Networking In the Linux Kernel (kerNet)

Lecture 11

TCP Structures In the Kernel, Assortment of Sockets

Announcements

- Peer 1 Reviews available
  - Click “View Grade” by Peer Review Return: Assignment 1
  - Download your PDFs
- Assignment 2 due Thursday night
  - GFP_USER is not your friend!
  - Pre-allocate your 2-D array in user-space
  - Static arrays may also do weird things: char** x; and then manual allocation usually works out better than char x[256][256].

Assignment 2 Discussion

- Why does the sleep() cause issues?

Testing Discussion

- “Basic arguments” – What does this mean?
- “I ran the standard test” - What does this mean?
- “When I tried with invalid arguments” – What does this mean?
- When you run multiple tests in a row, it’s possible they could be related! But the ordering of tests (and the exact details) were rarely shared
- “Multiple clients” / “Ran clients and killed the server” – when? How many clients? Did some/all of them finish writing?

Testing Discussion

- Screenshots are not required but can be helpful
- Exact commands and the order you run them in are also useful
- “I tried with values of x from 0 to 5” … is that
  - 0, 3, 5?
  - 0,1,2,3,4,5?
  - ???
- OS, compiler version, etc. can be helpful but I’ll consider “extra”
- If you can’t reproduce a bug, it’s still worth providing as much detail as you can – perhaps the original coder will have insight / will be able to run your tests many times and see the bug happen.

Testing Discussion

- Very few students had enough description to be sure I was reproducing their tests correctly (especially when there were multiple runs required).
- I'll be much pickier in future peer reviews about your explanation of testing!
- Communication intensive course – and being able to communicate precisely about testing is important (arguably more so than essay-style writing will be in most of your careers)

Sources for Today

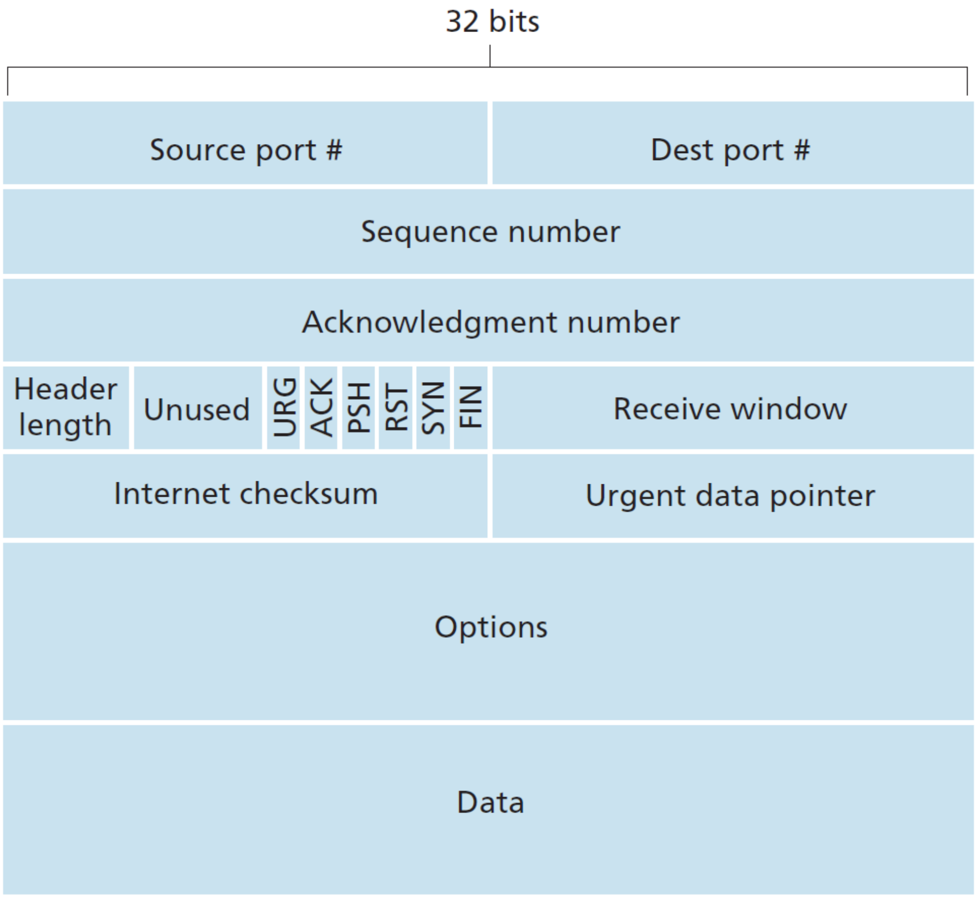
- Kurose, James F. and Ross, W. Keith. Computer Networking A Top-Down Approach. Pearson Education Inc., 2006.
  - Basically just for the TCP header

