

TCP Data structures

```
248 struct tcp_opt {
249     int    tcp_header_len; /* Bytes of tcp header to send    */
250
251 /*
252 *   Header prediction flags
253 *   0x5?10 << 16 + snd_wnd in net byte order
254 */
255     __u32  pred_flags;
256
257 /*
258 *   RFC793 variables by their proper names. This means you can
259 *   read the code and the spec side by side (and laugh ...)
260 *   See RFC793 and RFC1122. The RFC writes these in capitals.
261 */
262     __u32  rcv_nxt;    /* What we want to receive next    */
263     __u32  snd_nxt;    /* Next sequence we send          */
264
265     __u32  snd_una;    /* First byte we want an ack for    */
266     __u32  snd_sml;    /* Last byte of the most recently transmitted small packet */
267     __u32  rcv_tstamp; /* timestamp of last received ACK (for keepalives) */
268     __u32  lsndtime;   /* timestamp of last sent data packet (for restart window) */
269
270     /* Delayed ACK control data */
271     struct {
272         __u8  pending;    /* ACK is pending */
273         __u8  quick;      /* Scheduled number of quick acks    */
274         __u8  pingpong;   /* The session is interactive        */
275         __u8  blocked;    /* Delayed ACK was blocked by socket lock */
276         __u32  ato;        /* Predicted tick of soft clock      */
277         unsigned long timeout; /* Currently scheduled timeout    */
278         __u32  lrcvtime;   /* timestamp of last received data packet */
279         __u16  last_seg_size; /* Size of last incoming segment    */
280         __u16  rcv_mss;    /* MSS used for delayed ACK decisions */
281     } ack;
282
```

```

283 /* Data for direct copy to user */
284 struct {
285     struct sk_buff_head  prequeue;
286     int                  memory;
287     struct task_struct  *task;
288     struct iovec        *iov;
289     int                  len;
290 } ucopy;
291
292 __u32  snd_wll;          /* Sequence for window update      */
293 __u32  snd_wnd;         /* The window we expect to receive */
294 __u32  max_window;     /* Maximal window ever seen from peer */
295 __u32  pmtu_cookie;    /* Last pmtu seen by socket        */
296 __u16  mss_cache;      /* Cached effective mss, not including SACKS */
297 __u16  mss_clamp;     /* Maximal mss, negotiated at connection setup */
298 __u16  ext_header_len; /* Network protocol overhead (IP/IPv6 options) */
299 __u8   ca_state;       /* State of fast-retransmit machine */
300 __u8   retransmits;    /* Number of unrecovered RTO timeouts. */
301
302 __u8   reordering;     /* Packet reordering metric.      */
303 __u8   queue_shrunk;   /* Write queue has been shrunk recently. */
304 __u8   defer_accept;   /* User waits for some data after accept() */
305
306 /* RTT measurement */
307 __u8   backoff;        /* backoff                          */
308 __u32  srtt;           /* smothed round trip time << 3    */
309 __u32  mdev;          /* medium deviation                  */
310 __u32  mdev_max;     /* maximal mdev for the last rtt period */
311 __u32  rttvar;        /* smoothed mdev_max                */
312 __u32  rtt_seq;       /* sequence number to update rttvar */
313 __u32  rto;           /* retransmit timeout                */
314
315 __u32  packets_out;   /* Packets which are "in flight"    */
316 __u32  left_out;      /* Packets which leaved network     */
317 __u32  retrans_out;   /* Retransmitted packets out        */
318
319

```

```

320 /*
321 *   Slow start and congestion control (see also Nagle, and Karn & Partridge)
322 */
323   __u32  snd_ssthresh;           /* Slow start size threshold */
324   __u32  snd_cwnd;              /* Sending congestion window */
325   __u16  snd_cwnd_cnt;          /* Linear increase counter */
326   __u16  snd_cwnd_clamp;        /* Do not allow snd_cwnd to grow above this */
327   __u32  snd_cwnd_used;
328   __u32  snd_cwnd_stamp;
329
330   /* Two commonly used timers in both sender and receiver paths. */
331   unsigned long      timeout;
332   struct timer_list  retransmit_timer; /* Resend (no ack) */
333   struct timer_list  delack_timer;     /* Ack delay */
334
335   struct sk_buff_head  out_of_order_queue; /* Out of order segments go here */
336
337   struct tcp_func      *af_specific; /* Operations which are AF_INET{4,6} specific */
338   struct sk_buff      *send_head; /* Front of stuff to transmit */
339   struct page          *sndmsg_page; /* Cached page for sendmsg */
340   u32                  sndmsg_off; /* Cached offset for sendmsg */
341
342   __u32  rcv_wnd;           /* Current receiver window */
343   __u32  rcv_wup;          /* rcv_nxt on last window update sent */
344   __u32  write_seq;        /* Tail(+1) of data held in tcp send buffer */
345   __u32  pushed_seq;       /* Last pushed seq, required to talk to windows */
346   __u32  copied_seq;       /* Head of yet unread data */

