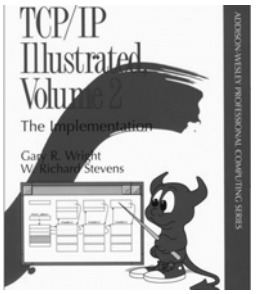
Internet Programming & Protocols Lecture 25

TCP implementation
- history
- kernel networking
- BSD TCP
- Linux TCP



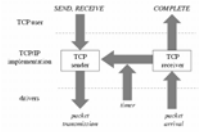
www.cs.utk.edu/~dunigan/ipp/

TCP genealogy

- BSD 4.1 ('83) incorporates DARPA TCP/IP protocol stack
- BSD descendants
  - NetBSD, FreeBSD, OpenBSD
  - SunOS/Solaris
  - IBM's AIX
  - MAC OS
  - Mach
- Other OS's variations from RFC's or peeking at BSD sources
  - System V → Unicos
  - SGI Irix
  - DEC TOPS10/TOPS20 (DARPA)
  - Microsoft/DOS PC implementations
  - Note: IBM, Cray, and SGI have Linux options
- Linux descends from PC instructional OS/network Minix

Kernel network stacks

- Given von Neuman computer architecture and traditional multi-user OS, the basic components and operation of a network protocol stack in the kernel will be about the same regardless of protocol or OS
- OS drivers for network interfaces
- OS support for pre-emptive scheduling, queue management, memory mgt., locks, threads, timers
- Network layer software for addressing and routing
- Transport layer software for port addressing, packet handling
- Event driven
  - Application requests to send data
    - System calls
  - Device interrupts
    - Packet arrivals or transmit completion
  - Timer events
    - Retransmit timeout, connection timeout, IP assembly timeout, delayed ACK



Queues and buffers

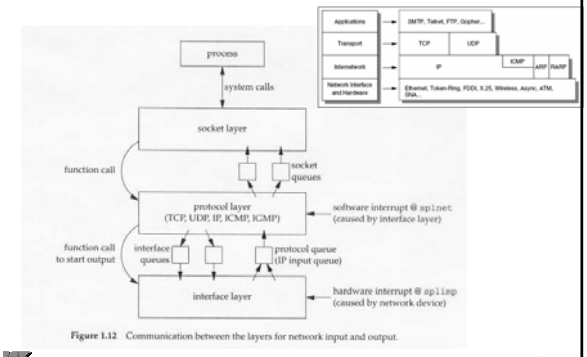


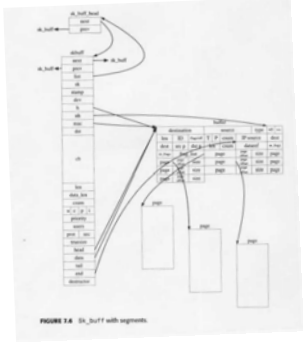
Figure 1.12 Communication between the layers for network input and output.

Network memory management

- Device interrupt handler needs memory buffer for incoming packets
- When application sends data, memory buffer in kernel needed to hold data til ACK'd
- Message buffers are constantly being acquired, queued, and released
- Headers for TCP, IP, and Ethernet need to be added (or removed) from data portion
- Memory copying is expensive so do most of the work with pointers
- Devices can scatter read/write so you can have separate buffers for headers and data
  - A chain of mbufs makes up a "segment"
- Performance issues in acquiring/releasing
  - Heap? Pre-allocated fixed size buffers (small, medium, large)? Dynamic?
  - Optimizations: cache aligned, page aligned

Linux network buffers

- Linux uses cache aligned and page aligned buffers (sk_buff)



BSD mbuf

- BSD uses 128-byte mbuf or 2K mbuf
- Example, header mbuf pre-pended to data mbuf's preparing for send

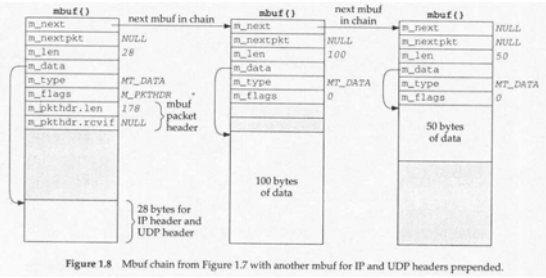


Figure 1.8 Mbuf chain from Figure 1.7 with another mbuf for IP and UDP headers prepended.