Quick Teaser About Windows…

```
/* Update our send window.
 *
 * Window update algorithm, described in RFC793/RFC1122 (used in linux-2.2
 * and in FreeBSD. NetBSD's one is even worse.) is wrong.
 */
static int tcp_ack_update_window(struct sock *sk, const struct sk_buff *skb, u32
ack, u32 ack_seq)
```

TCP Header

TCP Segment Structure



32 bits

Source port #

Dest port #

Sequence number

Acknowledgment number

Header length

Unused

URG

ACK

PSH

RST

SYN

FIN

Receive window

Internet checksum

Urgent data pointer

Options

Data

Figure 3.29 ♦ TCP segment structure

A Few Header Notes

- This gives us the basic header layout
- We haven’t discussed options
- We might have a variable number of TCP options.
  - Not all options have the same length

- Header length is 4 bits, representing the number of 4-byte words the header-with-options takes
  - Without options, we’re looking at 20 bytes (5 words), you’ll often see 0x5
  - Remember this is a half-byte, so it’ll get mashed together with “unused” to yield 0x50. With RFC 3540 you might see 0x51, 9th flag bit!

Header in the Kernel

- *tcp_hdr(skb)* returns a pointer to the transport header data in skb
- The return type is *struct tcphdr** defined in <linux/uapi/tcp.h>
- Let’s pull it up and compare it to the TCP Segment Structure graphic from earlier

tcphdr (1/3)

```
struct tcphdr {
        __be16      source;
        __be16      dest;
        __be32      seq;
        __be32      ack_seq;
```

tcphdr (2/3)

```
#elif defined(__BIG_ENDIAN_BITFIELD)
        __u16  doff:4,
               res1:4,
               cwr:1,
               ece:1,
               urg:1,
               ack:1,
               psh:1,
               rst:1,
               syn:1,
               fin:1;
```

tcphdr (3/3)

```
#endif
        __be16      window;
        __sum16     check;
        __be16      urg_ptr;
};
```

tcphdr enum

```
enum {
        TCP_FLAG_CWR = __constant_cpu_to_be32(0x00800000),
        TCP_FLAG_ECE = __constant_cpu_to_be32(0x00400000),
        TCP_FLAG_URG = __constant_cpu_to_be32(0x00200000),
        TCP_FLAG_ACK = __constant_cpu_to_be32(0x00100000),
        TCP_FLAG_PSH = __constant_cpu_to_be32(0x00080000),
        TCP_FLAG_RST = __constant_cpu_to_be32(0x00040000),
        TCP_FLAG_SYN = __constant_cpu_to_be32(0x00020000),
        TCP_FLAG_FIN = __constant_cpu_to_be32(0x00010000),
        TCP_RESERVED_BITS = __constant_cpu_to_be32(0x0F000000),
        TCP_DATA_OFFSET = __constant_cpu_to_be32(0xF0000000)
};  [source]
```

What About TCP Options?

