24

TCP: Transmission Control Protocol

24.1 Introduction

The Transmission Control Protocol, or TCP, provides a connection-oriented, reliable, byte-stream service between the two end points of an application. This is completely different from UDP's connectionless, unreliable, datagram service.

The implementation of UDP presented in Chapter 23 comprised 9 functions and about 800 lines of C code. The TCP implementation we're about to describe comprises 28 functions and almost 4,500 lines of C code. Therefore we divide the presentation of TCP into multiple chapters.

These chapters are not an introduction to TCP. We assume the reader is familiar with the operation of TCP from Chapters 17–24 of Volume 1.

24.2 Code Introduction

The TCP functions appear in six C files and numerous TCP definitions are in seven headers, as shown in Figure 24.1.

Figure 24.2 shows the relationship of the various TCP functions to other kernel functions. The shaded ellipses are the nine main TCP functions that we cover. Eight of these functions appear in the TCP `protosw` structure (Figure 24.8) and the ninth is `tcp_output`.

File	Description
netinet/tcp.h	tcphdr structure definition
netinet/tcp_debug.h	tcp_debug structure definition
netinet/tcp_fsm.h	definitions for TCP's finite state machine
netinet/tcp_seq.h	macros for comparing TCP sequence numbers
netinet/tcp_timer.h	definitions for TCP timers
netinet/tcp_var.h	tcpcb (control block) and tcpstat (statistics) structure definitions
netinet/tcpip.h	TCP plus IP header definition
netinet/tcp_debug.c	support for SO_DEBUG socket debugging (Section 27.10)
netinet/tcp_input.c	tcp_input and ancillary functions (Chapters 28 and 29)
netinet/tcp_output.c	tcp_output and ancillary functions (Chapter 26)
netinet/tcp_subr.c	miscellaneous TCP subroutines (Chapter 27)
netinet/tcp_timer.c	TCP timer handling (Chapter 25)
netinet/tcp_usrreq.c	PRU_*xxx* request handling (Chapter 30)

Figure 24.1 Files discussed in the TCP chapters.

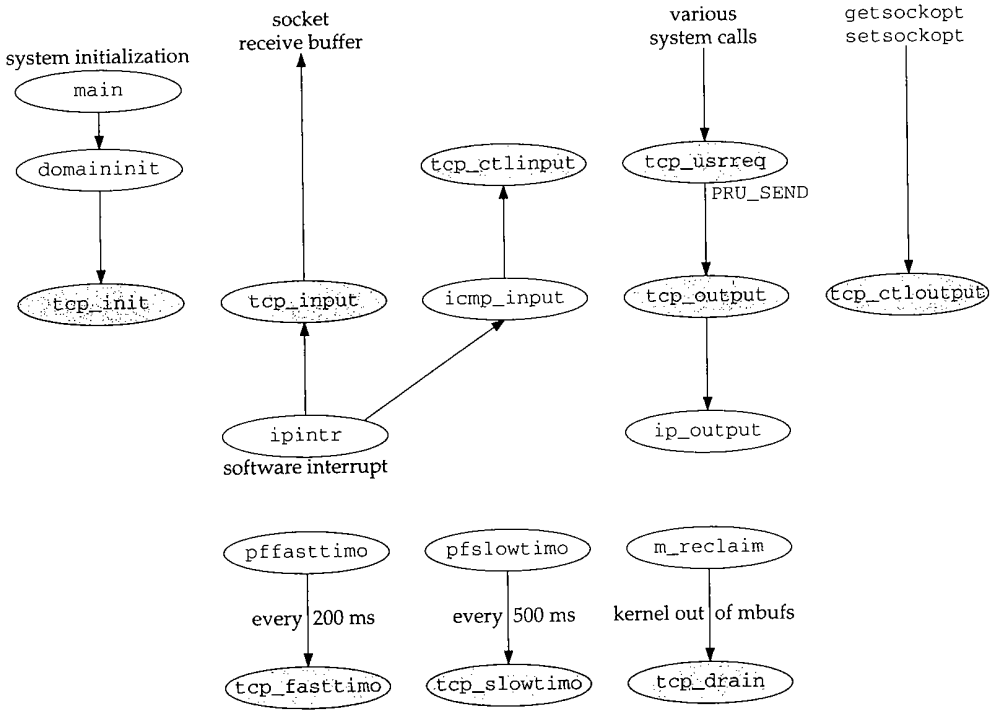


Figure 24.2 Relationship of TCP functions to rest of the kernel.

Global Variables

Figure 24.3 shows the global variables we encounter throughout the TCP functions.

Variable	Datatype	Description
`tcb`	`struct inpcb`	head of the TCP Internet PCB list
`tcp_last_inpcb`	`struct inpcb *`	pointer to PCB for last received segment: one-behind cache
`tcpstat`	`struct tcpstat`	TCP statistics (Figure 24.4)
`tcp_outflags`	`u_char`	array of output flags, indexed by connection state (Figure 24.16)
`tcp_recvspace`	`u_long`	default size of socket receive buffer (8192 bytes)
`tcp_sendspace`	`u_long`	default size of socket send buffer (8192 bytes)
`tcp_iss`	`tcp_seq`	initial send sequence number (ISS)
`tcprexmtthresh`	`int`	number of duplicate ACKs to trigger fast retransmit (3)
`tcp_mssdflt`	`int`	default MSS (512 bytes)
`tcp_rttdflt`	`int`	default RTT if no data (3 seconds)
`tcp_do_rfc1323`	`int`	if true (default), request window scale and timestamp options
`tcp_now`	`u_long`	500 ms counter for RFC 1323 timestamps
`tcp_keepidle`	`int`	keepalive: idle time before first probe (2 hours)
`tcp_keepintvl`	`int`	keepalive: interval between probes when no response (75 sec) (also used as timeout for `connect`)
`tcp_maxidle`	`int`	keepalive: time after probing before giving up (10 min)

Figure 24.3 Global variables introduced in the following chapters.

Statistics

Various TCP statistics are maintained in the global structure `tcpstat`, described in Figure 24.4. We'll see where these counters are incremented as we proceed through the code.

Figure 24.5 shows some sample output of these statistics, from the `netstat -s` command. These statistics were collected after the host had been up for 30 days. Since some counters come in pairs—one counts the number of packets and the other the number of bytes—we abbreviate these in the figure. For example, the two counters for the second line of the table are `tcps_sndpack` and `tcps_sndbyte`.

> The counter for `tcps_sndbyte` should be 3,722,884,824, not –22,194,928 bytes. This is an average of about 405 bytes per segment, which makes sense. Similarly, the counter for `tcps_rcvackbyte` should be 3,738,811,552, not –21,264,360 bytes (for an average of about 565 bytes per segment). These numbers are incorrectly printed as negative numbers because the `printf` calls in the `netstat` program use `%d` (signed decimal) instead of `%lu` (long integer, unsigned decimal). All the counters are unsigned long integers, and these two counters are near the maximum value of an unsigned 32-bit long integer ($2^{32} - 1 = 4,294,967,295$).

tcpstat member	Description	Used by SNMP
tcps_accepts	#SYNs received in LISTEN state	•
tcps_closed	#connections closed (includes drops)	
tcps_connattempt	#connections initiated (calls to connect)	•
tcps_conndrops	#embryonic connections dropped (before SYN received)	•
tcps_connects	#connections established actively or passively	
tcps_delack	#delayed ACKs sent	
tcps_drops	#connections dropped (after SYN received)	•
tcps_keepdrops	#connections dropped in keepalive (established or awaiting SYN)	
tcps_keepprobe	#keepalive probes sent	
tcps_keeptimeo	#times keepalive timer or connection-establishment timer expire	
tcps_pawsdrop	#segments dropped due to PAWS	
tcps_pcbcachemiss	#times PCB cache comparison fails	
tcps_persisttimeo	#times persist timer expires	
tcps_predack	#times header prediction correct for ACKs	
tcps_preddat	#times header prediction correct for data packets	
tcps_rcvackbyte	#bytes ACKed by received ACKs	
tcps_rcvackpack	#received ACK packets	
tcps_rcvacktoomuch	#received ACKs for unsent data	
tcps_rcvafterclose	#packets received after connection closed	
tcps_rcvbadoff	#packets received with invalid header length	•
tcps_rcvbadsum	#packets received with checksum errors	•
tcps_rcvbyte	#bytes received in sequence	
tcps_rcvbyteafterwin	#bytes received beyond advertised window	
tcps_rcvdupack	#duplicate ACKs received	
tcps_rcvdupbyte	#bytes received in completely duplicate packets	
tcps_rcvduppack	#packets received with completely duplicate bytes	
tcps_rcvoobyte	#out-of-order bytes received	
tcps_rcvoopack	#out-of-order packets received	
tcps_rcvpack	#packets received in sequence	
tcps_rcvpackafterwin	#packets with some data beyond advertised window	
tcps_rcvpartdupbyte	#duplicate bytes in part-duplicate packets	
tcps_rcvpartduppack	#packets with some duplicate data	
tcps_rcvshort	#packets received too short	•
tcps_rcvtotal	total #packets received	•
tcps_rcvwinprobe	#window probe packets received	
tcps_rcvwinupd	#received window update packets	
tcps_rexmttimeo	#retransmit timeouts	
tcps_rttupdated	#times RTT estimators updated	
tcps_segstimed	#segments for which TCP tried to measure RTT	
tcps_sndacks	#ACK-only packets sent (data length = 0)	
tcps_sndbyte	#data bytes sent	
tcps_sndctrl	#control (SYN, FIN, RST) packets sent (data length = 0)	
tcps_sndpack	#data packets sent (data length > 0)	
tcps_sndprobe	#window probes sent (1 byte of data forced by persist timer)	
tcps_sndrexmitbyte	#data bytes retransmitted	•
tcps_sndrexmitpack	#data packets retransmitted	•
tcps_sndtotal	total #packets sent	•
tcps_sndurg	#packets sent with URG-only (data length = 0)	
tcps_sndwinup	#window update-only packets sent (data length = 0)	
tcps_timeoutdrop	#connections dropped in retransmission timeout	

Figure 24.4 TCP statistics maintained in the tcpstat structure.

netstat -s output	tcpstat members
10,655,999 packets sent	tcps_sndtotal
9,177,823 data packets (-22,194,928 bytes)	tcps_snd{pack,byte}
257,295 data packets (81,075,086 bytes) retransmitted	tcps_sndrexmit{pack,byte}
862,900 ack-only packets (531,285 delayed)	tcps_sndacks,tcps_delack
229 URG-only packets	tcps_sndurg
3,453 window probe packets	tcps_sndprobe
74,925 window update packets	tcps_sndwinup
279,387 control packets	tcps_sndctrl
8,801,953 packets received	tcps_rcvtotal
6,617,079 acks (for -21,264,360 bytes)	tcps_rcvack{pack,byte}
235,311 duplicate acks	tcps_rcvdupack
0 acks for unsent data	tcps_rcvacktoomuch
4,670,615 packets (324,965,351 bytes) rcvd in-sequence	tcps_rcv{pack,byte}
46,953 completely duplicate packets (1,549,785 bytes)	tcps_rcvdup{pack,byte}
22 old duplicate packets	tcps_pawsdrop
3,442 packets with some dup. data (54,483 bytes duped)	tcps_rcvpartdup{pack,byte}
77,114 out-of-order packets (13,938,456 bytes)	tcps_rcvoo{pack,byte}
1,892 packets (1,755 bytes) of data after window	tcps_rcv{pack,byte}afterwin
1,755 window probes	tcps_rcvwinprobe
175,476 window update packets	tcps_rcvwindup
1,017 packets received after close	tcps_rcvafterclose
60,370 discarded for bad checksums	tcps_rcvbadsum
279 discarded for bad header offset fields	tcps_rcvbadoff
0 discarded because packet too short	tcps_rcvshort
144,020 connection requests	tcps_connattempt
92,595 connection accepts	tcps_accepts
126,820 connections established (including accepts)	tcps_connects
237,743 connections closed (including 1,061 drops)	tcps_closed,tcps_drops
110,016 embryonic connections dropped	tcps_conndrops
6,363,546 segments updated rtt (of 6,444,667 attempts)	tcps_{rttupdated,segstimed}
114,797 retransmit timeouts	tcps_rexmttimeo
86 connection dropped by rexmit timeout	tcps_timeoutdrop
1,173 persist timeouts	tcps_persisttimeo
16,419 keepalive timeouts	tcps_keeptimeo
6,899 keepalive probes sent	tcps_keepprobe
3,219 connections dropped by keepalive	tcps_keepdrops
733,130 correct ACK header predictions	tcps_predack
1,266,889 correct data packet header predictions	tcps_preddat
1,851,557 cache misses	tcps_pcbcachemiss

Figure 24.5 Sample TCP statistics.

SNMP Variables

Figure 24.6 shows the 14 simple SNMP variables in the TCP group and the counters from the `tcpstat` structure implementing that variable. The constant values shown for the first four entries are fixed by the Net/3 implementation. The counter `tcpCurrEstab` is computed as the number of Internet PCBs on the TCP PCB list.

Figure 24.7 shows `tcpTable`, the TCP listener table.

SNMP variable	tcpstat members or constant	Description
tcpRtoAlgorithm	4	algorithm used to calculate retransmission timeout value: 1 = none of the following, 2 = a constant RTO, 3 = MIL–STD–1778 Appendix B, 4 = Van Jacobson's algorithm.
tcpRtoMin	1000	minimum retransmission timeout value, in milliseconds
tcpRtoMax	64000	maximum retransmission timeout value, in milliseconds
tcpMaxConn	-1	maximum #TCP connections (–1 if dynamic)
tcpActiveOpens	tcps_connattempt	#transitions from CLOSED to SYN_SENT states
tcpPassiveOpens	tcps_accepts	#transitions from LISTEN to SYN_RCVD states
tcpAttemptFails	tcps_conndrops	#transitions from SYN_SENT or SYN_RCVD to CLOSED, plus #transitions from SYN_RCVD to LISTEN
tcpEstabResets	tcps_drops	#transitions from ESTABLISHED or CLOSE_WAIT states to CLOSED
tcpCurrEstab	(see text)	#connections currently in ESTABLISHED or CLOSE_WAIT states
tcpInSegs	tcps_rcvtotal	total #segments received
tcpOutSegs	tcps_sndtotal - tcps_sndrexmitpack	total #segments sent, excluding those containing only retransmitted bytes
tcpRetransSegs	tcps_sndrexmitpack	total #retransmitted segments
tcpInErrs	tcps_rcvbadsum + tcps_rcvbadoff + tcps_rcvshort	total #segments received with an error
tcpOutRsts	(not implemented)	total #segments sent with RST flag set

Figure 24.6 Simple SNMP variables in tcp group.

index = <*tcpConnLocalAddress*>.<*tcpConnLocalPort*>.<*tcpConnRemAddress*>.<*tcpConnRemPort*>		
SNMP variable	PCB variable	Description
tcpConnState	t_state	state of connection: 1 = CLOSED, 2 = LISTEN, 3 = SYN_SENT, 4 = SYN_RCVD, 5 = ESTABLISHED, 6 = FIN_WAIT_1, 7 = FIN_WAIT_2, 8 = CLOSE_WAIT, 9 = LAST_ACK, 10 = CLOSING, 11 = TIME_WAIT, 12 = delete TCP control block.
tcpConnLocalAddress	inp_laddr	local IP address
tcpConnLocalPort	inp_lport	local port number
tcpConnRemAddress	inp_faddr	foreign IP address
tcpConnRemPort	inp_fport	foreign port number

Figure 24.7 Variables in TCP listener table: tcpTable.

The first PCB variable (t_state) is from the TCP control block (Figure 24.13) and the remaining four are from the Internet PCB (Figure 22.4).

24.3 TCP protosw Structure

Figure 24.8 lists the TCP protosw structure, the protocol switch entry for TCP.

Member	inetsw[2]	Description
pr_type	*SOCK_STREAM*	TCP provides a byte-stream service
pr_domain	*&inetdomain*	TCP is part of the Internet domain
pr_protocol	*IPPROTO_TCP (6)*	appears in the ip_p field of the IP header
pr_flags	*PR_CONNREQUIRED|PR_WANTRCVD*	socket layer flags, not used by protocol processing
pr_input	*tcp_input*	receives messages from IP layer
pr_output	*0*	not used by TCP
pr_ctlinput	*tcp_ctlinput*	control input function for ICMP errors
pr_ctloutput	*tcp_ctloutput*	respond to administrative requests from a process
pr_usrreq	*tcp_usrreq*	respond to communication requests from a process
pr_init	*tcp_init*	initialization for TCP
pr_fasttimo	*tcp_fasttimo*	fast timeout function, called every 200 ms
pr_slowtimo	*tcp_slowtimo*	slow timeout function, called every 500 ms
pr_drain	*tcp_drain*	called when kernel runs out of mbufs
pr_sysctl	*0*	not used by TCP

Figure 24.8 The TCP protosw structure.

24.4 TCP Header

The TCP header is defined as a tcphdr structure. Figure 24.9 shows the C structure and Figure 24.10 shows a picture of the TCP header.

tcp.h

```
40 struct tcphdr {
41     u_short th_sport;          /* source port */
42     u_short th_dport;          /* destination port */
43     tcp_seq th_seq;            /* sequence number */
44     tcp_seq th_ack;            /* acknowledgement number */
45 #if BYTE_ORDER == LITTLE_ENDIAN
46     u_char  th_x2:4,           /* (unused) */
47             th_off:4;          /* data offset */
48 #endif
49 #if BYTE_ORDER == BIG_ENDIAN
50     u_char  th_off:4,          /* data offset */
51             th_x2:4;           /* (unused) */
52 #endif
53     u_char  th_flags;          /* ACK, FIN, PUSH, RST, SYN, URG */
54     u_short th_win;            /* advertised window */
55     u_short th_sum;            /* checksum */
56     u_short th_urp;            /* urgent offset */
57 };
```

tcp.h

Figure 24.9 tcphdr structure.

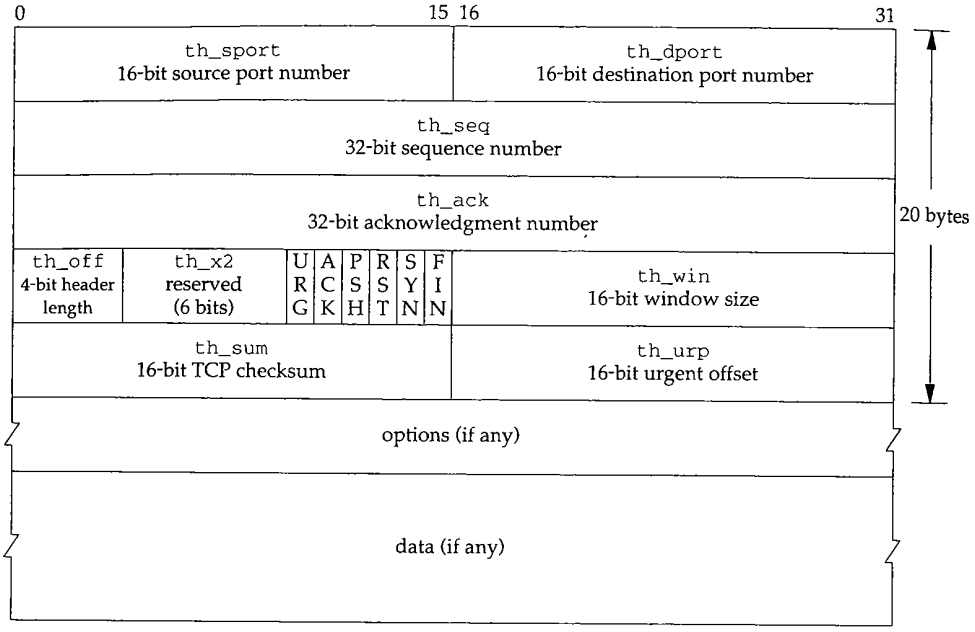


Figure 24.10 TCP header and optional data.

Most RFCs, most books (including Volume 1), and the code we'll examine call `th_urp` the *urgent pointer*. A better term is the *urgent offset*, since this field is a 16-bit unsigned offset that must be added to the sequence number field (`th_seq`) to give the 32-bit sequence number of the *last* byte of urgent data. (There is a continuing debate over whether this sequence number points to the last byte of urgent data or to the byte that follows. This is immaterial for the present discussion.) We'll see in Figure 24.13 that TCP correctly calls the 32-bit sequence number of the last byte of urgent data `snd_up` the *send urgent pointer*. But using the term *pointer* for the 16-bit offset in the TCP header is misleading. In Exercise 26.6 we'll reiterate the distinction between the urgent pointer and the urgent offset.

The 4-bit header length, the 6 reserved bits that follow, and the 6 flag bits are defined in C as two 4-bit bit-fields, followed by 8 bits of flags. To handle the difference in the order of these 4-bit fields within an 8-bit byte, the code contains an `#ifdef` based on the byte order of the system.

Also notice that we call the 4-bit `th_off` the *header length*, while the C code calls it the *data offset*. Both are correct since it is the length of the TCP header, including options, in 32-bit words, which is the offset of the first byte of data.

The `th_flags` member contains 6 flag bits, accessed using the names in Figure 24.11.

In Net/3 the TCP header is normally referenced as an IP header immediately followed by a TCP header. This is how `tcp_input` processes received IP datagrams and how `tcp_output` builds outgoing IP datagrams. This combined IP/TCP header is a `tcpiphdr` structure, shown in Figure 24.12.

th_flags	Description
TH_ACK	the acknowledgment number (th_ack) is valid
TH_FIN	the sender is finished sending data
TH_PUSH	receiver should pass the data to application without delay
TH_RST	reset the connection
TH_SYN	synchronize sequence numbers (establish connection)
TH_URG	the urgent offset (th_urp) is valid

Figure 24.11 th_flags values.

tcpip.h

```
38 struct tcpiphdr {
39     struct ipovly ti_i;          /* overlaid ip structure */
40     struct tcphdr ti_t;          /* tcp header */
41 };

42 #define ti_next     ti_i.ih_next
43 #define ti_prev     ti_i.ih_prev
44 #define ti_x1       ti_i.ih_x1
45 #define ti_pr       ti_i.ih_pr
46 #define ti_len      ti_i.ih_len
47 #define ti_src      ti_i.ih_src
48 #define ti_dst      ti_i.ih_dst
49 #define ti_sport    ti_t.th_sport
50 #define ti_dport    ti_t.th_dport
51 #define ti_seq      ti_t.th_seq
52 #define ti_ack      ti_t.th_ack
53 #define ti_x2       ti_t.th_x2
54 #define ti_off      ti_t.th_off
55 #define ti_flags    ti_t.th_flags
56 #define ti_win      ti_t.th_win
57 #define ti_sum      ti_t.th_sum
58 #define ti_urp      ti_t.th_urp
```

tcpip.h

Figure 24.12 tcpiphdr structure: combined IP/TCP header.

38–58 The 20-byte IP header is defined as an ipovly structure, which we showed earlier in Figure 23.12. As we discussed with Figure 23.19, this structure is not a real IP header, although the lengths are the same (20 bytes).

24.5 TCP Control Block

In Figure 22.1 we showed that TCP maintains its own control block, a tcpcb structure, in addition to the standard Internet PCB. In contrast, UDP has everything it needs in the Internet PCB—it doesn't need its own control block.

The TCP control block is a large structure, occupying 140 bytes. As shown in Figure 22.1 there is a one-to-one relationship between the Internet PCB and the TCP control block, and each points to the other. Figure 24.13 shows the definition of the TCP control block.

tcp_var.h

```
41 struct tcpcb {
42     struct tcpiphdr *seg_next;  /* reassembly queue of received segments */
43     struct tcpiphdr *seg_prev;  /* reassembly queue of received segments */
44     short   t_state;            /* connection state (Figure 24.16) */
45     short   t_timer[TCPT_NTIMERS];  /* tcp timers (Chapter 25) */
46     short   t_rxtshift;         /* log(2) of rexmt exp. backoff */
47     short   t_rxtcur;           /* current retransmission timeout (#ticks) */
48     short   t_dupacks;          /* #consecutive duplicate ACKs received */
49     u_short t_maxseg;           /* maximum segment size to send */
50     char    t_force;            /* 1 if forcing out a byte (persist/OOB) */
51     u_short t_flags;            /* (Figure 24.14) */
52     struct tcpiphdr *t_template;    /* skeletal packet for transmit */
53     struct inpcb *t_inpcb;      /* back pointer to internet PCB */
54 /*
55  * The following fields are used as in the protocol specification.
56  * See RFC783, Dec. 1981, page 21.
57  */
58 /* send sequence variables */
59     tcp_seq snd_una;            /* send unacknowledged */
60     tcp_seq snd_nxt;            /* send next */
61     tcp_seq snd_up;             /* send urgent pointer */
62     tcp_seq snd_wl1;            /* window update seg seq number */
63     tcp_seq snd_wl2;            /* window update seg ack number */
64     tcp_seq iss;                /* initial send sequence number */
65     u_long  snd_wnd;            /* send window */
66 /* receive sequence variables */
67     u_long  rcv_wnd;            /* receive window */
68     tcp_seq rcv_nxt;            /* receive next */
69     tcp_seq rcv_up;             /* receive urgent pointer */
70     tcp_seq irs;                /* initial receive sequence number */
71 /*
72  * Additional variables for this implementation.
73  */
74 /* receive variables */
75     tcp_seq rcv_adv;            /* advertised window by other end */
76 /* retransmit variables */
77     tcp_seq snd_max;            /* highest sequence number sent;
78                                  * used to recognize retransmits */
79 /* congestion control (slow start, source quench, retransmit after loss) */
80     u_long  snd_cwnd;           /* congestion-controlled window */
81     u_long  snd_ssthresh;       /* snd_cwnd size threshhold for slow start
82                                  * exponential to linear switch */
83 /*
84  * transmit timing stuff.  See below for scale of srtt and rttvar.
85  * "Variance" is actually smoothed difference.
86  */
87     short   t_idle;             /* inactivity time */
88     short   t_rtt;              /* round-trip time */
89     tcp_seq t_rtseq;            /* sequence number being timed */
90     short   t_srtt;             /* smoothed round-trip time */
91     short   t_rttvar;           /* variance in round-trip time */
92     u_short t_rttmin;           /* minimum rtt allowed */
93     u_long  max_sndwnd;         /* largest window peer has offered */
```

24.6

```
 94 /* out-of-band data */
 95     char    t_oobflags;         /* TCPOOB_HAVEDATA, TCPOOB_HADDATA */
 96     char    t_iobc;             /* input character, if not SO_OOBINLINE */
 97     short   t_softerror;        /* possible error not yet reported */
 98 /* RFC 1323 variables */
 99     u_char  snd_scale;          /* scaling for send window (0-14) */
100     u_char  rcv_scale;          /* scaling for receive window (0-14) */
101     u_char  request_r_scale;    /* our pending window scale */
102     u_char  requested_s_scale;  /* peer's pending window scale */
103     u_long  ts_recent;          /* timestamp echo data */
104     u_long  ts_recent_age;      /* when last updated */
105     tcp_seq last_ack_sent;      /* sequence number of last ack field */
106 };
107 #define intotcpcb(ip)   ((struct tcpcb *)(ip)->inp_ppcb)
108 #define sototcpcb(so)   (intotcpcb(sotoinpcb(so)))
```

tcp_var.h

Figure 24.13 `tcpcb` structure: TCP control block.

We'll save the discussion of these variables until we encounter them in the code.

Figure 24.14 shows the values for the `t_flags` member.

t_flags	Description
TF_ACKNOW	send ACK immediately
TF_DELACK	send ACK, but try to delay it
TF_NODELAY	don't delay packets to coalesce (disable Nagle algorithm)
TF_NOOPT	don't use TCP options (never set)
TF_SENTFIN	have sent FIN
TF_RCVD_SCALE	set when other side sends window scale option in SYN
TF_RCVD_TSTMP	set when other side sends timestamp option in SYN
TF_REQ_SCALE	have/will request window scale option in SYN
TF_REQ_TSTMP	have/will request timestamp option in SYN

Figure 24.14 `t_flags` values.

24.6 TCP State Transition Diagram

Many of TCP's actions, in response to different types of segments arriving on a connection, can be summarized in a state transition diagram, shown in Figure 24.15. We also duplicate this diagram on one of the front end papers, for easy reference while reading the TCP chapters.

These state transitions define the TCP finite state machine. Although the transition from LISTEN to SYN_SENT is allowed by TCP, there is no way to do this using the sockets API (i.e., a `connect` is not allowed after a `listen`).

The `t_state` member of the control block holds the current state of a connection, with the values shown in Figure 24.16.

This figure also shows the `tcp_outflags` array, which contains the outgoing flags for `tcp_output` to use when the connection is in that state.

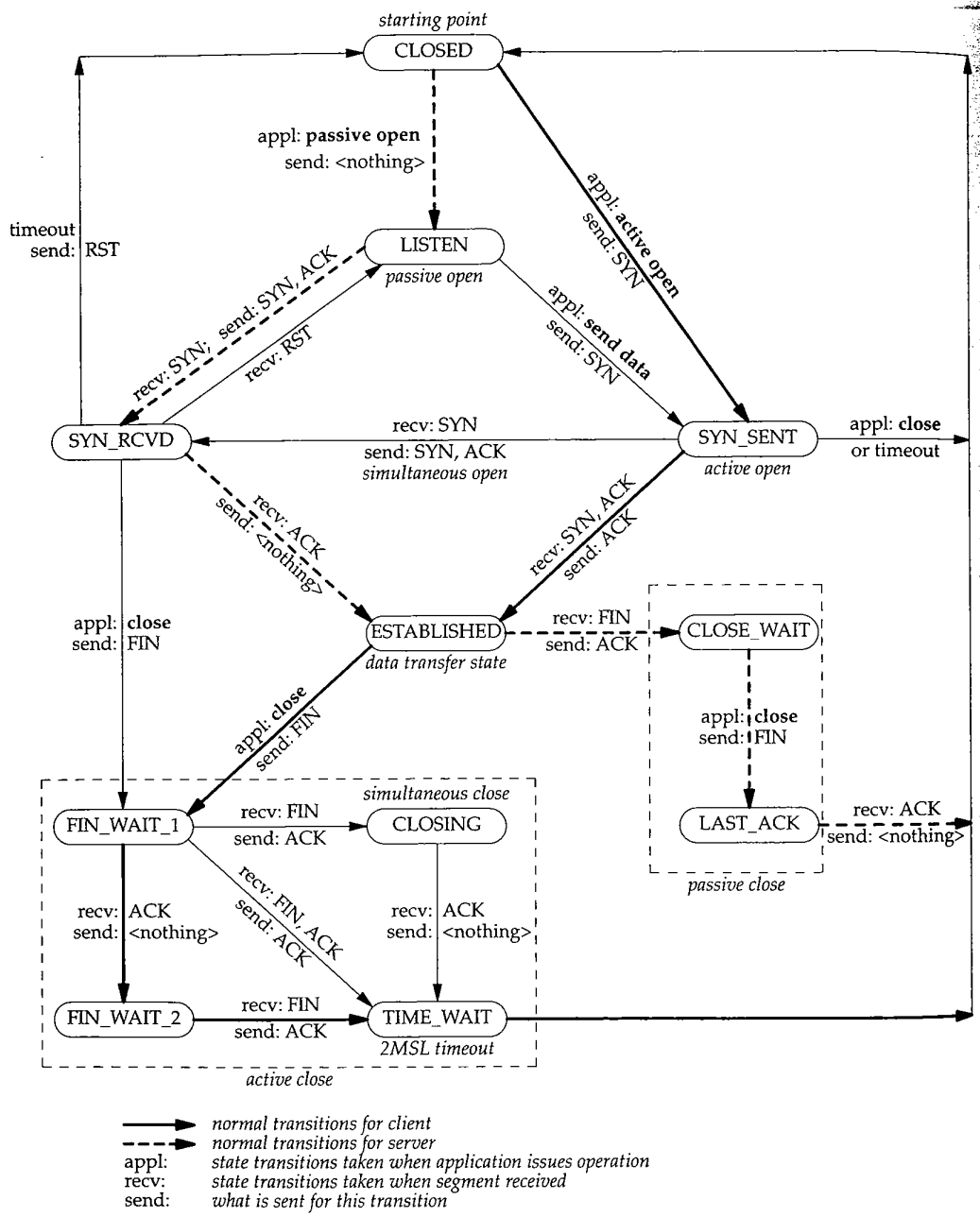


Figure 24.15 TCP state transition diagram.

Half-C

24.7

t_state	value	Description	tcp_outflags[]
TCPS_CLOSED	0	closed	TH_RST | TH_ACK
TCPS_LISTEN	1	listening for connection (passive open)	0
TCPS_SYN_SENT	2	have sent SYN (active open)	TH_SYN
TCPS_SYN_RECEIVED	3	have sent and received SYN; awaiting ACK	TH_SYN | TH_ACK
TCPS_ESTABLISHED	4	established (data transfer)	TH_ACK
TCPS_CLOSE_WAIT	5	received FIN, waiting for application close	TH_ACK
TCPS_FIN_WAIT_1	6	have closed, sent FIN; awaiting ACK and FIN	TH_FIN | TH_ACK
TCPS_CLOSING	7	simultaneous close; awaiting ACK	TH_FIN | TH_ACK
TCPS_LAST_ACK	8	received FIN have closed; awaiting ACK	TH_FIN | TH_ACK
TCPS_FIN_WAIT_2	9	have closed; awaiting FIN	TH_ACK
TCPS_TIME_WAIT	10	2MSL wait state after active close	TH_ACK

Figure 24.16 t_state values.

Figure 24.16 also shows the numerical values of these constants since the code uses their numerical relationships. For example, the following two macros are defined:

```
#define  TCPS_HAVERCVDSYN(s)   ((s) >= TCPS_SYN_RECEIVED)
#define  TCPS_HAVERCVDFIN(s)   ((s) >= TCPS_TIME_WAIT)
```

Similarly, we'll see that tcp_notify handles ICMP errors differently when the connection is not yet established, that is, when t_state is less than TCPS_ESTABLISHED.

> The name TCPS_HAVERCVDSYN is correct, but the name TCPS_HAVERCVDFIN is misleading. A FIN has also been received in the CLOSE_WAIT, CLOSING, and LAST_ACK states. We encounter this macro in Chapter 29.

Half-Close

When a process calls shutdown with a second argument of 1, it is called a *half-close*. TCP sends a FIN but allows the process to continue receiving on the socket. (Section 18.5 of Volume 1 contains examples of TCP's half-close.)

For example, even though we label the ESTABLISHED state "data transfer," if the process does a half-close, moving the connection to the FIN_WAIT_1 and then the FIN_WAIT_2 states, data can continue to be received by the process in these two states.

24.7 TCP Sequence Numbers

Every byte of data exchanged across a TCP connection, along with the SYN and FIN flags, is assigned a 32-bit *sequence number*. The sequence number field in the TCP header (Figure 24.10) contains the sequence number of the first byte of data in the segment. The *acknowledgment number* field in the TCP header contains the next sequence number that the sender of the ACK expects to receive, which acknowledges all data bytes through the acknowledgment number minus 1. In other words, the acknowledgment number is the *next* sequence number expected by the sender of the ACK. The acknowledgment number is valid only if the ACK flag is set in the header. We'll see

that TCP always sets the ACK flag except for the first SYN sent by an active open (the SYN_SENT state; see `tcp_outflags[2]` in Figure 24.16) and in some RST segments.

Since a TCP connection is *full-duplex*, each end must maintain a set of sequence numbers for both directions of data flow. In the TCP control block (Figure 24.13) there are 13 sequence numbers: eight for the send direction (the *send sequence space*) and five for the receive direction (the *receive sequence space*).

Figure 24.17 shows the relationship of four of the variables in the send sequence space: `snd_wnd`, `snd_una`, `snd_nxt`, and `snd_max`. In this example we number the bytes 1 through 11.

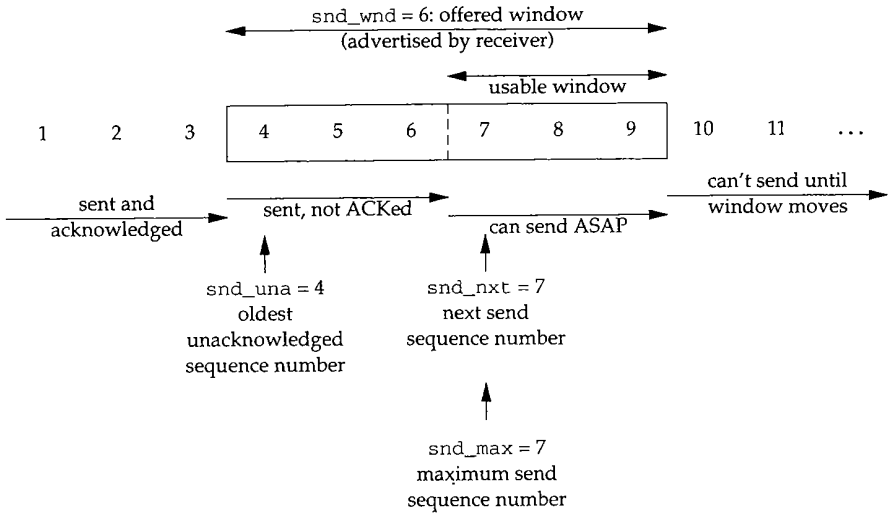


Figure 24.17 Example of send sequence space.

An *acceptable ACK* is one for which the following inequality holds:

```
snd_una < acknowledgment field <= snd_max
```

In Figure 24.17 an acceptable ACK has an acknowledgment field of 5, 6, or 7. An acknowledgment field less than or equal to `snd_una` is a duplicate ACK—it acknowledges data that has already been ACKed, or else `snd_una` would not have incremented past those bytes.

We encounter the following test a few times in `tcp_output`, which is true if a segment is being retransmitted:

```
snd_nxt < snd_max
```

Figure 24.18 shows the other end of the connection in Figure 24.17: the receive sequence space, assuming the segment containing sequence numbers 4, 5, and 6 has not been received yet. We show the three variables `rcv_nxt`, `rcv_wnd`, and `rcv_adv`.

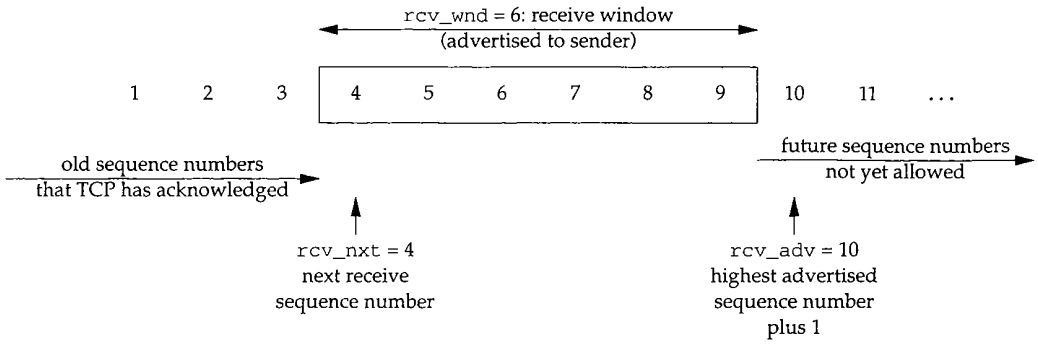


Figure 24.18 Example of receive sequence space.

The receiver considers a received segment valid if it contains data within the window, that is, if either of the following two inequalities is true:

```
rcv_nxt <= beginning sequence number of segment < rcv_nxt + rcv_wnd

rcv_nxt <= ending sequence number of segment < rcv_nxt + rcv_wnd
```

The beginning sequence number of a segment is just the sequence number field in the TCP header, `ti_seq`. The ending sequence number is the sequence number field plus the number of bytes of TCP data, minus 1.

For example, Figure 24.19 could represent the TCP segment containing the 3 bytes with sequence numbers 4, 5, and 6 in Figure 24.17.

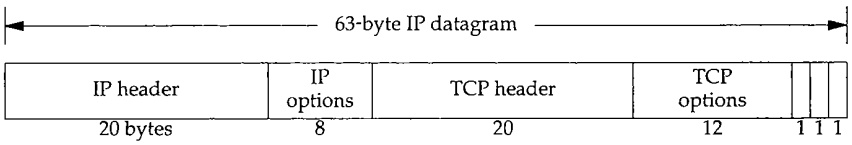


Figure 24.19 TCP segment transmitted as an IP datagram.

We assume that there are 8 bytes of IP options and 12 bytes of TCP options. Figure 24.20 shows the values of the relevant variables.

Variable	Value	Description
`ip_hl`	7	length of IP header + options in 32-bit words (= 28 bytes)
`ip_len`	63	length of IP datagram in bytes (20 + 8 + 20 + 12 + 3)
`ti_off`	8	length of TCP header + options in 32-bit words (= 32 bytes)
`ti_seq`	4	sequence number of first byte of data
`ti_len`	3	#bytes of TCP data: `ip_len` − (`ip_hl` × 4) − (`ti_off` × 4)
	6	sequence number of last byte of data: `ti_seq` + `ti_len` − 1

Figure 24.20 Values of variables corresponding to Figure 24.19.

`ti_len` is not a field that is transmitted in the TCP header. Instead, it is computed as shown in Figure 24.20 and stored in the overlaid IP structure (Figure 24.12) once the received header fields have been checksummed and verified. The last value in this figure is not stored in the header, but is computed from the other values when needed.

Modular Arithmetic with Sequence Numbers

A problem that TCP must deal with is that the sequence numbers are from a finite 32-bit number space: 0 through 4,294,967,295. If more than 2^{32} bytes of data are exchanged across a TCP connection, the sequence numbers will be reused. Sequence numbers wrap around from 4,294,967,295 to 0.

