru
2447          * tcp_reinit().
2448          */
2449         if (!connp->conn_debug)
2450                 connp->conn_debug = tcps->tcps_dbg;
2451 }
______unchanged_portion_omitted_
```

```
**********************************************************
  171991 Thu Nov  6 17:28:19 2014
new/usr/src/uts/common/inet/tcp/tcp_input.c
5295 remove maxburst logic from TCP's send algorithm Reviewed by: Dan McDonald <
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2011 Joyent, Inc. All rights reserved.
 * Copyright (c) 2014 by Delphix. All rights reserved.
 */

/* This file contains all TCP input processing functions. */

#include <sys/types.h>
#include <sys/stream.h>
#include <sys/strsun.h>
#include <sys/strsubr.h>
#include <sys/stropts.h>
#include <sys/strlog.h>
#define _SUN_TPI_VERSION 2
#include <sys/tihdr.h>
#include <sys/suntpi.h>
#include <sys/xti_inet.h>
#include <sys/squeue_impl.h>
#include <sys/squeue.h>
#include <sys/tsol/tnet.h>

#include <inet/common.h>
#include <inet/ip.h>
#include <inet/tcp.h>
#include <inet/tcp_impl.h>
#include <inet/tcp_cluster.h>
#include <inet/proto_set.h>
#include <inet/ipsec_impl.h>

/*
 * RFC1323-recommended phrasing of TSTAMP option, for easier parsing
 */

#ifdef _BIG_ENDIAN
#define TCPOPT_NOP_NOP_TSTAMP ((TCPOPT_NOP << 24) | (TCPOPT_NOP << 16) | \
        (TCPOPT_TSTAMP << 8) | 10)
#else
#define TCPOPT_NOP_NOP_TSTAMP ((10 << 24) | (TCPOPT_TSTAMP << 16) | \
```



```
        (TCPOPT_NOP << 8) | TCPOPT_NOP)
#endif

/*
 * Flags returned from tcp_parse_options.
 */
#define TCP_OPT_MSS_PRESENT     1
#define TCP_OPT_WSCALE_PRESENT  2
#define TCP_OPT_TSTAMP_PRESENT  4
#define TCP_OPT_SACK_OK_PRESENT 8
#define TCP_OPT_SACK_PRESENT    16

/*
 *  PAWS needs a timer for 24 days.  This is the number of ticks in 24 days
 */
#define PAWS_TIMEOUT    ((clock_t)(24*24*60*60*hz))

/*
 * Since tcp_listener is not cleared atomically with tcp_detached
 * being cleared we need this extra bit to tell a detached connection
 * apart from one that is in the process of being accepted.
 */
#define TCP_IS_DETACHED_NONEAGER(tcp)   \
        (TCP_IS_DETACHED(tcp) &&        \
            (!(tcp)->tcp_hard_binding))

/*
 * Steps to do when a tcp_t moves to TIME-WAIT state.
 *
 * This connection is done, we don't need to account for it.  Decrement
 * the listener connection counter if needed.
 *
 * Decrement the connection counter of the stack.  Note that this counter
 * is per CPU.  So the total number of connections in a stack is the sum of all
 * of them.  Since there is no lock for handling all of them exclusively, the
 * resulting sum is only an approximation.
 *
 * Unconditionally clear the exclusive binding bit so this TIME-WAIT
 * connection won't interfere with new ones.
 *
 * Start the TIME-WAIT timer.  If upper layer has not closed the connection,
 * the timer is handled within the context of this tcp_t.  When the timer
 * fires, tcp_clean_death() is called.  If upper layer closes the connection
 * during this period, tcp_time_wait_append() will be called to add this
 * tcp_t to the global TIME-WAIT list.  Note that this means that the
 * actual wait time in TIME-WAIT state will be longer than the
 * tcps_time_wait_interval since the period before upper layer closes the
 * connection is not accounted for when tcp_time_wait_append() is called.
 *
 * If uppser layer has closed the connection, call tcp_time_wait_append()
 * directly.
 *
 */
#define SET_TIME_WAIT(tcps, tcp, connp)                         \
{                                                               \
        (tcp)->tcp_state = TCPS_TIME_WAIT;                      \
        if ((tcp)->tcp_listen_cnt != NULL)                      \
                TCP_DECR_LISTEN_CNT(tcp);                       \
        atomic_dec_64(                                          \
            (uint64_t *)&(tcps)->tcps_sc[CPU->cpu_seqid]->tcp_sc_conn_cnt); \
        (connp)->conn_exclbind = 0;                             \
        if (!TCP_IS_DETACHED(tcp)) {                            \
                TCP_TIMER_RESTART(tcp, (tcps)->tcps_time_wait_interval); \
        } else {                                                \
                tcp_time_wait_append(tcp);                      \
                TCP_DBGSTAT(tcps, tcp_rput_time_wait);          \
```

```
 128         }                                                           \
 129 }
_____unchanged_portion_omitted_

2344 /*
2345  * Handle M_DATA messages from IP. Its called directly from IP via
2346  * squeue for received IP packets.
2347  *
2348  * The first argument is always the connp/tcp to which the mp belongs.
2349  * There are no exceptions to this rule. The caller has already put
2350  * a reference on this connp/tcp and once tcp_input_data() returns,
2351  * the squeue will do the refrele.
2352  *
2353  * The TH_SYN for the listener directly go to tcp_input_listener via
2354  * squeue. ICMP errors go directly to tcp_icmp_input().
2355  *
2356  * sqp: NULL = recursive, sqp != NULL means called from squeue
2357  */
2358 void
2359 tcp_input_data(void *arg, mblk_t *mp, void *arg2, ip_recv_attr_t *ira)
2360 {
2361         int32_t         bytes_acked;
2362         int32_t         gap;
2363         mblk_t          *mp1;
2364         uint_t          flags;
2365         uint32_t        new_swnd = 0;
2366         uchar_t         *iphdr;
2367         uchar_t         *rptr;
2368         int32_t         rgap;
2369         uint32_t        seg_ack;
2370         int             seg_len;
2371         uint_t          ip_hdr_len;
2372         uint32_t        seg_seq;
2373         tcpha_t         *tcpha;
2374         int             urp;
2375         tcp_opt_t       tcpopt;
2376         ip_pkt_t        ipp;
2377         boolean_t       ofo_seg = B_FALSE; /* Out of order segment */
2378         uint32_t        cwnd;
2379         uint32_t        add;
2380         int             npkt;
2381         int             mss;
2382         conn_t          *connp = (conn_t *)arg;
2383         squeue_t        *sqp = (squeue_t *)arg2;
2384         tcp_t           *tcp = connp->conn_tcp;
2385         tcp_stack_t     *tcps = tcp->tcp_tcps;
2386         sock_upcalls_t  *sockupcalls;

2388         /*
2389          * RST from fused tcp loopback peer should trigger an unfuse.
2390          */
2391         if (tcp->tcp_fused) {
2392                 TCP_STAT(tcps, tcp_fusion_aborted);
2393                 tcp_unfuse(tcp);
2394         }

2396         iphdr = mp->b_rptr;
2397         rptr = mp->b_rptr;
2398         ASSERT(OK_32PTR(rptr));

2400         ip_hdr_len = ira->ira_ip_hdr_length;
2401         if (connp->conn_recv_ancillary.crb_all != 0) {
2402                 /*
2403                  * Record packet information in the ip_pkt_t
2404                  */
2405                 ipp.ipp_fields = 0;
```



```
2406                 if (ira->ira_flags & IRAF_IS_IPV4) {
2407                         (void) ip_find_hdr_v4((ipha_t *)rptr, &ipp,
2408                             B_FALSE);
2409                 } else {
2410                         uint8_t nexthdrp;

2412                         /*
2413                          * IPv6 packets can only be received by applications
2414                          * that are prepared to receive IPv6 addresses.
2415                          * The IP fanout must ensure this.
2416                          */
2417                         ASSERT(connp->conn_family == AF_INET6);

2419                         (void) ip_find_hdr_v6(mp, (ip6_t *)rptr, B_TRUE, &ipp,
2420                             &nexthdrp);
2421                         ASSERT(nexthdrp == IPPROTO_TCP);

2423                         /* Could have caused a pullup? */
2424                         iphdr = mp->b_rptr;
2425                         rptr = mp->b_rptr;
2426                 }
2427         }
2428         ASSERT(DB_TYPE(mp) == M_DATA);
2429         ASSERT(mp->b_next == NULL);

2431         tcpha = (tcpha_t *)&rptr[ip_hdr_len];
2432         seg_seq = ntohl(tcpha->tha_seq);
2433         seg_ack = ntohl(tcpha->tha_ack);
2434         ASSERT((uintptr_t)(mp->b_wptr - rptr) <= (uintptr_t)INT_MAX);
2435         seg_len = (int)(mp->b_wptr - rptr) -
2436             (ip_hdr_len + TCP_HDR_LENGTH(tcpha));
2437         if ((mp1 = mp->b_cont) != NULL && mp1->b_datap->db_type == M_DATA) {
2438                 do {
2439                         ASSERT((uintptr_t)(mp1->b_wptr - mp1->b_rptr) <=
2440                             (uintptr_t)INT_MAX);
2441                         seg_len += (int)(mp1->b_wptr - mp1->b_rptr);
2442                 } while ((mp1 = mp1->b_cont) != NULL &&
2443                     mp1->b_datap->db_type == M_DATA);
2444         }

2446         DTRACE_TCP5(receive, mblk_t *, NULL, ip_xmit_attr_t *, connp->conn_ixa,
2447             __dtrace_tcp_void_ip_t *, iphdr, tcp_t *, tcp,
2448             __dtrace_tcp_tcph_t *, tcpha);

2450         if (tcp->tcp_state == TCPS_TIME_WAIT) {
2451                 tcp_time_wait_processing(tcp, mp, seg_seq, seg_ack,
2452                     seg_len, tcpha, ira);
2453                 return;
2454         }

2456         if (sqp != NULL) {
2457                 /*
2458                  * This is the correct place to update tcp_last_recv_time. Note
2459                  * that it is also updated for tcp structure that belongs to
2460                  * global and listener queues which do not really need updating.
2461                  * But that should not cause any harm.  And it is updated for
2462                  * all kinds of incoming segments, not only for data segments.
2463                  */
2464                 tcp->tcp_last_recv_time = LBOLT_FASTPATH;
2465         }

2467         flags = (unsigned int)tcpha->tha_flags & 0xFF;

2469         BUMP_LOCAL(tcp->tcp_ibsegs);
2470         DTRACE_PROBE2(tcp__trace__recv, mblk_t *, mp, tcp_t *, tcp);
```

```
2472         if ((flags & TH_URG) && sqp != NULL) {
2473                 /*
2474                  * TCP can't handle urgent pointers that arrive before
2475                  * the connection has been accept()ed since it can't
2476                  * buffer OOB data.  Discard segment if this happens.
2477                  *
2478                  * We can't just rely on a non-null tcp_listener to indicate
2479                  * that the accept() has completed since unlinking of the
2480                  * eager and completion of the accept are not atomic.
2481                  * tcp_detached, when it is not set (B_FALSE) indicates
2482                  * that the accept() has completed.
2483                  *
2484                  * Nor can it reassemble urgent pointers, so discard
2485                  * if it's not the next segment expected.
2486                  *
2487                  * Otherwise, collapse chain into one mblk (discard if
2488                  * that fails).  This makes sure the headers, retransmitted
2489                  * data, and new data all are in the same mblk.
2490                  */
2491                 ASSERT(mp != NULL);
2492                 if (tcp->tcp_detached || !pullupmsg(mp, -1)) {
2493                         freemsg(mp);
2494                         return;
2495                 }
2496                 /* Update pointers into message */
2497                 iphdr = rptr = mp->b_rptr;
2498                 tcpha = (tcpha_t *)&rptr[ip_hdr_len];
2499                 if (SEQ_GT(seg_seq, tcp->tcp_rnxt)) {
2500                         /*
2501                          * Since we can't handle any data with this urgent
2502                          * pointer that is out of sequence, we expunge
2503                          * the data.  This allows us to still register
2504                          * the urgent mark and generate the M_PCSIG,
2505                          * which we can do.
2506                          */
2507                         mp->b_wptr = (uchar_t *)tcpha + TCP_HDR_LENGTH(tcpha);
2508                         seg_len = 0;
2509                 }
2510         }

2512         sockupcalls = connp->conn_upcalls;
2513         /* A conn_t may have belonged to a now-closed socket.  Be careful. */
2514         if (sockupcalls == NULL)
2515                 sockupcalls = &tcp_dummy_upcalls;

2517         switch (tcp->tcp_state) {
2518         case TCPS_SYN_SENT:
2519                 if (connp->conn_final_sqp == NULL &&
2520                     tcp_outbound_squeue_switch && sqp != NULL) {
2521                         ASSERT(connp->conn_initial_sqp == connp->conn_sqp);
2522                         connp->conn_final_sqp = sqp;
2523                         if (connp->conn_final_sqp != connp->conn_sqp) {
2524                                 DTRACE_PROBE1(conn__final__sqp__switch,
2525                                     conn_t *, connp);
2526                                 CONN_INC_REF(connp);
2527                                 SQUEUE_SWITCH(connp, connp->conn_final_sqp);
2528                                 SQUEUE_ENTER_ONE(connp->conn_sqp, mp,
2529                                     tcp_input_data, connp, ira, ip_squeue_flag,
2530                                     SQTAG_CONNECT_FINISH);
2531                                 return;
2532                         }
2533                         DTRACE_PROBE1(conn__final__sqp__same, conn_t *, connp);
2534                 }
2535                 if (flags & TH_ACK) {
2536                         /*
2537                          * Note that our stack cannot send data before a
```



```
2538                          * connection is established, therefore the
2539                          * following check is valid.  Otherwise, it has
2540                          * to be changed.
2541                          */
2542                         if (SEQ_LEQ(seg_ack, tcp->tcp_iss) ||
2543                             SEQ_GT(seg_ack, tcp->tcp_snxt)) {
2544                                 freemsg(mp);
2545                                 if (flags & TH_RST)
2546                                         return;
2547                                 tcp_xmit_ctl("TCPS_SYN_SENT-Bad_seq",
2548                                     tcp, seg_ack, 0, TH_RST);
2549                                 return;
2550                         }
2551                         ASSERT(tcp->tcp_suna + 1 == seg_ack);
2552                 }
2553                 if (flags & TH_RST) {
2554                         if (flags & TH_ACK) {
2555                                 DTRACE_TCP5(connect__refused, mblk_t *, NULL,
2556                                     ip_xmit_attr_t *, connp->conn_ixa,
2557                                     void_ip_t *, iphdr, tcp_t *, tcp,
2558                                     tcph_t *, tcpha);
2559                                 (void) tcp_clean_death(tcp, ECONNREFUSED);
2560                         }
2561                         freemsg(mp);
2562                         return;
2563                 }
2564                 if (!(flags & TH_SYN)) {
2565                         freemsg(mp);
2566                         return;
2567                 }

2569                 /* Process all TCP options. */
2570                 tcp_process_options(tcp, tcpha);
2571                 /*
2572                  * The following changes our rwnd to be a multiple of the
2573                  * MIN(peer MSS, our MSS) for performance reason.
2574                  */
2575                 (void) tcp_rwnd_set(tcp, MSS_ROUNDUP(connp->conn_rcvbuf,
2576                     tcp->tcp_mss));

2578                 /* Is the other end ECN capable? */
2579                 if (tcp->tcp_ecn_ok) {
2580                         if ((flags & (TH_ECE|TH_CWR)) != TH_ECE) {
2581                                 tcp->tcp_ecn_ok = B_FALSE;
2582                         }
2583                 }
2584                 /*
2585                  * Clear ECN flags because it may interfere with later
2586                  * processing.
2587                  */
2588                 flags &= ~(TH_ECE|TH_CWR);

2590                 tcp->tcp_irs = seg_seq;
2591                 tcp->tcp_rack = seg_seq;
2592                 tcp->tcp_rnxt = seg_seq + 1;
2593                 tcp->tcp_tcpha->tha_ack = htonl(tcp->tcp_rnxt);
2594                 if (!TCP_IS_DETACHED(tcp)) {
2595                         /* Allocate room for SACK options if needed. */
2596                         connp->conn_wroff = connp->conn_ht_iphc_len;
2597                         if (tcp->tcp_snd_sack_ok)
2598                                 connp->conn_wroff += TCPOPT_MAX_SACK_LEN;
2599                         if (!tcp->tcp_loopback)
2600                                 connp->conn_wroff += tcps->tcps_wroff_xtra;

2602                         (void) proto_set_tx_wroff(connp->conn_rq, connp,
2603                             connp->conn_wroff);
```

```
2604                }
2605                if (flags & TH_ACK) {
2606                        /*
2607                         * If we can't get the confirmation upstream, pretend
2608                         * we didn't even see this one.
2609                         *
2610                         * XXX: how can we pretend we didn't see it if we
2611                         * have updated rnxt et. al.
2612                         *
2613                         * For loopback we defer sending up the T_CONN_CON
2614                         * until after some checks below.
2615                         */
2616                        mp1 = NULL;
2617                        /*
2618                         * tcp_sendmsg() checks tcp_state without entering
2619                         * the squeue so tcp_state should be updated before
2620                         * sending up connection confirmation.  Probe the
2621                         * state change below when we are sure the connection
2622                         * confirmation has been sent.
2623                         */
2624                        tcp->tcp_state = TCPS_ESTABLISHED;
2625                        if (!tcp_conn_con(tcp, iphdr, mp,
2626                            tcp->tcp_loopback ? &mp1 : NULL, ira)) {
2627                                tcp->tcp_state = TCPS_SYN_SENT;
2628                                freemsg(mp);
2629                                return;
2630                        }
2631                        TCPS_CONN_INC(tcps);
2632                        /* SYN was acked - making progress */
2633                        tcp->tcp_ip_forward_progress = B_TRUE;

2635                        /* One for the SYN */
2636                        tcp->tcp_suna = tcp->tcp_iss + 1;
2637                        tcp->tcp_valid_bits &= ~TCP_ISS_VALID;

2639                        /*
2640                         * If SYN was retransmitted, need to reset all
2641                         * retransmission info.  This is because this
2642                         * segment will be treated as a dup ACK.
2643                         */
2644                        if (tcp->tcp_rexmit) {
2645                                tcp->tcp_rexmit = B_FALSE;
2646                                tcp->tcp_rexmit_nxt = tcp->tcp_snxt;
2647                                tcp->tcp_rexmit_max = tcp->tcp_snxt;
2647                                tcp->tcp_snd_burst = tcp->tcp_localnet ?
2648                                    TCP_CWND_INFINITE : TCP_CWND_NORMAL;
2648                                tcp->tcp_ms_we_have_waited = 0;

2650                                /*
2651                                 * Set tcp_cwnd back to 1 MSS, per
2652                                 * recommendation from
2653                                 * draft-floyd-incr-init-win-01.txt,
2654                                 * Increasing TCP's Initial Window.
2655                                 */
2656                                tcp->tcp_cwnd = tcp->tcp_mss;
2657                        }

2659                        tcp->tcp_swl1 = seg_seq;
2660                        tcp->tcp_swl2 = seg_ack;

2662                        new_swnd = ntohs(tcpha->tha_win);
2663                        tcp->tcp_swnd = new_swnd;
2664                        if (new_swnd > tcp->tcp_max_swnd)
2665                                tcp->tcp_max_swnd = new_swnd;

2667                        /*
```



```
2668                         * Always send the three-way handshake ack immediately
2669                         * in order to make the connection complete as soon as
2670                         * possible on the accepting host.
2671                         */
2672                        flags |= TH_ACK_NEEDED;

2674                        /*
2675                         * Trace connect-established here.
2676                         */
2677                        DTRACE_TCP5(connect__established, mblk_t *, NULL,
2678                            ip_xmit_attr_t *, tcp->tcp_connp->conn_ixa,
2679                            void_ip_t *, iphdr, tcp_t *, tcp, tcph_t *, tcpha);

2681                        /* Trace change from SYN_SENT -> ESTABLISHED here */
2682                        DTRACE_TCP6(state__change, void, NULL, ip_xmit_attr_t *,
2683                            connp->conn_ixa, void, NULL, tcp_t *, tcp,
2684                            void, NULL, int32_t, TCPS_SYN_SENT);

2686                        /*
2687                         * Special case for loopback.  At this point we have
2688                         * received SYN-ACK from the remote endpoint.  In
2689                         * order to ensure that both endpoints reach the
2690                         * fused state prior to any data exchange, the final
2691                         * ACK needs to be sent before we indicate T_CONN_CON
2692                         * to the module upstream.
2693                         */
2694                        if (tcp->tcp_loopback) {
2695                                mblk_t *ack_mp;

2697                                ASSERT(!tcp->tcp_unfusable);
2698                                ASSERT(mp1 != NULL);
2699                                /*
2700                                 * For loopback, we always get a pure SYN-ACK
2701                                 * and only need to send back the final ACK
2702                                 * with no data (this is because the other
2703                                 * tcp is ours and we don't do T/TCP).  This
2704                                 * final ACK triggers the passive side to
2705                                 * perform fusion in ESTABLISHED state.
2706                                 */
2707                                if ((ack_mp = tcp_ack_mp(tcp)) != NULL) {
2708                                        if (tcp->tcp_ack_tid != 0) {
2709                                                (void) TCP_TIMER_CANCEL(tcp,
2710                                                    tcp->tcp_ack_tid);
2711                                                tcp->tcp_ack_tid = 0;
2712                                        }
2713                                        tcp_send_data(tcp, ack_mp);
2714                                        BUMP_LOCAL(tcp->tcp_obsegs);
2715                                        TCPS_BUMP_MIB(tcps, tcpOutAck);

2717                                        if (!IPCL_IS_NONSTR(connp)) {
2718                                                /* Send up T_CONN_CON */
2719                                                if (ira->ira_cred != NULL) {
2720                                                        mblk_setcred(mp1,
2721                                                            ira->ira_cred,
2722                                                            ira->ira_cpid);
2723                                                }
2724                                                putnext(connp->conn_rq, mp1);
2725                                        } else {
2726                                                (*sockupcalls->su_connected)
2727                                                    (connp->conn_upper_handle,
2728                                                    tcp->tcp_connid,
2729                                                    ira->ira_cred,
2730                                                    ira->ira_cpid);
2731                                                freemsg(mp1);
2732                                        }
```

```
2734                                    freemsg(mp);
2735                                    return;
2736                            }
2737                            /*
2738                             * Forget fusion; we need to handle more
2739                             * complex cases below.  Send the deferred
2740                             * T_CONN_CON message upstream and proceed
2741                             * as usual.  Mark this tcp as not capable
2742                             * of fusion.
2743                             */
2744                            TCP_STAT(tcps, tcp_fusion_unfusable);
2745                            tcp->tcp_unfusable = B_TRUE;
2746                            if (!IPCL_IS_NONSTR(connp)) {
2747                                    if (ira->ira_cred != NULL) {
2748                                            mblk_setcred(mp1, ira->ira_cred,
2749                                                ira->ira_cpid);
2750                                    }
2751                                    putnext(connp->conn_rq, mp1);
2752                            } else {
2753                                    (*sockupcalls->su_connected)
2754                                        (connp->conn_upper_handle,
2755                                        tcp->tcp_connid, ira->ira_cred,
2756                                        ira->ira_cpid);
2757                                    freemsg(mp1);
2758                            }
2759                    }

2761                    /*
2762                     * Check to see if there is data to be sent.  If
2763                     * yes, set the transmit flag.  Then check to see
2764                     * if received data processing needs to be done.
2765                     * If not, go straight to xmit_check.  This short
2766                     * cut is OK as we don't support T/TCP.
2767                     */
2768                    if (tcp->tcp_unsent)
2769                            flags |= TH_XMIT_NEEDED;

2771                    if (seg_len == 0 && !(flags & TH_URG)) {
2772                            freemsg(mp);
2773                            goto xmit_check;
2774                    }

2776                    flags &= ~TH_SYN;
2777                    seg_seq++;
2778                    break;
2779            }
2780            tcp->tcp_state = TCPS_SYN_RCVD;
2781            DTRACE_TCP6(state__change, void, NULL, ip_xmit_attr_t *,
2782                connp->conn_ixa, void_ip_t *, NULL, tcp_t *, tcp,
2783                tcph_t *, NULL, int32_t, TCPS_SYN_SENT);
2784            mp1 = tcp_xmit_mp(tcp, tcp->tcp_xmit_head, tcp->tcp_mss,
2785                NULL, NULL, tcp->tcp_iss, B_FALSE, NULL, B_FALSE);
2786            if (mp1 != NULL) {
2787                    tcp_send_data(tcp, mp1);
2788                    TCP_TIMER_RESTART(tcp, tcp->tcp_rto);
2789            }
2790            freemsg(mp);
2791            return;
2792        case TCPS_SYN_RCVD:
2793                if (flags & TH_ACK) {
2794                        uint32_t pinit_wnd;

2796                        /*
2797                         * In this state, a SYN|ACK packet is either bogus
2798                         * because the other side must be ACKing our SYN which
2799                         * indicates it has seen the ACK for their SYN and
```