- A bunch of #defines for option numbers and lengths in <net/tcp.h>
- Look for the definition of TCPOPT_SACK_PERM to find the corresponding blocks
- net/ipv4/tcp_input.c: tcp_parse_options()
  - We normally only do this during handshakes (SYN flag set), slowpath parsing
  - First attempts tcp_fast_parse_options() which only expects a timestamp
  - You’ll see *doff* used here, get used to it - this is the data offset.
- If you walk further up the call chain, tcp_fast_parse_options() or tcp_conn_request() or tcp_rcv_fastopen_synack() or tcp_synsent_state_process()
- Fast path only called through tcp_validate_incoming() which is…
  - Getting off track (for now), a lot about PAWS/RST here
  - Called by tcp_rcv_established() and tcp_rcv_state_process()

TCP Socket Structure

tcp_sock (1/25)

```
struct tcp_sock {
        /* inet_connection_sock has to be the first member of tcp_sock */
        struct inet_connection_sock   inet_conn;
        u16     tcp_header_len;      /* Bytes of tcp header to send                 */
        u16     gso_segs;       /* Max number of segs per GSO packet       */

/*
 *      Header prediction flags
 *      0x5?10 << 16 + snd_wnd in net byte order
 */
        __be32          pred_flags;
```

Detour: inet_connection_sock

```
struct inet_connection_sock {
        /* inet_sock has to be the first member! */
        struct inet_sock  icsk_inet;
      …
}

struct inet_sock {
        /* sk and pinet6 has to be the first two members of inet_sock */
        struct sock               sk;
#if IS_ENABLED(CONFIG_IPV6)
        struct ipv6_pinfo         *pinet6;
#endif
```

Detour: sock

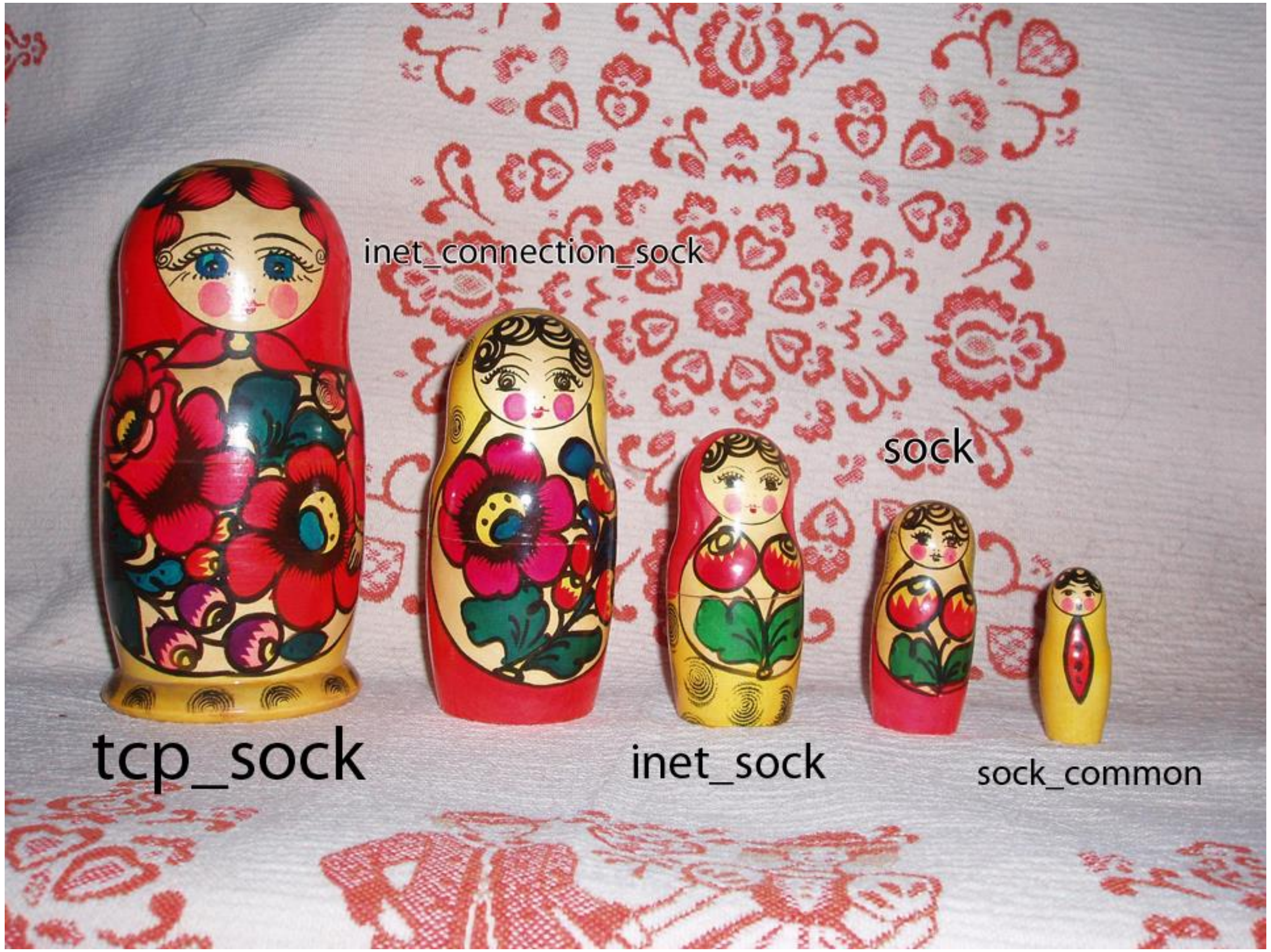
```
struct sock {
        /*
         * Now struct inet_timewait_sock also uses sock_common, so please just
         * don't add nothing before this first member (__sk_common) --acme
         */
        struct sock_common      __sk_common;




/**
 *      struct sock_common - minimal network layer representation of sockets
```