Even if less than 2^{32} bytes of data are exchanged, wrap around is still a problem because the sequence numbers for a connection don't necessarily start at 0. The initial sequence number for each direction of data flow across a connection can start anywhere between 0 and 4,294,967,295. This complicates the comparison of sequence numbers. For example, sequence number 1 is "greater than" 4,294,967,295, as we discuss below.

TCP sequence numbers are defined as `unsigned longs` in `tcp.h`:

```
typedef  u_long  tcp_seq;
```

The four macros shown in Figure 24.21 compare sequence numbers.

tcp_seq.h

```
40 #define SEQ_LT(a,b)     ((int)((a)-(b)) < 0)
41 #define SEQ_LEQ(a,b)    ((int)((a)-(b)) <= 0)
42 #define SEQ_GT(a,b)     ((int)((a)-(b)) > 0)
43 #define SEQ_GEQ(a,b)    ((int)((a)-(b)) >= 0)
```

tcp_seq.h

Figure 24.21 Macros for TCP sequence number comparison.

Example—Sequence Number Comparisons

Let's look at an example to see how TCP's sequence numbers operate. Assume 3-bit sequence numbers, 0 through 7. Figure 24.22 shows these eight sequence numbers, their 3-bit binary representation, and their two's complement representation. (To form the two's complement take the binary number, convert each 0 to a 1 and vice versa, then add 1.) We show the two's complement because to form $a - b$ we just add a to the two's complement of b.

The final three columns of this table are 0 minus x, 1 minus x, and 2 minus x. In these final three columns, if the value is considered to be a *signed* integer (notice the cast to `int` in all four macros in Figure 24.21), the value is less than 0 (the `SEQ_LT` macro) if the high-order bit is 1, and the value is greater than 0 (the `SEQ_GT` macro) if the high-order bit is 0 and the value is not 0. We show horizontal lines in these final three columns to distinguish between the four negative and the four nonnegative values.

If we look at the fourth column of Figure 24.22, (labeled "0 – x"), we see that 0 (i.e., x), is less than 1, 2, 3, and 4 (the high-order bit of the result is 1), and 0 is greater than 5, 6, and 7 (the high-order bit is 0 and the result is not 0). We show this relationship pictorially in Figure 24.23.

x	binary	two's complement	0 – x	1 – x	2 – x
0	000	000	000	001	010
1	001	111	111	000	001
2	010	110	110	111	000
3	011	101	101	110	111
4	100	100	100	101	110
5	101	011	011	100	101
6	110	010	010	011	100
7	111	001	001	010	011

Figure 24.22 Example using 3-bit sequence numbers.

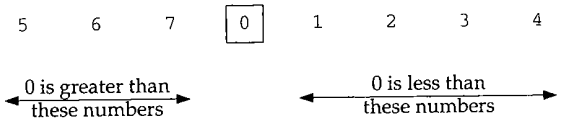


Figure 24.23 TCP sequence number comparisons for 3-bit sequence numbers.

Figure 24.24 shows a similar figure using the fifth row of the table (1 – x).

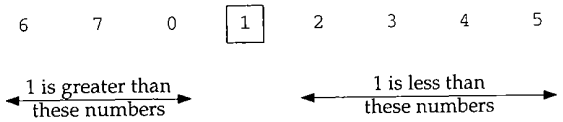


Figure 24.24 TCP sequence number comparisons for 3-bit sequence numbers.

Figure 24.25 is another representation of the two previous figures, using circles to reiterate the wrap around of sequence numbers.

Figure 24.25 Another way to visualize Figures 24.23 and 24.24.

With regard to TCP, these sequence number comparisons determine whether a given sequence number is in the future or in the past (a retransmission). For example, using Figure 24.24, if TCP is expecting sequence number 1 and sequence number 6 arrives, since 6 is less than 1 using the sequence number arithmetic we showed, the data byte is considered a retransmission of a previously received data byte and is discarded. But if sequence number 5 is received, since it is greater than 1 it is considered a future

data byte and is saved by TCP, awaiting the arrival of the missing bytes 2, 3, and 4 (assuming byte 5 is within the receive window).

Figure 24.26 is an expansion of the left circle in Figure 24.25, using TCP's 32-bit sequence numbers instead of 3-bit sequence numbers.

Figure 24.26 Comparisons against 0, using 32-bit sequence numbers.

The right circle in Figure 24.26 is to reiterate that one-half of the 32-bit sequence space uses 2^{31} numbers.

24.8 `tcp_init` Function

The `domaininit` function calls TCP's initialization function, `tcp_init` (Figure 24.27), at system initialization time.

tcp_subr.c

```
43 void
44 tcp_init()
45 {
46     tcp_iss = 1;                    /* wrong */
47     tcb.inp_next = tcb.inp_prev = &tcb;

48     if (max_protohdr < sizeof(struct tcpiphdr))
49                 max_protohdr = sizeof(struct tcpiphdr);
50     if (max_linkhdr + sizeof(struct tcpiphdr) > MHLEN)
51                 panic("tcp_init");
52 }
```

tcp_subr.c

Figure 24.27 `tcp_init` function.

Set initial send sequence number (ISS)

46 The initial send sequence number (ISS), `tcp_iss`, is initialized to 1. As the comment indicates, this is wrong. We discuss the implications behind this choice shortly, when we describe TCP's *quiet time*. Compare this to the initialization of the IP identifier in Figure 7.23, which used the time-of-day clock.

Initialize linked list of TCP Internet PCBs

47 The next and previous pointers in the head PCB (`tcb`) point to itself. This is an empty doubly linked list. The remainder of the `tcb` PCB is initialized to 0 (all uninitialized globals are set to 0), although the only other field used in this head PCB is `inp_lport`, the next TCP ephemeral port number to allocate. The first ephemeral port used by TCP will be 1024, for the reasons described in the solution for Exercise 22.4.

Calculate maximum protocol header length

48–51 If the maximum protocol header encountered so far is less than 40 bytes, `max_protohdr` is set to 40 (the size of the combined IP and TCP headers, without any options). This variable is described in Figure 7.17. If the sum of `max_linkhdr` (normally 16) and 40 is greater than the amount of data that fits into an mbuf with a packet header (100 bytes, `MHLEN` from Figure 2.7), the kernel panics (Exercise 24.2).

MSL and Quiet Time Concept

TCP requires any host that crashes without retaining any knowledge of the last sequence numbers used on active connections to refrain from sending any TCP segments for one MSL (2 minutes, the quiet time) on reboot. Few TCPs, if any, retain this knowledge over a crash or operator shutdown.

MSL is the *maximum segment lifetime*. Each implementation chooses a value for the MSL. It is the maximum amount of time any segment can exist in the network before being discarded. A connection that is actively closed remains in the CLOSE_WAIT state (Figure 24.15) for twice the MSL.

> RFC 793 [Postel 1981c] recommends an MSL of 2 minutes, but Net/3 uses an MSL of 30 seconds (the constant `TCPTV_MSL` in Figure 25.3).

The problem occurs if packets are delayed somewhere in the network (RFC 793 calls these *wandering duplicates*). Assume a Net/3 system starts up, initializes `tcp_iss` to 1 (as in Figure 24.27) and then crashes just after the sequence numbers wrap. We'll see in Section 25.5 that TCP increments `tcp_iss` by 128,000 every second, causing the wrap around of the ISS to occur about 9.3 hours after rebooting. Also, `tcp_iss` is incremented by 64,000 each time a `connect` is issued, which can cause the wrap around to occur earlier than 9.3 hours. The following scenario is one example of how an old segment can incorrectly be delivered to a connection:

1. A client and server have an established connection. The client's port number is 1024. The client sends a data segment with a starting sequence number of 2. This data segment gets trapped in a routing loop somewhere between the two end points and is not delivered to the server. This data segment becomes a wandering duplicate.
2. The client retransmits the data segment starting with sequence number 2, which is delivered to the server.
3. The client closes the connection.

4. The client host crashes.
5. The client host reboots about 40 seconds after crashing, causing TCP to initialize `tcp_iss` to 1 again.
6. Another connection is immediately established by the same client to the same server, using the same socket pair: the client uses 1024 again, and the server uses its well-known port. The client's SYN uses sequence number 1. This new connection using the same socket pair is called a new *incarnation* of the old connection.
7. The wandering duplicate from step 1 is delivered to the server, and it thinks this datagram belongs to the new connection, when it is really from the old connection.

Figure 24.28 is a time line of this sequence of steps.

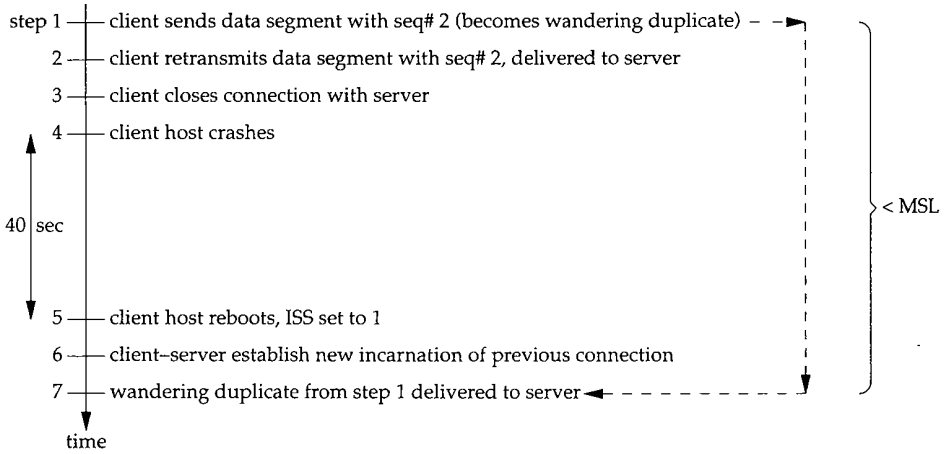


Figure 24.28 Example of old segment delivered to new incarnation of a connection.

This problem exists even if the rebooting TCP were to use an algorithm based on its time-of-day clock to choose the ISS on rebooting: regardless of the ISS for the previous incarnation of a connection, because of sequence number wrap it is possible for the ISS after rebooting to nearly equal the sequence number in use before the reboot.

Besides saving the sequence number of all established connections, the only other way around this problem is for the rebooting TCP to be quiet (i.e., not send any TCP segments) for MSL seconds after crashing. Few TCPs do this, however, since it takes most hosts longer than MSL seconds just to reboot.

24.9 Summary

This chapter is an introduction to the TCP source code in the six chapters that follow. TCP maintains its own control block for each connection, containing all the variable and state information for the connection.

A state transition diagram is defined for TCP that shows under what conditions TCP moves from one state to another and what segments get sent by TCP for each transition. This diagram shows how connections are established and terminated. We'll refer to this state transition diagram frequently in our description of TCP.

Every byte exchanged across a TCP connection has an associated sequence number, and TCP maintains numerous sequence numbers in the connection control block: some for sending and some for receiving (since TCP is full-duplex). Since these sequence numbers are from a finite 32-bit sequence space, they wrap around from the maximum value back to 0. We explained how the sequence numbers are compared to each other using less-than and greater-than tests, which we'll encounter repeatedly in the TCP code.

Finally, we looked at one of the simplest of the TCP functions, `tcp_init`, which initializes TCP's linked list of Internet PCBs. We also discussed TCP's choice of an initial send sequence number, which is used when actively opening a connection.

Exercises

24.1 What is the average number of bytes transmitted and received per connection from the statistics in Figure 24.5?

24.2 Is the kernel panic in `tcp_init` reasonable?

24.3 Execute `netstat -a` to see how many TCP end points your system currently has active.

25

TCP Timers

25.1 Introduction

We start our detailed description of the TCP source code by looking at the various TCP timers. We encounter these timers throughout most of the TCP functions.

TCP maintains seven timers for *each* connection. They are briefly described here, in the approximate order of their occurrence during the lifetime of a connection.

1. A *connection-establishment* timer starts when a SYN is sent to establish a new connection. If a response is not received within 75 seconds, the connection establishment is aborted.
2. A *retransmission* timer is set when TCP sends data. If the data is not acknowledged by the other end when this timer expires, TCP retransmits the data. The value of this timer (i.e., the amount of time TCP waits for an acknowledgment) is calculated dynamically, based on the round-trip time measured by TCP for this connection, and based on the number of times this data segment has been retransmitted. The retransmission timer is bounded by TCP to be between 1 and 64 seconds.
3. A *delayed ACK* timer is set when TCP receives data that must be acknowledged, but need not be acknowledged immediately. Instead, TCP waits up to 200 ms before sending the ACK. If, during this 200-ms time period, TCP has data to send on this connection, the pending acknowledgment is sent along with the data (called *piggybacking*).

4. A *persist* timer is set when the other end of a connection advertises a window of 0, stopping TCP from sending data. Since window advertisements from the other end are not sent reliably (that is, ACKs are not acknowledged, only data is acknowledged), there's a chance that a future window update, allowing TCP to send some data, can be lost. Therefore, if TCP has data to send and the other end advertises a window of 0, the persist timer is set and when it expires, 1 byte of data is sent to see if the window has opened. Like the retransmission timer, the persist timer value is calculated dynamically, based on the round-trip time. The value of this is bounded by TCP to be between 5 and 60 seconds.

5. A *keepalive* timer can be set by the process using the `SO_KEEPALIVE` socket option. If the connection is idle for 2 hours, the keepalive timer expires and a special segment is sent to the other end, forcing it to respond. If the expected response is received, TCP knows that the other host is still up, and TCP won't probe it again until the connection is idle for another 2 hours. Other responses to the keepalive probe tell TCP that the other host has crashed and rebooted. If no response is received to a fixed number of keepalive probes, TCP assumes that the other end has crashed, although it can't distinguish between the other end being down (i.e., it crashed and has not yet rebooted) and a temporary lack of connectivity to the other end (i.e., an intermediate router or phone line is down).

6. A *FIN_WAIT_2* timer. When a connection moves from the FIN_WAIT_1 state to the FIN_WAIT_2 state (Figure 24.15) *and* the connection cannot receive any more data (implying the process called `close`, instead of taking advantage of TCP's half-close with `shutdown`), this timer is set to 10 minutes. When this timer expires it is reset to 75 seconds, and when it expires the second time the connection is dropped. The purpose of this timer is to avoid leaving a connection in the FIN_WAIT_2 state forever, if the other end never sends a FIN. (We don't show this timeout in Figure 24.15.)

7. A *TIME_WAIT* timer, often called the *2MSL* timer. The term *2MSL* means twice the MSL, the maximum segment lifetime defined in Section 24.8. It is set when a connection enters the TIME_WAIT state (Figure 24.15), that is, when the connection is actively closed. Section 18.6 of Volume 1 describes the reasoning for the 2MSL wait state in detail. The timer is set to 1 minute (Net/3 uses an MSL of 30 seconds) when the connection enters the TIME_WAIT state and when it expires, the TCP control block and Internet PCB are deleted, allowing that socket pair to be reused.

TCP has two timer functions: one is called every 200 ms (the fast timer) and the other every 500 ms (the slow timer). The delayed ACK timer is different from the other six: when the delayed ACK timer is set for a connection it means that a delayed ACK must be sent the next time the 200-ms timer expires (i.e., the elapsed time is between 0 and 200 ms). The other six timers are decremented every 500 ms, and only when the counter reaches 0 does the corresponding action take place.

25.2 Code Introduction

The delayed ACK timer is enabled for a connection when the TF_DELACK flag (Figure 24.14) is set in the TCP control block. The array t_timer in the TCP control block contains four (TCPT_NTIMERS) counters used to implement the other six timers. The indexes into this array are shown in Figure 25.1. We describe briefly how the six timers (other than the delayed ACK timer) are implemented by these four counters.

Constant	Value	Description
TCPT_REXMT	0	retransmission timer
TCPT_PERSIST	1	persist timer
TCPT_KEEP	2	keepalive timer *or* connection-establishment timer
TCPT_2MSL	3	2MSL timer *or* FIN_WAIT_2 timer

Figure 25.1 Indexes into the t_timer array.

Each entry in the t_timer array contains the number of 500-ms clock ticks until the timer expires, with 0 meaning that the timer is not set. Since each timer is a short, if 16 bits hold a short, the maximum timer value is 16,383.5 seconds, or about 4.5 hours.

Notice in Figure 25.1 that four "timer counters" implement six TCP "timers," because some of the timers are mutually exclusive. We'll distinguish between the counters and the timers. The TCPT_KEEP counter implements both the keepalive timer and the connection-establishment timer, since the two timers are never used at the same time for a connection. Similarly, the 2MSL timer and the FIN_WAIT_2 timer are implemented using the TCPT_2MSL counter, since a connection is only in one state at a time. The first section of Figure 25.2 summarizes the implementation of the seven TCP timers. The second and third sections of the table show how four of the seven timers are initialized using three global variables from Figure 24.3 and two constants from Figure 25.3. Notice that two of the three globals are used with multiple timers. We've already said that the delayed ACK timer is tied to TCP's 200-ms timer, and we describe how the other two timers are set later in this chapter.

	conn. estab.	rexmit	delayed ACK	persist	keep-alive	FIN_-WAIT_2	2MSL
t_timer[TCPT_REXMT]		•					
t_timer[TCPT_PERSIST]				•			
t_timer[TCPT_KEEP]	•				•		
t_timer[TCPT_2MSL]						•	•
t_flags & TF_DELACK			•				
tcp_keepidle (2 hr)					•		
tcp_keepintvl (75 sec)					•	•	
tcp_maxidle (10 min)					•	•	
2 * TCPTV_MSL (60 sec)							•
TCPTV_KEEP_INIT (75 sec)	•						

Figure 25.2 Implementation of the seven TCP timers.

25.3

Figure 25.3 shows the fundamental timer values for the Net/3 implementation.

Constant	#500-ms clock ticks	#sec	Description
TCPTV_MSL	60	30	MSL, maximum segment lifetime
TCPTV_MIN	2	1	minimum value of retransmission timer
TCPTV_REXMTMAX	128	64	maximum value of retransmission timer
TCPTV_PERSMIN	10	5	minimum value of persist timer
TCPTV_PERSMAX	120	60	maximum value of persist timer
TCPTV_KEEP_INIT	150	75	connection-establishment timer value
TCPTV_KEEP_IDLE	14400	7200	idle time for connection before first probe (2 hours)
TCPTV_KEEPINTVL	150	75	time between probes when no response
TCPTV_SRTTBASE	0		special value to denote no measurements yet for connection
TCPTV_SRTTDFLT	6	3	default RTT when no measurements yet for connection

Figure 25.3 Fundamental timer values for the implementation.

Figure 25.4 shows other timer constants that we'll encounter.

Constant	Value	Description
TCP_LINGERTIME	120	maximum #seconds for SO_LINGER socket option
TCP_MAXRXTSHIFT	12	maximum #retransmissions waiting for an ACK
TCPTV_KEEPCNT	8	maximum #keepalive probes when no response received

25.4

Figure 25.4 Timer constants.

The TCPT_RANGESET macro, shown in Figure 25.5, sets a timer to a given value, making certain the value is between the specified minimum and maximum.

tcp_timer.h

```
102 #define TCPT_RANGESET(tv, value, tvmin, tvmax) { \
103     (tv) = (value); \
104     if ((tv) < (tvmin)) \
105         (tv) = (tvmin); \
106     else if ((tv) > (tvmax)) \
107         (tv) = (tvmax); \
108 }
```

tcp_timer.h

Figure 25.5 TCPT_RANGESET macro.

We see in Figure 25.3 that the retransmission timer and the persist timer have upper and lower bounds, since their values are calculated dynamically, based on the measured round-trip time. The other timers are set to constant values.

There is one additional timer that we allude to in Figure 25.4 but don't discuss in this chapter: the linger timer for a socket, set by the SO_LINGER socket option. This is a socket-level timer used by the close system call (Section 15.15). We will see in Figure 30.12 that when a socket is closed, TCP checks whether this socket option is set and whether the linger time is 0. If so, the connection is aborted with an RST instead of TCP's normal close.

25.3 tcp_canceltimers Function

The function tcp_canceltimers, shown in Figure 25.6, is called by tcp_input when the TIME_WAIT state is entered. All four timer counters are set to 0, which turns off the retransmission, persist, keepalive, and FIN_WAIT_2 timers, before tcp_input sets the 2MSL timer.

tcp_timer.c

```
107 void
108 tcp_canceltimers(tp)
109 struct tcpcb *tp;
110 {
111     int     i;

112     for (i = 0; i < TCPT_NTIMERS; i++)
113         tp->t_timer[i] = 0;
114 }
```

tcp_timer.c

Figure 25.6 tcp_canceltimers function.

25.4 tcp_fasttimo Function

The function tcp_fasttimo, shown in Figure 25.7, is called by pr_fasttimo every 200 ms. It handles only the delayed ACK timer.

tcp_timer.c

```
 41 void
 42 tcp_fasttimo()
 43 {
 44     struct inpcb *inp;
 45     struct tcpcb *tp;
 46     int     s = splnet();

 47     inp = tcb.inp_next;
 48     if (inp)
 49         for (; inp != &tcb; inp = inp->inp_next)
 50             if ((tp = (struct tcpcb *) inp->inp_ppcb) &&
 51                 (tp->t_flags & TF_DELACK)) {
 52                 tp->t_flags &= ~TF_DELACK;
 53                 tp->t_flags |= TF_ACKNOW;
 54                 tcpstat.tcps_delack++;
 55                 (void) tcp_output(tp);
 56             }
 57     splx(s);
 58 }
```

tcp_timer.c

Figure 25.7 tcp_fasttimo function, which is called every 200 ms.

Each Internet PCB on the TCP list that has a corresponding TCP control block is checked. If the TF_DELACK flag is set, it is cleared and the TF_ACKNOW flag is set instead. tcp_output is called, and since the TF_ACKNOW flag is set, an ACK is sent.

How can TCP have an Internet PCB on its PCB list that doesn't have a TCP control block (the test at line 50)? When a socket is created (the PRU_ATTACH request, in response to the socket system call) we'll see in Figure 30.11 that the creation of the Internet PCB is done first, followed by the creation of the TCP control block. Between these two operations a high-priority clock interrupt can occur (Figure 1.13), which calls tcp_fasttimo.

25.5 tcp_slowtimo Function

The function tcp_slowtimo, shown in Figure 25.8, is called by pr_slowtimo every 500 ms. It handles the other six TCP timers: connection establishment, retransmission, persist, keepalive, FIN_WAIT_2, and 2MSL.

71 tcp_maxidle is initialized to 10 minutes. This is the maximum amount of time TCP will send keepalive probes to another host, waiting for a response from that host. This variable is also used with the FIN_WAIT_2 timer, as we describe in Section 25.6. This initialization statement could be moved to tcp_init, since it only needs to be evaluated when the system is initialized (see Exercise 25.2).

Check each timer counter in all TCP control blocks

72–89 Each Internet PCB on the TCP list that has a corresponding TCP control block is checked. Each of the four timer counters for each connection is tested, and if nonzero, the counter is decremented. When the timer reaches 0, a PRU_SLOWTIMO request is issued. We'll see that this request calls the function tcp_timers, which we describe later in this chapter.

The fourth argument to tcp_usrreq is a pointer to an mbuf. But this argument is actually used for different purposes when the mbuf pointer is not required. Here we see the index i is passed, telling the request which timer has expired. The funny-looking cast of i to an mbuf pointer is to avoid a compile-time error.

Check if TCP control block has been deleted

90–93 Before examining the timers for a control block, a pointer to the next Internet PCB is saved in ipnxt. Each time the PRU_SLOWTIMO request returns, tcp_slowtimo checks whether the next PCB in the TCP list still points to the PCB that's being processed. If not, it means the control block has been deleted—perhaps the 2MSL timer expired or the retransmission timer expired and TCP is giving up on this connection—causing a jump to tpgone, skipping the remaining timers for this control block, and moving on to the next PCB.

Count idle time

94 t_idle is incremented for the control block. This counts the number of 500-ms clock ticks since the last segment was received on this connection. It is set to 0 by tcp_input when a segment is received on the connection and used for three purposes: (1) by the keepalive algorithm to send a probe after the connection is idle for 2 hours, (2) to drop a connection in the FIN_WAIT_2 state that is idle for 10 minutes and 75 seconds, and (3) by tcp_output to return to the slow start algorithm after the connection has been idle for a while.

95–

tcp_timer.c

```
64 void
65 tcp_slowtimo()
66 {
67     struct inpcb *ip, *ipnxt;
68     struct tcpcb *tp;
69     int     s = splnet();
70     int     i;

71     tcp_maxidle = TCPTV_KEEPCNT * tcp_keepintvl;
72     /*
73      * Search through tcb's and update active timers.
74      */
75     ip = tcb.inp_next;
76     if (ip == 0) {
77         splx(s);
78         return;
79     }
80     for (; ip != &tcb; ip = ipnxt) {
81         ipnxt = ip->inp_next;
82         tp = intotcpcb(ip);
83         if (tp == 0)
84             continue;
85         for (i = 0; i < TCPT_NTIMERS; i++) {
86             if (tp->t_timer[i] && --tp->t_timer[i] == 0) {
87                 (void) tcp_usrreq(tp->t_inpcb->inp_socket,
88                                   PRU_SLOWTIMO, (struct mbuf *) 0,
89                                   (struct mbuf *) i, (struct mbuf *) 0);
90                 if (ipnxt->inp_prev != ip)
91                     goto tpgone;
92             }
93         }
94         tp->t_idle++;
95         if (tp->t_rtt)
96             tp->t_rtt++;
97       tpgone:
98         ;
99     }
100    tcp_iss += TCP_ISSINCR / PR_SLOWHZ;     /* increment iss */
101    tcp_now++;                  /* for timestamps */
102    splx(s);
103 }
```

tcp_timer.c

Figure 25.8 `tcp_slowtimo` function, which is called every 500 ms.

Increment RTT counter

95–96 If this connection is timing an outstanding segment, `t_rtt` is nonzero and counts the number of 500-ms clock ticks until that segment is acknowledged. It is initialized to 1 by `tcp_output` when a segment is transmitted whose RTT should be timed. `tcp_slowtimo` increments this counter.

FIN_WA

Increment initial send sequence number

100 `tcp_iss` was initialized to 1 by `tcp_init`. Every 500 ms it is incremented by 64,000: 128,000 (`TCP_ISSINCR`) divided by 2 (`PR_SLOWHZ`). This is a rate of about once every 8 microseconds, although `tcp_iss` is incremented only twice a second. We'll see that `tcp_iss` is also incremented by 64,000 each time a connection is established, either actively or passively.

RFC 793 specifies that the initial sequence number should increment roughly every 4 microseconds, or 250,000 times a second. The Net/3 value increments at about one-half this rate.

Increment RFC 1323 timestamp value

101 `tcp_now` is initialized to 0 on bootstrap and incremented every 500 ms. It is used by the timestamp option defined in RFC 1323 [Jacobson, Braden, and Borman 1992], which we describe in Section 26.6.

75–79 Notice that if there are no TCP connections active on the host (`tcb.inp_next` is null), neither `tcp_iss` nor `tcp_now` is incremented. This would occur only when the system is being initialized, since it would be rare to find a Unix system attached to a network without a few TCP servers active.

25.6 `tcp_timers` Function

The function `tcp_timers` is called by TCP's `PRU_SLOWTIMO` request (Figure 30.10):

```
case PRU_SLOWTIMO:
    tp = tcp_timers(tp, (int)nam);
```

when any one of the four TCP timer counters reaches 0 (Figure 25.8).

The structure of the function is a `switch` statement with one `case` per timer, as outlined in Figure 25.9.

tcp_timer.c

```
120 struct tcpcb *
121 tcp_timers(tp, timer)
122 struct tcpcb *tp;
123 int     timer;
124 {
125     int     rexmt;
126     switch (timer) {

                /* switch cases */

256     }
257     return (tp);
258 }
```

tcp_timer.c 127–135

Figure 25.9 `tcp_timers` function: general organization.

We now discuss three of the four timer counters (five of TCP's timers), saving the retransmission timer for Section 25.11.

FIN_WAIT_2 and 2MSL Timers

TCP's TCPT_2MSL counter implements two of TCP's timers.

1. FIN_WAIT_2 timer. When tcp_input moves from the FIN_WAIT_1 state to the FIN_WAIT_2 state *and* the socket cannot receive any more data (implying the process called close, instead of taking advantage of TCP's half-close with shutdown), the FIN_WAIT_2 timer is set to 10 minutes (tcp_maxidle). We'll see that this prevents the connection from staying in the FIN_WAIT_2 state forever.
2. 2MSL timer. When TCP enters the TIME_WAIT state, the 2MSL timer is set to 60 seconds (TCPTV_MSL times 2).

Figure 25.10 shows the case for the 2MSL timer—executed when the timer reaches 0.

tcp_timer.c

```
127         /*
128          * 2 MSL timeout in shutdown went off.  If we're closed but
129          * still waiting for peer to close and connection has been idle
130          * too long, or if 2MSL time is up from TIME_WAIT, delete connection
131          * control block.  Otherwise, check again in a bit.
132          */
133     case TCPT_2MSL:
134         if (tp->t_state != TCPS_TIME_WAIT &&
135             tp->t_idle <= tcp_maxidle)
136             tp->t_timer[TCPT_2MSL] = tcp_keepintvl;
137         else
138             tp = tcp_close(tp);
139         break;
```

tcp_timer.c

Figure 25.10 tcp_timers function: expiration of 2MSL timer counter.

2MSL timer

127–139 The puzzling logic in the conditional is because the two different uses of the TCPT_2MSL counter are intermixed (Exercise 25.4). Let's first look at the TIME_WAIT state. When the timer expires after 60 seconds, tcp_close is called and the control blocks are released. We have the scenario shown in Figure 25.11. This figure shows the series of function calls that occurs when the 2MSL timer expires. We also see that setting one of the timers for N seconds in the future ($2 \times N$ ticks), causes the timer to expire somewhere between $2 \times N - 1$ and $2 \times N$ ticks in the future, since the time until the first decrement of the counter is between 0 and 500 ms in the future.

FIN_WAIT_2 timer

127–139 If the connection state is not TIME_WAIT, the TCPT_2MSL counter is the FIN_WAIT_2 timer. As soon as the connection has been idle for more than 10 minutes (tcp_maxidle) the connection is closed. But if the connection has been idle for less than or equal to 10 minutes, the FIN_WAIT_2 timer is reset for 75 seconds in the future. Figure 25.12 shows the typical scenario.

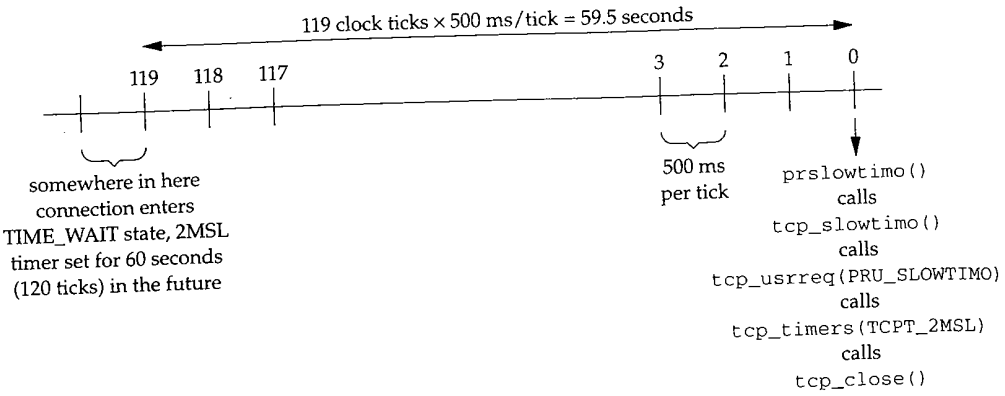


Figure 25.11 Setting and expiration of 2MSL timer in TIME_WAIT state.

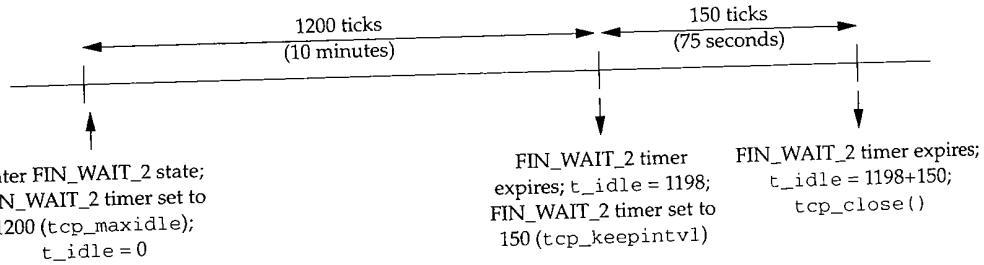


Figure 25.12 FIN_WAIT_2 timer to avoid infinite wait in FIN_WAIT_2 state.

The connection moves from the FIN_WAIT_1 state to the FIN_WAIT_2 state on the receipt of an ACK (Figure 24.15). Receiving this ACK sets `t_idle` to 0 and the FIN_WAIT_2 timer is set to 1200 (`tcp_maxidle`). In Figure 25.12 we show the up arrow just to the right of the tick mark starting the 10-minute period, to reiterate that the first decrement of the counter occurs between 0 and 500 ms after the counter is set. After 1199 ticks the timer expires, but since `t_idle` is incremented *after* the test and decrement of the four counters in Figure 25.8, `t_idle` is 1198. (We assume the connection is idle for this 10-minute period.) The comparison of 1198 as less than or equal to 1200 is true, so the FIN_WAIT_2 timer is set to 150 (`tcp_keepintvl`). When the timer expires again in 75 seconds, assuming the connection is still idle, `t_idle` is now 1348, the test is false, and `tcp_close` is called.

The reason for the 75-second timeout after the first 10-minute timeout is as follows: a connection in the FIN_WAIT_2 state is not dropped until the connection has been idle for *more than* 10 minutes. There's no reason to test `t_idle` until at least 10 minutes have expired, but once this time has passed, the value of `t_idle` is checked every 75 seconds. Since a duplicate segment could be received, say a duplicate of the ACK that

moved the connection from the FIN_WAIT_1 state to the FIN_WAIT_2 state, the 10-minute wait is restarted when the segment is received (since `t_idle` will be set to 0).

> Terminating an idle connection after more than 10 minutes in the FIN_WAIT_2 state violates the protocol specification, but this is practical. In the FIN_WAIT_2 state the process has called `close`, all outstanding data on the connection has been sent and acknowledged, the other end has acknowledged the FIN, and TCP is waiting for the process at the other end of the connection to issue its `close`. If the other process never closes its end of the connection, our end can remain in the FIN_WAIT_2 forever. A counter should be maintained for the number of connections terminated for this reason, to see how often this occurs.

Persist Timer

Figure 25.13 shows the `case` for when the persist timer expires.

tcp_timer.c

```
210         /*
211          * Persistence timer into zero window.
212          * Force a byte to be output, if possible.
213          */
214     case TCPT_PERSIST:
215         tcpstat.tcps_persisttimeo++;
216         tcp_setpersist(tp);
217         tp->t_force = 1;
218         (void) tcp_output(tp);
219         tp->t_force = 0;
220         break;
```

tcp_timer.c

Figure 25.13 `tcp_timers` function: expiration of persist timer.

Force window probe segment

210–220 When the persist timer expires, there is data to send on the connection but TCP has been stopped by the other end's advertisement of a zero-sized window. `tcp_setpersist` calculates the next value for the persist timer and stores it in the `TCPT_PERSIST` counter. The flag `t_force` is set to 1, forcing `tcp_output` to send 1 byte, even though the window advertised by the other end is 0.

Figure 25.14 shows typical values of the persist timer for a LAN, assuming the retransmission timeout for the connection is 1.5 seconds (see Figure 22.1 of Volume 1).

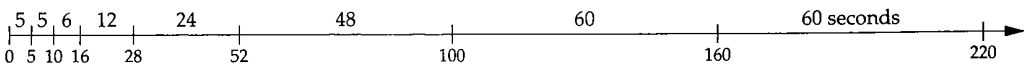


Figure 25.14 Time line of persist timer when probing a zero window.

Once the value of the persist timer reaches 60 seconds, TCP continues sending window probes every 60 seconds. The reason the first two values are both 5, and not 1.5 and 3, is that the persist timer is lower bounded at 5 seconds. It is also upper bounded at 60 seconds. The multiplication of each value by 2 to give the next value is called an *exponential backoff*, and we describe how it is calculated in Section 25.9.

Connection Establishment and Keepalive Timers

TCP's TCPT_KEEP counter implements two timers:

1. When a SYN is sent, the connection-establishment timer is set to 75 seconds (TCPTV_KEEP_INIT). This happens when connect is called, putting a connection into the SYN_SENT state (active open), or when a connection moves from the LISTEN to the SYN_RCVD state (passive open). If the connection doesn't enter the ESTABLISHED state within 75 seconds, the connection is dropped.
2. When a segment is received on a connection, tcp_input resets the keepalive timer for that connection to 2 hours (tcp_keepidle), and the t_idle counter for the connection is reset to 0. This happens for every TCP connection on the system, whether the keepalive option is enabled for the socket or not. If the keepalive timer expires (2 hours after the last segment was received on the connection), and if the socket option is set, a keepalive probe is sent to the other end. If the timer expires and the socket option is not set, the keepalive timer is just reset for 2 hours in the future.

Figure 25.16 shows the case for TCP's TCPT_KEEP counter.

Connection-establishment timer expires after 75 seconds

221–228 If the state is less than ESTABLISHED (Figure 24.16), the TCPT_KEEP counter is the connection-establishment timer. At the label dropit, tcp_drop is called to terminate the connection attempt with an error of ETIMEDOUT. We'll see that this error is the default error—if, for example, a soft error such as an ICMP host unreachable was received on the connection, the error returned to the process will be changed to EHOSTUNREACH instead of the default.

In Figure 30.4 we'll see that when TCP sends a SYN, two timers are initialized: the connection-establishment timer as we just described, with a value of 75 seconds, and the retransmission timer, to cause the SYN to be retransmitted if no response is received. Figure 25.15 shows these two timers.

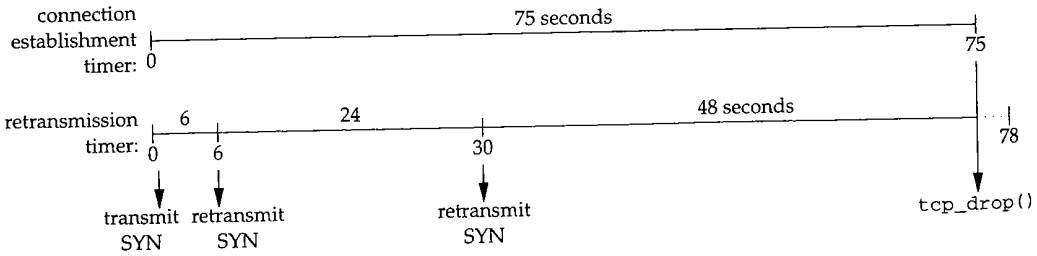


Figure 25.15 Connection-establishment timer and retransmission timer after SYN is sent. 229–230

The retransmission timer is initialized to 6 seconds for a new connection (Figure 25.19), and successive values are calculated to be 24 and 48 seconds. We describe how these values are calculated in Section 25.7. The retransmission timer causes the SYN to be

tcp_timer.c

```
221         /*
222          * Keep-alive timer went off; send something
223          * or drop connection if idle for too long.
224          */
225     case TCPT_KEEP:
226         tcpstat.tcps_keeptimeo++;
227         if (tp->t_state < TCPS_ESTABLISHED)
228             goto dropit;        /* connection establishment timer */
229         if (tp->t_inpcb->inp_socket->so_options & SO_KEEPALIVE &&
230             tp->t_state <= TCPS_CLOSE_WAIT) {
231             if (tp->t_idle >= tcp_keepidle + tcp_maxidle)
232                 goto dropit;
233             /*
234              * Send a packet designed to force a response
235              * if the peer is up and reachable:
236              * either an ACK if the connection is still alive,
237              * or an RST if the peer has closed the connection
238              * due to timeout or reboot.
239              * Using sequence number tp->snd_una-1
240              * causes the transmitted zero-length segment
241              * to lie outside the receive window;
242              * by the protocol spec, this requires the
243              * correspondent TCP to respond.
244              */
245             tcpstat.tcps_keepprobe++;
246             tcp_respond(tp, tp->t_template, (struct mbuf *) NULL,
247                         tp->rcv_nxt, tp->snd_una - 1, 0);
248             tp->t_timer[TCPT_KEEP] = tcp_keepintvl;
249         } else
250             tp->t_timer[TCPT_KEEP] = tcp_keepidle;
251         break;
252       dropit:
253         tcpstat.tcps_keepdrops++;
254         tp = tcp_drop(tp, ETIMEDOUT);
255         break;
```

tcp_timer.c

Figure 25.16 tcp_timers function: expiration of keepalive timer.

transmitted a total of three times, at times 0, 6, and 30. At time 75, 3 seconds before the retransmission timer would expire again, the connection-establishment timer expires, and tcp_drop terminates the connection attempt.

Keepalive timer expires after 2 hours of idle time

229–230 This timer expires after 2 hours of idle time on every connection, not just ones with the SO_KEEPALIVE socket option enabled. If the socket option is set, probes are sent only if the connection is in the ESTABLISHED or CLOSE_WAIT states (Figure 24.15). Once the process calls close (the states greater than CLOSE_WAIT), keepalive probes are not sent, even if the connection is idle for 2 hours.

Drop connection when no response

231–232 If the total idle time for the connection is greater than or equal to 2 hours (`tcp_keepidle`) plus 10 minutes (`tcp_maxidle`), the connection is dropped. This means that TCP has sent its limit of nine keepalive probes, 75 seconds apart (`tcp_keepintvl`), with no response. One reason TCP must send multiple keepalive probes before considering the connection dead is that the ACKs sent in response do not contain data and therefore are not reliably transmitted by TCP. An ACK that is a response to a keepalive probe can get lost.