Mbuf message queues

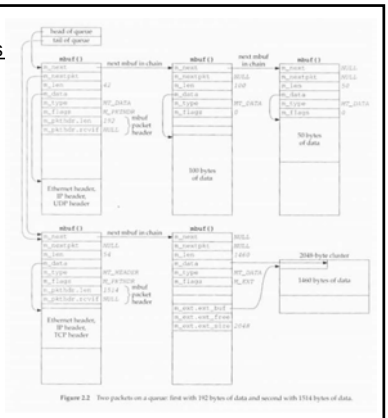


Figure 2.2 Two packets on a queue: first with 192 bytes of data and second with 1514 bytes of data.

UDP receive

- mbuf after Ethernet has received a UDP packet

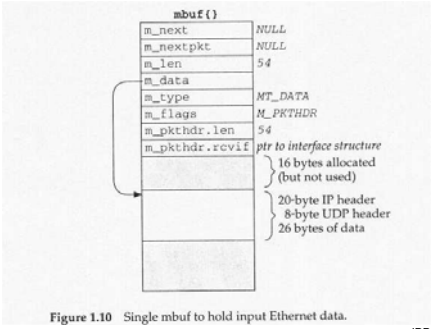


Figure 1.10 Single mbuf to hold input Ethernet data.

UDP receive

- Data portion of UDP packet is queued to application receive buffer
- After data is passed to application, mbuf's are released

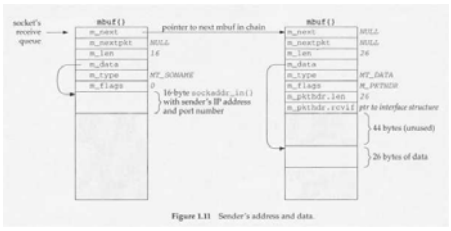


Figure 1.11 Sender's address and data.

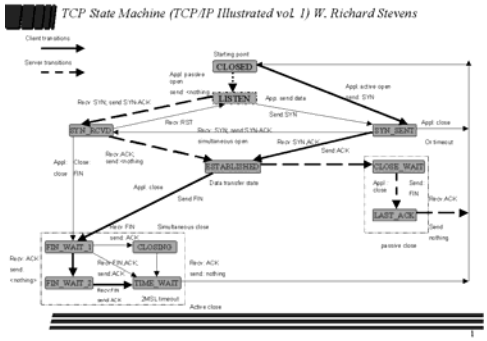
Transmission Control Protocol (TCP)

- TCP RFC 793 '81
- Provides a reliable stream of bytes on top of unreliable IP datagrams
- Connection oriented (circuit like)
- 16-bit port number (service)
- Stateful with timers, sequence numbers, flow control, congestion mgt.

4.4BSD lite (Stevens TCP/IP illustrated v2)
UDP: 9 functions, 800 lines of C code
TCP: 28 functions, 4500 lines of C code

Linux 2.6
UDP 1044 lines of C code
TCP 13050 lines of C code

TCP finite state machine



TCP state transitions

- There is a lot of code in the kernel (tcp_input.c) to manage the TCP state
- Establish and close a connection
  - SYN/SYN-ACK plus all the option negotiation

```
case TCPOPT_MAXSEG:
                        if (optlen != TCPOLEN_MAXSEG)
                                continue;
                        if (!(ti->ti_flags & TH_SYN))
                                continue;
                        bcopy((char *) cp + 2, (char *) &mss, sizeof(mss));
                        NTOHS(mss);
                        (void) tcp_mss(tp, mss);        /* sets t_maxseg */
                        break;
```

  - FIN and timeouts for closing
- In the source files you'll see explicit references to current "state"

```
switch (tp->t_state) {
 case TCPS_TIME_WAIT:
                    tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
                    _
 case TCPS_SYN_RECEIVED:
 case TCPS_ESTABLISHED:
                    tp->t_state = TCPS_CLOSE_WAIT;
                    break;
```


sending a TCP segment

- write() is a system call, passes address of user buffer to kernel
- If enough room in socket's SNDBUF, copies user message to kernel space, if not enough room, application may be blocked … eventually your application write() is "complete"
- TCP constructs MSS-sized packets from the SNDBUF, building TCP header with ACK and sequence numbers, may need to start timer
- Kernel constructs IP packet and calculates checksums (IP and TCP)
- IP layer looks up destination address in routing table, and may need to issue ARP request (asynchronous event)
- With Ether address of destination, construct Ether packet and queue to ether driver (packet could be dropped if TXQUE is full)
  - Linux has a queuing layer in front of device queues for traffic shaping etc.
- Ether driver checks TXQUE and sends out the next packet
  - NIC handles CSMA/CD, CRC
  - NIC issues interrupt when transfer complete (optional)