tcp_sock -> sock_common

Modified from https://en.wikipedia.org/wiki/Matryoshka_doll#/media/File:Russian-Matroshka.jpg, CC BY-SA 3.0



tcp_sock (2/25)

```
/*
 *          RFC793 variables by their proper names. This means you can
 *          read the code and the spec side by side (and laugh ...)
 *          See RFC793 and RFC1122. The RFC writes these in capitals.
 */
            u64         bytes_received;       /* RFC4898 tcpEStatsAppHCThruOctetsReceived
                                               * sum(delta(rcv_nxt)), or how many bytes
                                               * were acked.
                                               */
            u32         segs_in;    /* RFC4898 tcpEStatsPerfSegsIn
                                               * total number of segments in.
                                               */
            u32         data_segs_in;         /* RFC4898 tcpEStatsPerfDataSegsIn
                                               * total number of data segments in.
                                               */
```

tcp_sock (3/25)

```
u32   rcv_nxt;      /* What we want to receive next           */
u32   copied_seq;  /* Head of yet unread data                 */
u32   rcv_wup;      /* rcv_nxt on last window update sent
*/
u32   snd_nxt;      /* Next sequence we send                  */
u32   segs_out;     /* RFC4898 tcpEStatsPerfSegsOut
                     * The total number of segments sent.
                     */
```

tcp_sock (4/25)

```
u32     data_segs_out;   /* RFC4898 tcpEStatsPerfDataSegsOut
                          * total number of data segments sent.
                          */
u64     bytes_sent;      /* RFC4898 tcpEStatsPerfHCDataOctetsOut
                          * total number of data bytes sent.
                          */
u64     bytes_acked;     /* RFC4898 tcpEStatsAppHCThruOctetsAcked
                          * sum(delta(snd_una)), or how many bytes
                          * were acked.
                          */
u32     dsack_dups;      /* RFC4898 tcpEStatsStackDSACKDups
                          * total number of DSACK blocks received
                          */
```