Send a keepalive probe

233–248 If TCP hasn't reached the keepalive limit, `tcp_respond` sends a keepalive packet. The acknowledgment field of the keepalive packet (the fourth argument to `tcp_respond`) contains `rcv_nxt`, the next sequence number expected on the connection. The sequence number field of the keepalive packet (the fifth argument) deliberately contains `snd_una` minus 1, which is the sequence number of a byte of data that the other end has already acknowledged (Figure 24.17). Since this sequence number is outside the window, the other end must respond with an ACK, specifying the next sequence number it expects. 249–250

Figure 25.17 summarizes this use of the keepalive timer.

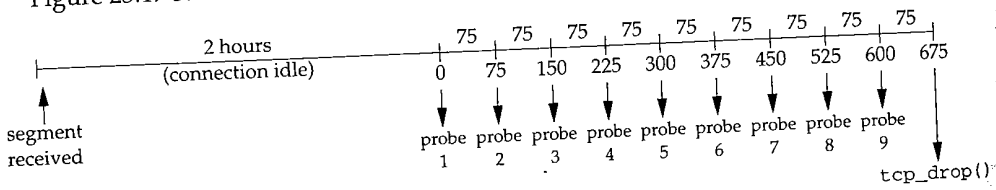


Figure 25.17 Summary of keepalive timer to detect unreachability of other end.

The nine keepalive probes are sent every 75 seconds, starting at time 0, through time 600. At time 675 (11.25 minutes after the 2-hour timer expired) the connection is dropped. Notice that nine keepalive probes are sent, even though the constant `TCPTV_KEEPCNT` (Figure 25.4) is 8. This is because the variable `t_idle` is incremented in Figure 25.8 *after* the timer is decremented, compared to 0, and possibly handled. When `tcp_input` receives a segment on a connection, it sets the keepalive timer to 14400 (`tcp_keepidle`) and `t_idle` to 0. The next time `tcp_slowtimo` is called, the keepalive timer is decremented to 14399 and `t_idle` is incremented to 1. About 2 hours later, when the keepalive timer is decremented from 1 to 0 and `tcp_timers` is called, the value of `t_idle` will be 14399. We can build the table in Figure 25.18 to see the value of `t_idle` each time `tcp_timers` is called.

The code in Figure 25.16 is waiting for `t_idle` to be greater than or equal to 15600 (`tcp_keepidle` + `tcp_maxidle`) and that only happens at time 675 in Figure 25.17, after nine keepalive probes have been sent.

25.7

probe#	time in Figure 25.17	t_idle
1	0	14399
2	75	14549
3	150	14699
4	225	14849
5	300	14999
6	375	15149
7	450	15299
8	525	15449
9	600	15599
	675	15749

Figure 25.18 The value of `t_idle` when `tcp_timers` is called for keepalive processing.

Reset keepalive timer

249–250 If the socket option is not set or the connection state is greater than CLOSE_WAIT, the keepalive timer for this connection is reset to 2 hours (`tcp_keepidle`).

Unfortunately the counter `tcps_keepdrops` (line 253) counts both uses of the `TCPT_KEEP` counter: the connection-establishment timer and the keepalive timer.

25.7 Retransmission Timer Calculations

The timers that we've described so far in this chapter have fixed times associated with them: 200 ms for the delayed ACK timer, 75 seconds for the connection-establishment timer, 2 hours for the keepalive timer, and so on. The final two timers that we describe, the retransmission timer and the persist timer, have values that depend on the measured RTT for the connection. Before going through the source code that calculates and sets these timers we need to understand how TCP measures the RTT for a connection.

Fundamental to the operation of TCP is setting a retransmission timer when a segment is transmitted and an ACK is required from the other end. If the ACK is not received when the retransmission timer expires, the segment is retransmitted. TCP requires an ACK for data segments but does not require an ACK for a segment without data (i.e., a pure ACK segment). If the calculated retransmission timeout is too small, it can expire prematurely, causing needless retransmissions. If the calculated value is too large, after a segment is lost, additional time is lost before the segment is retransmitted, degrading performance. Complicating this is that the round-trip times between two hosts can vary widely and dynamically over the course of a connection.

TCP in Net/3 calculates the retransmission timeout (*RTO*) by measuring the round-trip time (*nticks*) of data segments and keeping track of the smoothed RTT estimator (*srtt*) and a smoothed mean deviation estimator (*rttvar*). The mean deviation is a good approximation of the standard deviation, but easier to compute since, unlike the standard deviation, the mean deviation does not require square root calculations. [Jacobson 1988b] provides additional details on these RTT measurements, which lead to the following equations:

$$delta = nticks - srtt$$

$$srtt \leftarrow srtt + g \times delta$$

$$rttvar \leftarrow rttvar + h(|delta| - rttvar)$$

$$RTO = srtt + 4 \times rttvar$$

delta is the difference between the measured round trip just obtained (*nticks*) and the current smoothed RTT estimator (*srtt*). g is the gain applied to the RTT estimator and equals ⅛. h is the gain applied to the mean deviation estimator and equals ¼. The two gains and the multiplier 4 in the *RTO* calculation are purposely powers of 2, so they can be calculated using shift operations instead of multiplying or dividing.

> [Jacobson 1988b] specified $2 \times rttvar$ in the calculation of *RTO*, but after further research, [Jacobson 1990d] changed the value to $4 \times rttvar$, which is what appeared in the Net/1 implementation.

We now describe the variables and calculations used to calculate TCP's retransmission timer, as we'll encounter them throughout the TCP code. Figure 25.19 lists the variables in the control block related to the retransmission timer.

tcpcb member	Units	tcp_newtcpcb initial value	#sec	Description
t_srtt	ticks × 8	0		smoothed RTT estimator: *srtt* × 8
t_rttvar	ticks × 4	24	3	smoothed mean deviation estimator: *rttvar* × 4
t_rxtcur	ticks	12	6	current retransmission timeout: *RTO*
t_rttmin	ticks	2	1	minimum value for retransmission timeout
t_rxtshift	n.a.	0		index into tcp_backoff[] array (exponential backoff)

Figure 25.19 Control block variables for calculation of retransmission timer.

We show the `tcp_backoff` array at the end of Section 25.9. The `tcp_newtcpcb` function sets the initial values for these variables, and we cover it in the next section. The term *shift* in the variable `t_rxtshift` and its limit `TCP_MAXRXTSHIFT` is not entirely accurate. The former is not used for bit shifting, but as Figure 25.19 indicates, it is an index into an array.

The confusing part of TCP's timeout calculations is that the two smoothed estimators maintained in the C code (`t_srtt` and `t_rttvar`) are fixed-point integers, instead of floating-point values. This is done to avoid floating-point calculations within the kernel, but it complicates the code.

To keep the scaled and unscaled variables distinct, we'll use the italic variables *srtt* and *rttvar* to refer to the unscaled variables in the earlier equations, and `t_srtt` and `t_rttvar` to refer to the scaled variables in the TCP control block.

Figure 25.20 shows four constants we encounter, which define the scale factors of 8 for `t_srtt` and 4 for `t_rttvar`.

Constant	Value	Description	
TCP_RTT_SCALE	8	multiplier:	t_srtt = *srtt* × 8
TCP_RTT_SHIFT	3	shift:	t_srtt = *srtt* << 3
TCP_RTTVAR_SCALE	4	multiplier:	t_rttvar = *rttvar* × 4
TCP_RTTVAR_SHIFT	2	shift:	t_rttvar = *rttvar* << 2

Figure 25.20 Multipliers and shifts for RTT estimators.

25.8 tcp_newtcpcb Function

A new TCP control block is allocated and initialized by tcp_newtcpcb, shown in Figure 25.21. This function is called by TCP's PRU_ATTACH request when a new socket is created (Figure 30.2). The caller has previously allocated an Internet PCB for this connection, pointed to by the argument inp. We present this function now because it initializes the TCP timer variables.

tcp_subr.c

```
167 struct tcpcb *
168 tcp_newtcpcb(inp)
169 struct inpcb *inp;
170 {
171     struct tcpcb *tp;

172     tp = malloc(sizeof(*tp), M_PCB, M_NOWAIT);
173     if (tp == NULL)
174         return ((struct tcpcb *) 0);
175     bzero((char *) tp, sizeof(struct tcpcb));
176     tp->seg_next = tp->seg_prev = (struct tcpiphdr *) tp;
177     tp->t_maxseg = tcp_mssdflt;
178     tp->t_flags = tcp_do_rfc1323 ? (TF_REQ_SCALE | TF_REQ_TSTMP) : 0;
179     tp->t_inpcb = inp;
180     /*
181      * Init srtt to TCPTV_SRTTBASE (0), so we can tell that we have no
182      * rtt estimate.  Set rttvar so that srtt + 2 * rttvar gives
183      * reasonable initial retransmit time.
184      */
185     tp->t_srtt = TCPTV_SRTTBASE;
186     tp->t_rttvar = tcp_rttdflt * PR_SLOWHZ << 2;
187     tp->t_rttmin = TCPTV_MIN;
188     TCPT_RANGESET(tp->t_rxtcur,
189                   ((TCPTV_SRTTBASE >> 2) + (TCPTV_SRTTDFLT << 2)) >> 1,
190                   TCPTV_MIN, TCPTV_REXMTMAX);

191     tp->snd_cwnd = TCP_MAXWIN << TCP_MAX_WINSHIFT;
192     tp->snd_ssthresh = TCP_MAXWIN << TCP_MAX_WINSHIFT;

193     inp->inp_ip.ip_ttl = ip_defttl;
194     inp->inp_ppcb = (caddr_t) tp;
195     return (tp);
196 }
```

tcp_subr.c

Figure 25.21 tcp_newtcpcb function: create and initialize a new TCP control block.

167–175 The kernel's `malloc` function allocates memory for the control block, and `bzero` sets it to 0.

176 The two variables `seg_next` and `seg_prev` point to the reassembly queue for out-of-order segments received for this connection. We discuss this queue in detail in Section 27.9.

177–179 The maximum segment size to send, `t_maxseg`, defaults to 512 (`tcp_mssdflt`). This value can be changed by the `tcp_mss` function after an MSS option is received from the other end. (TCP also sends an MSS option to the other end when a new connection is established.) The two flags `TF_REQ_SCALE` and `TF_REQ_TSTMP` are set if the system is configured to request window scaling and timestamps as defined in RFC 1323 (the global `tcp_do_rfc1323` from Figure 24.3, which defaults to 1). The `t_inpcb` pointer in the TCP control block is set to point to the Internet PCB passed in by the caller.

180–185 The four variables `t_srtt`, `t_rttvar`, `t_rttmin`, and `t_rxtcur`, described in Figure 25.19, are initialized. First, the smoothed RTT estimator `t_srtt` is set to 0 (`TCPTV_SRTTBASE`), which is a special value that means no RTT measurements have been made yet for this connection. `tcp_xmit_timer` recognizes this special value when the first RTT measurement is made.

186–187 The smoothed mean deviation estimator `t_rttvar` is set to 24: 3 (`tcp_rttdflt`, from Figure 24.3) times 2 (`PR_SLOWHZ`) multiplied by 4 (the left shift of 2 bits). Since this scaled estimator is 4 times the variable *rttvar*, this value equals 6 clock ticks, or 3 seconds. The minimum *RTO*, stored in `t_rttmin`, is 2 ticks (`TCPTV_MIN`).

188–190 The current *RTO* in clock ticks is calculated and stored in `t_rxtcur`. It is bounded by a minimum value of 2 ticks (`TCPTV_MIN`) and a maximum value of 128 ticks (`TCPTV_REXMTMAX`). The value calculated as the second argument to `TCPT_RANGESET` is 12 ticks, or 6 seconds. This is the first *RTO* for the connection.

Understanding these C expressions involving the scaled RTT estimators can be a challenge. It helps to start with the unscaled equation and substitute the scaled variables. The unscaled equation we're solving is

$$RTO = srtt + 2 \times rttvar$$

where we use the multipler of 2 instead of 4 to calculate the first *RTO*.

> The use of the multiplier 2 instead of 4 appears to be a leftover from the original 4.3BSD Tahoe code [Paxson 1994].

Substituting the two scaling relationships

$$\texttt{t\_srtt} = 8 \times srtt$$

$$\texttt{t\_rttvar} = 4 \times rttvar$$

we get

$$RTO = \frac{\texttt{t\_srtt}}{8} + 2 \times \frac{\texttt{t\_rttvar}}{4}$$

$$= \frac{\frac{\texttt{t\_srtt}}{4} + \texttt{t\_rttvar}}{2}$$

which is the C code for the second argument to TCPT_RANGESET. In this code the variable t_rttvar is not used—the constant TCPTV_SRTTDFLT, whose value is 6 ticks, is used instead, and it must be multiplied by 4 to have the same scale as t_rttvar.

191–192 The congestion window (snd_cwnd) and slow start threshold (snd_ssthresh) are set to 1,073,725,440 (approximately one gigabyte), which is the largest possible TCP window if the window scale option is in effect. (Slow start and congestion avoidance are described in Section 21.6 of Volume 1.) It is calculated as the maximum value for the window size field in the TCP header (65535, TCP_MAXWIN) times 2^{14}, where 14 is the maximum value for the window scale factor (TCP_MAX_WINSHIFT). We'll see that when a SYN is sent or received on the connection, tcp_mss resets snd_cwnd to a single segment.

193–194 The default IP TTL in the Internet PCB is set to 64 (ip_defttl) and the PCB is set to point to the new TCP control block.

Not shown in this code is that numerous variables, such as the shift variable t_rxtshift, are implicitly initialized to 0 since the control block is initialized by bzero.

25.9 tcp_setpersist Function

The next function we look at that uses TCP's retransmission timeout calculations is tcp_setpersist. In Figure 25.13 we saw this function called when the persist timer expired. This timer is set when TCP has data to send on a connection, but the other end is advertising a window of 0. This function, shown in Figure 25.22, calculates and stores the next value for the timer.

tcp_output.c

```
493 void
494 tcp_setpersist(tp)
495 struct tcpcb *tp;
496 {
497     t = ((tp->t_srtt >> 2) + tp->t_rttvar) >> 1;

498     if (tp->t_timer[TCPT_REXMT])
499         panic("tcp_output REXMT");
500     /*
501      * Start/restart persistance timer.
502      */
503     TCPT_RANGESET(tp->t_timer[TCPT_PERSIST],
504                   t * tcp_backoff[tp->t_rxtshift],
505                   TCPTV_PERSMIN, TCPTV_PERSMAX);
506     if (tp->t_rxtshift < TCP_MAXRXTSHIFT)
507         tp->t_rxtshift++;
508 }
```

tcp_output.c

Figure 25.22 tcp_setpersist function: calculate and store a new value for the persist timer.

Check retransmission timer not enabled

493–499 A check is made that the retransmission timer is not enabled when the persist timer is about to be set, since the two timers are mutually exclusive: if data is being sent, the

other side must be advertising a nonzero window, but the persist timer is being set only if the advertised window is 0.

Calculate RTO

500–505 The variable `t` is set to the *RTO* value that was calculated at the beginning of the function. The equation being solved is

$$RTO = srtt + 2 \times rttvar$$

which is identical to the formula used at the end of the previous section. With substitution we get

$$RTO = \frac{\frac{t\_srtt}{4} + t\_rttvar}{2}$$

which is the value computed for the variable `t`.

Apply exponential backoff

506–507 An *exponential backoff* is also applied to the *RTO*. This is done by multiplying the *RTO* by a value from the `tcp_backoff` array:

```
int  tcp_backoff[TCP_MAXRXTSHIFT + 1] =
     { 1, 2, 4, 8, 16, 32, 64, 64, 64, 64, 64, 64, 64 };
```

When `tcp_output` initially sets the persist timer for a connection, the code is

```
tp->t_rxtshift = 0;
tcp_setpersist(tp);
```

so the first time `tcp_setpersist` is called, `t_rxtshift` is 0. Since the value of `tcp_backoff[0]` is 1, `t` is used as the persist timeout. The `TCPT_RANGESET` macro bounds this value between 5 and 60 seconds. `t_rxtshift` is incremented by 1 until it reaches a maximum of 12 (`TCP_MAXRXTSHIFT`), since `tcp_backoff[12]` is the final entry in the array.

25.10 `tcp_xmit_timer` Function

The next function we look at, `tcp_xmit_timer`, is called each time an RTT measurement is collected, to update the smoothed RTT estimator (*srtt*) and the smoothed mean deviation estimator (*rttvar*).

The argument `rtt` is the RTT measurement to be applied. It is the value *nticks* + 1, using the notation from Section 25.7. It can be from one of two sources:

1. If the timestamp option is present in a received segment, the measured RTT is the current time (`tcp_now`) minus the timestamp value. We'll examine the timestamp option in Section 26.6, but for now all we need to know is that `tcp_now` is incremented every 500 ms (Figure 25.8). When a data segment is sent, `tcp_now` is sent as the timestamp, and the other end echoes this timestamp in the acknowledgment it sends back. 1310–13.

2. If timestamps are not in use and a data segment is being timed, we saw in Figure 25.8 that the counter `t_rtt` is incremented every 500 ms for the connection. We also mentioned in Section 25.5 that this counter is initialized to 1, so when the acknowledgment is received the counter is the measured RTT (in ticks) plus 1.

Typical code in `tcp_input` that calls `tcp_xmit_timer` is

```
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);
```

If a timestamp was present in the segment (`ts_present`), the RTT estimators are updated using the current time (`tcp_now`) minus the echoed timestamp (`ts_ecr`) plus 1. (We describe the reason for adding 1 below.)

If a timestamp is not present, the RTT estimators are updated only if the received segment acknowledges a data segment that was being timed. There is only one RTT counter per TCP control block (`t_rtt`), so only one outstanding data segment can be timed per connection. The starting sequence number of that segment is stored in `t_rtseq` when the segment is transmitted, to tell when an acknowledgment is received that covers that sequence number. If the received acknowledgment number (`ti_ack`) is greater than the starting sequence number of the segment being timed (`t_rtseq`), the RTT estimators are updated using `t_rtt` as the measured RTT.

> Before RFC 1323 timestamps were supported, TCP measured the RTT only by counting clock ticks in `t_rtt`. But this variable is also used as a flag that specifies whether a segment is being timed (Figure 25.8): if `t_rtt` is greater than 0, then `tcp_slowtimo` adds 1 to it every 500 ms. Hence when `t_rtt` is nonzero, it is the number of ticks plus 1. We'll see shortly that `tcp_xmit_timer` always decrements its second argument by 1 to account for this offset. Therefore when timestamps are being used, 1 is added to the second argument to account for the decrement by 1 in `tcp_xmit_timer`.

The greater-than test of the sequence numbers is because ACKs are cumulative: if TCP sends and times a segment with sequence numbers 1–1024 (`t_rtseq` equals 1), then immediately sends (but can't time) a segment with sequence numbers 1025–2048, and then receives an ACK with `ti_ack` equal to 2049, this is an ACK for sequence numbers 1–2048 and the ACK acknowledges the first segment being timed as well as the second (untimed) segment. Notice that when RFC 1323 timestamps are in use there is no comparison of sequence numbers. If the other end sends a timestamp option, it chooses the echo reply value (`ts_ecr`) to allow TCP to calculate the RTT.

Figure 25.23 shows the first part of the function that updates the estimators.

Update smoothed estimators

1310–1325 Recall that `tcp_newtcpcb` initialized the smoothed RTT estimator (`t_srtt`) to 0, indicating that no measurements have been made for this connection. `delta` is the difference between the measured RTT and the current value of the smoothed RTT estimator, in unscaled ticks. `t_srtt` is divided by 8 to convert from scaled to unscaled ticks.

tcp_input.c

```
1310 void
1311 tcp_xmit_timer(tp, rtt)
1312 struct tcpcb *tp;
1313 short   rtt;
1314 {
1315     short   delta;

1316     tcpstat.tcps_rttupdated++;
1317     if (tp->t_srtt != 0) {
1318         /*
1319          * srtt is stored as fixed point with 3 bits after the
1320          * binary point (i.e., scaled by 8).  The following magic
1321          * is equivalent to the smoothing algorithm in rfc793 with
1322          * an alpha of .875 (srtt = rtt/8 + srtt*7/8 in fixed
1323          * point).  Adjust rtt to origin 0.
1324          */
1325         delta = rtt - 1 - (tp->t_srtt >> TCP_RTT_SHIFT);
1326         if ((tp->t_srtt += delta) <= 0)
1327             tp->t_srtt = 1;
1328         /*
1329          * We accumulate a smoothed rtt variance (actually, a
1330          * smoothed mean difference), then set the retransmit
1331          * timer to smoothed rtt + 4 times the smoothed variance.
1332          * rttvar is stored as fixed point with 2 bits after the
1333          * binary point (scaled by 4).  The following is
1334          * equivalent to rfc793 smoothing with an alpha of .75
1335          * (rttvar = rttvar*3/4 + |delta| / 4).  This replaces
1336          * rfc793's wired-in beta.
1337          */
1338         if (delta < 0)
1339             delta = -delta;
1340         delta -= (tp->t_rttvar >> TCP_RTTVAR_SHIFT);
1341         if ((tp->t_rttvar += delta) <= 0)
1342             tp->t_rttvar = 1;
1343     } else {
1344         /*
1345          * No rtt measurement yet - use the unsmoothed rtt.
1346          * Set the variance to half the rtt (so our first
1347          * retransmit happens at 3*rtt).
1348          */
1349         tp->t_srtt = rtt << TCP_RTT_SHIFT;
1350         tp->t_rttvar = rtt << (TCP_RTTVAR_SHIFT - 1);
1351     }
```

tcp_input.c

Figure 25.23 `tcp_xmit_timer` function: apply new RTT measurement to smoothed estimators.

1326–1327

The smoothed RTT estimator is updated using the equation

$$srtt \leftarrow srtt + g \times delta$$

Since the gain g is ⅛, this equation is

$$8 \times srtt \leftarrow 8 \times srtt + delta$$

which is

$$\texttt{t\_srtt} \leftarrow \texttt{t\_srtt} + delta$$

1328–1342 The mean deviation estimator is updated using the equation

$$rttvar \leftarrow rttvar + h(|delta| - rttvar)$$

Substituting ¼ for h and the scaled variable `t_rttvar` for $4 \times rttvar$, we get

$$\frac{\texttt{t\_rttvar}}{4} \leftarrow \frac{\texttt{t\_rttvar}}{4} + \frac{|delta| - \dfrac{\texttt{t\_rttvar}}{4}}{4}$$

which is

$$\texttt{t\_rttvar} \leftarrow \texttt{t\_rttvar} + |delta| - \frac{\texttt{t\_rttvar}}{4}$$

This final equation corresponds to the C code.

Initialize smoothed estimators on first RTT measurement

1343–1350 If this is the first RTT measured for this connection, the smoothed RTT estimator is initialized to the measured RTT. These calculations use the value of the argument `rtt`, which we said is the measured RTT plus 1 ($nticks + 1$), whereas the earlier calculation of `delta` subtracted 1 from `rtt`.

$$srtt = nticks + 1$$

or

$$\frac{\texttt{t\_srtt}}{8} = nticks + 1$$

which is

$$\texttt{t\_srtt} = (nticks + 1) \times 8$$

The smoothed mean deviation is set to one-half of the measured RTT:

$$rttvar = \frac{srtt}{2}$$

which is

$$\frac{\texttt{t\_rttvar}}{4} = \frac{nticks + 1}{2}$$

or

$$\texttt{t\_rttvar} = (nticks + 1) \times 2$$

The comment in the code states that this initial setting for the smoothed mean deviation yields an initial RTO of $3 \times srtt$. Since the RTO is calculated as

$$RTO = srtt + 4 \times rttvar$$

substituting for *rttvar* gives us

$$RTO = srtt + 4 \times \frac{srtt}{2}$$

which is indeed

$$RTO = 3 \times srtt$$

Figure 25.24 shows the final part of the `tcp_xmit_timer` function.

tcp_input.c

```
1352     tp->t_rtt = 0;
1353     tp->t_rxtshift = 0;

1354     /*
1355      * the retransmit should happen at rtt + 4 * rttvar.
1356      * Because of the way we do the smoothing, srtt and rttvar
1357      * will each average +1/2 tick of bias.  When we compute
1358      * the retransmit timer, we want 1/2 tick of rounding and
1359      * 1 extra tick because of +-1/2 tick uncertainty in the
1360      * firing of the timer.  The bias will give us exactly the
1361      * 1.5 tick we need.  But, because the bias is
1362      * statistical, we have to test that we don't drop below
1363      * the minimum feasible timer (which is 2 ticks).
1364      */
1365     TCPT_RANGESET(tp->t_rxtcur, TCP_REXMTVAL(tp),
1366                   tp->t_rttmin, TCPTV_REXMTMAX);

1367     /*
1368      * We received an ack for a packet that wasn't retransmitted;
1369      * it is probably safe to discard any error indications we've
1370      * received recently.  This isn't quite right, but close enough
1371      * for now (a route might have failed after we sent a segment,
1372      * and the return path might not be symmetrical).
1373      */
1374     tp->t_softerror = 0;
1375 }
```

tcp_input.c

Figure 25.24 `tcp_xmit_timer` function: final part.

1352–1353 The RTT counter (`t_rtt`) and the retransmission shift count (`t_rxtshift`) are both reset to 0 in preparation for timing and transmission of the next segment.

1354–1366 The next *RTO* to use for the connection (`t_rxtcur`) is calculated using the macro

```
#define  TCP_REXMTVAL(tp) \
         (((tp)->t_srtt >> TCP_RTT_SHIFT) + (tp)->t_rttvar)
```

This is the now-familiar equation

$$RTO = srtt + 4 \times rttvar$$

using the scaled variables updated by `tcp_xmit_timer`. Substituting these scaled variables for *srtt* and *rttvar*, we have

$$RTO = \frac{t\_srtt}{8} + 4 \times \frac{t\_rttvar}{4}$$

$$= \frac{\texttt{t\_srtt}}{8} + \texttt{t\_rttvar}$$

which corresponds to the macro. The calculated value for the *RTO* is bounded by the minimum *RTO* for this connection (`t_rttmin`, which `t_newtcpcb` set to 2 ticks), and 128 ticks (`TCPTV_REXMTMAX`).

Clear soft error variable

tcp_timer.c 1367–1374 Since `tcp_xmit_timer` is called only when an acknowledgment is received for a data segment that was sent, if a soft error was recorded for this connection (`t_softerror`), that error is discarded. We describe soft errors in more detail in the next section.

25.11 Retransmission Timeout: `tcp_timers` Function

We now return to the `tcp_timers` function and cover the final `case` that we didn't present in Section 25.6: the one that handles the expiration of the retransmission timer. This code is executed when a data segment that was transmitted has not been acknowledged by the other end within the *RTO*.

Figure 25.25 summarizes the actions caused by the retransmission timer. We assume that the first timeout calculated by `tcp_output` is 1.5 seconds, which is typical for a LAN (see Figure 21.1 of Volume 1).

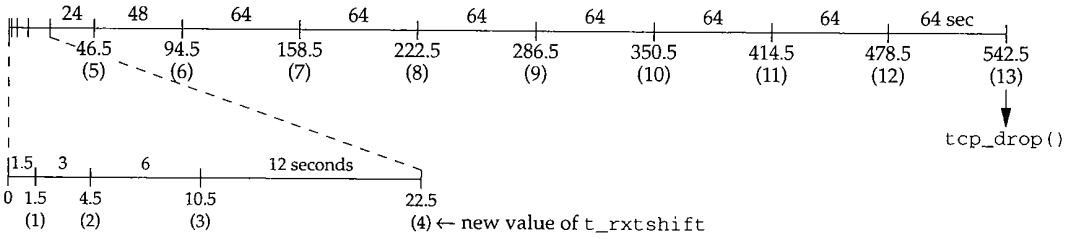


Figure 25.25 Summary of retransmission timer when sending data.

The x-axis is labeled with the time in seconds: 0, 1.5, 4.5, and so on. Below each of these numbers we show the value of `t_rxtshift` that is used in the code we're about to examine. Only after 12 retransmissions and a total of 542.5 seconds (just over 9 minutes) does TCP give up and drop the connection.

> RFC 793 recommended that an open of a new connection, active or passive, allow a parameter specifying the total timeout period for data sent by TCP. This is the total amount of time TCP will try to send a given segment before giving up and terminating the connection. The recommended default was 5 minutes.
>
> RFC 1122 requires that an application must be able to specify a parameter for a connection giving either the total number of retransmissions or the total timeout value for data sent by TCP. This parameter can be specified as "infinity," meaning TCP never gives up, allowing, perhaps, an interactive user the choice of when to give up.

We'll see in the code described shortly that Net/3 does not give the application any of this control: a fixed number of retransmissions (12) always occurs before TCP gives up, and the total timeout before giving up depends on the RTT.

The first half of the retransmission timeout case is shown in Figure 25.26.

tcp_timer.c

```
140         /*
141          * Retransmission timer went off.  Message has not
142          * been acked within retransmit interval.  Back off
143          * to a longer retransmit interval and retransmit one segment.
144          */
145     case TCPT_REXMT:
146         if (++tp->t_rxtshift > TCP_MAXRXTSHIFT) {
147             tp->t_rxtshift = TCP_MAXRXTSHIFT;
148             tcpstat.tcps_timeoutdrop++;
149             tp = tcp_drop(tp, tp->t_softerror ?
150                           tp->t_softerror : ETIMEDOUT);
151             break;
152         }
153         tcpstat.tcps_rexmttimeo++;
154         rexmt = TCP_REXMTVAL(tp) * tcp_backoff[tp->t_rxtshift];
155         TCPT_RANGESET(tp->t_rxtcur, rexmt,
156                       tp->t_rttmin, TCPTV_REXMTMAX);
157         tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;
158         /*
159          * If losing, let the lower level know and try for
160          * a better route.  Also, if we backed off this far,
161          * our srtt estimate is probably bogus.  Clobber it
162          * so we'll take the next rtt measurement as our srtt;
163          * move the current srtt into rttvar to keep the current
164          * retransmit times until then.
165          */
166         if (tp->t_rxtshift > TCP_MAXRXTSHIFT / 4) {
167             in_losing(tp->t_inpcb);
168             tp->t_rttvar += (tp->t_srtt >> TCP_RTT_SHIFT);
169             tp->t_srtt = 0;
170         }
171         tp->snd_nxt = tp->snd_una;
172         /*
173          * If timing a segment in this window, stop the timer.
174          */
175         tp->t_rtt = 0;
```

tcp_timer.c

Figure 25.26 `tcp_timers` function: expiration of retransmission timer, first half.

Increment shift count

146 The retransmission shift count (`t_rxtshift`) is incremented, and if the value exceeds 12 (`TCP_MAXRXTSHIFT`) it is time to drop the connection. This new value of `t_rxtshift` is what we show in Figure 25.25. Notice the difference between this dropping of a connection because an acknowledgment is not received from the other end in response to data sent by TCP, and the keepalive timer, which drops a connection after a

long period of inactivity and no response from the other end. Both report the error ETIMEDOUT to the process, unless a soft error is received for the connection.

Drop connection

147–152 A *soft error* is one that doesn't cause TCP to terminate an established connection or an attempt to establish a connection, but the soft error is recorded in case TCP gives up later. For example, if TCP retransmits a SYN segment to establish a connection, receiving nothing in response, the error returned to the process will be ETIMEDOUT. But if during the retransmissions an ICMP host unreachable is received for the connection, that is considered a soft error and stored in t_softerror by tcp_notify. If TCP finally gives up the retransmissions, the error returned to the process will be EHOSTUNREACH instead of ETIMEDOUT, providing more information to the process. If TCP receives an RST on the connection in response to the SYN, that's considered a *hard error* and the connection is terminated immediately with an error of ECONNREFUSED (Figure 28.18).

Calculate new RTO

153–157 The next *RTO* is calculated using the TCP_REXMTVAL macro, applying an exponential backoff. In this code, t_rxtshift will be 1 the first time a given segment is retransmitted, so the *RTO* will be twice the value calculated by TCP_REXMTVAL. This value is stored in t_rxtcur and as the retransmission timer for the connection, t_timer[TCPT_REXMT]. The value stored in t_rxtcur is used in tcp_input when the retransmission timer is restarted (Figures 28.12 and 29.6).

Ask IP to find a new route

158–167 If this segment has been retransmitted four or more times, in_losing releases the cached route (if there is one), so when the segment is retransmitted by tcp_output (at the end of this case statement in Figure 25.27) a new, and hopefully better, route will be chosen. In Figure 25.25 in_losing is called each time the retransmission timer expires, starting with the retransmission at time 22.5.

Clear estimators

168–170 The smoothed RTT estimator (t_srtt) is set to 0, which is what t_newtcpcb did. This forces tcp_xmit_timer to use the next measured RTT as the smoothed RTT estimator. This is done because the retransmitted segment has been sent four or more times, implying that TCP's smoothed RTT estimator is probably way off. But if the retransmission timer expires again, at the beginning of this case statement the *RTO* is calculated by TCP_REXMTVAL. That calculation should generate the same value as it did for this retransmission (which will then be exponentially backed off), even though t_srtt is set to 0. (The retransmission at time 42.464 in Figure 25.28 is an example of what's happening here.)

To accomplish this the value of t_rttvar is changed as follows. The next time the *RTO* is calculated, the equation

$$RTO = \frac{\texttt{t\_srtt}}{8} + \texttt{t\_rttvar}$$

is evaluated. Since t_srtt will be 0, if t_rttvar is increased by t_srtt divided by

8, *RTO* will have the same value. If the retransmission timer expires again for this segment (e.g., times 84.064 through 217.184 in Figure 25.28), when this code is executed again `t_srtt` will be 0, so `t_rttvar` won't change.

Force retransmission of oldest unacknowledged data

171 The next send sequence number (`snd_nxt`) is set to the oldest unacknowledged sequence number (`snd_una`). Recall from Figure 24.17 that `snd_nxt` can be greater than `snd_una`. By moving `snd_nxt` back, the retransmission will be the oldest segment that hasn't been acknowledged.

Karn's algorithm

172–175 The RTT counter, `t_rtt`, is set to 0, in case the last segment transmitted was being timed. Karn's algorithm says that even if an ACK of that segment is received, since the segment is about to be retransmitted, any timing of the segment is worthless since the ACK could be for the first transmission or for the retransmission. The algorithm is described in [Karn and Partridge 1987] and in Section 21.3 of Volume 1. Therefore the only segments that are timed using the `t_rtt` counter and used to update the RTT estimators are those that are not retransmitted. We'll see in Figure 29.6 that the use of RFC 1323 timestamps overrides Karn's algorithm.

Slow Start and Congestion Avoidance

The second half of this `case` is shown in Figure 25.27. It performs slow start and congestion avoidance and retransmits the oldest unacknowledged segment.

Since a retransmission timeout has occurred, this is a strong indication of congestion in the network. TCP's *congestion avoidance algorithm* comes into play, and when a segment is eventually acknowledged by the other end, TCP's *slow start* algorithm will continue the data transmission on the connection at a slower rate. Sections 20.6 and 21.6 of Volume 1 describe the two algorithms in detail.

176–205 `win` is set to one-half of the current window size (the minimum of the receiver's advertised window, `snd_wnd`, and the sender's congestion window, `snd_cwnd`) in segments, not bytes (hence the division by `t_maxseg`). Its minimum value is two segments. This records one-half of the window size when the congestion occurred, assuming one cause of the congestion is our sending segments too rapidly into the network. This becomes the slow start threshold, `t_ssthresh` (which is stored in bytes, hence the multiplication by `t_maxseg`). The congestion window, `snd_cwnd`, is set to one segment, which forces slow start.

> This code is enclosed in braces because it was added between the 4.3BSD and Net/1 releases and required its own local variable (`win`).

206 The counter of consecutive duplicate ACKs, `t_dupacks` (which is used by the fast retransmit algorithm in Section 29.4), is set to 0. We'll see how this counter is used with TCP's fast retransmit and fast recovery algorithms in Chapter 29.

208 `tcp_output` resends a segment containing the oldest unacknowledged sequence number. This is the retransmission caused by the retransmission timer expiring.

tcp_timer.c

```
176         /*
177          * Close the congestion window down to one segment
178          * (we'll open it by one segment for each ack we get).
179          * Since we probably have a window's worth of unacked
180          * data accumulated, this "slow start" keeps us from
181          * dumping all that data as back-to-back packets (which
182          * might overwhelm an intermediate gateway).
183          *
184          * There are two phases to the opening: Initially we
185          * open by one mss on each ack.  This makes the window
186          * size increase exponentially with time.  If the
187          * window is larger than the path can handle, this
188          * exponential growth results in dropped packet(s)
189          * almost immediately.  To get more time between
190          * drops but still "push" the network to take advantage
191          * of improving conditions, we switch from exponential
192          * to linear window opening at some threshhold size.
193          * For a threshhold, we use half the current window
194          * size, truncated to a multiple of the mss.
195          *
196          * (the minimum cwnd that will give us exponential
197          * growth is 2 mss.  We don't allow the threshhold
198          * to go below this.)
199          */
200         {
201             u_int   win = min(tp->snd_wnd, tp->snd_cwnd) / 2 / tp->t_maxseg;
202             if (win < 2)
203                 win = 2;
204             tp->snd_cwnd = tp->t_maxseg;
205             tp->snd_ssthresh = win * tp->t_maxseg;
206             tp->t_dupacks = 0;
207         }
208         (void) tcp_output(tp);
209         break;
```

tcp_timer.c

Figure 25.27 tcp_timers function: expiration of retransmission timer, second half.

Accuracy

How accurate are these estimators that TCP maintains? At first they appear too coarse, since the RTTs are measured in multiples of 500 ms. The mean and mean deviation are maintained with additional accuracy (factors of 8 and 4 respectively), but LANs have RTTs on the order of milliseconds, and a transcontinental RTT is around 60 ms. What these estimators provide is a solid upper bound on the RTT so that the retransmission timeout can be set without worrying that the timeout is too small, causing unnecessary and wasteful retransmissions.

[Brakmo, O'Malley, and Peterson 1994] describe a TCP implementation that provides higher-resolution RTT measurements. This is done by recording the system clock (which has a much higher resolution than 500 ms) when a segment is transmitted and reading the system clock when the ACK is received, calculating a higher-resolution RTT.

The timestamp option provided by Net/3 (Section 26.6) can provide higher-resolution RTTs, but Net/3 sets the resolution of these timestamps to 500 ms.

25.12 An RTT Example

We now go through an actual example to see how the calculations are performed. We transfer 12288 bytes from the host `bsdi` to `vangogh.cs.berkeley.edu`. During the transfer we purposely bring down the PPP link being used and then bring it back up, to see how timeouts and retransmissions are handled. To transfer the data we use our `sock` program (described in Appendix C of Volume 1) with the `-D` option, to enable the `SO_DEBUG` socket option (Section 27.10). After the transfer is complete we examine the debug records left in the kernel's circular buffer using the `trpt`(8) program and print the desired timer variables from the TCP control block.

Figure 25.28 shows the calculations that occur at the various times. We use the notation *M*:*N* to mean that sequence numbers *M* through and including $N-1$ are sent. Each segment in this example contains 512 bytes. The notation "ack *M*" means that the acknowledgment field of the ACK is *M*. The column labeled "actual delta (ms)" shows the time difference between the RTT timer going on and going off. The column labeled "`rtt` (arg.)" shows the second argument to the `tcp_xmit_timer` function: the number of clock ticks plus 1 between the RTT timer going on and going off.

The function `tcp_newtcpcb` initializes `t_srtt`, `t_rttvar`, and `t_rxtcur` to the values shown at time 0.0.

The first segment timed is the initial SYN. When its ACK is received 365 ms later, `tcp_xmit_timer` is called with an `rtt` argument of 2. Since this is the first RTT measurement (`t_srtt` is 0), the `else` clause in Figure 25.23 calculates the first values of the smoothed estimators.

The data segment containing bytes 1 through 512 is the next segment timed, and the RTT variables are updated at time 1.259 when its ACK is received.

The next three segments show how ACKs are cumulative. The timer is started at time 1.260 when bytes 513 through 1024 are sent. Another segment is sent with bytes 1025 through 1536, and the ACK received at time 2.206 acknowledges both data segments. The RTT estimators are then updated, since the ACK covers the starting sequence number being timed (513).

The segment with bytes 1537 through 2048 is transmitted at time 2.206 and the timer is started. Just that segment is acknowledged at time 3.132, and the estimators updated.

The data segment at time 3.132 is timed and the retransmission timer is set to 5 ticks (the current value of `t_rxtcur`). Somewhere around this time the PPP link between the routers `sun` and `netb` is taken down and then brought back up, a procedure that takes a few minutes. When the retransmission timer expires at time 6.064, the code in Figure 25.26 is executed to update the RTT variables. `t_rxtshift` is incremented from 0 to 1 and `t_rxtcur` is set to 10 ticks (the exponential backoff). A segment starting with the oldest unacknowledged sequence number (`snd_una`, which is 3073) is retransmitted. After 5 seconds the timer expires again, `t_rxtshift` is incremented to 2, and the retransmission timer is set to 20 ticks.

xmit time	send	recv	RTT timer	actual delta (ms)	rtt arg.	t_srtt (ticks × 8)	t_rttvar (ticks × 4)	t_rxtcur (ticks)	t_rxtshift
0.0	SYN		on			0	24	12	
0.365		SYN,ACK	off	365	2	16	4	6	
0.365	ACK								
0.415	1:513		on						
1.259		ack 513	off	844	2	15	4	5	
1.260	513:1025		on						
1.261	1025:1537								
2.206		ack 1537	off	946	3	16	4	6	
2.206	1537:2049		on						
2.207	2049:2561								
2.209	2561:3073								
3.132		ack 2049	off	926	3	16	3	5	
3.132	3073:3585		on						
3.133	3585:4097								
3.736		ack 2561							
3.736	4097:4609								
3.737	4609:5121								
3.739		ack 3073							
3.739	5121:5633								
3.740	5633:6145								
6.064	3073:3585		off			16	3	10	1
11.264	3073:3585		off			16	3	20	2
21.664	3073:3585		off			16	3	40	3
42.464	3073:3585		off			0	5	80	4
84.064	3073:3585		off			0	5	128	5
150.624	3073:3585		off			0	5	128	6
217.184	3073:3585		off			0	5	128	7
217.944		ack 6145							
217.944	6145:6657		on						
217.945	6657:7169								
218.834		ack 6657	off	890	3	24	6	9	
218.834	7169:7681		on						
218.836	7681:8193								
219.209		ack 7169							
219.209	8193:8705								
219.760		ack 7681	off	926	2	22	7	9	
219.760	8705:9217		on						
220.103		ack 8705							
220.103	9217:9729								
220.105	9729:10241								
220.106	10241:10753								
220.821		ack 9217	off	1061	3	22	6	8	
220.821	10753:11265		on						
221.310		ack 9729							
221.310	11265:11777								
221.312		ack 10241							
221.312	11777:12289								
221.674		ack 10753							
221.955		ack 11265	off	1134	3	22	5	7	

Figure 25.28 Values of RTT variables and estimators during example.