Receiving a TCP data segment (simplified)

- NIC receives datagram with its NIC address in destination address field, issues an interrupt
- Interrupt handler requests Ether driver to read the datagram
- Datagram copied into kernel address space (mbuf)
- Driver inspects Ether type field (IP, ARP) and queues packet to appropriate kernel handler
- (assuming not fragmented), kernel IP handler verifies IP checksum and other IP fields, inspects IP proto field and queues packet payload to TCP handler
- TCP handler verifies checksum and processes the ACK info
  - ACK processing includes, dup ACK, congestion avoidance etc.
- TCP handler adds bytes to socket's RCVBUF, schedules an ACK reply
- If associated process is blocked on read(), process is moved to "ready queue".
- when process runs, data copied into user's buffer

TCP timers & timeouts

- connect timeout: 75s
- delayed-ACK timeout: 200ms
- keepalive: 2 hr+
- retransmit: 3-5+ minutes
- close wait: 30s (2MSL)
- 0-window persist: forever @ 60s
- IP fragment assembly: 30s
- TCP uses a 200ms and 500ms timer to manage the various timeouts.
  - Every 500 ms, check for packet timeouts, bump tick count (10 ms today)
  - RTT estimator uses tick count from 500 ms timer
  - Timestamp is current tick count
  - Every 200 ms, see if any delayed ACKs or Nagle-data need to be transmitted
  - Faster timer (100 ms) can improve TCP performance when there are timeouts
  - Newer OS's have replaced 500 ms timer with 100 ms or 10 ms timer

Timer management

- Hardware interval timers – on interrupt dispatch kernel/TCP handler
  - One tick interrupt, then check all events for all processes
- TCP timer management can be on critical path
  - Timer events are added, deleted, modified
  - Handle timeout event
- Old BSD had slow timer (500 ms) and fast timer (200 ms)
  - On interrupt, handler would walk all the TCP control blocks, decrementing tick values in timer structs. If zero, invoke the proper event handler (retransmit, re-probe, etc)
- Linux has 10ms timer
  - Each TCP timeout control block is individually linked into timers event list in ascending order
  - Doubly-linked list
  - When timer interrupt occurs, process head of list for any expired events and schedule handler

BSD timer control block walk

```
/*
 * Search through tcb's and update active timers.
 */
ip = tcb.inp_next;
if (ip == 0) {
        splx(s);
        return;
}
for (; ip != &tcb; ip = ipnxt) {
        ipnxt = ip->inp_next;
        tp = intotcpcb(ip);
        if (tp == 0)
                continue;
        for (i = 0; i < TCPT_NTIMERS; i++) {
                if (tp->t_timer[i] && --tp->t_timer[i] == 0) {
                        (void) tcp_usrreq(tp->t_inpcb->inp_socket,
                            PRU_SLOWTIMO, (struct mbuf *)0,
                             (struct mbuf *)i, (struct mbuf *)0);
                        if (ipnxt->inp_prev != ip)
                                goto tpgone;
                }
        }
        tp->t_idle++;
```

Linux retransmit timeout

- Handler called by timer interrupt routine, pointer to skb
- Update cwnd/ssthresh, retransmit packet ,and do exponential backoff

```
                tcp_enter_loss(sk, 0);

                tcp_retransmit_skb(sk, skb_peek(&sk->sk_write_queue));

                __sk_dst_reset(sk);

                goto out_reset_timer;

out_reset_timer:

        tp->rto = min(tp->rto << 1, TCP_RTO_MAX);

        tcp_reset_xmit_timer(sk, TCP_TIME_RETRANS, tp->rto);
```


TCP data structures

- Flow control with send and receiver buffers
- Hold data at send side til ACKd
- Out-of-order queue at receiver
- Buffers are really a series of linked mbufs

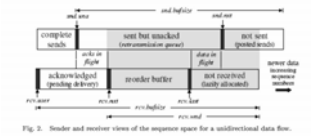


Fig. 2. Sender and receiver views of the sequence space for a unidirectional data flow.

