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

```
    /*
     * Fill in fields, remembering maximum advertised
     * window for use in delaying messages about window sizes.
     * If resending a FIN, be sure not to use a new sequence number.
     */
    if (flags & TH_FIN && tp->t_flags & TF_SENTFIN &&
        tp->snd_nxt == tp->snd_max)
        tp->snd_nxt--;
    /*
     * If we are doing retransmissions, then snd_nxt will
     * not reflect the first unsent octet.  For ACK only
     * packets, we do not want the sequence number of the
     * retransmitted packet, we want the sequence number
     * of the next unsent octet.  So, if there is no data
     * (and no SYN or FIN), use snd_max instead of snd_nxt
     * when filling in ti_seq.  But if we are in persist
     * state, snd_max might reflect one byte beyond the
     * right edge of the window, so use snd_nxt in that
     * case, since we know we aren't doing a retransmission.
     * (retransmit and persist are mutually exclusive...)
     */
    if (len || (flags & (TH_SYN | TH_FIN)) || tp->t_timer[TCPT_PERSIST])
        ti->ti_seq = htonl(tp->snd_nxt);
    else
        ti->ti_seq = htonl(tp->snd_max);

    ti->ti_ack = htonl(tp->rcv_nxt);

    if (optlen) {
        bcopy((caddr_t) opt, (caddr_t) (ti + 1), optlen);
        ti->ti_off = (sizeof(struct tcphdr) + optlen) >> 2;
    }
    ti->ti_flags = flags;
```

tcp_output.c

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

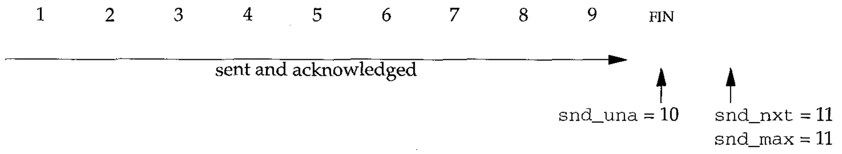


Figure 26.28 Send sequence space after FIN has been transmitted.

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

```
/*
 * Calculate receive window.  Don't shrink window,
 * but avoid silly window syndrome.
 */
if (win < (long) (so->so_rcv.sb_hiwat / 4) && win < (long) tp->t_maxseg)
    win = 0;
if (win > (long) TCP_MAXWIN << tp->rcv_scale)
    win = (long) TCP_MAXWIN << tp->rcv_scale;
if (win < (long) (tp->rcv_adv - tp->rcv_nxt))
    win = (long) (tp->rcv_adv - tp->rcv_nxt);
ti->ti_win = htons((u_short) (win >> tp->rcv_scale));

if (SEQ_GT(tp->snd_up, tp->snd_nxt)) {
    ti->ti_urp = htons((u_short) (tp->snd_up - tp->snd_nxt));
    ti->ti_flags |= TH_URG;
} else
    /*
     * If no urgent pointer to send, then we pull
     * the urgent pointer to the left edge of the send window
     * so that it doesn't drift into the send window on sequence
     * number wraparound.
     */
    tp->snd_up = tp->snd_una;   /* drag it along */

/*
 * Put TCP length in extended header, and then
 * checksum extended header and data.
 */
if (len + optlen)
    ti->ti_len = htons((u_short) (sizeof(struct tcphdr) +
                                  optlen + len));
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.

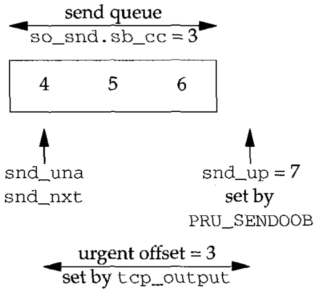


Figure 26.30 Example of urgent pointer and urgent offset calculation.

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

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

        /*
         * Advance snd_nxt over sequence space of this segment.
         */
        if (flags & (TH_SYN | TH_FIN)) {
            if (flags & TH_SYN)
                tp->snd_nxt++;
            if (flags & TH_FIN) {
                tp->snd_nxt++;
                tp->t_flags |= TF_SENTFIN;
            }
        }
        tp->snd_nxt += len;
        if (SEQ_GT(tp->snd_nxt, tp->snd_max)) {
            tp->snd_max = tp->snd_nxt;
            /*
             * Time this transmission if not a retransmission and
             * not currently timing anything.
             */
            if (tp->t_rtt == 0) {
                tp->t_rtt = 1;
                tp->t_rtseq = startseq;
                tcpstat.tcps_segstimed++;
            }
        }
        /*
         * Set retransmit timer if not currently set,
         * and not doing an ack or a keepalive probe.
         * Initial value for retransmit timer is smoothed
         * round-trip time + 2 * round-trip time variance.
         * Initialize  counter which is used for backoff
         * of retransmit time.
         */
        if (tp->t_timer[TCPT_REXMT] == 0 &&
            tp->snd_nxt != tp->snd_una) {
            tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;
            if (tp->t_timer[TCPT_PERSIST]) {
                tp->t_timer[TCPT_PERSIST] = 0;
                tp->t_rxtshift = 0;
            }
        }
    } else if (SEQ_GT(tp->snd_nxt + len, tp->snd_max))
        tp->snd_max = tp->snd_nxt + len;
```

tcp_output.c

Figure 26.31 tcp_output function: fill in remaining fields in TCP header and calculate checksum.

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

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

    /*
     * Fill in IP length and desired time to live and
     * send to IP level.  There should be a better way
     * to handle ttl and tos; we could keep them in
     * the template, but need a way to checksum without them.
     */
    m->m_pkthdr.len = hdrlen + len;
    ((struct ip *) ti)->ip_len = m->m_pkthdr.len;
    ((struct ip *) ti)->ip_ttl = tp->t_inpcb->inp_ip.ip_ttl;    /* XXX */
    ((struct ip *) ti)->ip_tos = tp->t_inpcb->inp_ip.ip_tos;    /* XXX */
    error = ip_output(m, tp->t_inpcb->inp_options, &tp->t_inpcb->inp_route,
                      so->so_options & SO_DONTROUTE, 0);
    if (error) {
      out:
        if (error == ENOBUFS) {
            tcp_quench(tp->t_inpcb, 0);
            return (0);
        }
        if ((error == EHOSTUNREACH || error == ENETDOWN)
            && TCPS_HAVERCVDSYN(tp->t_state)) {
            tp->t_softerror = error;
            return (0);
        }
        return (error);
    }
    tcpstat.tcps_sndtotal++;

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

    if (sendalot)
        goto again;
    return (0);
}
```

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

```
struct tcpcb *
tcp_drop(tp, errno)
struct tcpcb *tp;
int     errno;
{
    struct socket *so = tp->t_inpcb->inp_socket;

    if (TCPS_HAVERCVDSYN(tp->t_state)) {
        tp->t_state = TCPS_CLOSED;
        (void) tcp_output(tp);
        tcpstat.tcps_drops++;
    } else
        tcpstat.tcps_conndrops++;
    if (errno == ETIMEDOUT && tp->t_softerror)
        errno = tp->t_softerror;
    so->so_error = errno;
    return (tcp_close(tp));
}
```

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

```
struct tcpcb *
tcp_close(tp)
struct tcpcb *tp;
{
    struct tcpiphdr *t;
    struct inpcb *inp = tp->t_inpcb;
    struct socket *so = inp->inp_socket;
    struct mbuf *m;
    struct rtentry *rt;

    /*
     * If we sent enough data to get some meaningful characteristics,
     * save them in the routing entry.  'Enough' is arbitrarily
     * defined as the sendpipesize (default 8K) * 16.  This would
     * give us 16 rtt samples assuming we only get one sample per
     * window (the usual case on a long haul net).  16 samples is
     * enough for the srtt filter to converge to within 5% of the correct
     * value; fewer samples and we could save a very bogus rtt.
     *
     * Don't update the default route's characteristics and don't
     * update anything that the user "locked".
     */
    if (SEQ_LT(tp->iss + so->so_snd.sb_hiwat * 16, tp->snd_max) &&
        (rt = inp->inp_route.ro_rt) &&
     ((struct sockaddr_in *) rt_key(rt))->sin_addr.s_addr != INADDR_ANY) {
        u_long  i;

        if ((rt->rt_rmx.rmx_locks & RTV_RTT) == 0) {
            i = tp->t_srtt *
                (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTT_SCALE));
            if (rt->rt_rmx.rmx_rtt && i)
                /*
                 * filter this update to half the old & half
                 * the new values, converting scale.
                 * See route.h and tcp_var.h for a
                 * description of the scaling constants.
                 */
                rt->rt_rmx.rmx_rtt =
                    (rt->rt_rmx.rmx_rtt + i) / 2;
            else
                rt->rt_rmx.rmx_rtt = i;
        }
        if ((rt->rt_rmx.rmx_locks & RTV_RTTVAR) == 0) {
            i = tp->t_rttvar *
                (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTTVAR_SCALE));
            if (rt->rt_rmx.rmx_rttvar && i)
                rt->rt_rmx.rmx_rttvar =
                    (rt->rt_rmx.rmx_rttvar + i) / 2;
            else
                rt->rt_rmx.rmx_rttvar = i;
        }
```

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

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.

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

$$\begin{aligned} RTO &= \frac{\texttt{t\_srtt}}{8} + 2 \times \frac{\texttt{t\_rttvar}}{4} \\ &= \frac{\dfrac{\texttt{t\_srtt}}{4} + \texttt{t\_rttvar}}{2} \end{aligned}$$

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

```
    /*
     * The current mss, t_maxseg, was initialized to the default value
     * of 512 (tcp_mssdflt) by tcp_newtcpcb().
     * If we compute a smaller value, reduce the current mss.
     * If we compute a larger value, return it for use in sending
     * a max seg size option, but don't store it for use
     * unless we received an offer at least that large from peer.
     * However, do not accept offers under 32 bytes.
     */
    if (offer)
        mss = min(mss, offer);
    mss = max(mss, 32);         /* sanity */
    if (mss < tp->t_maxseg || offer != 0) {
        /*
         * If there's a pipesize, change the socket buffer
         * to that size.  Make the socket buffers an integral
         * number of mss units; if the mss is larger than
         * the socket buffer, decrease the mss.
         */
        if ((bufsize = rt->rt_rmx.rmx_sendpipe) == 0)
            bufsize = so->so_snd.sb_hiwat;
        if (bufsize < mss)
            mss = bufsize;
        else {
            bufsize = roundup(bufsize, mss);
            if (bufsize > sb_max)
                bufsize = sb_max;
            (void) sbreserve(&so->so_snd, bufsize);
        }
        tp->t_maxseg = mss;

        if ((bufsize = rt->rt_rmx.rmx_recvpipe) == 0)
            bufsize = so->so_rcv.sb_hiwat;
        if (bufsize > mss) {
            bufsize = roundup(bufsize, mss);
            if (bufsize > sb_max)
                bufsize = sb_max;
            (void) sbreserve(&so->so_rcv, bufsize);
        }
    }
    tp->snd_cwnd = mss;
    if (rt->rt_rmx.rmx_ssthresh) {
        /*
         * There's some sort of gateway or interface
         * buffer limit on the path.  Use this to set
         * the slow start threshhold, but set the
         * threshold to no less than 2*mss.
         */
        tp->snd_ssthresh = max(2 * mss, rt->rt_rmx.rmx_ssthresh);
    }
    return (mss);
}
```

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

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`.

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

```
void
tcp_quench(inp, errno)
struct inpcb *inp;
int     errno;
{
    struct tcpcb *tp = intotcpcb(inp);

    if (tp)
        tp->snd_cwnd = tp->t_maxseg;
}
```

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`

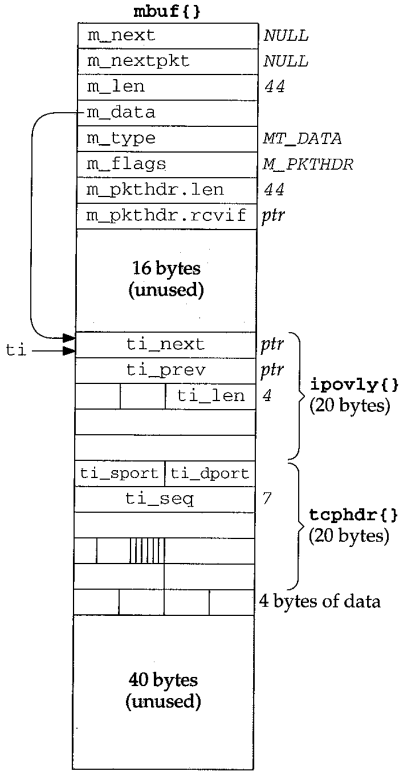


Figure 27.14 Example mbuf with IP and TCP headers and 4 bytes of data.

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 `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.

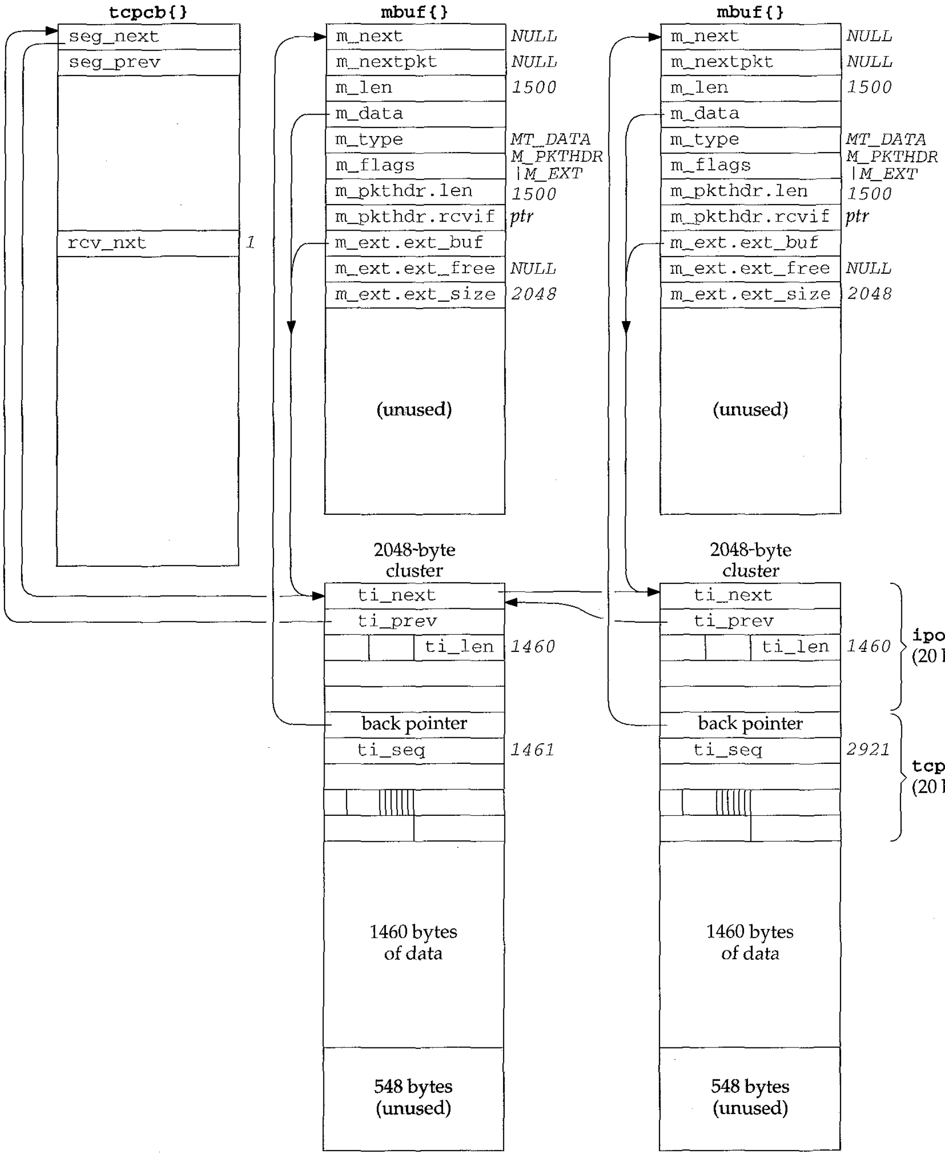


Figure 27.16 Two out-of-order TCP segments stored in mbuf clusters.

tcp_input.c

```
int
tcp_reass(tp, ti, m)
struct tcpcb *tp;
struct tcpiphdr *ti;
struct mbuf *m;
{
    struct tcpiphdr *q;
    struct socket *so = tp->t_inpcb->inp_socket;
    int     flags;

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

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

tcp_input.c

Figure 27.17 `tcp_reass` function: first part.

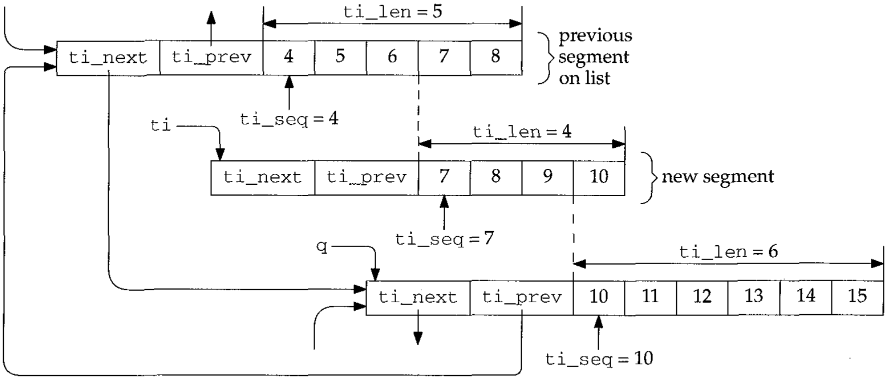


Figure 27.18 Example of TCP reassembly queue with overlapping segments.

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

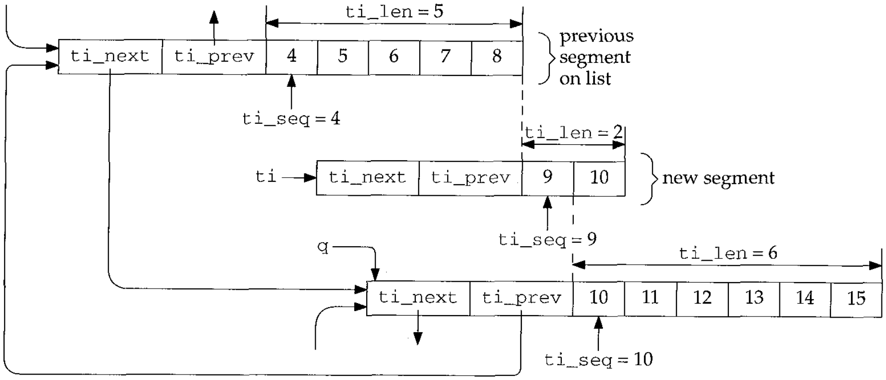


Figure 27.20 Update of Figure 27.18 after bytes 7 and 8 have been removed from new segment.

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

```
    /*
     * While we overlap succeeding segments trim them or,
     * if they are completely covered, dequeue them.
     */
    while (q != (struct tcpiphdr *) tp) {
        int     i = (ti->ti_seq + ti->ti_len) - q->ti_seq;
        if (i <= 0)
            break;
        if (i < q->ti_len) {
            q->ti_seq += i;
            q->ti_len -= i;
            m_adj(REASS_MBUF(q), i);
            break;
        }
        q = (struct tcpiphdr *) q->ti_next;
        m = REASS_MBUF((struct tcpiphdr *) q->ti_prev);
        remque(q->ti_prev);
        m_freem(m);
    }

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

tcp_input.c

Figure 27.21 `tcp_reass` function: third part.

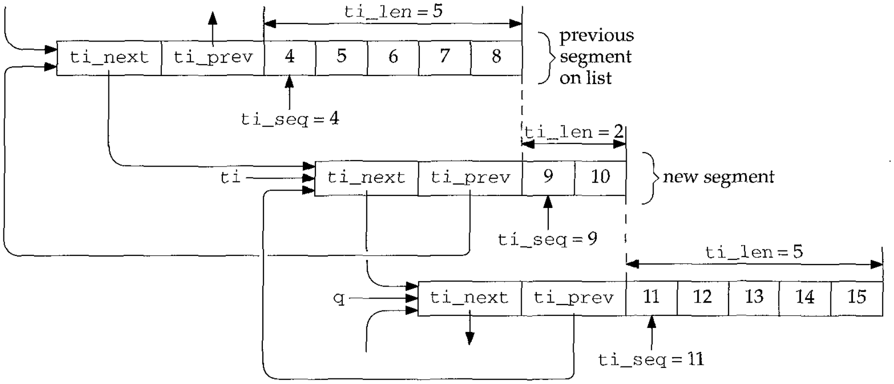


Figure 27.22 Update of Figure 27.20 after removal of all overlapping bytes.

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

```
void
tcp_trace(act, ostate, tp, ti, req)
short   act, ostate;
struct tcpcb *tp;
struct tcpiphdr *ti;
int     req;
{
    tcp_seq seq, ack;
    int     len, flags;
    struct tcp_debug *td = &tcp_debug[tcp_debx++];

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

    td->td_time = iptime();
    td->td_act = act;
    td->td_ostate = ostate;
    td->td_tcb = (caddr_t) tp;
    if (tp)
        td->td_cb = *tp;        /* structure assignment */
    else
        bzero((caddr_t) & td->td_cb, sizeof(*tp));
    if (ti)
        td->td_ti = *ti;        /* structure assignment */
    else
        bzero((caddr_t) & td->td_ti, sizeof(*ti));
    td->td_req = req;

#ifdef TCPDEBUG
    if (tcpconsdebug == 0)
        return;

                  /* output information on console */

#endif
}
```

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

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.

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

```
    /*
     * Checksum extended TCP header and data.
     */
    tlen = ((struct ip *) ti)->ip_len;
    len = sizeof(struct ip) + tlen;
    ti->ti_next = ti->ti_prev = 0;
    ti->ti_x1 = 0;
    ti->ti_len = (u_short) tlen;
    HTONS(ti->ti_len);
    if (ti->ti_sum = in_cksum(m, len)) {
        tcpstat.tcps_rcvbadsum++;
        goto drop;
    }
    /*
     * Check that TCP offset makes sense,
     * pull out TCP options and adjust length.      XXX
     */
    off = ti->ti_off << 2;
    if (off < sizeof(struct tcphdr) || off > tlen) {
        tcpstat.tcps_rcvbadoff++;
        goto drop;
    }
    tlen -= off;
    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

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

```
    /*
     * Segment received on connection.
     * Reset idle time and keepalive timer.
     */
    tp->t_idle = 0;
    tp->t_timer[TCPT_KEEP] = tcp_keepidle;

    /*
     * Process options if not in LISTEN state,
     * else do it below (after getting remote address).
     */
    if (optp && tp->t_state != TCPS_LISTEN)
        tcp_dooptions(tp, optp, optlen, ti,
                      &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

```
void
tcp_dooptions(tp, cp, cnt, ti, ts_present, ts_val, ts_ecr)
struct tcpcb *tp;
u_char *cp;
int     cnt;
struct tcpiphdr *ti;
int    *ts_present;
u_long *ts_val, *ts_ecr;
{
    u_short mss;
    int     opt, optlen;

    for (; cnt > 0; cnt -= optlen, cp += optlen) {
        opt = cp[0];
        if (opt == TCPOPT_EOL)
            break;
        if (opt == TCPOPT_NOP)
            optlen = 1;
        else {
            optlen = cp[1];
            if (optlen <= 0)
                break;
        }
        switch (opt) {

        default:
            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

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

        case TCPOPT_WINDOW:
            if (optlen != TCPOLEN_WINDOW)
                continue;
            if (!(ti->ti_flags & TH_SYN))
                continue;
            tp->t_flags |= TF_RCVD_SCALE;
            tp->requested_s_scale = min(cp[2], TCP_MAX_WINSHIFT);
            break;

        case TCPOPT_TIMESTAMP:
            if (optlen != TCPOLEN_TIMESTAMP)
                continue;
            *ts_present = 1;
            bcopy((char *) cp + 2, (char *) ts_val, sizeof(*ts_val));
            NTOHL(*ts_val);
            bcopy((char *) cp + 6, (char *) ts_ecr, sizeof(*ts_ecr));
            NTOHL(*ts_ecr);

            /*
             * A timestamp received in a SYN makes
             * it ok to send timestamp requests and replies.
             */
            if (ti->ti_flags & TH_SYN) {
                tp->t_flags |= TF_RCVD_TSTMP;
                tp->ts_recent = *ts_val;
                tp->ts_recent_age = tcp_now;
            }
            break;
        }
    }
}
```

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;
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

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

```
if (ti->ti_len == 0) {
    if (SEQ_GT(ti->ti_ack, tp->snd_una) &&
        SEQ_LEQ(ti->ti_ack, tp->snd_max) &&
        tp->snd_cwnd >= tp->snd_wnd) {
        /*
         * this is a pure ack for outstanding data.
         */
        ++tcpstat.tcps_predack;
        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);

        acked = ti->ti_ack - tp->snd_una;
        tcpstat.tcps_rcvackpack++;
        tcpstat.tcps_rcvackbyte += acked;
        sbdrop(&so->so_snd, acked);
        tp->snd_una = ti->ti_ack;
        m_freem(m);

        /*
         * If all outstanding data is acked, stop
         * retransmit timer, otherwise restart timer
         * using current (possibly backed-off) value.
         * If process is waiting for space,
         * wakeup/selwakeup/signal.  If data
         * is ready to send, let tcp_output
         * decide between more output or persist.
         */
        if (tp->snd_una == tp->snd_max)
            tp->t_timer[TCPT_REXMT] = 0;
        else if (tp->t_timer[TCPT_PERSIST] == 0)
            tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;

        if (so->so_snd.sb_flags & SB_NOTIFY)
            sowwakeup(so);
        if (so->so_snd.sb_cc)
            (void) tcp_output(tp);
        return;
    }
```

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

384–388 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

```
switch (tp->t_state) {

    /*
     * If the state is LISTEN then ignore segment if it contains an RST.
     * If the segment contains an ACK then it is bad and send an RST.
     * If it does not contain a SYN then it is not interesting; drop it.
     * Don't bother responding if the destination was a broadcast.
     * Otherwise initialize tp->rcv_nxt, and tp->irs, select an initial
     * tp->iss, and send a segment:
     *     <SEQ=ISS><ACK=RCV_NXT><CTL=SYN,ACK>
     * Also initialize tp->snd_nxt to tp->iss+1 and tp->snd_una to tp->iss.
     * Fill in remote peer address fields if not previously specified.
     * Enter SYN_RECEIVED state, and process any other fields of this
     * segment in this state.
     */
case TCPS_LISTEN:{
        struct mbuf *am;
        struct sockaddr_in *sin;

        if (tiflags & TH_RST)
            goto drop;
        if (tiflags & TH_ACK)
            goto dropwithreset;
        if ((tiflags & TH_SYN) == 0)
            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.