When the retransmission timer expires at time 42.464, `t_srtt` is set to 0 and `t_rttvar` is set to 5. As we mentioned in our discussion of Figure 25.26, this leaves the calculation of `t_rxtcur` the same (so the next calculation yields 160), but by setting `t_srtt` to 0, the next time the RTT estimators are updated (at time 218.834), the measured RTT becomes the smoothed RTT, as if the connection were starting fresh.

The rest of the data transfer continues, and the estimators are updated a few more times.

25.13 Summary

The two functions `tcp_fasttimo` and `tcp_slowtimo` are called by the kernel every 200 ms and every 500 ms, respectively. These two functions drive TCP's per-connection timer maintenance.

TCP maintains the following seven timers for each connection:

- a connection-establishment timer,
- a retransmission timer,
- a delayed ACK timer,
- a persist timer,
- a keepalive timer,
- a FIN_WAIT_2 timer, and
- a 2MSL timer.

The delayed ACK timer is different from the other six, since when it is set it means a delayed ACK must be sent the next time TCP's 200-ms timer expires. The other six timers are counters that are decremented by 1 every time TCP's 500-ms timer expires. When any one of the counters reaches 0, the appropriate action is taken: drop the connection, retransmit a segment, send a keepalive probe, and so on, as described in this chapter. Since some of the timers are mutually exclusive, the six timers are really implemented using four counters, which complicates the code.

This chapter also introduced the recommended way to calculate values for the retransmission timer. TCP maintains two smoothed estimators for a connection: the round-trip time and the mean deviation of the RTT. Although the algorithms are simple and elegant, these estimators are maintained as scaled fixed-point numbers (to provide adequate precision without using floating-point code within the kernel), which complicates the code.

Exercises

25.1 How efficient is TCP's fast timeout function? (*Hint*: Look at the number of delayed ACKs in Figure 24.5.) Suggest alternative implementations.

25.2 Why do you think the initialization of `tcp_maxidle` is in the `tcp_slowtimo` function instead of the `tcp_init` function?

25.3 `tcp_slowtimo` increments `t_idle`, which we said counts the clock ticks since a segment was last received on the connection. Should TCP also count the idle time since a segment was last sent on a connection?

25.4 Rewrite the code in Figure 25.10 to separate the logic for the two different uses of the `TCPT_2MSL` counter.

25.5 75 seconds after the connection in Figure 25.12 enters the FIN_WAIT_2 state a duplicate ACK is received on the connection. What happens?

25.6 A connection has been idle for 1 hour when the application sets the `SO_KEEPALIVE` option. Will the first keepalive probe be sent 1 or 2 hours in the future?

25.7 Why is `tcp_rttdflt` a global variable and not a constant?

25.8 Rewrite the code related to Exercise 25.6 to implement the alternate behavior.

26

TCP Output

26.1 Introduction

The function `tcp_output` is called whenever a segment needs to be sent on a connection. There are numerous calls to this function from other TCP functions:

- `tcp_usrreq` calls it for various requests: `PRU_CONNECT` to send the initial SYN, `PRU_SHUTDOWN` to send a FIN, `PRU_RCVD` in case a window update can be sent after the process has read some data from the socket receive buffer, `PRU_SEND` to send data, and `PRU_SENDOOB` to send out-of-band data.
- `tcp_fasttimo` calls it to send a delayed ACK.
- `tcp_timers` calls it to retransmit a segment when the retransmission timer expires.
- `tcp_timers` calls it to send a persist probe when the persist timer expires.
- `tcp_drop` calls it to send an RST.
- `tcp_disconnect` calls it to send a FIN.
- `tcp_input` calls it when output is required or when an immediate ACK should be sent.
- `tcp_input` calls it when a pure ACK is processed by the header prediction code and there is more data to send. (A *pure ACK* is a segment without data that just acknowledges data.)
- `tcp_input` calls it when the third consecutive duplicate ACK is received, to send a single segment (the fast retransmit algorithm).

tcp_output first determines whether a segment should be sent or not. TCP output is controlled by numerous factors other than data being ready to send to the other end of the connection. For example, the other end might be advertising a window of size 0 that stops TCP from sending anything, the Nagle algorithm prevents TCP from sending lots of small segments, and slow start and congestion avoidance limit the amount of data TCP can send on a connection. Conversely, some functions set flags just to force tcp_output to send a segment, such as the TF_ACKNOW flag that means an ACK should be sent immediately and not delayed. If tcp_output decides not to send a segment, the data (if any) is left in the socket's send buffer for a later call to this function.

26.2 tcp_output Overview

tcp_output is a large function, so we'll discuss it in 14 parts. Figure 26.1 shows the outline of the function.

Is an ACK expected from the other end?

61 idle is true if the maximum sequence number sent (snd_max) equals the oldest unacknowledged sequence number (snd_una), that is, if an ACK is not expected from the other end. In Figure 24.17 idle would be 0, since an ACK is expected for sequence numbers 4–6, which have been sent but not yet acknowledged.

Go back to slow start

62–68 If an ACK is not expected from the other end and a segment has not been received from the other end in one RTO, the congestion window is set to one segment (t_maxseg bytes). This forces slow start to occur for this connection the next time a segment is sent. When a significant pause occurs in the data transmission ("significant" being more than the RTT), the network conditions can change from what was previously measured on the connection. Net/3 assumes the worst and returns to slow start.

Send more than one segment

69–70 When send is jumped to, a single segment is sent by calling ip_output. But if tcp_output determines that more than one segment can be sent, sendalot is set to 1, and the function tries to send another segment. Therefore, one call to tcp_output can result in multiple segments being sent.

26.3 Determine if a Segment Should be Sent

Sometimes tcp_output is called but a segment is not generated. For example, the PRU_RCVD request is generated when the socket layer removes data from the socket's receive buffer, passing the data to a process. It is possible that the process removed enough data that TCP should send a segment to the other end with a new window advertisement, but this is just a possibility, not a certainty. The first half of tcp_output determines if there is a reason to send a segment to the other end. If not, the function returns without sending a segment.

tcp_output.c

```
43 int
44 tcp_output(tp)
45 struct tcpcb *tp;
46 {
47     struct socket *so = tp->t_inpcb->inp_socket;
48     long    len, win;
49     int     off, flags, error;
50     struct mbuf *m;
51     struct tcpiphdr *ti;
52     u_char  opt[MAX_TCPOPTLEN];
53     unsigned optlen, hdrlen;
54     int     idle, sendalot;

55     /*
56      * Determine length of data that should be transmitted
57      * and flags that will be used.
58      * If there are some data or critical controls (SYN, RST)
59      * to send, then transmit; otherwise, investigate further.
60      */
61     idle = (tp->snd_max == tp->snd_una);
62     if (idle && tp->t_idle >= tp->t_rxtcur)
63         /*
64          * We have been idle for "a while" and no acks are
65          * expected to clock out any data we send --
66          * slow start to get ack "clock" running again.
67          */
68         tp->snd_cwnd = tp->t_maxseg;

69 again:
70     sendalot = 0;   /* set nonzero if more than one segment to output */

                /* look for a reason to send a segment; */
                /* goto send if a segment should be sent */

218     /*
219      * No reason to send a segment, just return.
220      */
221     return (0);

222 send:

                /* form output segment, call ip_output() */

489     if (sendalot)
490         goto again;
491     return (0);
492 }
```

tcp_output.c

Figure 26.1 `tcp_output` function: overview.

Figure 26.2 shows the first of the tests to determine whether a segment should be sent.

tcp_output.c

```
71      off = tp->snd_nxt - tp->snd_una;
72      win = min(tp->snd_wnd, tp->snd_cwnd);

73      flags = tcp_outflags[tp->t_state];
74      /*
75       * If in persist timeout with window of 0, send 1 byte.
76       * Otherwise, if window is small but nonzero
77       * and timer expired, we will send what we can
78       * and go to transmit state.
79       */
80      if (tp->t_force) {
81          if (win == 0) {
82              /*
83               * If we still have some data to send, then
84               * clear the FIN bit.  Usually this would
85               * happen below when it realizes that we
86               * aren't sending all the data.  However,
87               * if we have exactly 1 byte of unsent data,
88               * then it won't clear the FIN bit below,
89               * and if we are in persist state, we wind
90               * up sending the packet without recording
91               * that we sent the FIN bit.
92               *
93               * We can't just blindly clear the FIN bit,
94               * because if we don't have any more data
95               * to send then the probe will be the FIN
96               * itself.
97               */
98              if (off < so->so_snd.sb_cc)
99                  flags &= ~TH_FIN;
100             win = 1;
101         } else {
102             tp->t_timer[TCPT_PERSIST] = 0;
103             tp->t_rxtshift = 0;
104         }
105     }
```

tcp_output.c

Figure 26.2 `tcp_output` function: data is being forced out.

71–72 `off` is the offset in bytes from the beginning of the send buffer of the first data byte to send. The first `off` bytes in the send buffer, starting with `snd_una`, have already been sent and are waiting to be ACKed.

`win` is the minimum of the window advertised by the receiver (`snd_wnd`) and the congestion window (`snd_cwnd`).

73 The `tcp_outflags` array was shown in Figure 24.16. The value of this array that is fetched and stored in `flags` depends on the current state of the connection. `flags` contains the combination of the `TH_ACK`, `TH_FIN`, `TH_RST`, and `TH_SYN` flag bits to send to the other end. The other two flag bits, `TH_PUSH` and `TH_URG`, will be logically ORed into `flags` if necessary before the segment is sent.

74–105 The flag `t_force` is set nonzero when the persist timer expires or when out-of-band data is being sent. These two conditions invoke `tcp_output` as follows:

```
tp->t_force = 1;
error = tcp_output(tp);
tp->t_force = 0;
```

This forces TCP to send a segment when it normally wouldn't send anything.

If `win` is 0, the connection is in the persist state (since `t_force` is nonzero). The FIN flag is cleared if there is more data in the socket's send buffer. `win` must be set to 1 byte to force out a single byte.

If `win` is nonzero, out-of-band data is being sent, so the persist timer is cleared and the exponential backoff index, `t_rxtshift`, is set to 0.

Figure 26.3 shows the next part of `tcp_output`, which calculates how much data to send.

tcp_output.c

```
106     len = min(so->so_snd.sb_cc, win) - off;
107     if (len < 0) {
108         /*
109          * If FIN has been sent but not acked,
110          * but we haven't been called to retransmit,
111          * len will be -1.  Otherwise, window shrank
112          * after we sent into it.  If window shrank to 0,
113          * cancel pending retransmit and pull snd_nxt
114          * back to (closed) window.  We will enter persist
115          * state below.  If the window didn't close completely,
116          * just wait for an ACK.
117          */
118         len = 0;
119         if (win == 0) {
120             tp->t_timer[TCPT_REXMT] = 0;
121             tp->snd_nxt = tp->snd_una;
122         }
123     }
124     if (len > tp->t_maxseg) {
125         len = tp->t_maxseg;
126         sendalot = 1;
127     }
128     if (SEQ_LT(tp->snd_nxt + len, tp->snd_una + so->so_snd.sb_cc))
129         flags &= ~TH_FIN;

130     win = sbspace(&so->so_rcv);
```

tcp_output.c

Figure 26.3 `tcp_output` function: calculate how much data to send.

Calculate amount of data to send

106 `len` is the minimum of the number of bytes in the send buffer and `win` (which is the minimum of the receiver's advertised window and the congestion window, perhaps 1 byte if output is being forced). `off` is subtracted because that many bytes at the beginning of the send buffer have already been sent and are awaiting acknowledgment.

Check for window shrink

107–117 One way for `len` to be less than 0 occurs if the receiver *shrinks* the window, that is, the receiver moves the right edge of the window to the left. The following example demonstrates how this can happen. First the receiver advertises a window of 6 bytes and TCP transmits a segment with bytes 4, 5, and 6. TCP immediately transmits another segment with bytes 7, 8, and 9. Figure 26.4 shows the status of our end after the two segments are sent.

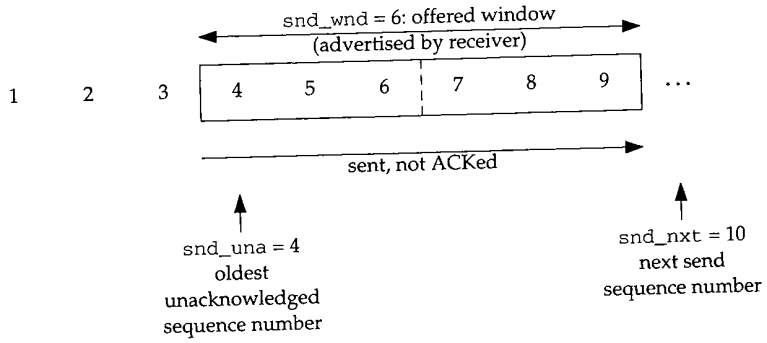
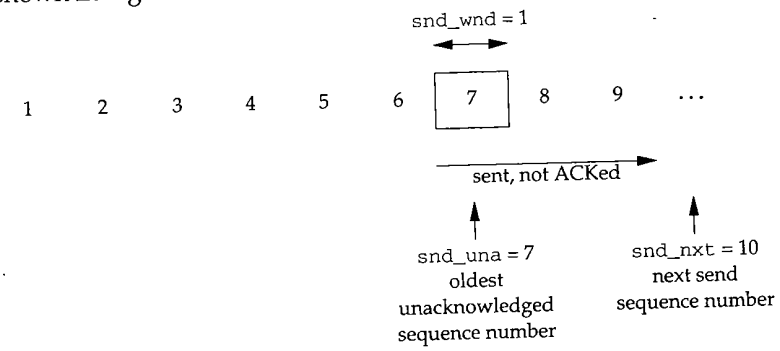


Figure 26.4 Send buffer after bytes 4 through 9 are sent.

Then an ACK is received with an acknowledgment field of 7 (acknowledging all data up through and including byte 6) but with a window of 1. The receiver has shrunk the window, as shown in Figure 26.5.



118–122

Figure 26.5 Send buffer after receiving acknowledgment of bytes 4 through 6.

124–127

Performing the calculations in Figures 26.2 and 26.3, after the window is shrunk, we have

128–129

```
off = snd_nxt - snd_una = 10 - 7 = 3
win = 1
len = min(so_snd.sb_cc, win) - off = min(3, 1) - 3 = -2
```

assuming the send buffer contains only bytes 7, 8, and 9.

Both RFC 793 and RFC 1122 strongly discourage shrinking the window. Nevertheless, implementations must be prepared for this. Handling scenarios such as this comes under the *Robustness Principle*, first mentioned in RFC 791: "Be liberal in what you accept, and conservative in what you send."

Another way for `len` to be less than 0 occurs if the FIN has been sent but not acknowledged and not retransmitted. (See Exercise 26.2.) We show this in Figure 26.6.

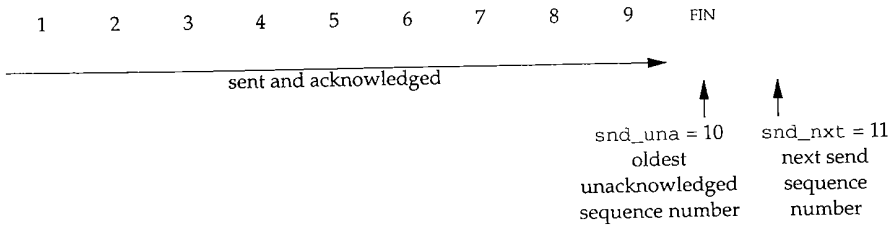


Figure 26.6 Bytes 1 through 9 have been sent and acknowledged, and then connection is closed.

This figure continues Figure 26.4, assuming the final segment with bytes 7, 8, and 9 is acknowledged, which sets `snd_una` to 10. The process then closes the connection, causing the FIN to be sent. We'll see later in this chapter that when the FIN is sent, `snd_nxt` is incremented by 1 (since the FIN takes a sequence number), which in this example sets `snd_nxt` to 11. The sequence number of the FIN is 10. Performing the calculations in Figures 26.2 and 26.3, we have

```
off = snd_nxt - snd_una = 11 - 10 = 1
win = 6
len = min(so_snd.sb_cc, win) - off = min(0, 6) - 1 = -1
```

We assume that the receiver advertises a window of 6, which makes no difference, since the number of bytes in the send buffer (0) is less than this.

Enter persist state

118–122 `len` is set to 0. If the advertised window is 0, any pending retransmission is canceled by setting the retransmission timer to 0. `snd_nxt` is also pulled to the left of the window by setting it to the value of `snd_una`. The connection will enter the persist state later in this function, and when the receiver finally opens its window, TCP starts retransmitting from the left of the window.

Send one segment at a time

124–127 If the amount of data to send exceeds one segment, `len` is set to a single segment and the `sendalot` flag is set to 1. As shown in Figure 26.1, this causes another loop through `tcp_output` after the segment is sent.

Turn off FIN flag if send buffer not emptied

128–129 If the send buffer is not being emptied by this output operation, the FIN flag must be cleared (in case it is set in `flags`). Figure 26.7 shows an example of this.

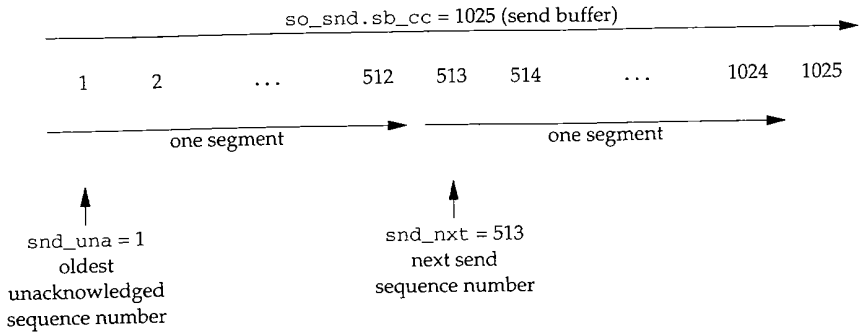


Figure 26.7 Example of send buffer not being emptied when FIN is set.

In this example the first 512-byte segment has already been sent (and is waiting to be acknowledged) and TCP is about to send the next 512-byte segment (bytes 512–1024). There is still 1 byte left in the send buffer (byte 1025) and the process closes the connection. `len` equals 512 (one segment), and the C expression becomes

```
SEQ_LT(1025, 1026)
```

which is true, so the FIN flag is cleared. If the FIN flag were mistakenly left on, TCP couldn't send byte 1025 to the receiver.

Calculate window advertisement

130 `win` is set to the amount of space available in the receive buffer, which becomes TCP's window advertisement to the other end. Be aware that this is the second use of this variable in this function. Earlier it contained the maximum amount of data TCP could send, but for the remainder of this function it contains the receive window advertised by this end of the connection. 149–150

151–152 The silly window syndrome (called *SWS* and described in Section 22.3 of Volume 1) occurs when small amounts of data, instead of full-sized segments, are exchanged across a connection. It can be caused by a receiver who advertises small windows and by a sender who transmits small segments. Correct avoidance of the silly window syndrome must be performed by both the sender and the receiver. Figure 26.8 shows silly window avoidance by the sender.

Sender silly window avoidance

142–143 If a full-sized segment can be sent, it is sent. 154–168

144–146 If an ACK is not expected (`idle` is true), or if the Nagle algorithm is disabled (`TF_NODELAY` is true) *and* TCP is emptying the send buffer, the data is sent. The Nagle algorithm (Section 19.4 of Volume 1) prevents TCP from sending less than a full-sized segment when an ACK is expected for the connection. It can be disabled using the `TCP_NODELAY` socket option. For a normal interactive connection (e.g., Telnet or Rlogin), if there is unacknowledged data, this `if` statement is false, since the Nagle algorithm is enabled by default .

147–148 If output is being forced by either the persist timer or sending out-of-band data, some data is sent.

tcp_output.c

```
131     /*
132      * Sender silly window avoidance.  If connection is idle
133      * and can send all data, a maximum segment,
134      * at least a maximum default-sized segment do it,
135      * or are forced, do it; otherwise don't bother.
136      * If peer's buffer is tiny, then send
137      * when window is at least half open.
138      * If retransmitting (possibly after persist timer forced us
139      * to send into a small window), then must resend.
140      */
141     if (len) {
142         if (len == tp->t_maxseg)
143             goto send;
144         if ((idle || tp->t_flags & TF_NODELAY) &&
145             len + off >= so->so_snd.sb_cc)
146             goto send;
147         if (tp->t_force)
148             goto send;
149         if (len >= tp->max_sndwnd / 2)
150             goto send;
151         if (SEQ_LT(tp->snd_nxt, tp->snd_max))
152             goto send;
153     }
```

tcp_output.c

Figure 26.8 `tcp_output` function: sender silly window avoidance.

149–150 If the receiver's window is at least half open, data is sent. This is to deal with peers that always advertise tiny windows, perhaps smaller than the segment size. The variable `max_sndwnd` is calculated by `tcp_input` as the largest window advertisement ever advertised by the other end. It is an attempt to guess the size of the other end's receive buffer and assumes the other end never reduces the size of its receive buffer.

151–152 If the retransmission timer expired, then a segment must be sent. `snd_max` is the highest sequence number that has been transmitted. We saw in Figure 25.26 that when the retransmission timer expires, `snd_nxt` is set to `snd_una`, that is, `snd_nxt` is moved to the left edge of the window, making it less than `snd_max`.

The next portion of `tcp_output`, shown in Figure 26.9, determines if TCP must send a segment just to advertise a new window to the other end. This is called a *window update*.

154–168 The expression

```
min(win, (long)TCP_MAXWIN << tp->rcv_scale)
```

is the smaller of the amount of available space in the socket's receive buffer (`win`) and the maximum size of the window allowed for this connection. This is the maximum window TCP can currently advertise to the other end. The expression

```
(tp->rcv_adv - tp->rcv_nxt)
```

is the number of bytes remaining in the last window advertisement that TCP sent to the other end. Subtracting this from the maximum window yields `adv`, the number of

tcp_output.c

```
154     /*
155      * Compare available window to amount of window
156      * known to peer (as advertised window less
157      * next expected input).  If the difference is at least two
158      * max size segments, or at least 50% of the maximum possible
159      * window, then want to send a window update to peer.
160      */
161     if (win > 0) {
162         /*
163          * "adv" is the amount we can increase the window,
164          * taking into account that we are limited by
165          * TCP_MAXWIN << tp->rcv_scale.
166          */
167         long    adv = min(win, (long) TCP_MAXWIN << tp->rcv_scale) -
168         (tp->rcv_adv - tp->rcv_nxt);
169         if (adv >= (long) (2 * tp->t_maxseg))
170             goto send;
171         if (2 * adv >= (long) so->so_rcv.sb_hiwat)
172             goto send;
173     }
```

tcp_output.c

Figure 26.9 `tcp_output` function: check if a window update should be sent.

bytes by which the window has opened. `rcv_nxt` is incremented by `tcp_input` when data is received in sequence, and `rcv_adv` is incremented by `tcp_output` in Figure 26.32 when the edge of the advertised window moves to the right.

Consider Figure 24.18 and assume that a segment with bytes 4, 5, and 6 is received and that these three bytes are passed to the process. Figure 26.10 shows the state of the receive space at this point in `tcp_output`.

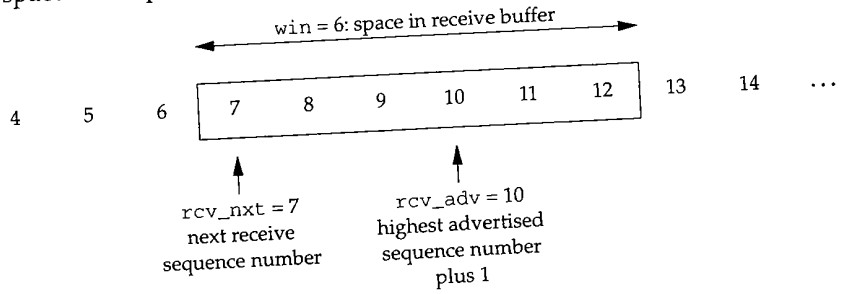


Figure 26.10 Transition from Figure 24.18 after bytes 4, 5, and 6 are received.

The value of `adv` is 3, since there are 3 more bytes of the receive space (bytes 10, 11, and 12) for the other end to fill.

169–170 If the window has opened by two or more segments, a window update is sent. When data is received as full-sized segments, this code causes every other received

segment to be acknowledged: TCP's ACK-every-other-segment property. (We show an example of this shortly.)

171–172 If the window has opened by at least 50% of the maximum possible window (the socket's receive buffer high-water mark), a window update is sent.

The next part of `tcp_output`, shown in Figure 26.11, checks whether various flags require TCP to send a segment.

tcp_output.c

```
174     /*
175      * Send if we owe peer an ACK.
176      */
177     if (tp->t_flags & TF_ACKNOW)
178         goto send;
179     if (flags & (TH_SYN | TH_RST))
180         goto send;
181     if (SEQ_GT(tp->snd_up, tp->snd_una))
182         goto send;
183     /*
184      * If our state indicates that FIN should be sent
185      * and we have not yet done so, or we're retransmitting the FIN,
186      * then we need to send.
187      */
188     if (flags & TH_FIN &&
189         ((tp->t_flags & TF_SENTFIN) == 0 || tp->snd_nxt == tp->snd_una))
190         goto send;
```

tcp_output.c

Figure 26.11 `tcp_output` function: should a segment should be sent?

174–178 If an immediate ACK is required, a segment is sent. The `TF_ACKNOW` flag is set by various functions: when the 200-ms delayed ACK timer expires, when a segment is received out of order (for the fast retransmit algorithm), when a SYN is received during the three-way handshake, when a persist probe is received, and when a FIN is received.

179–180 If `flags` specifies that a SYN or RST should be sent, a segment is sent.

181–182 If the urgent pointer, `snd_up`, is beyond the start of the send buffer, a segment is sent. The urgent pointer is set by the `PRU_SENDOOB` request (Figure 30.9).

183–190 If `flags` specifies that a FIN should be sent, a segment is sent only if the FIN has not already been sent, or if the FIN is being retransmitted. The flag `TF_SENTFIN` is set later in this function when the FIN is sent.

At this point in `tcp_output` there is no need to send a segment. Figure 26.12 shows the final piece of code before `tcp_output` returns.

191–217 If there is data in the send buffer to send (`so_snd.sb_cc` is nonzero) and both the retransmission timer and the persist timer are off, turn the persist timer on. This scenario happens when the window advertised by the other end is too small to receive a full-sized segment, and there is no other reason to send a segment.

218–221 `tcp_output` returns, since there is no reason to send a segment.

tcp_output.c

```
191     /*
192      * TCP window updates are not reliable, rather a polling protocol
193      * using 'persist' packets is used to ensure receipt of window
194      * updates.  The three 'states' for the output side are:
195      *  idle                 not doing retransmits or persists
196      *  persisting           to move a small or zero window
197      *  (re)transmitting     and thereby not persisting
198      *
199      * tp->t_timer[TCPT_PERSIST]
200      *      is set when we are in persist state.
201      * tp->t_force
202      *      is set when we are called to send a persist packet.
203      * tp->t_timer[TCPT_REXMT]
204      *      is set when we are retransmitting
205      * The output side is idle when both timers are zero.
206      *
207      * If send window is too small, there is data to transmit, and no
208      * retransmit or persist is pending, then go to persist state.
209      * If nothing happens soon, send when timer expires:
210      * if window is nonzero, transmit what we can,
211      * otherwise force out a byte.
212      */
213     if (so->so_snd.sb_cc && tp->t_timer[TCPT_REXMT] == 0 &&
214         tp->t_timer[TCPT_PERSIST] == 0) {
215         tp->t_rxtshift = 0;
216         tcp_setpersist(tp);
217     }
218     /*
219      * No reason to send a segment, just return.
220      */
221     return (0);
```

tcp_output.c

Figure 26.12 `tcp_output` function: enter persist state.

Example

A process writes 100 bytes, followed by a write of 50 bytes, on an idle connection. Assume a segment size of 512 bytes. When the first write occurs, the code in Figure 26.8 (lines 144–146) sends a segment with 100 bytes of data since the connection is idle and TCP is emptying the send buffer.

When 50-byte write occurs, the code in Figure 26.8 does not send a segment: the amount of data is not a full-sized segment, the connection is not idle (assume TCP is awaiting the ACK for the 100 bytes that it just sent), the Nagle algorithm is enabled by default, `t_force` is not set, and assuming a typical receive window of 4096, 50 is not greater than or equal to 2048. These 50 bytes remain in the send buffer, probably until the ACK for the 100 bytes is received. This ACK will probably be delayed by the other end, causing more delay in sending the final 50 bytes.

This example shows the timing delays that can occur when sending less than full-sized segments with the Nagle algorithm enabled. See also Exercise 26.12.

Example

This example demonstrates the ACK-every-other-segment property of TCP. Assume a connection is established with a segment size of 1024 bytes and a receive buffer size of 4096. There is no data to send—TCP is just receiving.

A window of 4096 is advertised in the ACK of the SYN, and Figure 26.13 shows the two variables `rcv_nxt` and `rcv_adv`. The receive buffer is empty.

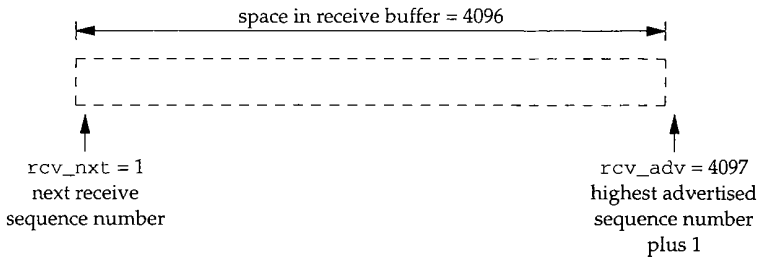


Figure 26.13 Receiver advertising a window of 4096.

The other end sends a segment with bytes 1–1024. `tcp_input` processes the segment, sets the delayed-ACK flag for the connection, and appends the 1024 bytes of data to the socket's receiver buffer (Figure 28.13). `rcv_nxt` is updated as shown in Figure 26.14.

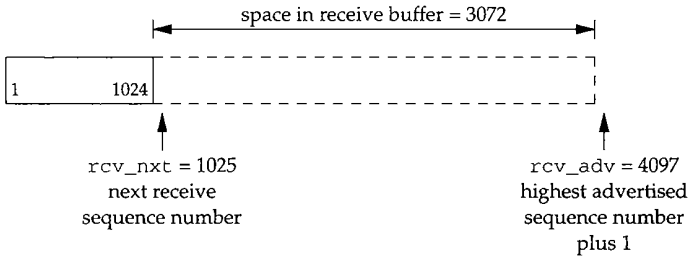


Figure 26.14 Transition from Figure 26.13 after bytes 1–1024 received.

The process reads the 1024 bytes in its socket receive buffer. We'll see in Figure 30.6 that the resulting `PRU_RCVD` request causes `tcp_output` to be called, because a window update might need to be sent after the process reads data from the receive buffer. When `tcp_output` is called, the two variables still have the values shown in Figure 26.14 and the only difference is that the amount of space in the receive buffer has increased to 4096 since the process has read the first 1024 bytes. The calculations in Figure 26.9 are performed:

```
adv = min(4096, 65535) - (4097 - 1025)
    = 1024
```

TCP_MAXWIN is 65535 and we assume a receive window scale shift of 0. Since the window has increased by less than two segments (2048), nothing is sent. But the delayed-ACK flag is still set, so if the 200-ms timer expires, an ACK will be sent.

When TCP receives the next segment with bytes 1025–2048, tcp_input processes the segment, sets the delayed-ACK flag for the connection (which was already on), and appends the 1024 bytes of data to the socket's receiver buffer. rcv_nxt is updated as shown in Figure 26.15.

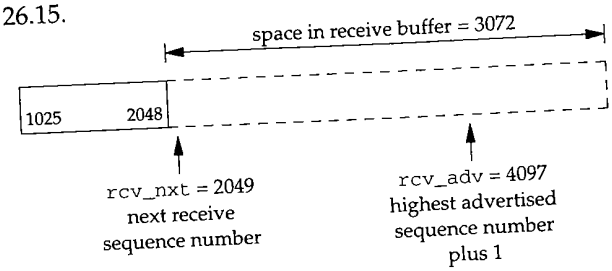


Figure 26.15 Transition from Figure 26.14 after bytes 1025–2048 received.

The process reads bytes 1025–2048 and tcp_output is called. The two variables still have the values shown in Figure 26.15, although the space in the receive buffer increases to 4096 when the process reads the 1024 bytes of data. The calculations in Figure 26.9 are performed:

```
adv = min(4096, 65535) - (4097 - 2049)
    = 2048
```

This value is now greater than or equal to two segments, so a segment is sent with an acknowledgment field of 2049 and an advertised window of 4096. This is a window update. The receiver is willing to receive bytes 2049 through 6145. We'll see later in this function that when this segment is sent, the value of rcv_adv also gets updated to 6145.

This example shows that when receiving data faster than the 200-ms delayed ACK timer, an ACK is sent when the receive window changes by more than two segments due to the process reading the data. If data is received for the connection but the process is not reading the data from the socket's receive buffer, the ACK-every-other-segment property won't occur. Instead the sender will only see the delayed ACKs, each advertising a smaller window, until the receive buffer is filled and the window goes to 0.

26.4 TCP Options

The TCP header can contain options. We digress to discuss these options since the next piece of tcp_output decides which options to send and constructs the options in the outgoing segment. Figure 26.16 shows the format of the options supported by Net/3.

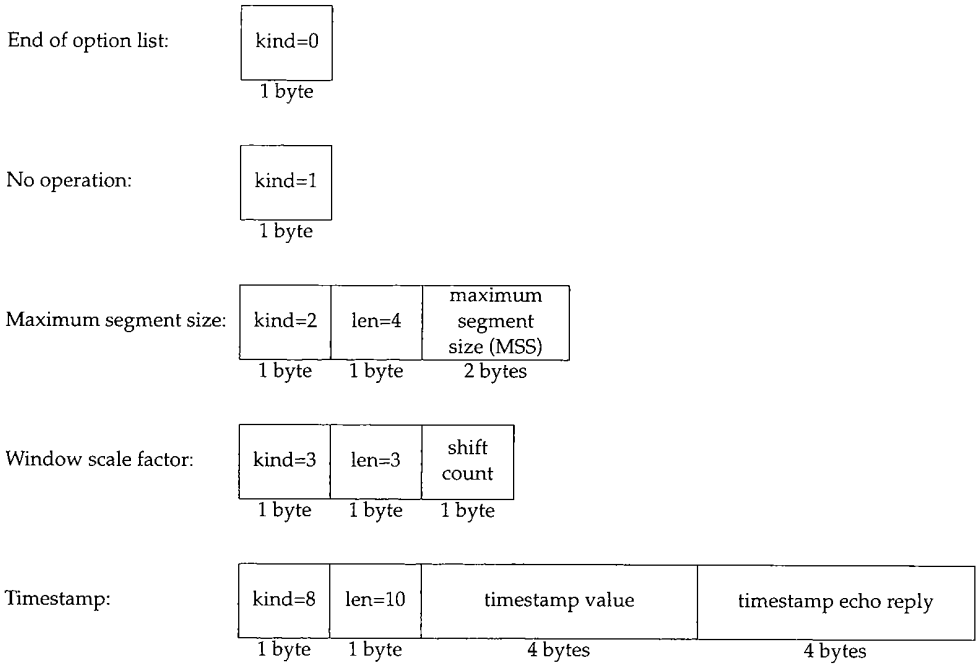


Figure 26.16 TCP options supported by Net/3.

Every option begins with a 1-byte *kind* that specifies the type of option. The first two options (with *kind*s of 0 and 1) are single-byte options. The other three are multibyte options with a *len* byte that follows the *kind* byte. The length is the total length, including the *kind* and *len* bytes.

The multibyte integers—the MSS and the two timestamp values—are stored in network byte order.

The final two options, window scale and timestamp, are new and therefore not supported by many systems. To provide interoperability with these older systems, the following rules apply.

1. TCP can send one of these options (or both) with the initial SYN segment corresponding to an active open (that is, a SYN without an ACK). Net/3 does this for both options if the global `tcp_do_rfc1323` is nonzero (it defaults to 1). This is done in `tcp_newtcpcb`.
2. The option is enabled only if the SYN reply from the other end also includes the desired option. This is handled in Figures 28.20 and 29.2.
3. If TCP performs a passive open and receives a SYN specifying the option, the response (the SYN plus ACK) must contain the option if TCP wants to enable the option. This is done in Figure 26.23.

Since a system must ignore options that it doesn't understand, the newer options are enabled by both ends only if both ends understand the option and both ends want the option enabled.

The processing of the MSS option is covered in Section 27.5. The next two sections summarize the Net/3 handling of the two newer options: window scale and timestamp.

Other options have been proposed. *kinds* of 4, 5, 6, and 7, called the selective-ACK and echo options, are defined in RFC 1072 [Jacobson and Braden 1988]. We don't show them in Figure 26.16 because the echo options were replaced with the timestamp option, and selective ACKs, as currently defined, are still under discussion and were not included in RFC 1323. Also, the T/TCP proposal for TCP transactions (RFC 1644 [Braden 1994], and Section 24.7 of Volume 1) specifies three options with *kinds* of 11, 12, and 13.

26.5 Window Scale Option

The window scale option, defined in RFC 1323, avoids the limitation of a 16-bit window size field in the TCP header (Figure 24.10). Larger windows are required for what are called *long fat pipes*, networks with either a high bandwidth or a long delay (i.e., a long RTT). Section 24.3 of Volume 1 gives examples of current networks that require larger windows to obtain maximum TCP throughput.

The 1-byte shift count in Figure 26.16 is between 0 (no scaling performed) and 14. This maximum value of 14 provides a maximum window of 1,073,725,440 bytes (65535×2^{14}). Internally Net/3 maintains window sizes as 32-bit values, not 16-bit values.

The window scale option can only appear in a SYN segment; therefore the scale factor is fixed in each direction when the connection is established.

The two variables `snd_scale` and `rcv_scale` in the TCP control block specify the shift count for the send window and the receive window, respectively. Both default to 0 for no scaling. Every 16-bit advertised window received from the other end is left shifted by `snd_scale` bits to obtain the real 32-bit advertised window size (Figure 28.6). Every time TCP sends a window advertisement to the other end, the internal 32-bit window size is right shifted by `rcv_scale` bits to give the value that is placed into the TCP header (Figure 26.29).

When TCP sends a SYN, either actively or passively, it chooses the value of `rcv_scale` to request, based on the size of the socket's receive buffer (Figures 28.7 and 30.4).

26.6 Timestamp Option

The timestamp option is also defined in RFC 1323 and lets the sender place a timestamp in every segment. The receiver sends the timestamp back in the acknowledgment, allowing the sender to calculate the RTT for each received ACK. Figure 26.17 summarizes the timestamp option and the variables involved.

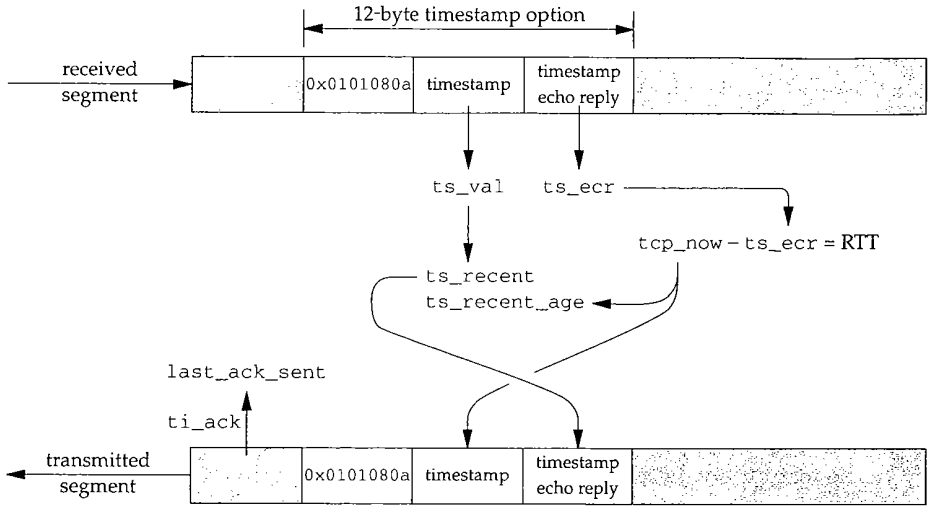


Figure 26.17 Summary of variables used with timestamp option.