```
2800                         * shouldn't retransmit it or we're crossing SYNs
2801                         * on active open.
2802                         */
2803                        if ((flags & TH_SYN) && !tcp->tcp_active_open) {
2804                                freemsg(mp);
2805                                tcp_xmit_ctl("TCPS_SYN_RCVD-bad_syn",
2806                                    tcp, seg_ack, 0, TH_RST);
2807                                return;
2808                        }
2809                        /*
2810                         * NOTE: RFC 793 pg. 72 says this should be
2811                         * tcp->tcp_suna <= seg_ack <= tcp->tcp_snxt
2812                         * but that would mean we have an ack that ignored
2813                         * our SYN.
2814                         */
2815                        if (SEQ_LEQ(seg_ack, tcp->tcp_suna) ||
2816                            SEQ_GT(seg_ack, tcp->tcp_snxt)) {
2817                                freemsg(mp);
2818                                tcp_xmit_ctl("TCPS_SYN_RCVD-bad_ack",
2819                                    tcp, seg_ack, 0, TH_RST);
2820                                return;
2821                        }
2822                        /*
2823                         * No sane TCP stack will send such a small window
2824                         * without receiving any data.  Just drop this invalid
2825                         * ACK.  We also shorten the abort timeout in case
2826                         * this is an attack.
2827                         */
2828                        pinit_wnd = ntohs(tcpha->tha_win) << tcp->tcp_snd_ws;
2829                        if (pinit_wnd < tcp->tcp_mss &&
2830                            pinit_wnd < tcp_init_wnd_chk) {
2831                                freemsg(mp);
2832                                TCP_STAT(tcps, tcp_zwin_ack_syn);
2833                                tcp->tcp_second_ctimer_threshold =
2834                                    tcp_early_abort * SECONDS;
2835                                return;
2836                        }
2837                }
2838                break;
2839        case TCPS_LISTEN:
2840                /*
2841                 * Only a TLI listener can come through this path when a
2842                 * acceptor is going back to be a listener and a packet
2843                 * for the acceptor hits the classifier. For a socket
2844                 * listener, this can never happen because a listener
2845                 * can never accept connection on itself and hence a
2846                 * socket acceptor can not go back to being a listener.
2847                 */
2848                ASSERT(!TCP_IS_SOCKET(tcp));
2849                /*FALLTHRU*/
2850        case TCPS_CLOSED:
2851        case TCPS_BOUND: {
2852                conn_t  *new_connp;
2853                ip_stack_t *ipst = tcps->tcps_netstack->netstack_ip;

2855                /*
2856                 * Don't accept any input on a closed tcp as this TCP logically
2857                 * does not exist on the system. Don't proceed further with
2858                 * this TCP. For instance, this packet could trigger another
2859                 * close of this tcp which would be disastrous for tcp_refcnt.
2860                 * tcp_close_detached / tcp_clean_death / tcp_closei_local must
2861                 * be called at most once on a TCP. In this case we need to
2862                 * refeed the packet into the classifier and figure out where
2863                 * the packet should go.
2864                 */
2865                new_connp = ipcl_classify(mp, ira, ipst);
```

```
2866                if (new_connp != NULL) {
2867                        /* Drops ref on new_connp */
2868                        tcp_reinput(new_connp, mp, ira, ipst);
2869                        return;
2870                }
2871                /* We failed to classify. For now just drop the packet */
2872                freemsg(mp);
2873                return;
2874        }
2875        case TCPS_IDLE:
2876                /*
2877                 * Handle the case where the tcp_clean_death() has happened
2878                 * on a connection (application hasn't closed yet) but a packet
2879                 * was already queued on squeue before tcp_clean_death()
2880                 * was processed. Calling tcp_clean_death() twice on same
2881                 * connection can result in weird behaviour.
2882                 */
2883                freemsg(mp);
2884                return;
2885        default:
2886                break;
2887        }

2889        /*
2890         * Already on the correct queue/perimeter.
2891         * If this is a detached connection and not an eager
2892         * connection hanging off a listener then new data
2893         * (past the FIN) will cause a reset.
2894         * We do a special check here where it
2895         * is out of the main line, rather than check
2896         * if we are detached every time we see new
2897         * data down below.
2898         */
2899        if (TCP_IS_DETACHED_NONEAGER(tcp) &&
2900            (seg_len > 0 && SEQ_GT(seg_seq + seg_len, tcp->tcp_rnxt))) {
2901                TCPS_BUMP_MIB(tcps, tcpInClosed);
2902                DTRACE_PROBE2(tcp__trace__recv, mblk_t *, mp, tcp_t *, tcp);
2903                freemsg(mp);
2904                tcp_xmit_ctl("new data when detached", tcp,
2905                    tcp->tcp_snxt, 0, TH_RST);
2906                (void) tcp_clean_death(tcp, EPROTO);
2907                return;
2908        }

2910        mp->b_rptr = (uchar_t *)tcpha + TCP_HDR_LENGTH(tcpha);
2911        urp = ntohs(tcpha->tha_urp) - TCP_OLD_URP_INTERPRETATION;
2912        new_swnd = ntohs(tcpha->tha_win) <<
2913            ((tcpha->tha_flags & TH_SYN) ? 0 : tcp->tcp_snd_ws);

2915        if (tcp->tcp_snd_ts_ok) {
2916                if (!tcp_paws_check(tcp, tcpha, &tcpopt)) {
2917                        /*
2918                         * This segment is not acceptable.
2919                         * Drop it and send back an ACK.
2920                         */
2921                        freemsg(mp);
2922                        flags |= TH_ACK_NEEDED;
2923                        goto ack_check;
2924                }
2925        } else if (tcp->tcp_snd_sack_ok) {
2926                tcpopt.tcp = tcp;
2927                /*
2928                 * SACK info in already updated in tcp_parse_options.  Ignore
2929                 * all other TCP options...
2930                 */
2931                (void) tcp_parse_options(tcpha, &tcpopt);
```



```
2932        }
2933 try_again:;
2934        mss = tcp->tcp_mss;
2935        gap = seg_seq - tcp->tcp_rnxt;
2936        rgap = tcp->tcp_rwnd - (gap + seg_len);
2937        /*
2938         * gap is the amount of sequence space between what we expect to see
2939         * and what we got for seg_seq.  A positive value for gap means
2940         * something got lost.  A negative value means we got some old stuff.
2941         */
2942        if (gap < 0) {
2943                /* Old stuff present.  Is the SYN in there? */
2944                if (seg_seq == tcp->tcp_irs && (flags & TH_SYN) &&
2945                    (seg_len != 0)) {
2946                        flags &= ~TH_SYN;
2947                        seg_seq++;
2948                        urp--;
2949                        /* Recompute the gaps after noting the SYN. */
2950                        goto try_again;
2951                }
2952                TCPS_BUMP_MIB(tcps, tcpInDataDupSegs);
2953                TCPS_UPDATE_MIB(tcps, tcpInDataDupBytes,
2954                    (seg_len > -gap ? -gap : seg_len));
2955                /* Remove the old stuff from seg_len. */
2956                seg_len += gap;
2957                /*
2958                 * Anything left?
2959                 * Make sure to check for unack'd FIN when rest of data
2960                 * has been previously ack'd.
2961                 */
2962                if (seg_len < 0 || (seg_len == 0 && !(flags & TH_FIN))) {
2963                        /*
2964                         * Resets are only valid if they lie within our offered
2965                         * window.  If the RST bit is set, we just ignore this
2966                         * segment.
2967                         */
2968                        if (flags & TH_RST) {
2969                                freemsg(mp);
2970                                return;
2971                        }

2973                        /*
2974                         * The arriving of dup data packets indicate that we
2975                         * may have postponed an ack for too long, or the other
2976                         * side's RTT estimate is out of shape. Start acking
2977                         * more often.
2978                         */
2979                        if (SEQ_GEQ(seg_seq + seg_len - gap, tcp->tcp_rack) &&
2980                            tcp->tcp_rack_cnt >= 1 &&
2981                            tcp->tcp_rack_abs_max > 2) {
2982                                tcp->tcp_rack_abs_max--;
2983                        }
2984                        tcp->tcp_rack_cur_max = 1;

2986                        /*
2987                         * This segment is "unacceptable".  None of its
2988                         * sequence space lies within our advertized window.
2989                         *
2990                         * Adjust seg_len to the original value for tracing.
2991                         */
2992                        seg_len -= gap;
2993                        if (connp->conn_debug) {
2994                                (void) strlog(TCP_MOD_ID, 0, 1, SL_TRACE,
2995                                    "tcp_rput: unacceptable, gap %d, rgap %d, "
2996                                    "flags 0x%x, seg_seq %u, seg_ack %u, "
2997                                    "seg_len %d, rnxt %u, snxt %u, %s",
```

```
2998                                    gap, rgap, flags, seg_seq, seg_ack,
2999                                    seg_len, tcp->tcp_rnxt, tcp->tcp_snxt,
3000                                    tcp_display(tcp, NULL,
3001                                    DISP_ADDR_AND_PORT));
3002                            }
3003
3004                            /*
3005                             * Arrange to send an ACK in response to the
3006                             * unacceptable segment per RFC 793 page 69. There
3007                             * is only one small difference between ours and the
3008                             * acceptability test in the RFC - we accept ACK-only
3009                             * packet with SEG.SEQ = RCV.NXT+RCV.WND and no ACK
3010                             * will be generated.
3011                             *
3012                             * Note that we have to ACK an ACK-only packet at least
3013                             * for stacks that send 0-length keep-alives with
3014                             * SEG.SEQ = SND.NXT-1 as recommended by RFC1122,
3015                             * section 4.2.3.6. As long as we don't ever generate
3016                             * an unacceptable packet in response to an incoming
3017                             * packet that is unacceptable, it should not cause
3018                             * "ACK wars".
3019                             */
3020                            flags |=  TH_ACK_NEEDED;
3021
3022                            /*
3023                             * Continue processing this segment in order to use the
3024                             * ACK information it contains, but skip all other
3025                             * sequence-number processing.  Processing the ACK
3026                             * information is necessary in order to
3027                             * re-synchronize connections that may have lost
3028                             * synchronization.
3029                             *
3030                             * We clear seg_len and flag fields related to
3031                             * sequence number processing as they are not
3032                             * to be trusted for an unacceptable segment.
3033                             */
3034                            seg_len = 0;
3035                            flags &= ~(TH_SYN | TH_FIN | TH_URG);
3036                            goto process_ack;
3037                    }
3038
3039                    /* Fix seg_seq, and chew the gap off the front. */
3040                    seg_seq = tcp->tcp_rnxt;
3041                    urp += gap;
3042                    do {
3043                            mblk_t  *mp2;
3044                            ASSERT((uintptr_t)(mp->b_wptr - mp->b_rptr) <=
3045                                (uintptr_t)UINT_MAX);
3046                            gap += (uint_t)(mp->b_wptr - mp->b_rptr);
3047                            if (gap > 0) {
3048                                    mp->b_rptr = mp->b_wptr - gap;
3049                                    break;
3050                            }
3051                            mp2 = mp;
3052                            mp = mp->b_cont;
3053                            freeb(mp2);
3054                    } while (gap < 0);
3055                    /*
3056                     * If the urgent data has already been acknowledged, we
3057                     * should ignore TH_URG below
3058                     */
3059                    if (urp < 0)
3060                            flags &= ~TH_URG;
3061            }
3062            /*
3063             * rgap is the amount of stuff received out of window.  A negative
```



```
3064             * value is the amount out of window.
3065             */
3066            if (rgap < 0) {
3067                    mblk_t  *mp2;
3068
3069                    if (tcp->tcp_rwnd == 0) {
3070                            TCPS_BUMP_MIB(tcps, tcpInWinProbe);
3071                    } else {
3072                            TCPS_BUMP_MIB(tcps, tcpInDataPastWinSegs);
3073                            TCPS_UPDATE_MIB(tcps, tcpInDataPastWinBytes, -rgap);
3074                    }
3075
3076                    /*
3077                     * seg_len does not include the FIN, so if more than
3078                     * just the FIN is out of window, we act like we don't
3079                     * see it.  (If just the FIN is out of window, rgap
3080                     * will be zero and we will go ahead and acknowledge
3081                     * the FIN.)
3082                     */
3083                    flags &= ~TH_FIN;
3084
3085                    /* Fix seg_len and make sure there is something left. */
3086                    seg_len += rgap;
3087                    if (seg_len <= 0) {
3088                            /*
3089                             * Resets are only valid if they lie within our offered
3090                             * window.  If the RST bit is set, we just ignore this
3091                             * segment.
3092                             */
3093                            if (flags & TH_RST) {
3094                                    freemsg(mp);
3095                                    return;
3096                            }
3097
3098                            /* Per RFC 793, we need to send back an ACK. */
3099                            flags |= TH_ACK_NEEDED;
3100
3101                            /*
3102                             * Send SIGURG as soon as possible i.e. even
3103                             * if the TH_URG was delivered in a window probe
3104                             * packet (which will be unacceptable).
3105                             *
3106                             * We generate a signal if none has been generated
3107                             * for this connection or if this is a new urgent
3108                             * byte. Also send a zero-length "unmarked" message
3109                             * to inform SIOCATMARK that this is not the mark.
3110                             *
3111                             * tcp_urp_last_valid is cleared when the T_exdata_ind
3112                             * is sent up. This plus the check for old data
3113                             * (gap >= 0) handles the wraparound of the sequence
3114                             * number space without having to always track the
3115                             * correct MAX(tcp_urp_last, tcp_rnxt). (BSD tracks
3116                             * this max in its rcv_up variable).
3117                             *
3118                             * This prevents duplicate SIGURGS due to a "late"
3119                             * zero-window probe when the T_EXDATA_IND has already
3120                             * been sent up.
3121                             */
3122                            if ((flags & TH_URG) &&
3123                                (!tcp->tcp_urp_last_valid || SEQ_GT(urp + seg_seq,
3124                                tcp->tcp_urp_last))) {
3125                                    if (IPCL_IS_NONSTR(connp)) {
3126                                            if (!TCP_IS_DETACHED(tcp)) {
3127                                                    (*sockupcalls->su_signal_oob)
3128                                                        (connp->conn_upper_handle,
3129                                                        urp);
```

```
3130                                    }
3131                            } else {
3132                                    mp1 = allocb(0, BPRI_MED);
3133                                    if (mp1 == NULL) {
3134                                            freemsg(mp);
3135                                            return;
3136                                    }
3137                                    if (!TCP_IS_DETACHED(tcp) &&
3138                                        !putnextctl1(connp->conn_rq,
3139                                        M_PCSIG, SIGURG)) {
3140                                            /* Try again on the rexmit. */
3141                                            freemsg(mp1);
3142                                            freemsg(mp);
3143                                            return;
3144                                    }
3145                                    /*
3146                                     * If the next byte would be the mark
3147                                     * then mark with MARKNEXT else mark
3148                                     * with NOTMARKNEXT.
3149                                     */
3150                                    if (gap == 0 && urp == 0)
3151                                            mp1->b_flag |= MSGMARKNEXT;
3152                                    else
3153                                            mp1->b_flag |= MSGNOTMARKNEXT;
3154                                    freemsg(tcp->tcp_urp_mark_mp);
3155                                    tcp->tcp_urp_mark_mp = mp1;
3156                                    flags |= TH_SEND_URP_MARK;
3157                            }
3158                            tcp->tcp_urp_last_valid = B_TRUE;
3159                            tcp->tcp_urp_last = urp + seg_seq;
3160                    }
3161                    /*
3162                     * If this is a zero window probe, continue to
3163                     * process the ACK part.  But we need to set seg_len
3164                     * to 0 to avoid data processing.  Otherwise just
3165                     * drop the segment and send back an ACK.
3166                     */
3167                    if (tcp->tcp_rwnd == 0 && seg_seq == tcp->tcp_rnxt) {
3168                            flags &= ~(TH_SYN | TH_URG);
3169                            seg_len = 0;
3170                            goto process_ack;
3171                    } else {
3172                            freemsg(mp);
3173                            goto ack_check;
3174                    }
3175            }
3176            /* Pitch out of window stuff off the end. */
3177            rgap = seg_len;
3178            mp2 = mp;
3179            do {
3180                    ASSERT((uintptr_t)(mp2->b_wptr - mp2->b_rptr) <=
3181                        (uintptr_t)INT_MAX);
3182                    rgap -= (int)(mp2->b_wptr - mp2->b_rptr);
3183                    if (rgap < 0) {
3184                            mp2->b_wptr += rgap;
3185                            if ((mp1 = mp2->b_cont) != NULL) {
3186                                    mp2->b_cont = NULL;
3187                                    freemsg(mp1);
3188                            }
3189                            break;
3190                    }
3191            } while ((mp2 = mp2->b_cont) != NULL);
3192    }
3193 ok:;
3194    /*
3195     * TCP should check ECN info for segments inside the window only.
```



```
3196     * Therefore the check should be done here.
3197     */
3198    if (tcp->tcp_ecn_ok) {
3199            if (flags & TH_CWR) {
3200                    tcp->tcp_ecn_echo_on = B_FALSE;
3201            }
3202            /*
3203             * Note that both ECN_CE and CWR can be set in the
3204             * same segment.  In this case, we once again turn
3205             * on ECN_ECHO.
3206             */
3207            if (connp->conn_ipversion == IPV4_VERSION) {
3208                    uchar_t tos = ((ipha_t *)rptr)->ipha_type_of_service;

3210                    if ((tos & IPH_ECN_CE) == IPH_ECN_CE) {
3211                            tcp->tcp_ecn_echo_on = B_TRUE;
3212                    }
3213            } else {
3214                    uint32_t vcf = ((ip6_t *)rptr)->ip6_vcf;

3216                    if ((vcf & htonl(IPH_ECN_CE << 20)) ==
3217                        htonl(IPH_ECN_CE << 20)) {
3218                            tcp->tcp_ecn_echo_on = B_TRUE;
3219                    }
3220            }
3221    }

3223    /*
3224     * Check whether we can update tcp_ts_recent.  This test is
3225     * NOT the one in RFC 1323 3.4.  It is from Braden, 1993, "TCP
3226     * Extensions for High Performance: An Update", Internet Draft.
3227     */
3228    if (tcp->tcp_snd_ts_ok &&
3229        TSTMP_GEQ(tcpopt.tcp_opt_ts_val, tcp->tcp_ts_recent) &&
3230        SEQ_LEQ(seg_seq, tcp->tcp_rack)) {
3231            tcp->tcp_ts_recent = tcpopt.tcp_opt_ts_val;
3232            tcp->tcp_last_rcv_lbolt = LBOLT_FASTPATH64;
3233    }

3235    if (seg_seq != tcp->tcp_rnxt || tcp->tcp_reass_head) {
3236            /*
3237             * FIN in an out of order segment.  We record this in
3238             * tcp_valid_bits and the seq num of FIN in tcp_ofo_fin_seq.
3239             * Clear the FIN so that any check on FIN flag will fail.
3240             * Remember that FIN also counts in the sequence number
3241             * space.  So we need to ack out of order FIN only segments.
3242             */
3243            if (flags & TH_FIN) {
3244                    tcp->tcp_valid_bits |= TCP_OFO_FIN_VALID;
3245                    tcp->tcp_ofo_fin_seq = seg_seq + seg_len;
3246                    flags &= ~TH_FIN;
3247                    flags |= TH_ACK_NEEDED;
3248            }
3249            if (seg_len > 0) {
3250                    /* Fill in the SACK blk list. */
3251                    if (tcp->tcp_snd_sack_ok) {
3252                            tcp_sack_insert(tcp->tcp_sack_list,
3253                                seg_seq, seg_seq + seg_len,
3254                                &(tcp->tcp_num_sack_blk));
3255                    }

3257                    /*
3258                     * Attempt reassembly and see if we have something
3259                     * ready to go.
3260                     */
3261                    mp = tcp_reass(tcp, mp, seg_seq);
```

```
3262                        /* Always ack out of order packets */
3263                        flags |= TH_ACK_NEEDED | TH_PUSH;
3264                        if (mp) {
3265                                ASSERT((uintptr_t)(mp->b_wptr - mp->b_rptr) <=
3266                                    (uintptr_t)INT_MAX);
3267                                seg_len = mp->b_cont ? msgdsize(mp) :
3268                                    (int)(mp->b_wptr - mp->b_rptr);
3269                                seg_seq = tcp->tcp_rnxt;
3270                                /*
3271                                 * A gap is filled and the seq num and len
3272                                 * of the gap match that of a previously
3273                                 * received FIN, put the FIN flag back in.
3274                                 */
3275                                if ((tcp->tcp_valid_bits & TCP_OFO_FIN_VALID) &&
3276                                    seg_seq + seg_len == tcp->tcp_ofo_fin_seq) {
3277                                        flags |= TH_FIN;
3278                                        tcp->tcp_valid_bits &=
3279                                            ~TCP_OFO_FIN_VALID;
3280                                }
3281                                if (tcp->tcp_reass_tid != 0) {
3282                                        (void) TCP_TIMER_CANCEL(tcp,
3283                                            tcp->tcp_reass_tid);
3284                                        /*
3285                                         * Restart the timer if there is still
3286                                         * data in the reassembly queue.
3287                                         */
3288                                        if (tcp->tcp_reass_head != NULL) {
3289                                                tcp->tcp_reass_tid = TCP_TIMER(
3290                                                    tcp, tcp_reass_timer,
3291                                                    tcps->tcps_reass_timeout);
3292                                        } else {
3293                                                tcp->tcp_reass_tid = 0;
3294                                        }
3295                                }
3296                        } else {
3297                                /*
3298                                 * Keep going even with NULL mp.
3299                                 * There may be a useful ACK or something else
3300                                 * we don't want to miss.
3301                                 *
3302                                 * But TCP should not perform fast retransmit
3303                                 * because of the ack number.  TCP uses
3304                                 * seg_len == 0 to determine if it is a pure
3305                                 * ACK.  And this is not a pure ACK.
3306                                 */
3307                                seg_len = 0;
3308                                ofo_seg = B_TRUE;

3310                                if (tcps->tcps_reass_timeout != 0 &&
3311                                    tcp->tcp_reass_tid == 0) {
3312                                        tcp->tcp_reass_tid = TCP_TIMER(tcp,
3313                                            tcp_reass_timer,
3314                                            tcps->tcps_reass_timeout);
3315                                }
3316                        }
3317                }
3318        } else if (seg_len > 0) {
3319                TCPS_BUMP_MIB(tcps, tcpInDataInorderSegs);
3320                TCPS_UPDATE_MIB(tcps, tcpInDataInorderBytes, seg_len);
3321                /*
3322                 * If an out of order FIN was received before, and the seq
3323                 * num and len of the new segment match that of the FIN,
3324                 * put the FIN flag back in.
3325                 */
3326                if ((tcp->tcp_valid_bits & TCP_OFO_FIN_VALID) &&
3327                    seg_seq + seg_len == tcp->tcp_ofo_fin_seq) {
```



```
3328                        flags |= TH_FIN;
3329                        tcp->tcp_valid_bits &= ~TCP_OFO_FIN_VALID;
3330                }
3331        }
3332        if ((flags & (TH_RST | TH_SYN | TH_URG | TH_ACK)) != TH_ACK) {
3333        if (flags & TH_RST) {
3334                freemsg(mp);
3335                switch (tcp->tcp_state) {
3336                case TCPS_SYN_RCVD:
3337                        (void) tcp_clean_death(tcp, ECONNREFUSED);
3338                        break;
3339                case TCPS_ESTABLISHED:
3340                case TCPS_FIN_WAIT_1:
3341                case TCPS_FIN_WAIT_2:
3342                case TCPS_CLOSE_WAIT:
3343                        (void) tcp_clean_death(tcp, ECONNRESET);
3344                        break;
3345                case TCPS_CLOSING:
3346                case TCPS_LAST_ACK:
3347                        (void) tcp_clean_death(tcp, 0);
3348                        break;
3349                default:
3350                        ASSERT(tcp->tcp_state != TCPS_TIME_WAIT);
3351                        (void) tcp_clean_death(tcp, ENXIO);
3352                        break;
3353                }
3354                return;
3355        }
3356        if (flags & TH_SYN) {
3357                /*
3358                 * See RFC 793, Page 71
3359                 *
3360                 * The seq number must be in the window as it should
3361                 * be "fixed" above.  If it is outside window, it should
3362                 * be already rejected.  Note that we allow seg_seq to be
3363                 * rnxt + rwnd because we want to accept 0 window probe.
3364                 */
3365                ASSERT(SEQ_GEQ(seg_seq, tcp->tcp_rnxt) &&
3366                    SEQ_LEQ(seg_seq, tcp->tcp_rnxt + tcp->tcp_rwnd));
3367                freemsg(mp);
3368                /*
3369                 * If the ACK flag is not set, just use our snxt as the
3370                 * seq number of the RST segment.
3371                 */
3372                if (!(flags & TH_ACK)) {
3373                        seg_ack = tcp->tcp_snxt;
3374                }
3375                tcp_xmit_ctl("TH_SYN", tcp, seg_ack, seg_seq + 1,
3376                    TH_RST|TH_ACK);
3377                ASSERT(tcp->tcp_state != TCPS_TIME_WAIT);
3378                (void) tcp_clean_death(tcp, ECONNRESET);
3379                return;
3380        }
3381        /*
3382         * urp could be -1 when the urp field in the packet is 0
3383         * and TCP_OLD_URP_INTERPRETATION is set. This implies that the urgent
3384         * byte was at seg_seq - 1, in which case we ignore the urgent flag.
3385         */
3386        if (flags & TH_URG && urp >= 0) {
3387                if (!tcp->tcp_urp_last_valid ||
3388                    SEQ_GT(urp + seg_seq, tcp->tcp_urp_last)) {
3389                        /*
3390                         * Non-STREAMS sockets handle the urgent data a litte
3391                         * differently from STREAMS based sockets. There is no
3392                         * need to mark any mblks with the MSG{NOT,}MARKNEXT
3393                         * flags to keep SIOCATMARK happy. Instead a
```