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).

tcp_input.c

```
506             tp->t_template = tcp_template(tp);
507             if (tp->t_template == 0) {
508                 tp = tcp_drop(tp, ENOBUFS);
509                 dropsocket = 0; /* socket is already gone */
510                 goto drop;
511             }
512             if (optp)
513                 tcp_dooptions(tp, optp, optlen, ti,
514                               &ts_present, &ts_val, &ts_ecr);
515             if (iss)
516                 tp->iss = iss;
517             else
518                 tp->iss = tcp_iss;
519             tcp_iss += TCP_ISSINCR / 2;
520             tp->irs = ti->ti_seq;
521             tcp_sendseqinit(tp);
522             tcp_rcvseqinit(tp);
523             tp->t_flags |= TF_ACKNOW;
524             tp->t_state = TCPS_SYN_RECEIVED;
525             tp->t_timer[TCPT_KEEP] = TCPTV_KEEP_INIT;
526             dropsocket = 0;     /* committed to socket */
527             tcpstat.tcps_accepts++;
528             goto trimthenstep6;
529         }
```

tcp_input.c

Figure 28.17 `tcp_input` function: complete processing of SYN received in LISTEN state.

Initialize ISS

515–519 The initial send sequence number is normally copied from the global `tcp_iss`, which is then incremented by 64,000 (`TCP_ISSINCR` divided by 2). If the local variable `iss` is nonzero, however, its value is used instead of `tcp_iss` to initialize the send sequence number for the connection.

The local `iss` variable is used for the following scenario.

- A server is started on port 27 on the host with an IP address of 128.1.2.3.
- A client on host 192.3.4.5 establishes a connection with this server. The client's ephemeral port is 3000. The socket pair on the server is {128.1.2.3, 27, 192.3.4.5, 3000}.
- The server actively closes the connection, putting this socket pair into the TIME_WAIT state. While the connection is in this state, the last receive sequence number is remembered in the TCP control block. Assume its value is 100,000.
- Before this connection leaves the TIME_WAIT state, a new SYN is received from the same port on the same client host (192.3.4.5, port 3000), which locates the PCB corresponding to the connection in the TIME_WAIT state, not the PCB for the listening server. Assume the sequence number of this new SYN is 200,000.

- Since this connection does not correspond to a listening socket in the LISTEN state, the code we just looked at is not executed. Instead, the code in Figure 28.29 is executed, and we'll see that it contains the following logic: if the sequence number of the new SYN (200,000) is greater than the last sequence number received from this client (100,000), then (1) the local variable `iss` is set to 100,000 plus 128,000, (2) the connection in the TIME_WAIT state is completely closed (its PCB and TCP control block are deleted), and (3) a jump is made to `findpcb` (Figure 28.5).
- This time the server's listening PCB will be located (assuming the listening server is still running), causing the code in this section to be executed. The local variable `iss` (now 228,000) is used in Figure 28.17 to initialize `tcp_iss` for the new connection.

This logic, which is allowed by RFC 1122, lets the same client and server reuse the same socket pair as long as the server does the active close. This also explains why the global variable `tcp_iss` is incremented by 64,000 each time any process issues a `connect` (Figure 30.4): to ensure that if a single client reopens the same connection with the same server repeatedly, a larger ISS is used each time, even if no data was transferred on the previous connection, and even if the 500-ms timer (which increments `tcp_iss`) has not expired since the last connection.

Initialize sequence number variables in control block

520–522 In Figure 28.17, the initial receive sequence number (`irs`) is copied from the sequence number in the SYN segment. The following two macros initialize the appropriate variables in the TCP control block:

```
#define  tcp_rcvseqinit(tp) \
    (tp)->rcv_adv = (tp)->rcv_nxt = (tp)->irs + 1

#define  tcp_sendseqinit(tp) \
    (tp)->snd_una = (tp)->snd_nxt = (tp)->snd_max = (tp)->snd_up = \
        (tp)->iss
```

The addition of 1 in the first macro is because the SYN occupies a sequence number.

ACK the SYN and change state

523–525 The `TF_ACKNOW` flag is set since the ACK of a SYN is not delayed. The connection state becomes SYN_RCVD, and the connection-establishment timer is set to 75 seconds (`TCPTV_KEEP_INIT`). Since the `TF_ACKNOW` flag is set, at the bottom of this function `tcp_output` will be called. Looking at Figure 24.16 we see that `tcp_outflags` will cause a segment with the SYN and ACK flags to be sent.

526–528 TCP is now committed to the new socket created in Figure 28.7, so the `dropsocket` flag is cleared. The code at `trimthenstep6` is jumped to, to complete processing of the SYN segment. Remember that a SYN segment can contain data, although the data cannot be passed to the application until the connection enters the ESTABLISHED state.

Completion of Active Open

Figure 28.18 shows the first part of processing when the connection is in the SYN_SENT state. TCP is expecting to receive a SYN.

tcp_input.c

```
530         /*
531          * If the state is SYN_SENT:
532          *  if seg contains an ACK, but not for our SYN, drop the input.
533          *  if seg contains an RST, then drop the connection.
534          *  if seg does not contain SYN, then drop it.
535          * Otherwise this is an acceptable SYN segment
536          *  initialize tp->rcv_nxt and tp->irs
537          *  if seg contains ack then advance tp->snd_una
538          *  if SYN has been acked change to ESTABLISHED else SYN_RCVD state
539          *  arrange for segment to be acked (eventually)
540          *  continue processing rest of data/controls, beginning with URG
541          */
542     case TCPS_SYN_SENT:
543         if ((tiflags & TH_ACK) &&
544             (SEQ_LEQ(ti->ti_ack, tp->iss) ||
545              SEQ_GT(ti->ti_ack, tp->snd_max)))
546             goto dropwithreset;
547         if (tiflags & TH_RST) {
548             if (tiflags & TH_ACK)
549                 tp = tcp_drop(tp, ECONNREFUSED);
550             goto drop;
551         }
552         if ((tiflags & TH_SYN) == 0)
553             goto drop;
```

tcp_input.c

Figure 28.18 `tcp_input` function: check if SYN in response to active open.

Verify received ACK

530–546 When TCP sends a SYN in response to an active open by a process, we'll see in Figure 30.4 that the connection's `iss` is copied from the global `tcp_iss` and the macro `tcp_sendseqinit` (shown at the end of the previous section) is executed. Assuming the ISS is 365, Figure 28.19 shows the send sequence variables after the SYN is sent by `tcp_output`.

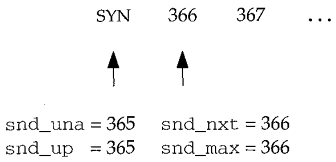


Figure 28.19 Send variables after SYN is sent with sequence number 365.

`tcp_sendseqinit` sets all four of these variables to 365, then Figure 26.31 increments two of them to 366 when the SYN segment is output. Therefore, if the received segment in Figure 28.18 contains an ACK, and if the acknowledgment field is less than or equal to `iss` (365) or greater than `snd_max` (366), the ACK is invalid, causing the segment to be dropped and an RST sent in reply. Notice that the received segment for a connection in the SYN_SENT state need not contain an ACK. It can contain only a SYN, which is called a *simultaneous open* (Figure 24.15), and is described shortly.

Process and drop RST segment

547–551 If the received segment contains an RST, it is dropped. But the ACK flag was checked first because receipt of an acceptable ACK (which was just verified) *and* an RST in response to a SYN is how the other end tells TCP that its connection request was refused. Normally this is caused by the server process not being started on the other host. In this case `tcp_drop` sets the socket's `so_error` variable, causing an error to be returned to the process that called `connect`.

Verify SYN flag set

552–553 If the SYN flag is not set in the received segment, it is dropped.

The remainder of this `case` handles the receipt of a SYN (with an optional ACK) in response to TCP's SYN. The next part of `tcp_input`, shown in Figure 28.20, continues processing the SYN.

Process ACK

554–558 If the received segment contains an ACK, `snd_una` is set to the acknowledgment field. In Figure 28.19, `snd_una` becomes 366, since 366 is the only acceptable value for the acknowledgment field. If `snd_nxt` is less than `snd_una` (which shouldn't happen, given Figure 28.19), `snd_nxt` is set to `snd_una`.

Turn off retransmission timer

559 The retransmission timer is turned off.

This is a bug. This timer should be turned off only if the ACK flag is set, since the receipt of a SYN without an ACK is a simultaneous open, and doesn't mean the other end received TCP's SYN.

Initialize receive sequence numbers

560–562 The initial receive sequence number is copied from the sequence number of the received segment. The `tcp_rcvseqinit` macro (shown at the end of the previous section) initializes `rcv_adv` and `rcv_nxt` to the receive sequence number, plus 1. The `TF_ACKNOW` flag is set, causing `tcp_output` to be called at the bottom of this function. The segment it sends will contain `rcv_nxt` as the acknowledgment field (Figure 26.27), which acknowledges the SYN just received.

563–564 If the received segment contains an ACK, and if `snd_una` is greater than the ISS for the connection, the active open is complete, and the connection is established.

This second test appears superfluous. At the beginning of Figure 28.20 `snd_una` was set to the received acknowledgment field if the ACK flag was on. Also the `if` following the `case`

tcp_input.c

```
554         if (tiflags & TH_ACK) {
555             tp->snd_una = ti->ti_ack;
556             if (SEQ_LT(tp->snd_nxt, tp->snd_una))
557                 tp->snd_nxt = tp->snd_una;
558         }
559         tp->t_timer[TCPT_REXMT] = 0;
560         tp->irs = ti->ti_seq;
561         tcp_rcvseqinit(tp);
562         tp->t_flags |= TF_ACKNOW;
563         if (tiflags & TH_ACK && SEQ_GT(tp->snd_una, tp->iss)) {
564             tcpstat.tcps_connects++;
565             soisconnected(so);
566             tp->t_state = TCPS_ESTABLISHED;
567             /* Do window scaling on this connection? */
568             if ((tp->t_flags & (TF_RCVD_SCALE | TF_REQ_SCALE)) ==
569                 (TF_RCVD_SCALE | TF_REQ_SCALE)) {
570                 tp->snd_scale = tp->requested_s_scale;
571                 tp->rcv_scale = tp->request_r_scale;
572             }
573             (void) tcp_reass(tp, (struct tcpiphdr *) 0,
574                              (struct mbuf *) 0);
575             /*
576              * if we didn't have to retransmit the SYN,
577              * use its rtt as our initial srtt & rtt var.
578              */
579             if (tp->t_rtt)
580                 tcp_xmit_timer(tp, tp->t_rtt);
581         } else
582             tp->t_state = TCPS_SYN_RECEIVED;
```

tcp_input.c

Figure 28.20 `tcp_input` function: process received SYN in response to an active open.

statement in Figure 28.18 verified that the received acknowledgment field is greater than the ISS. So at this point in the code, if the ACK flag is set, we're already guaranteed that `snd_una` is greater than the ISS.

Connection is established

565–566 `soisconnected` sets the socket state to connected, and the state of the TCP connection is set to ESTABLISHED.

Check for window scale option

567–572 If TCP sent the window scale option in its SYN and the received SYN also contains the option, the option is enabled and the two variables `snd_scale` and `rcv_scale` are set. Since the TCP control block is initialized to 0 by `tcp_newtcpcb`, these two variables correctly default to 0 if the window scale option is not used.

Pass any queued data to process

573–574 Since data can arrive for a connection before the connection is established, any such data is now placed in the receive buffer by calling `tcp_reass` with a null pointer as the second argument.

This test is unnecessary. In this piece of code, TCP has just received the SYN with an ACK that moves it from the SYN_SENT state to the ESTABLISHED state. If data appears with this received SYN segment, it isn't processed until the label `dodata` near the end of the function. If TCP just received a SYN without an ACK (a simultaneous open) but with some data, that data is handled later (Figure 29.2) when the ACK is received that moves the connection from the SYN_RCVD state to the ESTABLISHED state.

Although it is valid for data to accompany a SYN, and Net/3 handles this type of received segment correctly, Net/3 never generates such a segment.

Update RTT estimators

575–580 If the SYN that is ACKed was being timed, `tcp_xmit_timer` initializes the RTT estimators based on the measured RTT for the SYN.

TCP ignores a received timestamp option here, and checks only the `t_rtt` counter. TCP sends a timestamp in a SYN generated by an active open (Figure 26.24) and if the other end agrees to the option, the other end should echo the received timestamp in its SYN. (Net/3 echoes the received timestamp in a SYN in Figure 28.10.) This would allow TCP to use the received timestamp here, instead of `t_rtt`, but since both have the same precision (500 ms) there's no advantage in using the timestamp value. The real advantage in using the timestamp option, instead of the `t_rtt` counter, is with large pipes, when lots of segments are in flight at once, providing more RTT timings and (it is hoped) better estimators.

Simultaneous open

581–582 When TCP receives a SYN without an ACK in the SYN_SENT state, it is a simultaneous open and the connection moves to the SYN_RCVD state.

The next piece of code, shown in Figure 28.21, handles any data received with the SYN. The label `trimthenstep6` is also jumped to at the end of Figure 28.17.

tcp_input.c

```
    trimthenstep6:
        /*
         * Advance ti->ti_seq to correspond to first data byte.
         * If data, trim to stay within window,
         * dropping FIN if necessary.
         */
        ti->ti_seq++;
        if (ti->ti_len > tp->rcv_wnd) {
            todrop = ti->ti_len - tp->rcv_wnd;
            m_adj(m, -todrop);
            ti->ti_len = tp->rcv_wnd;
            tiflags &= ~TH_FIN;
            tcpstat.tcps_rcvpackafterwin++;
            tcpstat.tcps_rcvbyteafterwin += todrop;
        }
        tp->snd_wl1 = ti->ti_seq - 1;
        tp->rcv_up = ti->ti_seq;
        goto step6;
    }
```

tcp_input.c

Figure 28.21 `tcp_input` function: common processing for receipt of SYN.

584–589 The sequence number of the segment is incremented by 1 to account for the SYN. If there is any data in the segment, `ti_seq` now contains the starting sequence number of the first byte of data.

Drop any received data that follows receive window

590–597 `ti_len` is the number of data bytes in the segment. If it is greater than the receive window, the excess data (`ti_len` minus `rcv_wnd`) is dropped by `m_adj`. The negative argument to this function causes the data to be trimmed from the end of the mbuf chain (Figure 2.20). `ti_len` is updated to be the new amount of data in the mbuf chain and in case the FIN flag was set, it is cleared. This is because the FIN would follow the final data byte, which was just discarded because it was outside the receive window.

> If too much data is received with a SYN, and if the SYN is in response to an active open, the other end received TCP's SYN, which contained a window advertisement. This means the other end ignored the advertised window and is exhibiting unsocial behavior. But if too much data accompanies a SYN performing an active open, the other end has not received a window advertisement, so it has to guess how much data can accompany its SYN.

Force update of window variables

598–599 `snd_wl1` is set the received sequence number minus 1. We'll see in Figure 29.15 that this causes the three window update variables, `snd_wnd`, `snd_wl1`, and `snd_wl2`, to be updated. The receive urgent pointer (`rcv_up`) is set to the received sequence number. A jump is made to `step6`, which refers to a step in RFC 793, and we cover this in Figure 29.15.

28.7 PAWS: Protection Against Wrapped Sequence Numbers

The next part of `tcp_input`, shown in Figure 28.22, provides protection against wrapped sequence numbers: the PAWS algorithm from RFC 1323. Also recall our discussion of the timestamp option in Section 26.6.

Basic PAWS test

602–613 `ts_present` was set by `tcp_dooptions` if a timestamp option was present. If the following three conditions are all true, the segment is dropped:

1. the RST flag is not set (Exercise 28.8),
2. TCP has received a valid timestamp from this peer (`ts_recent` is nonzero), and
3. the received timestamp in this segment (`ts_val`) is less than the previously received timestamp from this peer.

PAWS is built on the premise that the 32-bit timestamp values wrap around at a much lower frequency than the 32-bit sequence numbers, on a high-speed connection. Exercise 28.6 shows that even at the highest possible timestamp counter frequency (incrementing by 1 bit every millisecond), the sign bit of the timestamp wraps around only every 24 days. On a high-speed network such as a gigabit network, the sequence

tcp_input.c

```
602     /*
603      * States other than LISTEN or SYN_SENT.
604      * First check timestamp, if present.
605      * Then check that at least some bytes of segment are within
606      * receive window.  If segment begins before rcv_nxt,
607      * drop leading data (and SYN); if nothing left, just ack.
608      *
609      * RFC 1323 PAWS: If we have a timestamp reply on this segment
610      * and it's less than ts_recent, drop it.
611      */
612     if (ts_present && (tiflags & TH_RST) == 0 && tp->ts_recent &&
613         TSTMP_LT(ts_val, tp->ts_recent)) {

614         /* Check to see if ts_recent is over 24 days old.  */
615         if ((int) (tcp_now - tp->ts_recent_age) > TCP_PAWS_IDLE) {
616             /*
617              * Invalidate ts_recent.  If this segment updates
618              * ts_recent, the age will be reset later and ts_recent
619              * will get a valid value.  If it does not, setting
620              * ts_recent to zero will at least satisfy the
621              * requirement that zero be placed in the timestamp
622              * echo reply when ts_recent isn't valid.  The
623              * age isn't reset until we get a valid ts_recent
624              * because we don't want out-of-order segments to be
625              * dropped when ts_recent is old.
626              */
627             tp->ts_recent = 0;
628         } else {
629             tcpstat.tcps_rcvduppack++;
630             tcpstat.tcps_rcvdupbyte += ti->ti_len;
631             tcpstat.tcps_pawsdrop++;
632             goto dropafterack;
633         }
634     }
```

tcp_input.c

Figure 28.22 `tcp_input` function: process timestamp option.

number can wrap in 17 seconds (Section 24.3 of Volume 1). Therefore, if the received timestamp value is less than the most recent one from this peer, this segment is old and must be discarded (subject to the outdated timestamp test that follows). The packet might be discarded later in the input processing because the sequence number is "old," but PAWS is intended for high-speed connections where the sequence numbers can wrap quickly.

Notice that the PAWS algorithm is symmetric: it not only discards duplicate data segments but also discards duplicate ACKs. All received segments are subject to PAWS. Recall that the header prediction code also applied the PAWS test (Figure 28.11).

Check for outdated timestamp

614–627 There is a small possibility that the reason the PAWS test fails is because the connection has been idle for a long time. The received segment is not a duplicate; it is just that because the connection has been idle for so long, the peer's timestamp value has wrapped around when compared to the most recent timestamp from that peer.

Whenever `ts_recent` is copied from the timestamp in a received segment, `ts_recent_age` records the current time (`tcp_now`). If the time at which `ts_recent` was saved is more than 24 days ago, it is set to 0 to invalidate it. The constant `TCP_PAWS_IDLE` is defined to be (24 × 24 × 60 × 60 × 2), the final 2 being the number of ticks per second. The received segment is not dropped in this case, since the problem is not a duplicated segment, but an outdated timestamp. See also Exercises 28.6 and 28.7.

Figure 28.23 shows an example of an outdated timestamp. The system on the left is a non-Net/3 system that increments its timestamp clock at the highest frequency allowed by RFC 1323: once every millisecond. The system on the right is a Net/3 system.

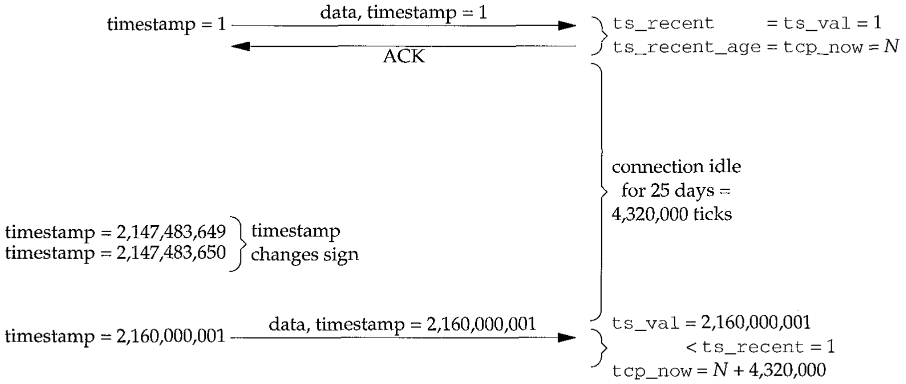


Figure 28.23 Example of outdated timestamp.

When the data segment arrives with a timestamp of 1, that value is saved in `ts_recent` and `ts_recent_age` is set to the current time (`tcp_now`), as shown in Figures 28.11 and 28.35. The connection is then idle for 25 days, during which time `tcp_now` will increase by 4,320,000 (25 × 24 × 60 × 60 × 2). During these 25 days the other end's timestamp clock will increase by 2,160,000,000 (25 × 24 × 60 × 60 × 1000). During this interval the timestamp "changes sign" with regard to the value 1, that is, 2,147,483,649 is greater than 1, but 2,147,483,650 is less than 1 (recall Figure 24.26). Therefore, when the data segment is received with a timestamp of 2,160,000,001, this value is less than `ts_recent` (1), when compared using the `TSTMP_LT` macro, so the PAWS test fails. But since `tcp_now` minus `ts_recent_age` is greater than 24 days, the reason for the failure is that the connection has been idle for more than 24 days, and the segment is accepted.

Drop duplicate segment

628–633 The segment is determined to be a duplicate based on the PAWS algorithm, and the timestamp is not outdated. It is dropped, after being acknowledged (since all duplicate segments are acknowledged).

Figure 24.5 shows a much smaller value for `tcps_pawsdrop` (22) than for `tcps_rcvduppack` (46,953). This is probably because fewer systems support the timestamp option today, causing most duplicate packets to be discarded by later tests in TCP's input processing instead of by PAWS.

28.8 Trim Segment so Data is Within Window

This section trims the received segment so that it contains only data that is within the advertised window:

- duplicate data at the beginning of the received segment is discarded, and
- data that is beyond the end of the window is discarded from the end of the segment.

What remains is new data within the window. The code shown in Figure 28.24 checks if there is any duplicate data at the beginning of the segment.

tcp_input.c

```
635     todrop = tp->rcv_nxt - ti->ti_seq;
636     if (todrop > 0) {
637         if (tiflags & TH_SYN) {
638             tiflags &= ~TH_SYN;
639             ti->ti_seq++;
640             if (ti->ti_urp > 1)
641                 ti->ti_urp--;
642             else
643                 tiflags &= ~TH_URG;
644             todrop--;
645         }
```

tcp_input.c

Figure 28.24 `tcp_input` function: check for duplicate data at beginning of segment.

Check if any duplicate data at front of segment

635–636 If the starting sequence number of the received segment (`ti_seq`) is less than the next receive sequence number expected (`rcv_nxt`), data at the beginning of the segment is old and `todrop` will be greater than 0. These data bytes have already been acknowledged and passed to the application (Figure 24.18).

Remove duplicate SYN

637–645 If the SYN flag is set, it refers to the first sequence number in the segment, which is known to be old. The SYN flag is cleared and the starting sequence number of the segment is incremented by 1 to skip over the duplicate SYN. Furthermore, if the urgent offset in the received segment (`ti_urp`) is greater than 1, it must be decremented by 1, since the urgent offset is relative to the starting sequence number, which was just incremented. If the urgent offset is 0 or 1, it is left alone, but in case it was 1, the URG flag is cleared. Finally `todrop` is decremented by 1 (since the SYN occupies a sequence number).