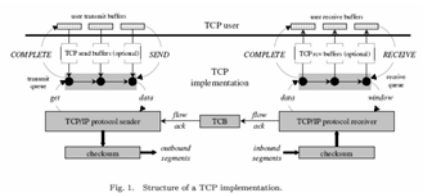
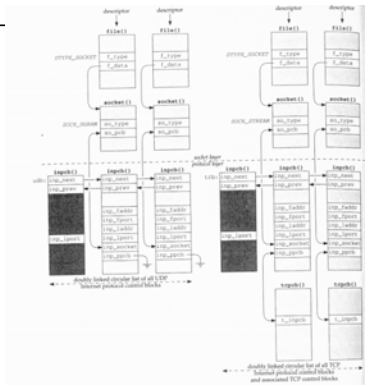


Fig. 1. Structure of a TCP implementation.

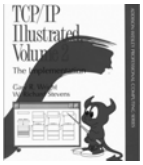
Control blocks



Kernel data associated with a TCP connection

- Kernel data structure associated with socket descriptor for active TCP connection
  - Send and receive buffers and control info
  - Sliding window control info (left edge, right edge)
  - Send and ACK sequence numbers (last sent, last ACKd)
  - State info (SYN-sent, ESTABLISHED, etc.)
  - Option info (nagle, so_reuse, etc)
  - Timer info, retry counters etc.
  - RTT variables
  - Urgent pointer stuff
  - Congestion control info (cwnd, ssthresh)
- C structs in kernel sources
  - Linux: tcp_opt
  - BSD: tcpcb

Peeking at the TCP source code



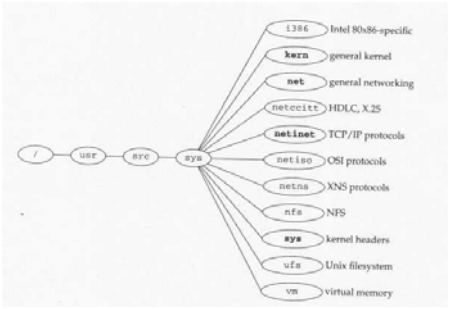
- ns (tcp.cc)
- In ~dunigan/ipp05/OS-tcp/
  - Linux
  - 4.4BSD (stevens)
  - FreeBSD
  - MAC OS
  - Solaris
- The key data structure is the socket struct or TCP control block for each flow
  - Linux: tcp_opt sock
  - BSD: tcpcb
- Most of the action is in
  - tcp_input.c
  - tcp_output.c

The main source files

- tcp_output.c
  - Send one segment
  - Send multiple segments (whatever cwnd will allow)
  - Retransmit one segment (maybe using SACK info)
  - Send ACK, delayed ACK, SACK info, FIN, RST
  - Send window probes
  - Nagle and silly window syndrome avoidance
- tcp_input.c
  - Handle response to ACK (dup ACKs, retransmit, AIMD, SACK)
  - Do RTT estimation (and Vegas/Westwood bandwidth estimation)
  - Queue incoming data to receiver or to out-of-order queue
  - For linux, handle “un do” for out of order packets