```
3394                         * su_signal_oob upcall is made to update the mark.
3395                         * Neither is a T_EXDATA_IND mblk needed to be
3396                         * prepended to the urgent data. The urgent data is
3397                         * delivered using the su_recv upcall, where we set
3398                         * the MSG_OOB flag to indicate that it is urg data.
3399                         *
3400                         * Neither TH_SEND_URP_MARK nor TH_MARKNEXT_NEEDED
3401                         * are used by non-STREAMS sockets.
3402                         */
3403                        if (IPCL_IS_NONSTR(connp)) {
3404                                if (!TCP_IS_DETACHED(tcp)) {
3405                                        (*sockupcalls->su_signal_oob)
3406                                            (connp->conn_upper_handle, urp);
3407                                }
3408                        } else {
3409                                /*
3410                                 * If we haven't generated the signal yet for
3411                                 * this urgent pointer value, do it now.  Also,
3412                                 * send up a zero-length M_DATA indicating
3413                                 * whether or not this is the mark. The latter
3414                                 * is not needed when a T_EXDATA_IND is sent up.
3415                                 * However, if there are allocation failures
3416                                 * this code relies on the sender retransmitting
3417                                 * and the socket code for determining the mark
3418                                 * should not block waiting for the peer to
3419                                 * transmit. Thus, for simplicity we always
3420                                 * send up the mark indication.
3421                                 */
3422                                mp1 = allocb(0, BPRI_MED);
3423                                if (mp1 == NULL) {
3424                                        freemsg(mp);
3425                                        return;
3426                                }
3427                                if (!TCP_IS_DETACHED(tcp) &&
3428                                    !putnextctl1(connp->conn_rq, M_PCSIG,
3429                                    SIGURG)) {
3430                                        /* Try again on the rexmit. */
3431                                        freemsg(mp1);
3432                                        freemsg(mp);
3433                                        return;
3434                                }
3435                                /*
3436                                 * Mark with NOTMARKNEXT for now.
3437                                 * The code below will change this to MARKNEXT
3438                                 * if we are at the mark.
3439                                 *
3440                                 * If there are allocation failures (e.g. in
3441                                 * dupmsg below) the next time tcp_input_data
3442                                 * sees the urgent segment it will send up the
3443                                 * MSGMARKNEXT message.
3444                                 */
3445                                mp1->b_flag |= MSGNOTMARKNEXT;
3446                                freemsg(tcp->tcp_urp_mark_mp);
3447                                tcp->tcp_urp_mark_mp = mp1;
3448                                flags |= TH_SEND_URP_MARK;
3449 #ifdef DEBUG
3450                                (void) strlog(TCP_MOD_ID, 0, 1, SL_TRACE,
3451                                    "tcp_rput: sent M_PCSIG 2 seq %x urp %x "
3452                                    "last %x, %s",
3453                                    seg_seq, urp, tcp->tcp_urp_last,
3454                                    tcp_display(tcp, NULL, DISP_PORT_ONLY));
3455 #endif /* DEBUG */
3456                        }
3457                        tcp->tcp_urp_last_valid = B_TRUE;
3458                        tcp->tcp_urp_last = urp + seg_seq;
3459                } else if (tcp->tcp_urp_mark_mp != NULL) {
```



```
3460                        /*
3461                         * An allocation failure prevented the previous
3462                         * tcp_input_data from sending up the allocated
3463                         * MSG*MARKNEXT message - send it up this time
3464                         * around.
3465                         */
3466                        flags |= TH_SEND_URP_MARK;
3467                }
3468
3469                /*
3470                 * If the urgent byte is in this segment, make sure that it is
3471                 * all by itself.  This makes it much easier to deal with the
3472                 * possibility of an allocation failure on the T_exdata_ind.
3473                 * Note that seg_len is the number of bytes in the segment, and
3474                 * urp is the offset into the segment of the urgent byte.
3475                 * urp < seg_len means that the urgent byte is in this segment.
3476                 */
3477                if (urp < seg_len) {
3478                        if (seg_len != 1) {
3479                                uint32_t  tmp_rnxt;
3480                                /*
3481                                 * Break it up and feed it back in.
3482                                 * Re-attach the IP header.
3483                                 */
3484                                mp->b_rptr = iphdr;
3485                                if (urp > 0) {
3486                                        /*
3487                                         * There is stuff before the urgent
3488                                         * byte.
3489                                         */
3490                                        mp1 = dupmsg(mp);
3491                                        if (!mp1) {
3492                                                /*
3493                                                 * Trim from urgent byte on.
3494                                                 * The rest will come back.
3495                                                 */
3496                                                (void) adjmsg(mp,
3497                                                    urp - seg_len);
3498                                                tcp_input_data(connp,
3499                                                    mp, NULL, ira);
3500                                                return;
3501                                        }
3502                                        (void) adjmsg(mp1, urp - seg_len);
3503                                        /* Feed this piece back in. */
3504                                        tmp_rnxt = tcp->tcp_rnxt;
3505                                        tcp_input_data(connp, mp1, NULL, ira);
3506                                        /*
3507                                         * If the data passed back in was not
3508                                         * processed (ie: bad ACK) sending
3509                                         * the remainder back in will cause a
3510                                         * loop. In this case, drop the
3511                                         * packet and let the sender try
3512                                         * sending a good packet.
3513                                         */
3514                                        if (tmp_rnxt == tcp->tcp_rnxt) {
3515                                                freemsg(mp);
3516                                                return;
3517                                        }
3518                                }
3519                                if (urp != seg_len - 1) {
3520                                        uint32_t  tmp_rnxt;
3521                                        /*
3522                                         * There is stuff after the urgent
3523                                         * byte.
3524                                         */
3525                                        mp1 = dupmsg(mp);
```

```
3526                                    if (!mp1) {
3527                                            /*
3528                                             * Trim everything beyond the
3529                                             * urgent byte.  The rest will
3530                                             * come back.
3531                                             */
3532                                            (void) adjmsg(mp,
3533                                                urp + 1 - seg_len);
3534                                            tcp_input_data(connp,
3535                                                mp, NULL, ira);
3536                                            return;
3537                                    }
3538                                    (void) adjmsg(mp1, urp + 1 - seg_len);
3539                                    tmp_rnxt = tcp->tcp_rnxt;
3540                                    tcp_input_data(connp, mp1, NULL, ira);
3541                                    /*
3542                                     * If the data passed back in was not
3543                                     * processed (ie: bad ACK) sending
3544                                     * the remainder back in will cause a
3545                                     * loop. In this case, drop the
3546                                     * packet and let the sender try
3547                                     * sending a good packet.
3548                                     */
3549                                    if (tmp_rnxt == tcp->tcp_rnxt) {
3550                                            freemsg(mp);
3551                                            return;
3552                                    }
3553                            }
3554                            tcp_input_data(connp, mp, NULL, ira);
3555                            return;
3556                    }
3557                    /*
3558                     * This segment contains only the urgent byte.  We
3559                     * have to allocate the T_exdata_ind, if we can.
3560                     */
3561                    if (IPCL_IS_NONSTR(connp)) {
3562                            int error;

3564                            (*sockupcalls->su_recv)
3565                                (connp->conn_upper_handle, mp, seg_len,
3566                                MSG_OOB, &error, NULL);
3567                            /*
3568                             * We should never be in middle of a
3569                             * fallback, the squeue guarantees that.
3570                             */
3571                            ASSERT(error != EOPNOTSUPP);
3572                            mp = NULL;
3573                            goto update_ack;
3574                    } else if (!tcp->tcp_urp_mp) {
3575                            struct T_exdata_ind *tei;
3576                            mp1 = allocb(sizeof (struct T_exdata_ind),
3577                                BPRI_MED);
3578                            if (!mp1) {
3579                                    /*
3580                                     * Sigh... It'll be back.
3581                                     * Generate any MSG*MARK message now.
3582                                     */
3583                                    freemsg(mp);
3584                                    seg_len = 0;
3585                                    if (flags & TH_SEND_URP_MARK) {


3588                                            ASSERT(tcp->tcp_urp_mark_mp);
3589                                            tcp->tcp_urp_mark_mp->b_flag &=
3590                                                ~MSGNOTMARKNEXT;
3591                                            tcp->tcp_urp_mark_mp->b_flag |=
```



```
3592                                                    MSGMARKNEXT;
3593                                            }
3594                                            goto ack_check;
3595                                    }
3596                                    mp1->b_datap->db_type = M_PROTO;
3597                                    tei = (struct T_exdata_ind *)mp1->b_rptr;
3598                                    tei->PRIM_type = T_EXDATA_IND;
3599                                    tei->MORE_flag = 0;
3600                                    mp1->b_wptr = (uchar_t *)&tei[1];
3601                                    tcp->tcp_urp_mp = mp1;
3602 #ifdef DEBUG
3603                                    (void) strlog(TCP_MOD_ID, 0, 1, SL_TRACE,
3604                                        "tcp_rput: allocated exdata_ind %s",
3605                                        tcp_display(tcp, NULL,
3606                                        DISP_PORT_ONLY));
3607 #endif /* DEBUG */
3608                                    /*
3609                                     * There is no need to send a separate MSG*MARK
3610                                     * message since the T_EXDATA_IND will be sent
3611                                     * now.
3612                                     */
3613                                    flags &= ~TH_SEND_URP_MARK;
3614                                    freemsg(tcp->tcp_urp_mark_mp);
3615                                    tcp->tcp_urp_mark_mp = NULL;
3616                            }
3617                            /*
3618                             * Now we are all set.  On the next putnext upstream,
3619                             * tcp_urp_mp will be non-NULL and will get prepended
3620                             * to what has to be this piece containing the urgent
3621                             * byte.  If for any reason we abort this segment below,
3622                             * if it comes back, we will have this ready, or it
3623                             * will get blown off in close.
3624                             */
3625                    } else if (urp == seg_len) {
3626                            /*
3627                             * The urgent byte is the next byte after this sequence
3628                             * number. If this endpoint is non-STREAMS, then there
3629                             * is nothing to do here since the socket has already
3630                             * been notified about the urg pointer by the
3631                             * su_signal_oob call above.
3632                             *
3633                             * In case of STREAMS, some more work might be needed.
3634                             * If there is data it is marked with MSGMARKNEXT and
3635                             * and any tcp_urp_mark_mp is discarded since it is not
3636                             * needed. Otherwise, if the code above just allocated
3637                             * a zero-length tcp_urp_mark_mp message, that message
3638                             * is tagged with MSGMARKNEXT. Sending up these
3639                             * MSGMARKNEXT messages makes SIOCATMARK work correctly
3640                             * even though the T_EXDATA_IND will not be sent up
3641                             * until the urgent byte arrives.
3642                             */
3643                            if (!IPCL_IS_NONSTR(tcp->tcp_connp)) {
3644                                    if (seg_len != 0) {
3645                                            flags |= TH_MARKNEXT_NEEDED;
3646                                            freemsg(tcp->tcp_urp_mark_mp);
3647                                            tcp->tcp_urp_mark_mp = NULL;
3648                                            flags &= ~TH_SEND_URP_MARK;
3649                                    } else if (tcp->tcp_urp_mark_mp != NULL) {
3650                                            flags |= TH_SEND_URP_MARK;
3651                                            tcp->tcp_urp_mark_mp->b_flag &=
3652                                                ~MSGNOTMARKNEXT;
3653                                            tcp->tcp_urp_mark_mp->b_flag |=
3654                                                MSGMARKNEXT;
3655                                    }
3656                            }
3657 #ifdef DEBUG
```

```
3658                        (void) strlog(TCP_MOD_ID, 0, 1, SL_TRACE,
3659                            "tcp_rput: AT MARK, len %d, flags 0x%x, %s",
3660                            seg_len, flags,
3661                            tcp_display(tcp, NULL, DISP_PORT_ONLY));
3662 #endif /* DEBUG */
3663                 }
3664 #ifdef DEBUG
3665                 else {
3666                         /* Data left until we hit mark */
3667                         (void) strlog(TCP_MOD_ID, 0, 1, SL_TRACE,
3668                             "tcp_rput: URP %d bytes left, %s",
3669                             urp - seg_len, tcp_display(tcp, NULL,
3670                             DISP_PORT_ONLY));
3671                 }
3672 #endif /* DEBUG */
3673         }

3675 process_ack:
3676         if (!(flags & TH_ACK)) {
3677                 freemsg(mp);
3678                 goto xmit_check;
3679         }
3680         }
3681         bytes_acked = (int)(seg_ack - tcp->tcp_suna);

3683         if (bytes_acked > 0)
3684                 tcp->tcp_ip_forward_progress = B_TRUE;
3685         if (tcp->tcp_state == TCPS_SYN_RCVD) {
3686                 /*
3687                  * tcp_sendmsg() checks tcp_state without entering
3688                  * the squeue so tcp_state should be updated before
3689                  * sending up a connection confirmation or a new
3690                  * connection indication.
3691                  */
3692                 tcp->tcp_state = TCPS_ESTABLISHED;

3694                 /*
3695                  * We are seeing the final ack in the three way
3696                  * hand shake of a active open'ed connection
3697                  * so we must send up a T_CONN_CON
3698                  */
3699                 if (tcp->tcp_active_open) {
3700                         if (!tcp_conn_con(tcp, iphdr, mp, NULL, ira)) {
3701                                 freemsg(mp);
3702                                 tcp->tcp_state = TCPS_SYN_RCVD;
3703                                 return;
3704                         }
3705                         /*
3706                          * Don't fuse the loopback endpoints for
3707                          * simultaneous active opens.
3708                          */
3709                         if (tcp->tcp_loopback) {
3710                                 TCP_STAT(tcps, tcp_fusion_unfusable);
3711                                 tcp->tcp_unfusable = B_TRUE;
3712                         }
3713                         /*
3714                          * For simultaneous active open, trace receipt of final
3715                          * ACK as tcp:::connect-established.
3716                          */
3717                         DTRACE_TCP5(connect__established, mblk_t *, NULL,
3718                             ip_xmit_attr_t *, connp->conn_ixa, void_ip_t *,
3719                             iphdr, tcp_t *, tcp, tcph_t *, tcpha);
3720                 } else if (IPCL_IS_NONSTR(connp)) {
3721                         /*
3722                          * 3-way handshake has completed, so notify socket
3723                          * of the new connection.
```



```
3724                          *
3725                          * We are here means eager is fine but it can
3726                          * get a TH_RST at any point between now and till
3727                          * accept completes and disappear. We need to
3728                          * ensure that reference to eager is valid after
3729                          * we get out of eager's perimeter. So we do
3730                          * an extra refhold.
3731                          */
3732                         CONN_INC_REF(connp);

3734                         if (!tcp_newconn_notify(tcp, ira)) {
3735                                 /*
3736                                  * The state-change probe for SYN_RCVD ->
3737                                  * ESTABLISHED has not fired yet. We reset
3738                                  * the state to SYN_RCVD so that future
3739                                  * state-change probes report correct state
3740                                  * transistions.
3741                                  */
3742                                 tcp->tcp_state = TCPS_SYN_RCVD;
3743                                 freemsg(mp);
3744                                 /* notification did not go up, so drop ref */
3745                                 CONN_DEC_REF(connp);
3746                                 /* ... and close the eager */
3747                                 ASSERT(TCP_IS_DETACHED(tcp));
3748                                 (void) tcp_close_detached(tcp);
3749                                 return;
3750                         }
3751                         /*
3752                          * tcp_newconn_notify() changes conn_upcalls and
3753                          * connp->conn_upper_handle.  Fix things now, in case
3754                          * there's data attached to this ack.
3755                          */
3756                         if (connp->conn_upcalls != NULL)
3757                                 sockupcalls = connp->conn_upcalls;
3758                         /*
3759                          * For passive open, trace receipt of final ACK as
3760                          * tcp:::accept-established.
3761                          */
3762                         DTRACE_TCP5(accept__established, mblk_t *, NULL,
3763                             ip_xmit_attr_t *, connp->conn_ixa, void_ip_t *,
3764                             iphdr, tcp_t *, tcp, tcph_t *, tcpha);
3765                 } else {
3766                         /*
3767                          * 3-way handshake complete - this is a STREAMS based
3768                          * socket, so pass up the T_CONN_IND.
3769                          */
3770                         tcp_t   *listener = tcp->tcp_listener;
3771                         mblk_t  *mp = tcp->tcp_conn.tcp_eager_conn_ind;

3773                         tcp->tcp_tconnind_started = B_TRUE;
3774                         tcp->tcp_conn.tcp_eager_conn_ind = NULL;
3775                         ASSERT(mp != NULL);
3776                         /*
3777                          * We are here means eager is fine but it can
3778                          * get a TH_RST at any point between now and till
3779                          * accept completes and disappear. We need to
3780                          * ensure that reference to eager is valid after
3781                          * we get out of eager's perimeter. So we do
3782                          * an extra refhold.
3783                          */
3784                         CONN_INC_REF(connp);

3786                         /*
3787                          * The listener also exists because of the refhold
3788                          * done in tcp_input_listener. Its possible that it
3789                          * might have closed. We will check that once we
```

```
3790                         * get inside listeners context.
3791                         */
3792                        CONN_INC_REF(listener->tcp_connp);
3793                        if (listener->tcp_connp->conn_sqp ==
3794                            connp->conn_sqp) {
3795                                /*
3796                                 * We optimize by not calling an SQUEUE_ENTER
3797                                 * on the listener since we know that the
3798                                 * listener and eager squeues are the same.
3799                                 * We are able to make this check safely only
3800                                 * because neither the eager nor the listener
3801                                 * can change its squeue. Only an active connect
3802                                 * can change its squeue
3803                                 */
3804                                tcp_send_conn_ind(listener->tcp_connp, mp,
3805                                    listener->tcp_connp->conn_sqp);
3806                                CONN_DEC_REF(listener->tcp_connp);
3807                        } else if (!tcp->tcp_loopback) {
3808                                SQUEUE_ENTER_ONE(listener->tcp_connp->conn_sqp,
3809                                    mp, tcp_send_conn_ind,
3810                                    listener->tcp_connp, NULL, SQ_FILL,
3811                                    SQTAG_TCP_CONN_IND);
3812                        } else {
3813                                SQUEUE_ENTER_ONE(listener->tcp_connp->conn_sqp,
3814                                    mp, tcp_send_conn_ind,
3815                                    listener->tcp_connp, NULL, SQ_NODRAIN,
3816                                    SQTAG_TCP_CONN_IND);
3817                        }
3818                        /*
3819                         * For passive open, trace receipt of final ACK as
3820                         * tcp:::accept-established.
3821                         */
3822                        DTRACE_TCP5(accept__established, mlbk_t *, NULL,
3823                            ip_xmit_attr_t *, connp->conn_ixa, void_ip_t *,
3824                            iphdr, tcp_t *, tcp, tcph_t *, tcpha);
3825                }
3826                TCPS_CONN_INC(tcps);

3828                tcp->tcp_suna = tcp->tcp_iss + 1;       /* One for the SYN */
3829                bytes_acked--;
3830                /* SYN was acked - making progress */
3831                tcp->tcp_ip_forward_progress = B_TRUE;

3833                /*
3834                 * If SYN was retransmitted, need to reset all
3835                 * retransmission info as this segment will be
3836                 * treated as a dup ACK.
3837                 */
3838                if (tcp->tcp_rexmit) {
3839                        tcp->tcp_rexmit = B_FALSE;
3840                        tcp->tcp_rexmit_nxt = tcp->tcp_snxt;
3841                        tcp->tcp_rexmit_max = tcp->tcp_snxt;
3843                        tcp->tcp_snd_burst = tcp->tcp_localnet ?
3844                            TCP_CWND_INFINITE : TCP_CWND_NORMAL;
3842                        tcp->tcp_ms_we_have_waited = 0;
3843                        tcp->tcp_cwnd = mss;
3844                }

3846                /*
3847                 * We set the send window to zero here.
3848                 * This is needed if there is data to be
3849                 * processed already on the queue.
3850                 * Later (at swnd_update label), the
3851                 * "new_swnd > tcp_swnd" condition is satisfied
3852                 * the XMIT_NEEDED flag is set in the current
3853                 * (SYN_RCVD) state. This ensures tcp_wput_data() is
```



```
3854                 * called if there is already data on queue in
3855                 * this state.
3856                 */
3857                tcp->tcp_swnd = 0;

3859                if (new_swnd > tcp->tcp_max_swnd)
3860                        tcp->tcp_max_swnd = new_swnd;
3861                tcp->tcp_swl1 = seg_seq;
3862                tcp->tcp_swl2 = seg_ack;
3863                tcp->tcp_valid_bits &= ~TCP_ISS_VALID;

3865                /* Trace change from SYN_RCVD -> ESTABLISHED here */
3866                DTRACE_TCP6(state__change, void, NULL, ip_xmit_attr_t *,
3867                    connp->conn_ixa, void, NULL, tcp_t *, tcp, void, NULL,
3868                    int32_t, TCPS_SYN_RCVD);

3870                /* Fuse when both sides are in ESTABLISHED state */
3871                if (tcp->tcp_loopback && do_tcp_fusion)
3872                        tcp_fuse(tcp, iphdr, tcpha);

3874        }
3875        /* This code follows 4.4BSD-Lite2 mostly. */
3876        if (bytes_acked < 0)
3877                goto est;

3879        /*
3880         * If TCP is ECN capable and the congestion experience bit is
3881         * set, reduce tcp_cwnd and tcp_ssthresh.  But this should only be
3882         * done once per window (or more loosely, per RTT).
3883         */
3884        if (tcp->tcp_cwr && SEQ_GT(seg_ack, tcp->tcp_cwr_snd_max))
3885                tcp->tcp_cwr = B_FALSE;
3886        if (tcp->tcp_ecn_ok && (flags & TH_ECE)) {
3887                if (!tcp->tcp_cwr) {
3888                        npkt = ((tcp->tcp_snxt - tcp->tcp_suna) >> 1) / mss;
3889                        tcp->tcp_cwnd_ssthresh = MAX(npkt, 2) * mss;
3890                        tcp->tcp_cwnd = npkt * mss;
3891                        /*
3892                         * If the cwnd is 0, use the timer to clock out
3893                         * new segments.  This is required by the ECN spec.
3894                         */
3895                        if (npkt == 0) {
3896                                TCP_TIMER_RESTART(tcp, tcp->tcp_rto);
3897                                /*
3898                                 * This makes sure that when the ACK comes
3899                                 * back, we will increase tcp_cwnd by 1 MSS.
3900                                 */
3901                                tcp->tcp_cwnd_cnt = 0;
3902                        }
3903                        tcp->tcp_cwr = B_TRUE;
3904                        /*
3905                         * This marks the end of the current window of in
3906                         * flight data.  That is why we don't use
3907                         * tcp_suna + tcp_swnd.  Only data in flight can
3908                         * provide ECN info.
3909                         */
3910                        tcp->tcp_cwr_snd_max = tcp->tcp_snxt;
3911                        tcp->tcp_ecn_cwr_sent = B_FALSE;
3912                }
3913        }

3915        mp1 = tcp->tcp_xmit_head;
3916        if (bytes_acked == 0) {
3917                if (!ofo_seg && seg_len == 0 && new_swnd == tcp->tcp_swnd) {
3918                        int dupack_cnt;
```

```
3920                        TCPS_BUMP_MIB(tcps, tcpInDupAck);
3921                        /*
3922                         * Fast retransmit.  When we have seen exactly three
3923                         * identical ACKs while we have unacked data
3924                         * outstanding we take it as a hint that our peer
3925                         * dropped something.
3926                         *
3927                         * If TCP is retransmitting, don't do fast retransmit.
3928                         */
3929                        if (mp1 && tcp->tcp_suna != tcp->tcp_snxt &&
3930                            ! tcp->tcp_rexmit) {
3931                                /* Do Limited Transmit */
3932                                if ((dupack_cnt = ++tcp->tcp_dupack_cnt) <
3933                                    tcps->tcps_dupack_fast_retransmit) {
3934                                        /*
3935                                         * RFC 3042
3936                                         *
3937                                         * What we need to do is temporarily
3938                                         * increase tcp_cwnd so that new
3939                                         * data can be sent if it is allowed
3940                                         * by the receive window (tcp_rwnd).
3941                                         * tcp_wput_data() will take care of
3942                                         * the rest.
3943                                         *
3944                                         * If the connection is SACK capable,
3945                                         * only do limited xmit when there
3946                                         * is SACK info.
3947                                         *
3948                                         * Note how tcp_cwnd is incremented.
3949                                         * The first dup ACK will increase
3950                                         * it by 1 MSS.  The second dup ACK
3951                                         * will increase it by 2 MSS.  This
3952                                         * means that only 1 new segment will
3953                                         * be sent for each dup ACK.
3954                                         */
3955                                        if (tcp->tcp_unsent > 0 &&
3956                                            (!tcp->tcp_snd_sack_ok ||
3957                                            (tcp->tcp_snd_sack_ok &&
3958                                            tcp->tcp_notsack_list != NULL))) {
3959                                                tcp->tcp_cwnd += mss <<
3960                                                    (tcp->tcp_dupack_cnt - 1);
3961                                                flags |= TH_LIMIT_XMIT;
3962                                        }
3963                                } else if (dupack_cnt ==
3964                                    tcps->tcps_dupack_fast_retransmit) {

3966                                /*
3967                                 * If we have reduced tcp_ssthresh
3968                                 * because of ECN, do not reduce it again
3969                                 * unless it is already one window of data
3970                                 * away.  After one window of data, tcp_cwr
3971                                 * should then be cleared.  Note that
3972                                 * for non ECN capable connection, tcp_cwr
3973                                 * should always be false.
3974                                 *
3975                                 * Adjust cwnd since the duplicate
3976                                 * ack indicates that a packet was
3977                                 * dropped (due to congestion.)
3978                                 */
3979                                if (!tcp->tcp_cwr) {
3980                                        npkt = ((tcp->tcp_snxt -
3981                                            tcp->tcp_suna) >> 1) / mss;
3982                                        tcp->tcp_cwnd_ssthresh = MAX(npkt, 2) *
3983                                            mss;
3984                                        tcp->tcp_cwnd = (npkt +
3985                                            tcp->tcp_dupack_cnt) * mss;
```



```
3986                                }
3987                                if (tcp->tcp_ecn_ok) {
3988                                        tcp->tcp_cwr = B_TRUE;
3989                                        tcp->tcp_cwr_snd_max = tcp->tcp_snxt;
3990                                        tcp->tcp_ecn_cwr_sent = B_FALSE;
3991                                }

3993                                /*
3994                                 * We do Hoe's algorithm.  Refer to her
3995                                 * paper "Improving the Start-up Behavior
3996                                 * of a Congestion Control Scheme for TCP,"
3997                                 * appeared in SIGCOMM'96.
3998                                 *
3999                                 * Save highest seq no we have sent so far.
4000                                 * Be careful about the invisible FIN byte.
4001                                 */
4002                                if ((tcp->tcp_valid_bits & TCP_FSS_VALID) &&
4003                                    (tcp->tcp_unsent == 0)) {
4004                                        tcp->tcp_rexmit_max = tcp->tcp_fss;
4005                                } else {
4006                                        tcp->tcp_rexmit_max = tcp->tcp_snxt;
4007                                }