The handling of duplicate data at the front of the segment continues in Figure 28.25.

tcp_input.c

```
646         if (todrop >= ti->ti_len) {
647             tcpstat.tcps_rcvduppack++;
648             tcpstat.tcps_rcvdupbyte += ti->ti_len;
649             /*
650              * If segment is just one to the left of the window,
651              * check two special cases:
652              * 1. Don't toss RST in response to 4.2-style keepalive.
653              * 2. If the only thing to drop is a FIN, we can drop
654              *    it, but check the ACK or we will get into FIN
655              *    wars if our FINs crossed (both CLOSING).
656              * In either case, send ACK to resynchronize,
657              * but keep on processing for RST or ACK.
658              */
659             if ((tiflags & TH_FIN && todrop == ti->ti_len + 1)
660                 ) {
661                 todrop = ti->ti_len;
662                 tiflags &= ~TH_FIN;
663                 tp->t_flags |= TF_ACKNOW;
664             } else {
665                 /*
666                  * Handle the case when a bound socket connects
667                  * to itself. Allow packets with a SYN and
668                  * an ACK to continue with the processing.
669                  */
670                 if (todrop != 0 || (tiflags & TH_ACK) == 0)
671                     goto dropafterack;
672             }
673         } else {
674             tcpstat.tcps_rcvpartduppack++;
675             tcpstat.tcps_rcvpartdupbyte += todrop;
676         }
677         m_adj(m, todrop);
678         ti->ti_seq += todrop;
679         ti->ti_len -= todrop;
680         if (ti->ti_urp > todrop)
681             ti->ti_urp -= todrop;
682         else {
683             tiflags &= ~TH_URG;
684             ti->ti_urp = 0;
685         }
686     }
```

tcp_input.c

Figure 28.25 `tcp_input` function: handle completely duplicate segment.

Check for entire duplicate packet

646–648 If the amount of duplicate data at the front of the segment is greater than or equal to the size of the segment, the entire segment is a duplicate.

Check for duplicate FIN

649–663 The next check is whether the FIN is duplicated. Figure 28.26 shows an example of this.

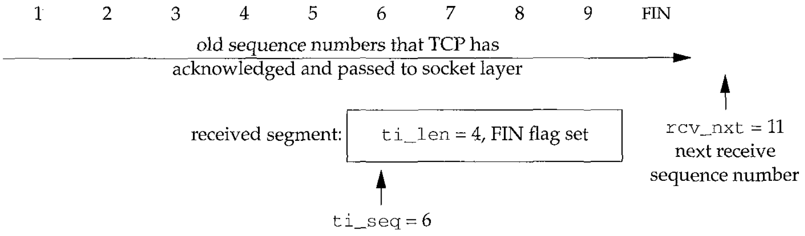


Figure 28.26 Example of duplicate packet with FIN flag set.

In this example `todrop` equals 5, which is greater than or equal to `ti_len` (4). Since the FIN flag is set and `todrop` equals `ti_len` plus 1, `todrop` is set to 4, the FIN flag is cleared, and the `TF_ACKNOW` flag is set, forcing an immediate ACK to be sent at the end of this function. This example also works for other segments if `ti_seq` plus `ti_len` equals 10.

The code contains the comment regarding 4.2BSD keepalives. This code (another test within the `if` statement) is omitted.

Generate duplicate ACK

664–672 If `todrop` is nonzero (the completely duplicate segment contains data) or the ACK flag is not set, the segment is dropped and an ACK is generated by `dropafterack`. This normally occurs when the other end did not receive our ACK, causing the other end to retransmit the segment. TCP generates another ACK.

Handle simultaneous open or self-connect

664–672 This code also handles either a simultaneous open or a socket that connects to itself. We go over both of these scenarios in the next section. If `todrop` equals 0 (there is no data in the completely duplicate segment) and the ACK flag is set, processing is allowed to continue.

This `if` statement is new with 4.4BSD. Earlier Berkeley-derived systems just had a jump to `dropafterack`. These systems could not handle either a simultaneous open or a socket connecting to itself.

Nevertheless, the piece of code in this figure still has bugs, which we describe at the end of this section.

Update statistics for partial duplicate segments

673–676 This `else` clause is executed when `todrop` is less than the segment length: only part of the segment contains duplicate bytes.

Remove duplicate data and update urgent offset

677–685 The duplicate bytes are removed from the front of the mbuf chain by `m_adj` and the starting sequence number and length adjusted appropriately. If the urgent offset points to data still in the mbuf, it is also adjusted. Otherwise the urgent offset is set to 0 and the URG flag is cleared.

The next part of input processing, shown in Figure 28.27, handles data that arrives after the process has terminated.

tcp_input.c

```
687     /*
688      * If new data is received on a connection after the
689      * user processes are gone, then RST the other end.
690      */
691     if ((so->so_state & SS_NOFDREF) &&
692         tp->t_state > TCPS_CLOSE_WAIT && ti->ti_len) {
693         tp = tcp_close(tp);
694         tcpstat.tcps_rcvafterclose++;
695         goto dropwithreset;
696     }
```

tcp_input.c

Figure 28.27 `tcp_input` function: handle data that arrives after the process terminates.

687–696 If the socket has no descriptor referencing it, the process has closed the connection (the state is any one of the five with a value greater than CLOSE_WAIT in Figure 24.16), and there is data in the received segment, the connection is closed. The segment is then dropped and an RST is output.

Because of TCP's half-close, if a process terminates unexpectedly (perhaps it is terminated by a signal), when the kernel closes all open descriptors as part of process termination, a FIN is output by TCP. The connection moves into the FIN_WAIT_1 state. But the receipt of the FIN by the other end doesn't tell TCP whether this end performed a half-close or a full-close. If the other end assumes a half-close, and sends more data, it will receive an RST from the code in Figure 28.27.

The next piece of code, shown in Figure 28.29, removes any data from the end of the received segment that is beyond the right edge of the advertised window.

Calculate number of bytes beyond right edge of window

697–703 `todrop` contains the number of bytes of data beyond the right edge of the window. For example, in Figure 28.28, `todrop` would be (6 + 5) minus (4 + 6), or 1.

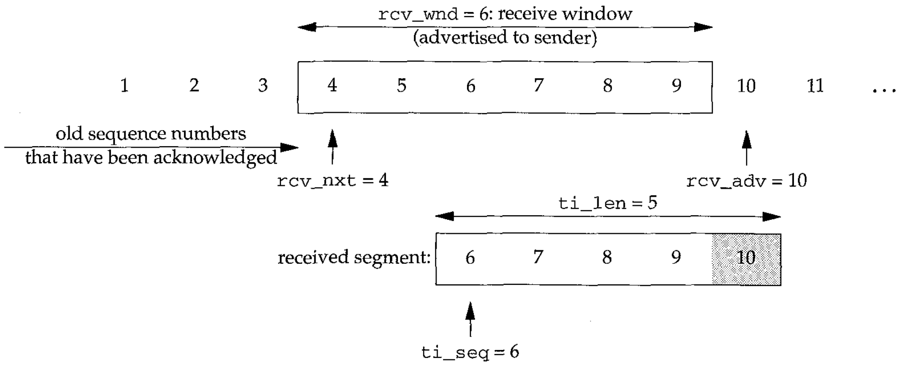


Figure 28.28 Example of received segment with data beyond right edge of window.

tcp_input.c

```
697     /*
698      * If segment ends after window, drop trailing data
699      * (and PUSH and FIN); if nothing left, just ACK.
700      */
701     todrop = (ti->ti_seq + ti->ti_len) - (tp->rcv_nxt + tp->rcv_wnd);
702     if (todrop > 0) {
703         tcpstat.tcps_rcvpackafterwin++;
704         if (todrop >= ti->ti_len) {
705             tcpstat.tcps_rcvbyteafterwin += ti->ti_len;
706             /*
707              * If a new connection request is received
708              * while in TIME_WAIT, drop the old connection
709              * and start over if the sequence numbers
710              * are above the previous ones.
711              */
712             if (tiflags & TH_SYN &&
713                 tp->t_state == TCPS_TIME_WAIT &&
714                 SEQ_GT(ti->ti_seq, tp->rcv_nxt)) {
715                 iss = tp->rcv_nxt + TCP_ISSINCR;
716                 tp = tcp_close(tp);
717                 goto findpcb;
718             }
719             /*
720              * If window is closed can only take segments at
721              * window edge, and have to drop data and PUSH from
722              * incoming segments.  Continue processing, but
723              * remember to ack.  Otherwise, drop segment
724              * and ack.
725              */
726             if (tp->rcv_wnd == 0 && ti->ti_seq == tp->rcv_nxt) {
727                 tp->t_flags |= TF_ACKNOW;
728                 tcpstat.tcps_rcvwinprobe++;
729             } else
730                 goto dropafterack;
731         } else
732             tcpstat.tcps_rcvbyteafterwin += todrop;
733         m_adj(m, -todrop);
734         ti->ti_len -= todrop;
735         tiflags &= ~(TH_PUSH | TH_FIN);
736     }
```

tcp_input.c

Figure 28.29 `tcp_input` function: remove data beyond right edge of window.

Check for new incarnation of a connection in the TIME_WAIT state

704–718 If `todrop` is greater than or equal to the length of the segment, the entire segment will be dropped. If the following three conditions are all true:

1. the SYN flag is set, and
2. the connection is in the TIME_WAIT state, and
3. the new starting sequence number is greater than the final sequence number for the connection,

this is a request for a new incarnation of a connection that was recently terminated and is currently in the TIME_WAIT state. This is allowed by RFC 1122, but the ISS for the new connection must be greater than the last sequence number used (`rcv_nxt`). TCP adds 128,000 (`TCP_ISSINCR`), which becomes the ISS when the code in Figure 28.17 is executed. The PCB and TCP control block for the connection in the TIME_WAIT state is discarded by `tcp_close`. A jump is made to `findpcb` (Figure 28.5) to locate the PCB for the listening server, assuming it is still running. The code in Figure 28.7 is then executed, creating a new socket for the new connection, and finally the code in Figures 28.16 and 28.17 will complete the new connection request.

Check for probe of closed window

719–728 If the receive window is closed (`rcv_wnd` equals 0) and the received segment starts at the left edge of the window (`rcv_nxt`), then the other end is probing TCP's closed window. An immediate ACK is sent as the reply, even though the ACK may still advertise a window of 0. Processing of the received segment also continues for this case.

Drop other segments that are completely outside window

729–730 The entire segment lies outside the window and it is not a window probe, so the segment is discarded and an ACK is sent as the reply. This ACK will contain the expected sequence number.

Handle segments that contain some valid data

731–735 The data to the right of the window is discarded from the mbuf chain by `m_adj` and `ti_len` is updated. In the case of a probe into a closed window, this discards all the data in the mbuf chain and sets `ti_len` to 0. Finally the FIN and PSH flags are cleared.

When to Drop an ACK

The code in Figure 28.25 has a bug that causes a jump to `dropafterack` in several cases when the code should fall through for further processing of the segment [Carlson 1993; Lanciani 1993]. In an actual scenario, when both ends of a connection had a hole in the data on the reassembly queue and both ends enter the persist state, the connection becomes deadlocked as both ends throw away perfectly good ACKs.

The fix is to simplify the code at the beginning of Figure 28.25. Instead of jumping to `dropafterack`, a completely duplicate segment causes the FIN flag to be turned off and an immediate ACK to be generated at the end of the function. Lines 646–676 in Figure 28.25 are replaced with the code shown in Figure 28.30. This code also corrects another bug present in the original code (Exercise 28.9).

```
if (todrop > ti->ti_len ||
    todrop == ti->ti_len && (tiflags & TH_FIN) == 0) {

    /*
     * Any valid FIN must be to the left of the window.
     * At this point the FIN must be a duplicate or
     * out of sequence; drop it.
     */
     tiflags &= ~TH_FIN;

    /*
     * Send an ACK to resynchronize and drop any data.
     * But keep on processing for RST or ACK.
     */
    tp->t_flags |= TF_ACKNOW;
    todrop = ti->ti_len;
    tcpstat.tcps_rcvdupbyte += todrop;
    tcpstat.tcps_rcvduppack++;

} else {
    tcpstat.tcps_rcvpartduppack++;
    tcpstat.tcps_rcvpartdupbyte += todrop;
}
```

Figure 28.30 Correction for lines 646–676 of Figure 28.25.

28.9 Self-Connects and Simultaneous Opens

It is instructive to look at the steps involved in a socket connecting to itself to see how the one-line fix to Figure 28.25 that was added to 4.4BSD allows this. This same fix allowed simultaneous opens to work, which wasn't handled correctly prior to 4.4BSD.

A process creates a socket and connects it to itself using the system calls: `socket`, `bind` a local port (say 3000), and then `connect` to this same port and some local IP address. If the `connect` succeeds, the socket is connected to itself: anything written to the socket can be read back from the socket. This is similar to a full-duplex pipe, but with a single descriptor instead of two descriptors. Although this is of limited use within a process, we'll see that the state transitions are the same as they are for a simultaneous open. If your system doesn't allow a socket to connect to itself, it probably doesn't handle simultaneous opens correctly either, and the latter are required by RFC 1122. Some people are surprised that a self-connect even works, given that a single Internet PCB and a single TCP control block are used. But TCP is a full-duplex, symmetric protocol and it maintains separate variables for each direction of data flow.

Figure 28.31 shows the send sequence space when the process calls `connect`. A SYN segment is sent and the state becomes SYN_SENT.

The SYN is received and processed in Figures 28.18 and 28.20, but since the SYN does not contain an ACK the resulting state is SYN_RCVD. According to the state transition diagram (Figure 24.15), this looks like a simultaneous open. Figure 28.32 shows the receive sequence space.

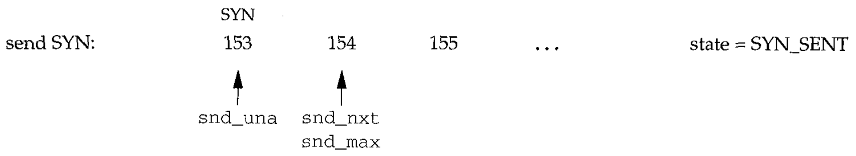


Figure 28.31 Send sequence space when SYN is sent for self-connect.

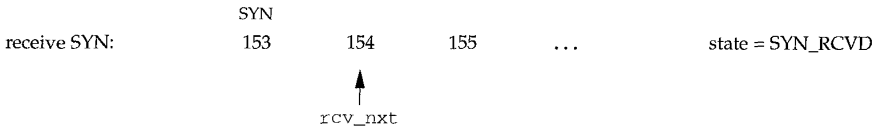


Figure 28.32 Receive sequence space after received SYN is processed.

Figure 28.20 sets the `TF_ACKNOW` flag and the segment generated by `tcp_output` will contain a SYN and an ACK (the `tcp_outflags` value in Figure 24.16). The sequence number of the SYN is 153 and the acknowledgment number is 154.

Nothing changes in the send sequence space from Figure 28.20, except the state is now SYN_SENT. Figure 28.33 shows the receive sequence space when the segment with the SYN and ACK is received.

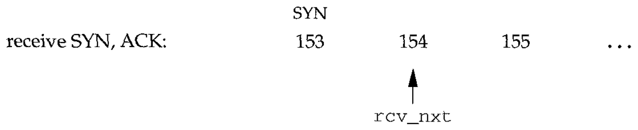


Figure 28.33 Receive sequence space when segment with SYN and ACK received.

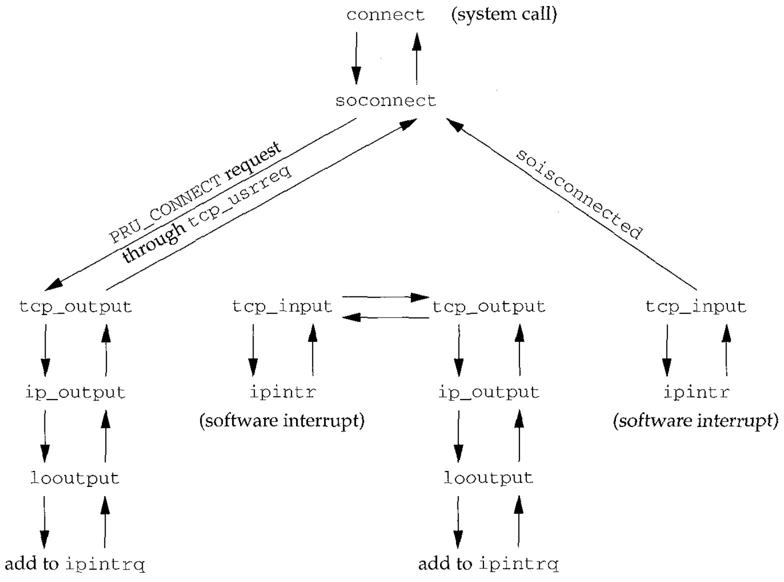
Since the connection state is SYN_RCVD, the segment is not processed by the active open or passive open code that we saw earlier in this chapter. It must be processed by the SYN_RCVD code that we'll examine in Figure 29.2. But it is first processed by Figure 28.24, and it looks like a duplicate SYN:

```
todrop = rcv_nxt - ti_seq
       = 154 - 153
       = 1
```

Since the SYN flag is set, the flag is cleared, `ti_seq` becomes 154, and `todrop` becomes 0. But the test at the beginning of Figure 28.25 is true, because `todrop` equals the length of the segment (0). The segment is counted as a duplicate packet and the code with the comment "Handle the case when a bound socket connects to itself" is executed. Earlier releases jumped to `dropafterack`, which skipped the necessary code to handle the SYN_RCVD state, preventing the connection from ever being established. Instead, Net/3 continues processing the received segment if `todrop` equals 0 and the

ACK flag is set, both of which are true in this example. This allows the SYN_RCVD processing to happen later in the function, which moves the connection to the ESTABLISHED state.

It is also interesting to look at the sequence of function calls in this self-connect. This is shown in Figure 28.34.



action:	send SYN	process SYN	send SYN, ACK	process SYN, ACK
starting state:	CLOSED	SYN_SENT	SYN_RCVD	SYN_RCVD
ending state:	SYN_SENT	SYN_RCVD	SYN_RCVD	ESTABLISHED

Figure 28.34 Sequence of function calls for self-connect.

The order of the operations goes from the left to the right. The steps that we show begin with the process calling `connect`. This issues the `PRU_CONNECT` request, which sends a SYN down the protocol stack. Since the segment is destined for the host's own IP address it is routed to the loopback interface, which adds the segment to `ipintrq` and generates a software interrupt.

The software interrupt causes `ipintr` to execute, which calls `tcp_input`. This function calls `tcp_output`, causing a SYN segment with an ACK to be sent down the protocol stack. It is again added to `ipintrq` by the loopback interface, and a software interrupt is generated. When this interrupt is processed by `ipintr`, the function `tcp_input` is called, and it moves the connection to the ESTABLISHED state.

28.10 Record Timestamp

The next part of `tcp_input`, shown in Figure 28.35, handles a received timestamp option.

tcp_input.c

```
737     /*
738      * If last ACK falls within this segment's sequence numbers,
739      * record its timestamp.
740      */
741     if (ts_present && SEQ_LEQ(ti->ti_seq, tp->last_ack_sent) &&
742         SEQ_LT(tp->last_ack_sent, ti->ti_seq + ti->ti_len +
743                ((tiflags & (TH_SYN | TH_FIN)) != 0))) {
744         tp->ts_recent_age = tcp_now;
745         tp->ts_recent = ts_val;
746     }
```

tcp_input.c

Figure 28.35 `tcp_input` function: record timestamp.

737–746 If the received segment contains a timestamp, the timestamp value is saved in `ts_recent`. We discussed in Section 26.6 how this code used by Net/3 is flawed. The expression

```
((tiflags & (TH_SYN|TH_FIN)) != 0)
```

is 0 if neither of the two flags is set, or 1 if either is set. This effectively adds 1 to `ti_len` if either flag is set.

28.11 RST Processing

Figure 28.36 shows the `switch` statement to handle the RST flag, which depends on the connection state.

SYN_RCVD state

759–761 The socket's error code is set to ECONNREFUSED, and a jump is made a few lines forward to close the socket.

This state can be entered from two directions. Normally it is entered from the LISTEN state, after a SYN has been received. TCP replied with a SYN and an ACK but received an RST in reply. Perhaps the other end sent its SYN and then terminated before the reply arrived, causing it to send an RST. In this case the socket referred to by `so` is the new socket created by `sonewconn` in Figure 28.7. Since `dropsocket` will still be true, the socket is discarded at the label `drop`. The listening descriptor isn't affected at all. This is why we show the state transition from SYN_RCVD back to LISTEN in Figure 24.15.

This state can also be entered by a simultaneous open, after a process has called `connect`. In this case the socket error is returned to the process.

tcp_input.c

```
747     /*
748      * If the RST bit is set examine the state:
749      *    SYN_RECEIVED state:
750      *  If passive open, return to LISTEN state.
751      *  If active open, inform user that connection was refused.
752      *    ESTABLISHED, FIN_WAIT_1, FIN_WAIT2, CLOSE_WAIT states:
753      *  Inform user that connection was reset, and close tcb.
754      *    CLOSING, LAST_ACK, TIME_WAIT states
755      *  Close the tcb.
756      */
757     if (tiflags & TH_RST)
758         switch (tp->t_state) {

759         case TCPS_SYN_RECEIVED:
760             so->so_error = ECONNREFUSED;
761             goto close;

762         case TCPS_ESTABLISHED:
763         case TCPS_FIN_WAIT_1:
764         case TCPS_FIN_WAIT_2:
765         case TCPS_CLOSE_WAIT:
766             so->so_error = ECONNRESET;
767           close:
768             tp->t_state = TCPS_CLOSED;
769             tcpstat.tcps_drops++;
770             tp = tcp_close(tp);
771             goto drop;

772         case TCPS_CLOSING:
773         case TCPS_LAST_ACK:
774         case TCPS_TIME_WAIT:
775             tp = tcp_close(tp);
776             goto drop;
777         }
```

tcp_input.c

Figure 28.36 `tcp_input` function: process RST flag.

Other states

762–777 The receipt of an RST in the ESTABLISHED, FIN_WAIT_1, FIN_WAIT_2, or CLOSE_WAIT states returns the error `ECONNRESET`. In the CLOSING, LAST_ACK, and TIME_WAIT state an error is not generated, since the process has closed the socket.

Allowing an RST to terminate a connection in the TIME_WAIT state circumvents the reason this state exists. RFC 1337 [Braden 1992] discusses this and other forms of "TIME_WAIT assassination hazards" and recommends *not* letting an RST prematurely terminate the TIME_WAIT state. See Exercise 28.10 for an example.

The next piece of code, shown in Figure 28.37, checks for erroneous SYNs and verifies that an ACK is present.

tcp_input.c

```
778     /*
779      * If a SYN is in the window, then this is an
780      * error and we send an RST and drop the connection.
781      */
782     if (tiflags & TH_SYN) {
783         tp = tcp_drop(tp, ECONNRESET);
784         goto dropwithreset;
785     }
786     /*
787      * If the ACK bit is off we drop the segment and return.
788      */
789     if ((tiflags & TH_ACK) == 0)
790         goto drop;
```

tcp_input.c

Figure 28.37 `tcp_input` function: handle SYN-full and ACK-less segments.

778–785 If the SYN flag is still set, this is an error and the connection is dropped with the error `ECONNRESET`.

786–790 If the ACK flag is not set, the segment is dropped. The remainder of this function, which we continue in the next chapter, assumes the ACK flag is set.

28.12 Summary

This chapter has started our detailed look at TCP input. It continues in the next chapter.

The code in this chapter verifies the segment's checksum, processes any TCP options, handles SYNs that initiate or complete connection requests, trims excess data from the beginning or end of the segment, and processes the RST flag.

Header prediction is a successful attempt to handle common cases with the minimum amount of processing. Although the general processing steps that we've covered handle all possible cases (which they must), many segments are well behaved and the processing steps can be minimized.

Exercises

28.1 Given that the maximum size of a socket buffer is 262,144 in Net/3, what are the possible window scale shift factors calculated by Figure 28.7?

28.2 Given that the maximum size of a socket buffer is 262,144 in Net/3, what is the maximum throughput possible with a round-trip time of 60 ms? (*Hint*: See Figure 24.5 in Volume 1 and solve for the bandwidth.)

28.3 Why are the two timestamp values fetched using `bcopy` in Figure 28.10?

28.4 We mentioned in Section 26.6 that TCP correctly handles timestamp options in a format other than the one recommended in Appendix A of RFC 1323. While this is true, what is the penalty for not following the recommended format?

28.5 The `PRU_ATTACH` request allocates the PCB and the TCP control block, but doesn't call `tcp_template` to allocate the header template. Instead we saw in Figure 28.17 that the header template is allocated when the SYN arrives. Why doesn't the `PRU_ATTACH` request allocate this template?

28.6 Read RFC 1323 to determine why the limit of 24 days was chosen in Figure 28.22.

28.7 The comparison of `tcp_now` minus `ts_recent_age` to `TCP_PAWS_IDLE` in Figure 28.22 is also subject to sign bit wrap around, if the connection is idle for a period much longer than 24 days. With the 500-ms timestamp clock used by Net/3, when does this become a problem?

28.8 Read RFC 1323 to find out why RST segments are exempt from the PAWS test in Figure 28.22.

28.9 A client sends a SYN and the server responds with a SYN/ACK. The client moves to the ESTABLISHED state and responds with an ACK, but this ACK is lost. The server resends its SYN/ACK. Describe the processing steps when the client receives this duplicate SYN/ACK.