BSD TCP

- Source tree



BSD socket struct (tcp_var.h)

```
struct tcpcb {
        struct  tcpiphdr *seg_next;     /* sequencing queue */
        struct  tcpiphdr *seg_prev;
        short   t_state;                /* state of this connection */
        short   t_timer[TCPT_NTIMERS];  /* tcp timers */
        short   t_rxtshift;             /* log(2) of rexmt exp. backoff */
        short   t_rxtcur;               /* current retransmit value */
        short   t_dupacks;              /* consecutive dup acks recd */
        u_short t_maxseg;               /* maximum segment size */
/* send sequence variables */
        tcp_seq snd_una;                /* send unacknowledged */
        tcp_seq snd_nxt;                /* send next */
        tcp_seq snd_up;                 /* send urgent pointer */
        tcp_seq snd_wl1;                /* window update seg seq number */
        tcp_seq snd_wl2;                /* window update seg ack number */
        tcp_seq iss;                    /* initial send sequence number */
        u_long  snd_wnd;                /* send window */
/* receive sequence variables */
        u_long  rcv_wnd;                /* receive window */
        tcp_seq rcv_nxt;                /* receive next */
        tcp_seq rcv_up;                 /* receive urgent pointer */
        tcp_seq irs;                    /* initial receive sequence number */
```



```
/* congestion control (for slow start, source quench, retransmit after loss) */
        u_long  snd_cwnd;               /* congestion-controlled window */
        u_long  snd_ssthresh;           /* snd_cwnd size threshhold for
                                         * for slow start exponential to
                                         * linear switch
                                         */
/*
 * transmit timing stuff.  See below for scale of srtt and rttvar.
 * "Variance" is actually smoothed difference.
 */
        short   t_idle;                 /* inactivity time */
        short   t_rtt;                  /* round trip time */
        tcp_seq t_rtseq;                /* sequence number being timed */
        short   t_srtt;                 /* smoothed round-trip time */
        short   t_rttvar;               /* variance in round-trip time */
        u_short t_rttmin;               /* minimum rtt allowed */
        u_long  max_sndwnd;             /* largest window peer has offered */
/* RFC 1323 variables */
        u_char  snd_scale;              /* window scaling for send window */
        u_char  rcv_scale;              /* window scaling for recv window */
        u_char  request_r_scale;        /* pending window scaling */
        u_char  requested_s_scale;
        u_long  ts_recent;              /* timestamp echo data */
        u_long  ts_recent_age;          /* when last updated */
        tcp_seq last_ack_sent;
```


TCP output

- Mbufs for output

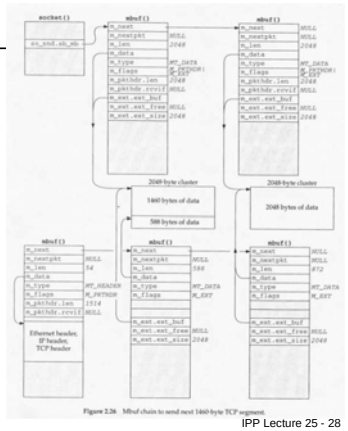


Figure 2.26 Mbuf chain to send next 1460-byte TCP segment.

TCP output

- Set retransmit timer if not set

```
if (tp->t_timer[TCPT_REXMT] == 0 &&
                    tp->snd_nxt != tp->snd_una) {
                        tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;
                        if (tp->t_timer[TCPT_PERSIST]) {
                                tp->t_timer[TCPT_PERSIST] = 0;
                                tp->t_rxtshift = 0;
                        }
                }
```

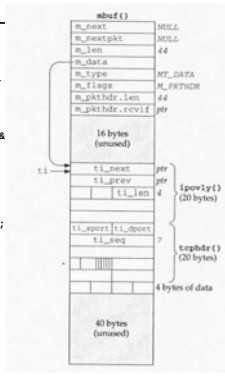

TCP input

- Received data handed to tcp_input.c
- If data is in sequence, copy to receive buffer and "notify" application

```
if ((ti)->ti_seq == (tp)->rcv_nxt &&
        (tp)->seg_next == (struct tcpiphdr *)(tp) &&
           (tp)->t_state == TCPS_ESTABLISHED) {
                tp->t_flags |= TF_DELACK;
                (tp)->rcv_nxt += (ti)->ti_len;
                flags = (ti)->ti_flags & TH_FIN;
                tcpstat.tcps_rcvpack++;
                tcpstat.tcps_rcvbyte += (ti)->ti_len;
                sbappend(&(so)->so_rcv, (m));
                sorwakeup(so);
```

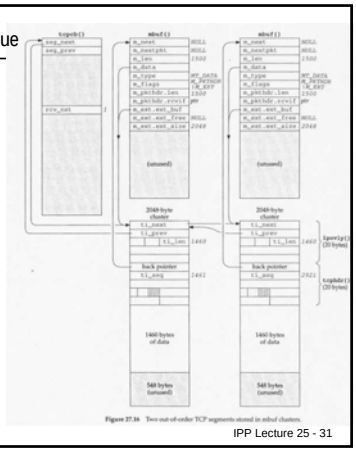
- If out of order, add to out of order queue (overlap?)

```
for (q = tp->seg_next; q != (struct tcpiphdr *)tp;
            q = (struct tcpiphdr *)q->ti_next)
                if (SEQ_GT(q->ti_seq, ti->ti_seq))
                        break;
```