4009                                /*
4013                                 * Do not allow bursty traffic during.
4014                                 * fast recovery.  Refer to Fall and Floyd's
4015                                 * paper "Simulation-based Comparisons of
4016                                 * Tahoe, Reno and SACK TCP" (in CCR?)
4017                                 * This is a best current practise.
4018                                 */
4019                                tcp->tcp_snd_burst = TCP_CWND_SS;

4021                                /*
4010                                 * For SACK:
4011                                 * Calculate tcp_pipe, which is the
4012                                 * estimated number of bytes in
4013                                 * network.
4014                                 *
4015                                 * tcp_fack is the highest sack'ed seq num
4016                                 * TCP has received.
4017                                 *
4018                                 * tcp_pipe is explained in the above quoted
4019                                 * Fall and Floyd's paper.  tcp_fack is
4020                                 * explained in Mathis and Mahdavi's
4021                                 * "Forward Acknowledgment: Refining TCP
4022                                 * Congestion Control" in SIGCOMM '96.
4023                                 */
4024                                if (tcp->tcp_snd_sack_ok) {
4025                                        if (tcp->tcp_notsack_list != NULL) {
4026                                                tcp->tcp_pipe = tcp->tcp_snxt -
4027                                                    tcp->tcp_fack;
4028                                                tcp->tcp_sack_snxt = seg_ack;
4029                                                flags |= TH_NEED_SACK_REXMIT;
4030                                        } else {
4031                                                /*
4032                                                 * Always initialize tcp_pipe
4033                                                 * even though we don't have
4034                                                 * any SACK info.  If later
4035                                                 * we get SACK info and
4036                                                 * tcp_pipe is not initialized,
4037                                                 * funny things will happen.
4038                                                 */
4039                                                tcp->tcp_pipe =
4040                                                    tcp->tcp_cwnd_ssthresh;
4041                                        }
4042                                } else {
```

```
4043                                    flags |= TH_REXMIT_NEEDED;
4044                            } /* tcp_snd_sack_ok */

4046                            } else {
4047                                    /*
4048                                     * Here we perform congestion
4049                                     * avoidance, but NOT slow start.
4050                                     * This is known as the Fast
4051                                     * Recovery Algorithm.
4052                                     */
4053                                    if (tcp->tcp_snd_sack_ok &&
4054                                        tcp->tcp_notsack_list != NULL) {
4055                                            flags |= TH_NEED_SACK_REXMIT;
4056                                            tcp->tcp_pipe -= mss;
4057                                            if (tcp->tcp_pipe < 0)
4058                                                    tcp->tcp_pipe = 0;
4059                                    } else {
4060                                    /*
4061                                     * We know that one more packet has
4062                                     * left the pipe thus we can update
4063                                     * cwnd.
4064                                     */
4065                                    cwnd = tcp->tcp_cwnd + mss;
4066                                    if (cwnd > tcp->tcp_cwnd_max)
4067                                            cwnd = tcp->tcp_cwnd_max;
4068                                    tcp->tcp_cwnd = cwnd;
4069                                    if (tcp->tcp_unsent > 0)
4070                                            flags |= TH_XMIT_NEEDED;
4071                                    }
4072                            }
4073                    }
4074            } else if (tcp->tcp_zero_win_probe) {
4075                    /*
4076                     * If the window has opened, need to arrange
4077                     * to send additional data.
4078                     */
4079                    if (new_swnd != 0) {
4080                            /* tcp_suna != tcp_snxt */
4081                            /* Packet contains a window update */
4082                            TCPS_BUMP_MIB(tcps, tcpInWinUpdate);
4083                            tcp->tcp_zero_win_probe = 0;
4084                            tcp->tcp_timer_backoff = 0;
4085                            tcp->tcp_ms_we_have_waited = 0;

4087                            /*
4088                             * Transmit starting with tcp_suna since
4089                             * the one byte probe is not ack'ed.
4090                             * If TCP has sent more than one identical
4091                             * probe, tcp_rexmit will be set.  That means
4092                             * tcp_ss_rexmit() will send out the one
4093                             * byte along with new data.  Otherwise,
4094                             * fake the retransmission.
4095                             */
4096                            flags |= TH_XMIT_NEEDED;
4097                            if (!tcp->tcp_rexmit) {
4098                                    tcp->tcp_rexmit = B_TRUE;
4099                                    tcp->tcp_dupack_cnt = 0;
4100                                    tcp->tcp_rexmit_nxt = tcp->tcp_suna;
4101                                    tcp->tcp_rexmit_max = tcp->tcp_suna + 1;
4102                            }
4103                    }
4104            }
4105            goto swnd_update;
4106    }

4108    /*
```



```
4109     * Check for "acceptability" of ACK value per RFC 793, pages 72 - 73.
4110     * If the ACK value acks something that we have not yet sent, it might
4111     * be an old duplicate segment.  Send an ACK to re-synchronize the
4112     * other side.
4113     * Note: reset in response to unacceptable ACK in SYN_RECEIVE
4114     * state is handled above, so we can always just drop the segment and
4115     * send an ACK here.
4116     *
4117     * In the case where the peer shrinks the window, we see the new window
4118     * update, but all the data sent previously is queued up by the peer.
4119     * To account for this, in tcp_process_shrunk_swnd(), the sequence
4120     * number, which was already sent, and within window, is recorded.
4121     * tcp_snxt is then updated.
4122     *
4123     * If the window has previously shrunk, and an ACK for data not yet
4124     * sent, according to tcp_snxt is recieved, it may still be valid. If
4125     * the ACK is for data within the window at the time the window was
4126     * shrunk, then the ACK is acceptable. In this case tcp_snxt is set to
4127     * the sequence number ACK'ed.
4128     *
4129     * If the ACK covers all the data sent at the time the window was
4130     * shrunk, we can now set tcp_is_wnd_shrnk to B_FALSE.
4131     *
4132     * Should we send ACKs in response to ACK only segments?
4133     */

4135    if (SEQ_GT(seg_ack, tcp->tcp_snxt)) {
4136            if ((tcp->tcp_is_wnd_shrnk) &&
4137                (SEQ_LEQ(seg_ack, tcp->tcp_snxt_shrunk))) {
4138                    uint32_t data_acked_ahead_snxt;

4140                    data_acked_ahead_snxt = seg_ack - tcp->tcp_snxt;
4141                    tcp_update_xmit_tail(tcp, seg_ack);
4142                    tcp->tcp_unsent -= data_acked_ahead_snxt;
4143            } else {
4144                    TCPS_BUMP_MIB(tcps, tcpInAckUnsent);
4145                    /* drop the received segment */
4146                    freemsg(mp);

4148                    /*
4149                     * Send back an ACK.  If tcp_drop_ack_unsent_cnt is
4150                     * greater than 0, check if the number of such
4151                     * bogus ACKs is greater than that count.  If yes,
4152                     * don't send back any ACK.  This prevents TCP from
4153                     * getting into an ACK storm if somehow an attacker
4154                     * successfully spoofs an acceptable segment to our
4155                     * peer.  If this continues (count > 2 X threshold),
4156                     * we should abort this connection.
4157                     */
4158                    if (tcp_drop_ack_unsent_cnt > 0 &&
4159                        ++tcp->tcp_in_ack_unsent >
4160                        tcp_drop_ack_unsent_cnt) {
4161                            TCP_STAT(tcps, tcp_in_ack_unsent_drop);
4162                            if (tcp->tcp_in_ack_unsent > 2 *
4163                                tcp_drop_ack_unsent_cnt) {
4164                                    (void) tcp_clean_death(tcp, EPROTO);
4165                            }
4166                            return;
4167                    }
4168                    mp = tcp_ack_mp(tcp);
4169                    if (mp != NULL) {
4170                            BUMP_LOCAL(tcp->tcp_obsegs);
4171                            TCPS_BUMP_MIB(tcps, tcpOutAck);
4172                            tcp_send_data(tcp, mp);
4173                    }
4174                    return;
```

```
4175                }
4176        } else if (tcp->tcp_is_wnd_shrnk && SEQ_GEQ(seg_ack,
4177            tcp->tcp_snxt_shrunk)) {
4178                    tcp->tcp_is_wnd_shrnk = B_FALSE;
4179        }

4181        /*
4182         * TCP gets a new ACK, update the notsack'ed list to delete those
4183         * blocks that are covered by this ACK.
4184         */
4185        if (tcp->tcp_snd_sack_ok && tcp->tcp_notsack_list != NULL) {
4186                tcp_notsack_remove(&(tcp->tcp_notsack_list), seg_ack,
4187                    &(tcp->tcp_num_notsack_blk), &(tcp->tcp_cnt_notsack_list));
4188        }

4190        /*
4191         * If we got an ACK after fast retransmit, check to see
4192         * if it is a partial ACK.  If it is not and the congestion
4193         * window was inflated to account for the other side's
4194         * cached packets, retract it.  If it is, do Hoe's algorithm.
4195         */
4196        if (tcp->tcp_dupack_cnt >= tcps->tcps_dupack_fast_retransmit) {
4197                ASSERT(tcp->tcp_rexmit == B_FALSE);
4198                if (SEQ_GEQ(seg_ack, tcp->tcp_rexmit_max)) {
4199                        tcp->tcp_dupack_cnt = 0;
4200                        /*
4201                         * Restore the orig tcp_cwnd_ssthresh after
4202                         * fast retransmit phase.
4203                         */
4204                        if (tcp->tcp_cwnd > tcp->tcp_cwnd_ssthresh) {
4205                                tcp->tcp_cwnd = tcp->tcp_cwnd_ssthresh;
4206                        }
4207                        tcp->tcp_rexmit_max = seg_ack;
4208                        tcp->tcp_cwnd_cnt = 0;
4221                        tcp->tcp_snd_burst = tcp->tcp_localnet ?
4222                            TCP_CWND_INFINITE : TCP_CWND_NORMAL;

4210                        /*
4211                         * Remove all notsack info to avoid confusion with
4212                         * the next fast retrasnmit/recovery phase.
4213                         */
4214                        if (tcp->tcp_snd_sack_ok) {
4215                                TCP_NOTSACK_REMOVE_ALL(tcp->tcp_notsack_list,
4216                                    tcp);
4217                        }
4218                } else {
4219                        if (tcp->tcp_snd_sack_ok &&
4220                            tcp->tcp_notsack_list != NULL) {
4221                                flags |= TH_NEED_SACK_REXMIT;
4222                                tcp->tcp_pipe -= mss;
4223                                if (tcp->tcp_pipe < 0)
4224                                        tcp->tcp_pipe = 0;
4225                        } else {
4226                                /*
4227                                 * Hoe's algorithm:
4228                                 *
4229                                 * Retransmit the unack'ed segment and
4230                                 * restart fast recovery.  Note that we
4231                                 * need to scale back tcp_cwnd to the
4232                                 * original value when we started fast
4233                                 * recovery.  This is to prevent overly
4234                                 * aggressive behaviour in sending new
4235                                 * segments.
4236                                 */
4237                                tcp->tcp_cwnd = tcp->tcp_cwnd_ssthresh +
4238                                    tcps->tcps_dupack_fast_retransmit * mss;
```



```
4239                                tcp->tcp_cwnd_cnt = tcp->tcp_cwnd;
4240                                flags |= TH_REXMIT_NEEDED;
4241                        }
4242                }
4243        } else {
4244                tcp->tcp_dupack_cnt = 0;
4245                if (tcp->tcp_rexmit) {
4246                        /*
4247                         * TCP is retranmitting.  If the ACK ack's all
4248                         * outstanding data, update tcp_rexmit_max and
4249                         * tcp_rexmit_nxt.  Otherwise, update tcp_rexmit_nxt
4250                         * to the correct value.
4251                         *
4252                         * Note that SEQ_LEQ() is used.  This is to avoid
4253                         * unnecessary fast retransmit caused by dup ACKs
4254                         * received when TCP does slow start retransmission
4255                         * after a time out.  During this phase, TCP may
4256                         * send out segments which are already received.
4257                         * This causes dup ACKs to be sent back.
4258                         */
4259                        if (SEQ_LEQ(seg_ack, tcp->tcp_rexmit_max)) {
4260                                if (SEQ_GT(seg_ack, tcp->tcp_rexmit_nxt)) {
4261                                        tcp->tcp_rexmit_nxt = seg_ack;
4262                                }
4263                                if (seg_ack != tcp->tcp_rexmit_max) {
4264                                        flags |= TH_XMIT_NEEDED;
4265                                }
4266                        } else {
4267                                tcp->tcp_rexmit = B_FALSE;
4268                                tcp->tcp_rexmit_nxt = tcp->tcp_snxt;
4283                                tcp->tcp_snd_burst = tcp->tcp_localnet ?
4284                                    TCP_CWND_INFINITE : TCP_CWND_NORMAL;
4269                        }
4270                        tcp->tcp_ms_we_have_waited = 0;
4271                }
4272        }

4274        TCPS_BUMP_MIB(tcps, tcpInAckSegs);
4275        TCPS_UPDATE_MIB(tcps, tcpInAckBytes, bytes_acked);
4276        tcp->tcp_suna = seg_ack;
4277        if (tcp->tcp_zero_win_probe != 0) {
4278                tcp->tcp_zero_win_probe = 0;
4279                tcp->tcp_timer_backoff = 0;
4280        }

4282        /*
4283         * If tcp_xmit_head is NULL, then it must be the FIN being ack'ed.
4284         * Note that it cannot be the SYN being ack'ed.  The code flow
4285         * will not reach here.
4286         */
4287        if (mp1 == NULL) {
4288                goto fin_acked;
4289        }

4291        /*
4292         * Update the congestion window.
4293         *
4294         * If TCP is not ECN capable or TCP is ECN capable but the
4295         * congestion experience bit is not set, increase the tcp_cwnd as
4296         * usual.
4297         */
4298        if (!tcp->tcp_ecn_ok || !(flags & TH_ECE)) {
4299                cwnd = tcp->tcp_cwnd;
4300                add = mss;

4302                if (cwnd >= tcp->tcp_cwnd_ssthresh) {
```

```
4303                        /*
4304                         * This is to prevent an increase of less than 1 MSS of
4305                         * tcp_cwnd.  With partial increase, tcp_wput_data()
4306                         * may send out tinygrams in order to preserve mblk
4307                         * boundaries.
4308                         *
4309                         * By initializing tcp_cwnd_cnt to new tcp_cwnd and
4310                         * decrementing it by 1 MSS for every ACKs, tcp_cwnd is
4311                         * increased by 1 MSS for every RTTs.
4312                         */
4313                        if (tcp->tcp_cwnd_cnt <= 0) {
4314                                tcp->tcp_cwnd_cnt = cwnd + add;
4315                        } else {
4316                                tcp->tcp_cwnd_cnt -= add;
4317                                add = 0;
4318                        }
4319                }
4320                tcp->tcp_cwnd = MIN(cwnd + add, tcp->tcp_cwnd_max);
4321        }

4323        /* See if the latest urgent data has been acknowledged */
4324        if ((tcp->tcp_valid_bits & TCP_URG_VALID) &&
4325            SEQ_GT(seg_ack, tcp->tcp_urg))
4326                tcp->tcp_valid_bits &= ~TCP_URG_VALID;

4328        /* Can we update the RTT estimates? */
4329        if (tcp->tcp_snd_ts_ok) {
4330                /* Ignore zero timestamp echo-reply. */
4331                if (tcpopt.tcp_opt_ts_ecr != 0) {
4332                        tcp_set_rto(tcp, (int32_t)LBOLT_FASTPATH -
4333                            (int32_t)tcpopt.tcp_opt_ts_ecr);
4334                }

4336                /* If needed, restart the timer. */
4337                if (tcp->tcp_set_timer == 1) {
4338                        TCP_TIMER_RESTART(tcp, tcp->tcp_rto);
4339                        tcp->tcp_set_timer = 0;
4340                }
4341                /*
4342                 * Update tcp_csuna in case the other side stops sending
4343                 * us timestamps.
4344                 */
4345                tcp->tcp_csuna = tcp->tcp_snxt;
4346        } else if (SEQ_GT(seg_ack, tcp->tcp_csuna)) {
4347                /*
4348                 * An ACK sequence we haven't seen before, so get the RTT
4349                 * and update the RTO. But first check if the timestamp is
4350                 * valid to use.
4351                 */
4352                if ((mp1->b_next != NULL) &&
4353                    SEQ_GT(seg_ack, (uint32_t)(uintptr_t)(mp1->b_next)))
4354                        tcp_set_rto(tcp, (int32_t)LBOLT_FASTPATH -
4355                            (int32_t)(intptr_t)mp1->b_prev);
4356                else
4357                        TCPS_BUMP_MIB(tcps, tcpRttNoUpdate);

4359                /* Remeber the last sequence to be ACKed */
4360                tcp->tcp_csuna = seg_ack;
4361                if (tcp->tcp_set_timer == 1) {
4362                        TCP_TIMER_RESTART(tcp, tcp->tcp_rto);
4363                        tcp->tcp_set_timer = 0;
4364                }
4365        } else {
4366                TCPS_BUMP_MIB(tcps, tcpRttNoUpdate);
4367        }
```



```
4369        /* Eat acknowledged bytes off the xmit queue. */
4370        for (;;) {
4371                mblk_t  *mp2;
4372                uchar_t *wptr;

4374                wptr = mp1->b_wptr;
4375                ASSERT((uintptr_t)(wptr - mp1->b_rptr) <= (uintptr_t)INT_MAX);
4376                bytes_acked -= (int)(wptr - mp1->b_rptr);
4377                if (bytes_acked < 0) {
4378                        mp1->b_rptr = wptr + bytes_acked;
4379                        /*
4380                         * Set a new timestamp if all the bytes timed by the
4381                         * old timestamp have been ack'ed.
4382                         */
4383                        if (SEQ_GT(seg_ack,
4384                            (uint32_t)(uintptr_t)(mp1->b_next))) {
4385                                mp1->b_prev =
4386                                    (mblk_t *)(uintptr_t)LBOLT_FASTPATH;
4387                                mp1->b_next = NULL;
4388                        }
4389                        break;
4390                }
4391                mp1->b_next = NULL;
4392                mp1->b_prev = NULL;
4393                mp2 = mp1;
4394                mp1 = mp1->b_cont;

4396                /*
4397                 * This notification is required for some zero-copy
4398                 * clients to maintain a copy semantic. After the data
4399                 * is ack'ed, client is safe to modify or reuse the buffer.
4400                 */
4401                if (tcp->tcp_snd_zcopy_aware &&
4402                    (mp2->b_datap->db_struioflag & STRUIO_ZCNOTIFY))
4403                        tcp_zcopy_notify(tcp);
4404                freeb(mp2);
4405                if (bytes_acked == 0) {
4406                        if (mp1 == NULL) {
4407                                /* Everything is ack'ed, clear the tail. */
4408                                tcp->tcp_xmit_tail = NULL;
4409                                /*
4410                                 * Cancel the timer unless we are still
4411                                 * waiting for an ACK for the FIN packet.
4412                                 */
4413                                if (tcp->tcp_timer_tid != 0 &&
4414                                    tcp->tcp_snxt == tcp->tcp_suna) {
4415                                        (void) TCP_TIMER_CANCEL(tcp,
4416                                            tcp->tcp_timer_tid);
4417                                        tcp->tcp_timer_tid = 0;
4418                                }
4419                                goto pre_swnd_update;
4420                        }
4421                        if (mp2 != tcp->tcp_xmit_tail)
4422                                break;
4423                        tcp->tcp_xmit_tail = mp1;
4424                        ASSERT((uintptr_t)(mp1->b_wptr - mp1->b_rptr) <=
4425                            (uintptr_t)INT_MAX);
4426                        tcp->tcp_xmit_tail_unsent = (int)(mp1->b_wptr -
4427                            mp1->b_rptr);
4428                        break;
4429                }
4430                if (mp1 == NULL) {
4431                        /*
4432                         * More was acked but there is nothing more
4433                         * outstanding.  This means that the FIN was
4434                         * just acked or that we're talking to a clown.
```

```
4435                          */
4436 fin_acked:
4437                         ASSERT(tcp->tcp_fin_sent);
4438                         tcp->tcp_xmit_tail = NULL;
4439                         if (tcp->tcp_fin_sent) {
4440                                 /* FIN was acked - making progress */
4441                                 if (!tcp->tcp_fin_acked)
4442                                         tcp->tcp_ip_forward_progress = B_TRUE;
4443                                 tcp->tcp_fin_acked = B_TRUE;
4444                                 if (tcp->tcp_linger_tid != 0 &&
4445                                     TCP_TIMER_CANCEL(tcp,
4446                                     tcp->tcp_linger_tid) >= 0) {
4447                                         tcp_stop_lingering(tcp);
4448                                         freemsg(mp);
4449                                         mp = NULL;
4450                                 }
4451                         } else {
4452                                 /*
4453                                  * We should never get here because
4454                                  * we have already checked that the
4455                                  * number of bytes ack'ed should be
4456                                  * smaller than or equal to what we
4457                                  * have sent so far (it is the
4458                                  * acceptability check of the ACK).
4459                                  * We can only get here if the send
4460                                  * queue is corrupted.
4461                                  *
4462                                  * Terminate the connection and
4463                                  * panic the system.  It is better
4464                                  * for us to panic instead of
4465                                  * continuing to avoid other disaster.
4466                                  */
4467                                 tcp_xmit_ctl(NULL, tcp, tcp->tcp_snxt,
4468                                     tcp->tcp_rnxt, TH_RST|TH_ACK);
4469                                 panic("Memory corruption "
4470                                     "detected for connection %s.",
4471                                     tcp_display(tcp, NULL,
4472                                     DISP_ADDR_AND_PORT));
4473                                 /*NOTREACHED*/
4474                         }
4475                         goto pre_swnd_update;
4476                 }
4477                 ASSERT(mp2 != tcp->tcp_xmit_tail);
4478         }
4479         if (tcp->tcp_unsent) {
4480                 flags |= TH_XMIT_NEEDED;
4481         }
4482 pre_swnd_update:
4483         tcp->tcp_xmit_head = mp1;
4484 swnd_update:
4485         /*
4486          * The following check is different from most other implementations.
4487          * For bi-directional transfer, when segments are dropped, the
4488          * "normal" check will not accept a window update in those
4489          * retransmitted segemnts.  Failing to do that, TCP may send out
4490          * segments which are outside receiver's window.  As TCP accepts
4491          * the ack in those retransmitted segments, if the window update in
4492          * the same segment is not accepted, TCP will incorrectly calculates
4493          * that it can send more segments.  This can create a deadlock
4494          * with the receiver if its window becomes zero.
4495          */
4496         if (SEQ_LT(tcp->tcp_swl2, seg_ack) ||
4497             SEQ_LT(tcp->tcp_swl1, seg_seq) ||
4498             (tcp->tcp_swl1 == seg_seq && new_swnd > tcp->tcp_swnd)) {
4499                 /*
4500                  * The criteria for update is:
```



```
4501                  *
4502                  * 1. the segment acknowledges some data.  Or
4503                  * 2. the segment is new, i.e. it has a higher seq num. Or
4504                  * 3. the segment is not old and the advertised window is
4505                  * larger than the previous advertised window.
4506                  */
4507                 if (tcp->tcp_unsent && new_swnd > tcp->tcp_swnd)
4508                         flags |= TH_XMIT_NEEDED;
4509                 tcp->tcp_swnd = new_swnd;
4510                 if (new_swnd > tcp->tcp_max_swnd)
4511                         tcp->tcp_max_swnd = new_swnd;
4512                 tcp->tcp_swl1 = seg_seq;
4513                 tcp->tcp_swl2 = seg_ack;
4514         }
4515 est:
4516         if (tcp->tcp_state > TCPS_ESTABLISHED) {

4518                 switch (tcp->tcp_state) {
4519                 case TCPS_FIN_WAIT_1:
4520                         if (tcp->tcp_fin_acked) {
4521                                 tcp->tcp_state = TCPS_FIN_WAIT_2;
4522                                 DTRACE_TCP6(state__change, void, NULL,
4523                                     ip_xmit_attr_t *, connp->conn_ixa,
4524                                     void, NULL, tcp_t *, tcp, void, NULL,
4525                                     int32_t, TCPS_FIN_WAIT_1);
4526                                 /*
4527                                  * We implement the non-standard BSD/SunOS
4528                                  * FIN_WAIT_2 flushing algorithm.
4529                                  * If there is no user attached to this
4530                                  * TCP endpoint, then this TCP struct
4531                                  * could hang around forever in FIN_WAIT_2
4532                                  * state if the peer forgets to send us
4533                                  * a FIN.  To prevent this, we wait only
4534                                  * 2*MSL (a convenient time value) for
4535                                  * the FIN to arrive.  If it doesn't show up,
4536                                  * we flush the TCP endpoint.  This algorithm,
4537                                  * though a violation of RFC-793, has worked
4538                                  * for over 10 years in BSD systems.
4539                                  * Note: SunOS 4.x waits 675 seconds before
4540                                  * flushing the FIN_WAIT_2 connection.
4541                                  */
4542                                 TCP_TIMER_RESTART(tcp,
4543                                     tcp->tcp_fin_wait_2_flush_interval);
4544                         }
4545                         break;
4546                 case TCPS_FIN_WAIT_2:
4547                         break;  /* Shutdown hook? */
4548                 case TCPS_LAST_ACK:
4549                         freemsg(mp);
4550                         if (tcp->tcp_fin_acked) {
4551                                 (void) tcp_clean_death(tcp, 0);
4552                                 return;
4553                         }
4554                         goto xmit_check;
4555                 case TCPS_CLOSING:
4556                         if (tcp->tcp_fin_acked) {
4557                                 SET_TIME_WAIT(tcps, tcp, connp);
4558                                 DTRACE_TCP6(state__change, void, NULL,
4559                                     ip_xmit_attr_t *, connp->conn_ixa, void,
4560                                     NULL, tcp_t *, tcp, void, NULL, int32_t,
4561                                     TCPS_CLOSING);
4562                         }
4563                         /*FALLTHRU*/
4564                 case TCPS_CLOSE_WAIT:
4565                         freemsg(mp);
4566                         goto xmit_check;
```