28.10 A client and server have an established connection and the server performs the active close. The connection terminates normally and the socket pair goes into the TIME_WAIT state on the server. Before this 2MSL wait expires on the server, the same client (i.e., the same socket pair on the client) sends a SYN to the server's socket pair but with a sequence number that is less than the ending sequence number from the previous incarnation of this connection. Describe what happens.

29

TCP Input (Continued)

29.1 Introduction

This chapter continues the discussion of TCP input processing, picking up where the previous chapter left off. Recall that the final test in Figure 28.37 was that either the ACK flag was set or, if not, the segment was dropped.

The ACK flag is handled, the window information is updated, the URG flag is processed, and any data in the segment is processed. Finally the FIN flag is processed and `tcp_output` is called, if required.

29.2 ACK Processing Overview

We begin this chapter with ACK processing, a summary of which is shown in Figure 29.1. The SYN_RCVD state is handled specially, followed by common processing for all remaining states. (Remember that a received ACK in either the LISTEN or SYN_SENT state was discussed in the previous chapter.) This is followed by special processing for the three states in which a received ACK causes a state transition, and for the TIME_WAIT state, in which the receipt of an ACK causes the 2MSL timer to be restarted.

29.3 Completion of Passive Opens and Simultaneous Opens

The first part of the ACK processing, shown in Figure 29.2, handles the SYN_RCVD state. As mentioned in the previous chapter, this handles the completion of a passive open (the common case) and also handles simultaneous opens and self-connects (the infrequent case).

```
switch (tp->t_state) {

case TCPS_SYN_RECEIVED:
    complete processing of passive open and process
        simultaneous open or self-connect;
    /* fall into ... */

case TCPS_ESTABLISHED:
case TCPS_FIN_WAIT_1:
case TCPS_FIN_WAIT_2:
case TCPS_CLOSE_WAIT:
case TCPS_CLOSING:
case TCPS_LAST_ACK:
case TCPS_TIME_WAIT:
    process duplicate ACK;
    update RTT estimators;
    if all outstanding data ACKed, turn off retransmission timer;
    remove ACKed data from socket send buffer;
    switch (tp->t_state) {
    case TCPS_FIN_WAIT_1:
        if (FIN is ACKed) {
            move to FIN_WAIT_2 state;
            start FIN_WAIT_2 timer;
        }
        break;
    case TCPS_CLOSING:
        if (FIN is ACKed) {
            move to TIME_WAIT state;
            start TIME_WAIT timer;
        }
        break;
    case TCPS_LAST_ACK:
        if (FIN is ACKed)
            move to CLOSED state;
        break;
    case TCPS_TIME_WAIT:
        restart TIME_WAIT timer;
        goto dropafterack;
    }
}
```

Figure 29.1 Summary of ACK processing.

Verify received ACK

801–806 For the ACK to acknowledge the SYN that was sent, it must be greater than `snd_una` (which is set to the ISS for the connection, the sequence number of the SYN, by `tcp_sendseqinit`) and less than or equal to `snd_max`. If so, the socket is marked as connected and the state becomes ESTABLISHED.

tcp_input.c

```
791     /*
792      * Ack processing.
793      */
794     switch (tp->t_state) {

795         /*
796          * In SYN_RECEIVED state if the ack ACKs our SYN then enter
797          * ESTABLISHED state and continue processing, otherwise
798          * send an RST.
799          */
800     case TCPS_SYN_RECEIVED:
801         if (SEQ_GT(tp->snd_una, ti->ti_ack) ||
802             SEQ_GT(ti->ti_ack, tp->snd_max))
803             goto dropwithreset;
804         tcpstat.tcps_connects++;
805         soisconnected(so);
806         tp->t_state = TCPS_ESTABLISHED;
807         /* Do window scaling? */
808         if ((tp->t_flags & (TF_RCVD_SCALE | TF_REQ_SCALE)) ==
809             (TF_RCVD_SCALE | TF_REQ_SCALE)) {
810             tp->snd_scale = tp->requested_s_scale;
811             tp->rcv_scale = tp->request_r_scale;
812         }
813         (void) tcp_reass(tp, (struct tcpiphdr *) 0, (struct mbuf *) 0);
814         tp->snd_wl1 = ti->ti_seq - 1;
815         /* fall into ... */
```

tcp_input.c

Figure 29.2 `tcp_input` function: received ACK in SYN_RCVD state.

Since `soisconnected` wakes up the process that performed the passive open (normally a server), we see that this doesn't occur until the last of the three segments in the three-way handshake has been received. If the server is blocked in a call to `accept`, that call now returns; if the server is blocked in a call to `select` waiting for the listening descriptor to become readable, it is now readable.

Check for window scale option

807–812 If TCP sent a window scale option and received one, the send and receive scale factors are saved in the TCP control block. Otherwise the default values of `snd_scale` and `rcv_scale` in the TCP control block are 0 (no scaling).

Pass queued data to process

813 Any data queued for the connection can now be passed to the process. This is done by `tcp_reass` with a null pointer as the second argument. This data would have arrived with the SYN that moved the connection into the SYN_RCVD state.

814 `snd_wl1` is set to the received sequence number minus 1. We'll see in Figure 29.15 that this causes the three window update variables to be updated.

29.4 Fast Retransmit and Fast Recovery Algorithms

The next part of ACK processing, shown in Figure 29.3, handles duplicate ACKs and determines if TCP's fast retransmit and fast recovery algorithms [Jacobson 1990c] should come into play. The two algorithms are separate but are normally implemented together [Floyd 1994].

- The *fast retransmit* algorithm occurs when TCP deduces from a small number (normally 3) of consecutive duplicate ACKs that a segment has been lost and deduces the starting sequence number of the missing segment. The missing segment is retransmitted. The algorithm is mentioned in Section 4.2.2.21 of RFC 1122, which states that TCP may generate an immediate ACK when an out-of-order segment is received. We saw that Net/3 generates the immediate duplicate ACKs in Figure 27.15. This algorithm first appeared in the 4.3BSD Tahoe release and the subsequent Net/1 release. In these two implementations, after the missing segment was retransmitted, the slow start phase was entered.
- The *fast recovery* algorithm says that after the fast retransmit algorithm (that is, after the missing segment has been retransmitted), congestion avoidance but not slow start is performed. This is an improvement that allows higher throughput under moderate congestion, especially for large windows. This algorithm appeared in the 4.3BSD Reno release and the subsequent Net/2 release.

Net/3 implements both fast retransmit and fast recovery, as we describe shortly.

In the discussion of Figure 24.17 we noted that an acceptable ACK must be in the range

`snd_una` < acknowledgment field <= `snd_max`

This first test of the acknowledgment field compares it only to `snd_una`. The comparison against `snd_max` is in Figure 29.5. The reason for separating the tests is so that the following five tests can be applied to the received segment:

1. If the acknowledgment field is less than or equal to `snd_una`, and
2. the length of the received segment is 0, and
3. the advertised window (`tiwin`) has not changed, and
4. TCP has outstanding data that has not been acknowledged (the retransmission timer is nonzero), and
5. the received segment contains the biggest ACK TCP has seen (the acknowledgment field equals `snd_una`),

then this segment is a completely duplicate ACK. (Tests 1, 2, and 3 are in Figure 29.3; tests 4 and 5 are at the beginning of Figure 29.4.)

TCP counts the number of these duplicate ACKs that are received in a row (in the variable `t_dupacks`), and when the number reaches a threshold of 3 (`tcprexmtthresh`), the lost segment is retransmitted. This is the *fast retransmit* algorithm described in Section 21.7 of Volume 1. It works in conjunction with the code we

tcp_input.c

```
    /*
     * In ESTABLISHED state: drop duplicate ACKs; ACK out-of-range
     * ACKs.  If the ack is in the range
     *  tp->snd_una < ti->ti_ack <= tp->snd_max
     * then advance tp->snd_una to ti->ti_ack and drop
     * data from the retransmission queue.  If this ACK reflects
     * more up-to-date window information we update our window information.
     */
case TCPS_ESTABLISHED:
case TCPS_FIN_WAIT_1:
case TCPS_FIN_WAIT_2:
case TCPS_CLOSE_WAIT:
case TCPS_CLOSING:
case TCPS_LAST_ACK:
case TCPS_TIME_WAIT:

    if (SEQ_LEQ(ti->ti_ack, tp->snd_una)) {
        if (ti->ti_len == 0 && tiwin == tp->snd_wnd) {
            tcpstat.tcps_rcvdupack++;
            /*
             * If we have outstanding data (other than
             * a window probe), this is a completely
             * duplicate ack (ie, window info didn't
             * change), the ack is the biggest we've
             * seen and we've seen exactly our rexmt
             * threshold of them, assume a packet
             * has been dropped and retransmit it.
             * Kludge snd_nxt & the congestion
             * window so we send only this one
             * packet.
             *
             * We know we're losing at the current
             * window size so do congestion avoidance
             * (set ssthresh to half the current window
             * and pull our congestion window back to
             * the new ssthresh).
             *
             * Dup acks mean that packets have left the
             * network (they're now cached at the receiver)
             * so bump cwnd by the amount in the receiver
             * to keep a constant cwnd packets in the
             * network.
             */
```

tcp_input.c

Figure 29.3 `tcp_input` function: check for completely duplicate ACK.

saw in Figure 27.15: when TCP receives an out-of-order segment, it is required to generate an immediate duplicate ACK, telling the other end that a segment might have been been lost and telling it the value of the next expected sequence number. The goal of the fast retransmit algorithm is for TCP to retransmit immediately what appears to be the missing segment, instead of waiting for the retransmission timer to expire. Figure 21.7 of Volume 1 gives a detailed example of how this algorithm works.

The receipt of a duplicate ACK also tells TCP that a packet has "left the network," because the other end had to receive an out-of-order segment to send the duplicate ACK. The *fast recovery* algorithm says that after some number of consecutive duplicate ACKs have been received, TCP should perform congestion avoidance (i.e., slow down) but need not wait for the pipe to empty between the two connection end points (slow start). The expression "a packet has left the network" means a packet has been received by the other end and has been added to the out-of-order queue for the connection. The packet is not still in transit somewhere between the two end points.

If only the first three tests shown earlier are true, the ACK is still a duplicate and is counted by the statistic `tcps_rcvdupack`, but the counter of the number of consecutive duplicate ACKs for this connection (`t_dupacks`) is reset to 0. If only the first test is true, the counter `t_dupacks` is reset to 0.

The remainder of the fast recovery algorithm is shown in Figure 29.4. When all five tests are true, the fast recovery algorithm processes the segment depending on the number of these consecutive duplicate ACKs that have been received.

1. `t_dupacks` equals 3 (`tcprexmtthresh`). Congestion avoidance is performed and the missing segment is retransmitted.
2. `t_dupacks` exceeds 3. Increase the congestion window and perform normal TCP output.
3. `t_dupacks` is less than 3. Do nothing.

Number of consecutive duplicate ACKs reaches threshold of 3

861–868 When `t_dupacks` reaches 3 (`tcprexmtthresh`), the value of `snd_nxt` is saved in `onxt` and the slow start threshold (`ssthresh`) is set to one-half the current congestion window, with a minimum value of two segments. This is what was done with the slow start threshold when the retransmission timer expired in Figure 25.27, but we'll see later in this piece of code that the fast recovery algorithm does not set the congestion window to one segment, as was done with the timeout.

Turn off retransmission timer

869–870 The retransmission timer is turned off and, in case a segment is currently being timed, `t_rtt` is set to 0.

Retransmit missing segment

871–873 `snd_nxt` is set to the starting sequence number of the segment that appears to have been lost (the acknowledgment field of the duplicate ACK) and the congestion window is set to one segment. This causes `tcp_output` to send only the missing segment. (This is shown by segment 63 in Figure 21.7 of Volume 1.)

Set congestion window

874–875 The congestion window is set to the slow start threshold plus the number of segments that the other end has cached. By *cached* we mean the number of out-of-order segments that the other end has received and generated duplicate ACKs for. These cannot be passed to the process at the other end until the missing segment (which was just

tcp_input.c

```
858                 if (tp->t_timer[TCPT_REXMT] == 0 ||
859                     ti->ti_ack != tp->snd_una)
860                     tp->t_dupacks = 0;
861                 else if (++tp->t_dupacks == tcprexmtthresh) {
862                     tcp_seq onxt = tp->snd_nxt;
863                     u_int win =
864                         min(tp->snd_wnd, tp->snd_cwnd) / 2 /
865                             tp->t_maxseg;

866                     if (win < 2)
867                         win = 2;
868                     tp->snd_ssthresh = win * tp->t_maxseg;
869                     tp->t_timer[TCPT_REXMT] = 0;
870                     tp->t_rtt = 0;
871                     tp->snd_nxt = ti->ti_ack;
872                     tp->snd_cwnd = tp->t_maxseg;
873                     (void) tcp_output(tp);
874                     tp->snd_cwnd = tp->snd_ssthresh +
875                         tp->t_maxseg * tp->t_dupacks;
876                     if (SEQ_GT(onxt, tp->snd_nxt))
877                         tp->snd_nxt = onxt;
878                     goto drop;
879                 } else if (tp->t_dupacks > tcprexmtthresh) {
880                     tp->snd_cwnd += tp->t_maxseg;
881                     (void) tcp_output(tp);
882                     goto drop;
883                 }
884             } else
885                 tp->t_dupacks = 0;
886             break;              /* beyond ACK processing (to step 6) */
887         }
```

tcp_input.c

Figure 29.4 `tcp_input` function: duplicate ACK processing.

sent) is received. Figures 21.10 and 21.11 in Volume 1 show what happens with the congestion window and slow start threshold when the fast recovery algorithm is in effect.

Set `snd_nxt`

876–878 The value of the next sequence number to send is set to the maximum of its previous value (`onxt`) and its current value. Its current value was modified by `tcp_output` when the segment was retransmitted. Normally this causes `snd_nxt` to be set back to its previous value, which means that only the missing segment is retransmitted, and that future calls to `tcp_output` carry on with the next segment in sequence.

Number of consecutive duplicate ACKs exceeds threshold of 3

879–883 The missing segment was retransmitted when `t_dupacks` equaled 3, so the receipt of each additional duplicate ACK means that another packet has left the network. The congestion window is incremented by one segment. `tcp_output` sends the next segment in sequence, and the duplicate ACK is dropped. (This is shown by segments 67, 69, and 71 in Figure 21.7 of Volume 1.)

884–885 This statement is executed when the received segment contains a duplicate ACK, but either the length is nonzero or the advertised window changed. Only the first of the five tests described earlier is true. The counter of consecutive duplicate ACKs is set to 0.

Skip remainder of ACK processing

886 This `break` is executed in three cases: (1) only the first of the five tests described earlier is true, or (2) only the first three of the five tests is true, or (3) the ACK is a duplicate, but the number of consecutive duplicates is less than the threshold of 3. For any of these cases the ACK is still a duplicate and the `break` goes to the end of the `switch` that started in Figure 29.2, which continues processing at the label `step6`.

To understand the purpose in this aggressive window manipulation, consider the following example. Assume the window is eight segments, and segments 1 through 8 are sent. Segment 1 is lost, but the remainder arrive OK and are acknowledged. After the ACKs for segments 2, 3, and 4 arrive, the missing segment (1) is retransmitted. TCP would like the subsequent ACKs for 5 through 8 to allow some of the segments starting with 9 to be sent, to keep the pipe full. But the window is 8, which prevents segments 9 and above from being sent. Therefore, the congestion window is temporarily inflated by one segment each time another duplicate ACK is received, since the receipt of the duplicate ACK tells TCP that another segment has left the pipe at the other end. When the acknowledgment of segment 1 is finally received, the next figure reduces the congestion window back to the slow start threshold. This increase in the congestion window as the duplicate ACKs arrive, and its subsequent decrease when the fresh ACK arrives, can be seen visually in Figure 21.10 of Volume 1.

29.5 ACK Processing

The ACK processing continues with Figure 29.5.

tcp_input.c

```
        /*
         * If the congestion window was inflated to account
         * for the other side's cached packets, retract it.
         */
        if (tp->t_dupacks > tcprexmtthresh &&
            tp->snd_cwnd > tp->snd_ssthresh)
            tp->snd_cwnd = tp->snd_ssthresh;
        tp->t_dupacks = 0;

        if (SEQ_GT(ti->ti_ack, tp->snd_max)) {
            tcpstat.tcps_rcvacktoomuch++;
            goto dropafterack;
        }
        acked = ti->ti_ack - tp->snd_una;
        tcpstat.tcps_rcvackpack++;
        tcpstat.tcps_rcvackbyte += acked;
```

tcp_input.c

Figure 29.5 `tcp_input` function: ACK processing continued.

Adjust congestion window

888–895 If the number of consecutive duplicate ACKs exceeds the threshold of 3, this is the first nonduplicate ACK after a string of four or more duplicate ACKs. The fast recovery algorithm is complete. Since the congestion window was incremented by one segment for every consecutive duplicate after the third, if it now exceeds the slow start threshold, it is set back to the slow start threshold. The counter of consecutive duplicate ACKs is set to 0.

Check for out-of-range ACK

896–899 Recall the definition of an acceptable ACK,

`snd_una` < acknowledgment field <= `snd_max`

If the acknowledgment field is greater than `snd_max`, the other end is acknowledging data that TCP hasn't even sent yet! This probably occurs on a high-speed connection when the sequence numbers wrap and a missing ACK reappears later. As we can see in Figure 24.5, this rarely happens (since today's networks aren't fast enough).

Calculate number of bytes acknowledged

900–902 At this point TCP knows that it has an acceptable ACK. `acked` is the number of bytes acknowledged.

The next part of ACK processing, shown in Figure 29.6, deals with RTT measurements and the retransmission timer.

Update RTT estimators

903–915 If either (1) a timestamp option was present, or (2) a segment was being timed and the acknowledgment number is greater than the starting sequence number of the segment being timed, `tcp_xmit_timer` updates the RTT estimators. Notice that the second argument to this function when timestamps are used is the current time (`tcp_now`) minus the timestamp echo reply (`ts_ecr`) plus 1 (since the function subtracts 1).

Delayed ACKs are the reason for the greater-than test of the sequence numbers. For example, if TCP sends and times a segment with bytes 1–1024, followed by a segment with bytes 1025–2048, if an ACK of 2049 is returned, this test will consider whether 2049 is greater than 1 (the starting sequence number of the segment being timed), and since this is true, the RTT estimators are updated.

Check if all outstanding data has been acknowledged

916–924 If the acknowledgment field of the received segment (`ti_ack`) equals the maximum sequence number that TCP has sent (`snd_max`), all outstanding data has been acknowledged. The retransmission timer is turned off and the `needoutput` flag is set to 1. This flag forces a call to `tcp_output` at the end of this function. Since there is no more data waiting to be acknowledged, TCP may have more data to send that it has not been able to send earlier because the data was beyond the right edge of the window. Now that a new ACK has been received, the window will probably move to the right (`snd_una` is updated in Figure 29.8), which could allow more data to be sent.

tcp_input.c

```
903         /*
904          * If we have a timestamp reply, update smoothed
905          * round-trip time.  If no timestamp is present but
906          * transmit timer is running and timed sequence
907          * number was acked, update smoothed round-trip time.
908          * Since we now have an rtt measurement, cancel the
909          * timer backoff (cf., Phil Karn's retransmit alg.).
910          * Recompute the initial retransmit timer.
911          */
912         if (ts_present)
913             tcp_xmit_timer(tp, tcp_now - ts_ecr + 1);
914         else if (tp->t_rtt && SEQ_GT(ti->ti_ack, tp->t_rtseq))
915             tcp_xmit_timer(tp, tp->t_rtt);

916         /*
917          * If all outstanding data is acked, stop retransmit
918          * timer and remember to restart (more output or persist).
919          * If there is more data to be acked, restart retransmit
920          * timer, using current (possibly backed-off) value.
921          */
922         if (ti->ti_ack == tp->snd_max) {
923             tp->t_timer[TCPT_REXMT] = 0;
924             needoutput = 1;
925         } else if (tp->t_timer[TCPT_PERSIST] == 0)
926             tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;
```

tcp_input.c

Figure 29.6 `tcp_input` function: RTT measurements and retransmission timer.

Unacknowledged data outstanding

925–926 Since there is additional data that has been sent but not acknowledged, if the persist timer is not on, the retransmission timer is restarted using the current value of `t_rxtcur`.

Karn's Algorithm and Timestamps

Notice that timestamps overrule the portion of Karn's algorithm (Section 21.3 of Volume 1) that says: when a timeout and retransmission occurs, the RTT estimators cannot be updated when the acknowledgment for the retransmitted data is received (the *retransmission ambiguity problem*). In Figure 25.26 we saw that `t_rtt` was set to 0 when a retransmission took place, because of Karn's algorithm. If timestamps are not present and it is a retransmission, the code in Figure 29.6 does not update the RTT estimators because `t_rtt` will be 0 from the retransmission. But if a timestamp is present, `t_rtt` isn't examined, allowing the RTT estimators to be updated using the received timestamp echo reply. With RFC 1323 timestamps the ambiguity is gone since the `ts_ecr` value was copied by the other end from the segment being acknowledged. The other half of Karn's algorithm, specifying that an exponential backoff must be used with retransmissions, still holds, of course.

Figure 29.7 shows the next part of ACK processing, updating the congestion window.

tcp_input.c

```
927         /*
928          * When new data is acked, open the congestion window.
929          * If the window gives us less than ssthresh packets
930          * in flight, open exponentially (maxseg per packet).
931          * Otherwise open linearly: maxseg per window
932          * (maxseg^2 / cwnd per packet), plus a constant
933          * fraction of a packet (maxseg/8) to help larger windows
934          * open quickly enough.
935          */
936         {
937             u_int   cw = tp->snd_cwnd;
938             u_int   incr = tp->t_maxseg;

939             if (cw > tp->snd_ssthresh)
940                 incr = incr * incr / cw + incr / 8;
941             tp->snd_cwnd = min(cw + incr, TCP_MAXWIN << tp->snd_scale);
942         }
```

tcp_input.c

Figure 29.7 `tcp_input` function: open congestion window in response to ACKs.

Update congestion window

927–942 One of the rules of slow start and congestion avoidance is that a received ACK increases the congestion window. By default the congestion window is increased by one segment for each received ACK (slow start). But if the current congestion window is greater than the slow start threshold, it is increased by 1 divided by the congestion window, plus a constant fraction of a segment. The term

```
incr * incr / cw
```

is

```
t_maxseg * t_maxseg / snd_cwnd
```

which is 1 divided by the congestion window, taking into account that `snd_cwnd` is maintained in bytes, not segments. The constant fraction is the segment size divided by 8. The congestion window is then limited by the maximum value of the send window for this connection. Example calculations of this algorithm are in Section 21.8 of Volume 1.

> Adding in the constant fraction (the segment size divided by 8) is wrong [Floyd 1994]. But it has been in the BSD sources since 4.3BSD Reno and is still in 4.4BSD and Net/3. It should be removed.

The next part of `tcp_input`, shown in Figure 29.8, removes the acknowledged data from the send buffer.

tcp_input.c

```
943         if (acked > so->so_snd.sb_cc) {
944             tp->snd_wnd -= so->so_snd.sb_cc;
945             sbdrop(&so->so_snd, (int) so->so_snd.sb_cc);
946             ourfinisacked = 1;
947         } else {
948             sbdrop(&so->so_snd, acked);
949             tp->snd_wnd -= acked;
950             ourfinisacked = 0;
951         }
952         if (so->so_snd.sb_flags & SB_NOTIFY)
953             sowwakeup(so);
954         tp->snd_una = ti->ti_ack;
955         if (SEQ_LT(tp->snd_nxt, tp->snd_una))
956             tp->snd_nxt = tp->snd_una;
```

tcp_input.c

Figure 29.8 `tcp_input` function: remove acknowledged data from send buffer.

Remove acknowledged bytes from the send buffer

943–946 If the number of bytes acknowledged *exceeds* the number of bytes on the send buffer, `snd_wnd` is decremented by the number of bytes in the send buffer and TCP knows that its FIN has been ACKed. That number of bytes is then removed from the send buffer by `sbdrop`. This method for detecting the ACK of a FIN works only because the FIN occupies 1 byte in the sequence number space.

947–951 Otherwise the number of bytes acknowledged is less than or equal to the number of bytes in the send buffer, so `ourfinisacked` is set to 0, and `acked` bytes of data are dropped from the send buffer.

Wakeup processes waiting on send buffer

951–956 `sowwakeup` awakens any processes waiting on the send buffer. `snd_una` is updated to contain the oldest unacknowledged sequence number. If this new value of `snd_una` exceeds `snd_nxt`, the latter is updated, since the intervening bytes have been acknowledged.

Figure 29.9 shows how `snd_nxt` can end up with a sequence number that is less than `snd_una`. Assume two segments are transmitted, the first with bytes 1–512 and the second with bytes 513–1024.

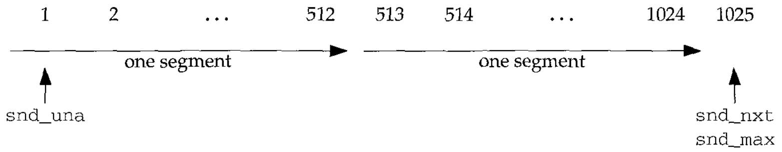


Figure 29.9 Two segments sent on a connection.