The global variable `tcp_now` is the timestamp clock. It is initialized to 0 when the kernel is initialized and incremented by 1 every 500 ms (Figure 25.8). Three variables are maintained in the TCP control block for the timestamp option:

- `ts_recent` is a copy of the most-recent valid timestamp from the other end. (We describe shortly what makes a timestamp "valid.")
- `ts_recent_age` is the value of `tcp_now` when `ts_recent` was last copied from a received segment.
- `last_ack_sent` is the value of the acknowledgment field (`ti_ack`) the last time a segment was sent (Figure 26.32). This is normally equal to `rcv_nxt`, the next expected sequence number, unless ACKs are delayed.

The two variables `ts_val` and `ts_ecr` are local variables in the function `tcp_input` that contain the two values from the timestamp option.

- `ts_val` is the timestamp sent by the other end with its data.
- `ts_ecr` is the timestamp from the segment that is being acknowledged by the received segment.

In an outgoing segment, the first 4 bytes of the timestamp option are set to `0x0101080a`. This is the recommended value from Appendix A of RFC 1323. The 2 bytes of 1 are NOPs from Figure 26.16, followed by a *kind* of 8 and a *len* of 10, which identify the timestamp option. By placing two NOPs in front of the option, the two 32-bit timestamps in the option and the data that follows are aligned on 32-bit boundaries. Also, we show the received timestamp option in Figure 26.17 with the recommended 12-byte format (which Net/3 always generates), but the code that processes

received options (Figure 28.10) does not require this format. The 10-byte format shown in Figure 26.16, without two preceding NOPs, is handled fine on input (but see Exercise 28.4).

The RTT of a transmitted segment and its ACK is calculated as `tcp_now` minus `ts_ecr`. The units are 500-ms clock ticks, since that is the units of the Net/3 timestamps.

The presence of the timestamp option also allows TCP to perform PAWS: protection against wrapped sequence numbers. We describe this algorithm in Section 28.7. The variable `ts_recent_age` is used with PAWS.

`tcp_output` builds a timestamp option in an outgoing segment by copying `tcp_now` into the timestamp and `ts_recent` into the echo reply (Figure 26.24). This is done for every segment when the option is in use, unless the RST flag is set.

Which Timestamp to Echo, RFC 1323 Algorithm

The test for a valid timestamp determines whether the value in `ts_recent` is updated, and since this value is always sent as the timestamp echo reply, the test for validity determines which timestamp gets echoed back to the other end. RFC 1323 specified the following test:

```
ti_seq <= last_ack_sent < ti_seq + ti_len
```

which is implemented in C as shown in Figure 26.18.

```
if (ts_present && SEQ_LEQ(ti->ti_seq, tp->last_ack_sent) &&
    SEQ_LT(tp->last_ack_sent, ti->ti_seq + ti->ti_len)) {
      tp->ts_recent_age = tcp_now;
      tp->ts_recent = ts_val;
}
```

Figure 26.18 Typical code to determine if received timestamp is valid.

The variable `ts_present` is true if a timestamp option was received in the segment. We encounter this code twice in `tcp_input`: Figure 28.11 does the test in the header prediction code, and Figure 28.35 does the test in the normal input processing.

To see what this test is doing, Figure 26.19 shows show five different scenarios, corresponding to five different segments received on a connection. In each scenario `ti_len` is 3.

The left edge of the receive window begins with sequence number 4. In scenario 1 the segment contains completely duplicate data. The `SEQ_LEQ` test in Figure 28.11 is true, but the `SEQ_LT` test fails. For scenarios 2, 3, and 4, both the `SEQ_LEQ` and `SEQ_LT` tests are true because the left edge of the window is advanced by any one of these three segments, even though scenario 2 contains two duplicate bytes of data, and scenario 3 contains one duplicate byte of data. Scenario 5 fails the `SEQ_LEQ` test, because it doesn't advance the left edge of the window. This segment is one in the future that's not the next expected, implying that a previous segment was lost or reordered.

Unfortunately this test to determine whether to update `ts_recent` is flawed [Braden 1993]. Consider the following example.

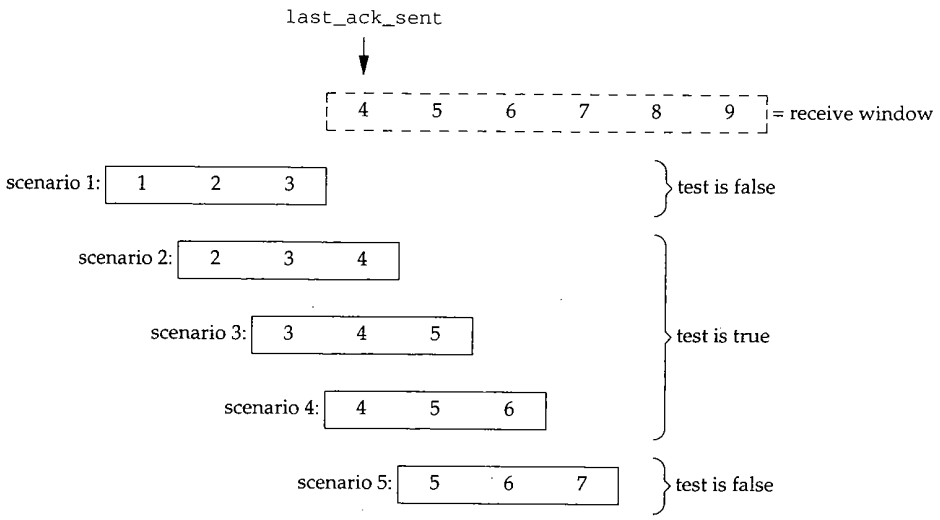


Figure 26.19 Example receive window and five different scenarios of received segment.

1. In Figure 26.19 a segment that we don't show arrives with bytes 1, 2, and 3. The timestamp in this segment is saved in `ts_recent` because `last_ack_sent` is 1. An ACK is sent with an acknowledgment field of 4, and `last_ack_sent` is set to 4 (the value of `rcv_nxt`). We have the receive window shown in Figure 26.19.
2. This ACK is lost.
3. The other end times out and retransmits the segment with bytes 1, 2, and 3. This segment arrives and is the one labeled "scenario 1" in Figure 26.19. Since the `SEQ_LT` test in Figure 26.18 fails, `ts_recent` is not updated with the value from the retransmitted segment.
4. A duplicate ACK is sent with an acknowledgment field of 4, but the timestamp echo reply is `ts_recent`, the value copied from the segment in step 1. But when the receiver calculates the RTT using this value, it will (incorrectly) take into account the original transmission, the lost ACK, the timeout, the retransmission, and the duplicate ACK.

For correct RTT estimation by the other end, the timestamp value from the retransmission should be returned in the duplicate ACK.

The tests in Figure 26.18 also fail to update `ts_recent` if the length of the received segment is 0, since the left edge of the window is not moved. This incorrect test can also lead to problems with long-lived (greater than 24 days, the PAWS limit described in Section 28.7), unidirectional connections (all the data flow is in one direction so the sender of the data always sends the same ACKs).

Which Timestamp to Echo, Corrected Algorithm

The algorithm we'll encounter in the Net/3 sources is from Figure 26.18. The correct algorithm given in [Braden 1993] replaces Figure 26.18 with the one in Figure 26.20.

```
if (ts_present && TSTMP_GEQ(ts_val, tp->ts_recent) &&
    SEQ_LEQ(ti->ti_seq, tp->last_ack_sent)) {
```

Figure 26.20 Correct code to determine if received timestamp is valid.

This doesn't test whether the left edge of the window moves or not, it just verifies that the new timestamp (`ts_val`) is greater than or equal to the previous timestamp (`ts_recent`), and that the starting sequence number of the received segment is not greater than the left edge of the window. Scenario 5 in Figure 26.19 would fail this new test since it is out of order.

The macro `TSTMP_GEQ` is identical to `SEQ_GEQ` in Figure 24.21. It is used with timestamps, since timestamps are 32-bit unsigned values that wrap around just like sequence numbers.

Timestamps and Delayed ACKs

It is constructive to see how timestamps and RTT calculations are affected by delayed ACKs. Recall from Figure 26.17 that the value saved by TCP in `ts_recent` becomes the echoed timestamp in segments that are sent, which are used by the other end in calculating its RTT. When ACKs are delayed, the delay time should be taken into account by the side that sees the delays, or else it might retransmit too quickly. In the example that follows we only consider the code in Figure 26.20, but the incorrect code in Figure 26.18 also handles delayed ACKs correctly.

Consider the receive sequence space in Figure 26.21 when the received segment contains bytes 4 and 5.

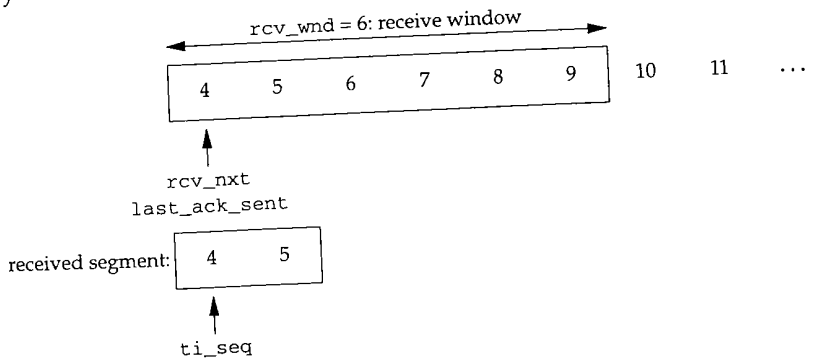


Figure 26.21 Receive sequence space when segment with bytes 4 and 5 arrives.

Since `ti_seq` is less than or equal to `last_ack_sent`, `ts_recent` is copied from the segment. `rcv_nxt` is also increased by 2.

Assume that the ACK for these 2 bytes is delayed, and before that delayed ACK is sent, the next in-order segment arrives. This is shown in Figure 26.22.

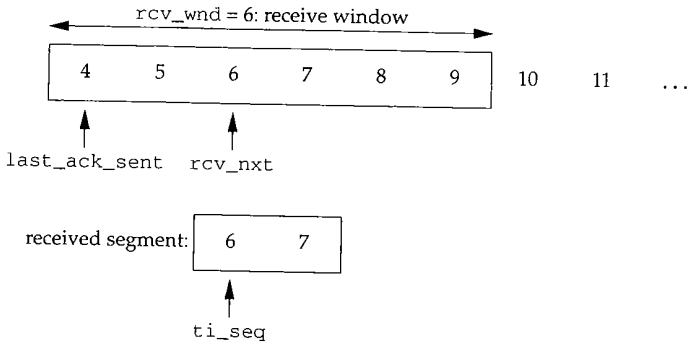


Figure 26.22 Receive sequence space when segment with bytes 6 and 7 arrives.

This time `ti_seq` is greater than `last_ack_sent`, so `ts_recent` is not updated. This is intentional. Assuming TCP now sends an ACK for sequence numbers 4–7, the other end's RTT will take into account the delayed ACK, since the echoed timestamp (Figure 26.24) is the one from the segment with sequence numbers 4 and 5. These figures also demonstrate that `rcv_nxt` equals `last_ack_sent` except when ACKs are delayed.

26.7 Send a Segment

The last half of `tcp_output` sends the segment—it fills in all the fields in the TCP header and passes the segment to IP for output.

Figure 26.23 shows the first part, which sends the MSS and window scale options with a SYN segment.

223–234 The TCP options are built in the array `opt`, and the integer `optlen` keeps a count of the number of bytes accumulated (since multiple options can be sent at once). If the SYN flag bit is set, `snd_nxt` is set to the initial send sequence number (`iss`). If TCP is performing an active open, `iss` is set by the `PRU_CONNECT` request when the TCP control block is created. If this is a passive open, `tcp_input` creates the TCP control block and sets `iss`. In both cases, `iss` is set from the global `tcp_iss`.

235 The flag `TF_NOOPT` is checked, but this flag is never enabled and there is no way to turn it on. Hence, the MSS option is always sent with a SYN segment.

> In the Net/1 version of `tcp_newtcpcb`, the comment "send options!" appeared on the line that initialized `t_flags` to 0. The `TF_NOOPT` flag is probably a historical artifact from a pre-Net/1 system that had problems interoperating with other hosts when it sent the MSS option, so the default was to not send the option.

tcp_output.c

```
222   send:
223     /*
224      * Before ESTABLISHED, force sending of initial options
225      * unless TCP set not to do any options.
226      * NOTE: we assume that the IP/TCP header plus TCP options
227      * always fit in a single mbuf, leaving room for a maximum
228      * link header, i.e.
229      *  max_linkhdr + sizeof (struct tcpiphdr) + optlen <= MHLEN
230      */
231     optlen = 0;
232     hdrlen = sizeof(struct tcpiphdr);
233     if (flags & TH_SYN) {
234         tp->snd_nxt = tp->iss;
235         if ((tp->t_flags & TF_NOOPT) == 0) {
236             u_short mss;

237             opt[0] = TCPOPT_MAXSEG;
238             opt[1] = 4;
239             mss = htons((u_short) tcp_mss(tp, 0));
240             bcopy((caddr_t) & mss, (caddr_t) (opt + 2), sizeof(mss));
241             optlen = 4;

242             if ((tp->t_flags & TF_REQ_SCALE) &&
243                 ((flags & TH_ACK) == 0 ||
244                  (tp->t_flags & TF_RCVD_SCALE))) {
245                 *((u_long *) (opt + optlen)) = htonl(TCPOPT_NOP << 24 |
246                                                      TCPOPT_WINDOW << 16 |
247                                                      TCPOLEN_WINDOW << 8 |
248                                                      tp->request_r_scale);
249                 optlen += 4;
250             }
251         }
252     }
```

tcp_output.c

Figure 26.23 `tcp_output` function: send options with first SYN segment.

Build MSS option

236–241 `opt[0]` is set to 2 (`TCPOPT_MAXSEG`) and `opt[1]` is set to 4, the length of the MSS option in bytes. The function `tcp_mss` calculates the MSS to announce to the other end; we cover this function in Section 27.5. The 16-bit MSS is stored in `opt[2]` and `opt[3]` by `bcopy` (Exercise 26.5). Notice that Net/3 always sends an MSS announcement with the SYN for a connection.

Should window scale option be sent?

242–244 If TCP is to request the window scale option, this option is sent only if this is an active open (`TH_ACK` is not set) or if this is a passive open and the window scale option was received in the SYN from the other end. Recall that `t_flags` was set to `TF_REQ_SCALE|TF_REQ_TSTMP` when the TCP control block was created in Figure 25.21, if the global variable `tcp_do_rfc1323` was nonzero (its default value).

Build window scale option

245–249 Since the window scale option occupies 3 bytes (Figure 26.16), a 1-byte NOP is stored before the option, forcing the option length to be 4 bytes. This causes the data in the segment that follows the options to be aligned on a 4-byte boundary. If this is an active open, `request_r_scale` is calculated by the `PRU_CONNECT` request. If this is a passive open, the window scale factor is calculated by `tcp_input` when the SYN is received.

RFC 1323 specifies that if TCP is prepared to scale windows it should send this option even if its own shift count is 0. This is because the option serves two purposes: to notify the other end that it supports the option, and to announce its shift count. Even though TCP may calculate its own shift count as 0, the other end might want to use a different value.

The next part of `tcp_output` is shown in Figure 26.24. It finishes building the options in the outgoing segment.

tcp_output.c

```
253     /*
254      * Send a timestamp and echo-reply if this is a SYN and our side
255      * wants to use timestamps (TF_REQ_TSTMP is set) or both our side
256      * and our peer have sent timestamps in our SYN's.
257      */
258     if ((tp->t_flags & (TF_REQ_TSTMP | TF_NOOPT)) == TF_REQ_TSTMP &&
259         (flags & TH_RST) == 0 &&
260         ((flags & (TH_SYN | TH_ACK)) == TH_SYN ||
261          (tp->t_flags & TF_RCVD_TSTMP))) {
262         u_long *lp = (u_long *) (opt + optlen);

263         /* Form timestamp option as shown in appendix A of RFC 1323. */
264         *lp++ = htonl(TCPOPT_TSTAMP_HDR);
265         *lp++ = htonl(tcp_now);
266         *lp = htonl(tp->ts_recent);
267         optlen += TCPOLEN_TSTAMP_APPA;
268     }
269     hdrlen += optlen;

270     /*
271      * Adjust data length if insertion of options will
272      * bump the packet length beyond the t_maxseg length.
273      */
274     if (len > tp->t_maxseg - optlen) {
275         len = tp->t_maxseg - optlen;
276         sendalot = 1;
277     }
```

tcp_output.c

Figure 26.24 `tcp_output` function: finish sending options.

Should timestamp option be sent?

253–261 If the following three conditions are all true, a timestamp option is sent: (1) TCP is configured to request the timestamp option, (2) the segment being formed does not contain the RST flag, and (3) either this is an active open (i.e., `flags` specifies the SYN flag

but not the ACK flag) or TCP has received a timestamp from the other end (TF_RCVD_TSTMP). Unlike the MSS and window scale options, a timestamp option can be sent with every segment once both ends agree to use the option.

Build timestamp option

263–267 The timestamp option (Section 26.6) consists of 12 bytes (TCPOLEN_TSTAMP_APPA). The first 4 bytes are 0x0101080a (the constant TCPOPT_TSTAMP_HDR), as described with Figure 26.17. The timestamp value is taken from tcp_now (the number of 500-ms clock ticks since the system was initialized), and the timestamp echo reply is taken from ts_recent, which is set by tcp_input.

Check if options have overflowed segment

270–277 The size of the TCP header is incremented by the number of option bytes (optlen). If the amount of data to send (len) exceeds the MSS minus the size of the options (optlen), the data length is decreased accordingly and the sendalot flag is set, to force another loop through this function after this segment is sent (Figure 26.1).

The MSS and window scale options only appear in SYN segments, which Net/3 always sends without data, so this adjustment of the data length doesn't apply. When the timestamp option is in use, however, it appears in all segments. This reduces the amount of data in each full-sized data segment from the announced MSS to the announced MSS minus 12 bytes.

The next part of tcp_output, shown in Figure 26.25, updates some statistics and allocates an mbuf for the IP and TCP headers. This code is executed when the segment being output contains some data (len is greater than 0).

Update statistics

284–292 If t_force is nonzero and TCP is sending a single byte of data, this is a window probe. If snd_nxt is less than snd_max, this is a retransmission. Otherwise, this is normal data transmission.

Allocate an mbuf for IP and TCP headers

293–297 An mbuf with a packet header is allocated by MGETHDR. This is for the IP and TCP headers, and possibly the data (if there's room). Although tcp_output is often called as part of a system call (e.g., write) it is also called at the software interrupt level by tcp_input, and as part of the timer processing. Therefore M_DONTWAIT is specified. If an error is returned, a jump is made to the label out. This label is near the end of the function, in Figure 26.32.

Copy data into mbuf

298–308 If the amount of data is less than 44 bytes (100 – 40 – 16, assuming no TCP options), the data is copied directly from the socket send buffer into the new packet header mbuf by m_copydata. Otherwise m_copy creates a new mbuf chain with the data from the socket send buffer and this chain is linked to the new packet header mbuf. Recall our description of m_copy in Section 2.9, where we showed that if the data is in a cluster, m_copy just references that cluster and doesn't make a copy of the data. 309–316

tcp_output.c

```
278     /*
279      * Grab a header mbuf, attaching a copy of data to
280      * be transmitted, and initialize the header from
281      * the template for sends on this connection.
282      */
283     if (len) {
284         if (tp->t_force && len == 1)
285             tcpstat.tcps_sndprobe++;
286         else if (SEQ_LT(tp->snd_nxt, tp->snd_max)) {
287             tcpstat.tcps_sndrexmitpack++;
288             tcpstat.tcps_sndrexmitbyte += len;
289         } else {
290             tcpstat.tcps_sndpack++;
291             tcpstat.tcps_sndbyte += len;
292         }
293         MGETHDR(m, M_DONTWAIT, MT_HEADER);
294         if (m == NULL) {
295             error = ENOBUFS;
296             goto out;
297         }
298         m->m_data += max_linkhdr;
299         m->m_len = hdrlen;
300         if (len <= MHLEN - hdrlen - max_linkhdr) {
301             m_copydata(so->so_snd.sb_mb, off, (int) len,
302                        mtod(m, caddr_t) + hdrlen);
303             m->m_len += len;
304         } else {
305             m->m_next = m_copy(so->so_snd.sb_mb, off, (int) len);
306             if (m->m_next == 0)
307                 len = 0;
308         }
309         /*
310          * If we're sending everything we've got, set PUSH.
311          * (This will keep happy those implementations that
312          * give data to the user only when a buffer fills or
313          * a PUSH comes in.)
314          */
315         if (off + len == so->so_snd.sb_cc)
316             flags |= TH_PUSH;
```

tcp_output.c

Figure 26.25 `tcp_output` function: update statistics, allocate mbuf for IP and TCP headers.

Set PSH flag

309–316 If TCP is sending everything it has from the send buffer, the PSH flag is set. As the comment indicates, this is intended for receiving systems that only pass received data to an application when the PSH flag is received or when a buffer fills. We'll see in `tcp_input` that Net/3 never holds data in a socket receive buffer waiting for a received PSH flag.

The next part of tcp_output, shown in Figure 26.26, starts with the code that is executed when len equals 0: there is no data in the segment TCP is sending.

tcp_output.c

```
317     } else {                        /* len == 0 */
318         if (tp->t_flags & TF_ACKNOW)
319             tcpstat.tcps_sndacks++;
320         else if (flags & (TH_SYN | TH_FIN | TH_RST))
321             tcpstat.tcps_sndctrl++;
322         else if (SEQ_GT(tp->snd_up, tp->snd_una))
323             tcpstat.tcps_sndurg++;
324         else
325             tcpstat.tcps_sndwinup++;

326         MGETHDR(m, M_DONTWAIT, MT_HEADER);
327         if (m == NULL) {
328             error = ENOBUFS;
329             goto out;
330         }
331         m->m_data += max_linkhdr;
332         m->m_len = hdrlen;
333     }
334     m->m_pkthdr.rcvif = (struct ifnet *) 0;
335     ti = mtod(m, struct tcpiphdr *);
336     if (tp->t_template == 0)
337         panic("tcp_output");
338     bcopy((caddr_t) tp->t_template, (caddr_t) ti, sizeof(struct tcpiphdr));
```

tcp_output.c

Figure 26.26 tcp_output function: update statistics and allocate mbuf for IP and TCP headers.

Update statistics

318–325 Various statistics are updated: TF_ACKNOW and a length of 0 means this is an ACK-only segment. If any one of the flags SYN, FIN, or RST is set, this is a control segment. If the urgent pointer exceeds snd_una, the segment is being sent to notify the other end of the urgent pointer. If none of these conditions are true, this segment is a window update.

Get mbuf for IP and TCP headers

326–335 An mbuf with a packet header is allocated to contain the IP and TCP headers.

Copy IP and TCP header templates into mbuf

336–338 The template of the IP and TCP headers is copied from t_template into the mbuf by bcopy. This template was created by tcp_template.

Figure 26.27 shows the next part of tcp_output, which fills in some remaining fields in the TCP header.

Decrement snd_nxt if FIN is being retransmitted

339–346 If TCP has already transmitted the FIN, the send sequence space appears as shown in Figure 26.28.

tcp_output.c

```
339     /*
340      * Fill in fields, remembering maximum advertised
341      * window for use in delaying messages about window sizes.
342      * If resending a FIN, be sure not to use a new sequence number.
343      */
344     if (flags & TH_FIN && tp->t_flags & TF_SENTFIN &&
345         tp->snd_nxt == tp->snd_max)
346         tp->snd_nxt--;
347     /*
348      * If we are doing retransmissions, then snd_nxt will
349      * not reflect the first unsent octet.  For ACK only
350      * packets, we do not want the sequence number of the
351      * retransmitted packet, we want the sequence number
352      * of the next unsent octet.  So, if there is no data
353      * (and no SYN or FIN), use snd_max instead of snd_nxt
354      * when filling in ti_seq.  But if we are in persist
355      * state, snd_max might reflect one byte beyond the
356      * right edge of the window, so use snd_nxt in that
357      * case, since we know we aren't doing a retransmission.
358      * (retransmit and persist are mutually exclusive...)
359      */
360     if (len || (flags & (TH_SYN | TH_FIN)) || tp->t_timer[TCPT_PERSIST])
361         ti->ti_seq = htonl(tp->snd_nxt);
362     else
363         ti->ti_seq = htonl(tp->snd_max);
364     ti->ti_ack = htonl(tp->rcv_nxt);
365     if (optlen) {
366         bcopy((caddr_t) opt, (caddr_t) (ti + 1), optlen);
367         ti->ti_off = (sizeof(struct tcphdr) + optlen) >> 2;
368     }
369     ti->ti_flags = flags;
```

tcp_output.c

Figure 26.27 `tcp_output` function: set `ti_seq`, `ti_ack`, and `ti_flags`.

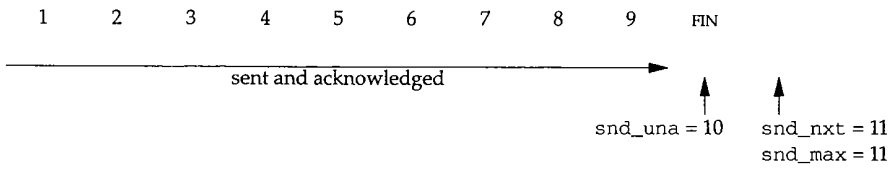


Figure 26.28 Send sequence space after FIN has been transmitted.

Therefore, if the FIN flag is set, and if the `TF_SENTFIN` flag is set, and if `snd_nxt` equals `snd_max`, TCP knows the FIN is being retransmitted. We'll see shortly (Figure 26.31) that when a FIN is sent, `snd_nxt` is incremented 1 one (since the FIN occupies a sequence number), so this piece of code decrements `snd_nxt` by 1.

Set sequence number field of segment

347–363 The sequence number field of the segment is normally set to `snd_nxt`, but is set to `snd_max` if (1) there is no data to send (`len` equals 0), (2) neither the SYN flag nor the FIN flag is set, and (3) the persist timer is not set.

Set acknowledgment field of segment

364 The acknowledgment field of the segment is always set to `rcv_nxt`, the next expected receive sequence number.

Set header length if options present

365–368 If TCP options are present (`optlen` is greater than 0), the options are copied into the TCP header and the 4-bit header length in the TCP header (`th_off` in Figure 24.10) is set to the fixed size of the TCP header (20 bytes) plus the length of the options, divided by 4. This field is the number of 32-bit words in the TCP header, including options.

369 The flags field in the TCP header is set from the variable `flags`.

The next part of code, shown in Figure 26.29, fills in more fields in the TCP header and calculates the TCP checksum.

Don't advertise less than one full-sized segment

370–375 Avoidance of the silly window syndrome is performed, this time in calculating the window size that is advertised to the other end (`ti_win`). Recall that `win` was set at the end of Figure 26.3 to the amount of space in the socket's receive buffer. If `win` is less than one-fourth of the receive buffer size (`so_rcv.sb_hiwat`) and less than one full-sized segment, the advertised window will be 0. This is subject to the later test that prevents the window from shrinking. In other words, when the amount of available space reaches either one-fourth of the receive buffer size or one full-sized segment, the available space will be advertised.

Observe upper limit for advertised window on this connection

376–377 If `win` is larger than the maximum value for this connection, reduce it to its maximum value.

Do not shrink window

378–379 Recall from Figure 26.10 that `rcv_adv` minus `rcv_nxt` is the amount of space still available to the sender that was previously advertised. If `win` is less than this value, `win` is set to this value, because we must not shrink the window. This can happen when the available space is less than one full-sized segment (hence `win` was set to 0 at the beginning of this figure), but there is room in the receive buffer for some data. Figure 22.3 of Volume 1 shows an example of this scenario.

Set urgent offset

381–383 If the urgent pointer (`snd_up`) is greater than `snd_nxt`, TCP is in urgent mode. The urgent offset in the TCP header is set to the 16-bit offset of the urgent pointer from the starting sequence number of the segment, and the URG flag bit is set. TCP sends the urgent offset and the URG flag regardless of whether the referenced byte of urgent data is contained in this segment or not.

tcp_output.c

```
370     /*
371      * Calculate receive window.  Don't shrink window,
372      * but avoid silly window syndrome.
373      */
374     if (win < (long) (so->so_rcv.sb_hiwat / 4) && win < (long) tp->t_maxseg)
375         win = 0;
376     if (win > (long) TCP_MAXWIN << tp->rcv_scale)
377         win = (long) TCP_MAXWIN << tp->rcv_scale;
378     if (win < (long) (tp->rcv_adv - tp->rcv_nxt))
379         win = (long) (tp->rcv_adv - tp->rcv_nxt);
380     ti->ti_win = htons((u_short) (win >> tp->rcv_scale));
381     if (SEQ_GT(tp->snd_up, tp->snd_nxt)) {
382         ti->ti_urp = htons((u_short) (tp->snd_up - tp->snd_nxt));
383         ti->ti_flags |= TH_URG;
384     } else
385         /*
386          * If no urgent pointer to send, then we pull
387          * the urgent pointer to the left edge of the send window
388          * so that it doesn't drift into the send window on sequence
389          * number wraparound.
390          */
391         tp->snd_up = tp->snd_una;   /* drag it along */
392     /*
393      * Put TCP length in extended header, and then
394      * checksum extended header and data.
395      */
396     if (len + optlen)
397         ti->ti_len = htons((u_short) (sizeof(struct tcphdr) +
398                                       optlen + len));
399     ti->ti_sum = in_cksum(m, (int) (hdrlen + len));
```

tcp_output.c

Figure 26.29 `tcp_output` function: fill in more TCP header fields and calculate checksum.

Figure 26.30 shows an example of how the urgent offset is calculated, assuming the process executes

```
send(fd, buf, 3, MSG_OOB);
```

and the send buffer is empty when this call to `send` takes place. This shows that Berkeley-derived systems consider the urgent pointer to point to the first byte of data *after* the out-of-band byte. Recall our discussion after Figure 24.10 where we distinguished between the 32-bit *urgent pointer* in the data stream (`snd_up`), and the 16-bit *urgent offset* in the TCP header (`ti_urp`).

> There is a subtle bug here. The bug occurs when the send buffer is larger than 65535, regardless of whether the window scale option is in use or not. If the send buffer is greater than 65535 and is nearly full, and the process sends out-of-band data, the offset of the urgent pointer from `snd_nxt` can exceed 65535. But the urgent pointer is a 16-bit unsigned value, and if the calculated value exceeds 65535, the 16 high-order bits are discarded, delivering a bogus urgent pointer to the other end. See Exercise 26.6 for a solution.

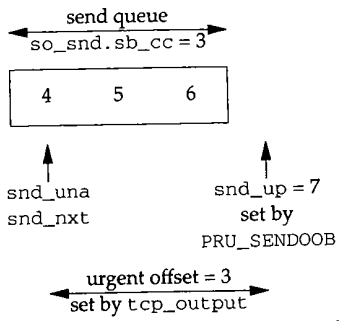


Figure 26.30 Example of urgent pointer and urgent offset calculation.

384–391 If TCP is not in urgent mode, the urgent pointer is moved to the left edge of the window (`snd_una`).

392–399 The TCP length is stored in the pseudo-header and the TCP checksum is calculated. All the fields in the TCP header have been filled in, and when the IP and TCP header template were copied from `t_template` (Figure 26.26), the fields in the IP header that are used as the pseudo-header were initialized (as shown in Figure 23.19 for the UDP checksum calculation).

The next part of `tcp_output`, shown in Figure 26.31, updates the sequence number if the SYN or FIN flags are set and initializes the retransmission timer.

Remember starting sequence number

400–405 If TCP is not in the persist state, the starting sequence number is saved in `startseq`. This is used later in Figure 26.31 if the segment is timed.

Increment `snd_nxt`

406–417 Since both the SYN and FIN flags take a sequence number, `snd_nxt` is incremented if either is set. TCP also remembers that the FIN has been sent, by setting the flag `TF_SENTFIN`. `snd_nxt` is then incremented by the number of bytes of data (`len`), which can be 0.

Update `snd_max`

418–419 If the new value of `snd_nxt` is larger than `snd_max`, this is not a retransmission. The new value of `snd_max` is stored.

420–428 If a segment is not currently being timed for this connection (`t_rtt` equals 0), the timer is started (`t_rtt` is set to 1) and the starting sequence number of the segment being timed is saved in `t_rtseq`. This sequence number is used by `tcp_input` to determine when the segment being timed is acknowledged, to update the RTT estimators. The sample code we discussed in Section 25.10 looked like

```
if (tp->t_rtt && SEQ_GT(ti->ti_ack, tp->t_rtseq))
    tcp_xmit_timer(tp, tp->t_rtt);
```

tcp_output.c

```
400     /*
401      * In transmit state, time the transmission and arrange for
402      * the retransmit.  In persist state, just set snd_max.
403      */
404     if (tp->t_force == 0 || tp->t_timer[TCPT_PERSIST] == 0) {
405         tcp_seq startseq = tp->snd_nxt;

406         /*
407          * Advance snd_nxt over sequence space of this segment.
408          */
409         if (flags & (TH_SYN | TH_FIN)) {
410             if (flags & TH_SYN)
411                 tp->snd_nxt++;
412             if (flags & TH_FIN) {
413                 tp->snd_nxt++;
414                 tp->t_flags |= TF_SENTFIN;
415             }
416         }
417         tp->snd_nxt += len;
418         if (SEQ_GT(tp->snd_nxt, tp->snd_max)) {
419             tp->snd_max = tp->snd_nxt;
420             /*
421              * Time this transmission if not a retransmission and
422              * not currently timing anything.
423              */
424             if (tp->t_rtt == 0) {
425                 tp->t_rtt = 1;
426                 tp->t_rtseq = startseq;
427                 tcpstat.tcps_segstimed++;
428             }
429         }
430         /*
431          * Set retransmit timer if not currently set,
432          * and not doing an ack or a keepalive probe.
433          * Initial value for retransmit timer is smoothed
434          * round-trip time + 2 * round-trip time variance.
435          * Initialize  counter which is used for backoff
436          * of retransmit time.
437          */
438         if (tp->t_timer[TCPT_REXMT] == 0 &&
439             tp->snd_nxt != tp->snd_una) {
440             tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;
441             if (tp->t_timer[TCPT_PERSIST]) {
442                 tp->t_timer[TCPT_PERSIST] = 0;
443                 tp->t_rxtshift = 0;
444             }
445         }
446     } else if (SEQ_GT(tp->snd_nxt + len, tp->snd_max))
447         tp->snd_max = tp->snd_nxt + len;
```

tcp_output.c

Figure 26.31 `tcp_output` function: update sequence number, initialize retransmit timer.

Set retransmission timer

430–440 If the retransmission timer is not currently set, and if this segment contains data, the retransmission timer is set to `t_rxtcur`. Recall that `t_rxtcur` is set by `tcp_xmit_timer`, when an RTT measurement is made. This is an ACK-only segment if `snd_nxt` equals `snd_una` (since `len` was added to `snd_nxt` earlier in this figure), and the retransmission timer is set only for segments containing data.

441–444 If the persist timer is enabled, it is disabled. Either the retransmission timer or the persist timer can be enabled at any time for a given connection, but not both.

Persist state

446–447 The connection is in the persist state since `t_force` is nonzero and the persist timer is enabled. (This `else` clause is associated with the `if` at the beginning of the figure.) `snd_max` is updated, if necessary. In the persist state, `len` will be one.

The final part of `tcp_output`, shown in Figure 26.32 completes the formation of the outgoing segment and calls `ip_output` to send the datagram.

Add trace record for socket debugging

448–452 If the `SO_DEBUG` socket option is enabled, `tcp_trace` adds a record to TCP's circular trace buffer. We describe this function in Section 27.10.

Set IP length, TTL, and TOS

453–462 The final three fields in the IP header that must be set by the transport layer are stored: IP length, TTL, and TOS. These three fields are marked with an asterisk at the bottom of Figure 23.19.

> The comments XXX are because the latter two fields normally remain constant for a connection and should be stored in the header template, instead of being assigned explicitly each time a segment is sent. But these two fields cannot be stored in the IP header until after the TCP checksum is calculated.

Pass datagram to IP

463–464 `ip_output` sends the datagram containing the TCP segment. The socket options are logically ANDed with `SO_DONTROUTE`, which means that the only socket option passed to `ip_output` is `SO_DONTROUTE`. The only other socket option examined by `ip_output` is `SO_BROADCAST`, so this logical AND turns off the `SO_BROADCAST` bit, if set. This means that a process cannot issue a `connect` to a broadcast address, even if it sets the `SO_BROADCAST` socket option.

467–470 The error `ENOBUFS` is returned if the interface queue is full or if IP needs to obtain an mbuf and can't. The function `tcp_quench` puts the connection into slow start, by setting the congestion window to one full-sized segment. Notice that `tcp_output` still returns 0 (OK) in this case, instead of the error, even though the datagram was discarded. This differs from `udp_output` (Figure 23.20), which returned the error. The difference is that UDP is unreliable, so the `ENOBUFS` error return is the only indication to the process that the datagram was discarded. TCP, however, will time out (if the segment contains data) and retransmit the datagram, and it is hoped that there will be space on the interface output queue or more available mbufs. If the TCP segment

tcp_output.c

```
448     /*
449      * Trace.
450      */
451     if (so->so_options & SO_DEBUG)
452         tcp_trace(TA_OUTPUT, tp->t_state, tp, ti, 0);

453     /*
454      * Fill in IP length and desired time to live and
455      * send to IP level.  There should be a better way
456      * to handle ttl and tos; we could keep them in
457      * the template, but need a way to checksum without them.
458      */
459     m->m_pkthdr.len = hdrlen + len;
460     ((struct ip *) ti)->ip_len = m->m_pkthdr.len;
461     ((struct ip *) ti)->ip_ttl = tp->t_inpcb->inp_ip.ip_ttl;     /* XXX */
462     ((struct ip *) ti)->ip_tos = tp->t_inpcb->inp_ip.ip_tos;     /* XXX */
463     error = ip_output(m, tp->t_inpcb->inp_options, &tp->t_inpcb->inp_route,
464                       so->so_options & SO_DONTROUTE, 0);
465     if (error) {
466       out:
467         if (error == ENOBUFS) {
468             tcp_quench(tp->t_inpcb, 0);
469             return (0);
470         }
471         if ((error == EHOSTUNREACH || error == ENETDOWN)
472             && TCPS_HAVERCVDSYN(tp->t_state)) {
473             tp->t_softerror = error;
474             return (0);
475         }
476         return (error);
477     }
478     tcpstat.tcps_sndtotal++;

479     /*
480      * Data sent (as far as we can tell).
481      * If this advertises a larger window than any other segment,
482      * then remember the size of the advertised window.
483      * Any pending ACK has now been sent.
484      */
485     if (win > 0 && SEQ_GT(tp->rcv_nxt + win, tp->rcv_adv))
486         tp->rcv_adv = tp->rcv_nxt + win;
487     tp->last_ack_sent = tp->rcv_nxt;
488     tp->t_flags &= ~(TF_ACKNOW | TF_DELACK);

489     if (sendalot)
490         goto again;
491     return (0);
492 }
```

tcp_output.c

Figure 26.32 `tcp_output` function: call `ip_output` to send segment.

doesn't contain data, the other end will time out when the ACK isn't received and will retransmit the data whose ACK was discarded.

471–475 If a route can't be located for the destination, and if the connection has received a SYN, the error is recorded as a soft error for the connection.

When `tcp_output` is called by `tcp_usrreq` as part of a system call by a process (Chapter 30, the `PRU_CONNECT`, `PRU_SEND`, `PRU_SENDOOB`, and `PRU_SHUTDOWN` requests), the process receives the return value from `tcp_output`. Other functions that call `tcp_output`, such as `tcp_input` and the fast and slow timeout functions, ignore the return value (because these functions don't return an error to a process).

Update `rcv_adv` and `last_ack_sent`

479–486 If the highest sequence number advertised in this segment (`rcv_nxt` plus `win`) is larger than `rcv_adv`, the new value is saved. Recall that `rcv_adv` was used in Figure 26.9 to determine how much the window had opened since the last segment that was sent, and in Figure 26.29 to make certain TCP was not shrinking the window.

487 The value of the acknowledgment field in the segment is saved in `last_ack_sent`. This variable is used by `tcp_input` with the timestamp option (Section 26.6).

488 Any pending ACK has been sent, so the `TF_ACKNOW` and `TF_DELACK` flags are cleared.

More data to send?

489–490 If the `sendalot` flag is set, a jump is made back to the label `again` (Figure 26.1). This occurs if the send buffer contains more than one full-sized segment that can be sent (Figure 26.3), or if a full-sized segment was being sent and TCP options were included that reduced the amount of data in the segment (Figure 26.24).

26.8 `tcp_template` Function

The function `tcp_newtcpcb` (from the previous chapter) is called when the socket is created, to allocate and partially initialize the TCP control block. When the first segment is sent or received on the socket (an active open is performed, the `PRU_CONNECT` request, or a SYN arrives for a listening socket), `tcp_template` creates a template of the IP and TCP headers for the connection. This minimizes the amount of work required by `tcp_output` when a segment is sent on the connection.

Figure 26.33 shows the `tcp_template` function.

Allocate mbuf

59–72 The template of the IP and TCP headers is formed in an mbuf, and a pointer to the mbuf is stored in the `t_template` member of the TCP control block. Since this function can be called at the software interrupt level, from `tcp_input`, the `M_DONTWAIT` flag is specified.