```
4567                default:
4568                        ASSERT(tcp->tcp_state != TCPS_TIME_WAIT);
4569                        break;
4570                }
4571        }
4572        if (flags & TH_FIN) {
4573                /* Make sure we ack the fin */
4574                flags |= TH_ACK_NEEDED;
4575                if (!tcp->tcp_fin_rcvd) {
4576                        tcp->tcp_fin_rcvd = B_TRUE;
4577                        tcp->tcp_rnxt++;
4578                        tcpha = tcp->tcp_tcpha;
4579                        tcpha->tha_ack = htonl(tcp->tcp_rnxt);

4581                        /*
4582                         * Generate the ordrel_ind at the end unless the
4583                         * conn is detached or it is a STREAMS based eager.
4584                         * In the eager case we defer the notification until
4585                         * tcp_accept_finish has run.
4586                         */
4587                        if (!TCP_IS_DETACHED(tcp) && (IPCL_IS_NONSTR(connp) ||
4588                            (tcp->tcp_listener == NULL &&
4589                            !tcp->tcp_hard_binding)))
4590                                flags |= TH_ORDREL_NEEDED;
4591                        switch (tcp->tcp_state) {
4592                        case TCPS_SYN_RCVD:
4593                                tcp->tcp_state = TCPS_CLOSE_WAIT;
4594                                DTRACE_TCP6(state__change, void, NULL,
4595                                    ip_xmit_attr_t *, connp->conn_ixa,
4596                                    void, NULL, tcp_t *, tcp, void, NULL,
4597                                    int32_t, TCPS_SYN_RCVD);
4598                                /* Keepalive? */
4599                                break;
4600                        case TCPS_ESTABLISHED:
4601                                tcp->tcp_state = TCPS_CLOSE_WAIT;
4602                                DTRACE_TCP6(state__change, void, NULL,
4603                                    ip_xmit_attr_t *, connp->conn_ixa,
4604                                    void, NULL, tcp_t *, tcp, void, NULL,
4605                                    int32_t, TCPS_ESTABLISHED);
4606                                /* Keepalive? */
4607                                break;
4608                        case TCPS_FIN_WAIT_1:
4609                                if (!tcp->tcp_fin_acked) {
4610                                        tcp->tcp_state = TCPS_CLOSING;
4611                                        DTRACE_TCP6(state__change, void, NULL,
4612                                            ip_xmit_attr_t *, connp->conn_ixa,
4613                                            void, NULL, tcp_t *, tcp, void,
4614                                            NULL, int32_t, TCPS_FIN_WAIT_1);
4615                                        break;
4616                                }
4617                                /* FALLTHRU */
4618                        case TCPS_FIN_WAIT_2:
4619                                SET_TIME_WAIT(tcps, tcp, connp);
4620                                DTRACE_TCP6(state__change, void, NULL,
4621                                    ip_xmit_attr_t *, connp->conn_ixa, void,
4622                                    NULL, tcp_t *, tcp, void, NULL, int32_t,
4623                                    TCPS_FIN_WAIT_2);
4624                                if (seg_len) {
4625                                        /*
4626                                         * implies data piggybacked on FIN.
4627                                         * break to handle data.
4628                                         */
4629                                        break;
4630                                }
4631                                freemsg(mp);
4632                                goto ack_check;
```



```
4633                        }
4634                }
4635        }
4636        if (mp == NULL)
4637                goto xmit_check;
4638        if (seg_len == 0) {
4639                freemsg(mp);
4640                goto xmit_check;
4641        }
4642        if (mp->b_rptr == mp->b_wptr) {
4643                /*
4644                 * The header has been consumed, so we remove the
4645                 * zero-length mblk here.
4646                 */
4647                mp1 = mp;
4648                mp = mp->b_cont;
4649                freeb(mp1);
4650        }
4651 update_ack:
4652        tcpha = tcp->tcp_tcpha;
4653        tcp->tcp_rack_cnt++;
4654        {
4655                uint32_t cur_max;

4657                cur_max = tcp->tcp_rack_cur_max;
4658                if (tcp->tcp_rack_cnt >= cur_max) {
4659                        /*
4660                         * We have more unacked data than we should - send
4661                         * an ACK now.
4662                         */
4663                        flags |= TH_ACK_NEEDED;
4664                        cur_max++;
4665                        if (cur_max > tcp->tcp_rack_abs_max)
4666                                tcp->tcp_rack_cur_max = tcp->tcp_rack_abs_max;
4667                        else
4668                                tcp->tcp_rack_cur_max = cur_max;
4669                } else if (TCP_IS_DETACHED(tcp)) {
4670                        /* We don't have an ACK timer for detached TCP. */
4671                        flags |= TH_ACK_NEEDED;
4672                } else if (seg_len < mss) {
4673                        /*
4674                         * If we get a segment that is less than an mss, and we
4675                         * already have unacknowledged data, and the amount
4676                         * unacknowledged is not a multiple of mss, then we
4677                         * better generate an ACK now.  Otherwise, this may be
4678                         * the tail piece of a transaction, and we would rather
4679                         * wait for the response.
4680                         */
4681                        uint32_t udif;
4682                        ASSERT((uintptr_t)(tcp->tcp_rnxt - tcp->tcp_rack) <=
4683                            (uintptr_t)INT_MAX);
4684                        udif = (int)(tcp->tcp_rnxt - tcp->tcp_rack);
4685                        if (udif && (udif % mss))
4686                                flags |= TH_ACK_NEEDED;
4687                        else
4688                                flags |= TH_ACK_TIMER_NEEDED;
4689                } else {
4690                        /* Start delayed ack timer */
4691                        flags |= TH_ACK_TIMER_NEEDED;
4692                }
4693        }
4694        tcp->tcp_rnxt += seg_len;
4695        tcpha->tha_ack = htonl(tcp->tcp_rnxt);

4697        if (mp == NULL)
4698                goto xmit_check;
```

```
4700         /* Update SACK list */
4701         if (tcp->tcp_snd_sack_ok && tcp->tcp_num_sack_blk > 0) {
4702                 tcp_sack_remove(tcp->tcp_sack_list, tcp->tcp_rnxt,
4703                     &(tcp->tcp_num_sack_blk));
4704         }

4706         if (tcp->tcp_urp_mp) {
4707                 tcp->tcp_urp_mp->b_cont = mp;
4708                 mp = tcp->tcp_urp_mp;
4709                 tcp->tcp_urp_mp = NULL;
4710                 /* Ready for a new signal. */
4711                 tcp->tcp_urp_last_valid = B_FALSE;
4712 #ifdef DEBUG
4713                 (void) strlog(TCP_MOD_ID, 0, 1, SL_TRACE,
4714                     "tcp_rput: sending exdata_ind %s",
4715                     tcp_display(tcp, NULL, DISP_PORT_ONLY));
4716 #endif /* DEBUG */
4717         }

4719         /*
4720          * Check for ancillary data changes compared to last segment.
4721          */
4722         if (connp->conn_recv_ancillary.crb_all != 0) {
4723                 mp = tcp_input_add_ancillary(tcp, mp, &ipp, ira);
4724                 if (mp == NULL)
4725                         return;
4726         }

4728         if (IPCL_IS_NONSTR(connp)) {
4729                 /*
4730                  * Non-STREAMS socket
4731                  */
4732                 boolean_t push = flags & (TH_PUSH|TH_FIN);
4733                 int error;

4735                 if ((*sockupcalls->su_recv)(connp->conn_upper_handle,
4736                     mp, seg_len, 0, &error, &push) <= 0) {
4737                         /*
4738                          * We should never be in middle of a
4739                          * fallback, the squeue guarantees that.
4740                          */
4741                         ASSERT(error != EOPNOTSUPP);
4742                         if (error == ENOSPC)
4743                                 tcp->tcp_rwnd -= seg_len;
4744                 } else if (push) {
4745                         /* PUSH bit set and sockfs is not flow controlled */
4746                         flags |= tcp_rwnd_reopen(tcp);
4747                 }
4748         } else if (tcp->tcp_listener != NULL || tcp->tcp_hard_binding) {
4749                 /*
4750                  * Side queue inbound data until the accept happens.
4751                  * tcp_accept/tcp_rput drains this when the accept happens.
4752                  * M_DATA is queued on b_cont. Otherwise (T_OPTDATA_IND or
4753                  * T_EXDATA_IND) it is queued on b_next.
4754                  * XXX Make urgent data use this. Requires:
4755                  *      Removing tcp_listener check for TH_URG
4756                  *      Making M_PCPROTO and MARK messages skip the eager case
4757                  */

4759                 tcp_rcv_enqueue(tcp, mp, seg_len, ira->ira_cred);
4760         } else {
4761                 /* Active STREAMS socket */
4762                 if (mp->b_datap->db_type != M_DATA ||
4763                     (flags & TH_MARKNEXT_NEEDED)) {
4764                         if (tcp->tcp_rcv_list != NULL) {
```



```
4765                                 flags |= tcp_rcv_drain(tcp);
4766                         }
4767                         ASSERT(tcp->tcp_rcv_list == NULL ||
4768                             tcp->tcp_fused_sigurg);

4770                         if (flags & TH_MARKNEXT_NEEDED) {
4771 #ifdef DEBUG
4772                                 (void) strlog(TCP_MOD_ID, 0, 1, SL_TRACE,
4773                                     "tcp_rput: sending MSGMARKNEXT %s",
4774                                     tcp_display(tcp, NULL,
4775                                     DISP_PORT_ONLY));
4776 #endif /* DEBUG */
4777                                 mp->b_flag |= MSGMARKNEXT;
4778                                 flags &= ~TH_MARKNEXT_NEEDED;
4779                         }

4781                         if (is_system_labeled())
4782                                 tcp_setcred_data(mp, ira);

4784                         putnext(connp->conn_rq, mp);
4785                         if (!canputnext(connp->conn_rq))
4786                                 tcp->tcp_rwnd -= seg_len;
4787                 } else if ((flags & (TH_PUSH|TH_FIN)) ||
4788                     tcp->tcp_rcv_cnt + seg_len >= connp->conn_rcvbuf >> 3) {
4789                         if (tcp->tcp_rcv_list != NULL) {
4790                                 /*
4791                                  * Enqueue the new segment first and then
4792                                  * call tcp_rcv_drain() to send all data
4793                                  * up.  The other way to do this is to
4794                                  * send all queued data up and then call
4795                                  * putnext() to send the new segment up.
4796                                  * This way can remove the else part later
4797                                  * on.
4798                                  *
4799                                  * We don't do this to avoid one more call to
4800                                  * canputnext() as tcp_rcv_drain() needs to
4801                                  * call canputnext().
4802                                  */
4803                                 tcp_rcv_enqueue(tcp, mp, seg_len,
4804                                     ira->ira_cred);
4805                                 flags |= tcp_rcv_drain(tcp);
4806                         } else {
4807                                 if (is_system_labeled())
4808                                         tcp_setcred_data(mp, ira);

4810                                 putnext(connp->conn_rq, mp);
4811                                 if (!canputnext(connp->conn_rq))
4812                                         tcp->tcp_rwnd -= seg_len;
4813                         }
4814                 } else {
4815                         /*
4816                          * Enqueue all packets when processing an mblk
4817                          * from the co queue and also enqueue normal packets.
4818                          */
4819                         tcp_rcv_enqueue(tcp, mp, seg_len, ira->ira_cred);
4820                 }
4821                 /*
4822                  * Make sure the timer is running if we have data waiting
4823                  * for a push bit. This provides resiliency against
4824                  * implementations that do not correctly generate push bits.
4825                  */
4826                 if (tcp->tcp_rcv_list != NULL && tcp->tcp_push_tid == 0) {
4827                         /*
4828                          * The connection may be closed at this point, so don't
4829                          * do anything for a detached tcp.
4830                          */
```

```
4831                        if (!TCP_IS_DETACHED(tcp))
4832                                tcp->tcp_push_tid = TCP_TIMER(tcp,
4833                                    tcp_push_timer,
4834                                    tcps->tcps_push_timer_interval);
4835                }
4836        }

4838 xmit_check:
4839        /* Is there anything left to do? */
4840        ASSERT(!(flags & TH_MARKNEXT_NEEDED));
4841        if ((flags & (TH_REXMIT_NEEDED|TH_XMIT_NEEDED|TH_ACK_NEEDED|
4842            TH_NEED_SACK_REXMIT|TH_LIMIT_XMIT|TH_ACK_TIMER_NEEDED|
4843            TH_ORDREL_NEEDED|TH_SEND_URP_MARK)) == 0)
4844                goto done;

4846        /* Any transmit work to do and a non-zero window? */
4847        if ((flags & (TH_REXMIT_NEEDED|TH_XMIT_NEEDED|TH_NEED_SACK_REXMIT|
4848            TH_LIMIT_XMIT)) && tcp->tcp_swnd != 0) {
4849                if (flags & TH_REXMIT_NEEDED) {
4850                        uint32_t snd_size = tcp->tcp_snxt - tcp->tcp_suna;

4852                        TCPS_BUMP_MIB(tcps, tcpOutFastRetrans);
4853                        if (snd_size > mss)
4854                                snd_size = mss;
4855                        if (snd_size > tcp->tcp_swnd)
4856                                snd_size = tcp->tcp_swnd;
4857                        mp1 = tcp_xmit_mp(tcp, tcp->tcp_xmit_head, snd_size,
4858                            NULL, NULL, tcp->tcp_suna, B_TRUE, &snd_size,
4859                            B_TRUE);

4861                        if (mp1 != NULL) {
4862                                tcp->tcp_xmit_head->b_prev =
4863                                    (mblk_t *)LBOLT_FASTPATH;
4864                                tcp->tcp_csuna = tcp->tcp_snxt;
4865                                TCPS_BUMP_MIB(tcps, tcpRetransSegs);
4866                                TCPS_UPDATE_MIB(tcps, tcpRetransBytes,
4867                                    snd_size);
4868                                tcp_send_data(tcp, mp1);
4869                        }
4870                }
4871                if (flags & TH_NEED_SACK_REXMIT) {
4872                        tcp_sack_rexmit(tcp, &flags);
4873                }
4874                /*
4875                 * For TH_LIMIT_XMIT, tcp_wput_data() is called to send
4876                 * out new segment.  Note that tcp_rexmit should not be
4877                 * set, otherwise TH_LIMIT_XMIT should not be set.
4878                 */
4879                if (flags & (TH_XMIT_NEEDED|TH_LIMIT_XMIT)) {
4880                        if (!tcp->tcp_rexmit) {
4881                                tcp_wput_data(tcp, NULL, B_FALSE);
4882                        } else {
4883                                tcp_ss_rexmit(tcp);
4884                        }
4885                }
4886                /*
4887                 * Adjust tcp_cwnd back to normal value after sending
4888                 * new data segments.
4889                 */
4890                if (flags & TH_LIMIT_XMIT) {
4891                        tcp->tcp_cwnd -= mss << (tcp->tcp_dupack_cnt - 1);
4892                        /*
4893                         * This will restart the timer.  Restarting the
4894                         * timer is used to avoid a timeout before the
4895                         * limited transmitted segment's ACK gets back.
4896                         */
```



```
4897                        if (tcp->tcp_xmit_head != NULL)
4898                                tcp->tcp_xmit_head->b_prev =
4899                                    (mblk_t *)LBOLT_FASTPATH;
4900                }

4902                /* Anything more to do? */
4903                if ((flags & (TH_ACK_NEEDED|TH_ACK_TIMER_NEEDED|
4904                    TH_ORDREL_NEEDED|TH_SEND_URP_MARK)) == 0)
4905                        goto done;
4906        }
4907 ack_check:
4908        if (flags & TH_SEND_URP_MARK) {
4909                ASSERT(tcp->tcp_urp_mark_mp);
4910                ASSERT(!IPCL_IS_NONSTR(connp));
4911                /*
4912                 * Send up any queued data and then send the mark message
4913                 */
4914                if (tcp->tcp_rcv_list != NULL) {
4915                        flags |= tcp_rcv_drain(tcp);

4917                }
4918                ASSERT(tcp->tcp_rcv_list == NULL || tcp->tcp_fused_sigurg);
4919                mp1 = tcp->tcp_urp_mark_mp;
4920                tcp->tcp_urp_mark_mp = NULL;
4921                if (is_system_labeled())
4922                        tcp_setcred_data(mp1, ira);

4924                putnext(connp->conn_rq, mp1);
4925 #ifdef DEBUG
4926                (void) strlog(TCP_MOD_ID, 0, 1, SL_TRACE,
4927                    "tcp_rput: sending zero-length %s %s",
4928                    ((mp1->b_flag & MSGMARKNEXT) ? "MSGMARKNEXT" :
4929                    "MSGNOTMARKNEXT"),
4930                    tcp_display(tcp, NULL, DISP_PORT_ONLY));
4931 #endif /* DEBUG */
4932                flags &= ~TH_SEND_URP_MARK;
4933        }
4934        if (flags & TH_ACK_NEEDED) {
4935                /*
4936                 * Time to send an ack for some reason.
4937                 */
4938                mp1 = tcp_ack_mp(tcp);

4940                if (mp1 != NULL) {
4941                        tcp_send_data(tcp, mp1);
4942                        BUMP_LOCAL(tcp->tcp_obsegs);
4943                        TCPS_BUMP_MIB(tcps, tcpOutAck);
4944                }
4945                if (tcp->tcp_ack_tid != 0) {
4946                        (void) TCP_TIMER_CANCEL(tcp, tcp->tcp_ack_tid);
4947                        tcp->tcp_ack_tid = 0;
4948                }
4949        }
4950        if (flags & TH_ACK_TIMER_NEEDED) {
4951                /*
4952                 * Arrange for deferred ACK or push wait timeout.
4953                 * Start timer if it is not already running.
4954                 */
4955                if (tcp->tcp_ack_tid == 0) {
4956                        tcp->tcp_ack_tid = TCP_TIMER(tcp, tcp_ack_timer,
4957                            tcp->tcp_localnet ?
4958                            tcps->tcps_local_dack_interval :
4959                            tcps->tcps_deferred_ack_interval);
4960                }
4961        }
4962        if (flags & TH_ORDREL_NEEDED) {
```

```
4963                 /*
4964                  * Notify upper layer about an orderly release. If this is
4965                  * a non-STREAMS socket, then just make an upcall. For STREAMS
4966                  * we send up an ordrel_ind, unless this is an eager, in which
4967                  * case the ordrel will be sent when tcp_accept_finish runs.
4968                  * Note that for non-STREAMS we make an upcall even if it is an
4969                  * eager, because we have an upper handle to send it to.
4970                  */
4971                 ASSERT(IPCL_IS_NONSTR(connp) || tcp->tcp_listener == NULL);
4972                 ASSERT(!tcp->tcp_detached);

4974                 if (IPCL_IS_NONSTR(connp)) {
4975                         ASSERT(tcp->tcp_ordrel_mp == NULL);
4976                         tcp->tcp_ordrel_done = B_TRUE;
4977                         (*sockupcalls->su_opctl)(connp->conn_upper_handle,
4978                             SOCK_OPCTL_SHUT_RECV, 0);
4979                         goto done;
4980                 }

4982                 if (tcp->tcp_rcv_list != NULL) {
4983                         /*
4984                          * Push any mblk(s) enqueued from co processing.
4985                          */
4986                         flags |= tcp_rcv_drain(tcp);
4987                 }
4988                 ASSERT(tcp->tcp_rcv_list == NULL || tcp->tcp_fused_sigurg);

4990                 mp1 = tcp->tcp_ordrel_mp;
4991                 tcp->tcp_ordrel_mp = NULL;
4992                 tcp->tcp_ordrel_done = B_TRUE;
4993                 putnext(connp->conn_rq, mp1);
4994         }
4995 done:
4996         ASSERT(!(flags & TH_MARKNEXT_NEEDED));
4997 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  101416 Thu Nov  6 17:28:19 2014
new/usr/src/uts/common/inet/tcp/tcp_output.c
5295 remove maxburst logic from TCP's send algorithm Reviewed by: Dan McDonald <
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2014 by Delphix. All rights reserved.
 */

/* This file contains all TCP output processing functions. */

#include <sys/types.h>
#include <sys/stream.h>
#include <sys/strsun.h>
#include <sys/strsubr.h>
#include <sys/stropts.h>
#include <sys/strlog.h>
#define _SUN_TPI_VERSION 2
#include <sys/tihdr.h>
#include <sys/suntpi.h>
#include <sys/xti_inet.h>
#include <sys/timod.h>
#include <sys/pattr.h>
#include <sys/squeue_impl.h>
#include <sys/squeue.h>
#include <sys/sockio.h>
#include <sys/tsol/tnet.h>

#include <inet/common.h>
#include <inet/ip.h>
#include <inet/tcp.h>
#include <inet/tcp_impl.h>
#include <inet/snmpcom.h>
#include <inet/proto_set.h>
#include <inet/ipsec_impl.h>
#include <inet/ip_ndp.h>

static mblk_t   *tcp_get_seg_mp(tcp_t *, uint32_t, int32_t *);
static void     tcp_wput_cmdblk(queue_t *, mblk_t *);
static void     tcp_wput_flush(tcp_t *, mblk_t *);
static void     tcp_wput_iocdata(tcp_t *tcp, mblk_t *mp);
static int      tcp_xmit_end(tcp_t *);
static int      tcp_send(tcp_t *, const int, const int, const int,
                    const int, int *, uint_t *, int *, mblk_t **, mblk_t *);
```



```
static void     tcp_xmit_early_reset(char *, mblk_t *, uint32_t, uint32_t,
                    int, ip_recv_attr_t *, ip_stack_t *, conn_t *);
static boolean_t        tcp_send_rst_chk(tcp_stack_t *);
static void     tcp_process_shrunk_swnd(tcp_t *, uint32_t);
static void     tcp_fill_header(tcp_t *, uchar_t *, clock_t, int);

/*
 * Functions called directly via squeue having a prototype of edesc_t.
 */
static void     tcp_wput_nondata(void *, mblk_t *, void *, ip_recv_attr_t *);
static void     tcp_wput_ioctl(void *, mblk_t *, void *, ip_recv_attr_t *);
static void     tcp_wput_proto(void *, mblk_t *, void *, ip_recv_attr_t *);

/*
 * This controls how tiny a write must be before we try to copy it
 * into the mblk on the tail of the transmit queue.  Not much
 * speedup is observed for values larger than sixteen.  Zero will
 * disable the optimisation.
 */
static int tcp_tx_pull_len = 16;

void
tcp_wput(queue_t *q, mblk_t *mp)
{
        conn_t  *connp = Q_TO_CONN(q);
        tcp_t   *tcp;
        void (*output_proc)();
        t_scalar_t type;
        uchar_t *rptr;
        struct iocblk   *iocp;
        size_t size;

        ASSERT(connp->conn_ref >= 2);

        switch (DB_TYPE(mp)) {
        case M_DATA:
                tcp = connp->conn_tcp;
                ASSERT(tcp != NULL);

                size = msgdsize(mp);

                mutex_enter(&tcp->tcp_non_sq_lock);
                tcp->tcp_squeue_bytes += size;
                if (TCP_UNSENT_BYTES(tcp) > connp->conn_sndbuf) {
                        tcp_setqfull(tcp);
                }
                mutex_exit(&tcp->tcp_non_sq_lock);

                CONN_INC_REF(connp);
                SQUEUE_ENTER_ONE(connp->conn_sqp, mp, tcp_output, connp,
                    NULL, tcp_squeue_flag, SQTAG_TCP_OUTPUT);
                return;

        case M_CMD:
                tcp_wput_cmdblk(q, mp);
                return;

        case M_PROTO:
        case M_PCPROTO:
                /*
                 * if it is a snmp message, don't get behind the squeue
                 */
                tcp = connp->conn_tcp;
                rptr = mp->b_rptr;
                if ((mp->b_wptr - rptr) >= sizeof (t_scalar_t)) {
                        type = ((union T_primitives *)rptr)->type;
```

```
128                 } else {
129                         if (connp->conn_debug) {
130                                 (void) strlog(TCP_MOD_ID, 0, 1,
131                                     SL_ERROR|SL_TRACE,
132                                     "tcp_wput_proto, dropping one...");
133                         }
134                         freemsg(mp);
135                         return;
136                 }
137                 if (type == T_SVR4_OPTMGMT_REQ) {
138                         /*
139                          * All Solaris components should pass a db_credp
140                          * for this TPI message, hence we ASSERT.
141                          * But in case there is some other M_PROTO that looks
142                          * like a TPI message sent by some other kernel
143                          * component, we check and return an error.
144                          */
145                         cred_t  *cr = msg_getcred(mp, NULL);

147                         ASSERT(cr != NULL);
148                         if (cr == NULL) {
149                                 tcp_err_ack(tcp, mp, TSYSERR, EINVAL);
150                                 return;
151                         }
152                         if (snmpcom_req(q, mp, tcp_snmp_set, ip_snmp_get,
153                             cr)) {
154                                 /*
155                                  * This was a SNMP request
156                                  */
157                                 return;
158                         } else {
159                                 output_proc = tcp_wput_proto;
160                         }
161                 } else {
162                         output_proc = tcp_wput_proto;
163                 }
164                 break;
165         case M_IOCTL:
166                 /*
167                  * Most ioctls can be processed right away without going via
168                  * squeues - process them right here. Those that do require
169                  * squeue (currently _SIOCSOCKFALLBACK)
170                  * are processed by tcp_wput_ioctl().
171                  */
172                 iocp = (struct iocblk *)mp->b_rptr;
173                 tcp = connp->conn_tcp;

175                 switch (iocp->ioc_cmd) {
176                 case TCP_IOC_ABORT_CONN:
177                         tcp_ioctl_abort_conn(q, mp);
178                         return;
179                 case TI_GETPEERNAME:
180                 case TI_GETMYNAME:
181                         mi_copyin(q, mp, NULL,
182                             SIZEOF_STRUCT(strbuf, iocp->ioc_flag));
183                         return;

185                 default:
186                         output_proc = tcp_wput_ioctl;
187                         break;
188                 }
189                 break;
190         default:
191                 output_proc = tcp_wput_nondata;
192                 break;
193         }
```



```
195         CONN_INC_REF(connp);
196         SQUEUE_ENTER_ONE(connp->conn_sqp, mp, output_proc, connp,
197             NULL, tcp_squeue_flag, SQTAG_TCP_WPUT_OTHER);
198 }
_____unchanged_portion_omitted_