tcp_sock (5/25)

```
        u32     snd_una;        /* First byte we want an ack for         */
        u32     snd_sml;        /* Last byte of the most recently transmitted small
packet */
        u32     rcv_tstamp;     /* timestamp of last received ACK (for keepalives) */
        u32     lsndtime;       /* timestamp of last sent data packet (for restart
window) */
        u32     last_oow_ack_time;  /* timestamp of last out-of-window ACK */
        u32     compressed_ack_rcv_nxt;
```

- Minshall’s modification to Nagle’s algorithm: [here]
- TCP SACK Compression added in 4.19.7: [discussion]

tcp_sock (6/25)

```
        u32     tsoffset;       /* timestamp offset */

        struct list_head tsq_node; /* anchor in tsq_tasklet.head list */
        struct list_head tsorted_sent_queue; /* time-sorted sent but un-SACKed skbs */

        /* Data for direct copy to user */
        struct ucopy; <----- no longer exists, see [prequeue discussion]
```

Detour: tsq?

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

TSQ Supplemental Reading

- https://lwn.net/Articles/507065/
  - Has a link to the patch proposal, which is at: https://lwn.net/Articles/506237/

tcp_sock (7/25)

```
u32     snd_wl1;       /* Sequence for window update            */
u32     snd_wnd;       /* The window we expect to receive       */
u32     max_window;            /* Maximal window ever seen from peer
*/
u32     mss_cache;     /* Cached effective mss, not including SACKS */


u32     window_clamp;          /* Maximal window to advertise
*/
u32     rcv_ssthresh; /* Current window clamp                   */
```

SND.WL2?

- From RFC793:
  - SND.WL1 - segment sequence number used for last window update
  - SND.WL2 - segment acknowledgment number used for last window update
  - These plus some other variables should be in a “Transmission Control Block, or TCB” but we have them in *tcp_sock*
  - Not to be confused with *tcp_skb_cb* (discussed next time)
- Claim: WL2 is redundant
  - Reading: https://www.ietf.org/mail-archive/web/tsvwg/current/msg03445.html

tcp_sock (8/25)

```
/* Information of the most recently (s)acked skb */
struct tcp_rack {
        struct skb_mstamp mstamp; /* (Re)sent time of the skb */
        u8 advanced; /* mstamp advanced since last lost marking */
        u8 reord;    /* reordering detected */
} rack;
```

See also, comment above tcp_rack_detect_loss() and draft-ietf-tcpm-rack-01

Lots of work on this – two years ago we were on revision 05, last year revision 11.

Current version is a full RFC: RFC 8985

tcp_sock (9/25)

```
u16    advmss;                 /* Advertised MSS                         */
u8     compressed_ack;
u8     dup_ack_counter:2,
       tlp_retrans:1,  /* TLP is a retransmission */
       unused:5;
u32    chrono_start;  /* Start time in jiffies of a TCP chrono */
u32    chrono_stat[3];         /* Time in jiffies for chrono_stat stats */
u8     chrono_type:2,          /* current chronograph type */
       rate_app_limited:1,  /* rate_{delivered,interval_us} limited? */
       ….
```

tcp_sock (10/25)

```
u8      nonagle     : 4,/* Disable Nagle algorithm?           */
        thin_lto    : 1,/* Use linear timeouts for thin streams */
        thin_dupack : 1,/* Fast retransmit on first dupack     */
        repair      : 1,
        frto        : 1;/* F-RTO (RFC5682) activated in CA_Loss */
```