```

```

347 /*
348 *   Options received (usually on last packet, some only on SYN packets).
349 */
350 char  tstamp_ok,    /* TIMESTAMP seen on SYN packet    */
351       wscale_ok,    /* Wscale seen on SYN packet        */
352       sack_ok;      /* SACK seen on SYN packet          */
353 char  saw_tstamp;   /* Saw TIMESTAMP on last packet     */
354 __u8  snd_wscale;   /* Window scaling received from sender */
355 __u8  rcv_wscale;   /* Window scaling to send to receiver */
356 __u8  nonagle;      /* Disable Nagle algorithm?         */
357 __u8  keepalive_probes; /* num of allowed keep alive probes */
358
359 /*   PAWS/RTTM data */
360 __u32 rcv_tsval;    /* Time stamp value                  */
361 __u32 rcv_tsecr;    /* Time stamp echo reply             */
362 __u32 ts_recent;    /* Time stamp to echo next           */
363 long  ts_recent_stamp; /* Time we stored ts_recent (for aging) */
364
365 /*   SACKs data */
366 __u16 user_mss;     /* mss requested by user in ioctl */
367 __u8  dsack;        /* D-SACK is scheduled              */
368 __u8  eff_sacks;    /* Size of SACK array to send with next packet */
369 struct tcp_sack_block duplicate_sack[1]; /* D-SACK block */
370 struct tcp_sack_block selective_acks[4]; /* The SACKS themselves */
371
372 __u32 window_clamp; /* Maximal window to advertise     */
373 __u32 rcv_ssthresh; /* Current window clamp             */
374 __u8  probes_out;   /* unanswered 0 window probes       */
375 __u8  num_sacks;    /* Number of SACK blocks             */
376 __u16 advmss;      /* Advertised MSS                    */
377
378 __u8  syn_retries;  /* num of allowed syn retries */
379 __u8  ecn_flags;    /* ECN status bits.              */
380 __u16 prior_ssthresh; /* ssthresh saved at recovery start */
381 __u32 lost_out;     /* Lost packets                    */
382 __u32 sacked_out;   /* SACK'd packets                  */
383 __u32 fackets_out;  /* FACK'd packets                  */
384 __u32 high_seq;     /* snd_nxt at onset of congestion */

```

```
385
386  __u32  retrans_stamp;  /* Timestamp of the last retransmit,
387                          * also used in SYN-SENT to remember stamp of
388                          * the first SYN. */
389  __u32  undo_marker;    /* tracking retrans started here. */
390  int    undo_retrans;   /* number of undoable retransmissions. */
391  __u32  urg_seq;        /* Seq of received urgent pointer */
392  __u16  urg_data;       /* Saved octet of OOB data and control flags */
393  __u8   pending;        /* Scheduled timer event      */
394  __u8   urg_mode;       /* In urgent mode              */
395  __u32  snd_up;         /* Urgent pointer              */
```

```
396
397 /* The syn_wait_lock is necessary only to avoid tcp_get_info having
398 * to grab the main lock sock while browsing the listening hash
399 * (otherwise it's deadlock prone).
400 * This lock is acquired in read mode only from tcp_get_info() and
401 * it's acquired in write mode _only_ from code that is actively
402 * changing the syn_wait_queue. All readers that are holding
403 * the master sock lock don't need to grab this lock in read mode
404 * too as the syn_wait_queue writes are always protected from
405 * the main sock lock.
406 */
407 rlock_t          syn_wait_lock;
408 struct tcp_listen_opt *listen_opt;
409
410 /* FIFO of established children */
411 struct open_request *accept_queue;
412 struct open_request *accept_queue_tail;
413
414 int              write_pending; /* A write to socket waits to start. */
415
416 unsigned int     keepalive_time; /* time before keep alive takes place */
417 unsigned int     keepalive_intvl; /* time interval between keep alive probes */
418 int              linger2;
419
420 unsigned long last_synq_overflow;
421 };
```

The *tcp* header

```
23 struct tcphdr {
24     __u16    source;
25     __u16    dest;
26     __u32    seq;
27     __u32    ack_seq;
28 #if defined(__LITTLE_ENDIAN_BITFIELD)
29     __u16    res1:4,
30             doff:4,
31             fin:1,
32             syn:1,
33             rst:1,
34             psh:1,
35             ack:1,
36             urg:1,
37             ece:1,
38             cwr:1;
39 #elif defined(__BIG_ENDIAN_BITFIELD)
40     __u16    doff:4,
41             res1:4,
42             cwr:1,
43             ece:1,
44             urg:1,
45             ack:1,
46             psh:1,
47             rst:1,
48             syn:1,
49             fin:1;
50 #else
51 #error "Adjust your <asm/byteorder.h> defines"
52 #endif
53     __u16    window;
54     __u16    check;
55     __u16    urg_ptr;
56 };
```