Initialize header fields

73–88 All the fields in the IP and TCP headers are set to 0 except as follows: `ti_pr` is set to the IP protocol value for TCP (6); `ti_len` is set to 20, the default length of the TCP

tcp_subr.c

```
59 struct tcpiphdr *
60 tcp_template(tp)
61 struct tcpcb *tp;
62 {
63     struct inpcb *inp = tp->t_inpcb;
64     struct mbuf *m;
65     struct tcpiphdr *n;

66     if ((n = tp->t_template) == 0) {
67         m = m_get(M_DONTWAIT, MT_HEADER);
68         if (m == NULL)
69             return (0);
70         m->m_len = sizeof(struct tcpiphdr);
71         n = mtod(m, struct tcpiphdr *);
72     }
73     n->ti_next = n->ti_prev = 0;
74     n->ti_x1 = 0;
75     n->ti_pr = IPPROTO_TCP;
76     n->ti_len = htons(sizeof(struct tcpiphdr) - sizeof(struct ip));
77     n->ti_src = inp->inp_laddr;
78     n->ti_dst = inp->inp_faddr;
79     n->ti_sport = inp->inp_lport;
80     n->ti_dport = inp->inp_fport;
81     n->ti_seq = 0;
82     n->ti_ack = 0;
83     n->ti_x2 = 0;
84     n->ti_off = 5;                  /* 5 32-bit words = 20 bytes */
85     n->ti_flags = 0;
86     n->ti_win = 0;
87     n->ti_sum = 0;
88     n->ti_urp = 0;
89     return (n);
90 }
```

tcp_subr.c

Figure 26.33 `tcp_template` function: create template of IP and TCP headers.

header; and `ti_off` is set to 5, the number of 32-bit words in the 20-byte TCP header. Also the source and destination IP addresses and TCP port numbers are copied from the Internet PCB into the TCP header template.

Pseudo-header for TCP checksum computation

73–88 The initialization of many of the fields in the combined IP and TCP header simplifies the computation of the TCP checksum, using the same pseudo-header technique as discussed for UDP in Section 23.6. Examining the `udpiphdr` structure in Figure 23.19 shows why `tcp_template` initializes fields such as `ti_next` and `ti_prev` to 0.

26.9 `tcp_respond` Function

The function `tcp_respond` is a special-purpose function that also calls `ip_output` to send IP datagrams. `tcp_respond` is called in two cases:

1. by `tcp_input` to generate an RST segment, with or without an ACK, and
2. by `tcp_timers` to send a keepalive probe.

Instead of going through all the logic of `tcp_output` for these two cases, the special-purpose function `tcp_respond` is called. We also note that the function `tcp_drop` that we cover in the next chapter also generates RST segments by calling `tcp_output`. Not all RST segments are generated by `tcp_respond`.

Figure 26.34 shows the first half of `tcp_respond`.

tcp_subr.c

```
104 void
105 tcp_respond(tp, ti, m, ack, seq, flags)
106 struct tcpcb *tp;
107 struct tcpiphdr *ti;
108 struct mbuf *m;
109 tcp_seq ack, seq;
110 int     flags;
111 {
112     int     tlen;
113     int     win = 0;
114     struct route *ro = 0;

115     if (tp) {
116         win = sbspace(&tp->t_inpcb->inp_socket->so_rcv);
117         ro = &tp->t_inpcb->inp_route;
118     }
119     if (m == 0) {                   /* generate keepalive probe */
120         m = m_gethdr(M_DONTWAIT, MT_HEADER);
121         if (m == NULL)
122             return;
123         tlen = 0;                   /* no data is sent */
124         m->m_data += max_linkhdr;
125         *mtod(m, struct tcpiphdr *) = *ti;
126         ti = mtod(m, struct tcpiphdr *);
127         flags = TH_ACK;
128     } else {                        /* generate RST segment */
129         m_freem(m->m_next);
130         m->m_next = 0;
131         m->m_data = (caddr_t) ti;
132         m->m_len = sizeof(struct tcpiphdr);
133         tlen = 0;
134 #define xchg(a,b,type) { type t; t=a; a=b; b=t; }
135         xchg(ti->ti_dst.s_addr, ti->ti_src.s_addr, u_long);
136         xchg(ti->ti_dport, ti->ti_sport, u_short);
137 #undef xchg
138     }
```

tcp_subr.c

Figure 26.34 `tcp_respond` function: first half.

104–110 Figure 26.35 shows the different arguments to `tcp_respond` for the three cases in which it is called.

	Arguments					
	tp	ti	m	ack	seq	flags
generate RST without ACK	tp	ti	m	0	ti_ack	TH_RST
generate RST with ACK	tp	ti	m	ti_seq + ti_len	0	TH_RST | TH_ACK
generate keepalive	tp	t_template	NULL	rcv_nxt	snd_una	0

Figure 26.35 Arguments to tcp_respond.

tp is a pointer to the TCP control block (possibly a null pointer); ti is a pointer to an IP/TCP header template; m is a pointer to the mbuf containing the segment causing the RST to be generated; and the last three arguments are the acknowledgment field, sequence number field, and flags field of the segment being generated.

113–118 It is possible for tcp_input to generate an RST when a segment is received that does not have an associated TCP control block. This happens, for example, when a segment is received that doesn't reference an existing connection (e.g., a SYN for a port without an associated listening server). In this case tp is null and the initial values for win and ro are used. If tp is not null, the amount of space in the receive buffer will be sent as the advertised window, and the pointer to the cached route is saved in ro for the call to ip_output.

Send keepalive probe when keepalive timer expires

119–127 The argument m is a pointer to the mbuf chain for the received segment. But a keepalive probe is sent in response to the keepalive timer expiring, not in response to a received TCP segment. Therefore m is null and m_gethdr allocates a packet header mbuf to contain the IP and TCP headers. tlen, the length of the TCP data, is set to 0, since the keepalive probe doesn't contain any data.

> Some older implementations based on 4.2BSD do not respond to these keepalive probes unless the segment contains data. Net/3 can be configured to send 1 garbage byte of data in the probe to elicit the response by defining the name TCP_COMPAT_42 when the kernel is compiled. This assigns 1, instead of 0, to tlen. The garbage byte causes no harm, because it is not the expected byte (it is a byte that the receiver has previously received and acknowledged), so it is thrown away by the receiver.

The assignment of *ti copies the TCP header template structure pointed to by ti into the data portion of the mbuf. The pointer ti is then set to point to the header template in the mbuf.

Send RST segment in response to received segment

128–138 An RST segment is being sent by tcp_input in response to a received segment. The mbuf containing the input segment is reused for the response. All the mbufs on the chain are released by m_free except the first mbuf (the packet header), since the segment generated by tcp_respond consists of only an IP header and a TCP header. The source and destination IP address and port numbers are swapped in the IP and TCP header.

Figure 26.36 shows the final half of `tcp_respond`.

tcp_subr.c

```
139     ti->ti_len = htons((u_short) (sizeof(struct tcphdr) + tlen));
140     tlen += sizeof(struct tcpiphdr);
141     m->m_len = tlen;
142     m->m_pkthdr.len = tlen;
143     m->m_pkthdr.rcvif = (struct ifnet *) 0;
144     ti->ti_next = ti->ti_prev = 0;
145     ti->ti_x1 = 0;
146     ti->ti_seq = htonl(seq);
147     ti->ti_ack = htonl(ack);
148     ti->ti_x2 = 0;
149     ti->ti_off = sizeof(struct tcphdr) >> 2;
150     ti->ti_flags = flags;
151     if (tp)
152         ti->ti_win = htons((u_short) (win >> tp->rcv_scale));
153     else
154         ti->ti_win = htons((u_short) win);
155     ti->ti_urp = 0;
156     ti->ti_sum = 0;
157     ti->ti_sum = in_cksum(m, tlen);
158     ((struct ip *) ti)->ip_len = tlen;
159     ((struct ip *) ti)->ip_ttl = ip_defttl;
160     (void) ip_output(m, NULL, ro, 0, NULL);
161 }
```

tcp_subr.c

Figure 26.36 `tcp_respond` function: second half.

139–157 The fields in the IP and TCP headers must be initialized for the TCP checksum computation. These statements are similar to the way `tcp_template` initializes the `t_template` field. The sequence number and acknowledgment fields are passed by the caller as arguments. Finally `ip_output` sends the datagram.

26.10 Summary

This chapter has looked at the general-purpose function that generates most TCP segments (`tcp_output`) and the special-purpose function that generates RST segments and keepalive probes (`tcp_respond`).

Many factors determine whether TCP can send a segment or not: the flags in the segment, the window advertised by the other end, the amount of data ready to send, whether unacknowledged data already exists for the connection, and so on. Therefore the logic of `tcp_output` determines whether a segment can be sent (the first half of the function), and if so, what values to set all the TCP header fields to (the last half of the function). If a segment is sent, the TCP control block variables for the send sequence space must be updated.

One segment at a time is generated by `tcp_output`, and at the end of the function a check is made of whether more data can still be sent. If so, the function loops around and tries to send another segment. This looping continues until there is no more data to

send, or until some other condition (e.g., the receiver's advertised window) stops the transmission.

A TCP segment can also contain options. The options supported by Net/3 specify the maximum segment size, a window scale factor, and a pair of timestamps. The first two can only appear with SYN segments, while the timestamp option (if supported by both ends) normally appears in every segment. Since the window scale and timestamp options are newer and optional, if the first end to send a SYN wants to use the option, it sends the option with its SYN and uses the option only if the other end's SYN also contains the option.

Exercises

26.1 Slow start is resumed in Figure 26.1 when there is a pause in the *sending* of data, yet the amount of idle time is calculated as the amount of time since the last segment was *received* on the connection. Why doesn't TCP calculate the idle time as the amount of time since the last segment was *sent* on the connection?

26.2 With Figure 26.6 we said that `len` is less than 0 if the FIN has been sent but not acknowledged and not retransmitted. What happens if the FIN is retransmitted?

26.3 Net/3 always sends the window scale and timestamp options with an active open. Why does the global variable `tcp_do_rfc1323` exist?

26.4 In Figure 25.28, which did not use the timestamp option, the RTT estimators are updated eight times. If the timestamp option had been used in this example, how many times would the RTT estimators have been updated?

26.5 In Figure 26.23 `bcopy` is called to store the received MSS in the variable `mss`. Why not cast the pointer to `opt[2]` into a pointer to an unsigned short and perform an assignment?

26.6 After Figure 26.29 we described a bug in the code, which can cause a bogus urgent offset to be sent. Propose a solution. (*Hint*: What is the largest amount of TCP data that can be sent in a segment?)

26.7 With Figure 26.32 we mentioned that an error of `ENOBUFS` is not returned to the process because (1) if the discarded segment contained data, the retransmission timer will expire and the data will be retransmitted, or (2) if the discarded segment was an ACK-only segment, the other end will retransmit its data when it doesn't receive the ACK. What if the discarded segment contains an RST?

26.8 Explain the settings of the PSH flag in Figure 20.3 of Volume 1.

26.9 Why does Figure 26.36 use the value of `ip_defttl` for the TTL, while Figure 26.32 uses the value in the PCB?

26.10 Describe what happens with the mbuf allocated in Figure 26.25 when IP options are specified by the process for the TCP connection. Implement a better solution.

26.11 `tcp_output` is a long function (about 500 lines, including comments), which can appear to be inefficient. But lots of the code handles special cases. Assume the function is called with a full-sized segment ready to be sent, and no special cases: no IP options and no special flags such as SYN, FIN, or URG. About how many lines of C code are actually executed? How many functions are called before the segment is passed to `ip_output`?

26.12 In the example at the end of Section 26.3 in which the application did a write of 100 bytes followed by a write of 50 bytes, would anything change if the application called `writev` once for both buffers, instead of calling `write` twice? Does anything change with `writev` if the two buffer lengths are 200 and 300, instead of 100 and 50?

26.13 The timestamp that is sent in the timestamp option is taken from the global `tcp_now`, which is incremented every 500 ms. Modify TCP to use a higher resolution timestamp value.

27

TCP Functions

27.1 Introduction

This chapter presents numerous TCP functions that we need to cover before discussing TCP input in the next two chapters:

- `tcp_drain` is the protocol's drain function, called when the kernel is out of mbufs. It does nothing.
- `tcp_drop` aborts a connection by sending an RST.
- `tcp_close` performs the normal TCP connection termination: send a FIN and wait for the four-way exchange to complete. Section 18.2 of Volume 1 talks about the four packets that are exchanged when a connection is closed.
- `tcp_mss` processes a received MSS option and calculates the MSS to announce when TCP sends an MSS option of its own.
- `tcp_ctlinput` is called when an ICMP error is received in response to a TCP segment, and it calls `tcp_notify` to process the ICMP error. `tcp_quench` is a special case function that handles ICMP source quench errors.
- The `TCP_REASS` macro and the `tcp_reass` function manipulate segments on TCP's reassembly queue for a given connection. This queue handles the receipt of out-of-order segments, some of which might overlap.
- `tcp_trace` adds records to the kernel's circular debug buffer for TCP (the `SO_DEBUG` socket option) that can be printed with the `trpt`(8) program.

27.2 tcp_drain Function

The simplest of all the TCP functions is `tcp_drain`. It is the protocol's `pr_drain` function, called by `m_reclaim` when the kernel runs out of mbufs. We saw in Figure 10.32 that `ip_drain` discards all the fragments on its reassembly queue, and UDP doesn't define a drain function. Although TCP holds onto mbufs—segments that have arrived out of order, but within the receive window for the socket—the Net/3 implementation of TCP does not discard these pending mbufs if the kernel runs out of space. Instead, `tcp_drain` does nothing, on the assumption that a received (but out-of-order) TCP segment is "more important" than an IP fragment.

27.3 tcp_drop Function

`tcp_drop` is called from numerous places to drop a connection by sending an RST and to report an error to the process. This differs from closing a connection (the `tcp_disconnect` function), which sends a FIN to the other end and follows the connection termination steps in the state transition diagram.

Figure 27.1 shows the seven places where `tcp_drop` is called and the `errno` argument.

Function	errno	Description
tcp_input	ENOBUFS	SYN arrives on listening socket, but kernel out of mbufs for t_template.
tcp_input	ECONNREFUSED	RST received in response to SYN.
tcp_input	ECONNRESET	RST received on existing connection.
tcp_timers	ETIMEDOUT	Retransmission timer has expired 13 times in a row with no ACK from other end (Figure 25.25).
tcp_timers	ETIMEDOUT	Connection-establishment timer has expired (Figure 25.15), or keepalive timer has expired with no response to nine consecutive probes (Figure 25.17)
tcp_usrreq	ECONNABORTED	PRU_ABORT request.
tcp_usrreq	0	Socket closed and SO_LINGER socket option set with linger time of 0.

Figure 27.1 Calls to `tcp_drop` and `errno` argument.

Figure 27.2 shows the `tcp_drop` function.

202–213 If TCP has received a SYN, the connection is synchronized and an RST must be sent to the other end. This is done by setting the state to CLOSED and calling `tcp_output`. In Figure 24.16 the value of `tcp_outflags` for the CLOSED state includes the RST flag.

214–216 If the error is `ETIMEDOUT` but a soft error was received on the connection (e.g., `EHOSTUNREACH`), the soft error becomes the socket error, instead of the less specific `ETIMEDOUT`.

217 `tcp_close` finishes closing the socket.

tcp_subr.c

```
202 struct tcpcb *
203 tcp_drop(tp, errno)
204 struct tcpcb *tp;
205 int     errno;
206 {
207     struct socket *so = tp->t_inpcb->inp_socket;

208     if (TCPS_HAVERCVDSYN(tp->t_state)) {
209         tp->t_state = TCPS_CLOSED;
210         (void) tcp_output(tp);
211         tcpstat.tcps_drops++;
212     } else
213         tcpstat.tcps_conndrops++;
214     if (errno == ETIMEDOUT && tp->t_softerror)
215         errno = tp->t_softerror;
216     so->so_error = errno;
217     return (tcp_close(tp));
218 }
```

tcp_subr.c

Figure 27.2 tcp_drop function.

27.4 tcp_close Function

tcp_close is normally called by tcp_input when the process has done a passive close and the ACK is received in the LAST_ACK state, and by tcp_timers when the 2MSL timer expires and the socket moves from the TIME_WAIT to CLOSED state. It is also called in other states, possibly after an error has occurred, as we saw in the previous section. It releases the memory occupied by the connection (the IP and TCP header template, the TCP control block, the Internet PCB, and any out-of-order segments remaining on the connection's reassembly queue) and updates the route characteristics.

We describe this function in three parts, the first two dealing with the route characteristics and the final part showing the release of resources.

Route Characteristics

Nine variables are maintained in the rt_metrics structure (Figure 18.26), six of which are used by TCP. Eight of these can be examined and changed with the route(8) command (the ninth, rmx_pksent is never used): these variables are shown in Figure 27.3.

Additionally, the -lock modifier can be used with the route command to set the corresponding RTV_*xxx* bit in the rmx_locks member (Figure 20.13). Setting the RTV_*xxx* bit tells the kernel not to update that metric.

When a TCP socket is closed, tcp_close updates three of the routing metrics—the smoothed RTT estimator, the smoothed mean deviation estimator, and the slow start threshold—but only if enough data was transferred on the connection to yield meaningful statistics and the variable is not locked.

Figure 27.4 shows the first part of tcp_close.

rt_metrics member	saved by tcp_close?	used by tcp_mss?	route(8) modifier
rmx_expire			-expire
rmx_hopcount			-hopcount
rmx_mtu		•	-mtu
rmx_recvpipe		•	-recvpipe
rmx_rtt	•	•	-rtt
rmx_rttvar	•	•	-rttvar
rmx_sendpipe		•	-sendpipe
rmx_ssthresh	•	•	-ssthresh

Figure 27.3 Members of the rt_metrics structure used by TCP.

Check if enough data sent to update statistics

234–248 The default send buffer size is 8192 bytes (sb_hiwat), so the first test is whether 131,072 bytes (16 full buffers) have been transferred across the connection. The initial send sequence number is compared to the maximum sequence number sent on the connection. Additionally, the socket must have a cached route and that route cannot be the default route. (See Exercise 19.2.)

> Notice there is a small chance for an error in the first test, because of sequence number wrap, if the amount of data transferred is within $N \times 2^{32}$ and $N \times 2^{32} + 131072$, for any N greater than 1. But few connections (today) transfer 4 gigabytes of data.
>
> Despite the prevalence of default routes in the Internet, this information is still useful to maintain in the routing table. If a host continually exchanges data with another host (or network), even if a default route can be used, a host-specific or network-specific route can be entered into the routing table with the route command just to maintain this information across connections. (See Exercise 19.2.) This information is lost when the system is rebooted.

250 The administrator can lock any of the variables from Figure 27.3, preventing them from being updated by the kernel, so before modifying each variable this lock must be checked.

Update RTT

251–264 t_srtt is stored as ticks × 8 (Figure 25.19) and rmx_rtt is stored as microseconds. So t_srtt is multiplied by 1,000,000 (RTM_RTTUNIT) and then divided by 2 (ticks/second) times 8. If a value for rmx_rtt already exists, the new value is one-half the old value plus one-half the new value. Otherwise the new value is stored in rmx_rtt.

Update mean deviation

265–273 The same algorithm is applied to the mean deviation estimator. It too is stored as microseconds, requiring a conversion from the t_rttvar units of ticks × 4.

tcp_subr.c

```
225 struct tcpcb *
226 tcp_close(tp)
227 struct tcpcb *tp;
228 {
229     struct tcpiphdr *t;
230     struct inpcb *inp = tp->t_inpcb;
231     struct socket *so = inp->inp_socket;
232     struct mbuf *m;
233     struct rtentry *rt;
234     /*
235      * If we sent enough data to get some meaningful characteristics,
236      * save them in the routing entry.  'Enough' is arbitrarily
237      * defined as the sendpipesize (default 8K) * 16.  This would
238      * give us 16 rtt samples assuming we only get one sample per
239      * window (the usual case on a long haul net).  16 samples is
240      * enough for the srtt filter to converge to within 5% of the correct
241      * value; fewer samples and we could save a very bogus rtt.
242      *
243      * Don't update the default route's characteristics and don't
244      * update anything that the user "locked".
245      */
246     if (SEQ_LT(tp->iss + so->so_snd.sb_hiwat * 16, tp->snd_max) &&
247         (rt = inp->inp_route.ro_rt) &&
248      ((struct sockaddr_in *) rt_key(rt))->sin_addr.s_addr != INADDR_ANY) {
249         u_long  i;
250         if ((rt->rt_rmx.rmx_locks & RTV_RTT) == 0) {
251             i = tp->t_srtt *
252                 (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTT_SCALE));
253             if (rt->rt_rmx.rmx_rtt && i)
254                 /*
255                  * filter this update to half the old & half
256                  * the new values, converting scale.
257                  * See route.h and tcp_var.h for a
258                  * description of the scaling constants.
259                  */
260                 rt->rt_rmx.rmx_rtt =
261                     (rt->rt_rmx.rmx_rtt + i) / 2;
262             else
263                 rt->rt_rmx.rmx_rtt = i;
264         }
265         if ((rt->rt_rmx.rmx_locks & RTV_RTTVAR) == 0) {
266             i = tp->t_rttvar *
267                 (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTTVAR_SCALE));
268             if (rt->rt_rmx.rmx_rttvar && i)
269                 rt->rt_rmx.rmx_rttvar =
270                     (rt->rt_rmx.rmx_rttvar + i) / 2;
271             else
272                 rt->rt_rmx.rmx_rttvar = i;
273         }
```

tcp_subr.c

Figure 27.4 tcp_close function: update RTT and mean deviation.

Figure 27.5 shows the next part of `tcp_close`, which updates the slow start threshold for the route.

tcp_subr.c

```
274         /*
275          * update the pipelimit (ssthresh) if it has been updated
276          * already or if a pipesize was specified & the threshhold
277          * got below half the pipesize.  I.e., wait for bad news
278          * before we start updating, then update on both good
279          * and bad news.
280          */
281         if ((rt->rt_rmx.rmx_locks & RTV_SSTHRESH) == 0 &&
282             (i = tp->snd_ssthresh) && rt->rt_rmx.rmx_ssthresh ||
283             i < (rt->rt_rmx.rmx_sendpipe / 2)) {
284             /*
285              * convert the limit from user data bytes to
286              * packets then to packet data bytes.
287              */
288             i = (i + tp->t_maxseg / 2) / tp->t_maxseg;
289             if (i < 2)
290                 i = 2;
291             i *= (u_long) (tp->t_maxseg + sizeof(struct tcpiphdr));
292             if (rt->rt_rmx.rmx_ssthresh)
293                 rt->rt_rmx.rmx_ssthresh =
294                     (rt->rt_rmx.rmx_ssthresh + i) / 2;
295             else
296                 rt->rt_rmx.rmx_ssthresh = i;
297         }
298     }
```

tcp_subr.c

Figure 27.5 `tcp_close` function: update slow start threshold.

274–283 The slow start threshold is updated only if (1) it has been updated already (`rmx_ssthresh` is nonzero) or (2) `rmx_sendpipe` is specified by the administrator and the new value of `snd_ssthresh` is less than one-half the value of `rmx_sendpipe`. As the comment in the code indicates, TCP does not update the value of `rmx_ssthresh` until it is forced to because of packet loss; from that point on it considers itself free to adjust the value either up or down.

284–290 The variable `snd_ssthresh` is maintained in bytes. The first conversion divides this variable by the MSS (`t_maxseg`), yielding the number of segments. The addition of one-half `t_maxseg` rounds the integer result. The lower bound on this result is two segments.

291–297 The size of the IP and TCP headers (40) is added to the MSS and multipled by the number of segments. This value updates `rmx_ssthresh`, using the same filtering as in Figure 27.4 (one-half the old plus one-half the new).

Resource Release

The final part of `tcp_close`, shown in Figure 27.6, releases the memory resources held by the socket.

tcp_subr.c

```
299     /* free the reassembly queue, if any */
300     t = tp->seg_next;
301     while (t != (struct tcpiphdr *) tp) {
302         t = (struct tcpiphdr *) t->ti_next;
303         m = REASS_MBUF((struct tcpiphdr *) t->ti_prev);
304         remque(t->ti_prev);
305         m_freem(m);
306     }
307     if (tp->t_template)
308         (void) m_free(dtom(tp->t_template));
309     free(tp, M_PCB);
310     inp->inp_ppcb = 0;
311     soisdisconnected(so);
312     /* clobber input pcb cache if we're closing the cached connection */
313     if (inp == tcp_last_inpcb)
314         tcp_last_inpcb = &tcb;
315     in_pcbdetach(inp);
316     tcpstat.tcps_closed++;
317     return ((struct tcpcb *) 0);
318 }
```

tcp_subr.c

Figure 27.6 `tcp_close` function: release connection resources.

Release any mbufs on reassembly queue

299–306 If any segments are left on the connection's reassembly queue, they are discarded. This queue is for segments that arrive out of order but within the receive window. They are held in a reassembly queue until the required "earlier" segments are received, at which time they are reassembled and passed to the application in the correct order. We discuss this in more detail in Section 27.9.

Release header template and TCP control block

307–311 The template of the IP and TCP headers is released by `m_free` and the TCP control block is released by `free`. `soisdisconnected` marks the socket as disconnected.

Release PCB

312–318 If the Internet PCB for this socket is the one currently cached by TCP, the cache is marked as empty by setting `tcp_last_inpcb` to the head of TCP's PCB list. The PCB is then detached, which releases the memory used by the PCB.

27.5 `tcp_mss` Function

The `tcp_mss` function is called from two other functions:

1. from `tcp_output`, when a SYN segment is being sent, to include an MSS option, and
2. from `tcp_input`, when an MSS option is received in a SYN segment.

The tcp_mss function checks for a cached route to the destination and calculates the MSS to use for this connection.

Figure 27.7 shows the first part of tcp_mss, which acquires a route to the destination if one is not already held by the PCB.

tcp_input.c

```
1391 int
1392 tcp_mss(tp, offer)
1393 struct tcpcb *tp;
1394 u_int   offer;
1395 {
1396     struct route *ro;
1397     struct rtentry *rt;
1398     struct ifnet *ifp;
1399     int     rtt, mss;
1400     u_long  bufsize;
1401     struct inpcb *inp;
1402     struct socket *so;
1403     extern int tcp_mssdflt;

1404     inp = tp->t_inpcb;
1405     ro = &inp->inp_route;

1406     if ((rt = ro->ro_rt) == (struct rtentry *) 0) {
1407         /* No route yet, so try to acquire one */
1408         if (inp->inp_faddr.s_addr != INADDR_ANY) {
1409             ro->ro_dst.sa_family = AF_INET;
1410             ro->ro_dst.sa_len = sizeof(ro->ro_dst);
1411             ((struct sockaddr_in *) &ro->ro_dst)->sin_addr =
1412                 inp->inp_faddr;
1413             rtalloc(ro);
1414         }
1415         if ((rt = ro->ro_rt) == (struct rtentry *) 0)
1416             return (tcp_mssdflt);
1417     }
1418     ifp = rt->rt_ifp;
1419     so = inp->inp_socket;
```

tcp_input.c

1420–143

Figure 27.7 tcp_mss function: acquire a route if one is not held by the PCB.

Acquire a route if necessary

1391–1417 If the socket does not have a cached route, rtalloc acquires one. The interface pointer associated with the outgoing route is saved in ifp. Knowing the outgoing interface is important, since its associated MTU can affect the MSS announced by TCP. If a route is not acquired, the default of 512 (tcp_mssdflt) is returned immediately.

1433–143

The next part of tcp_mss, shown in Figure 27.8, checks whether the route has metrics associated with it; if so, the variables t_rttmin, t_srtt, and t_rttvar can be initialized from the metrics.

tcp_input.c

```
1420    /*
1421     * While we're here, check if there's an initial rtt
1422     * or rttvar.  Convert from the route-table units
1423     * to scaled multiples of the slow timeout timer.
1424     */
1425    if (tp->t_srtt == 0 && (rtt = rt->rt_rmx.rmx_rtt)) {
1426        /*
1427         * XXX the lock bit for RTT indicates that the value
1428         * is also a minimum value; this is subject to time.
1429         */
1430        if (rt->rt_rmx.rmx_locks & RTV_RTT)
1431            tp->t_rttmin = rtt / (RTM_RTTUNIT / PR_SLOWHZ);
1432        tp->t_srtt = rtt / (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTT_SCALE));

1433        if (rt->rt_rmx.rmx_rttvar)
1434            tp->t_rttvar = rt->rt_rmx.rmx_rttvar /
1435                (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTTVAR_SCALE));
1436        else
1437            /* default variation is +- 1 rtt */
1438            tp->t_rttvar =
1439                tp->t_srtt * TCP_RTTVAR_SCALE / TCP_RTT_SCALE;

1440        TCPT_RANGESET(tp->t_rxtcur,
1441                      ((tp->t_srtt >> 2) + tp->t_rttvar) >> 1,
1442                      tp->t_rttmin, TCPTV_REXMTMAX);
1443    }
```

tcp_input.c

Figure 27.8 tcp_mss function: check if the route has an associated RTT metric.

Initialize smoothed RTT estimator

1420–1432 If there are no RTT measurements yet for the connection (t_srtt is 0) and rmx_rtt is nonzero, the latter initializes the smoothed RTT estimator t_srtt. If the RTV_RTT bit in the routing metric lock flag is set, it indicates that rmx_rtt should also be used to initialize the minimum RTT for this connection (t_rttmin). We saw that tcp_newtcpcb initializes t_rttmin to 2 ticks.

rmx_rtt (in units of microseconds) is converted to t_srtt (in units of ticks × 8). This is the reverse of the conversion done in Figure 27.4. Notice that t_rttmin is set to one-eighth the value of t_srtt, since the former is not divided by the scale factor TCP_RTT_SCALE.

Initialize smoothed mean deviation estimator

1433–1439 If the stored value of rmx_rttvar is nonzero, it is converted from units of microseconds into ticks × 4 and stored in t_rttvar. But if the value is 0, t_rttvar is set to t_rtt, that is, the variation is set to the mean. This defaults the variation to ± 1 RTT. Since the units of the former are ticks × 4 and the units of the latter are ticks × 8, the value of t_srtt is converted accordingly.

Calculate initial RTO

1440–1442 The current *RTO* is calculated and stored in `t_rxtcur`, using the unscaled equation

$$RTO = srtt + 2 \times rttvar$$

A multipler of 2, instead of 4, is used to calculate the first *RTO*. This is the same equation that was used in Figure 25.21. Substituting the scaling relationships we get

$$RTO = \frac{t\_srtt}{8} + 2 \times \frac{t\_rttvar}{4}$$

$$= \frac{\frac{t\_srtt}{4} + t\_rttvar}{2}$$

which is the second argument to `TCPT_RANGESET`.

The next part of `tcp_mss`, shown in Figure 27.9, calculates the MSS.

tcp_input.c

```
1444     /*
1445      * if there's an mtu associated with the route, use it
1446      */
1447     if (rt->rt_rmx.rmx_mtu)
1448         mss = rt->rt_rmx.rmx_mtu - sizeof(struct tcpiphdr);
1449     else {
1450         mss = ifp->if_mtu - sizeof(struct tcpiphdr);

1451 #if (MCLBYTES & (MCLBYTES - 1)) == 0
1452         if (mss > MCLBYTES)
1453             mss &= ~(MCLBYTES - 1);
1454 #else
1455         if (mss > MCLBYTES)
1456             mss = mss / MCLBYTES * MCLBYTES;
1457 #endif

1458         if (!in_localaddr(inp->inp_faddr))
1459             mss = min(mss, tcp_mssdflt);
1460     }
```

tcp_input.c

Figure 27.9 `tcp_mss` function: calculate MSS.

Use MSS from routing table MTU

1444–1450 If the MTU is set in the routing table, `mss` is set to that value. Otherwise `mss` starts at the value of the outgoing interface MTU minus 40 (the default size of the IP and TCP headers). For an Ethernet, `mss` would start at 1460.

Round MSS down to multiple of MCLBYTES

1451–1457 The goal of these lines of code is to reduce the value of `mss` to the next-lower multiple of the mbuf cluster size, if `mss` exceeds `MCLBYTES`. If the value of `MCLBYTES` (typically 1024 or 2048) logically ANDed with the value minus 1 equals 0, then `MCLBYTES` is a power of 2. For example, 1024 (`0x400`) logically ANDed with 1023 (`0x3ff`) is 0.

The value of mss is reduced to the next-lower multiple of MCLBYTES by clearing the appropriate number of low-order bits: if the cluster size is 1024, logically ANDing mss with the one's complement of 1023 (0xfffffc00) clears the low-order 10 bits. For an Ethernet, this reduces mss from 1460 to 1024. If the cluster size is 2048, logically ANDing mss with the one's complement of 2047 (0xffff8000) clears the low-order 11 bits. For a token ring with an MTU of 4464, this reduces the value of mss from 4424 to 4096. If MCLBYTES is not a power of 2, the rounding down to the next-lower multiple of MCLBYTES is done with an integer division followed by a multiplication.

Check if destination local or nonlocal

1458–1459 If the foreign IP address is not local (in_localaddr returns 0), and if mss is greater than 512 (tcp_mssdflt), it is set to 512.

> Whether an IP address is "local" or not depends on the value of the global subnetsarelocal, which is initialized from the symbol SUBNETSARELOCAL when the kernel is compiled. The default value is 1, meaning that an IP address with the same network ID as one of the host's interfaces is considered local. If the value is 0, an IP address must have the same network ID and the same subnet ID as one of the host's interfaces to be considered local.
>
> This minimization for nonlocal hosts is an attempt to avoid fragmentation across wide-area networks. It is a historical artifact from the ARPANET when the MTU across most WAN links was 1006. As discussed in Section 11.7 of Volume 1, most WANs today support an MTU of 1500 or greater. See also the discussion of the path MTU discovery feature (RFC 1191 [Mogul and Deering 1990]), in Section 24.2 of Volume 1. Net/3 does not support path MTU discovery.

The final part of tcp_mss is shown in Figure 27.10.

Other end's MSS is upper bound

1461–1472 The argument offer is nonzero when this function is called from tcp_input, and its value is the MSS advertised by the other end. If the value of mss is greater than the value advertised by the other end, it is set to the value of offer. For example, if the function calculates an mss of 1024 but the advertised value from the other end is 512, mss must be set to 512. Conversely, if mss is calculated as 536 (say the outgoing MTU is 576) and the other end advertises an MSS of 1460, TCP will use 536. TCP can always use a value less than the advertised MSS, but it can't exceed the advertised value. The argument offer is 0 when this function is called by tcp_output to send an MSS option. The value of mss is also lower-bounded by 32.

1473–1483 If the value of mss has decreased from the default set by tcp_newtcpcb in the variable t_maxseg (512), or if TCP is processing a received MSS option (offer is nonzero), the following steps occur. First, if the value of rmx_sendpipe has been stored for the route, its value will be used as the send buffer high-water mark (Figure 16.4). If the buffer size is less than mss, the smaller value is used. This should never happen unless the application explicitly sets the send buffer size to a small value, or the administrator sets rmx_sendpipe to a small value, since the high-water mark of the send buffer defaults to 8192, larger than most values for the MSS.

tcp_input.c

```
1461    /*
1462     * The current mss, t_maxseg, was initialized to the default value
1463     * of 512 (tcp_mssdflt) by tcp_newtcpcb().
1464     * If we compute a smaller value, reduce the current mss.
1465     * If we compute a larger value, return it for use in sending
1466     * a max seg size option, but don't store it for use
1467     * unless we received an offer at least that large from peer.
1468     * However, do not accept offers under 32 bytes.
1469     */
1470    if (offer)
1471        mss = min(mss, offer);
1472    mss = max(mss, 32);         /* sanity */
1473    if (mss < tp->t_maxseg || offer != 0) {
1474        /*
1475         * If there's a pipesize, change the socket buffer
1476         * to that size.  Make the socket buffers an integral
1477         * number of mss units; if the mss is larger than
1478         * the socket buffer, decrease the mss.
1479         */
1480        if ((bufsize = rt->rt_rmx.rmx_sendpipe) == 0)
1481            bufsize = so->so_snd.sb_hiwat;
1482        if (bufsize < mss)
1483            mss = bufsize;
1484        else {
1485            bufsize = roundup(bufsize, mss);
1486            if (bufsize > sb_max)
1487                bufsize = sb_max;
1488            (void) sbreserve(&so->so_snd, bufsize);
1489        }
1490        tp->t_maxseg = mss;
1491        if ((bufsize = rt->rt_rmx.rmx_recvpipe) == 0)
1492            bufsize = so->so_rcv.sb_hiwat;
1493        if (bufsize > mss) {
1494            bufsize = roundup(bufsize, mss);
1495            if (bufsize > sb_max)
1496                bufsize = sb_max;
1497            (void) sbreserve(&so->so_rcv, bufsize);
1498        }
1499    }
1500    tp->snd_cwnd = mss;
1501    if (rt->rt_rmx.rmx_ssthresh) {
1502        /*
1503         * There's some sort of gateway or interface
1504         * buffer limit on the path.  Use this to set
1505         * the slow start threshhold, but set the
1506         * threshold to no less than 2*mss.
1507         */
1508        tp->snd_ssthresh = max(2 * mss, rt->rt_rmx.rmx_ssthresh);
1509    }
1510    return (mss);
1511 }
```

tcp_input.c

Figure 27.10 tcp_mss function: complete processing.

Round buffer sizes to multiple of MSS

1484–1489 The send buffer size is rounded up to the next integral multiple of the MSS, bounded by the value of sb_max (262,144 on Net/3, which is 256×1024). The socket's high-water mark is set by sbreserve. For example, the default high-water mark is 8192, but for a local TCP connection on an Ethernet with a cluster size of 2048 (i.e., an MSS of 1460) this code increases the high-water mark to 8760 (which is 6×1460). But for a nonlocal connection with an MSS of 512, the high-water mark is left at 8192.

1490 The value of t_maxseg is set, either because it decreased from the default (512) or because an MSS option was received from the other end.

1491–1499 The same logic just applied to the send buffer is also applied to the receive buffer.

Initialize congestion window and slow start threshold

1500–1509 The value of the congestion window, snd_cwnd, is set to one segment. If the rmx_ssthresh value in the routing table is nonzero, the slow start threshold (snd_ssthresh) is set to that value, but the value must not be less than two segments.

1510 The value of mss is returned by the function. tcp_input ignores this value in Figure 28.10 (since it received an MSS from the other end), but tcp_output sends this value as the announced MSS in Figure 26.23.

Example

Let's go through an example of a TCP connection establishment and the operation of tcp_mss, since it can be called twice: once when the SYN is sent and once when a SYN is received with an MSS option.

1. The socket is created and tcp_newtcpcb sets t_maxseg to 512.
2. The process calls connect, and tcp_output calls tcp_mss with an offer argument of 0, to include an MSS option with the SYN. Assuming a local destination, an Ethernet LAN, and an mbuf cluster size of 2048, mss is set to 1460 by the code in Figure 27.9. Since offer is 0, Figure 27.10 leaves the value as 1460 and this is the function's return value. The buffer sizes aren't modified, since 1460 is larger than the default (512) and a value hasn't been received from the other end yet. tcp_output sends an MSS option announcing a value of 1460.
3. The other end replies with its SYN, announcing an MSS of 1024. tcp_input calls tcp_mss with an offer argument of 1024. The logic in Figure 27.9 still yields a value of 1460 for mss, but the call to min at the beginning of Figure 27.10 reduces this to 1024. Since the value of offer is nonzero, the buffer sizes are rounded up to the next integral multiple of 1024 (i.e., they're left at 8192). t_maxseg is set to 1024.

> It might appear that the logic of tcp_mss is flawed: TCP announces an MSS of 1460 but receives an MSS of 1024 from the other end. While TCP is restricted to sending 1024-byte segments, the other end is free to send 1460-byte segments. We might think that the send buffer should be a multiple of 1024, but the receive buffer should be a multiple of 1460. Yet the code in Figure 27.10 sets both buffer sizes based on the *received* MSS. The reasoning is that even if TCP announces an MSS of 1460, since it receives an MSS of 1024 from the other end, the other end probably won't send 1460-byte segments, but will restrict itself to 1024-byte segments.

27.6 tcp_ctlinput Function

Recall from Figure 22.32 that tcp_ctlinput processes five types of ICMP errors: destination unreachable, parameter problem, source quench, time exceeded, and redirects. All redirects are passed to both TCP and UDP. For the other four errors, tcp_ctlinput is called only if a TCP segment caused the error.

tcp_ctlinput is shown in Figure 27.11. It is similar to udp_ctlinput, shown in Figure 23.30.

tcp_subr.c

```
355 void
356 tcp_ctlinput(cmd, sa, ip)
357 int     cmd;
358 struct sockaddr *sa;
359 struct ip *ip;
360 {
361     struct tcphdr *th;
362     extern struct in_addr zeroin_addr;
363     extern u_char inetctlerrmap[];
364     void    (*notify) (struct inpcb *, int) = tcp_notify;

365     if (cmd == PRC_QUENCH)
366         notify = tcp_quench;
367     else if (!PRC_IS_REDIRECT(cmd) &&
368              ((unsigned) cmd > PRC_NCMDS || inetctlerrmap[cmd] == 0))
369         return;
370     if (ip) {
371         th = (struct tcphdr *) ((caddr_t) ip + (ip->ip_hl << 2));
372         in_pcbnotify(&tcb, sa, th->th_dport, ip->ip_src, th->th_sport,
373                      cmd, notify);
374     } else
375         in_pcbnotify(&tcb, sa, 0, zeroin_addr, 0, cmd, notify);
376 }
```

tcp_subr.c

328–34

Figure 27.11 tcp_ctlinput function.

365–366 The only difference in the logic from udp_ctlinput is how an ICMP source quench error is handled. UDP ignores these errors since the PRC_QUENCH entry of inetctlerrmap is 0. TCP explicitly checks for this error, changing the notify function from its default of tcp_notify to tcp_quench.