This part actually comes AFTER the next slide, just putting it here for space reasons:

```
u32     tcp_tx_delay;  /* delay (in usec) added to TX packets */
u64     tcp_wstamp_ns;        /* departure time for next sent data packet */
u64     tcp_clock_cache; /* cache last tcp_clock_ns() (see
tcp_mstamp_refresh()) */
```

tcp_sock (11/25)

```
u8      repair_queue;
u8      do_early_retrans:1,/* Enable RFC5827 early-retransmit  */
        syn_data:1,     /* SYN includes data */
        syn_fastopen:1,         /* SYN includes Fast Open option */
        syn_fastopen_exp:1,/* SYN includes Fast Open exp. option */
        syn_data_acked:1,/* data in SYN is acked by SYN-ACK */
        save_syn:1,     /* Save headers of SYN packet */
        is_cwnd_limited:1;/* forward progress limited by snd_cwnd? */
u32     tlp_high_seq;  /* snd_nxt at the time of TLP retransmit. */
```

tcp_sock (12/25)

```
/* RTT measurement */
        u32     srtt_us;        /* smoothed round trip time << 3 in usecs */
        u32     mdev_us;        /* medium deviation                      */
        u32     mdev_max_us;    /* maximal mdev for the last rtt period
        */
        u32     rttvar_us;      /* smoothed mdev_max                     */
        u32     rtt_seq;        /* sequence number to update rttvar      */
        struct  minmax rtt_min;
```

tcp_sock (13/25)

```
u32     packets_out;  /* Packets which are "in flight"      */
u32     retrans_out;  /* Retransmitted packets out                  */
u32     max_packets_out;  /* max packets_out in last window */
u32     max_packets_seq;  /* right edge of max_packets_out flight */

u16     urg_data;        /* Saved octet of OOB data and control flags */
u8      ecn_flags;       /* ECN status bits.                      */
u8      keepalive_probes; /* num of allowed keep alive probes*/
u32     reordering;      /* Packet reordering metric.                  */
u32     snd_up;                      /* Urgent pointer                */
```

tcp_sock (14/25)

```
/*
 *     Options received (usually on last packet, some only on SYN packets).
 */
                struct tcp_options_received rx_opt;

/*
 *              Slow start and congestion control (see also Nagle, and Karn & Partridge)
 */
                u32         snd_ssthresh;           /* Slow start size threshold                    */
                u32         snd_cwnd;  /* Sending congestion window                    */
                u32         snd_cwnd_cnt;           /* Linear increase counter                      */
                u32         snd_cwnd_clamp; /* Do not allow snd_cwnd to grow above this */
                u32         snd_cwnd_used;
                u32         snd_cwnd_stamp;
```

tcp_sock (15/25)

```
u32    prior_cwnd;    /* Congestion window at start of Recovery. */
u32    prr_delivered; /* Number of newly delivered packets to
                       * receiver in Recovery. */
u32    prr_out;       /* Total number of pkts sent during Recovery. */
u32    delivered;     /* Total data packets delivered incl. rexmits */
u32    lost;          /* Total data packets lost incl. rexmits */
u32    app_limited;   /* limited until "delivered" reaches this val */
u64    first_tx_mstamp;  /* start of window send phase */
u64    delivered_mstamp; /* time we reached "delivered" */
u32    rate_delivered;    /* saved rate sample: packets delivered */
u32    rate_interval_us;  /* saved rate sample: time elapsed */
```

See [net/ipv4/tcp_rate.c] for more on rate variables

tcp_sock (16/25)

```
        u32     rcv_wnd;        /* Current receiver window                */
        u32     write_seq;      /* Tail(+1) of data held in tcp send buffer */
        u32     notsent_lowat;          /* TCP_NOTSENT_LOWAT */
        u32     pushed_seq;/* Last pushed seq, required to talk to
windows */
        u32     lost_out;       /* Lost packets                           */
        u32     sacked_out; /* SACK'd packets                             */
        u32     fackets_out; /* FACK'd packets                            */
```