1761 /*
1762  * tcp_send() is called by tcp_wput_data() and returns one of the following:
1763  *
1764  * -1 = failed allocation.
1765  *  0 = We've either successfully sent data, or our usable send window is too
1766  *      small and we'd rather wait until later before sending again.
1764  *  0 = success; burst count reached, or usable send window is too small,
1765  *      and that we'd rather wait until later before sending again.
1767  */
1768 static int
1769 tcp_send(tcp_t *tcp, const int mss, const int total_hdr_len,
1770     const int tcp_hdr_len, const int num_sack_blk, int *usable,
1771     uint_t *snxt, int *tail_unsent, mblk_t **xmit_tail, mblk_t *local_time)
1772 {
1772         int             num_burst_seg = tcp->tcp_snd_burst;
1773         int             num_lso_seg = 1;
1774         uint_t          lso_usable;
1775         boolean_t       do_lso_send = B_FALSE;
1776         tcp_stack_t     *tcps = tcp->tcp_tcps;
1777         conn_t          *connp = tcp->tcp_connp;
1778         ip_xmit_attr_t  *ixa = connp->conn_ixa;

1780         /*
1781          * Check LSO possibility. The value of tcp->tcp_lso indicates whether
1782          * the underlying connection is LSO capable. Will check whether having
1783          * enough available data to initiate LSO transmission in the for(){}
1784          * loops.
1785          */
1786         if (tcp->tcp_lso && (tcp->tcp_valid_bits & ~TCP_FSS_VALID) == 0)
1787                 do_lso_send = B_TRUE;

1789         for (;;) {
1790                 struct datab    *db;
1791                 tcpha_t         *tcpha;
1792                 uint32_t        sum;
1793                 mblk_t          *mp, *mp1;
1794                 uchar_t         *rptr;
1795                 int             len;

1797                 /*
1798                  * Burst count reached, return successfully.
1799                  */
1800                 if (num_burst_seg == 0)
1801                         break;

1803                 /*
1798                  * Calculate the maximum payload length we can send at one
1799                  * time.
1800                  */
1801                 if (do_lso_send) {
1802                         /*
1803                          * Determine whether or not it's possible to do LSO,
1804                          * and if so, how much data we can send.
1809                          * Check whether be able to to do LSO for the current
1810                          * available data.
1805                          */
1806                         if ((*usable - 1) / mss >= 1) {
1812                         if (num_burst_seg >= 2 && (*usable - 1) / mss >= 1) {
1807                                 lso_usable = MIN(tcp->tcp_lso_max, *usable);
```

```
1814                                lso_usable = MIN(lso_usable,
1815                                    num_burst_seg * mss);

1808                                num_lso_seg = lso_usable / mss;
1809                                if (lso_usable % mss) {
1810                                        num_lso_seg++;
1811                                        tcp->tcp_last_sent_len = (ushort_t)
1812                                            (lso_usable % mss);
1813                                } else {
1814                                        tcp->tcp_last_sent_len = (ushort_t)mss;
1815                                }
1816                        } else {
1817                                do_lso_send = B_FALSE;
1818                                num_lso_seg = 1;
1819                                lso_usable = mss;
1820                        }
1821                }

1823                ASSERT(num_lso_seg <= IP_MAXPACKET / mss + 1);
1833 #ifdef DEBUG
1834                DTRACE_PROBE2(tcp_send_lso, int, num_lso_seg, boolean_t,
1835                    do_lso_send);
1836 #endif
1837                /*
1838                 * Adjust num_burst_seg here.
1839                 */
1840                num_burst_seg -= num_lso_seg;

1825                len = mss;
1826                if (len > *usable) {
1827                        ASSERT(do_lso_send == B_FALSE);

1829                        len = *usable;
1830                        if (len <= 0) {
1831                                /* Terminate the loop */
1832                                break;  /* success; too small */
1833                        }
1834                        /*
1835                         * Sender silly-window avoidance.
1836                         * Ignore this if we are going to send a
1837                         * zero window probe out.
1838                         *
1839                         * TODO: force data into microscopic window?
1840                         *      ==> (!pushed || (unsent > usable))
1841                         */
1842                        if (len < (tcp->tcp_max_swnd >> 1) &&
1843                            (tcp->tcp_unsent - (*snxt - tcp->tcp_snxt)) > len &&
1844                            !((tcp->tcp_valid_bits & TCP_URG_VALID) &&
1845                            len == 1) && (! tcp->tcp_zero_win_probe)) {
1846                                /*
1847                                 * If the retransmit timer is not running
1848                                 * we start it so that we will retransmit
1849                                 * in the case when the receiver has
1850                                 * decremented the window.
1851                                 */
1852                                if (*snxt == tcp->tcp_snxt &&
1853                                    *snxt == tcp->tcp_suna) {
1854                                        /*
1855                                         * We are not supposed to send
1856                                         * anything.  So let's wait a little
1857                                         * bit longer before breaking SWS
1858                                         * avoidance.
1859                                         *
1860                                         * What should the value be?
1861                                         * Suggestion: MAX(init rexmit time,
1862                                         * tcp->tcp_rto)
```



```
1863                                         */
1864                                        TCP_TIMER_RESTART(tcp, tcp->tcp_rto);
1865                                }
1866                                break;  /* success; too small */
1867                        }
1868                }

1870                tcpha = tcp->tcp_tcpha;

1872                /*
1873                 * The reason to adjust len here is that we need to set flags
1874                 * and calculate checksum.
1875                 */
1876                if (do_lso_send)
1877                        len = lso_usable;

1879                *usable -= len; /* Approximate - can be adjusted later */
1880                if (*usable > 0)
1881                        tcpha->tha_flags = TH_ACK;
1882                else
1883                        tcpha->tha_flags = (TH_ACK | TH_PUSH);

1885                /*
1886                 * Prime pump for IP's checksumming on our behalf.
1887                 * Include the adjustment for a source route if any.
1888                 * In case of LSO, the partial pseudo-header checksum should
1889                 * exclusive TCP length, so zero tha_sum before IP calculate
1890                 * pseudo-header checksum for partial checksum offload.
1891                 */
1892                if (do_lso_send) {
1893                        sum = 0;
1894                } else {
1895                        sum = len + tcp_hdr_len + connp->conn_sum;
1896                        sum = (sum >> 16) + (sum & 0xFFFF);
1897                }
1898                tcpha->tha_sum = htons(sum);
1899                tcpha->tha_seq = htonl(*snxt);

1901                /*
1902                 * Branch off to tcp_xmit_mp() if any of the VALID bits is
1903                 * set.  For the case when TCP_FSS_VALID is the only valid
1904                 * bit (normal active close), branch off only when we think
1905                 * that the FIN flag needs to be set.  Note for this case,
1906                 * that (snxt + len) may not reflect the actual seg_len,
1907                 * as len may be further reduced in tcp_xmit_mp().  If len
1908                 * gets modified, we will end up here again.
1909                 */
1910                if (tcp->tcp_valid_bits != 0 &&
1911                    (tcp->tcp_valid_bits != TCP_FSS_VALID ||
1912                    ((*snxt + len) == tcp->tcp_fss))) {
1913                        uchar_t         *prev_rptr;
1914                        uint32_t        prev_snxt = tcp->tcp_snxt;

1916                        if (*tail_unsent == 0) {
1917                                ASSERT((*xmit_tail)->b_cont != NULL);
1918                                *xmit_tail = (*xmit_tail)->b_cont;
1919                                prev_rptr = (*xmit_tail)->b_rptr;
1920                                *tail_unsent = (int)((*xmit_tail)->b_wptr -
1921                                    (*xmit_tail)->b_rptr);
1922                        } else {
1923                                prev_rptr = (*xmit_tail)->b_rptr;
1924                                (*xmit_tail)->b_rptr = (*xmit_tail)->b_wptr -
1925                                    *tail_unsent;
1926                        }
1927                        mp = tcp_xmit_mp(tcp, *xmit_tail, len, NULL, NULL,
1928                            *snxt, B_FALSE, (uint32_t *)&len, B_FALSE);
```

```
1929                        /* Restore tcp_snxt so we get amount sent right. */
1930                        tcp->tcp_snxt = prev_snxt;
1931                        if (prev_rptr == (*xmit_tail)->b_rptr) {
1932                                /*
1933                                 * If the previous timestamp is still in use,
1934                                 * don't stomp on it.
1935                                 */
1936                                if ((*xmit_tail)->b_next == NULL) {
1937                                        (*xmit_tail)->b_prev = local_time;
1938                                        (*xmit_tail)->b_next =
1939                                            (mblk_t *)(uintptr_t)(*snxt);
1940                                }
1941                        } else
1942                                (*xmit_tail)->b_rptr = prev_rptr;

1944                        if (mp == NULL) {
1945                                return (-1);
1946                        }
1947                        mp1 = mp->b_cont;

1949                        if (len <= mss) /* LSO is unusable (!do_lso_send) */
1950                                tcp->tcp_last_sent_len = (ushort_t)len;
1951                        while (mp1->b_cont) {
1952                                *xmit_tail = (*xmit_tail)->b_cont;
1953                                (*xmit_tail)->b_prev = local_time;
1954                                (*xmit_tail)->b_next =
1955                                    (mblk_t *)(uintptr_t)(*snxt);
1956                                mp1 = mp1->b_cont;
1957                        }
1958                        *snxt += len;
1959                        *tail_unsent = (*xmit_tail)->b_wptr - mp1->b_wptr;
1960                        BUMP_LOCAL(tcp->tcp_obsegs);
1961                        TCPS_BUMP_MIB(tcps, tcpOutDataSegs);
1962                        TCPS_UPDATE_MIB(tcps, tcpOutDataBytes, len);
1963                        tcp_send_data(tcp, mp);
1964                        continue;
1965                }

1967                *snxt += len;   /* Adjust later if we don't send all of len */
1968                TCPS_BUMP_MIB(tcps, tcpOutDataSegs);
1969                TCPS_UPDATE_MIB(tcps, tcpOutDataBytes, len);

1971                if (*tail_unsent) {
1972                        /* Are the bytes above us in flight? */
1973                        rptr = (*xmit_tail)->b_wptr - *tail_unsent;
1974                        if (rptr != (*xmit_tail)->b_rptr) {
1975                                *tail_unsent -= len;
1976                                if (len <= mss) /* LSO is unusable */
1977                                        tcp->tcp_last_sent_len = (ushort_t)len;
1978                                len += total_hdr_len;
1979                                ixa->ixa_pktlen = len;

1981                                if (ixa->ixa_flags & IXAF_IS_IPV4) {
1982                                        tcp->tcp_ipha->ipha_length = htons(len);
1983                                } else {
1984                                        tcp->tcp_ip6h->ip6_plen =
1985                                            htons(len - IPV6_HDR_LEN);
1986                                }

1988                                mp = dupb(*xmit_tail);
1989                                if (mp == NULL) {
1990                                        return (-1);    /* out_of_mem */
1991                                }
1992                                mp->b_rptr = rptr;
1993                                /*
1994                                 * If the old timestamp is no longer in use,
```



```
1995                                 * sample a new timestamp now.
1996                                 */
1997                                if ((*xmit_tail)->b_next == NULL) {
1998                                        (*xmit_tail)->b_prev = local_time;
1999                                        (*xmit_tail)->b_next =
2000                                            (mblk_t *)(uintptr_t)(*snxt-len);
2001                                }
2002                                goto must_alloc;
2003                        }
2004                } else {
2005                        *xmit_tail = (*xmit_tail)->b_cont;
2006                        ASSERT((uintptr_t)((*xmit_tail)->b_wptr -
2007                            (*xmit_tail)->b_rptr) <= (uintptr_t)INT_MAX);
2008                        *tail_unsent = (int)((*xmit_tail)->b_wptr -
2009                            (*xmit_tail)->b_rptr);
2010                }

2012                (*xmit_tail)->b_prev = local_time;
2013                (*xmit_tail)->b_next = (mblk_t *)(uintptr_t)(*snxt - len);

2015                *tail_unsent -= len;
2016                if (len <= mss) /* LSO is unusable (!do_lso_send) */
2017                        tcp->tcp_last_sent_len = (ushort_t)len;

2019                len += total_hdr_len;
2020                ixa->ixa_pktlen = len;

2022                if (ixa->ixa_flags & IXAF_IS_IPV4) {
2023                        tcp->tcp_ipha->ipha_length = htons(len);
2024                } else {
2025                        tcp->tcp_ip6h->ip6_plen = htons(len - IPV6_HDR_LEN);
2026                }

2028                mp = dupb(*xmit_tail);
2029                if (mp == NULL) {
2030                        return (-1);    /* out_of_mem */
2031                }

2033                len = total_hdr_len;
2034                /*
2035                 * There are four reasons to allocate a new hdr mblk:
2036                 *  1) The bytes above us are in use by another packet
2037                 *  2) We don't have good alignment
2038                 *  3) The mblk is being shared
2039                 *  4) We don't have enough room for a header
2040                 */
2041                rptr = mp->b_rptr - len;
2042                if (!OK_32PTR(rptr) ||
2043                    ((db = mp->b_datap), db->db_ref != 2) ||
2044                    rptr < db->db_base) {
2045                        /* NOTE: we assume allocb returns an OK_32PTR */

2047                must_alloc:;
2048                        mp1 = allocb(connp->conn_ht_iphc_allocated +
2049                            tcps->tcps_wroff_xtra, BPRI_MED);
2050                        if (mp1 == NULL) {
2051                                freemsg(mp);
2052                                return (-1);    /* out_of_mem */
2053                        }
2054                        mp1->b_cont = mp;
2055                        mp = mp1;
2056                        /* Leave room for Link Level header */
2057                        len = total_hdr_len;
2058                        rptr = &mp->b_rptr[tcps->tcps_wroff_xtra];
2059                        mp->b_wptr = &rptr[len];
2060                }
```

```
2062                /*
2063                 * Fill in the header using the template header, and add
2064                 * options such as time-stamp, ECN and/or SACK, as needed.
2065                 */
2066                tcp_fill_header(tcp, rptr, (clock_t)local_time, num_sack_blk);

2068                mp->b_rptr = rptr;

2070                if (*tail_unsent) {
2071                        int spill = *tail_unsent;

2073                        mp1 = mp->b_cont;
2074                        if (mp1 == NULL)
2075                                mp1 = mp;

2077                        /*
2078                         * If we're a little short, tack on more mblks until
2079                         * there is no more spillover.
2080                         */
2081                        while (spill < 0) {
2082                                mblk_t *nmp;
2083                                int nmpsz;

2085                                nmp = (*xmit_tail)->b_cont;
2086                                nmpsz = MBLKL(nmp);

2088                                /*
2089                                 * Excess data in mblk; can we split it?
2090                                 * If LSO is enabled for the connection,
2091                                 * keep on splitting as this is a transient
2092                                 * send path.
2093                                 */
2094                                if (!do_lso_send && (spill + nmpsz > 0)) {
2095                                        /*
2096                                         * Don't split if stream head was
2097                                         * told to break up larger writes
2098                                         * into smaller ones.
2099                                         */
2100                                        if (tcp->tcp_maxpsz_multiplier > 0)
2101                                                break;

2103                                        /*
2104                                         * Next mblk is less than SMSS/2
2105                                         * rounded up to nearest 64-byte;
2106                                         * let it get sent as part of the
2107                                         * next segment.
2108                                         */
2109                                        if (tcp->tcp_localnet &&
2110                                            !tcp->tcp_cork &&
2111                                            (nmpsz < roundup((mss >> 1), 64)))
2112                                                break;
2113                                }

2115                                *xmit_tail = nmp;
2116                                ASSERT((uintptr_t)nmpsz <= (uintptr_t)INT_MAX);
2117                                /* Stash for rtt use later */
2118                                (*xmit_tail)->b_prev = local_time;
2119                                (*xmit_tail)->b_next =
2120                                    (mblk_t *)(uintptr_t)(*snxt - len);
2121                                mp1->b_cont = dupb(*xmit_tail);
2122                                mp1 = mp1->b_cont;

2124                                spill += nmpsz;
2125                                if (mp1 == NULL) {
2126                                        *tail_unsent = spill;
```



```
2127                                            freemsg(mp);
2128                                            return (-1);    /* out_of_mem */
2129                                    }
2130                            }

2132                            /* Trim back any surplus on the last mblk */
2133                            if (spill >= 0) {
2134                                    mp1->b_wptr -= spill;
2135                                    *tail_unsent = spill;
2136                            } else {
2137                                    /*
2138                                     * We did not send everything we could in
2139                                     * order to remain within the b_cont limit.
2140                                     */
2141                                    *usable -= spill;
2142                                    *snxt += spill;
2143                                    tcp->tcp_last_sent_len += spill;
2144                                    TCPS_UPDATE_MIB(tcps, tcpOutDataBytes, spill);
2145                                    /*
2146                                     * Adjust the checksum
2147                                     */
2148                                    tcpha = (tcpha_t *)(rptr +
2149                                        ixa->ixa_ip_hdr_length);
2150                                    sum += spill;
2151                                    sum = (sum >> 16) + (sum & 0xFFFF);
2152                                    tcpha->tha_sum = htons(sum);
2153                                    if (connp->conn_ipversion == IPV4_VERSION) {
2154                                            sum = ntohs(
2155                                                ((ipha_t *)rptr)->ipha_length) +
2156                                                spill;
2157                                            ((ipha_t *)rptr)->ipha_length =
2158                                                htons(sum);
2159                                    } else {
2160                                            sum = ntohs(
2161                                                ((ip6_t *)rptr)->ip6_plen) +
2162                                                spill;
2163                                            ((ip6_t *)rptr)->ip6_plen =
2164                                                htons(sum);
2165                                    }
2166                                    ixa->ixa_pktlen += spill;
2167                                    *tail_unsent = 0;
2168                            }
2169                    }
2170                    if (tcp->tcp_ip_forward_progress) {
2171                            tcp->tcp_ip_forward_progress = B_FALSE;
2172                            ixa->ixa_flags |= IXAF_REACH_CONF;
2173                    } else {
2174                            ixa->ixa_flags &= ~IXAF_REACH_CONF;
2175                    }

2177                    if (do_lso_send) {
2178                            /* Append LSO information to the mp. */
2179                            lso_info_set(mp, mss, HW_LSO);
2180                            ixa->ixa_fragsize = IP_MAXPACKET;
2181                            ixa->ixa_extra_ident = num_lso_seg - 1;

2183                            DTRACE_PROBE2(tcp_send_lso, int, num_lso_seg,
2184                                boolean_t, B_TRUE);

2186                            tcp_send_data(tcp, mp);

2188                            /*
2189                             * Restore values of ixa_fragsize and ixa_extra_ident.
2190                             */
2191                            ixa->ixa_fragsize = ixa->ixa_pmtu;
2192                            ixa->ixa_extra_ident = 0;
```

```
2193                        tcp->tcp_obsegs += num_lso_seg;
2194                        TCP_STAT(tcps, tcp_lso_times);
2195                        TCP_STAT_UPDATE(tcps, tcp_lso_pkt_out, num_lso_seg);
2196                } else {
2197                        /*
2198                         * Make sure to clean up LSO information. Wherever a
2199                         * new mp uses the prepended header room after dupb(),
2200                         * lso_info_cleanup() should be called.
2201                         */
2202                        lso_info_cleanup(mp);
2203                        tcp_send_data(tcp, mp);
2204                        BUMP_LOCAL(tcp->tcp_obsegs);
2205                }
2206        }

2208        return (0);
2209 }
```

_____unchanged_portion_omitted_

```
3407 /*
3408  * tcp_ss_rexmit() is called to do slow start retransmission after a timeout
3409  * or ICMP errors.
3427  *
3428  * To limit the number of duplicate segments, we limit the number of segment
3429  * to be sent in one time to tcp_snd_burst, the burst variable.
3410  */
3411 void
3412 tcp_ss_rexmit(tcp_t *tcp)
3413 {
3414         uint32_t        snxt;
3415         uint32_t        smax;
3416         int32_t         win;
3417         int32_t         mss;
3418         int32_t         off;
3439         int32_t         burst = tcp->tcp_snd_burst;
3419         mblk_t          *snxt_mp;
3420         tcp_stack_t     *tcps = tcp->tcp_tcps;

3422         /*
3423          * Note that tcp_rexmit can be set even though TCP has retransmitted
3424          * all unack'ed segments.
3425          */
3426         if (SEQ_LT(tcp->tcp_rexmit_nxt, tcp->tcp_rexmit_max)) {
3427                 smax = tcp->tcp_rexmit_max;
3428                 snxt = tcp->tcp_rexmit_nxt;
3429                 if (SEQ_LT(snxt, tcp->tcp_suna)) {
3430                         snxt = tcp->tcp_suna;
3431                 }
3432                 win = MIN(tcp->tcp_cwnd, tcp->tcp_swnd);
3433                 win -= snxt - tcp->tcp_suna;
3434                 mss = tcp->tcp_mss;
3435                 snxt_mp = tcp_get_seg_mp(tcp, snxt, &off);

3437                 while (SEQ_LT(snxt, smax) && (win > 0) && (snxt_mp != NULL)) {
3458                 while (SEQ_LT(snxt, smax) && (win > 0) &&
3459                     (burst > 0) && (snxt_mp != NULL)) {
3438                         mblk_t  *xmit_mp;
3439                         mblk_t  *old_snxt_mp = snxt_mp;
3440                         uint32_t cnt = mss;

3442                         if (win < cnt) {
3443                                 cnt = win;
3444                         }
3445                         if (SEQ_GT(snxt + cnt, smax)) {
3446                                 cnt = smax - snxt;
3447                         }
```



```
3448                         xmit_mp = tcp_xmit_mp(tcp, snxt_mp, cnt, &off,
3449                             &snxt_mp, snxt, B_TRUE, &cnt, B_TRUE);
3450                         if (xmit_mp == NULL)
3451                                 return;

3453                         tcp_send_data(tcp, xmit_mp);

3455                         snxt += cnt;
3456                         win -= cnt;
3457                         /*
3458                          * Update the send timestamp to avoid false
3459                          * retransmission.
3460                          */
3461                         old_snxt_mp->b_prev = (mblk_t *)ddi_get_lbolt();
3462                         TCPS_BUMP_MIB(tcps, tcpRetransSegs);
3463                         TCPS_UPDATE_MIB(tcps, tcpRetransBytes, cnt);

3465                         tcp->tcp_rexmit_nxt = snxt;
3488                         burst--;
3466                 }
3467                 /*
3468                  * If we have transmitted all we have at the time
3469                  * we started the retranmission, we can leave
3470                  * the rest of the job to tcp_wput_data().  But we
3471                  * need to check the send window first.  If the
3472                  * win is not 0, go on with tcp_wput_data().
3473                  */
3474                 if (SEQ_LT(snxt, smax) || win == 0) {
3475                         return;
3476                 }
3477         }
3478         /* Only call tcp_wput_data() if there is data to be sent. */
3479         if (tcp->tcp_unsent) {
3480                 tcp_wput_data(tcp, NULL, B_FALSE);
3481         }
3482 }

3484 /*
3485  * Do slow start retransmission after ICMP errors of PMTU changes.
3486  */
3487 void
3488 tcp_rexmit_after_error(tcp_t *tcp)
3489 {
3490         /*
3491          * All sent data has been acknowledged or no data left to send, just
3492          * to return.
3493          */
3494         if (!SEQ_LT(tcp->tcp_suna, tcp->tcp_snxt) ||
3495             (tcp->tcp_xmit_head == NULL))
3496                 return;

3498         if ((tcp->tcp_valid_bits & TCP_FSS_VALID) && (tcp->tcp_unsent == 0))
3499                 tcp->tcp_rexmit_max = tcp->tcp_fss;
3500         else
3501                 tcp->tcp_rexmit_max = tcp->tcp_snxt;

3503         tcp->tcp_rexmit_nxt = tcp->tcp_suna;
3504         tcp->tcp_rexmit = B_TRUE;
3505         tcp->tcp_dupack_cnt = 0;
3529         tcp->tcp_snd_burst = TCP_CWND_SS;
3506         tcp_ss_rexmit(tcp);
3507 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   33419 Thu Nov  6 17:28:20 2014
new/usr/src/uts/common/inet/tcp/tcp_timers.c
5295 remove maxburst logic from TCP's send algorithm Reviewed by: Dan McDonald <
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2011 Nexenta Systems, Inc. All rights reserved.
 * Copyright 2011 Joyent, Inc.  All rights reserved.
 * Copyright (c) 2014 by Delphix. All rights reserved.
 */

#include <sys/types.h>
#include <sys/strlog.h>
#include <sys/strsun.h>
#include <sys/squeue_impl.h>
#include <sys/squeue.h>
#include <sys/callo.h>
#include <sys/strsubr.h>

#include <inet/common.h>
#include <inet/ip.h>
#include <inet/ip_ire.h>
#include <inet/ip_rts.h>
#include <inet/tcp.h>
#include <inet/tcp_impl.h>

/*
 * Implementation of TCP Timers.
 * =============================
 *
 * INTERFACE:
 *
 * There are two basic functions dealing with tcp timers:
 *
 *      timeout_id_t    tcp_timeout(connp, func, time)
 *      clock_t         tcp_timeout_cancel(connp, timeout_id)
 *      TCP_TIMER_RESTART(tcp, intvl)
 *
 * tcp_timeout() starts a timer for the 'tcp' instance arranging to call 'func'
 * after 'time' ticks passed. The function called by timeout() must adhere to
 * the same restrictions as a driver soft interrupt handler - it must not sleep
 * or call other functions that might sleep. The value returned is the opaque
 * non-zero timeout identifier that can be passed to tcp_timeout_cancel() to
 * cancel the request. The call to tcp_timeout() may fail in which case it
```