The retransmission timer then expires before an acknowledgment is returned. The code in Figure 25.26 sets `snd_nxt` back to `snd_una`, slow start is entered, `tcp_output` is called, and one segment containing bytes 1–512 is retransmitted. `tcp_output` increases `snd_nxt` to 513, and we have the scenario shown in Figure 29.10.

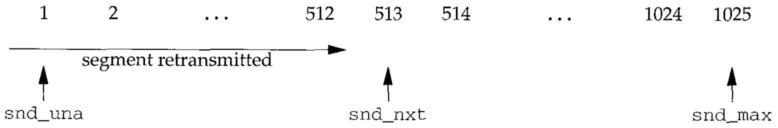


Figure 29.10 Continuation of Figure 29.9 after retransmission timer expires.

At this point an ACK of 1025 arrives (either the two original segments or the ACK was delayed somewhere in the network). The ACK is valid since it is less than or equal to `snd_max`, but `snd_nxt` will be less than the updated value of `snd_una`.

The general ACK processing is now complete, and the `switch` shown in Figure 29.11 handles four special cases.

tcp_input.c

```
957         switch (tp->t_state) {

958             /*
959              * In FIN_WAIT_1 state in addition to the processing
960              * for the ESTABLISHED state if our FIN is now acknowledged
961              * then enter FIN_WAIT_2.
962              */
963         case TCPS_FIN_WAIT_1:
964             if (ourfinisacked) {
965                 /*
966                  * If we can't receive any more
967                  * data, then closing user can proceed.
968                  * Starting the timer is contrary to the
969                  * specification, but if we don't get a FIN
970                  * we'll hang forever.
971                  */
972                 if (so->so_state & SS_CANTRCVMORE) {
973                     soisdisconnected(so);
974                     tp->t_timer[TCPT_2MSL] = tcp_maxidle;
975                 }
976                 tp->t_state = TCPS_FIN_WAIT_2;
977             }
978             break;
```

tcp_input.c

Figure 29.11 `tcp_input` function: receipt of ACK in FIN_WAIT_1 state.

Receipt of ACK in FIN_WAIT_1 state

958–971 In this state the process has closed the connection and TCP has sent the FIN. But other ACKs can be received for data segments sent before the FIN. Therefore the connection moves into the FIN_WAIT_2 state only when the FIN has been acknowledged. The flag `ourfinisacked` is set in Figure 29.8; this depends on whether the number of bytes ACKed exceeds the amount of data in the send buffer or not.

Set FIN_WAIT_2 timer

972–975 We also described in Section 25.6 how Net/3 sets a FIN_WAIT_2 timer to prevent an infinite wait in the FIN_WAIT_2 state. This timer is set only if the process completely closed the connection (i.e., the `close` system call or its kernel equivalent if the process was terminated by a signal), and not if the process performed a half-close (i.e., the FIN was sent but the process can still receive data on the connection).

Figure 29.12 shows the receipt of an ACK in the CLOSING state.

tcp_input.c

```
979             /*
980              * In CLOSING state in addition to the processing for
981              * the ESTABLISHED state if the ACK acknowledges our FIN
982              * then enter the TIME-WAIT state, otherwise ignore
983              * the segment.
984              */
985         case TCPS_CLOSING:
986             if (ourfinisacked) {
987                 tp->t_state = TCPS_TIME_WAIT;
988                 tcp_canceltimers(tp);
989                 tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
990                 soisdisconnected(so);
991             }
992             break;
```

tcp_input.c

Figure 29.12 `tcp_input` function: receipt of ACK in CLOSING state.

Receipt of ACK in CLOSING state

979–992 If the ACK is for the FIN (and not for some previous data segment), the connection moves into the TIME_WAIT state. Any pending timers are cleared (such as a pending retransmission timer), and the TIME_WAIT timer is started with a value of twice the MSL.

The processing of an ACK in the LAST_ACK state is shown in Figure 29.13.

tcp_input.c

```
 993            /*
 994             * In LAST_ACK, we may still be waiting for data to drain
 995             * and/or to be acked, as well as for the ack of our FIN.
 996             * If our FIN is now acknowledged, delete the TCB,
 997             * enter the closed state, and return.
 998             */
 999        case TCPS_LAST_ACK:
1000            if (ourfinisacked) {
1001                tp = tcp_close(tp);
1002                goto drop;
1003            }
1004            break;
```

tcp_input.c

Figure 29.13 `tcp_input` function: receipt of ACK in LAST_ACK state.

Receipt of ACK in LAST_ACK state

993–1004 If the FIN is ACKed, the new state is CLOSED. This state transition is handled by `tcp_close`, which also releases the Internet PCB and TCP control block.

Figure 29.14 shows the processing of an ACK in the TIME_WAIT state.

tcp_input.c

```
1005            /*
1006             * In TIME_WAIT state the only thing that should arrive
1007             * is a retransmission of the remote FIN.  Acknowledge
1008             * it and restart the finack timer.
1009             */
1010        case TCPS_TIME_WAIT:
1011            tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
1012            goto dropafterack;
1013        }
1014    }
```

tcp_input.c

Figure 29.14 `tcp_input` function: receipt of ACK in TIME_WAIT state.

Receipt of ACK in TIME_WAIT state

1005–1014 In this state both ends have sent a FIN and both FINs have been acknowledged. If TCP's ACK of the remote FIN was lost, however, the other end will retransmit the FIN (with an ACK). TCP drops the segment and resends the ACK. Additionally, the TIME_WAIT timer must be restarted with a value of twice the MSL.

29.6 Update Window Information

There are two variables in the TCP control block that we haven't described yet: `snd_wl1` and `snd_wl2`.

- `snd_wl1` records the sequence number of the last segment used to update the send window (`snd_wnd`).
- `snd_wl2` records the acknowledgment number of the last segment used to update the send window.

Our only encounter with these variables so far was when a connection was established (active, passive, or simultaneous open) and `snd_wl1` was set to `ti_seq` minus 1. We said this was to guarantee a window update, which we'll see in the following code.

The send window (`snd_wnd`) is updated from the advertised window in the received segment (`tiwin`) if any one of the following three conditions is true:

1. The segment contains new data. Since `snd_wl1` contains the starting sequence number of the last segment that was used to update the send window, if

   ```
   snd_wl1 < ti_seq
   ```

   this condition is true.

2. The segment does not contain new data (`snd_wl1` equals `ti_seq`), but the segment acknowledges new data. The latter condition is true if

   `snd_wl2 < ti_ack`

   since `snd_wl2` records the acknowledgment number of the last segment that updated the send window.
3. The segment does not contain new data, and the segment does not acknowledge new data, but the advertised window is larger than the current send window.

The purpose of these tests is to prevent an old segment from affecting the send window, since the send window is not an absolute sequence number, but is an offset from `snd_una`.

Figure 29.15 shows the code that implements the update of the send window.

tcp_input.c

```
1015  step6:
1016    /*
1017     * Update window information.
1018     * Don't look at window if no ACK: TAC's send garbage on first SYN.
1019     */
1020    if ((tiflags & TH_ACK) &&
1021        (SEQ_LT(tp->snd_wl1, ti->ti_seq) || tp->snd_wl1 == ti->ti_seq &&
1022         (SEQ_LT(tp->snd_wl2, ti->ti_ack) ||
1023          tp->snd_wl2 == ti->ti_ack && tiwin > tp->snd_wnd))) {

1024        /* keep track of pure window updates */
1025        if (ti->ti_len == 0 &&
1026            tp->snd_wl2 == ti->ti_ack && tiwin > tp->snd_wnd)
1027            tcpstat.tcps_rcvwinupd++;

1028        tp->snd_wnd = tiwin;
1029        tp->snd_wl1 = ti->ti_seq;
1030        tp->snd_wl2 = ti->ti_ack;
1031        if (tp->snd_wnd > tp->max_sndwnd)
1032            tp->max_sndwnd = tp->snd_wnd;
1033        needoutput = 1;
1034    }
```

tcp_input.c

Figure 29.15 `tcp_input` function: update window information.

Check if send window should be updated

1015–1023 This `if` test verifies that the ACK flag is set along with any one of the three previously stated conditions. Recall that a jump was made to `step6` after the receipt of a SYN in either the LISTEN or SYN_SENT state, and in the LISTEN state the SYN does not contain an ACK.

The term *TAC* referred to in the comment is a "terminal access controller." These were Telnet clients on the ARPANET.

1024–1027 If the received segment is a pure window update (the length is 0 and the ACK does not acknowledge new data, but the advertised window is larger), the statistic `tcps_rcvwinupd` is incremented.

Update variables

1028–1033 The send window is updated and new values of `snd_wl1` and `snd_wl2` are recorded. Additionally, if this advertised window is the largest one TCP has received from this peer, the new value is recorded in `max_sndwnd`. This is an attempt to guess the size of the other end's receive buffer, and it is used in Figure 26.8. `needoutput` is set to 1 since the new value of `snd_wnd` might enable a segment to be sent.

29.7 Urgent Mode Processing

The next part of TCP input processing handles segments with the URG flag set.

tcp_input.c

```
1035     /*
1036      * Process segments with URG.
1037      */
1038     if ((tiflags & TH_URG) && ti->ti_urp &&
1039         TCPS_HAVERCVDFIN(tp->t_state) == 0) {
1040         /*
1041          * This is a kludge, but if we receive and accept
1042          * random urgent pointers, we'll crash in
1043          * soreceive.  It's hard to imagine someone
1044          * actually wanting to send this much urgent data.
1045          */
1046         if (ti->ti_urp + so->so_rcv.sb_cc > sb_max) {
1047             ti->ti_urp = 0;      /* XXX */
1048             tiflags &= ~TH_URG; /* XXX */
1049             goto dodata;        /* XXX */
1050         }
```

tcp_input.c

Figure 29.16 `tcp_input` function: urgent mode processing.

Check if URG flag should be processed

1035–1039 These segments must have the URG flag set, a nonzero urgent offset (`ti_urp`), and the connection must not have received a FIN. The macro `TCPS_HAVERCVDFIN` is true only for the TIME_WAIT state, so the URG is processed in any other state. This is contrary to a comment appearing later in the code stating that the URG flag is ignored in the CLOSE_WAIT, CLOSING, LAST_ACK, or TIME_WAIT states.

Ignore bogus urgent offsets

1040–1050 If the urgent offset plus the number of bytes already in the receive buffer exceeds the maximum size of a socket buffer, the urgent notification is ignored. The urgent offset is set to 0, the URG flag is cleared, and the rest of the urgent mode processing is skipped.

The next piece of code, shown in Figure 29.17, processes the urgent pointer.

tcp_input.c

```
1051         /*
1052          * If this segment advances the known urgent pointer,
1053          * then mark the data stream.  This should not happen
1054          * in CLOSE_WAIT, CLOSING, LAST_ACK or TIME_WAIT states since
1055          * a FIN has been received from the remote side.
1056          * In these states we ignore the URG.
1057          *
1058          * According to RFC961 (Assigned Protocols),
1059          * the urgent pointer points to the last octet
1060          * of urgent data.  We continue, however,
1061          * to consider it to indicate the first octet
1062          * of data past the urgent section as the original
1063          * spec states (in one of two places).
1064          */
1065         if (SEQ_GT(ti->ti_seq + ti->ti_urp, tp->rcv_up)) {
1066             tp->rcv_up = ti->ti_seq + ti->ti_urp;
1067             so->so_oobmark = so->so_rcv.sb_cc +
1068                 (tp->rcv_up - tp->rcv_nxt) - 1;
1069             if (so->so_oobmark == 0)
1070                 so->so_state |= SS_RCVATMARK;
1071             sohasoutofband(so);
1072             tp->t_oobflags &= ~(TCPOOB_HAVEDATA | TCPOOB_HADDATA);
1073         }
1074         /*
1075          * Remove out-of-band data so doesn't get presented to user.
1076          * This can happen independent of advancing the URG pointer,
1077          * but if two URG's are pending at once, some out-of-band
1078          * data may creep in... ick.
1079          */
1080         if (ti->ti_urp <= ti->ti_len
1081 #ifdef SO_OOBINLINE
1082             && (so->so_options & SO_OOBINLINE) == 0
1083 #endif
1084             )
1085             tcp_pulloutofband(so, ti, m);
1086     } else {
1087         /*
1088          * If no out-of-band data is expected, pull receive
1089          * urgent pointer along with the receive window.
1090          */
1091         if (SEQ_GT(tp->rcv_nxt, tp->rcv_up))
1092             tp->rcv_up = tp->rcv_nxt;
1093     }
```

tcp_input.c

Figure 29.17 `tcp_input` function: processing of received urgent pointer.

1051–1065 If the starting sequence number of the received segment plus the urgent offset exceeds the current receive urgent pointer, a new urgent pointer has been received. For example, when the 3-byte segment that was sent in Figure 26.30 arrives at the receiver, we have the scenario shown in Figure 29.18.

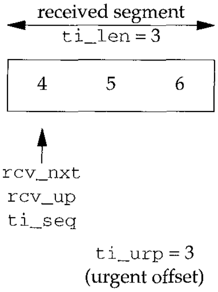


Figure 29.18 Receiver side when segment from Figure 26.30 arrives.

Normally the receive urgent pointer (`rcv_up`) equals `rcv_nxt`. In this example, since the `if` test is true (4 plus 3 is greater than 4), the new value of `rcv_up` is calculated as 7.

Calculate receive urgent pointer

1066–1070 The out-of-band mark in the socket's receive buffer is calculated, taking into account any data bytes already in the receive buffer (`so_rcv.sb_cc`). In our example, assuming there is no data already in the receive buffer, `so_oobmark` is set to 2: that is, the byte with the sequence number 6 is considered the out-of-band byte. If this out-of-band mark is 0, the socket is currently at the out-of-band mark. This happens if the `send` system call that sends the out-of-band byte specifies a length of 1, and if the receive buffer is empty when this segment arrives at the other end. This reiterates that Berkeley-derived systems consider the urgent pointer to point to the first byte of data *after* the out-of-band byte.

Notify process of TCP's urgent mode

1071–1072 `sohasoutofband` notifies the process that out-of-band data has arrived for the socket. The two flags `TCPOOB_HAVEDATA` and `TCPOOB_HADDATA` are cleared. These two flags are used with the `PRU_RCVOOB` request in Figure 30.8.

Pull out-of-band byte out of normal data stream

1074–1085 If the urgent offset is less than or equal to the number of bytes in the received segment, the out-of-band byte is contained in the segment. With TCP's urgent mode it is possible for the urgent offset to point to a data byte that has not yet been received. If the `SO_OOBINLINE` constant is defined (which it always is for Net/3), and if the corresponding socket option is not enabled, the receiving process wants the out-of-band byte pulled out of the normal stream of data and placed into the variable `t_iobc`. This is done by `tcp_pulloutofband`, which we cover in the next section.

Notice that the receiving process is notified that the sender has entered urgent mode, regardless of whether the byte pointed to by the urgent pointer is readable or not. This is a feature of TCP's urgent mode.

Adjust receive urgent pointer if not urgent mode

1086–1093 When the receiver is not processing an urgent pointer, if `rcv_nxt` is greater than the receive urgent pointer, `rcv_up` is moved to the right and set equal to `rcv_nxt`. This keeps the receive urgent pointer at the left edge of the receive window so that the comparison using `SEQ_GT` at the beginning of Figure 29.17 will work correctly when an URG flag is received.

If the solution to Exercise 26.6 is implemented, corresponding changes will have to go into Figures 29.16 and 29.17 also.

29.8 `tcp_pulloutofband` Function

This function is called from Figure 29.17 when

1. urgent mode notification arrives in a received segment, and
2. the out-of-band byte is contained within the segment (i.e., the urgent pointer points into the received segment), and
3. the `SO_OOBINLINE` socket option is not enabled for this socket.

This function removes the out-of-band byte from the normal stream of data (i.e., the mbuf chain containing the received segment) and places it into the `t_iobc` variable in the TCP control block for the connection. The process reads this variable using the `MSG_OOB` flag with the `recv` system call: the `PRU_RCVOOB` request in Figure 30.8. Figure 29.19 shows the function.

tcp_input.c

```
void
tcp_pulloutofband(so, ti, m)
struct socket *so;
struct tcpiphdr *ti;
struct mbuf *m;
{
        int     cnt = ti->ti_urp - 1;

        while (cnt >= 0) {
            if (m->m_len > cnt) {
                char    *cp = mtod(m, caddr_t) + cnt;
                struct tcpcb *tp = sototcpcb(so);

                tp->t_iobc = *cp;
                tp->t_oobflags |= TCPOOB_HAVEDATA;
                bcopy(cp + 1, cp, (unsigned) (m->m_len - cnt - 1));
                m->m_len--;
                return;
            }
            cnt -= m->m_len;
            m = m->m_next;
            if (m == 0)
                break;
        }
        panic("tcp_pulloutofband");
}
```

tcp_input.c

Figure 29.19 `tcp_pulloutofband` function: place out-of-band byte into `t_iobc`.

1282–1289 Consider the example in Figure 29.20. The urgent offset is 3, therefore the urgent pointer is 7, and the sequence number of the out-of-band byte is 6. There are 5 bytes in the received segment, all contained in a single mbuf.

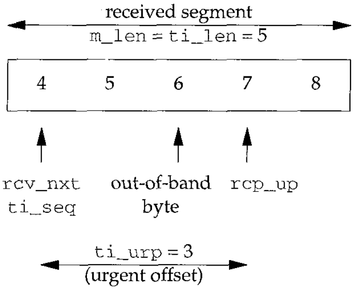


Figure 29.20 Received segment with an out-of-band byte.

The variable `cnt` is 2 and since `m_len` (which is 5) is greater than 2, the true portion of the `if` statement is executed.

1290–1298 `cp` points to the shaded byte with a sequence number of 6. This is placed into the variable `t_iobc`, which contains the out-of-band byte. The `TCPOOB_HAVEDATA` flag is set and `bcopy` moves the next 2 bytes (with sequence numbers 7 and 8) left 1 byte, giving the arrangement shown in Figure 29.21.

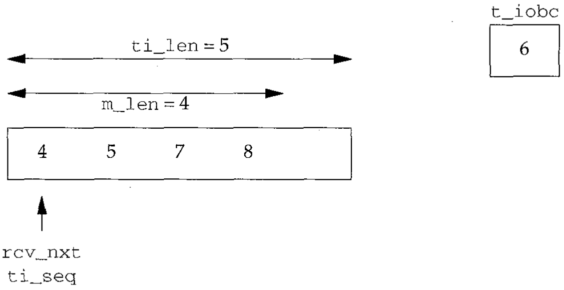


Figure 29.21 Result from Figure 29.20 after removal of out-of-band byte.

Remember that the numbers 7 and 8 specify the sequence numbers of the data bytes, not the contents of the data bytes. The length of the mbuf is decremented from 5 to 4 but `ti_len` is left as 5, for sequencing of the segment into the socket's receive buffer. Both the `TCP_REASS` macro and the `tcp_reass` function (which are called in the next section) increment `rcv_nxt` by `ti_len`, which in this example must be 5, because the next expected receive sequence number is 9. Also notice in this function that the length field in the packet header (`m_pkthdr.len`) in the first mbuf is not decremented by 1. This is because that length field is not used by `sbappend`, which appends the data to the socket's receive buffer.

Skip to next mbuf in chain

1299–1302 The out-of-band byte is not contained in this mbuf, so `cnt` is decremented by the number of bytes in the mbuf and the next mbuf in the chain is processed. Since this function is called only when the urgent offset points into the received segment, if there is not another mbuf on the chain, the `break` causes the call to `panic`.

29.9 Processing of Received Data

`tcp_input` continues by taking the received data (if any) and either appending it to the socket's receive buffer (if it is the next expected segment) or placing it onto the socket's out-of-order queue. Figure 29.22 shows the code that performs this task.

tcp_input.c

```
1094   dodata:                      /* XXX */
1095     /*
1096      * Process the segment text, merging it into the TCP sequencing queue,
1097      * and arranging for acknowledgment of receipt if necessary.
1098      * This process logically involves adjusting tp->rcv_wnd as data
1099      * is presented to the user (this happens in tcp_usrreq.c,
1100      * case PRU_RCVD).  If a FIN has already been received on this
1101      * connection then we just ignore the text.
1102      */
1103     if ((ti->ti_len || (tiflags & TH_FIN)) &&
1104         TCPS_HAVERCVDFIN(tp->t_state) == 0) {
1105         TCP_REASS(tp, ti, m, so, tiflags);
1106         /*
1107          * Note the amount of data that peer has sent into
1108          * our window, in order to estimate the sender's
1109          * buffer size.
1110          */
1111         len = so->so_rcv.sb_hiwat - (tp->rcv_adv - tp->rcv_nxt);
1112     } else {
1113         m_freem(m);
1114         tiflags &= ~TH_FIN;
1115     }
```

tcp_input.c

Figure 29.22 `tcp_input` function: merge received data into sequencing queue for socket.

1094–1105 Segment data is processed if

1. the length of the received data is greater than 0 or the FIN flag is set, and
2. a FIN has not yet been received for the connection.

The macro `TCP_REASS` processes the data. If the data is in sequence (i.e., the next expected data for this connection), the delayed-ACK flag is set, `rcv_nxt` is incremented, and the data is appended to the socket's receive buffer. If the data is out of order, the macro calls `tcp_reass` to add the data to the connection's reassembly queue (which might fill a hole and cause already-queued data to be appended to the socket's receive buffer).

Recall that the final argument to the macro (`tiflags`) can be modified. Specifically, if the data is out of order, `tcp_reass` sets `tiflags` to 0, clearing the FIN flag (if it was set). That's why the `if` statement is true if the FIN flag is set even if there is no data in the segment.

Consider the following example. A connection is established and the sender immediately transmits three segments: one with bytes 1–1024, another with bytes 1025–2048, and another with the FIN flag but no data. The first segment is lost, so when the second arrives (bytes 1025–2048) the receiver places it onto the out-of-order list and generates an immediate ACK. When the third segment with the FIN flag is received, the code in Figure 29.22 is executed. Even though the data length is 0, since the FIN flag is set, `TCP_REASS` is invoked, which calls `tcp_reass`. Since `ti_seq` (2049, the sequence number of the FIN) does not equal `rcv_nxt` (1), `tcp_reass` returns 0 (Figure 27.23), which in the `TCP_REASS` macro sets `tiflags` to 0. This clears the FIN flag, preventing the code that follows (Section 29.10) from processing the FIN flag.

Guess size of other end's send buffer

1106–1111 The calculation of `len` is attempt to guess the size of the other end's send buffer. Consider the following example. A socket has a receive buffer size of 8192 (the Net/3 default), so TCP advertises a window of 8192 in its SYN. The first segment with bytes 1–1024 is then received. Figure 29.23 shows the state of the receive space after `TCP_REASS` has incremented `rcv_nxt` to account for the received segment.

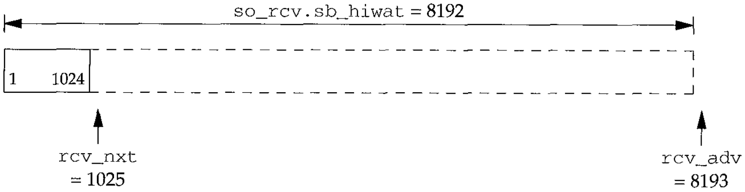


Figure 29.23 Receipt of bytes 1–1024 into a 8192-byte receive window.

The calculation of `len` yields 1024. The value of `len` will increase as the other end sends more data into the receive window, but it will never exceed the size of the other end's send buffer. Recall that the variable `max_sndwnd`, calculated in Figure 29.15, is an attempt to guess the size of the other end's receive buffer.

This variable `len` is never used! It is left over code from Net/1 when the variable `max_rcvd` was stored in the TCP control block after the calculation of `len`:

```
if (len > tp->max_rcvd)
    tp->max_rcvd = len;
```

But even in Net/1 the variable `max_rcvd` was never used.

1112–1115 If the length is 0 and the FIN flag is not set, or if a FIN has already been received for the connection, the received mbuf chain is discarded and the FIN flag is cleared.

29.10 FIN Processing

The next step in `tcp_input`, shown in Figure 29.24, handles the FIN flag.

tcp_input.c

```
1116    /*
1117     * If FIN is received ACK the FIN and let the user know
1118     * that the connection is closing.
1119     */
1120    if (tiflags & TH_FIN) {
1121        if (TCPS_HAVERCVDFIN(tp->t_state) == 0) {
1122            socantrcvmore(so);
1123            tp->t_flags |= TF_ACKNOW;
1124            tp->rcv_nxt++;
1125        }
1126        switch (tp->t_state) {

1127            /*
1128             * In SYN_RECEIVED and ESTABLISHED states
1129             * enter the CLOSE_WAIT state.
1130             */
1131        case TCPS_SYN_RECEIVED:
1132        case TCPS_ESTABLISHED:
1133            tp->t_state = TCPS_CLOSE_WAIT;
1134            break;
```

tcp_input.c

Figure 29.24 `tcp_input` function: FIN processing, first half.

Process first FIN received on connection

1116–1125 If the FIN flag is set and this is the first FIN received for this connection, `socantrcvmore` marks the socket as write-only, `TF_ACKNOW` is set to acknowledge the FIN immediately (i.e., it is not delayed), and `rcv_nxt` steps over the FIN in the sequence space.