tcp_sock (17/25)

```
struct hrtimer          pacing_timer;
struct hrtimer          compressed_ack_timer;

/* from STCP, retrans queue hinting */
struct sk_buff* lost_skb_hint;
struct sk_buff *retransmit_skb_hint;

/* OOO segments go in this rbtree. Socket lock must be held. */
struct rb_root          out_of_order_queue;
struct sk_buff*ooo_last_skb; /* cache rb_last(out_of_order_queue) */
```

tcp_sock (18/25)

```
/* SACKs data, these 2 need to be together (see tcp_options_write) */
struct tcp_sack_block duplicate_sack[1]; /* D-SACK block */
struct tcp_sack_block selective_acks[4]; /* The SACKS themselves*/

struct tcp_sack_block recv_sack_cache[4];

struct sk_buff *highest_sack;   /* skb just after the highest
                                * skb with SACKed bit set
                                * (validity guaranteed only if
                                * sacked_out > 0)
                                */
```

tcp_sock (19/25)

```
int    lost_cnt_hint;

u32      prior_ssthresh; /* ssthresh saved at recovery start    */
u32      high_seq;        /* snd_nxt at onset of congestion       */

u32      retrans_stamp;           /* Timestamp of the last retransmit,
                                   * also used in SYN-SENT to remember stamp of
                                   * the first SYN. */
u32      undo_marker; /* snd_una upon a new recovery episode. */
int      undo_retrans; /* number of undoable retransmissions. */
```

tcp_sock (20/25)

```
u64         bytes_retrans;      /* RFC4898 tcpEStatsPerfOctetsRetrans
                                 * Total data bytes retransmitted
                                 */
u32         total_retrans;      /* Total retransmits for entire connection */

u32         urg_seq;/* Seq of received urgent pointer */
unsigned int                    keepalive_time;     /* time before keep alive takes place */
unsigned int                    keepalive_intvl;  /* time interval between keep alive probes */

int                             linger2;
```

tcp_sock (21/25)

```
        u16 timeout_rehash;  /* Timeout-triggered rehash attempts */

        u32 rcv_ooopack; /* Received out-of-order packets, for tcpinfo */

/* Receiver side RTT estimation */
        u32 rcv_rtt_last_tsecr;
        struct {
                u32     rtt;
                u32     seq;
                u32     time;
        } rcv_rtt_est;
```

tcp_sock (22/25)

```
/* Receiver queue space */
        struct {
                int     space;
                u32     seq;
                u32     time;
        } rcvq_space;
```

tcp_sock (23/25)

```
/* TCP-specific MTU probe information. */
        struct {
                u32             probe_seq_start;
                u32             probe_seq_end;
        } mtu_probe;
        u32     mtu_info; /* We received an ICMP_FRAG_NEEDED /
ICMPV6_PKT_TOOBIG
                          * while socket was owned by user.
                          */
```

tcp_sock (24/25)

```
#ifdef CONFIG_TCP_MD5SIG
/* TCP AF-Specific parts; only used by MD5 Signature support so far */
        const struct tcp_sock_af_ops     *af_specific;

/* TCP MD5 Signature Option information */
        struct tcp_md5sig_info    __rcu *md5sig_info;
#endif
```

tcp_sock (25/25)

```
/* TCP fastopen related information */
        struct tcp_fastopen_request *fastopen_req;
        /* fastopen_rsk points to request_sock that resulted in this big
         * socket. Used to retransmit SYNACKs etc.
         */
        struct request_sock *fastopen_rsk;
        u32     *saved_syn;
};
```

See also: RFC 7413

For Next Time

- Monday: TCP CBs, Connection and States
  - Assignment 2 Peer Review will also be assigned
- Thursday: TCP Congestion Avoidance

Looking forward:

- Monday (10/25): Projects 1 AND 2 will be released
- Monday 10/25, Thursday 10/28: In-class time for Project 1
- Monday 11/1 – Monday 11/8: Project 1 Presentations (these are in-person) and Project 1 Peer Reviews (presentation feedback)