```
 * returns zero. This is different from the timeout(9F) function which never
 * fails.
 *
 * The call-back function 'func' always receives 'connp' as its single
 * argument. It is always executed in the squeue corresponding to the tcp
 * structure. The tcp structure is guaranteed to be present at the time the
 * call-back is called.
 *
 * NOTE: The call-back function 'func' is never called if tcp is in
 *      the TCPS_CLOSED state.
 *
 * tcp_timeout_cancel() attempts to cancel a pending tcp_timeout()
 * request. locks acquired by the call-back routine should not be held across
 * the call to tcp_timeout_cancel() or a deadlock may result.
 *
 * tcp_timeout_cancel() returns -1 if the timeout request is invalid.
 * Otherwise, it returns an integer value greater than or equal to 0.
 *
 * NOTE: both tcp_timeout() and tcp_timeout_cancel() should always be called
 *      within squeue context corresponding to the tcp instance. Since the
 *      call-back is also called via the same squeue, there are no race
 *      conditions described in untimeout(9F) manual page since all calls are
 *      strictly serialized.
 *
 *      TCP_TIMER_RESTART() is a macro that attempts to cancel a pending timeout
 *      stored in tcp_timer_tid and starts a new one using
 *      MSEC_TO_TICK(intvl). It always uses tcp_timer() function as a call-back
 *      and stores the return value of tcp_timeout() in the tcp->tcp_timer_tid
 *      field.
 *
 * IMPLEMENTATION:
 *
 * TCP timers are implemented using three-stage process. The call to
 * tcp_timeout() uses timeout(9F) function to call tcp_timer_callback() function
 * when the timer expires. The tcp_timer_callback() arranges the call of the
 * tcp_timer_handler() function via squeue corresponding to the tcp
 * instance. The tcp_timer_handler() calls actual requested timeout call-back
 * and passes tcp instance as an argument to it. Information is passed between
 * stages using the tcp_timer_t structure which contains the connp pointer, the
 * tcp call-back to call and the timeout id returned by the timeout(9F).
 *
 * The tcp_timer_t structure is not used directly, it is embedded in an mblk_t -
 * like structure that is used to enter an squeue. The mp->b_rptr of this pseudo
 * mblk points to the beginning of tcp_timer_t structure. The tcp_timeout()
 * returns the pointer to this mblk.
 *
 * The pseudo mblk is allocated from a special tcp_timer_cache kmem cache. It
 * looks like a normal mblk without actual dblk attached to it.
 *
 * To optimize performance each tcp instance holds a small cache of timer
 * mblocks. In the current implementation it caches up to two timer mblocks per
 * tcp instance. The cache is preserved over tcp frees and is only freed when
 * the whole tcp structure is destroyed by its kmem destructor. Since all tcp
 * timer processing happens on a corresponding squeue, the cache manipulation
 * does not require any locks. Experiments show that majority of timer mblocks
 * allocations are satisfied from the tcp cache and do not involve kmem calls.
 *
 * The tcp_timeout() places a refhold on the connp instance which guarantees
 * that it will be present at the time the call-back function fires. The
 * tcp_timer_handler() drops the reference after calling the call-back, so the
 * call-back function does not need to manipulate the references explicitly.
 */

kmem_cache_t *tcp_timercache;

static void     tcp_ip_notify(tcp_t *);
```

```
128 static void     tcp_timer_callback(void *);
129 static void     tcp_timer_free(tcp_t *, mblk_t *);
130 static void     tcp_timer_handler(void *, mblk_t *, void *, ip_recv_attr_t *);

132 /*
133  * tim is in millisec.
134  */
135 timeout_id_t
136 tcp_timeout(conn_t *connp, void (*f)(void *), hrtime_t tim)
137 {
138         mblk_t *mp;
139         tcp_timer_t *tcpt;
140         tcp_t *tcp = connp->conn_tcp;

142         ASSERT(connp->conn_sqp != NULL);

144         TCP_DBGSTAT(tcp->tcp_tcps, tcp_timeout_calls);

146         if (tcp->tcp_timercache == NULL) {
147                 mp = tcp_timermp_alloc(KM_NOSLEEP | KM_PANIC);
148         } else {
149                 TCP_DBGSTAT(tcp->tcp_tcps, tcp_timeout_cached_alloc);
150                 mp = tcp->tcp_timercache;
151                 tcp->tcp_timercache = mp->b_next;
152                 mp->b_next = NULL;
153                 ASSERT(mp->b_wptr == NULL);
154         }

156         CONN_INC_REF(connp);
157         tcpt = (tcp_timer_t *)mp->b_rptr;
158         tcpt->connp = connp;
159         tcpt->tcpt_proc = f;
160         /*
161          * TCP timers are normal timeouts. Plus, they do not require more than
162          * a 10 millisecond resolution. By choosing a coarser resolution and by
163          * rounding up the expiration to the next resolution boundary, we can
164          * batch timers in the callout subsystem to make TCP timers more
165          * efficient. The roundup also protects short timers from expiring too
166          * early before they have a chance to be cancelled.
167          */
168         tcpt->tcpt_tid = timeout_generic(CALLOUT_NORMAL, tcp_timer_callback, mp,
169             tim * MICROSEC, CALLOUT_TCP_RESOLUTION, CALLOUT_FLAG_ROUNDUP);
170         VERIFY(!(tcpt->tcpt_tid & CALLOUT_ID_FREE));

172         return ((timeout_id_t)mp);
173 }
```

unchanged_portion_omitted_

```
635 /*
636  * tcp_timer is the timer service routine.  It handles the retransmission,
637  * FIN_WAIT_2 flush, and zero window probe timeout events.  It figures out
638  * from the state of the tcp instance what kind of action needs to be done
639  * at the time it is called.
640  */
641 void
642 tcp_timer(void *arg)
643 {
644         mblk_t          *mp;
645         clock_t         first_threshold;
646         clock_t         second_threshold;
647         clock_t         ms;
648         uint32_t        mss;
649         conn_t          *connp = (conn_t *)arg;
650         tcp_t           *tcp = connp->conn_tcp;
651         tcp_stack_t     *tcps = tcp->tcp_tcps;
652         boolean_t       dont_timeout = B_FALSE;
```



```
654         tcp->tcp_timer_tid = 0;

656         if (tcp->tcp_fused)
657                 return;

659         first_threshold =  tcp->tcp_first_timer_threshold;
660         second_threshold = tcp->tcp_second_timer_threshold;
661         switch (tcp->tcp_state) {
662         case TCPS_IDLE:
663         case TCPS_BOUND:
664         case TCPS_LISTEN:
665                 return;
666         case TCPS_SYN_RCVD: {
667                 tcp_t   *listener = tcp->tcp_listener;

669                 if (tcp->tcp_syn_rcvd_timeout == 0 && (listener != NULL)) {
670                         /* it's our first timeout */
671                         tcp->tcp_syn_rcvd_timeout = 1;
672                         mutex_enter(&listener->tcp_eager_lock);
673                         listener->tcp_syn_rcvd_timeout++;
674                         if (!tcp->tcp_dontdrop && !tcp->tcp_closemp_used) {
675                                 /*
676                                  * Make this eager available for drop if we
677                                  * need to drop one to accomodate a new
678                                  * incoming SYN request.
679                                  */
680                                 MAKE_DROPPABLE(listener, tcp);
681                         }
682                         if (!listener->tcp_syn_defense &&
683                             (listener->tcp_syn_rcvd_timeout >
684                             (tcps->tcps_conn_req_max_q0 >> 2)) &&
685                             (tcps->tcps_conn_req_max_q0 > 200)) {
686                                 /* We may be under attack. Put on a defense. */
687                                 listener->tcp_syn_defense = B_TRUE;
688                                 cmn_err(CE_WARN, "High TCP connect timeout "
689                                     "rate! System (port %d) may be under a "
690                                     "SYN flood attack!",
691                                     ntohs(listener->tcp_connp->conn_lport));

693                                 listener->tcp_ip_addr_cache = kmem_zalloc(
694                                     IP_ADDR_CACHE_SIZE * sizeof (ipaddr_t),
695                                     KM_NOSLEEP);
696                         }
697                         mutex_exit(&listener->tcp_eager_lock);
698                 } else if (listener != NULL) {
699                         mutex_enter(&listener->tcp_eager_lock);
700                         tcp->tcp_syn_rcvd_timeout++;
701                         if (tcp->tcp_syn_rcvd_timeout > 1 &&
702                             !tcp->tcp_closemp_used) {
703                                 /*
704                                  * This is our second timeout. Put the tcp in
705                                  * the list of droppable eagers to allow it to
706                                  * be dropped, if needed. We don't check
707                                  * whether tcp_dontdrop is set or not to
708                                  * protect ourselve from a SYN attack where a
709                                  * remote host can spoof itself as one of the
710                                  * good IP source and continue to hold
711                                  * resources too long.
712                                  */
713                                 MAKE_DROPPABLE(listener, tcp);
714                         }
715                         mutex_exit(&listener->tcp_eager_lock);
716                 }
717         }
718                 /* FALLTHRU */
```

```
719         case TCPS_SYN_SENT:
720                 first_threshold =  tcp->tcp_first_ctimer_threshold;
721                 second_threshold = tcp->tcp_second_ctimer_threshold;

723                 /*
724                  * If an app has set the second_threshold to 0, it means that
725                  * we need to retransmit forever, unless this is a passive
726                  * open.  We need to set second_threshold back to a normal
727                  * value such that later comparison with it still makes
728                  * sense.  But we set dont_timeout to B_TRUE so that we will
729                  * never time out.
730                  */
731                 if (second_threshold == 0) {
732                         second_threshold = tcps->tcps_ip_abort_linterval;
733                         if (tcp->tcp_active_open)
734                                 dont_timeout = B_TRUE;
735                 }
736                 break;
737         case TCPS_ESTABLISHED:
738         case TCPS_CLOSE_WAIT:
739                 /*
740                  * If the end point has not been closed, TCP can retransmit
741                  * forever.  But if the end point is closed, the normal
742                  * timeout applies.
743                  */
744                 if (second_threshold == 0) {
745                         second_threshold = tcps->tcps_ip_abort_linterval;
746                         dont_timeout = B_TRUE;
747                 }
748                 /* FALLTHRU */
749         case TCPS_FIN_WAIT_1:
750         case TCPS_CLOSING:
751         case TCPS_LAST_ACK:
752                 /* If we have data to rexmit */
753                 if (tcp->tcp_suna != tcp->tcp_snxt) {
754                         clock_t time_to_wait;

756                         TCPS_BUMP_MIB(tcps, tcpTimRetrans);
757                         if (!tcp->tcp_xmit_head)
758                                 break;
759                         time_to_wait = ddi_get_lbolt() -
760                             (clock_t)tcp->tcp_xmit_head->b_prev;
761                         time_to_wait = tcp->tcp_rto -
762                             TICK_TO_MSEC(time_to_wait);
763                         /*
764                          * If the timer fires too early, 1 clock tick earlier,
765                          * restart the timer.
766                          */
767                         if (time_to_wait > msec_per_tick) {
768                                 TCP_STAT(tcps, tcp_timer_fire_early);
769                                 TCP_TIMER_RESTART(tcp, time_to_wait);
770                                 return;
771                         }
772                         /*
773                          * When we probe zero windows, we force the swnd open.
774                          * If our peer acks with a closed window swnd will be
775                          * set to zero by tcp_rput(). As long as we are
776                          * receiving acks tcp_rput will
777                          * reset 'tcp_ms_we_have_waited' so as not to trip the
778                          * first and second interval actions.  NOTE: the timer
779                          * interval is allowed to continue its exponential
780                          * backoff.
781                          */
782                         if (tcp->tcp_swnd == 0 || tcp->tcp_zero_win_probe) {
783                                 if (connp->conn_debug) {
784                                         (void) strlog(TCP_MOD_ID, 0, 1,
```



```
785                                             SL_TRACE, "tcp_timer: zero win");
786                                 }
787                         } else {
788                                 /*
789                                  * After retransmission, we need to do
790                                  * slow start.  Set the ssthresh to one
791                                  * half of current effective window and
792                                  * cwnd to one MSS.  Also reset
793                                  * tcp_cwnd_cnt.
794                                  *
795                                  * Note that if tcp_ssthresh is reduced because
796                                  * of ECN, do not reduce it again unless it is
797                                  * already one window of data away (tcp_cwr
798                                  * should then be cleared) or this is a
799                                  * timeout for a retransmitted segment.
800                                  */
801                                 uint32_t npkt;

803                                 if (!tcp->tcp_cwr || tcp->tcp_rexmit) {
804                                         npkt = ((tcp->tcp_timer_backoff ?
805                                             tcp->tcp_cwnd_ssthresh :
806                                             tcp->tcp_snxt -
807                                             tcp->tcp_suna) >> 1) / tcp->tcp_mss;
808                                         tcp->tcp_cwnd_ssthresh = MAX(npkt, 2) *
809                                             tcp->tcp_mss;
810                                 }
811                                 tcp->tcp_cwnd = tcp->tcp_mss;
812                                 tcp->tcp_cwnd_cnt = 0;
813                                 if (tcp->tcp_ecn_ok) {
814                                         tcp->tcp_cwr = B_TRUE;
815                                         tcp->tcp_cwr_snd_max = tcp->tcp_snxt;
816                                         tcp->tcp_ecn_cwr_sent = B_FALSE;
817                                 }
818                         }
819                         break;
820                 }
821                 /*
822                  * We have something to send yet we cannot send.  The
823                  * reason can be:
824                  *
825                  * 1. Zero send window: we need to do zero window probe.
826                  * 2. Zero cwnd: because of ECN, we need to "clock out
827                  * segments.
828                  * 3. SWS avoidance: receiver may have shrunk window,
829                  * reset our knowledge.
830                  *
831                  * Note that condition 2 can happen with either 1 or
832                  * 3.  But 1 and 3 are exclusive.
833                  */
834                 if (tcp->tcp_unsent != 0) {
835                         /*
836                          * Should not hold the zero-copy messages for too long.
837                          */
838                         if (tcp->tcp_snd_zcopy_aware && !tcp->tcp_xmit_zc_clean)
839                                 tcp->tcp_xmit_head = tcp_zcopy_backoff(tcp,
840                                     tcp->tcp_xmit_head, B_TRUE);

842                         if (tcp->tcp_cwnd == 0) {
843                                 /*
844                                  * Set tcp_cwnd to 1 MSS so that a
845                                  * new segment can be sent out.  We
846                                  * are "clocking out" new data when
847                                  * the network is really congested.
848                                  */
849                                 ASSERT(tcp->tcp_ecn_ok);
850                                 tcp->tcp_cwnd = tcp->tcp_mss;
```

```
851                         }
852                         if (tcp->tcp_swnd == 0) {
853                                 /* Extend window for zero window probe */
854                                 tcp->tcp_swnd++;
855                                 tcp->tcp_zero_win_probe = B_TRUE;
856                                 TCPS_BUMP_MIB(tcps, tcpOutWinProbe);
857                         } else {
858                                 /*
859                                  * Handle timeout from sender SWS avoidance.
860                                  * Reset our knowledge of the max send window
861                                  * since the receiver might have reduced its
862                                  * receive buffer.  Avoid setting tcp_max_swnd
863                                  * to one since that will essentially disable
864                                  * the SWS checks.
865                                  *
866                                  * Note that since we don't have a SWS
867                                  * state variable, if the timeout is set
868                                  * for ECN but not for SWS, this
869                                  * code will also be executed.  This is
870                                  * fine as tcp_max_swnd is updated
871                                  * constantly and it will not affect
872                                  * anything.
873                                  */
874                                 tcp->tcp_max_swnd = MAX(tcp->tcp_swnd, 2);
875                         }
876                         tcp_wput_data(tcp, NULL, B_FALSE);
877                         return;
878                 }
879                 /* Is there a FIN that needs to be to re retransmitted? */
880                 if ((tcp->tcp_valid_bits & TCP_FSS_VALID) &&
881                     !tcp->tcp_fin_acked)
882                         break;
883                 /* Nothing to do, return without restarting timer. */
884                 TCP_STAT(tcps, tcp_timer_fire_miss);
885                 return;
886         case TCPS_FIN_WAIT_2:
887                 /*
888                  * User closed the TCP endpoint and peer ACK'ed our FIN.
889                  * We waited some time for for peer's FIN, but it hasn't
890                  * arrived.  We flush the connection now to avoid
891                  * case where the peer has rebooted.
892                  */
893                 if (TCP_IS_DETACHED(tcp)) {
894                         (void) tcp_clean_death(tcp, 0);
895                 } else {
896                         TCP_TIMER_RESTART(tcp,
897                             tcp->tcp_fin_wait_2_flush_interval);
898                 }
899                 return;
900         case TCPS_TIME_WAIT:
901                 (void) tcp_clean_death(tcp, 0);
902                 return;
903         default:
904                 if (connp->conn_debug) {
905                         (void) strlog(TCP_MOD_ID, 0, 1, SL_TRACE|SL_ERROR,
906                             "tcp_timer: strange state (%d) %s",
907                             tcp->tcp_state, tcp_display(tcp, NULL,
908                             DISP_PORT_ONLY));
909                 }
910                 return;
911         }

913         /*
914          * If the system is under memory pressure or the max number of
915          * connections have been established for the listener, be more
916          * aggressive in aborting connections.
```



```
917          */
918         if (tcps->tcps_reclaim || (tcp->tcp_listen_cnt != NULL &&
919             tcp->tcp_listen_cnt->tlc_cnt > tcp->tcp_listen_cnt->tlc_max)) {
920                 second_threshold = tcp_early_abort * SECONDS;

922                 /* We will ignore the never timeout promise in this case... */
923                 dont_timeout = B_FALSE;
924         }

926         ASSERT(second_threshold != 0);

928         if ((ms = tcp->tcp_ms_we_have_waited) > second_threshold) {
929                 /*
930                  * Should not hold the zero-copy messages for too long.
931                  */
932                 if (tcp->tcp_snd_zcopy_aware && !tcp->tcp_xmit_zc_clean)
933                         tcp->tcp_xmit_head = tcp_zcopy_backoff(tcp,
934                             tcp->tcp_xmit_head, B_TRUE);

936                 if (dont_timeout) {
937                         /*
938                          * Reset tcp_ms_we_have_waited to avoid overflow since
939                          * we are going to retransmit forever.
940                          */
941                         tcp->tcp_ms_we_have_waited = second_threshold;
942                         goto timer_rexmit;
943                 }

945                 /*
946                  * For zero window probe, we need to send indefinitely,
947                  * unless we have not heard from the other side for some
948                  * time...
949                  */
950                 if ((tcp->tcp_zero_win_probe == 0) ||
951                     (TICK_TO_MSEC(ddi_get_lbolt() - tcp->tcp_last_recv_time) >
952                     second_threshold)) {
953                         TCPS_BUMP_MIB(tcps, tcpTimRetransDrop);
954                         /*
955                          * If TCP is in SYN_RCVD state, send back a
956                          * RST|ACK as BSD does.  Note that tcp_zero_win_probe
957                          * should be zero in TCPS_SYN_RCVD state.
958                          */
959                         if (tcp->tcp_state == TCPS_SYN_RCVD) {
960                                 tcp_xmit_ctl("tcp_timer: RST sent on timeout "
961                                     "in SYN_RCVD",
962                                     tcp, tcp->tcp_snxt,
963                                     tcp->tcp_rnxt, TH_RST | TH_ACK);
964                         }
965                         (void) tcp_clean_death(tcp,
966                             tcp->tcp_client_errno ?
967                             tcp->tcp_client_errno : ETIMEDOUT);
968                         return;
969                 } else {
970                         /*
971                          * If the system is under memory pressure, we also
972                          * abort connection in zero window probing.
973                          */
974                         if (tcps->tcps_reclaim) {
975                                 (void) tcp_clean_death(tcp,
976                                     tcp->tcp_client_errno ?
977                                     tcp->tcp_client_errno : ETIMEDOUT);
978                                 TCP_STAT(tcps, tcp_zwin_mem_drop);
979                                 return;
980                         }
981                         /*
982                          * Set tcp_ms_we_have_waited to second_threshold
```

```
 983                         * so that in next timeout, we will do the above
 984                         * check (ddi_get_lbolt() - tcp_last_recv_time).
 985                         * This is also to avoid overflow.
 986                         *
 987                         * We don't need to decrement tcp_timer_backoff
 988                         * to avoid overflow because it will be decremented
 989                         * later if new timeout value is greater than
 990                         * tcp_rto_max.  In the case when tcp_rto_max is
 991                         * greater than second_threshold, it means that we
 992                         * will wait longer than second_threshold to send
 993                         * the next
 994                         * window probe.
 995                         */
 996                        tcp->tcp_ms_we_have_waited = second_threshold;
 997                }
 998        } else if (ms > first_threshold) {
 999                /*
1000                 * Should not hold the zero-copy messages for too long.
1001                 */
1002                if (tcp->tcp_snd_zcopy_aware && !tcp->tcp_xmit_zc_clean)
1003                        tcp->tcp_xmit_head = tcp_zcopy_backoff(tcp,
1004                            tcp->tcp_xmit_head, B_TRUE);

1006                /*
1007                 * We have been retransmitting for too long...  The RTT
1008                 * we calculated is probably incorrect.  Reinitialize it.
1009                 * Need to compensate for 0 tcp_rtt_sa.  Reset
1010                 * tcp_rtt_update so that we won't accidentally cache a
1011                 * bad value.  But only do this if this is not a zero
1012                 * window probe.
1013                 */
1014                if (tcp->tcp_rtt_sa != 0 && tcp->tcp_zero_win_probe == 0) {
1015                        tcp->tcp_rtt_sd += (tcp->tcp_rtt_sa >> 3) +
1016                            (tcp->tcp_rtt_sa >> 5);
1017                        tcp->tcp_rtt_sa = 0;
1018                        tcp_ip_notify(tcp);
1019                        tcp->tcp_rtt_update = 0;
1020                }
1021        }

1023 timer_rexmit:
1024        tcp->tcp_timer_backoff++;
1025        if ((ms = (tcp->tcp_rtt_sa >> 3) + tcp->tcp_rtt_sd +
1026            tcps->tcps_rexmit_interval_extra + (tcp->tcp_rtt_sa >> 5)) <
1027            tcp->tcp_rto_min) {
1028                /*
1029                 * This means the original RTO is tcp_rexmit_interval_min.
1030                 * So we will use tcp_rexmit_interval_min as the RTO value
1031                 * and do the backoff.
1032                 */
1033                ms = tcp->tcp_rto_min << tcp->tcp_timer_backoff;
1034        } else {
1035                ms <<= tcp->tcp_timer_backoff;
1036        }
1037        if (ms > tcp->tcp_rto_max) {
1038                ms = tcp->tcp_rto_max;
1039                /*
1040                 * ms is at max, decrement tcp_timer_backoff to avoid
1041                 * overflow.
1042                 */
1043                tcp->tcp_timer_backoff--;
1044        }
1045        tcp->tcp_ms_we_have_waited += ms;
1046        if (tcp->tcp_zero_win_probe == 0) {
1047                tcp->tcp_rto = ms;
1048        }
```



```
1049        TCP_TIMER_RESTART(tcp, ms);
1050        /*
1051         * This is after a timeout and tcp_rto is backed off.  Set
1052         * tcp_set_timer to 1 so that next time RTO is updated, we will
1053         * restart the timer with a correct value.
1054         */
1055        tcp->tcp_set_timer = 1;
1056        mss = tcp->tcp_snxt - tcp->tcp_suna;
1057        if (mss > tcp->tcp_mss)
1058                mss = tcp->tcp_mss;
1059        if (mss > tcp->tcp_swnd && tcp->tcp_swnd != 0)
1060                mss = tcp->tcp_swnd;

1062        if ((mp = tcp->tcp_xmit_head) != NULL)
1063                mp->b_prev = (mblk_t *)ddi_get_lbolt();
1064        mp = tcp_xmit_mp(tcp, mp, mss, NULL, NULL, tcp->tcp_suna, B_TRUE, &mss,
1065            B_TRUE);

1067        /*
1068         * When slow start after retransmission begins, start with
1069         * this seq no.  tcp_rexmit_max marks the end of special slow
1070         * start phase.
1069         * start phase.  tcp_snd_burst controls how many segments
1070         * can be sent because of an ack.
1071         */
1072        tcp->tcp_rexmit_nxt = tcp->tcp_suna;
1073        tcp->tcp_snd_burst = TCP_CWND_SS;
1073        if ((tcp->tcp_valid_bits & TCP_FSS_VALID) &&
1074            (tcp->tcp_unsent == 0)) {
1075                tcp->tcp_rexmit_max = tcp->tcp_fss;
1076        } else {
1077                tcp->tcp_rexmit_max = tcp->tcp_snxt;
1078        }
1079        tcp->tcp_rexmit = B_TRUE;
1080        tcp->tcp_dupack_cnt = 0;

1082        /*
1083         * Remove all rexmit SACK blk to start from fresh.
1084         */
1085        if (tcp->tcp_snd_sack_ok)
1086                TCP_NOTSACK_REMOVE_ALL(tcp->tcp_notsack_list, tcp);
1087        if (mp == NULL) {
1088                return;
1089        }

1091        tcp->tcp_csuna = tcp->tcp_snxt;
1092        TCPS_BUMP_MIB(tcps, tcpRetransSegs);
1093        TCPS_UPDATE_MIB(tcps, tcpRetransBytes, mss);
1094        tcp_send_data(tcp, mp);

1096 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   29962 Thu Nov  6 17:28:20 2014
new/usr/src/uts/common/inet/tcp_impl.h
5295 remove maxburst logic from TCP's send algorithm Reviewed by: Dan McDonald <
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2011, Joyent Inc. All rights reserved.
 * Copyright (c) 2013, OmniTI Computer Consulting, Inc. All rights reserved.
 * Copyright (c) 2013, 2014 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 */

#ifndef _INET_TCP_IMPL_H
#define _INET_TCP_IMPL_H

/*
 * TCP implementation private declarations.  These interfaces are
 * used to build the IP module and are not meant to be accessed
 * by any modules except IP itself.  They are undocumented and are
 * subject to change without notice.
 */

#ifdef  __cplusplus
extern "C" {
#endif

#ifdef _KERNEL

#include <sys/cpuvar.h>
#include <sys/clock_impl.h>     /* For LBOLT_FASTPATH{,64} */
#include <inet/optcom.h>
#include <inet/tcp.h>
#include <inet/tunables.h>

#define TCP_MOD_ID      5105

extern struct qinit     tcp_sock_winit;
extern struct qinit     tcp_winit;

extern sock_downcalls_t sock_tcp_downcalls;