346–35

27.7 tcp_notify Function

tcp_notify is called by tcp_ctlinput to handle destination unreachable, parameter problem, time exceeded, and redirect errors. This function is more complicated than its UDP counterpart, since TCP must intelligently handle soft errors for an established connection. Figure 27.12 shows the tcp_notify function.

tcp_subr.c

```
328 void
329 tcp_notify(inp, error)
330 struct inpcb *inp;
331 int     error;
332 {
333     struct tcpcb *tp = (struct tcpcb *) inp->inp_ppcb;
334     struct socket *so = inp->inp_socket;

335     /*
336      * Ignore some errors if we are hooked up.
337      * If connection hasn't completed, has retransmitted several times,
338      * and receives a second error, give up now.  This is better
339      * than waiting a long time to establish a connection that
340      * can never complete.
341      */
342     if (tp->t_state == TCPS_ESTABLISHED &&
343         (error == EHOSTUNREACH || error == ENETUNREACH ||
344          error == EHOSTDOWN)) {
345         return;
346     } else if (tp->t_state < TCPS_ESTABLISHED && tp->t_rxtshift > 3 &&
347                tp->t_softerror)
348         so->so_error = error;
349     else
350         tp->t_softerror = error;
351     wakeup((caddr_t) & so->so_timeo);
352     sorwakeup(so);
353     sowwakeup(so);
354 }
```

tcp_subr.c

Figure 27.12 tcp_notify function.

328–345 If the connection is ESTABLISHED, the errors EHOSTUNREACH, ENETUNREACH, and EHOSTDOWN are ignored.

> This handling of these three errors is new with 4.4BSD. Net/2 and earlier releases recorded these errors in the connection's soft error variable (t_softerror), and the error was reported to the process should the connection eventually fail. Recall that tcp_xmit_timer resets this variable to 0 when an ACK is received for a segment that hasn't been retransmitted.

346–353 If the connection is not yet established, TCP has retransmitted the current segment four or more times, and an error has already been recorded in t_softerror, the current error is recorded in the socket's so_error variable. By setting this socket variable, the socket becomes readable and writable if the process calls select. Otherwise the current error is just saved in t_softerror. We saw that tcp_drop sets the socket error to this saved value if the connection is subsequently dropped because of a timeout. Any processes waiting to receive or send on the socket are then awakened to receive the error.

27.8 tcp_quench Function

tcp_quench, which is shown in Figure 27.13, is called by tcp_ctlinput when a source quench is received for the connection, and by tcp_output (Figure 26.32) when ip_output returns ENOBUFS.

tcp_subr.c

```
381 void
382 tcp_quench(inp, errno)
383 struct inpcb *inp;
384 int     errno;
385 {
386     struct tcpcb *tp = intotcpcb(inp);
387     if (tp)
388         tp->snd_cwnd = tp->t_maxseg;
389 }
```

tcp_subr.c

Figure 27.13 tcp_quench function.

The congestion window is set to one segment, causing slow start to take over. The slow start threshold is not changed (as it is when tcp_timers handles a retransmission timeout), so the window will open up exponentially until snd_ssthresh is reached, or congestion occurs.

27.9 TCP_REASS Macro and tcp_reass Function

TCP segments can arrive out of order, and it is TCP's responsibility to place the misordered segments into the correct order for presentation to the process. For example, if a receiver advertises a window of 4096 with byte number 0 as the next expected byte, and receives a segment with bytes 0–1023 (an in-order segment) followed by a segment with bytes 2048–3071, this second segment is out of order. TCP does not discard the out-of-order segment if it is within the receive window. Instead it places the segment on the reassembly list for the connection, waiting for the missing segment to arrive (with bytes 1024–2047), at which time it can acknowledge bytes 1024–3071 and pass these 2048 bytes to the process. In this section we examine the code that manipulates the TCP reassembly queue, before discussing tcp_input in the next two chapters.

If we assume that a single mbuf contains the IP header, TCP header, and 4 bytes of TCP data (recall the left half of Figure 2.14) we would have the arrangement shown in Figure 27.14. We also assume the data bytes are sequence numbers 7, 8, 9, and 10.

The ipovly and tcphdr structures form the tcpiphdr structure, which we showed in Figure 24.12. We showed a picture of the tcphdr structure in Figure 24.10. In Figure 27.14 we show only the variables used in the reassembly: ti_next, ti_prev, ti_len, ti_sport, ti_dport, and ti_seq. The first two are pointers that form a doubly linked list of all the out-of-order segments for a given connection. The head of this list is the TCP control block for the connection: the seg_next and seg_prev members, which are the first two members of the structure. The ti_next and ti_prev

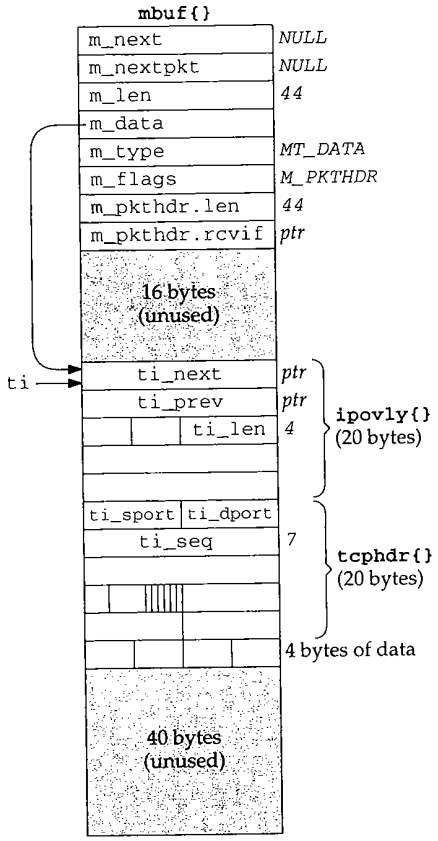


Figure 27.14 Example mbuf with IP and TCP headers and 4 bytes of data.

pointers overlay the first 8 bytes of the IP header, which aren't needed once the datagram reaches TCP. `ti_len` is the length of the TCP data, and is calculated and stored by TCP before verifying the TCP checksum.

TCP_REASS Macro

When data is received by `tcp_input`, the macro `TCP_REASS`, shown in Figure 27.15, is invoked to place the data onto the connection's reassembly queue. This macro is called from only one place: see Figure 29.22.

54–63 `tp` is a pointer to the TCP control block for the connection and `ti` is a pointer to the `tcpiphdr` structure for the received segment. If the following three conditions are all true:

1. this segment is in-order (the sequence number `ti_seq` equals the next expected sequence number for the connection, `rcv_nxt`), and

tcp_input.c

```
53 #define TCP_REASS(tp, ti, m, so, flags) { \
54     if ((ti)->ti_seq == (tp)->rcv_nxt && \
55         (tp)->seg_next == (struct tcpiphdr *)(tp) && \
56         (tp)->t_state == TCPS_ESTABLISHED) { \
57         tp->t_flags |= TF_DELACK; \
58         (tp)->rcv_nxt += (ti)->ti_len; \
59         flags = (ti)->ti_flags & TH_FIN; \
60         tcpstat.tcps_rcvpack++; \
61         tcpstat.tcps_rcvbyte += (ti)->ti_len; \
62         sbappend(&(so)->so_rcv, (m)); \
63         sorwakeup(so); \
64     } else { \
65         (flags) = tcp_reass((tp), (ti), (m)); \
66         tp->t_flags |= TF_ACKNOW; \
67     } \
68 }
```

tcp_input.c

Figure 27.15 TCP_REASS macro: add data to reassembly queue for connection.

2. the reassembly queue for the connection is empty (`seg_next` points to itself, not some mbuf), and
3. the connection is ESTABLISHED,

the following steps take place: a delayed ACK is scheduled, `rcv_nxt` is updated with the amount of data in the segment, the `flags` argument is set to `TH_FIN` if the FIN flag is set in the TCP header of the segment, two statistics are updated, the data is appended to the socket's receive buffer, and any receiving processes waiting for the socket are awakened.

The reason all three conditions must be true is that, first, if the data is out of order, it must be placed onto the connection's reassembly queue and the "preceding" segments must be received before anything can be passed to the process. Second, even if the data is in order, if there is out-of-order data already on the reassembly queue, there's a chance that the new segment might fill a hole, allowing the received segment and one or more segments on the queue to all be passed to the process. Third, it is OK for data to arrive with a SYN segment that establishes a connection, but that data cannot be passed to the process until the connection is ESTABLISHED—any such data is just added to the reassembly queue when it arrives.

64–67 If these three conditions are not all true, the `TCP_REASS` macro calls the function `tcp_reass` to add the segment to the reassembly queue. Since the segment is either out of order, or the segment might fill a hole from previously received out-of-order segments, an immediate ACK is scheduled. One important feature of TCP is that a receiver should generate an immediate ACK when an out-of-order segment is received. This aids the *fast retransmit* algorithm (Section 29.4).

Before looking at the code for the `tcp_reass` function, we need to explain what's done with the two port numbers in the TCP header in Figure 27.14, `ti_sport` and

`tcp_re`

69–83

84–90

ti_dport. Once the TCP control block is located and tcp_reass is called, these two port numbers are no longer needed. Therefore, when a TCP segment is placed on a reassembly queue, the address of the corresponding mbuf is stored over these two port numbers. In Figure 27.14 this isn't needed, because the IP and TCP headers are in the data portion of the mbuf, so the dtom macro works. But recalling our discussion of m_pullup in Section 2.6, if the IP and TCP headers are in a cluster (as in Figure 2.16, which is the normal case for a full-sized TCP segment), the dtom macro doesn't work. We mentioned in that section that TCP stores its own back pointer from the TCP header to the mbuf, and that back pointer is stored over the two TCP port numbers.

Figure 27.16 shows an example of this technique with two out-of-order segments for a connection, each segment stored in an mbuf cluster. The head of the doubly linked list of out-of-order segments is the seg_next member of the control block for this connection. To simplify the figure we don't show the seg_prev pointer and the ti_next pointer of the last segment on the list.

The next expected sequence number is 1 (rcv_nxt) but we assume that segment was lost. The next two segments have been received, containing bytes 1461–4380, but they are out of order. The segments were placed into clusters by m_devget, as shown in Figure 2.16.

The first 32 bits of the TCP header contain a back pointer to the corresponding mbuf. This back pointer is used in the tcp_reass function, shown next.

tcp_reass Function

Figure 27.17 shows the first part of the tcp_reass function. The arguments are: tp, a pointer to the TCP control block for the received segment; ti, a pointer to the IP and TCP headers of the received segment; and m, a pointer to the mbuf chain for the received segment. As mentioned earlier, ti can point into the data area of the mbuf pointed to by m, or ti can point into a cluster.

69–83 We'll see that tcp_input calls tcp_reass with a null ti pointer when a SYN is acknowledged (Figures 28.20 and 29.2). This means the connection is now established, and any data that might have arrived with the SYN (which tcp_reass had to queue earlier) can now be passed to the application. Data that arrives with a SYN cannot be passed to the process until the connection is established. The label present is in Figure 27.23.

84–90 Go through the list of segments for this connection, starting at seg_next, to find the first one with a sequence number that is greater than the received sequence number (ti_seq). Note that the if statement is the entire body of the for loop.

Figure 27.18 shows an example with two out-of-order segments already on the queue when a new segment arrives. We show the pointer q pointing to the next segment on the list, the one with bytes 10–15. In this figure we also show the two pointers ti_next and ti_prev, the starting sequence number (ti_seq), the length (ti_len), and the sequence numbers of the data bytes. With the small segments we show, each segment is probably in a single mbuf, as in Figure 27.14.

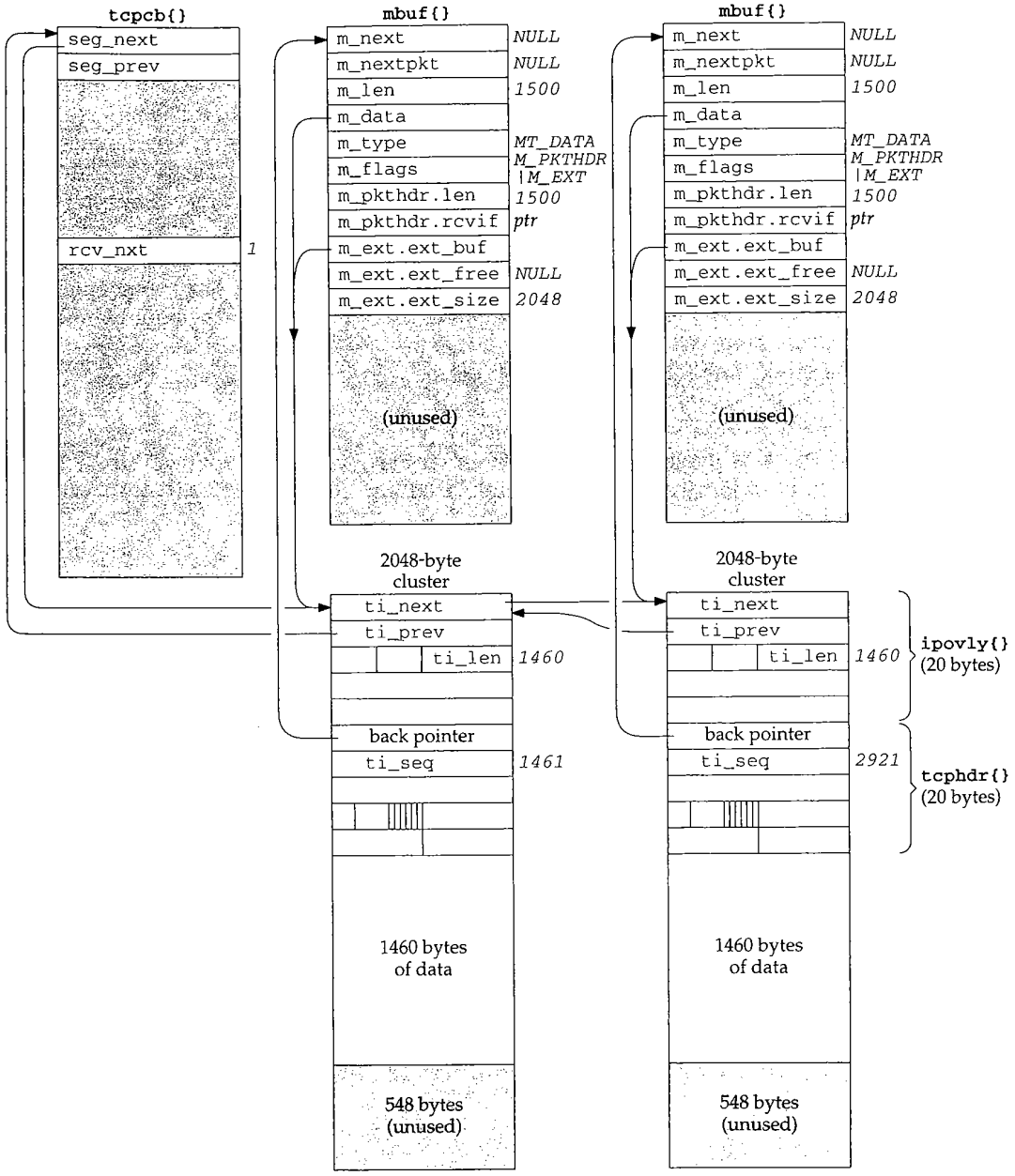


Figure 27.16 Two out-of-order TCP segments stored in mbuf clusters.

tcp_input.c

```
69 int
70 tcp_reass(tp, ti, m)
71 struct tcpcb *tp;
72 struct tcpiphdr *ti;
73 struct mbuf *m;
74 {
75     struct tcpiphdr *q;
76     struct socket *so = tp->t_inpcb->inp_socket;
77     int     flags;

78     /*
79      * Call with ti==0 after become established to
80      * force pre-ESTABLISHED data up to user socket.
81      */
82     if (ti == 0)
83         goto present;

84     /*
85      * Find a segment that begins after this one does.
86      */
87     for (q = tp->seg_next; q != (struct tcpiphdr *) tp;
88          q = (struct tcpiphdr *) q->ti_next)
89         if (SEQ_GT(q->ti_seq, ti->ti_seq))
90             break;
```

tcp_input.c

Figure 27.17 tcp_reass function: first part.

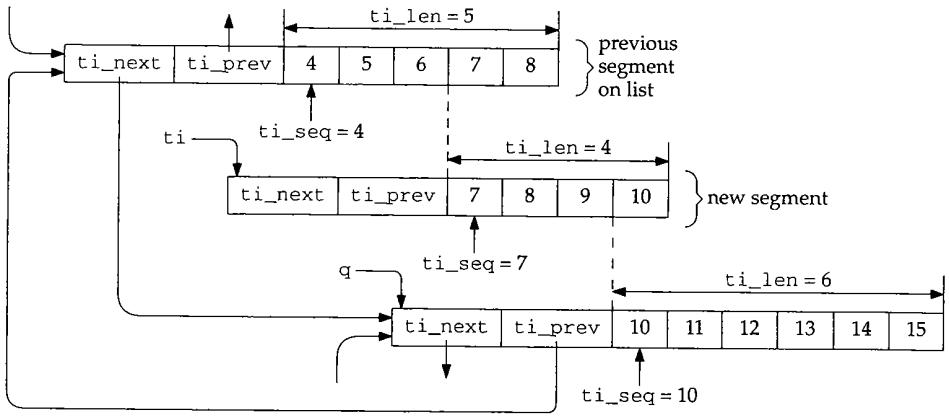


Figure 27.18 Example of TCP reassembly queue with overlapping segments.

The next part of tcp_reass is shown in Figure 27.19.

tcp_input.c

```
91      /*
92       * If there is a preceding segment, it may provide some of
93       * our data already.  If so, drop the data from the incoming
94       * segment.  If it provides all of our data, drop us.
95       */
96      if ((struct tcpiphdr *) q->ti_prev != (struct tcpiphdr *) tp) {
97          int     i;
98          q = (struct tcpiphdr *) q->ti_prev;
99          /* conversion to int (in i) handles seq wraparound */
100         i = q->ti_seq + q->ti_len - ti->ti_seq;
101         if (i > 0) {
102             if (i >= ti->ti_len) {
103                 tcpstat.tcps_rcvduppack++;
104                 tcpstat.tcps_rcvdupbyte += ti->ti_len;
105                 m_freem(m);
106                 return (0);
107             }
108             m_adj(m, i);
109             ti->ti_len -= i;
110             ti->ti_seq += i;
111         }
112         q = (struct tcpiphdr *) (q->ti_next);
113     }
114     tcpstat.tcps_rcvoopack++;
115     tcpstat.tcps_rcvoobyte += ti->ti_len;
116     REASS_MBUF(ti) = m;          /* XXX */
```

tcp_input.c

Figure 27.19 tcp_reass function: second part.

91–107 If there is a segment before the one pointed to by q, that segment may overlap the new segment. The pointer q is moved to the previous segment on the list (the one with bytes 4–8 in Figure 27.18) and the number of bytes of overlap is calculated and stored in i:

```
i = q->ti_seq + q->ti_len - ti->ti_seq;
  = 4 + 5 - 7
  = 2
```

If i is greater than 0, there is overlap, as we have in our example. If the number of bytes of overlap in the previous segment on the list (i) is greater than or equal to the size of the new segment, then all the data bytes in the new segment are already contained in the previous segment on the list. In this case the duplicate segment is discarded.

108–112 If there is only partial overlap (as there is in Figure 27.18), m_adj discards i bytes of data from the beginning of the new segment. The sequence number and length of the new segment are updated accordingly. q is moved to the next segment on the list. Figure 27.20 shows our example at this point.

116 The address of the mbuf m is stored in the TCP header, over the source and destination TCP ports. We mentioned earlier in this section that this provides a back pointer

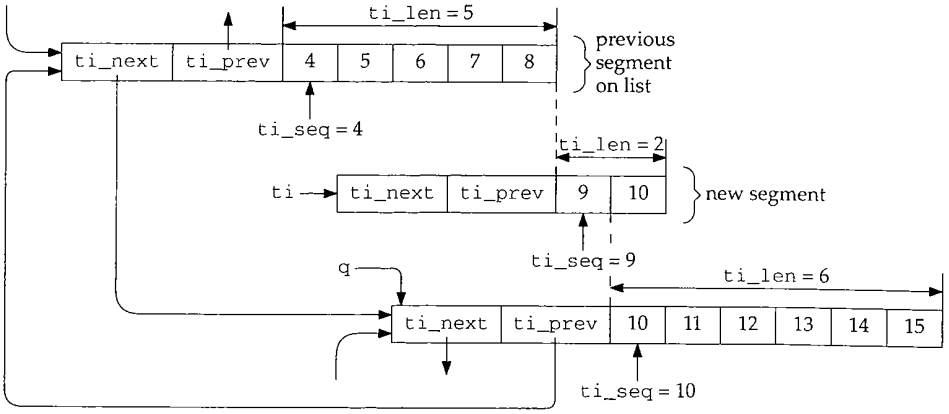


Figure 27.20 Update of Figure 27.18 after bytes 7 and 8 have been removed from new segment.

from the TCP header to the mbuf, in case the TCP header is stored in a cluster, meaning that the macro `dtom` won't work. The macro `REASS_MBUF` is

```
#define  REASS_MBUF(ti)  (*(struct mbuf **)&((ti)->ti_t))
```

`ti_t` is the `tcphdr` structure (Figure 24.12) and the first two members of the structure are the two 16-bit port numbers. The comment `XXX` in Figure 27.19 is because this hack assumes that a pointer fits in the 32 bits occupied by the two port numbers.

The third part of `tcp_reass` is shown in Figure 27.21. It removes any overlap from the next segment in the queue.

117–135 If there is another segment on the list, the number of bytes of overlap between the new segment and that segment is calculated in `i`. In our example we have

```
i = 9 + 2 - 10
  = 1
```

since byte number 10 overlaps the two segments.

Depending on the value of `i`, one of three conditions exists:

1. If `i` is less than or equal to 0, there is no overlap.
2. If `i` is less than the number of bytes in the next segment (`q->ti_len`), there is partial overlap and `m_adj` removes the first `i` bytes from the next segment on the list.
3. If `i` is greater than or equal to the number of bytes in the next segment, there is complete overlap and that next segment on the list is deleted.

136–139 The new segment is inserted into the reassembly list for this connection by `insque`. Figure 27.22 shows the state of our example at this point.

tcp_input.c

```
117     /*
118      * While we overlap succeeding segments trim them or,
119      * if they are completely covered, dequeue them.
120      */
121     while (q != (struct tcpiphdr *) tp) {
122         int     i = (ti->ti_seq + ti->ti_len) - q->ti_seq;
123         if (i <= 0)
124             break;
125         if (i < q->ti_len) {
126             q->ti_seq += i;
127             q->ti_len -= i;
128             m_adj(REASS_MBUF(q), i);
129             break;
130         }
131         q = (struct tcpiphdr *) q->ti_next;
132         m = REASS_MBUF((struct tcpiphdr *) q->ti_prev);
133         remque(q->ti_prev);
134         m_freem(m);
135     }

136     /*
137      * Stick new segment in its place.
138      */
139     insque(ti, q->ti_prev);
```

tcp_input.c

Figure 27.21 `tcp_reass` function: third part.

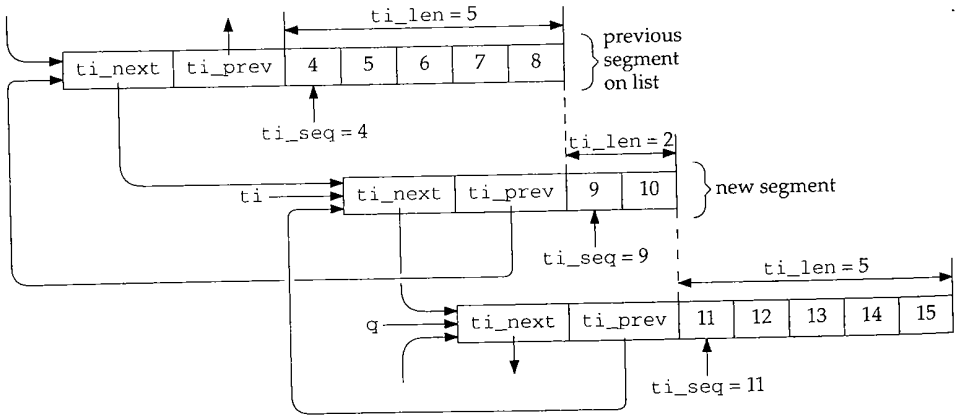


Figure 27.22 Update of Figure 27.20 after removal of all overlapping bytes.

Figure 27.23 shows the final part of `tcp_reass`. It passes the data to the process, if possible.

tcp_input.c

```
140   present:
141     /*
142      * Present data to user, advancing rcv_nxt through
143      * completed sequence space.
144      */
145     if (TCPS_HAVERCVDSYN(tp->t_state) == 0)
146         return (0);
147     ti = tp->seg_next;
148     if (ti == (struct tcpiphdr *) tp || ti->ti_seq != tp->rcv_nxt)
149         return (0);
150     if (tp->t_state == TCPS_SYN_RECEIVED && ti->ti_len)
151         return (0);
152     do {
153         tp->rcv_nxt += ti->ti_len;
154         flags = ti->ti_flags & TH_FIN;
155         remque(ti);
156         m = REASS_MBUF(ti);
157         ti = (struct tcpiphdr *) ti->ti_next;
158         if (so->so_state & SS_CANTRCVMORE)
159             m_freem(m);
160         else
161             sbappend(&so->so_rcv, m);
162     } while (ti != (struct tcpiphdr *) tp && ti->ti_seq == tp->rcv_nxt);
163     sorwakeup(so);
164     return (flags);
165 }
```

tcp_input.c

Figure 27.23 tcp_reass function: fourth part.

145–146 If the connection has not received a SYN (i.e., it is in the LISTEN or SYN_SENT state), data cannot be passed to the process and the function returns. When this function is called by TCP_REASS, the return value of 0 is stored in the flags argument to the macro. This can have the side effect of clearing the FIN flag. We'll see that this side effect is a possibility when TCP_REASS is invoked in Figure 29.22, and the received segment contains a SYN, FIN, and data (not a typical segment, but valid).

147–149 ti starts at the first segment on the list. If the list is empty, or if the starting sequence number of the first segment on the list (ti->ti_seq) does not equal the next receive sequence number (rcv_nxt), the function returns a value of 0. If the second condition is true, there is still a hole in the received data starting with the next expected sequence number. For instance, in our example (Figure 27.22), if the segment with bytes 4–8 is the first on the list but rcv_nxt equals 2, bytes 2 and 3 are still missing, so bytes 4–15 cannot be passed to the process. The return of 0 turns off the FIN flag (if set), because one or more data segments are still missing, so a received FIN cannot be processed yet.

150–151 If the state is SYN_RCVD and the length of the segment is nonzero, the function returns a value of 0. If both of these conditions are true, the socket is a listening socket that has received in-order data with the SYN. The data is left on the connection's queue, waiting for the three-way handshake to complete.

152–164 This loop starts with the first segment on the list (which is known to be in order) and appends it to the socket's receive buffer. `rcv_nxt` is incremented by the number of bytes in the segment. The loop stops when the list is empty or when the sequence number of the next segment on the list is out of order (i.e., there is a hole in the sequence space). When the loop terminates, the `flags` variable (which becomes the return value of the function) is 0 or `TH_FIN`, depending on whether the final segment placed in the socket's receive buffer has the FIN flag set or not.

After all the mbufs have been placed onto the socket's receive buffer, `sorwakeup` wakes any process waiting for data to be received on the socket.

27.10 `tcp_trace` Function

In `tcp_output`, before sending a segment to IP for output, we saw the following call to `tcp_trace` in Figure 26.32:

```
if (so->so_options & SO_DEBUG)
    tcp_trace(TA_OUTPUT, tp->t_state, tp, ti, 0);
```

This call adds a record to a circular buffer in the kernel that can be examined with the `trpt`(8) program. Additionally, if the kernel is compiled with `TCPDEBUG` defined, and if the variable `tcpconsdebug` is nonzero, information is output on the system console.

> Any process can set the `SO_DEBUG` socket option for a TCP socket, causing the information to be stored in the kernel's circular buffer. But `trpt` must read the kernel memory (`/dev/kmem`) to fetch this information, and this often requires special privileges.
>
> The `SO_DEBUG` socket option can be set for any type of socket (e.g., UDP or raw IP), but TCP is the only protocol that looks at the option.

The information saved by the kernel is a `tcp_debug` structure, shown in Figure 27.24.

tcp_debug.h

```
35 struct tcp_debug {
36     n_time  td_time;          /* iptime(): ms since midnight, UTC */
37     short   td_act;           /* TA_xxx value (Figure 27.25) */
38     short   td_ostate;        /* old state */
39     caddr_t td_tcb;           /* addr of TCP connection block */
40     struct tcpiphdr td_ti;    /* IP and TCP headers */
41     short   td_req;           /* PRU_xxx value for TA_USER */
42     struct tcpcb td_cb;       /* TCP connection block */
43 };

53 #define TCP_NDEBUG 100
54 struct tcp_debug tcp_debug[TCP_NDEBUG];
55 int     tcp_debx;
```

tcp_debug.h

Figure 27.24 `tcp_debug` structure.

35–43 This is a large structure (196 bytes), since it contains two other structures: the `tcpiphdr` structure with the IP and TCP headers; and the `tcpcb` structure, the entire TCP control block. Since the entire TCP control block is saved, any variable in the

control block can be printed by trpt. Also, if trpt doesn't print the variable we're interested in, we can modify the source code (it is available with the Net/3 release) to print whatever information we would like from the control block. The RTT variables in Figure 25.28 were obtained using this technique.

53–55 We also show the declaration of the array tcp_debug, which is used as the circular buffer. The index into the array (tcp_debx) is initialized to 0. This array occupies almost 20,000 bytes.

There are only four calls to tcp_trace in the kernel. Each call stores a different value in the td_act member of the structure, as shown in Figure 27.25.

td_act	Description	Reference
TA_DROP	from tcp_input, when input segment is dropped	Figure 29.27
TA_INPUT	after input processing complete, before call to tcp_output	Figure 29.26
TA_OUTPUT	before calling ip_output to send segment	Figure 26.32
TA_USER	from tcp_usrreq, after processing PRU_*xxx* request	Figure 30.1

Figure 27.25 td_act values and corresponding call to tcp_trace.

Figure 27.27 shows the main body of the tcp_trace function. We omit the code that outputs directly to the console.

48–133 ostate is the old state of the connection, when the function was called. By saving this value and the new state of the connection (which is in the control block) we can see the state transition that occurred. In Figure 27.25, TA_OUTPUT doesn't change the state of the connection, but the other three calls can change the state.

Sample Output

Figure 27.26 shows the first four lines of tcpdump output corresponding to the three-way handshake and the first data segment from the example in Section 25.12. (Appendix A of Volume 1 provides additional details on the tcpdump output format.)

```
1  0.0                    bsdi.1025 > vangogh.discard: S 20288001:20288001(0)
                                                       win 4096 <mss 512>
2  0.362719 (0.3627)      vangogh.discard > bsdi.1025: S 3202722817:3202722817(0)
                                                       ack 20288002 win 8192
                                                       <mss 512>
3  0.364316 (0.0016)      bsdi.1025 > vangogh.discard: . ack 1 win 4096
4  0.415859 (0.0515)      bsdi.1025 > vangogh.discard: . 1:513(512) ack 1 win 4096
```

Figure 27.26 tcpdump output from example in Figure 25.28.

Figure 27.28 shows the corresponding output from trpt.

This output contains a few changes from the normal trpt output. The 32-bit decimal sequence numbers are printed as unsigned values (trpt incorrectly prints them as signed numbers). Some values printed by trpt in hexadecimal have been output in decimal. The values from t_rtt through t_rxtcur were added to trpt by the authors, for Figure 25.28.

tcp_debug.c

```
48 void
49 tcp_trace(act, ostate, tp, ti, req)
50 short   act, ostate;
51 struct tcpcb *tp;
52 struct tcpiphdr *ti;
53 int     req;
54 {
55     tcp_seq seq, ack;
56     int     len, flags;
57     struct tcp_debug *td = &tcp_debug[tcp_debx++];

58     if (tcp_debx == TCP_NDEBUG)
59         tcp_debx = 0;           /* circle back to start */

60     td->td_time = iptime();
61     td->td_act = act;
62     td->td_ostate = ostate;
63     td->td_tcb = (caddr_t) tp;
64     if (tp)
65         td->td_cb = *tp;        /* structure assignment */
66     else
67         bzero((caddr_t) & td->td_cb, sizeof(*tp));
68     if (ti)
69         td->td_ti = *ti;        /* structure assignment */
70     else
71         bzero((caddr_t) & td->td_ti, sizeof(*ti));
72     td->td_req = req;

73 #ifdef TCPDEBUG
74     if (tcpconsdebug == 0)
75         return;

                /* output information on console */

132 #endif
133 }
```

tcp_debug.c

Figure 27.27 `tcp_trace` function: save information in kernel's circular buffer.

At time 953738 the SYN is sent. Notice that only the lower 6 digits of the millisecond time are output—it would take 8 digits to represent 1 minute before midnight. The ending sequence number that is output is wrong (20288005). Four bytes are sent with the SYN, but these are the MSS option, not data. The retransmit timer is 6 seconds (`REXMT`) and the keepalive timer is 75 seconds (`KEEP`). These timer values are in 500-ms ticks. The value of 1 for `t_rtt` means this segment is being timed for an RTT measurement.

This SYN segment is sent in response to the process calling `connect`. One millisecond later the trace record for this system call is added to the kernel's buffer. Even though the call to `connect` generates the SYN segment, since the call to `tcp_trace`

```
953738 SYN_SENT: output 20288001:20288005(4) @0 (win=4096)
       <SYN> -> SYN_SENT
       rcv_nxt 0, rcv_wnd 0
       snd_una 20288001, snd_nxt 20288002, snd_max 20288002
       snd_wl1 0, snd_wl2 0, snd_wnd 0
       REXMT=12 (t_rxtshift=0), KEEP=150
       t_rtt=1, t_srtt=0, t_rttvar=24, t_rxtcur=12

953739 CLOSED: user CONNECT -> SYN_SENT
       rcv_nxt 0, rcv_wnd 0
       snd_una 20288001, snd_nxt 20288002, snd_max 20288002
       snd_wl1 0, snd_wl2 0, snd_wnd 0
       REXMT=12 (t_rxtshift=0), KEEP=150
       t_rtt=1, t_srtt=0, t_rttvar=24, t_rxtcur=12

954103 SYN_SENT: input 3202722817:3202722817(0) @20288002 (win=8192)
       <SYN,ACK> -> ESTABLISHED
       rcv_nxt 3202722818, rcv_wnd 4096
       snd_una 20288002, snd_nxt 20288002, snd_max 20288002
       snd_wl1 3202722818, snd_wl2 20288002, snd_wnd 8192
       KEEP=14400
       t_rtt=0, t_srtt=16, t_rttvar=4, t_rxtcur=6

954103 ESTABLISHED: output 20288002:20288002(0) @3202722818 (win=4096)
       <ACK> -> ESTABLISHED
       rcv_nxt 3202722818, rcv_wnd 4096
       snd_una 20288002, snd_nxt 20288002, snd_max 20288002
       snd_wl1 3202722818, snd_wl2 20288002, snd_wnd 8192
       KEEP=14400
       t_rtt=0, t_srtt=16, t_rttvar=4, t_rxtcur=6

954153 ESTABLISHED: output 20288002:20288514(512) @3202722818 (win=4096)
       <ACK> -> ESTABLISHED
       rcv_nxt 3202722818, rcv_wnd 4096
       snd_una 20288002, snd_nxt 20288514, snd_max 20288514
       snd_wl1 3202722818, snd_wl2 20288002, snd_wnd 8192
       REXMT=6 (t_rxtshift=0), KEEP=14400
       t_rtt=1, t_srtt=16, t_rttvar=4, t_rxtcur=6
```

Figure 27.28 `trpt` output from example in Figure 25.28.

appears after processing the `PRU_CONNECT` request, the two trace records appear backward in the buffer. Also, when the process called `connect`, the connection state was CLOSED, and it changes to SYN_SENT. Nothing else changes from the first trace record to this one.

The third trace record, at time 954103, occurs 365 ms after the first. (`tcpdump` shows a 362.7 ms difference.) This is how the values in the column "actual delta (ms)" in Figure 25.28 were computed. The connection state changes from SYN_SENT to ESTABLISHED when the segment with a SYN and an ACK is received. The RTT estimators are updated because the segment being timed was acknowledged.

The fourth trace record is the third segment of the three-way handshake: the ACK of the other end's SYN. Since this segment contains no data, it is not timed (`rtt` is 0).

After the ACK has been sent at time 954103, the `connect` system call returns to the process, which then calls `write` to send data. This generates TCP output, shown in trace record 5 at time 954153, 50 ms after the three-way handshake is complete. 512 bytes of data are sent, starting with sequence number 20288002. The retransmission timer is set to 3 seconds and the segment is timed.

This output is caused by an application `write`. Although we don't show any more trace records, the next four are from `PRU_SEND` requests. The first `PRU_SEND` request generates the output of the first 512-byte segment that we show, but the other three do not cause output, since the connection has just started and is in slow start. Four trace records are generated because the system used for this example uses a TCP send buffer of 4096 and a cluster size of 1024. Once the send buffer is full, the process is put to sleep.

27.11 Summary

This chapter has covered a wide range of TCP functions that we'll encounter in the following chapters.

TCP connections can be aborted by sending an RST or they can be closed down gracefully, by sending a FIN and waiting for the four-way exchange of segments to complete.

Eight variables are stored in each routing table entry, three of which are updated when a connection is closed and six of which can be used later when a new connection is established. This lets the kernel keep track of certain variables, such as the RTT estimators and the slow start threshold, between successive connections to the same destination. The system administrator can also set and lock some of these variables, such as the MTU, receive pipe size, and send pipe size, that affect TCP connections to that destination.

TCP is tolerant of received ICMP errors—none cause Net/3 to terminate an established connection. This handling of ICMP errors by Net/3 differs from earlier Berkeley releases.

Received TCP segments can arrive out of order and can contain duplicate data, and TCP must handle these anomalies. We saw that a reassembly queue is maintained for each connection, and this holds the out-of-order segments along with segments that arrive before they can be passed to the application.

Finally we looked at the type of information saved by the kernel when the `SO_DEBUG` socket option is enabled for a TCP socket. This trace information can be a useful diagnostic tool in addition to programs such as `tcpdump`.

Exercises

27.1 Why is the `errno` value 0 for the last row in Figure 27.1?

27.2 What is the maximum value that can be stored in `rmx_rtt`?

27.3 To save the route information in Figure 27.3 for a given host, we enter a route into the routing table by hand for this destination. We then run the FTP client to send data to this host, making certain we send enough data, as described with Figure 27.4. But after terminating the FTP client we look at the routing table, and all the values for this host are still 0. What's happening?

28.1

28

TCP Input

28.1 Introduction

TCP input processing is the largest piece of code that we examine in this text. The function `tcp_input` is about 1100 lines of code. The processing of incoming segments is not complicated, just long and detailed. Many implementations, including the one in Net/3, closely follow the input event processing steps in RFC 793, which spell out in detail how to respond to the various input segments, based on the current state of the connection.

The `tcp_input` function is called by `ipintr` (through the `pr_input` function in the protocol switch table) when a datagram is received with a protocol field of TCP. `tcp_input` executes at the software interrupt level.

The function is so long that we divide its discussion into two chapters. Figure 28.1 outlines the processing steps in `tcp_input`. This chapter discusses the steps through RST processing, and the next chapter starts with ACK processing.

The first few steps are typical: validate the input segment (checksum, length, etc.) and locate the PCB for this connection. Given the length of the remainder of the function, however, an attempt is made to bypass all this logic with an algorithm called *header prediction* (Section 28.4). This algorithm is based on the assumption that segments are not typically lost or reordered, hence for a given connection TCP can often guess what the next received segment will be. If the header prediction algorithm works, notice that the function returns. This is the fast path through `tcp_input`.

The slow path through the function ends up at the label `dodata`, which tests a few flags and calls `tcp_output` if a segment should be sent in response to the received segment.

```
void
tcp_input()
{
    checksum TCP header and data;

findpcb:
    locate PCB for segment;
    if (not found)
        goto dropwithreset;

    reset idle time to 0 and keepalive timer to 2 hours;

    process options if not LISTEN state;

    if (packet matched by header prediction) {
        completely process received segment;
        return;
    }

    switch (tp->t_state) {
    case TCPS_LISTEN:
        if SYN flag set, accept new connection request;
        goto trimthenstep6;

    case TCPS_SYN_SENT:
        if ACK of our SYN, connection completed;
trimthenstep6:
        trim any data not within window;
        goto step6;
    }

    process RFC 1323 timestamp;

    check if some data bytes are within the receive window;

    trim data segment to fit within window;

    if (RST flag set) {
        process depending on state;
        goto drop;
    }                           /* Chapter 28 finishes here */

    if (ACK flag set) {         /* Chapter 29 starts here */
        if (SYN_RCVD state)
            passive open or simultaneous open complete;
        if (duplicate ACK)
            fast recovery algorithm;
        update RTT estimators if segment timed;
        open congestion window;
        remove ACKed data from send buffer;
        change state if in FIN_WAIT_1, CLOSING, or LAST_ACK state;
    }

step6:
    update window information;

    process URG flag;
```

```
dodata:
    process data in segment, add to reassembly queue;

    if (FIN flag is set)
        process depending on state;

    if (SO_DEBUG socket option)
        tcp_trace(TA_INPUT);

    if (need output || ACK now)
        tcp_output();
    return;

dropafterack:
    tcp_output() to generate ACK;
    return;

dropwithreset:
    tcp_respond() to generate RST;
    return;

drop:
    if (SO_DEBUG socket option)
        tcp_trace(TA_DROP);
    return;
}
```

Figure 28.1 Summary of TCP input processing steps.