1126 The remainder of FIN processing is handled by a `switch` that depends on the connection state. Notice that the FIN is not processed in the CLOSED, LISTEN, or SYN_SENT states, since in these three states a SYN has not been received to synchronize the received sequence number, making it impossible to validate the sequence number of the FIN. A FIN is also ignored in the CLOSING, CLOSE_WAIT, and LAST_ACK states, because in these three states the FIN is a duplicate.

SYN_RCVD or ESTABLISHED states

1127–1134 From either the ESTABLISHED or SYN_RCVD states, the CLOSE_WAIT state is entered.

> The receipt of a FIN in the SYN_RCVD state is unusual, but legal. It is not shown in Figure 24.15. It means a socket is in the LISTEN state when a segment containing a SYN and a FIN is received. Alternatively, a SYN is received for a listening socket, moving the connection to the SYN_RCVD state but before the ACK is received a FIN is received. (We know the segment does not contain a valid ACK, because if it did the code in Figure 29.2 would have moved the connection to the ESTABLISHED state.)

The next part of FIN processing is shown in Figure 29.25

tcp_input.c

```
1135            /*
1136             * If still in FIN_WAIT_1 state FIN has not been acked so
1137             * enter the CLOSING state.
1138             */
1139        case TCPS_FIN_WAIT_1:
1140            tp->t_state = TCPS_CLOSING;
1141            break;

1142            /*
1143             * In FIN_WAIT_2 state enter the TIME_WAIT state,
1144             * starting the time-wait timer, turning off the other
1145             * standard timers.
1146             */
1147        case TCPS_FIN_WAIT_2:
1148            tp->t_state = TCPS_TIME_WAIT;
1149            tcp_canceltimers(tp);
1150            tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
1151            soisdisconnected(so);
1152            break;

1153            /*
1154             * In TIME_WAIT state restart the 2 MSL time_wait timer.
1155             */
1156        case TCPS_TIME_WAIT:
1157            tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
1158            break;
1159        }
1160    }
```

tcp_input.c

Figure 29.25 `tcp_input` function: FIN processing, second half.

FIN_WAIT_1 state

1135–1141 Since ACK processing is already complete for this segment, if the connection is in the FIN_WAIT_1 state when the FIN is processed, it means a simultaneous close is taking place—the two FINs from each end have passed in the network. The connection enters the CLOSING state.

FIN_WAIT_2 state

1142–1148 The receipt of the FIN moves the connection into the TIME_WAIT state. When a segment containing a FIN and an ACK is received in the FIN_WAIT_1 state (the typical scenario), although Figure 24.15 shows the transition directly from the FIN_WAIT_1 state to the TIME_WAIT state, the ACK is processed in Figure 29.11, moving the connection to the FIN_WAIT_2 state. The FIN processing here moves the connection into the TIME_WAIT state. Because the ACK is processed before the FIN, the FIN_WAIT_2 state is always passed through, albeit momentarily.

Start TIME_WAIT timer

1149–1152 Any pending TCP timer is turned off and the TIME_WAIT timer is started with a value of twice the MSL. (If the received segment contained a FIN and an ACK, Figure 29.11 started the FIN_WAIT_2 timer.) The socket is disconnected.

TIME_WAIT state

1153–1159 If a FIN arrives in the TIME_WAIT state, it is a duplicate, and similar to Figure 29.14, the TIME_WAIT timer is restarted with a value of twice the MSL.

29.11 Final Processing

The final part of the slow path through `tcp_input` along with the label `dropafterack` is shown in Figure 29.26.

tcp_input.c

```
1161     if (so->so_options & SO_DEBUG)
1162         tcp_trace(TA_INPUT, ostate, tp, &tcp_saveti, 0);

1163     /*
1164      * Return any desired output.
1165      */
1166     if (needoutput || (tp->t_flags & TF_ACKNOW))
1167         (void) tcp_output(tp);
1168     return;

1169   dropafterack:
1170     /*
1171      * Generate an ACK dropping incoming segment if it occupies
1172      * sequence space, where the ACK reflects our state.
1173      */
1174     if (tiflags & TH_RST)
1175         goto drop;
1176     m_freem(m);
1177     tp->t_flags |= TF_ACKNOW;
1178     (void) tcp_output(tp);
1179     return;
```

tcp_input.c

Figure 29.26 `tcp_input` function: final processing.

`SO_DEBUG` socket option

1161–1162 If the `SO_DEBUG` socket option is enabled, `tcp_trace` appends the trace record to the kernel's circular buffer. Remember that the code in Figure 28.7 saved both the original connection state and the IP and TCP headers, since these values may have changed in this function.

Call `tcp_output`

1163–1168 If either the `needoutput` flag was set (Figures 29.6 and 29.15) or if an immediate ACK is required, `tcp_output` is called.

`dropafterack`

1169–1179 An ACK is generated only if the RST flag was not set. (A segment with an RST is never ACKed.) The mbuf chain containing the received segment is released, and `tcp_output` generates an immediate ACK.

Figure 29.27 completes the `tcp_input` function.

tcp_input.c

```
1180 dropwithreset:
1181     /*
1182      * Generate an RST, dropping incoming segment.
1183      * Make ACK acceptable to originator of segment.
1184      * Don't bother to respond if destination was broadcast/multicast.
1185      */
1186     if ((tiflags & TH_RST) || m->m_flags & (M_BCAST | M_MCAST) ||
1187         IN_MULTICAST(ti->ti_dst.s_addr))
1188         goto drop;
1189     if (tiflags & TH_ACK)
1190         tcp_respond(tp, ti, m, (tcp_seq) 0, ti->ti_ack, TH_RST);
1191     else {
1192         if (tiflags & TH_SYN)
1193             ti->ti_len++;
1194         tcp_respond(tp, ti, m, ti->ti_seq + ti->ti_len, (tcp_seq) 0,
1195                     TH_RST | TH_ACK);
1196     }
1197     /* destroy temporarily created socket */
1198     if (dropsocket)
1199         (void) soabort(so);
1200     return;

1201 drop:
1202     /*
1203      * Drop space held by incoming segment and return.
1204      */
1205     if (tp && (tp->t_inpcb->inp_socket->so_options & SO_DEBUG))
1206         tcp_trace(TA_DROP, ostate, tp, &tcp_saveti, 0);
1207     m_freem(m);
1208     /* destroy temporarily created socket */
1209     if (dropsocket)
1210         (void) soabort(so);
1211     return;
1212 }
```

tcp_input.c

Figure 29.27 `tcp_input` function: final processing.

`dropwithreset`

1180–1188 An RST is generated unless the received segment also contained an RST, or the received segment was sent as a broadcast or multicast. An RST is never generated in response to an RST, since this could lead to RST storms (a continual exchange of RST segments between two end points).

> This code contains the same error that we noted in Figure 28.16: it does not check whether the destination address of the received segment was a broadcast address.
>
> Similarly, the destination address argument to `IN_MULTICAST` needs to be converted to host byte order.

Sequence number and acknowledgment number of RST segment

1189–1196 The values of the sequence number field, the acknowledgment field, and the ACK flag of the RST segment depend on whether the received segment contained an ACK.

Figure 29.28 summarizes these fields in the RST segment that is generated.

	RST segment generated		
received segment	seq#	ack. field	flags
contains ACK	received ack. field	0	`TH_RST`
ACK-less	0	received seq# field	`TH_RST | TH_ACK`

Figure 29.28 Values of fields in RST segment generated.

Realize that the ACK flag is normally set in all segments except when an initial SYN is sent (Figure 24.16). The fourth argument to `tcp_respond` is the acknowledgment field, and the fifth argument is the sequence number.

Rejecting connections

1192–1193 If the SYN flag is set, `ti_len` must be incremented by 1, causing the acknowledgment field of the RST to be 1 greater than the received sequence number of the SYN. This code is executed when a SYN arrives for a nonexistent server. When the Internet PCB is not found in Figure 28.6, a jump is made to `dropwithreset`. But for the received RST to be acceptable to the other end, the acknowledgment field must ACK the SYN (Figure 28.18). Figure 18.14 of Volume 1 contains an example of this type of RST segment.

Finally note that `tcp_respond` builds the RST in the first mbuf of the received chain and releases any remaining mbufs in the chain. When that mbuf finally makes its way to the device driver, it will be discarded.

Destroy temporarily created socket

1197–1199 If a temporary socket was created in Figure 28.7 for a listening server, but the code in Figure 28.16 found the received segment to contain an error, `dropsocket` will be 1. If so, that socket is now destroyed.

Drop (without ACK or RST)

1201–1206 `tcp_trace` is called when a segment is dropped without generating an ACK or an RST. If the `SO_DEBUG` flag is set and an ACK is generated, `tcp_output` generates a trace record. If the `SO_DEBUG` flag is set and an RST is generated, a trace record is not generated for the RST.

1207–1211 The mbuf chain containing the received segment is released and the temporary socket is destroyed if `dropsocket` is nonzero.

29.12 Implementation Refinements

The refinements to speed up TCP processing are similar to the ones described for UDP (Section 23.12). Multiple passes over the data should be avoided and the checksum computation should be combined with a copy. [Dalton et al. 1993] describe these modifications.

The linear search of the TCP PCBs is also a bottleneck when the number of connections increases. [McKenney and Dove 1992] address this problem by replacing the linear search with hash tables.

[Partridge 1993] describes a research implementation being developed by Van Jacobson that greatly reduces the TCP input processing. The received packet is processed by IP (about 25 instructions on a RISC system), then by a demultiplexer to locate the PCB (about 10 instructions), and then by TCP (about 30 instructions). These 30 instructions perform header prediction and calculate the pseudo-header checksum. If the segment passes the header prediction test, contains data, and the process is waiting for the data, the data is copied into the process buffer and the remainder of the TCP checksum is calculated and verified (a one-pass copy and checksum). If the TCP header prediction fails, the slow path through the TCP input processing occurs.

29.13 Header Compression

We now describe TCP *header compression*. Although header compression is not part of TCP input, we needed to cover TCP thoroughly before describing header compression. Header compression is described in detail in RFC 1144 [Jacobson 1990a]. It was designed by Van Jacobson and is sometimes called *VJ header compression*. Our purpose in this section is not to go through the header compression source code (a well-commented version of which is presented in RFC 1144, and which is approximately the same size as `tcp_output`), but to provide an overview of the algorithm. Be sure to distinguish between header prediction (Section 28.4) and header compression.

Introduction

Most implementations of SLIP and PPP support header compression. Although header compression could, in theory, be used with any data link, it is intended for slow-speed serial links. Header compression works with TCP segments only—it does nothing with other IP datagrams (e.g., ICMP, IGMP, UDP, etc.). Header compression reduces the size of the combined IP/TCP header from its normal 40 bytes to as few as 3 bytes. This reduces the size of a typical TCP segment from an interactive application such as Rlogin or Telnet from 41 bytes to 4 bytes—a big saving on a slow-speed serial link.

Each end of the serial link maintains two connection state tables, one for datagrams sent and one for datagrams received. Each table allows a maximum of 256 entries, but typically there are 16 entries in this table, allowing up to 16 different TCP connections to be compressed at any time. Each entry contains an 8-bit connection ID (hence the limit of 256), some flags, and the complete uncompressed IP/TCP header from the most recent datagram. The 96-bit socket pair that uniquely identifies each connection—the source and destination IP addresses and source and destination TCP ports—are contained in this uncompressed header. Figure 29.29 shows an example of these tables.

Since a TCP connection is full duplex, header compression can be applied in both directions. Each end must implement both compression and decompression. A connection appears in both tables, as shown in Figure 29.29. In this example, the entry with a connection ID of 1 in the top two tables has a source IP address of 128.1.2.3, source TCP port of 1500, destination IP address of 192.3.4.5, and a destination TCP port of 25. The entry with a connection ID of 2 in the bottom two tables is for the other direction of the same connection.

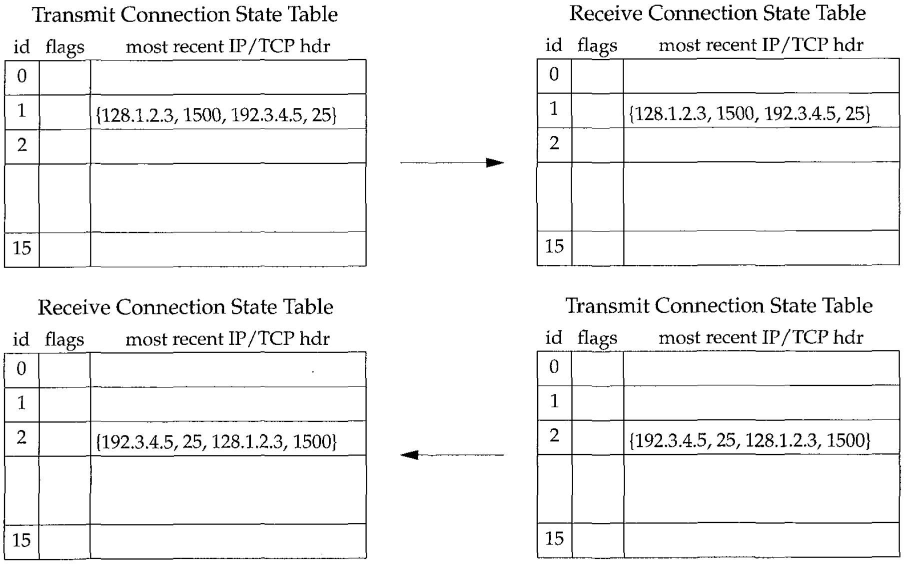


Figure 29.29 A pair of connection state tables at each end of a link (e.g., SLIP link).

We show these tables as arrays, but the source code defines each entry as a structure, and a connection table is a circular linked list of these structures. The most recently used structure is stored at the head of the list.

By saving the most recent uncompressed header at each end, only the *differences* in various header fields from the previous datagram to the current datagram are transmitted across the link (along with a special first byte indicating which fields follow). Since some header fields don't change at all from one datagram to the next, and other header fields change by small amounts, this differential coding provides the savings. Header compression works with the IP and TCP headers only—the data contents of the TCP segment are not modified.

Figure 29.30 shows the steps involved at the sending side when it has an IP datagram to send across a link using header compression.

Three different types of datagrams are sent and must be recognized at the receiver:

1. Type `IP` is specified with the high-order 4 bits of the first byte equal to 4. This is the normal IP version number in the IP header (Figure 8.8). The normal, uncompressed datagram is transmitted across the link.
2. Type `COMPRESSED_TCP` is specified by setting the high-order bit of the first byte. This looks like an IP version between 8 and 15 (i.e., the remaining 7 bits of this byte are used by the compression algorithm). The compressed header and uncompressed data are transmitted across the link, as we describe later in this section.

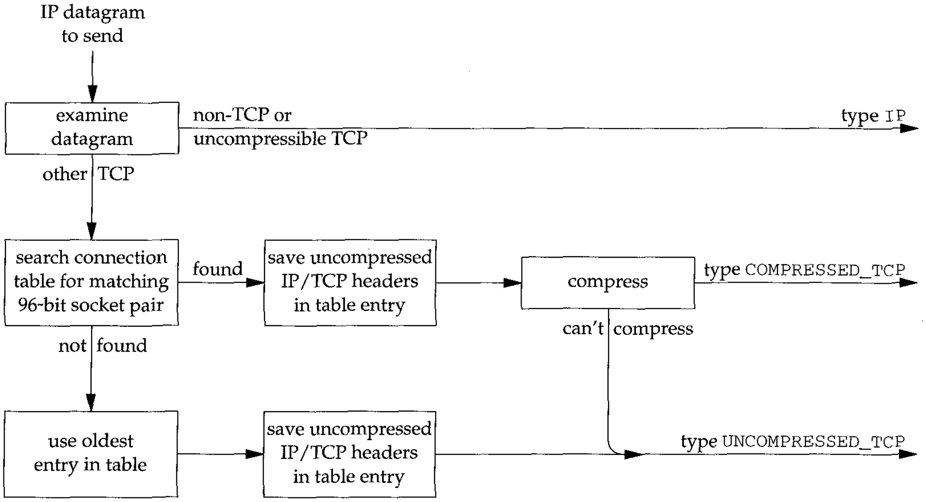


Figure 29.30 Steps involved in header compression at sender side.

3. Type `UNCOMPRESSED_TCP` is specified with the high-order 4 bits of the first byte equal to 7. The normal, uncompressed datagram is transmitted across the link, but the IP protocol field (which equals 6 for TCP), is replaced with the connection ID. This identifies the connection state table entry for the receiver.

The receiver can identify the datagram type by examining its first byte. The code that does this was shown in Figure 5.13. In Figure 5.16 the sender calls `sl_compress_tcp` to check if a TCP segment is compressible, and the return value of this function is logically ORed into the first byte of the datagram.

Figure 29.31 shows an illustration of the first byte that is sent across the link.

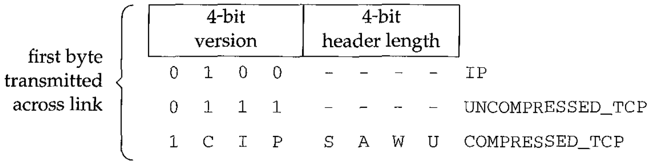


Figure 29.31 First byte transmitted across link.

The 4 bits shown as "-" comprise the normal IP header length field. The 7 bits shown as `C`, `I`, `P`, `S`, `A`, `W`, and `U` indicate which optional fields follow. We describe these fields shortly.

Figure 29.32 shows the complete IP datagram for the various datagrams that are sent.

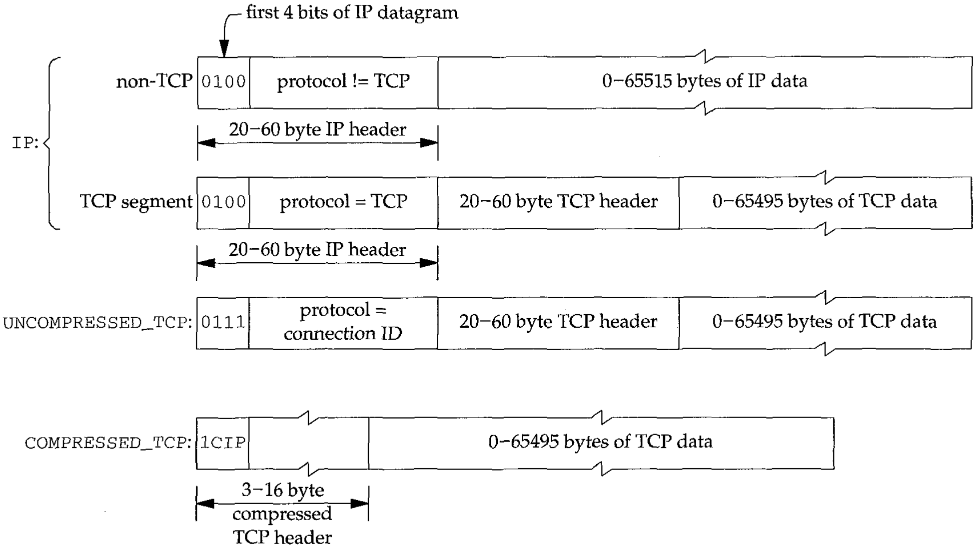


Figure 29.32 Different types of IP datagrams possible with header compression.

We show two datagrams with a type of `IP`: one that is not a TCP segment (e.g., a protocol of UDP, ICMP, or IGMP), and one that is a TCP segment. This is to illustrate the differences between the TCP segment sent as type `IP` and the TCP segment sent as type `UNCOMPRESSED_TCP`: the first 4 bits are different as is the protocol field in the IP header.

Datagrams are not candidates for header compression if the protocol is not TCP, or if the protocol is TCP but any one of the following conditions is true.

- The datagram is an IP fragment: either the fragment offset is nonzero or the more-fragments bit is set.
- Any one of the SYN, FIN, or RST flags is set.
- The ACK flag is not set.

If any one of these three conditions is true, the datagram is sent as type `IP`.

Furthermore, even if the datagram is a TCP segment that looks compressible, it is possible to abort the compression and send the datagram as type `UNCOMPRESSED_TCP` if certain fields have changed between the current datagram and the last datagram sent for this connection. These are fields that normally do not change for a given connection, so the compression scheme was not designed to encode their differences from one datagram to the next. The TOS field and the don't fragment bit are examples. Also, when the differences in some fields are greater than 65535, the compression algorithm fails and the datagram is sent uncompressed.

Compression of Header Fields

We now describe how the fields in the IP and TCP headers, shown in Figure 29.33, are compressed. The shaded fields normally don't change during a connection.

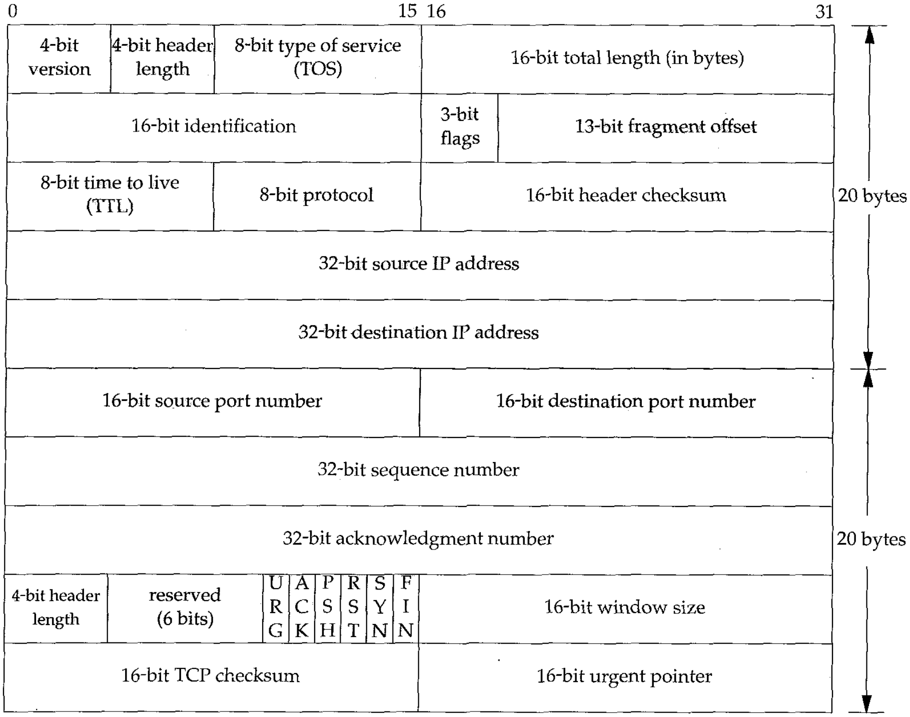


Figure 29.33 Combined IP and TCP headers: shaded fields normally don't change.

If any of the shaded fields have changed from the previous segment on this connection to the current segment, the segment is sent uncompressed. We don't show IP options or TCP options in this figure, but if either are present and have changed from the previous segment, the segment is sent uncompressed (Exercise 29.7).

If the algorithm transmitted only the nonshaded fields when the shaded fields do not change from the previous segment, about a 50% savings would result. VJ header compression does even better than this, by knowing which fields in the IP and TCP headers *normally* don't change. Figure 29.34 shows the format of the compressed IP/TCP header.

The smallest compressed header consists of 3 bytes: the first byte (the flag bits) followed by the 16-bit TCP checksum. For protection against possible link errors, the TCP checksum is always transmitted without any change. (SLIP provides no link-layer checksum, although PPP does provide one.)

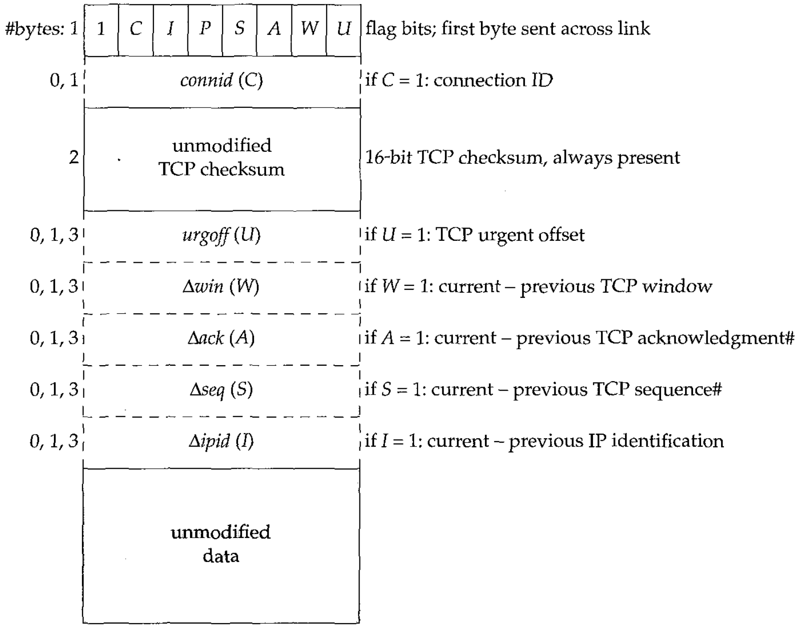


Figure 29.34 Format of compressed IP/TCP header.

The other six fields, *connid*, *urgoff*, Δ*win*, Δ*ack*, Δ*seq*, and Δ*ipid*, are optional. We show the number of bytes used to encode all the fields to the left of the field in Figure 29.34. The largest compressed header appears to be 19 bytes, but we'll see shortly that the 4 bits *SAWU* can never be set at the same time in a compressed header, so the largest size is actually 16 bytes.