/*
 * Note that by default, the _snd_lowat_fraction tunable controls the value of
 * the transmit low water mark.  TCP_XMIT_LOWATER (and thus the _xmit_lowat
 * property) is only used if the administrator has disabled _snd_lowat_fraction
```



```
 * by setting it to 0.
 */
#define TCP_XMIT_LOWATER        4096
#define TCP_XMIT_HIWATER        49152
#define TCP_RECV_LOWATER        2048
#define TCP_RECV_HIWATER        128000

/*
 * Bind hash list size and has function.  It has to be a power of 2 for
 * hashing.
 */
#define TCP_BIND_FANOUT_SIZE    1024
#define TCP_BIND_HASH(lport) (ntohs(lport) & (TCP_BIND_FANOUT_SIZE - 1))

/*
 * This implementation follows the 4.3BSD interpretation of the urgent
 * pointer and not RFC 1122. Switching to RFC 1122 behavior would cause
 * incompatible changes in protocols like telnet and rlogin.
 */
#define TCP_OLD_URP_INTERPRETATION      1

/* TCP option length */
#define TCPOPT_NOP_LEN          1
#define TCPOPT_MAXSEG_LEN       4
#define TCPOPT_WS_LEN           3
#define TCPOPT_REAL_WS_LEN      (TCPOPT_WS_LEN+1)
#define TCPOPT_TSTAMP_LEN       10
#define TCPOPT_REAL_TS_LEN      (TCPOPT_TSTAMP_LEN+2)
#define TCPOPT_SACK_OK_LEN      2
#define TCPOPT_REAL_SACK_OK_LEN (TCPOPT_SACK_OK_LEN+2)
#define TCPOPT_REAL_SACK_LEN    4
#define TCPOPT_MAX_SACK_LEN     36
#define TCPOPT_HEADER_LEN       2

/* Round up the value to the nearest mss. */
#define MSS_ROUNDUP(value, mss)         ((((value) - 1) / (mss) + 1) * (mss))

/*
 * Was this tcp created via socket() interface?
 */
#define TCP_IS_SOCKET(tcp)      ((tcp)->tcp_issocket)

/*
 * Is this tcp not attached to any upper client?
 */
#define TCP_IS_DETACHED(tcp)    ((tcp)->tcp_detached)

/* TCP timers related data strucutres.  Refer to tcp_timers.c. */
typedef struct tcp_timer_s {
        conn_t  *connp;
        void    (*tcpt_proc)(void *);
        callout_id_t   tcpt_tid;
} tcp_timer_t;
______unchanged_portion_omitted_

/* Increment and decrement the number of connections in tcp_stack_t. */
#define TCPS_CONN_INC(tcps)                                             \
        atomic_inc_64(                                                  \
            (uint64_t *)&(tcps)->tcps_sc[CPU->cpu_seqid]->tcp_sc_conn_cnt)

#define TCPS_CONN_DEC(tcps)                                             \
        atomic_dec_64(                                                  \
            (uint64_t *)&(tcps)->tcps_sc[CPU->cpu_seqid]->tcp_sc_conn_cnt)

/*
 * When the system is under memory pressure, stack variable tcps_reclaim is
```

```
 390  * true, we shorten the connection timeout abort interval to tcp_early_abort
 391  * seconds.  Defined in tcp.c.
 392  */
 393 extern uint32_t tcp_early_abort;

 395 /*
 396  * To reach to an eager in Q0 which can be dropped due to an incoming
 397  * new SYN request when Q0 is full, a new doubly linked list is
 398  * introduced. This list allows to select an eager from Q0 in O(1) time.
 399  * This is needed to avoid spending too much time walking through the
 400  * long list of eagers in Q0 when tcp_drop_q0() is called. Each member of
 401  * this new list has to be a member of Q0.
 402  * This list is headed by listener's tcp_t. When the list is empty,
 403  * both the pointers - tcp_eager_next_drop_q0 and tcp_eager_prev_drop_q0,
 404  * of listener's tcp_t point to listener's tcp_t itself.
 405  *
 406  * Given an eager in Q0 and a listener, MAKE_DROPPABLE() puts the eager
 407  * in the list. MAKE_UNDROPPABLE() takes the eager out of the list.
 408  * These macros do not affect the eager's membership to Q0.
 409  */
 410 #define MAKE_DROPPABLE(listener, eager)                                 \
 411         if ((eager)->tcp_eager_next_drop_q0 == NULL) {                  \
 412                 (listener)->tcp_eager_next_drop_q0->tcp_eager_prev_drop_q0\
 413                     = (eager);                                          \
 414                 (eager)->tcp_eager_prev_drop_q0 = (listener);           \
 415                 (eager)->tcp_eager_next_drop_q0 =                       \
 416                     (listener)->tcp_eager_next_drop_q0;                 \
 417                 (listener)->tcp_eager_next_drop_q0 = (eager);           \
 418         }

 420 #define MAKE_UNDROPPABLE(eager)                                         \
 421         if ((eager)->tcp_eager_next_drop_q0 != NULL) {                  \
 422                 (eager)->tcp_eager_next_drop_q0->tcp_eager_prev_drop_q0 \
 423                     = (eager)->tcp_eager_prev_drop_q0;                  \
 424                 (eager)->tcp_eager_prev_drop_q0->tcp_eager_next_drop_q0 \
 425                     = (eager)->tcp_eager_next_drop_q0;                  \
 426                 (eager)->tcp_eager_prev_drop_q0 = NULL;                 \
 427                 (eager)->tcp_eager_next_drop_q0 = NULL;                 \
 428         }

 430 /*
 431  * The format argument to pass to tcp_display().
 432  * DISP_PORT_ONLY means that the returned string has only port info.
 433  * DISP_ADDR_AND_PORT means that the returned string also contains the
 434  * remote and local IP address.
 435  */
 436 #define DISP_PORT_ONLY          1
 437 #define DISP_ADDR_AND_PORT      2

 439 #define IP_ADDR_CACHE_SIZE      2048
 440 #define IP_ADDR_CACHE_HASH(faddr)                                       \
 441         (ntohl(faddr) & (IP_ADDR_CACHE_SIZE -1))

 443 /* TCP cwnd burst factor. */
 444 #define TCP_CWND_INFINITE       65535
 445 #define TCP_CWND_SS             3
 446 #define TCP_CWND_NORMAL         5

 443 /*
 444  * TCP reassembly macros.  We hide starting and ending sequence numbers in
 445  * b_next and b_prev of messages on the reassembly queue.  The messages are
 446  * chained using b_cont.  These macros are used in tcp_reass() so we don't
 447  * have to see the ugly casts and assignments.
 448  */
 449 #define TCP_REASS_SEQ(mp)               ((uint32_t)(uintptr_t)((mp)->b_next))
 450 #define TCP_REASS_SET_SEQ(mp, u)        ((mp)->b_next = \
```



```
 451                                         (mblk_t *)(uintptr_t)(u))
 452 #define TCP_REASS_END(mp)               ((uint32_t)(uintptr_t)((mp)->b_prev))
 453 #define TCP_REASS_SET_END(mp, u)        ((mp)->b_prev = \
 454                                         (mblk_t *)(uintptr_t)(u))

 456 #define tcps_time_wait_interval         tcps_propinfo_tbl[0].prop_cur_uval
 457 #define tcps_conn_req_max_q             tcps_propinfo_tbl[1].prop_cur_uval
 458 #define tcps_conn_req_max_q0            tcps_propinfo_tbl[2].prop_cur_uval
 459 #define tcps_conn_req_min               tcps_propinfo_tbl[3].prop_cur_uval
 460 #define tcps_conn_grace_period          tcps_propinfo_tbl[4].prop_cur_uval
 461 #define tcps_cwnd_max_                  tcps_propinfo_tbl[5].prop_cur_uval
 462 #define tcps_dbg                        tcps_propinfo_tbl[6].prop_cur_uval
 463 #define tcps_smallest_nonpriv_port      tcps_propinfo_tbl[7].prop_cur_uval
 464 #define tcps_ip_abort_cinterval         tcps_propinfo_tbl[8].prop_cur_uval
 465 #define tcps_ip_abort_linterval         tcps_propinfo_tbl[9].prop_cur_uval
 466 #define tcps_ip_abort_interval          tcps_propinfo_tbl[10].prop_cur_uval
 467 #define tcps_ip_notify_cinterval        tcps_propinfo_tbl[11].prop_cur_uval
 468 #define tcps_ip_notify_interval         tcps_propinfo_tbl[12].prop_cur_uval
 469 #define tcps_ipv4_ttl                   tcps_propinfo_tbl[13].prop_cur_uval
 470 #define tcps_keepalive_interval_high    tcps_propinfo_tbl[14].prop_max_uval
 471 #define tcps_keepalive_interval         tcps_propinfo_tbl[14].prop_cur_uval
 472 #define tcps_keepalive_interval_low     tcps_propinfo_tbl[14].prop_min_uval
 473 #define tcps_maxpsz_multiplier          tcps_propinfo_tbl[15].prop_cur_uval
 474 #define tcps_mss_def_ipv4               tcps_propinfo_tbl[16].prop_cur_uval
 475 #define tcps_mss_max_ipv4               tcps_propinfo_tbl[17].prop_cur_uval
 476 #define tcps_mss_min                    tcps_propinfo_tbl[18].prop_cur_uval
 477 #define tcps_naglim_def                 tcps_propinfo_tbl[19].prop_cur_uval
 478 #define tcps_rexmit_interval_initial_high       \
 479                                         tcps_propinfo_tbl[20].prop_max_uval
 480 #define tcps_rexmit_interval_initial    tcps_propinfo_tbl[20].prop_cur_uval
 481 #define tcps_rexmit_interval_initial_low        \
 482                                         tcps_propinfo_tbl[20].prop_min_uval
 483 #define tcps_rexmit_interval_max_high   tcps_propinfo_tbl[21].prop_max_uval
 484 #define tcps_rexmit_interval_max        tcps_propinfo_tbl[21].prop_cur_uval
 485 #define tcps_rexmit_interval_max_low    tcps_propinfo_tbl[21].prop_min_uval
 486 #define tcps_rexmit_interval_min_high   tcps_propinfo_tbl[22].prop_max_uval
 487 #define tcps_rexmit_interval_min        tcps_propinfo_tbl[22].prop_cur_uval
 488 #define tcps_rexmit_interval_min_low    tcps_propinfo_tbl[22].prop_min_uval
 489 #define tcps_deferred_ack_interval      tcps_propinfo_tbl[23].prop_cur_uval
 490 #define tcps_snd_lowat_fraction         tcps_propinfo_tbl[24].prop_cur_uval
 491 #define tcps_dupack_fast_retransmit     tcps_propinfo_tbl[25].prop_cur_uval
 492 #define tcps_ignore_path_mtu            tcps_propinfo_tbl[26].prop_cur_bval
 493 #define tcps_smallest_anon_port         tcps_propinfo_tbl[27].prop_cur_uval
 494 #define tcps_largest_anon_port          tcps_propinfo_tbl[28].prop_cur_uval
 495 #define tcps_xmit_hiwat                 tcps_propinfo_tbl[29].prop_cur_uval
 496 #define tcps_xmit_lowat                 tcps_propinfo_tbl[30].prop_cur_uval
 497 #define tcps_recv_hiwat                 tcps_propinfo_tbl[31].prop_cur_uval
 498 #define tcps_recv_hiwat_minmss          tcps_propinfo_tbl[32].prop_cur_uval
 499 #define tcps_fin_wait_2_flush_interval_high     \
 500                                         tcps_propinfo_tbl[33].prop_max_uval
 501 #define tcps_fin_wait_2_flush_interval  tcps_propinfo_tbl[33].prop_cur_uval
 502 #define tcps_fin_wait_2_flush_interval_low      \
 503                                         tcps_propinfo_tbl[33].prop_min_uval
 504 #define tcps_max_buf                    tcps_propinfo_tbl[34].prop_cur_uval
 505 #define tcps_strong_iss                 tcps_propinfo_tbl[35].prop_cur_uval
 506 #define tcps_rtt_updates                tcps_propinfo_tbl[36].prop_cur_uval
 507 #define tcps_wscale_always              tcps_propinfo_tbl[37].prop_cur_bval
 508 #define tcps_tstamp_always              tcps_propinfo_tbl[38].prop_cur_bval
 509 #define tcps_tstamp_if_wscale           tcps_propinfo_tbl[39].prop_cur_bval
 510 #define tcps_rexmit_interval_extra      tcps_propinfo_tbl[40].prop_cur_uval
 511 #define tcps_deferred_acks_max          tcps_propinfo_tbl[41].prop_cur_uval
 512 #define tcps_slow_start_after_idle      tcps_propinfo_tbl[42].prop_cur_uval
 513 #define tcps_slow_start_initial         tcps_propinfo_tbl[43].prop_cur_uval
 514 #define tcps_sack_permitted             tcps_propinfo_tbl[44].prop_cur_uval
 515 #define tcps_ipv6_hoplimit              tcps_propinfo_tbl[45].prop_cur_uval
 516 #define tcps_mss_def_ipv6               tcps_propinfo_tbl[46].prop_cur_uval
```

```
#define tcps_mss_max_ipv6               tcps_propinfo_tbl[47].prop_cur_uval
#define tcps_rev_src_routes             tcps_propinfo_tbl[48].prop_cur_bval
#define tcps_local_dack_interval        tcps_propinfo_tbl[49].prop_cur_uval
#define tcps_local_dacks_max            tcps_propinfo_tbl[50].prop_cur_uval
#define tcps_ecn_permitted              tcps_propinfo_tbl[51].prop_cur_uval
#define tcps_rst_sent_rate_enabled      tcps_propinfo_tbl[52].prop_cur_bval
#define tcps_rst_sent_rate              tcps_propinfo_tbl[53].prop_cur_uval
#define tcps_push_timer_interval        tcps_propinfo_tbl[54].prop_cur_uval
#define tcps_use_smss_as_mss_opt        tcps_propinfo_tbl[55].prop_cur_bval
#define tcps_keepalive_abort_interval_high \
                                        tcps_propinfo_tbl[56].prop_max_uval
#define tcps_keepalive_abort_interval \
                                        tcps_propinfo_tbl[56].prop_cur_uval
#define tcps_keepalive_abort_interval_low \
                                        tcps_propinfo_tbl[56].prop_min_uval
#define tcps_wroff_xtra                 tcps_propinfo_tbl[57].prop_cur_uval
#define tcps_dev_flow_ctl               tcps_propinfo_tbl[58].prop_cur_bval
#define tcps_reass_timeout              tcps_propinfo_tbl[59].prop_cur_uval
#define tcps_iss_incr                   tcps_propinfo_tbl[65].prop_cur_uval

extern struct qinit tcp_rinitv4, tcp_rinitv6;
extern boolean_t do_tcp_fusion;

/*
 * Object to represent database of options to search passed to
 * {sock,tpi}optcom_req() interface routine to take care of option
 * management and associated methods.
 */
extern optdb_obj_t      tcp_opt_obj;
extern uint_t           tcp_max_optsize;

extern int tcp_squeue_flag;

extern uint_t tcp_free_list_max_cnt;

/*
 * Functions in tcp.c.
 */
extern void     tcp_acceptor_hash_insert(t_uscalar_t, tcp_t *);
extern tcp_t    *tcp_acceptor_hash_lookup(t_uscalar_t, tcp_stack_t *);
extern void     tcp_acceptor_hash_remove(tcp_t *);
extern mblk_t   *tcp_ack_mp(tcp_t *);
extern int      tcp_build_hdrs(tcp_t *);
extern void     tcp_cleanup(tcp_t *);
extern int      tcp_clean_death(tcp_t *, int);
extern void     tcp_clean_death_wrapper(void *, mblk_t *, void *,
                    ip_recv_attr_t *);
extern void     tcp_close_common(conn_t *, int);
extern void     tcp_close_detached(tcp_t *);
extern void     tcp_close_mpp(mblk_t **);
extern void     tcp_closei_local(tcp_t *);
extern sock_lower_handle_t tcp_create(int, int, int, sock_downcalls_t **,
                    uint_t *, int *, int, cred_t *);
extern conn_t   *tcp_create_common(cred_t *, boolean_t, boolean_t, int *);
extern void     tcp_disconnect(tcp_t *, mblk_t *);
extern char     *tcp_display(tcp_t *, char *, char);
extern int      tcp_do_bind(conn_t *, struct sockaddr *, socklen_t, cred_t *,
                    boolean_t);
extern int      tcp_do_connect(conn_t *, const struct sockaddr *, socklen_t,
                    cred_t *, pid_t);
extern int      tcp_do_listen(conn_t *, struct sockaddr *, socklen_t, int,
                    cred_t *, boolean_t);
extern int      tcp_do_unbind(conn_t *);
extern boolean_t        tcp_eager_blowoff(tcp_t *, t_scalar_t);
extern void     tcp_eager_cleanup(tcp_t *, boolean_t);
extern void     tcp_eager_kill(void *, mblk_t *, void *, ip_recv_attr_t *);
```



```
extern void     tcp_eager_unlink(tcp_t *);
extern void     tcp_init_values(tcp_t *, tcp_t *);
extern void     tcp_ipsec_cleanup(tcp_t *);
extern int      tcp_maxpsz_set(tcp_t *, boolean_t);
extern void     tcp_mss_set(tcp_t *, uint32_t);
extern void     tcp_reinput(conn_t *, mblk_t *, ip_recv_attr_t *, ip_stack_t *);
extern void     tcp_rsrv(queue_t *);
extern uint_t   tcp_rwnd_reopen(tcp_t *);
extern int      tcp_rwnd_set(tcp_t *, uint32_t);
extern int      tcp_set_destination(tcp_t *);
extern void     tcp_set_ws_value(tcp_t *);
extern void     tcp_stop_lingering(tcp_t *);
extern void     tcp_update_pmtu(tcp_t *, boolean_t);
extern mblk_t   *tcp_zcopy_backoff(tcp_t *, mblk_t *, boolean_t);
extern boolean_t        tcp_zcopy_check(tcp_t *);
extern void     tcp_zcopy_notify(tcp_t *);
extern void     tcp_get_proto_props(tcp_t *, struct sock_proto_props *);

/*
 * Bind related functions in tcp_bind.c
 */
extern int      tcp_bind_check(conn_t *, struct sockaddr *, socklen_t,
                    cred_t *, boolean_t);
extern void     tcp_bind_hash_insert(tf_t *, tcp_t *, int);
extern void     tcp_bind_hash_remove(tcp_t *);
extern in_port_t        tcp_bindi(tcp_t *, in_port_t, const in6_addr_t *,
                    int, boolean_t, boolean_t, boolean_t);
extern in_port_t        tcp_update_next_port(in_port_t, const tcp_t *,
                    boolean_t);

/*
 * Fusion related functions in tcp_fusion.c.
 */
extern void     tcp_fuse(tcp_t *, uchar_t *, tcpha_t *);
extern void     tcp_unfuse(tcp_t *);
extern boolean_t tcp_fuse_output(tcp_t *, mblk_t *, uint32_t);
extern void     tcp_fuse_output_urg(tcp_t *, mblk_t *);
extern boolean_t tcp_fuse_rcv_drain(queue_t *, tcp_t *, mblk_t **);
extern size_t   tcp_fuse_set_rcv_hiwat(tcp_t *, size_t);
extern int      tcp_fuse_maxpsz(tcp_t *);
extern void     tcp_fuse_backenable(tcp_t *);
extern void     tcp_iss_key_init(uint8_t *, int, tcp_stack_t *);

/*
 * Output related functions in tcp_output.c.
 */
extern void     tcp_close_output(void *, mblk_t *, void *, ip_recv_attr_t *);
extern void     tcp_output(void *, mblk_t *, void *, ip_recv_attr_t *);
extern void     tcp_output_urgent(void *, mblk_t *, void *, ip_recv_attr_t *);
extern void     tcp_rexmit_after_error(tcp_t *);
extern void     tcp_sack_rexmit(tcp_t *, uint_t *);
extern void     tcp_send_data(tcp_t *, mblk_t *);
extern void     tcp_send_synack(void *, mblk_t *, void *, ip_recv_attr_t *);
extern void     tcp_shutdown_output(void *, mblk_t *, void *, ip_recv_attr_t *);
extern void     tcp_ss_rexmit(tcp_t *);
extern void     tcp_update_xmit_tail(tcp_t *, uint32_t);
extern void     tcp_wput(queue_t *, mblk_t *);
extern void     tcp_wput_data(tcp_t *, mblk_t *, boolean_t);
extern void     tcp_wput_sock(queue_t *, mblk_t *);
extern void     tcp_wput_fallback(queue_t *, mblk_t *);
extern void     tcp_xmit_ctl(char *, tcp_t *, uint32_t, uint32_t, int);
extern void     tcp_xmit_listeners_reset(mblk_t *, ip_recv_attr_t *,
                    ip_stack_t *i, conn_t *);
extern mblk_t   *tcp_xmit_mp(tcp_t *, mblk_t *, int32_t, int32_t *,
                    mblk_t **, uint32_t, boolean_t, uint32_t *, boolean_t);
```

```
649 /*
650  * Input related functions in tcp_input.c.
651  */
652 extern void     tcp_icmp_input(void *, mblk_t *, void *, ip_recv_attr_t *);
653 extern void     tcp_input_data(void *, mblk_t *, void *, ip_recv_attr_t *);
654 extern void     tcp_input_listener_unbound(void *, mblk_t *, void *,
655                     ip_recv_attr_t *);
656 extern boolean_t        tcp_paws_check(tcp_t *, tcpha_t *, tcp_opt_t *);
657 extern uint_t   tcp_rcv_drain(tcp_t *);
658 extern void     tcp_rcv_enqueue(tcp_t *, mblk_t *, uint_t, cred_t *);
659 extern boolean_t        tcp_verifyicmp(conn_t *, void *, icmph_t *, icmp6_t *,
660                             ip_recv_attr_t *);

662 /*
663  * Kernel socket related functions in tcp_socket.c.
664  */
665 extern int      tcp_fallback(sock_lower_handle_t, queue_t *, boolean_t,
666                     so_proto_quiesced_cb_t, sock_quiesce_arg_t *);
667 extern boolean_t tcp_newconn_notify(tcp_t *, ip_recv_attr_t *);

669 /*
670  * Timer related functions in tcp_timers.c.
671  */
672 extern void     tcp_ack_timer(void *);
673 extern void     tcp_close_linger_timeout(void *);
674 extern void     tcp_keepalive_timer(void *);
675 extern void     tcp_push_timer(void *);
676 extern void     tcp_reass_timer(void *);
677 extern mblk_t   *tcp_timermp_alloc(int);
678 extern void     tcp_timermp_free(tcp_t *);
679 extern timeout_id_t tcp_timeout(conn_t *, void (*)(void *), hrtime_t);
680 extern clock_t  tcp_timeout_cancel(conn_t *, timeout_id_t);
681 extern void     tcp_timer(void *arg);
682 extern void     tcp_timers_stop(tcp_t *);

684 /*
685  * TCP TPI related functions in tcp_tpi.c.
686  */
687 extern void     tcp_addr_req(tcp_t *, mblk_t *);
688 extern void     tcp_capability_req(tcp_t *, mblk_t *);
689 extern boolean_t        tcp_conn_con(tcp_t *, uchar_t *, mblk_t *,
690                             mblk_t **, ip_recv_attr_t *);
691 extern void     tcp_err_ack(tcp_t *, mblk_t *, int, int);
692 extern void     tcp_err_ack_prim(tcp_t *, mblk_t *, int, int, int);
693 extern void     tcp_info_req(tcp_t *, mblk_t *);
694 extern void     tcp_send_conn_ind(void *, mblk_t *, void *);
695 extern void     tcp_send_pending(void *, mblk_t *, void *, ip_recv_attr_t *);
696 extern void     tcp_tpi_accept(queue_t *, mblk_t *);
697 extern void     tcp_tpi_bind(tcp_t *, mblk_t *);
698 extern int      tcp_tpi_close(queue_t *, int);
699 extern int      tcp_tpi_close_accept(queue_t *);
700 extern void     tcp_tpi_connect(tcp_t *, mblk_t *);
701 extern int      tcp_tpi_opt_get(queue_t *, t_scalar_t, t_scalar_t, uchar_t *);
702 extern int      tcp_tpi_opt_set(queue_t *, uint_t, int, int, uint_t, uchar_t *,
703                     uint_t *, uchar_t *, void *, cred_t *);
704 extern void     tcp_tpi_unbind(tcp_t *, mblk_t *);
705 extern void     tcp_tli_accept(tcp_t *, mblk_t *);
706 extern void     tcp_use_pure_tpi(tcp_t *);
707 extern void     tcp_do_capability_ack(tcp_t *, struct T_capability_ack *,
708                     t_uscalar_t);

710 /*
711  * TCP option processing related functions in tcp_opt_data.c
712  */
713 extern int      tcp_opt_get(conn_t *, int, int, uchar_t *);
714 extern int      tcp_opt_set(conn_t *, uint_t, int, int, uint_t, uchar_t *,
```



```
715                     uint_t *, uchar_t *, void *, cred_t *);

717 /*
718  * TCP time wait processing related functions in tcp_time_wait.c.
719  */
720 extern void             tcp_time_wait_append(tcp_t *);
721 extern void             tcp_time_wait_collector(void *);
722 extern boolean_t        tcp_time_wait_remove(tcp_t *, tcp_squeue_priv_t *);
723 extern void             tcp_time_wait_processing(tcp_t *, mblk_t *, uint32_t,
724                             uint32_t, int, tcpha_t *, ip_recv_attr_t *);

726 /*
727  * Misc functions in tcp_misc.c.
728  */
729 extern uint32_t tcp_find_listener_conf(tcp_stack_t *, in_port_t);
730 extern void     tcp_ioctl_abort_conn(queue_t *, mblk_t *);
731 extern void     tcp_listener_conf_cleanup(tcp_stack_t *);
732 extern void     tcp_stack_cpu_add(tcp_stack_t *, processorid_t);

734 #endif  /* _KERNEL */

736 #ifdef  __cplusplus
737 }
```

_____***unchanged_portion_omitted_***