TCP out-of-order receive queue

- Doubly linked list



3 dup ACKs

```
if (++tp->t_dupacks == tcprexmtthresh) {
                                        tcp_seq onxt = tp->snd_nxt;
                                          u_int win =
                                              min(tp->snd_wnd, tp->snd_cwnd) / 2 /
                                                  tp->t_maxseg;
                                          if (win < 2)     win = 2;
                                          tp->snd_ssthresh = win * tp->t_maxseg;
                                          tp->t_timer[TCPT_REXMT] = 0;
                                          tp->t_rtt = 0;
                                          tp->snd_nxt = ti->ti_ack;
                                          tp->snd_cwnd = tp->t_maxseg;
                                          (void) tcp_output(tp);
                                          tp->snd_cwnd = tp->snd_ssthresh +
                                                tp->t_maxseg * tp->t_dupacks;
                                          if (SEQ_GT(onxt, tp->snd_nxt))
                                                  tp->snd_nxt = onxt;
```

… **slow start or linear**

```
register u_int cw = tp->snd_cwnd;
register u_int incr = tp->t_maxseg;
if (cw > tp->snd_ssthresh)
                        incr = incr * incr / cw + incr / 8;
                tp->snd_cwnd = min(cw + incr, TCP_MAXWIN<<tp->snd_scale);
```


TCP NewReno partial ACK (FreeBSD 4.6)

```
if (SEQ_LT(th->th_ack, tp->snd_recover)) {
                tcp_seq onxt = tp->snd_nxt;
                u_long  ocwnd = tp->snd_cwnd;

                callout_stop(tp->tt_rexmt);
                tp->t_rtttime = 0;
                tp->snd_nxt = th->th_ack;
                /*
                 * Set snd_cwnd to one segment beyond acknowledged offset
                 * (tp->snd_una has not yet been updated when this function
                 *  is called)
                 */
                tp->snd_cwnd = tp->t_maxseg + (th->th_ack - tp->snd_una);
                (void) tcp_output(tp);
                tp->snd_cwnd = ocwnd;
                if (SEQ_GT(onxt, tp->snd_nxt))
                        tp->snd_nxt = onxt;
                /*
                 * Partial window deflation.  Relies on fact that tp->snd_una
                 * not updated yet.
                 */
                tp->snd_cwnd -= (th->th_ack - tp->snd_una - tp->t_maxseg);
                return (1);
```


RTT estimation

```
if (ts_present)
                        tcp_xmit_timer(tp, tcp_now-ts_ecr+1);
                else if (tp->t_rtt && SEQ_GT(ti->ti_ack, tp->t_rtseq))
                        tcp_xmit_timer(tp,tp->t_rtt);


…
 delta = rtt - 1 - (tp->t_srtt >> TCP_RTT_SHIFT);
                if ((tp->t_srtt += delta) <= 0)
                        tp->t_srtt = 1;
 if (delta < 0)
                        delta = -delta;
                delta -= (tp->t_rttvar >> TCP_RTTVAR_SHIFT);
                if ((tp->t_rttvar += delta) <= 0)
                        tp->t_rttvar = 1;
```


Linux TCP

- Linux derived from Minix
- Network buffers are cache and page aligned
- A lot of active development now in Linux network stack (messy)
- Source tree /usr/src/linux

```
arch/           drivers/  kernel/     Module.symvers  security/
COPYING         fs/       lib/        net/            sound/
CREDITS         include/  MAINTAINERS  README          System.map
crypto/         init/     Makefile    REPORTING-BUGS  usr/
Documentation/  ipc/      mm/         scripts/        vmlinux*

net/
802/        bridge/     ethernet/  key/      nonet.c  socket.c      unix/
8021q/      built-in.o  ipv4/      lapb/     packet/  socket.o      wanrouter/
appletalk/  compat.c    ipv6/      llc/      rose/    sunrpc/       x25/
atm/        core/       ipx/       Makefile  rxrpc/   sysctl_net.c  xfrm/
ax25/       decnet/     irda/      netlink/  sched/   sysctl_net.o
bluetooth/  econet/     Kconfig    netrom/   sctp/    TUNABLE

In ipv4/
tcp_bic.c   tcp_diag.c       tcp_hybla.c  tcp_minisocks.c  tcp_timer.c
tcp.c       tcp_highspeed.c  tcp_input.c  tcp_output.c     tcp_vegas.c
tcp_cong.c  tcp_htcp.c       tcp_ipv4.c   tcp_scalable.c   tcp_westwood.c
```