There are also three labels at the end of the function that are jumped to when errors occur: `dropafterack`, `dropwithreset`, and `drop`. The term *drop* means to drop the segment being processed, not drop the connection, but when an RST is sent by `dropwithreset` it normally causes the connection to be dropped.

The only other branching in the function occurs when a valid SYN is received in either the LISTEN or SYN_SENT states, at the `switch` following header prediction. When the code at `trimthenstep6` finishes, it jumps to `step6`, which continues the normal flow.

28.2 Preliminary Processing

Figure 28.2 shows the declarations and the initial processing of the received TCP segment.

Get IP and TCP headers in first mbuf

170–204 The argument `iphlen` is the length of the IP header, including possible IP options. If the length is greater than 20 bytes, options are present, and `ip_stripoptions` discards the options. TCP ignores all IP options other than a source route, which is saved specially by IP (Section 9.6) and fetched later by TCP in Figure 28.7. If the number of bytes in the first mbuf in the chain is less than the size of the combined IP/TCP header (40 bytes), `m_pullup` moves the first 40 bytes into the first mbuf.

tcp_input.c

```
170 void
171 tcp_input(m, iphlen)
172 struct mbuf *m;
173 int     iphlen;
174 {
175     struct tcpiphdr *ti;
176     struct inpcb *inp;
177     caddr_t optp = NULL;
178     int     optlen;
179     int     len, tlen, off;
180     struct tcpcb *tp = 0;
181     int     tiflags;
182     struct socket *so;
183     int     todrop, acked, ourfinisacked, needoutput = 0;
184     short   ostate;
185     struct in_addr laddr;
186     int     dropsocket = 0;
187     int     iss = 0;
188     u_long  tiwin, ts_val, ts_ecr;
189     int     ts_present = 0;

190     tcpstat.tcps_rcvtotal++;
191     /*
192      * Get IP and TCP header together in first mbuf.
193      * Note: IP leaves IP header in first mbuf.
194      */
195     ti = mtod(m, struct tcpiphdr *);
196     if (iphlen > sizeof(struct ip))
197                 ip_stripoptions(m, (struct mbuf *) 0);
198     if (m->m_len < sizeof(struct tcpiphdr)) {
199         if ((m = m_pullup(m, sizeof(struct tcpiphdr))) == 0) {
200             tcpstat.tcps_rcvshort++;
201             return;
202         }
203         ti = mtod(m, struct tcpiphdr *);
204     }
```

tcp_input.c

Figure 28.2 `tcp_input` function: declarations and preliminary processing.

The next piece of code, shown in Figure 28.3, verifies the TCP checksum and offset field. *230–236*

Verify TCP checksum

205–217 `tlen` is the TCP length, the number of bytes following the IP header. Recall that IP has already subtracted the IP header length from `ip_len`. The variable `len` is then set to the length of the IP datagram, the number of bytes to be checksummed, including the pseudo-header. The fields in the pseudo-header are set, as required for the checksum calculation, as shown in Figure 23.19.

Verify TCP offset field

218–228 The TCP offset field, `ti_off`, is the number of 32-bit words in the TCP header, including any TCP options. It is multiplied by 4 (to become the byte offset of the first

tcp_input.c

```
205     /*
206      * Checksum extended TCP header and data.
207      */
208     tlen = ((struct ip *) ti)->ip_len;
209     len = sizeof(struct ip) + tlen;
210     ti->ti_next = ti->ti_prev = 0;
211     ti->ti_x1 = 0;
212     ti->ti_len = (u_short) tlen;
213     HTONS(ti->ti_len);
214     if (ti->ti_sum = in_cksum(m, len)) {
215         tcpstat.tcps_rcvbadsum++;
216         goto drop;
217     }
218     /*
219      * Check that TCP offset makes sense,
220      * pull out TCP options and adjust length.       XXX
221      */
222     off = ti->ti_off << 2;
223     if (off < sizeof(struct tcphdr) || off > tlen) {
224         tcpstat.tcps_rcvbadoff++;
225         goto drop;
226     }
227     tlen -= off;
228     ti->ti_len = tlen;
```

tcp_input.c

Figure 28.3 `tcp_input` function: verify TCP checksum and offset field.

data byte in the TCP segment) and checked for sanity. It must be greater than or equal to the size of the standard TCP header (20) and less than or equal to the TCP length.

The byte offset of the first data byte is subtracted from the TCP length, leaving `tlen` with the number of bytes of data in the segment (possibly 0). This value is stored back into the TCP header, in the variable `ti_len`, and will be used throughout the function.

Figure 28.4 shows the next part of processing: handling of certain TCP options.

Get headers plus option into first mbuf

230–236 If the byte offset of the first data byte is greater than 20, TCP options are present. `m_pullup` is called, if necessary, to place the standard IP header, standard TCP header, and any TCP options in the first mbuf in the chain. Since the maximum size of these three pieces is 80 bytes (20 + 20 + 40), they all fit into the first packet header mbuf on the chain.

> Since the only way `m_pullup` can fail here is when fewer than 20 plus `off` bytes are in the IP datagram, and since the TCP checksum has already been verified, we expect this call to `m_pullup` never to fail. Unfortunately the counter `tcps_rcvshort` is also shared by the call to `m_pullup` in Figure 28.2, so looking at the counter doesn't tell us which call failed. Nevertheless, Figure 24.5 shows that after receiving almost 9 million TCP segments, this counter is 0.

tcp_input.c

```
229     if (off > sizeof(struct tcphdr)) {
230         if (m->m_len < sizeof(struct ip) + off) {
231             if ((m = m_pullup(m, sizeof(struct ip) + off)) == 0) {
232                 tcpstat.tcps_rcvshort++;
233                 return;
234             }
235             ti = mtod(m, struct tcpiphdr *);
236         }
237         optlen = off - sizeof(struct tcphdr);
238         optp = mtod(m, caddr_t) + sizeof(struct tcpiphdr);
239         /*
240          * Do quick retrieval of timestamp options ("options
241          * prediction?").  If timestamp is the only option and it's
242          * formatted as recommended in RFC 1323 Appendix A, we
243          * quickly get the values now and not bother calling
244          * tcp_dooptions(), etc.
245          */
246         if ((optlen == TCPOLEN_TSTAMP_APPA ||
247              (optlen > TCPOLEN_TSTAMP_APPA &&
248               optp[TCPOLEN_TSTAMP_APPA] == TCPOPT_EOL)) &&
249             *(u_long *) optp == htonl(TCPOPT_TSTAMP_HDR) &&
250             (ti->ti_flags & TH_SYN) == 0) {
251             ts_present = 1;
252             ts_val = ntohl(*(u_long *) (optp + 4));
253             ts_ecr = ntohl(*(u_long *) (optp + 8));
254             optp = NULL;        /* we've parsed the options */
255         }
256     }
```

tcp_input.c

Figure 28.4 `tcp_input` function: handle certain TCP options.

Process timestamp option quickly

237–255 `optlen` is the number of bytes of options, and `optp` is a pointer to the first option byte. If the following three conditions are all true, only the timestamp option is present and it is in the desired format:

1. (a) The TCP option length equals 12 (`TCPOLEN_TSTAMP_APPA`), or (b) the TCP option length is greater than 12 and `optp[12]` equals the end-of-option byte.
2. The first 4 bytes of options equals `0x0101080a` (`TCPOPT_TSTAMP_HDR`, which we described in Section 26.6).
3. The SYN flag is not set (i.e., this segment is for an established connection, hence if a timestamp option is present, we know both sides have agreed to use the option).

If all three conditions are true, `ts_present` is set to 1; the two timestamp values are fetched and stored in `ts_val` and `ts_ecr`; and `optp` is set to null, since all the options have been parsed. The benefit in recognizing the timestamp option this way is to avoid calling the general option processing function `tcp_dooptions` later in the code. The general option processing function is OK for the other options that appear only with the

257–264

265–279

SYN segment that creates a connection (the MSS and window scale options), but when the timestamp option is being used, it will appear with almost every segment on an established connection, so the faster it can be recognized, the better.

The next piece of code, shown in Figure 28.5, locates the Internet PCB for the segment.

tcp_input.c

```
257     tiflags = ti->ti_flags;

258     /*
259      * Convert TCP protocol specific fields to host format.
260      */
261     NTOHL(ti->ti_seq);
262     NTOHL(ti->ti_ack);
263     NTOHS(ti->ti_win);
264     NTOHS(ti->ti_urp);

265     /*
266      * Locate pcb for segment.
267      */
268   findpcb:
269     inp = tcp_last_inpcb;
270     if (inp->inp_lport != ti->ti_dport ||
271         inp->inp_fport != ti->ti_sport ||
272         inp->inp_faddr.s_addr != ti->ti_src.s_addr ||
273         inp->inp_laddr.s_addr != ti->ti_dst.s_addr) {
274         inp = in_pcblookup(&tcb, ti->ti_src, ti->ti_sport,
275                            ti->ti_dst, ti->ti_dport, INPLOOKUP_WILDCARD);
276         if (inp)
277             tcp_last_inpcb = inp;
278         ++tcpstat.tcps_pcbcachemiss;
279     }
```

tcp_input.c

Figure 28.5 `tcp_input` function: locate Internet PCB for segment.

Save input flags and convert fields to host byte order

257–264 The received flags (SYN, FIN, etc.) are saved in the local variable `tiflags`, since they are referenced throughout the code. Two 16-bit values and the two 32-bit values in the TCP header are converted from network byte order to host byte order. The two 16-bit port numbers are left in network byte order, since the port numbers in the Internet PCB are in that order.

Locate Internet PCB

265–279 TCP maintains a one-behind cache (`tcp_last_inpcb`) containing the address of the PCB for the last received TCP segment. This is the same technique used by UDP. The comparison of the four elements in the socket pair is in the same order as done by `udp_input`. If the cache entry does not match, `in_pcblookup` is called, and the cache is set to the new PCB entry.

TCP does not have the same problem that we encountered with UDP: wildcard entries in the cache causing a high miss rate. The only time a TCP socket has a wildcard entry is for a server listening for connection requests. Once a connection is made, all

four entries in the socket pair contain nonwildcard values. In Figure 24.5 we see a cache hit rate of almost 80%.

Figure 28.6 shows the next piece of code.

tcp_input.c

```
280     /*
281      * If the state is CLOSED (i.e., TCB does not exist) then
282      * all data in the incoming segment is discarded.
283      * If the TCB exists but is in CLOSED state, it is embryonic,
284      * but should either do a listen or a connect soon.
285      */
286     if (inp == 0)
287         goto dropwithreset;
288     tp = intotcpcb(inp);
289     if (tp == 0)
290         goto dropwithreset;
291     if (tp->t_state == TCPS_CLOSED)
292         goto drop;
293     /* Unscale the window into a 32-bit value. */
294     if ((tiflags & TH_SYN) == 0)
295         tiwin = ti->ti_win << tp->snd_scale;
296     else
297         tiwin = ti->ti_win;
```

tcp_input.c

Figure 28.6 `tcp_input` function: check if segment should be dropped.

Drop segment and generate RST

280–287 If the PCB was not found, the input segment is dropped and an RST is sent as a reply. This is how TCP handles SYNs that arrive for a server that doesn't exist, for example. Recall that UDP sends an ICMP port unreachable in this case.

288–290 If the PCB exists but a corresponding TCP control block does not exist, the socket is probably being closed (`tcp_close` releases the TCP control block first, and then releases the PCB), so the input segment is dropped and an RST is sent as a reply.

Silently drop segment

291–292 If the TCP control block exists, but the connection state is CLOSED, the socket has been created and a local address and local port may have been assigned, but neither `connect` nor `listen` has been called. The segment is dropped but nothing is sent as a reply. This scenario can happen if a client catches a server between the server's call to `bind` and `listen`. By silently dropping the segment and not replying with an RST, the client's connection request should time out, causing the client to retransmit the SYN.

Unscale advertised window

293–297 If window scaling is to take place for this connection, both ends must specify their send scale factor using the window scale option when the connection is established. If the segment contains a SYN, the window scale factor has not been established yet, so `tiwin` is copied from the value in the TCP header. Otherwise the 16-bit value in the header is left shifted by the send scale factor into a 32-bit value.

The next piece of code, shown in Figure 28.7, does some preliminary processing if the socket debug option is enabled or if the socket is listening for incoming connection requests.

tcp_input.c

```
298     so = inp->inp_socket;
299     if (so->so_options & (SO_DEBUG | SO_ACCEPTCONN)) {
300         if (so->so_options & SO_DEBUG) {
301             ostate = tp->t_state;
302             tcp_saveti = *ti;
303         }
304         if (so->so_options & SO_ACCEPTCONN) {
305             so = sonewconn(so, 0);
306             if (so == 0)
307                 goto drop;
308             /*
309              * This is ugly, but ....
310              *
311              * Mark socket as temporary until we're
312              * committed to keeping it.  The code at
313              * 'drop' and 'dropwithreset' check the
314              * flag dropsocket to see if the temporary
315              * socket created here should be discarded.
316              * We mark the socket as discardable until
317              * we're committed to it below in TCPS_LISTEN.
318              */
319             dropsocket++;
320             inp = (struct inpcb *) so->so_pcb;
321             inp->inp_laddr = ti->ti_dst;
322             inp->inp_lport = ti->ti_dport;
323 #if BSD>=43
324             inp->inp_options = ip_srcroute();
325 #endif
326             tp = intotcpcb(inp);
327             tp->t_state = TCPS_LISTEN;

328             /* Compute proper scaling value from buffer space */
329             while (tp->request_r_scale < TCP_MAX_WINSHIFT &&
330                    TCP_MAXWIN << tp->request_r_scale < so->so_rcv.sb_hiwat)
331                 tp->request_r_scale++;
332         }
333     }
```

tcp_input.c

Figure 28.7 `tcp_input` function: handle debug option and listening sockets.

Save connection state and IP/TCP headers if socket debug option enabled

300–303 If the `SO_DEBUG` socket option is enabled the current connection state is saved (`ostate`) as well as the IP and TCP headers (`tcp_saveti`). These become arguments to `tcp_trace` when it is called at the end of the function (Figure 29.26).

Create new socket if segment arrives for listening socket

304–319 When a segment arrives for a listening socket (`SO_ACCEPTCONN` is enabled by `listen`), a new socket is created by `sonewconn`. This issues the protocol's

PRU_ATTACH request (Figure 30.2), which allocates an Internet PCB and a TCP control block. But more processing is needed before TCP commits to accept the connection request (such as the fundamental question of whether the segment contains a SYN or not), so the flag dropsocket is set, to cause the code at the labels drop and dropwithreset to discard the new socket if an error is encountered. If the received segment is OK, dropsocket is set back to 0 in Figure 28.17.

320–326 inp and tp point to the new socket that has been created. The local address and local port are copied from the destination address and destination port of the IP and TCP headers. If the input datagram contained a source route, it was saved by save_rte. TCP calls ip_srcroute to fetch that source route, saving a pointer to the mbuf containing the source route option in inp_options. This option is passed to ip_output by tcp_output, and the reverse route is used for datagrams sent on this connection.

327 The state of the new socket is set to LISTEN. If the received segment contains a SYN, the code in Figure 28.16 completes the connection request.

Compute window scale factor

328–331 The window scale factor that will be requested is calculated from the size of the receive buffer. 65535 (TCP_MAXWIN) is left shifted until the result exceeds the size of the receive buffer, or until the maximum window scale factor is encountered (14, TCP_MAX_WINSHIFT). Notice that the requested window scale factor is chosen based on the size of the listening socket's receive buffer. This means the process must set the SO_RCVBUF socket option before listening for incoming connection requests or it inherits the default value in tcp_recvspace.

> The maximum scale factor is 14, and 65535×2^{14} is 1,073,725,440. This is far greater than the maximum size of the receive buffer (262,144 in Net/3), so the loop should always terminate with a scale factor much less than 14. See Exercises 28.1 and 28.2.

Figure 28.8 shows the next part of TCP input processing.

tcp_input.c

```
334     /*
335      * Segment received on connection.
336      * Reset idle time and keepalive timer.
337      */
338     tp->t_idle = 0;
339     tp->t_timer[TCPT_KEEP] = tcp_keepidle;

340     /*
341      * Process options if not in LISTEN state,
342      * else do it below (after getting remote address).
343      */
344     if (optp && tp->t_state != TCPS_LISTEN)
345         tcp_dooptions(tp, optp, optlen, ti,
346                       &ts_present, &ts_val, &ts_ecr);
```

tcp_input.c

Figure 28.8 tcp_input function: reset idle time and keepalive timer, process options.

Reset idle time and keepalive timer

334–339 `t_idle` is set to 0 since a segment has been received on the connection. The keepalive timer is also reset to 2 hours.

Process TCP options if not in LISTEN state

340–346 If options are present in the TCP header, and if the connection state is not LISTEN, `tcp_dooptions` processes the options. Recall that if only a timestamp option appears for an established connection, and that option is in the format recommended by Appendix A of RFC 1323, it was already processed in Figure 28.4 and `optp` was set to a null pointer. If the socket is in the LISTEN state, `tcp_dooptions` is called in Figure 28.17 after the peer's address has been recorded in the PCB, because processing the MSS option requires knowledge of the route that will be used to this peer.

28.3 tcp_dooptions Function

This function processes the five TCP options supported by Net/3 (Section 26.4): the EOL, NOP, MSS, window scale, and timestamp options. Figure 28.9 shows the first part of this function.

tcp_input.c

```
1213 void
1214 tcp_dooptions(tp, cp, cnt, ti, ts_present, ts_val, ts_ecr)
1215 struct tcpcb *tp;
1216 u_char *cp;
1217 int     cnt;
1218 struct tcpiphdr *ti;
1219 int    *ts_present;
1220 u_long *ts_val, *ts_ecr;
1221 {
1222     u_short mss;
1223     int     opt, optlen;

1224     for (; cnt > 0; cnt -= optlen, cp += optlen) {
1225         opt = cp[0];
1226         if (opt == TCPOPT_EOL)
1227             break;
1228         if (opt == TCPOPT_NOP)
1229             optlen = 1;
1230         else {
1231             optlen = cp[1];
1232             if (optlen <= 0)
1233                 break;
1234         }
1235         switch (opt) {

1236         default:
1237             continue;
```

tcp_input.c

Figure 28.9 `tcp_dooptions` function: handle EOL and NOP options.

Fetch option type and length

1213–1229 The options are scanned and an EOL (end-of-options) terminates the processing, causing the function to return. The length of a NOP is set to 1, since this option is not followed by a length byte (Figure 26.16). The NOP will be ignored via the `default` in the `switch` statement.

1230–1234 All other options have a length byte that is stored in `optlen`.

Any new options that are not understood by this implementation of TCP are also ignored. This occurs because:

1. Any new options defined in the future will have an option length (NOP and EOL are the only two without a length), and the `for` loop skips `optlen` bytes each time around the loop.
2. The `default` in the `switch` statement ignores unknown options.

The final part of `tcp_dooptions`, shown in Figure 28.10, handles the MSS, window scale, and timestamp options.

MSS option

1238–1246 If the length is not 4 (`TCPOLEN_MAXSEG`), or the segment does not have the SYN flag set, the option is ignored. Otherwise the 2 MSS bytes are copied into a local variable, converted to host byte order, and processed by `tcp_mss`. This has the side effect of setting the variable `t_maxseg` in the control block, the maximum number of bytes that can be sent in a segment to the other end.

Window scale option

1247–1254 If the length is not 3 (`TCPOLEN_WINDOW`), or the segment does not have the SYN flag set, the option is ignored. Net/3 remembers that it received a window scale request, and the scale factor is saved in `requested_s_scale`. Since only 1 byte is referenced by `cp[2]`, there can't be alignment problems. When the ESTABLISHED state is entered, if both ends requested window scaling, it is enabled.

Timestamp option

1255–1273 If the length is not 10 (`TCPOLEN_TIMESTAMP`), the segment is ignored. Otherwise the flag pointed to by `ts_present` is set to 1, and the two timestamps are saved in the variables pointed to by `ts_val` and `ts_ecr`. If the received segment contains the SYN flag, Net/3 remembers that a timestamp request was received. `ts_recent` is set to the received timestamp and `ts_recent_age` is set to `tcp_now`, the counter of the number of 500-ms clock ticks since the system was initialized.

28.4 Header Prediction

We now continue with the code in `tcp_input`, from where we left off in Figure 28.8.

Header prediction was put into the 4.3BSD Reno release by Van Jacobson. The only description of the algorithm, other than the source code we're about to examine, is in [Jacobson 1990b], which is a copy of three slides showing the code.

tcp_input.c

```
1238            case TCPOPT_MAXSEG:
1239                if (optlen != TCPOLEN_MAXSEG)
1240                    continue;
1241                if (!(ti->ti_flags & TH_SYN))
1242                    continue;
1243                bcopy((char *) cp + 2, (char *) &mss, sizeof(mss));
1244                NTOHS(mss);
1245                (void) tcp_mss(tp, mss);    /* sets t_maxseg */
1246                break;

1247            case TCPOPT_WINDOW:
1248                if (optlen != TCPOLEN_WINDOW)
1249                    continue;
1250                if (!(ti->ti_flags & TH_SYN))
1251                    continue;
1252                tp->t_flags |= TF_RCVD_SCALE;
1253                tp->requested_s_scale = min(cp[2], TCP_MAX_WINSHIFT);
1254                break;

1255            case TCPOPT_TIMESTAMP:
1256                if (optlen != TCPOLEN_TIMESTAMP)
1257                    continue;
1258                *ts_present = 1;
1259                bcopy((char *) cp + 2, (char *) ts_val, sizeof(*ts_val));
1260                NTOHL(*ts_val);
1261                bcopy((char *) cp + 6, (char *) ts_ecr, sizeof(*ts_ecr));
1262                NTOHL(*ts_ecr);

1263                /*
1264                 * A timestamp received in a SYN makes
1265                 * it ok to send timestamp requests and replies.
1266                 */
1267                if (ti->ti_flags & TH_SYN) {
1268                    tp->t_flags |= TF_RCVD_TSTMP;
1269                    tp->ts_recent = *ts_val;
1270                    tp->ts_recent_age = tcp_now;
1271                }
1272                break;
1273            }
1274        }
1275    }
```

tcp_input.c

Figure 28.10 `tcp_dooptions` function: process MSS, window scale, and timestamp options.

Header prediction helps unidirectional data transfer by handling the two common cases.

1. If TCP is sending data, the next expected segment for this connection is an ACK for outstanding data.
2. If TCP is receiving data, the next expected segment for this connection is the next in-sequence data segment.

In both cases a small set of tests determines if the next expected segment has been received, and if so, it is handled in-line, faster than the general processing that follows later in this chapter and the next.

[Partridge 1993] shows an even faster version of TCP header prediction from a research implementation developed by Van Jacobson.

Figure 28.11 shows the first part of header prediction.

tcp_input.c

```
347     /*
348      * Header prediction: check for the two common cases
349      * of a uni-directional data xfer.  If the packet has
350      * no control flags, is in-sequence, the window didn't
351      * change and we're not retransmitting, it's a
352      * candidate.  If the length is zero and the ack moved
353      * forward, we're the sender side of the xfer.  Just
354      * free the data acked & wake any higher-level process
355      * that was blocked waiting for space.  If the length
356      * is non-zero and the ack didn't move, we're the
357      * receiver side.  If we're getting packets in order
358      * (the reassembly queue is empty), add the data to
359      * the socket buffer and note that we need a delayed ack.
360      */
361     if (tp->t_state == TCPS_ESTABLISHED &&
362      (tiflags & (TH_SYN | TH_FIN | TH_RST | TH_URG | TH_ACK)) == TH_ACK &&
363         (!ts_present || TSTMP_GEQ(ts_val, tp->ts_recent)) &&
364         ti->ti_seq == tp->rcv_nxt &&
365         tiwin && tiwin == tp->snd_wnd &&
366         tp->snd_nxt == tp->snd_max) {

367         /*
368          * If last ACK falls within this segment's sequence numbers,
369          *  record the timestamp.
370          */
371         if (ts_present && SEQ_LEQ(ti->ti_seq, tp->last_ack_sent) &&
372             SEQ_LT(tp->last_ack_sent, ti->ti_seq + ti->ti_len)) {
373             tp->ts_recent_age = tcp_now;
374             tp->ts_recent = ts_val;                                    367–3
375         }
```

tcp_input.c

Figure 28.11 `tcp_input` function: header prediction, first part.

Check if segment is the next expected

347–366 The following six conditions must *all* be true for the segment to be the next expected data segment or the next expected ACK:

1. The connection state must be ESTABLISHED.
2. The following four control flags must not be on: SYN, FIN, RST, or URG. The ACK flag must be on. In other words, of the six TCP control flags, the ACK flag must be set, the four just listed must be cleared, and it doesn't matter whether 376–3

PSH is set or cleared. (Normally in the ESTABLISHED state the ACK flag is always on unless the RST flag is on.)

3. If the segment contains a timestamp option, the timestamp value from the other end (`ts_val`) must be greater than or equal to the previous timestamp received for this connection (`ts_recent`). This is basically the PAWS test, which we describe in detail in Section 28.7. If `ts_val` is less than `ts_recent`, this segment is out of order because it was sent before the most previous segment received on this connection. Since the other end always sends its timestamp clock (the global variable `tcp_now` in Net/3) as its timestamp value, the received timestamps of in-order segments always form a monotonic increasing sequence.

   The timestamp need not increase with every in-order segment. Indeed, on a Net/3 system that increments the timestamp clock (`tcp_now`) every 500 ms, multiple segments are often sent on a connection before that clock is incremented. Think of the timestamp and sequence number as forming a 64-bit value, with the sequence number in the low-order 32 bits and the timestamp in the high-order 32 bits. This 64-bit value always increases by at least 1 for every in-order segment (taking into account the modulo arithmetic).

4. The starting sequence number of the segment (`ti_seq`) must equal the next expected receive sequence number (`rcv_nxt`). If this test is false, then the received segment is either a retransmission or a segment beyond the one expected.

5. The window advertised by the segment (`tiwin`) must be nonzero, and must equal the current send window (`snd_wnd`). This means the window has not changed.

6. The next sequence number to send (`snd_nxt`) must equal the highest sequence number sent (`snd_max`). This means the last segment sent by TCP was not a retransmission.

Update `ts_recent` from received timestamp

367–375 If a timestamp option is present and if its value passes the test described with Figure 26.18, the received timestamp (`ts_val`) is saved in `ts_recent`. Also, the current time (`tcp_now`) is recorded in `ts_recent_age`.

> Recall our discussion with Figure 26.18 on how this test for a valid timestamp is flawed, and the correct test presented in Figure 26.20. In this header prediction code the `TSTMP_GEQ` test in Figure 26.20 is redundant, since it was already done as step 3 of the `if` test at the beginning of Figure 28.11.

The next part of the header prediction code, shown in Figure 28.12, is for the sender of unidirectional data: process an ACK for outstanding data.

Test for pure ACK

376–379 If the following four conditions are all true, this segment is a pure ACK.

tcp_input.c

```
376         if (ti->ti_len == 0) {
377             if (SEQ_GT(ti->ti_ack, tp->snd_una) &&
378                 SEQ_LEQ(ti->ti_ack, tp->snd_max) &&
379                 tp->snd_cwnd >= tp->snd_wnd) {
380                 /*
381                  * this is a pure ack for outstanding data.
382                  */
383                 ++tcpstat.tcps_predack;
384                 if (ts_present)
385                     tcp_xmit_timer(tp, tcp_now - ts_ecr + 1);
386                 else if (tp->t_rtt &&
387                         SEQ_GT(ti->ti_ack, tp->t_rtseq))
388                     tcp_xmit_timer(tp, tp->t_rtt);
389                 acked = ti->ti_ack - tp->snd_una;
390                 tcpstat.tcps_rcvackpack++;
391                 tcpstat.tcps_rcvackbyte += acked;
392                 sbdrop(&so->so_snd, acked);
393                 tp->snd_una = ti->ti_ack;
394                 m_freem(m);
395                 /*
396                  * If all outstanding data is acked, stop
397                  * retransmit timer, otherwise restart timer
398                  * using current (possibly backed-off) value.
399                  * If process is waiting for space,
400                  * wakeup/selwakeup/signal.  If data
401                  * is ready to send, let tcp_output
402                  * decide between more output or persist.
403                  */
404                 if (tp->snd_una == tp->snd_max)
405                     tp->t_timer[TCPT_REXMT] = 0;
406                 else if (tp->t_timer[TCPT_PERSIST] == 0)
407                     tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;
408                 if (so->so_snd.sb_flags & SB_NOTIFY)
409                     sowwakeup(so);
410                 if (so->so_snd.sb_cc)
411                     (void) tcp_output(tp);
412                 return;
413             }
```

tcp_input.c

Figure 28.12 `tcp_input` function: header prediction, sender processing.

1. The segment contains no data (`ti_len` is 0).
2. The acknowledgment field in the segment (`ti_ack`) is greater than the largest unacknowledged sequence number (`snd_una`). Since this test is "greater than" and not "greater than or equal to," it is true only if some positive amount of data is acknowledged by the ACK.
3. The acknowledgment field in the segment (`ti_ack`) is less than or equal to the maximum sequence number sent (`snd_max`).

4. The congestion window (`snd_cwnd`) is greater than or equal to the current send window (`snd_wnd`). This test is true only if the window is fully open, that is, the connection is not in the middle of slow start or congestion avoidance.

Update RTT estimators

364–388 If the segment contains a timestamp option, or if a segment was being timed and the acknowledgment field is greater than the starting sequence number being timed, `tcp_xmit_timer` updates the RTT estimators.

Delete acknowledged bytes from send buffer

389–394 `acked` is the number of bytes acknowledged by the segment. `sbdrop` deletes those bytes from the send buffer. The largest unacknowledged sequence number (`snd_una`) is set to the acknowledgment field and the received mbuf chain is released. (Since the length is 0, there should be just a single mbuf containing the headers.)

Stop retransmit timer

395–407 If the received segment acknowledges all outstanding data (`snd_una` equals `snd_max`), the retransmission timer is turned off. Otherwise, if the persist timer is off, the retransmit timer is restarted using `t_rxtcur` as the timeout.

Recall that when `tcp_output` sends a segment, it sets the retransmit timer only if the timer is not currently enabled. If two segments are sent one right after the other, the timer is set when the first is sent, but not touched when the second is sent. But if an ACK is received only for the first segment, the retransmit timer must be restarted, in case the second was lost.

Awaken waiting processes

408–409 If a process must be awakened when the send buffer is modified, `sowwakeup` is called. From Figure 16.5, `SB_NOTIFY` is true if a process is waiting for space in the buffer, if a process is `select`ing on the buffer, or if a process wants the `SIGIO` signal for this socket.

Generate more output

410–411 If there is data in the send buffer, `tcp_output` is called because the sender's window has moved to the right. `snd_una` was just incremented and `snd_wnd` did not change, so in Figure 24.17 the entire window has shifted to the right.

The next part of header prediction, shown in Figure 28.13, is the receiver processing when the segment is the next in-sequence data segment.

Test for next in-sequence data segment

414–416 If the following four conditions are all true, this segment is the next expected data segment for the connection, and there is room in the socket buffer for the data.

1. The amount of data in the segment (`ti_len`) is greater than 0. This is the `else` portion of the `if` at the beginning of Figure 28.12.
2. The acknowledgment field (`ti_ack`) equals the largest unacknowledged sequence number. This means no data is acknowledged by this segment.

tcp_input.c

```
414         } else if (ti->ti_ack == tp->snd_una &&
415                    tp->seg_next == (struct tcpiphdr *) tp &&
416                    ti->ti_len <= sbspace(&so->so_rcv)) {
417             /*
418              * this is a pure, in-sequence data packet
419              * with nothing on the reassembly queue and
420              * we have enough buffer space to take it.
421              */
422             ++tcpstat.tcps_preddat;
423             tp->rcv_nxt += ti->ti_len;
424             tcpstat.tcps_rcvpack++;
425             tcpstat.tcps_rcvbyte += ti->ti_len;
426             /*
427              * Drop TCP, IP headers and TCP options then add data
428              * to socket buffer.
429              */
430             m->m_data += sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);
431             m->m_len -= sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);
432             sbappend(&so->so_rcv, m);
433             sorwakeup(so);
434             tp->t_flags |= TF_DELACK;
435             return;
436         }
437     }
```

tcp_input.c

Figure 28.13 `tcp_input` function: header prediction, receiver processing.

3. The reassembly list of out-of-order segments for the connection is empty (`seg_next` equals `tp`).
4. There is room in the receive buffer for the data in the segment.

Complete processing of received data

423–435 The next expected receive sequence number (`rcv_nxt`) is incremented by the number of bytes of data. The IP header, TCP header, and any TCP options are dropped from the mbuf, and the mbuf chain is appended to the socket's receive buffer. The receiving process is awakened by `sorwakeup`. Notice that this code avoids calling the `TCP_REASS` macro, since the tests performed by that macro have already been performed by the header prediction tests. The delayed-ACK flag is set and the input processing is complete.

Statistics

How useful is header prediction? A few simple unidirectional transfers were run across a LAN (between `bsdi` and `svr4`, in both directions) and across a WAN (between `vangogh.cs.berkeley.edu` and `ftp.uu.net` in both directions). The `netstat` output (Figure 24.5) shows the two header prediction counters.

On the LAN, with no packet loss but a few duplicate ACKs, header prediction worked between 97 and 100% of the time. Across the WAN, however, the header prediction percentages dropped slightly to between 83 and 99%.

Realize that header prediction works on a per-connection basis, regardless how much additional TCP traffic is being received by the host, while the PCB cache works on a per-host basis. Even though lots of TCP traffic can cause PCB cache misses, if packets are not lost on a given connection, header prediction still works on that connection.

28.5 TCP Input: Slow Path Processing

We continue with the code that's executed if header prediction fails, the slow path through `tcp_input`. Figure 28.14 shows the next piece of code, which prepares the received segment for input processing.

tcp_input.c

```
438     /*
439      * Drop TCP, IP headers and TCP options.
440      */
441     m->m_data += sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);
442     m->m_len -= sizeof(struct tcpiphdr) + off - sizeof(struct tcphdr);

443     /*
444      * Calculate amount of space in receive window,
445      * and then do TCP input processing.
446      * Receive window is amount of space in rcv queue,
447      * but not less than advertised window.
448      */
449     {
450         int     win;

451         win = sbspace(&so->so_rcv);
452         if (win < 0)
453             win = 0;
454         tp->rcv_wnd = max(win, (int) (tp->rcv_adv - tp->rcv_nxt));
455     }
```

tcp_input.c

Figure 28.14 `tcp_input` function: drop IP and TCP headers.

Drop IP and TCP headers, including TCP options

438–442 The data pointer and length of the first mbuf in the chain are updated to skip over the IP header, TCP header, and any TCP options. Since `off` is the number of bytes in the TCP header, including options, the size of the normal TCP header (20) must be subtracted from the expression.

Calculate receive window

443–455 `win` is set to the number of bytes available in the socket's receive buffer. `rcv_adv` minus `rcv_nxt` is the current advertised window. The receive window is the maximum of these two values. The `max` is taken to ensure that the value is not less than the currently advertised window. Also, if the process has taken data out of the socket

receive buffer since the window was last advertised, `win` could exceed the advertised window, so TCP accepts up to `win` bytes of data (even though the other end should not be sending more than the advertised window).

This value is calculated now, since the code later in this function must determine how much of the received data (if any) fits within the advertised window. Any received data outside the advertised window is dropped: data to the left of the window is duplicate data that has already been received and acknowledged, and data to the right should not be sent by the other end.

28.6 Initiation of Passive Open, Completion of Active Open

If the state is LISTEN or SYN_SENT, the code shown in this section is executed. The expected segment in these two states is a SYN, and we'll see that any other received segment is dropped.

Initiation of Passive Open

Figure 28.15 shows the processing when the connection is in the LISTEN state. In this code the variables `tp` and `inp` refer to the *new* socket that was created in Figure 28.7, not the server's listening socket.

tcp_input.c

```
456     switch (tp->t_state) {

457         /*
458          * If the state is LISTEN then ignore segment if it contains an RST.
459          * If the segment contains an ACK then it is bad and send an RST.
460          * If it does not contain a SYN then it is not interesting; drop it.
461          * Don't bother responding if the destination was a broadcast.
462          * Otherwise initialize tp->rcv_nxt, and tp->irs, select an initial
463          * tp->iss, and send a segment:
464          *     <SEQ=ISS><ACK=RCV_NXT><CTL=SYN,ACK>
465          * Also initialize tp->snd_nxt to tp->iss+1 and tp->snd_una to tp->iss.
466          * Fill in remote peer address fields if not previously specified.
467          * Enter SYN_RECEIVED state, and process any other fields of this
468          * segment in this state.
469          */
470     case TCPS_LISTEN:{
471             struct mbuf *am;
472             struct sockaddr_in *sin;

473             if (tiflags & TH_RST)
474                 goto drop;
475             if (tiflags & TH_ACK)
476                 goto dropwithreset;
477             if ((tiflags & TH_SYN) == 0)
478                 goto drop;
```

tcp_input.c

Figure 28.15 `tcp_input` function: check if SYN received for listening socket.

Drop if RST, ACK, or no SYN

473–478 If the received segment contains the RST flag, it is dropped. If it contains an ACK, it is dropped and an RST is sent as the reply. (The initial SYN to open a connection is one of the few segments that does not contain an ACK.) If the SYN flag is not set, the segment is dropped. The remaining code for this `case` handles the reception of a SYN for a connection in the LISTEN state. The new state will be SYN_RCVD.

Figure 28.16 shows the next piece of code for this `case`.

tcp_input.c

```
479             /*
480              * RFC1122 4.2.3.10, p. 104: discard bcast/mcast SYN
481              * in_broadcast() should never return true on a received
482              * packet with M_BCAST not set.
483              */
484             if (m->m_flags & (M_BCAST | M_MCAST) ||
485                 IN_MULTICAST(ti->ti_dst.s_addr))
486                 goto drop;
487             am = m_get(M_DONTWAIT, MT_SONAME);  /* XXX */
488             if (am == NULL)
489                 goto drop;
490             am->m_len = sizeof(struct sockaddr_in);
491             sin = mtod(am, struct sockaddr_in *);
492             sin->sin_family = AF_INET;
493             sin->sin_len = sizeof(*sin);
494             sin->sin_addr = ti->ti_src;
495             sin->sin_port = ti->ti_sport;
496             bzero((caddr_t) sin->sin_zero, sizeof(sin->sin_zero));
497             laddr = inp->inp_laddr;
498             if (inp->inp_laddr.s_addr == INADDR_ANY)
499                 inp->inp_laddr = ti->ti_dst;
500             if (in_pcbconnect(inp, am)) {
501                 inp->inp_laddr = laddr;
502                 (void) m_free(am);
503                 goto drop;
504             }
505             (void) m_free(am);
```

tcp_input.c

Figure 28.16 `tcp_input` function: process SYN for listening socket.

Drop if broadcast or multicast

479–486 If the packet was sent to a broadcast or multicast address, it is dropped. TCP is defined only for unicast applications. Recall that the `M_BCAST` and `M_MCAST` flags were set by `ether_input`, based on the destination hardware address of the frame. The `IN_MULTICAST` macro tests whether the IP address is a class D address.

The comment reference to `in_broadcast` is because the Net/1 code (which did not support multicasting) called that function here, to check whether the destination IP address was a broadcast address. The setting of the `M_BCAST` and `M_MCAST` flags by `ether_input`, based on the destination hardware address, was introduced with Net/2.

This Net/3 code tests only whether the destination hardware address is a broadcast address, and does not call `in_broadcast` to test whether the destination IP address is a broadcast address, on the assumption that a packet should never be received with a destination IP address that is a broadcast address unless the packet was sent to the hardware broadcast address. This assumption is made to avoid calling `in_broadcast`. Nevertheless, if a Net/3 system receives a SYN destined for a broadcast IP address but a unicast hardware address, that segment will be processed by the code in Figure 28.16.

The destination address argument to `IN_MULTICAST` needs to be converted to host byte order.

Get mbuf for client's IP address and port

487–496 An mbuf is allocated to hold a `sockaddr_in` structure, and the structure is filled in with the client's IP address and port number. The IP address is copied from the source address in the IP header and the port number is copied from the source port number in the TCP header. This structure is used shortly to connect the server's PCB to the client, and then the mbuf is released.

The `XXX` comment is probably because of the cost associated with obtaining an mbuf just for the call to `in_pcbconnect` that follows. But this is the slow processing path for TCP input. Figure 24.5 shows that less than 2% of all received segments execute this code.

Set local address in PCB

497–499 `laddr` is the local address bound to the socket. If the server bound the wildcard address to the socket (the normal scenario), the destination address from the IP header becomes the local address in the PCB. Note that the destination address from the IP header is used, regardless of which local interface the datagram was received on.

Notice that `laddr` cannot be the wildcard address, because in Figure 28.7 it is explicitly set to the destination IP address from the received datagram.

Connect PCB to peer

500–505 `in_pcbconnect` connects the server's PCB to the client. This fills in the foreign address and foreign process in the PCB. The mbuf is then released. *515–51*

The next piece of code, shown in Figure 28.17 completes the processing for this `case`.

Allocate and initialize IP and TCP header template

506–511 A template of the IP and TCP headers is created by `tcp_template`. The call to `sonewconn` in Figure 28.7 allocated the PCB and TCP control block for the new connection, but not the header template.

Process any TCP options

512–514 If TCP options are present, they are processed by `tcp_dooptions`. The call to this function in Figure 28.8 was done only if the connection was not in the LISTEN state. This function is called now for a listening socket, after the foreign address is set in the PCB, since the foreign address is used by the `tcp_mss` function: to get a route to the peer, and to check if the peer is "local" or "foreign" (with regard to the peer's network ID and subnet ID, used to select the MSS).