Six of the 7 bits in the first byte specify which of the six optional fields are present. The high-order bit of the first byte is always set to 1. This identifies the datagram type as `COMPRESSED_TCP`. Figure 29.35 summarizes the 7 bits, which we now describe.

Flag bit	Description	Structure member	Meaning if flag = 0	Meaning if flag = 1
C	connection ID		same connection ID as last	*connid* = connection ID
I	IP identification	`ip_id`	`ip_id` has increased by 1	Δ*ipid* = current – previous
P	TCP push flag		PSH flag off	PSH flag on
S	TCP sequence#	`th_seq`	same `th_seq` as last	Δ*seq* = current – previous
A	TCP acknowledgment#	`th_ack`	same `th_ack` as last	Δ*ack* = current – previous
W	TCP window	`th_win`	same `th_win` as last	Δ*win* = current – previous
U	TCP urgent offset	`th_urg`	URG flag not set	*urgoff* = urgent offset

Figure 29.35 The 7 bits in the compressed header.

C	If this bit is 0, this segment has the same connection ID as the previous compressed or uncompressed segment. If this flag is 1, *connid* is the connection ID, a value between 0 and 255.
I	If this bit is 0, the IP identification field has increased by 1 (the typical case). If this bit is 1, Δ*ipid* is the current value of `ip_id` minus its previous value.
P	This bit is a copy of the PSH flag from the TCP segment. Since the PSH flag doesn't follow any established pattern, it must be explicitly specified for each segment.
S	If this bit is 0, the TCP sequence number has not changed. If this bit is 1, Δ*seq* is the current value of `th_seq` minus its previous value.
A	If this bit is 0, the TCP acknowledgment number has not changed (the typical case). If this bit is 1, Δ*ack* is the current value of `th_ack` minus its previous value.
W	If this bit is 0, the TCP window has not changed (the typical case). If this bit is 1, Δ*win* is the current value of `th_win` minus its previous value.
U	If this bit is 0, the URG flag in the segment is not set and the urgent offset has not changed from its previous value (the typical case). If this bit is 1, *urgoff* is the current value of `th_urg` and the URG flag is set. If the urgent offset changes without the URG flag being set, the segment is sent uncompressed. (This often occurs in the first segment following urgent data.)

The differences are encoded as the current value minus the previous value, because most of these differences will be small positive numbers (with Δ*win* being an exception) given the way these fields normally change.

We note that five of the optional fields in Figure 29.34 are encoded in 0, 1, or 3 bytes.

0 bytes:	If the corresponding flag is not set, nothing is encoded for the field.
1 byte:	If the value to send is between 1 and 255, a single byte encodes the value.
3 bytes:	If the value to send is either 0 or between 256 and 65535, 3 bytes encode the value: the first byte is 0, followed by the 2-byte value. This always works for the three 16-bit values, *urgoff*, Δ*win*, and Δ*ipid*; but if the difference to encode for the two 32-bit values, Δ*ack* and Δ*seq*, is less than 0 or greater than 65535, the segment is sent uncompressed.

If we compare the nonshaded fields in Figure 29.33 with the possible fields in Figure 29.34 we notice that some fields are never transmitted.

- The IP total length field is not transmitted since most link layers provide the length of a received message to the receiver.
- Since the only field in the IP header that is being transmitted is the identification field, the IP checksum is also omitted. This is a hop-by-hop checksum that protects only the IP header across any given link.

Special Cases

Two common cases are detected and transmitted as special combinations of the 4 low-order bits: *SAWU*. Since urgent data is rare, if the URG flag in the segment is set and both the sequence number and window also change (implying that the 4 low-order bits would be 1011 or 1111), the segment is sent uncompressed. Therefore if the 4 low-order bits are sent as 1011 (called **SA*) or 1111 (called **S*), the following two special cases apply:

**SA* The sequence number and acknowledgment number both increase by the amount of data in the last segment, the window and urgent offset don't change, and the URG flag is not set. This special case avoids encoding both Δseq and Δack.

This case occurs frequently for both directions of echoed terminal traffic. Figures 19.3 and 19.4 of Volume 1 give examples of this type of data flow across an Rlogin connection.

**S* The sequence number changes by the amount of data in the last segment, the acknowledgment number, window, and urgent offset don't change, and the URG flag is not set. This special case avoids encoding Δseq.

This case occurs frequently for the sending side of a unidirectional data transfer (e.g., FTP). Figures 20.1, 20.2, and 20.3 of Volume 1 give examples of this type of data transfer. This case also occurs for the sender of nonechoed terminal traffic (e.g., commands that are not echoed by a full-screen editor).

Examples

Two simple examples were run across the SLIP link between the systems `bsdi` and `slip` in Figure 1.17. This SLIP link uses header compression in both directions. The `tcpdump` program described in Appendix A of Volume 1 was also run on the host `bsdi` to save a copy of all the frames. This program has an option that outputs the compressed header, showing all the fields in Figure 29.34.

Two traces were obtained: a short portion of an Rlogin connection and a file transfer from `bsdi` to `slip` using FTP. Figure 29.36 shows a summary of the different frame types for both connections.

The two entries of 75 verify our claim that this special case often occurs for both directions of echoed terminal traffic. The entry of 325 verifies our claim that this special case occurs frequently for the sending side of a unidirectional data transfer.

The 10 frames of type `IP` for the FTP example correspond to four segments with the SYN flag set and six segments with the FIN flag set. FTP uses two connections: one for the interactive commands and one for the file transfer.

The `UNCOMPRESSED_TCP` frame types normally correspond to the first segment following connection establishment, the one that establishes the connection ID. An additional few are seen in these examples when the type of service is set (the Net/3 Rlogin and FTP clients and servers all set the TOS field *after* the connection is established).

frame type	Rlogin		FTP	
	input	output	input	output
IP	1	1	5	5
UNCOMPRESSED_TCP	3	2	2	3
COMPRESSED_TCP				
*SA special case	75	75	0	0
*S special case	25	1	1	325
nonspecial	9	93	337	13
Total	113	172	345	346

Figure 29.36 Counts of different frame types for Rlogin and FTP connections.

#bytes	Rlogin		FTP	
	input	output	input	output
3	102	44	2	250
4		94		78
5	7	12	5	2
6		6	325	5
7		13	2	1
8				1
9			4	1
Total	109	169	338	338

Figure 29.37 Distribution of compressed-header sizes.

Figure 29.37 shows the distribution of the compressed-header sizes. The average size of the compressed header for the final four columns in Figure 29.37 is 3.1, 4.1, 6.0, and 3.3 bytes, a significant savings compared to the uncompressed 40-byte headers, especially for the interactive connection.

Almost all of the 325 6-byte headers in the FTP input column contain only a Δ*ack* of 256, which being greater than 255 is encoded in 3 bytes. The SLIP MTU is 296, so TCP uses an MSS of 256. Almost all of the 250 3-byte headers in the FTP output column contain the *S special case (sequence number change only) with a change of 256 bytes. But since this change refers to the amount of data in the previous segment, nothing is transmitted other than the flag byte and the TCP checksum. The 78 4-byte headers in the FTP output column are this same special case, but with a change in the IP identification field also (Exercise 29.8).

Configuration

Header compression must be enabled on a given SLIP or PPP link. With a SLIP link there are normally two flags that can be set when the interface is configured: enable header compression and autoenable header compression. These two flags are set using the `link0` and `link2` flags to the `ifconfig` command, respectively. Normally a client (the dialin host) decides whether to use header compression or not. The server (the host or terminal server to which the client dials in) specifies the autoenable flag only. If header compression is enabled by the client, its TCP will send a datagram of type `UNCOMPRESSED_TCP` to specify the connection ID. When the server sees this packet it enables header compression (since it was in the autoenable mode). If the server never sees this type of packet, it never enables header compression for this line.

PPP allows the negotiation of options between the two ends of the link when the link is established. One of the options that can be negotiated is whether to use header compression or not.

29.14 Summary

This chapter completes our detailed look at TCP input processing. We started with the processing of an ACK in the SYN_RCVD state, which completes a passive open, a simultaneous open, or a self-connect.

The fast retransmit algorithm lets TCP detect a dropped segment after receiving a specified number of consecutive duplicate ACKs and retransmit the segment before the retransmission timer expires. Net/3 combines the fast retransmit algorithm with the fast recovery algorithm, which tries to keep the data flowing from the sender to the receiver, albeit at a slower rate, using congestion avoidance but not slow start.

ACK processing then discards the acknowledged data from the socket's send buffer and handles a few TCP states specially, when the receipt of an ACK changes the connection state.

The URG flag is processed, if set, and TCP's urgent mode is mapped into the socket abstraction of out-of-band data. This is complicated because the process can receive the out-of-band byte inline or in a special out-of-band buffer, and TCP can receive urgent notification before the data byte referenced by the urgent pointer has been received.

TCP input processing completes by calling `TCP_REASS` to merge the received data into either the socket's receive buffer or the socket's out-of-order queue, processing the FIN flag, and calling `tcp_output` if a segment must be generated in response to the received segment.

TCP header compression is a technique used on SLIP and PPP links to reduce the size of the IP and TCP headers from the normal 40 bytes to around 3–6 bytes (typically). This is done by recognizing that most fields in these headers don't change from one segment to the next on a given connection, and the fields that do change often change by a small amount. This allows a flag byte to be sent indicating which fields have changed, and the changes are encoded as differences from the previous segment.

Exercises

29.1 A client connects to a server and no segments are lost. Which process, the client or server, completes its open of the connection first?

29.2 A Net/3 system receives a SYN for a listening socket and the SYN segment also contains 50 bytes of data. What happens?

29.3 Continue the previous exercise assuming that the client does not retransmit the 50 bytes of data; instead the client responds with a segment that acknowledges the server's SYN/ACK and contains a FIN. What happens?

29.4 A Net/3 client performs a passive open to a listening server. The server's response to the client's SYN is a segment with the expected SYN/ACK, but the segment also contains 50 bytes of data and the FIN flag. List the processing steps for the client's TCP.

29.5 Figure 18.19 in Volume 1 and Figure 14 in RFC 793 both show four segments exchanged during a simultaneous close. But if we trace a simultaneous close between two Net/3 systems, or if we watch the close sequence following a self-connect on a Net/3 system, we see six segments, not four. The extra two segments are a retransmission of the FIN by each end when the other's FIN is received. Where is the bug and what is the fix?

29.6 Page 72 of RFC 793 says that when data in the send buffer is acknowledged by the other end "Users should receive positive acknowledgments for buffers which have been sent and fully acknowledged (i.e., send buffer should be returned with 'ok' response)." Does Net/3 provide this notification?

29.7 What effect do the options defined in RFC 1323 have on TCP header compression?

29.8 What effect does the Net/3 assignment of the IP identification field have on TCP header compression?

30

TCP User Requests

30.1 Introduction

This chapter looks at the TCP user-request function `tcp_usrreq`, which is called as the protocol's `pr_usrreq` function to handle many of the system calls that reference a TCP socket. We also look at `tcp_ctloutput`, which is called when the process calls `setsockopt` for a TCP socket.

30.2 `tcp_usrreq` Function

TCP's user-request function is called for a variety of operations. Figure 30.1 shows the beginning and end of `tcp_usrreq`. The body of the `switch` is shown in following figures. The function arguments, some of which differ depending on the request, are described in Figure 15.17.

`in_control` processes `ioctl` requests

45–58 The `PRU_CONTROL` request is from the `ioctl` system call. The function `in_control` processes the request completely.

Control information is invalid

59–64 A call to `sendmsg` specifying control information is invalid for a TCP socket. If this happens, the mbufs are released and `EINVAL` is returned.

65–66 This remainder of the function executes at `splnet`. This is overly conservative locking to avoid sprinkling the individual `case` statements with calls to `splnet` when the calls are really necessary. As we mentioned with Figure 23.15, setting the processor priority to `splnet` only stops a software interrupt from causing the IP input routine to

tcp_usrreq.c

```
int
tcp_usrreq(so, req, m, nam, control)
struct socket *so;
int     req;
struct mbuf *m, *nam, *control;
{
    struct inpcb *inp;
    struct tcpcb *tp;
    int     s;
    int     error = 0;
    int     ostate;

    if (req == PRU_CONTROL)
        return (in_control(so, (int) m, (caddr_t) nam,
                           (struct ifnet *) control));
    if (control && control->m_len) {
        m_freem(control);
        if (m)
            m_freem(m);
        return (EINVAL);
    }
    s = splnet();
    inp = sotoinpcb(so);
    /*
     * When a TCP is attached to a socket, then there will be
     * a (struct inpcb) pointed at by the socket, and this
     * structure will point at a subsidary (struct tcpcb).
     */
    if (inp == 0 && req != PRU_ATTACH) {
        splx(s);
        return (EINVAL);        /* XXX */
    }
    if (inp) {
        tp = intotcpcb(inp);
        /* WHAT IF TP IS 0? */
        ostate = tp->t_state;
    } else
        ostate = 0;

    switch (req) {


                              /* switch cases */


    default:
        panic("tcp_usrreq");
    }
    if (tp && (so->so_options & SO_DEBUG))
        tcp_trace(TA_USER, ostate, tp, (struct tcpiphdr *) 0, req);
    splx(s);
    return (error);
}
```

tcp_usrreq.c

Figure 30.1 Body of `tcp_usrreq` function.

be executed (which could call `tcp_input`). It does not prevent the interface layer from accepting incoming packets and placing them onto IP's input queue.

The pointer to the Internet PCB is obtained from the `socket` structure pointer. The only time the resulting PCB pointer is allowed to be a null pointer is when the `PRU_ATTACH` request is issued, which occurs in response to the `socket` system call.

67–81 If `inp` is nonnull, the current connection state is saved in `ostate` for the call to `tcp_trace` at the end of the function.

We now discuss the individual `case` statements. The `PRU_ATTACH` request, shown in Figure 30.2, is issued by the `socket` system call and by `sonewconn` when a connection request arrives for a listening socket (Figure 28.7).

tcp_usrreq.c

```
 83         /*
 84          * TCP attaches to socket via PRU_ATTACH, reserving space,
 85          * and an internet control block.
 86          */
 87     case PRU_ATTACH:
 88         if (inp) {
 89             error = EISCONN;
 90             break;
 91         }
 92         error = tcp_attach(so);
 93         if (error)
 94             break;
 95         if ((so->so_options & SO_LINGER) && so->so_linger == 0)
 96             so->so_linger = TCP_LINGERTIME;
 97         tp = sototcpcb(so);
 98         break;

 99         /*
100          * PRU_DETACH detaches the TCP protocol from the socket.
101          * If the protocol state is non-embryonic, then can't
102          * do this directly: have to initiate a PRU_DISCONNECT,
103          * which may finish later; embryonic TCB's can just
104          * be discarded here.
105          */
106     case PRU_DETACH:
107         if (tp->t_state > TCPS_LISTEN)
108             tp = tcp_disconnect(tp);
109         else
110             tp = tcp_close(tp);
111         break;
```

tcp_usrreq.c

Figure 30.2 `tcp_usrreq` function: `PRU_ATTACH` and `PRU_DETACH` requests.

PRU_ATTACH request

83–94 If the socket structure already points to a PCB, `EISCONN` is returned. `tcp_attach` completes the processing: it allocates and initializes the Internet PCB and the TCP control block.

95–96 If the `SO_LINGER` socket option is set, and the linger time is 0, it is set to 120 (`TCP_LINGERTIME`).

How can a socket option be set before the `PRU_ATTACH` request is issued? It is impossible to set a socket option before calling `socket`, but `sonewconn` also issues the `PRU_ATTACH` request. The `PRU_ATTACH` request is issued after `sonewconn` copies the `so_options` from the listening socket to the newly created socket. This code prevents a newly accepted connection from inheriting a linger time of 0 from the listening socket.

There is a bug here. The constant `TCP_LINGERTIME` is initialized to 120 in the header `tcp_timer.h` with the comment "linger at most 2 minutes." But the `so_linger` value becomes the final argument to the kernel's `tsleep` function (called from `soclose`), which becomes the final argument to the kernel's `timeout` function and is in clock ticks, not seconds. If the system's clock-tick frequency (`hz`) is 100, this value for the linger time is 1.2 seconds, not 2 minutes.

97 `tp` is now set to the pointer to the socket's TCP control block. This is required at the end, in case the `SO_DEBUG` socket option is set.

`PRU_DETACH` request

99–111 The `close` system call issues the `PRU_DETACH` request if the `PRU_DISCONNECT` request fails. If the connection has not been completed (the connection state is less than ESTABLISHED), nothing needs to be sent to the other end. But if the connection has been established, `tcp_disconnect` initiates TCP's connection-close sequence (e.g., any pending data is sent, followed by a FIN).

The test for the state being greater than LISTEN is incorrect, because if the state is SYN_SENT or SYN_RCVD, both of which are greater than LISTEN, `tcp_disconnect` just calls `tcp_close`. This `case` could be simplified by just calling `tcp_disconnect`.

Figure 30.3 shows the processing for the `bind` and `listen` system calls.

tcp_usrreq.c

```
112        /*
113         * Give the socket an address.
114         */
115    case PRU_BIND:
116        error = in_pcbbind(inp, nam);
117        if (error)
118            break;
119        break;

120        /*
121         * Prepare to accept connections.
122         */
123    case PRU_LISTEN:
124        if (inp->inp_lport == 0)
125            error = in_pcbbind(inp, (struct mbuf *) 0);
126        if (error == 0)
127            tp->t_state = TCPS_LISTEN;
128        break;
```

tcp_usrreq.c

Figure 30.3 `tcp_usrreq` function: `PRU_BIND` and `PRU_LISTEN` requests.

112–119 All the work for a `PRU_BIND` request is done by `in_pcbbind`.

120–128 For the `PRU_LISTEN` request, if the socket has not been bound with a local port, `in_pcbbind` assigns one automatically. This is rare, since most servers explicitly bind their well-known port, although RPC (remote procedure call) servers typically bind an ephemeral port and then register the port with the *Port Mapper*. (Section 29.4 of Volume 1 describes the Port Mapper.) The connection state is set to LISTEN. This is the main purpose of `listen`: to set the socket's state so that incoming connections are accepted (i.e., a passive open).

Figure 30.4 shows the processing for the `connect` system call: an active open normally initiated by a client.

tcp_usrreq.c

```
        /*
         * Initiate connection to peer.
         * Create a template for use in transmissions on this connection.
         * Enter SYN_SENT state, and mark socket as connecting.
         * Start keepalive timer, and seed output sequence space.
         * Send initial segment on connection.
         */
    case PRU_CONNECT:
        if (inp->inp_lport == 0) {
            error = in_pcbbind(inp, (struct mbuf *) 0);
            if (error)
                break;
        }
        error = in_pcbconnect(inp, nam);
        if (error)
            break;

        tp->t_template = tcp_template(tp);
        if (tp->t_template == 0) {
            in_pcbdisconnect(inp);
            error = ENOBUFS;
            break;
        }
        /* Compute window scaling to request.  */
        while (tp->request_r_scale < TCP_MAX_WINSHIFT &&
               (TCP_MAXWIN << tp->request_r_scale) < so->so_rcv.sb_hiwat)
            tp->request_r_scale++;
        soisconnecting(so);
        tcpstat.tcps_connattempt++;
        tp->t_state = TCPS_SYN_SENT;
        tp->t_timer[TCPT_KEEP] = TCPTV_KEEP_INIT;

        tp->iss = tcp_iss;
        tcp_iss += TCP_ISSINCR / 2;
        tcp_sendseqinit(tp);

        error = tcp_output(tp);
        break;
```

tcp_usrreq.c

Figure 30.4 `tcp_usrreq` function: `PRU_CONNECT` request.

Assign ephemeral port

129–141 If the socket has not been bound with a local port, `in_pcbbind` assigns one automatically. This is typical for clients, which normally don't care about the value of the local port.

Connect PCB

142–144 `in_pcbconnect` acquires a route to the destination, determines the outgoing interface, and verifies that the socket pair is unique.

Initialize IP and TCP headers

145–150 `tcp_template` allocates an mbuf for a copy of the IP and TCP headers, and it initializes both headers with as much information as possible. The only way for this function to fail is for the kernel to run out of mbufs.

Calculate window scale factor

151–154 The window scale value for the receive buffer is calculated: 65535 (`TCP_MAXWIN`) is left shifted until the value is greater than or equal to the size of the receive buffer (`so_rcv.sb_hiwat`). The resulting shift count (between 0 and 14) is the scale factor that will be sent in the SYN. (We saw identical code in Figure 28.7 that was executed for a passive open.) Since the window scale option is sent in the SYN resulting from a `connect`, the process must set the `SO_RCVBUF` socket option before calling `connect`, or the default buffer size is used (`tcp_recvspace` from Figure 24.3).

Set socket and connection state

155–158 `soisconnecting` sets the appropriate bits in the socket's state variable, and the state of the TCP connection is set to SYN_SENT. This causes the call to `tcp_output` that follows to send the SYN (see the `tcp_outflags` value in Figure 24.16). The connection-establishment timer is initialized to 75 seconds. `tcp_output` will also set the retransmission timer for the SYN, as shown in Figure 25.15.

Initialize sequence numbers

159–161 The initial send sequence number is copied from the global `tcp_iss`. This global is then incremented by 64,000 (`TCP_ISSINCR` divided by 2). We saw this same handling of `tcp_iss` when the ISS was initialized after a listening server received a SYN (Figure 28.17). The send sequence numbers are then initialized by `tcp_sendseqinit`.

Send initial SYN

162 `tcp_output` sends the initial SYN to initiate the connection. A local error (for example, out of mbufs or no route to destination) is returned by `tcp_output`, which becomes the return value from `tcp_usrreq`, which is returned to the process.

Figure 30.5 shows the processing for the `PRU_CONNECT2`, `PRU_DISCONNECT`, and `PRU_ACCEPT` requests.

164–169 The `PRU_CONNECT2` request, a result of the `socketpair` system call, is invalid for the TCP protocol.

170–183 The `close` system call issues the `PRU_DISCONNECT` request. If the connection has been established, a FIN must be sent and the normal TCP close sequence followed. This is done by `tcp_disconnect`.

tcp_usrreq.c

```
164         /*
165          * Create a TCP connection between two sockets.
166          */
167     case PRU_CONNECT2:
168         error = EOPNOTSUPP;
169         break;

170         /*
171          * Initiate disconnect from peer.
172          * If connection never passed embryonic stage, just drop;
173          * else if don't need to let data drain, then can just drop anyway,
174          * else have to begin TCP shutdown process: mark socket disconnecting,
175          * drain unread data, state switch to reflect user close, and
176          * send segment (e.g. FIN) to peer.  Socket will be really disconnected
177          * when peer sends FIN and acks ours.
178          *
179          * SHOULD IMPLEMENT LATER PRU_CONNECT VIA REALLOC TCPCB.
180          */
181     case PRU_DISCONNECT:
182         tp = tcp_disconnect(tp);
183         break;

184         /*
185          * Accept a connection.  Essentially all the work is
186          * done at higher levels; just return the address
187          * of the peer, storing through addr.
188          */
189     case PRU_ACCEPT:
190         in_setpeeraddr(inp, nam);
191         break;
```

tcp_usrreq.c

Figure 30.5 `tcp_usrreq` function: `PRU_CONNECT2`, `PRU_DISCONNECT`, and `PRU_ACCEPT` requests.

> The comment beginning with "SHOULD IMPLEMENT" refers to the fact that a socket that encounters an error cannot be reused. For example, if a client issues a `connect` and receives an error, it cannot issue another `connect` on the same socket. Instead, the socket with the error must be closed, a new socket created with `socket`, and the `connect` issued on the new socket.

184–191 All the work associated with the `accept` system call is done by the socket layer and the protocol layer. The `PRU_ACCEPT` request just returns the IP address and port number of the peer to the process.

The `PRU_SHUTDOWN`, `PRU_RCVD`, and `PRU_SEND` requests are processed in Figure 30.6.

`PRU_SHUTDOWN` request

192–200 This request is issued by `soshutdown` when the process calls `shutdown` to prevent any further output. `socantsendmore` sets the socket's flags to prevent any future output. `tcp_usrclosed` sets the connection state according to Figure 24.15. `tcp_output` attempts to send the FIN, but if there is still pending data to send to the other end, that data is sent before the FIN is sent.

tcp_usrreq.c

```
192         /*
193          * Mark the connection as being incapable of further output.
194          */
195     case PRU_SHUTDOWN:
196         socantsendmore(so);
197         tp = tcp_usrclosed(tp);
198         if (tp)
199             error = tcp_output(tp);
200         break;

201         /*
202          * After a receive, possibly send window update to peer.
203          */
204     case PRU_RCVD:
205         (void) tcp_output(tp);
206         break;

207         /*
208          * Do a send by putting data in output queue and updating urgent
209          * marker if URG set.  Possibly send more data.
210          */
211     case PRU_SEND:
212         sbappend(&so->so_snd, m);
213         error = tcp_output(tp);
214         break;
```

tcp_usrreq.c

Figure 30.6 `tcp_usrreq` function: `PRU_SHUTDOWN`, `PRU_RCVD`, and `PRU_SEND` requests.

`PRU_RCVD` request

201–206 This request is issued by `soreceive` after the process has read data from the socket's receive buffer. TCP needs to know about this since the receive buffer may now have enough room to allow the advertised window to increase. `tcp_output` will determine whether a window update segment should be sent.