Linux socket struct (linux/tcp.h)

```
struct tcp_opt {
        __u32   rcv_nxt;        /* What we want to receive next         */
        __u32   snd_nxt;        /* Next sequence we send                */
        __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) */
/* Delayed ACK control data */
        struct {
                __u8    pending;        /* ACK is pending */
                __u8    quick;          /* Scheduled number of quick acks       */
                __u8    pingpong;       /* The session is interactive           */
                __u8    blocked;        /* Delayed ACK was blocked by socket lock*/
                __u32   ato;            /* Predicted tick of soft clock         */
                unsigned long timeout;  /* Currently scheduled timeout          */
                __u32   lrcvtime;       /* timestamp of last received data packet*/
                __u16   last_seg_size;  /* Size of last incoming segment        */
                __u16   rcv_mss;        /* MSS used for delayed ACK decisions   */
   }
```

```
        __u8    ca_state;       /* State of fast-retransmit machine     */
        __u8    retransmits;    /* Number of unrecovered RTO timeouts.  */

        __u8    reordering;     /* Packet reordering metric.            */
        __u8    frto_counter;   /* Number of new acks after RTO */
        __u32   frto_highmark;  /* snd_nxt when RTO occurred */

        __u8    adv_cong;       /* Using Vegas, Westwood, or BIC */
/* RTT measurement */
        __u8    backoff;        /* backoff                              */
        __u32   srtt;           /* smoothed round trip time << 3        */
        __u32   mdev;           /* medium deviation                     */
        __u32   mdev_max;       /* maximal mdev for the last rtt period */
        __u32   rttvar;         /* smoothed mdev_max                    */
        __u32   rtt_seq;        /* sequence number to update rttvar     */
        __u32   rto;            /* retransmit timeout                   */
```



```
/*
 *      Slow start and congestion control (see also Nagle, and Karn & Partridge)
 */
        __u32   snd_ssthresh;   /* Slow start size threshold            */
        __u32   snd_cwnd;       /* Sending congestion window            */
        __u16   snd_cwnd_cnt;   /* Linear increase counter              */
        __u16   snd_cwnd_clamp; /* Do not allow snd_cwnd to grow above this */
        __u32   snd_cwnd_used;
        __u32   snd_cwnd_stamp;

        /* Two commonly used timers in both sender and receiver paths. */
        unsigned long           timeout;
        struct timer_list       retransmit_timer;       /* Resend (no ack)      */
        struct timer_list       delack_timer;           /* Ack delay            */

        struct sk_buff_head     out_of_order_queue; /* Out of order segments go
    here */
/*      SACKs data      */
        __u16   user_mss;       /* mss requested by user in ioctl */
        __u8    dsack;          /* D-SACK is scheduled                  */
        __u8    eff_sacks;      /* Size of SACK array to send with next packet */
        struct tcp_sack_block duplicate_sack[1]; /* D-SACK block */
        struct tcp_sack_block selective_acks[4]; /* The SACKS themselves*/
```


Linux code snippets

- Adding bytes to out-of-order queue

```
if (skb_queue_len(&tp->out_of_order_queue)) {
                tcp_ofo_queue(sk);

                /* RFC2581. 4.2. SHOULD send immediate ACK, when
                 * gap in queue is filled.
                 */
                if (!skb_queue_len(&tp->out_of_order_queue))
                        tp->ack.pingpong = 0;
        }
```

- Congestion avoidance, adjusting cwnd

```
if (tcp_westwood_cwnd(tp))
                tp->snd_ssthresh = tp->snd_cwnd;
        else
                tp->snd_cwnd = min(tp->snd_cwnd, tp->snd_ssthresh);
        tp->snd_cwnd_stamp = tcp_time_stamp;
```


Linux TCP optimizations

- Cache (in router cache) reorder_threshold and ssthresh for path
- Adaptive reorder_threshold (dup thresh)
  - Initially 3
  - If packet arrives “early” after retransmit, assume out of order
  - Cancel (un do) congestion avoidance
  - Increment reorder_threshold
  - D-SACK info also used to update reorder_threshold
- Max burst limit for back-to-back sends
- Receiver ACK’s every packet on initial slow-start
- Send and receive buffer auto-tuning
- Linux 2.6.13 has pluggable congestion avoidance modules
  - HS TCP, STCP, TCP-hybla, H-TCP, Westwood, Vegas
  - BI-TCP default
  - Newreno/SACK/FACK/ECN

Tuning the TCP stack

- In the “old” days, use a kernel debugger to set variables in kernel memory or on disk image of kernel
- If values were hard-wired into kernel, then edit config files or sources and rebuild kernel (Appendix B)
- Today most interesting kernel variables can be viewed/modified with sysctl (and /proc in linux) or Windows registry

```
net.ipv4.tcp_low_latency = 0
net.ipv4.tcp_frto = 0
net.ipv4.tcp_tw_reuse = 0
net.ipv4.tcp_adv_win_scale = 2
net.ipv4.tcp_app_win = 31
net.ipv4.tcp_rmem = 4096        87380   174760
net.ipv4.tcp_wmem = 4096        16384   131072
net.ipv4.tcp_mem = 97280        97792   98304
net.ipv4.tcp_dsack = 1
net.ipv4.tcp_ecn = 0
net.ipv4.tcp_reordering = 3
net.ipv4.tcp_fack = 1
net.ipv4.tcp_orphan_retries = 0
net.ipv4.tcp_max_syn_backlog = 1024
net.ipv4.tcp_rfc1337 = 0
net.ipv4.tcp_stdurg = 0
net.ipv4.tcp_abort_on_overflow = 0
net.ipv4.tcp_tw_recycle = 0
net.ipv4.tcp_syncookies = 0
net.ipv4.tcp_fin_timeout = 60
net.ipv4.tcp_retries2 = 15
```


OS tuning incantations

- FreeBSD max buffer size sysctl -w kern.maxsockbuf=524288
- Linux
  - **echo 1 > /proc/sys/net/ipv4/tcp_timestamps**
  - **echo 1 > /proc/sys/net/ipv4/tcp_window_scaling**
  - **echo 1 > /proc/sys/net/ipv4/tcp_sack**
  - **echo 8388608 > /proc/sys/net/core/wmem_max**
  - **echo 8388608 > /proc/sys/net/core/rmem_max**
  - **echo "4096 87380 4194304" > /proc/sys/net/ipv4/tcp_rmem**
  - **echo "4096 65536 4194304" > /proc/sys/net/ipv4/tcp_wmem**
- Windows XP registry
  HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters
  GlobalMaxTcpWindowSize="256960"
  Tcp1323Opts="1"
- See PSC tuning table

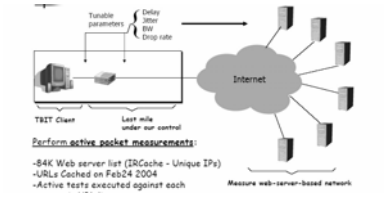
Known implementation problems (RFC 2525)

- No initial slow start
- No slow start after time out
- Failed to initialized cwnd
- Failure to retain out of order data
- Extra additive constant in AIMD
- Initial RTO too low
- No window deflation exiting recovery
- Short connection keepalive
- No exponential backoff on timeout
- Window probe deadlock
- Stretch ACKs
- Retransmit multiple packets
- FIN/RST logic broken

Hacker tools (nmap, quesos,…) remotely identify an OS by sending variously formed IP/TCP packets. Each OS responds a bit differently.

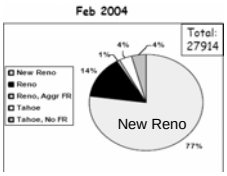
TBIT, TCP behavior inference tool

- How to determine the flavor of TCP a remote host is running?
- Passive info (tcpdump): observe SYN/SYN-ACK and window advertisements
  window scale, timestamps, SACK, ECN, MTU discovery
- For TCP, all the "action" is on the sender side – make remote send data
- Request web pages and induce packet loss and observe recovery
  - Emulator test bed and/or TBIT tool



TCP survey

TBIT results



ECN < 1%
SACK ~ 80%
timestamp ~ 13%
window 64k ~ 73%
window scale ~ 15%

Next time ...

- TCP instrumentation
- TCP overhead