`PRU_SEND` request

207–214 In Figure 23.14 we showed how the five write functions ended up issuing this request. `sbappend` adds the data to the socket's send buffer (where it must wait until acknowledged by the other end), and `tcp_output` sends a segment, if possible.

Figure 30.7 shows the processing of the `PRU_ABORT` and `PRU_SENSE` requests.

`PRU_ABORT` request

215–220 A `PRU_ABORT` request is issued for a TCP socket by `soclose` if the socket is a listening socket (e.g., a server) and if there are pending connections for the server that have already initiated or completed the three-way handshake, but have not been `accept`ed by the server yet. `tcp_drop` sends an RST if the connection is synchronized.

tcp_usrreq.c

```
215         /*
216          * Abort the TCP.
217          */
218     case PRU_ABORT:
219         tp = tcp_drop(tp, ECONNABORTED);
220         break;

221     case PRU_SENSE:
222         ((struct stat *) m)->st_blksize = so->so_snd.sb_hiwat;
223         (void) splx(s);
224         return (0);
```

tcp_usrreq.c

Figure 30.7 `tcp_usrreq` function: `PRU_ABORT` and `PRU_SENSE` requests.

`PRU_SENSE` request

221–224 The `fstat` system call generates the `PRU_SENSE` request. TCP returns the size of the send buffer as the `st_blksize` element of the `stat` structure.

Figure 30.8 shows the `PRU_RCVOOB` request, issued by `soreceive` when the process issues a read system call specifying the `MSG_OOB` flag to read out-of-band data.

tcp_usrreq.c

```
225     case PRU_RCVOOB:
226         if ((so->so_oobmark == 0 &&
227              (so->so_state & SS_RCVATMARK) == 0) ||
228             so->so_options & SO_OOBINLINE ||
229             tp->t_oobflags & TCPOOB_HADDATA) {
230             error = EINVAL;
231             break;
232         }
233         if ((tp->t_oobflags & TCPOOB_HAVEDATA) == 0) {
234             error = EWOULDBLOCK;
235             break;
236         }
237         m->m_len = 1;
238         *mtod(m, caddr_t) = tp->t_iobc;
239         if (((int) nam & MSG_PEEK) == 0)
240             tp->t_oobflags ^= (TCPOOB_HAVEDATA | TCPOOB_HADDATA);
241         break;
```

tcp_usrreq.c

Figure 30.8 `tcp_usrreq` function: `PRU_RCVOOB` request.

Verify that reading out-of-band data is appropriate

225–232 It is an error for the process to try to read out-of-band data if any one of the following three conditions is true:

1. if the socket's out-of-band mark is 0 (`so_oobmark`) and the socket is not at the mark (the `SS_RCVATMARK` flag is not set), or

2. if the `SO_OOBINLINE` socket option is set, or
3. if the `TCPOOB_HADDATA` flag is set for the connection (i.e., the connection did have an out-of-band byte, but it has already been read).

The error `EINVAL` is returned if any one of these is true.

Check that out-of-band byte has arrived

233–236 If none of the three conditions above is true, but the `TCPOOB_HAVEDATA` flag is false, this indicates that TCP has received an urgent mode notification from the other end, but the byte whose sequence number is 1 less than the urgent pointer has not been received yet (Figure 29.17). The error `EWOULDBLOCK` is returned. It is possible for TCP to send an urgent notification with an urgent offset referencing a byte that the sender has not been able to send yet. Figure 26.7 of Volume 1 shows an example of this scenario, which often happens if the sender's data transmission has been stopped by a zero-window advertisement.

Return out-of-band byte

237–238 The single byte of out-of-band data that was stored in `t_iobc` by `tcp_pulloutofband` is returned to the process.

Flip flags

239–241 If the process is actually reading the out-of-band byte (as compared to peeking at it with the `MSG_PEEK` flag), this exclusive OR turns the `HAVE` flag off and the `HAD` flag on. We are guaranteed at this point in the `case` statement that the `HAVE` flag is set and the `HAD` flag is cleared. The purpose of the `HAD` flag is to prevent the process from trying to read the out-of-band byte more than once. Once the `HAD` flag is set, it is not cleared until a new urgent pointer is received from the other end (Figure 29.17).

> The reason for this hard-to-understand exclusive OR, instead of the simpler
>
> ```
> tp->t_oobflags = TCPOOB_HADDATA;
> ```
>
> is to allow additional bits in `t_oobflags` to be used. Net/3, however, only uses the 2 bits that we've described.

The `PRU_SENDOOB` request, shown in Figure 30.9, is issued by `sosend` when the process writes data and specifies the `MSG_OOB` flag.

Check for room and append to send buffer

242–247 The process is allowed to exceed the size of the send buffer by up to 512 bytes when sending out-of-band data. The socket layer is more permissive, allowing out-of-band data to exceed the size of the send buffer by 1024 bytes (Figure 16.24). `sbappend` adds the data to the end of the send buffer.

Calculate urgent pointer

248–257 The urgent pointer (`snd_up`) points to the byte following the final byte from the write request. We showed this in Figure 26.30, assuming the process writes 3 bytes of data with the `MSG_OOB` flag set and that the send buffer was empty. Realize that if the

tcp_usrreq.c

```
242         case PRU_SENDOOB:
243             if (sbspace(&so->so_snd) < -512) {
244                 m_freem(m);
245                 error = ENOBUFS;
246                 break;
247             }
248             /*
249              * According to RFC961 (Assigned Protocols),
250              * the urgent pointer points to the last octet
251              * of urgent data.  We continue, however,
252              * to consider it to indicate the first octet
253              * of data past the urgent section.
254              * Otherwise, snd_up should be one lower.
255              */
256             sbappend(&so->so_snd, m);
257             tp->snd_up = tp->snd_una + so->so_snd.sb_cc;

258             tp->t_force = 1;
259             error = tcp_output(tp);
260             tp->t_force = 0;

261             break;
```

tcp_usrreq.c

Figure 30.9 `tcp_usrreq` function: `PRU_SENDOOB` request.

process writes more than 1 byte of data with the `MSG_OOB` flag set, only the final byte is considered the out-of-band byte when the data is received by a Berkeley-derived system.

Force TCP output

258–261 `t_force` is set to 1 and `tcp_output` is called. This causes a segment to be sent with the URG flag set and with a nonzero urgent offset, even if no data can be sent because of a zero-window advertisement. Figure 26.7 of Volume 1 shows the transmission of an urgent segment into a closed window.

The final three requests are shown in Figure 30.10.

262–267 The `getsockname` and `getpeername` system calls issue the `PRU_SOCKADDR` and `PRU_PEERADDR` requests, respectively. The functions `in_setsockaddr` and `in_setpeeraddr` fetch the information from the PCB, storing the result in the `addr` argument.

268–275 The `PRU_SLOWTIMO` request is issued by the `tcp_slowtimo` function. As the comment indicates, the only reason `tcp_slowtimo` doesn't call `tcp_timers` directly is to allow the timer expiration to be traced by the call to `tcp_trace` at the end of the function (Figure 30.1). For the trace record to show which one of the four TCP timer counters expired, `tcp_slowtimo` passes the index into the `t_timer` array (Figure 25.1) as the `nam` argument, and this is left shifted 8 bits and logically ORed into the request value (`req`). The `trpt` program knows about this hack and handles it accordingly.

tcp_usrreq.c

```
    case PRU_SOCKADDR:
        in_setsockaddr(inp, nam);
        break;

    case PRU_PEERADDR:
        in_setpeeraddr(inp, nam);
        break;

        /*
         * TCP slow timer went off; going through this
         * routine for tracing's sake.
         */
    case PRU_SLOWTIMO:
        tp = tcp_timers(tp, (int) nam);
        req |= (int) nam << 8;  /* for debug's sake */
        break;
```

tcp_usrreq.c

Figure 30.10 `tcp_usrreq` function: `PRU_SOCKADDR`, `PRU_PEERADDR`, and `PRU_SLOWTIMO` requests.

30.3 `tcp_attach` Function

The `tcp_attach` function is called by `tcp_usrreq` to process the `PRU_ATTACH` request (i.e., when the `socket` system call is issued or when a new connection request arrives for a listening socket). Figure 30.11 shows the code.

Allocate space for send buffer and receive buffer

361–372 If space has not been allocated for the socket's send and receive buffers, `sbreserve` sets them both to 8192, the default values of the global variables `tcp_sendspace` and `tcp_recvspace` (Figure 24.3).

Whether these defaults are adequate depends on the MSS for each direction of the connection, which depends on the MTU. For example, [Comer and Lin 1994] show that anomalous behavior occurs if the send buffer is less than three times the MSS, which drastically reduces performance. Some implementations have much higher defaults, such as 61,444 bytes, realizing the effect these defaults have on performance, especially with higher MTUs (e.g., FDDI and ATM).

Allocate Internet PCB and TCP control block

373–377 `in_pcballoc` allocates an Internet PCB and `tcp_newtcpcb` allocates a TCP control block and links it to the PCB.

378–384 The code with the comment `XXX` is executed if the call to `malloc` in `tcp_newtcpcb` fails. Remember that the `PRU_ATTACH` request is issued as a result of the `socket` system call, and when a connection request arrives for a listening socket (`sonewconn`). In the latter case the socket flag `SS_NOFDREF` is set. If this flag is left on, the call to `sofree` by `in_pcbdetach` releases the `socket` structure. As we saw in `tcp_input`, this structure should not be released until that function is done with the received segment (the `dropsocket` flag in Figure 29.27). Therefore the current value of the `SS_NOFDREF` flag is saved in the variable `nofd` when `in_pcbdetach` is called, and reset before `tcp_attach` returns.

385–386 The TCP connection state is initialized to CLOSED.

tcp_usrreq.c

```
int
tcp_attach(so)
struct socket *so;
{
    struct tcpcb *tp;
    struct inpcb *inp;
    int     error;

    if (so->so_snd.sb_hiwat == 0 || so->so_rcv.sb_hiwat == 0) {
        error = soreserve(so, tcp_sendspace, tcp_recvspace);
        if (error)
            return (error);
    }
    error = in_pcballoc(so, &tcb);
    if (error)
        return (error);
    inp = sotoinpcb(so);
    tp = tcp_newtcpcb(inp);
    if (tp == 0) {
        int     nofd = so->so_state & SS_NOFDREF;   /* XXX */

        so->so_state &= ~SS_NOFDREF;    /* don't free the socket yet */
        in_pcbdetach(inp);
        so->so_state |= nofd;
        return (ENOBUFS);
    }
    tp->t_state = TCPS_CLOSED;
    return (0);
}
```

tcp_usrreq.c

Figure 30.11 `tcp_attach` function: create a new TCP socket.

30.4 `tcp_disconnect` Function

`tcp_disconnect`, shown in Figure 30.12, initiates a TCP disconnect.

Connection not yet synchronized

396–402 If the socket is not yet in the ESTABLISHED state (i.e., LISTEN, SYN_SENT, or SYN_RCVD), `tcp_close` just releases the PCB and the TCP control block. Nothing needs to be sent to the other end since the connection has not been synchronized.

Hard disconnect

403–404 If the connection is synchronized, the `SO_LINGER` socket option is set, and the linger time (`so_linger`) is set to 0, the connection is dropped by `tcp_drop`. This sets the connection state to CLOSED, sends an RST to the other end, and releases the PCB and TCP control block. The connection does not pass through the TIME_WAIT state. The call to `close` that caused the `PRU_DISCONNECT` request will discard any data still in the send or receive buffers.

If the `SO_LINGER` socket option has been set with a nonzero linger time, it is handled by `soclose`.

tcp_usrreq.c

```
396 struct tcpcb *
397 tcp_disconnect(tp)
398 struct tcpcb *tp;
399 {
400     struct socket *so = tp->t_inpcb->inp_socket;

401     if (tp->t_state < TCPS_ESTABLISHED)
402         tp = tcp_close(tp);
403     else if ((so->so_options & SO_LINGER) && so->so_linger == 0)
404         tp = tcp_drop(tp, 0);
405     else {
406         soisdisconnecting(so);
407         sbflush(&so->so_rcv);
408         tp = tcp_usrclosed(tp);
409         if (tp)
410             (void) tcp_output(tp);
411     }
412     return (tp);
413 }
```

tcp_usrreq.c

Figure 30.12 `tcp_disconnect` function: initiate TCP disconnect.

Graceful disconnect

405–406 This code is executed when the connection has been synchronized but the `SO_LINGER` option either was not set or was set with a nonzero linger time. TCP's normal connection termination steps must be followed. `soisdisconnecting` sets the socket's state.

Discard pending receive data

407 Any pending data in the receive buffer is discarded by `sbflush`, since the process has closed the socket. The send buffer is left alone, however, and `tcp_output` will try to send what remains. We say "try" because there's no guarantee that the data still to be sent will be transmitted successfully. The other end might crash before it receives and acknowledges the data, or even if the TCP module at the other end receives and acknowledges the data, the system might crash before the application at the other end reads the data. Since the local process has closed the socket, if TCP gives up trying to send what remains in the send buffer (because its retransmission timer finally expires), there is no way to notify the process of the error.

Change connection state

408–410 `tcp_usrclosed` moves the connection into the next state, based on the current state. This normally moves the connection to the FIN_WAIT_1 state, since the connection is typically closed from the ESTABLISHED state. We'll see that `tcp_usrclosed` always returns the current control block pointer (`tp`), since the state must be synchronized to get to this point in the code, so `tcp_output` is always called to send a segment. If the connection moves from the ESTABLISHED to the FIN_WAIT_1 state, this causes a FIN to be sent.

30.5 `tcp_usrclosed` Function

This function, shown in Figure 30.13, is called from `tcp_disconnect` and when the `PRU_SHUTDOWN` request is processed.

tcp_usrreq.c

```
424 struct tcpcb *
425 tcp_usrclosed(tp)
426 struct tcpcb *tp;
427 {

428     switch (tp->t_state) {

429     case TCPS_CLOSED:
430     case TCPS_LISTEN:
431     case TCPS_SYN_SENT:
432         tp->t_state = TCPS_CLOSED;
433         tp = tcp_close(tp);
434         break;

435     case TCPS_SYN_RECEIVED:
436     case TCPS_ESTABLISHED:
437         tp->t_state = TCPS_FIN_WAIT_1;
438         break;

439     case TCPS_CLOSE_WAIT:
440         tp->t_state = TCPS_LAST_ACK;
441         break;
442     }
443     if (tp && tp->t_state >= TCPS_FIN_WAIT_2)
444         soisdisconnected(tp->t_inpcb->inp_socket);
445     return (tp);
446 }
```

tcp_usrreq.c

Figure 30.13 `tcp_usrclosed` function: move connection to next state, based on process close.

Simple close when SYN not received

429–434 If a SYN has not been received on the connection, a FIN need not be sent. The new state is CLOSED and `tcp_close` releases the Internet PCB and the TCP control block.

Move to FIN_WAIT_1 state

435–438 In the SYN_RCVD and ESTABLISHED states, the new state is FIN_WAIT_1, which causes the next call to `tcp_output` to send a FIN (the `tcp_outflags` value in Figure 24.16).

Move to LAST_ACK state

439–441 In the CLOSE_WAIT state, the close moves the connection into the LAST_ACK state. The next call to `tcp_output` will cause a FIN to be sent.

443–444 If the connection state is either FIN_WAIT_2 or TIME_WAIT, `soisdisconnected` marks the socket state appropriately.

30.6 `tcp_ctloutput` Function

The `tcp_ctloutput` function is called by the `getsockopt` and `setsockopt` system calls when the descriptor argument refers to a TCP socket and when the level is not `SOL_SOCKET`. Figure 30.14 shows the two socket options supported by TCP.

optname	Variable	Access	Description
`TCP_NODELAY`	`t_flags`	read, write	Nagle algorithm (Figure 26.8)
`TCP_MAXSEG`	`t_maxseg`	read, write	maximum segment size TCP will send

Figure 30.14 Socket options supported by TCP.

Figure 30.15 shows the first part of the function.

tcp_usrreq.c

```
284 int
285 tcp_ctloutput(op, so, level, optname, mp)
286 int     op;
287 struct socket *so;
288 int     level, optname;
289 struct mbuf **mp;
290 {
291     int     error = 0, s;
292     struct inpcb *inp;
293     struct tcpcb *tp;
294     struct mbuf *m;
295     int     i;

296     s = splnet();
297     inp = sotoinpcb(so);
298     if (inp == NULL) {
299         splx(s);
300         if (op == PRCO_SETOPT && *mp)
301             (void) m_free(*mp);
302         return (ECONNRESET);
303     }
304     if (level != IPPROTO_TCP) {
305         error = ip_ctloutput(op, so, level, optname, mp);
306         splx(s);
307         return (error);
308     }
309     tp = intotcpcb(inp);
```

tcp_usrreq.c

Figure 30.15 `tcp_ctloutput` function: first part.

296–303 The processor priority is set to `splnet` while the function executes, and `inp` points to the Internet PCB for the socket. If `inp` is null, the mbuf is released if the operation was to set a socket option, and an error is returned.

304–308 If the *level* (the second argument to the `getsockopt` and `setsockopt` system calls) is not `IPPROTO_TCP`, the command is for some other protocol (i.e., IP). For example, it is possible to create a TCP socket and set the IP source routing socket option. In this example IP processes the socket option, not TCP. `ip_ctloutput` handles the command.

309 The command is for TCP, so `tp` is set to the TCP control block.

The remainder of the function is a `switch` with two cases: one for `PRCO_SETOPT` (shown in Figure 30.16) and one for `PRCO_GETOPT` (shown in Figure 30.17).

tcp_usrreq.c

```
310     switch (op) {

311     case PRCO_SETOPT:
312         m = *mp;
313         switch (optname) {

314         case TCP_NODELAY:
315             if (m == NULL || m->m_len < sizeof(int))
316                     error = EINVAL;
317                 else if (*mtod(m, int *))
318                         tp->t_flags |= TF_NODELAY;
319                 else
320                     tp->t_flags &= ~TF_NODELAY;
321             break;

322         case TCP_MAXSEG:
323             if (m && (i = *mtod(m, int *)) > 0 && i <= tp->t_maxseg)
324                     tp->t_maxseg = i;
325             else
326                 error = EINVAL;
327             break;

328         default:
329             error = ENOPROTOOPT;
330             break;
331         }
332         if (m)
333             (void) m_free(m);
334         break;
```

tcp_usrreq.c

Figure 30.16 `tcp_ctloutput` function: set a socket option.

315–316 `m` is an mbuf containing the fourth argument to `setsockopt`. For both of the TCP options the mbuf must contain an integer value. If either the mbuf pointer is null, or the amount of data in the mbuf is less than the size of an integer, an error is returned.

`TCP_NODELAY` option

317–321 If the integer value is nonzero, the `TF_NODELAY` flag is set. This disables the Nagle algorithm in Figure 26.8. If the integer value is 0, the Nagle algorithm is enabled (the default) and the `TF_NODELAY` flag is cleared.

`TCP_MAXSEG` option

322–327 A process can only decrease the MSS. When a TCP socket is created, `tcp_newtcpcb` initializes `t_maxseg` to its default of 512. When a SYN is received from the other end with an MSS option, `tcp_input` calls `tcp_mss`, and `t_maxseg` can be set as high as the outgoing interface MTU (minus 40 bytes for the default IP and TCP headers), which is 1460 for an Ethernet. Therefore, after a call to `socket` but before a connection is established, a process can only decrease the MSS from its default of 512. After a connection is established, the process can decrease the MSS from whatever value was selected by `tcp_mss`.

> 4.4BSD was the first Berkeley release to allow the MSS to be set with a socket option. Prior releases only allowed a `getsockopt` for the MSS.

Release mbuf

332–333 The mbuf chain is released.

Figure 30.17 shows the processing for the `PRCO_GETOPT` command.

tcp_usrreq.c

```
335     case PRCO_GETOPT:
336         *mp = m = m_get(M_WAIT, MT_SOOPTS);
337         m->m_len = sizeof(int);

338         switch (optname) {
339         case TCP_NODELAY:
340             *mtod(m, int *) = tp->t_flags & TF_NODELAY;
341             break;
342         case TCP_MAXSEG:
343             *mtod(m, int *) = tp->t_maxseg;
344             break;
345         default:
346             error = ENOPROTOOPT;
347             break;
348         }
349         break;
350     }
351     splx(s);
352     return (error);
353 }
```

tcp_usrreq.c

Figure 30.17 `tcp_ctloutput` function: get a socket option.

335–337 Both TCP socket options return an integer to the process, so `m_get` obtains an mbuf and its length is set to the size of an integer.

339–341 `TCP_NODELAY` returns the current status of the `TF_NODELAY` flag: 0 if the flag is not set (the Nagle algorithm is enabled) or `TF_NODELAY` if the flag is set.

342–344 The `TCP_MAXSEG` option returns the current value of `t_maxseg`. As we said in our discussion of the `PRCO_SETOPT` command, the value returned depends whether the socket has been connected yet.

30.7 Summary

The `tcp_usrreq` function is straightforward because most of the required processing is done by other functions. The `PRU_`*xxx* requests form the glue between the protocol-independent system calls and the TCP protocol processing.

The `tcp_ctloutput` function is also simple because only two socket options are supported by TCP: enable or disable the Nagle algorithm, and set or fetch the maximum segment size.

Exercises

30.1 Now that we've covered all of TCP, list the processing steps and the TCP state transitions when a client goes through the normal steps of `socket`, `connect`, `write` (a request to the server), `read` (a reply from the server), and `close`. Do the same exercise for the server end.

30.2 If a process sets the `SO_LINGER` socket option with a linger time of 0 and then calls `close`, we showed how `tcp_disconnect` is called, which causes an RST to be sent. What happens if a process sets this socket option with a linger time of 0 but is then killed by a signal instead of calling `close`? Is the RST segment still sent?

30.3 The description for `TCP_LINGERTIME` in Figure 25.4 is the "maximum #seconds for `SO_LINGER` socket option." Given the code in Figure 30.2, is this description correct?

30.4 A Net/3 client calls `socket` and `connect` to actively open a connection to a server. The server is reached through the client's default router. A total of 1,129 segments are sent by the client host to the server. Assuming the route to the destination does not change, how many routing table lookups are done on the client host for this connection? Explain.

30.5 Obtain the `sock` program described in Appendix C of Volume 1. Run it as a sink server with a pause before reading (`-P`) and a large receive buffer. Then run the same program on another system as a source client. Watch the data with `tcpdump`. Verify that TCP's ACK-every-other-segment does not occur and that the only ACKs seen from the server are delayed ACKs.

30.6 Modify the `SO_KEEPALIVE` socket option so that the parameters can be configured on a per-connection basis.

30.7 Read RFC 1122 to determine why it recommends that an implementation should allow an RST to carry data. Modify the Net/3 code to implement this.

31

BPF: BSD Packet Filter

31.1 Introduction

The BSD Packet Filter (BPF) is a software device that "taps" network interfaces. A process accesses a BPF device by opening `/dev/bpf0`, `/dev/bpf1`, and so on. Each BPF device can be opened only by one process at a time.

> Since each BPF device allocates 8192 bytes of buffer space, the system administrator typically limits the number of BPF devices. If `open` returns `EBUSY`, the device is in use, and a process tries the next device until the `open` succeeds.

The device is configured with several `ioctl` commands that associate the device with a network interface and install filters to receive incoming packets selectively. Packets are received by reading from the device, and packets are queued on the network interface by writing to the device.

> We will use the term *packet* even though *frame* is more accurate, since BPF works at the data-link layer and includes the link-layer headers in the frames it sends and receives.

BPF works only with network interfaces that been modified to support BPF. In Chapter 3 we saw that the Ethernet, SLIP, and loopback drivers call `bpfattach`. This call configures the interface for access through the BPF devices. In this section we show how the BPF device driver is organized and how packets move between the driver and the network interfaces.

BPF is normally used as a diagnostic tool to examine the traffic on a locally attached network. The `tcpdump` program is the best example of such a tool and is described in Appendix A of Volume 1. Normally the user is interested in packets between a given set of machines, or for a particular protocol, or even for a particular TCP connection. A BPF device can be configured with a filter that discards or accepts incoming packets according to a filter specification. Filters are specified as instructions to a pseudo-machine. The details of BPF filters are not discussed in this text. For more information about filters, see `bpf`(4) and [McCanne and Jacobson 1993].

31.2 Code Introduction

The code for the portion of the BPF device driver that we describe resides in the two headers and one C file listed in Figure 31.1.

File	Description
`net/bpf.h`	BPF constants
`net/bpfdesc.h`	BPF structures
`net/bpf.c`	BPF device support

Figure 31.1 *Files discussed in this chapter.*

Global Variables

The global variables introduced in this chapter are shown in Figure 31.2.

Variable	Datatype	Description
`bpf_iflist`	`struct bpf_if *`	linked list of BPF-capable interfaces
`bpf_dtab`	`struct bpf_d []`	array of BPF descriptor structures
`bpf_bufsize`	`int`	default size of BPF buffers

Figure 31.2 *Global variables introduced in this chapter.*

Statistics

Figure 31.3 shows the two statistics collected in the `bpf_d` structure for every active BPF device.

`bpf_d` member	Description
`bd_rcount`	#packets received from network interface
`bd_dcount`	#packets dropped because of insufficient buffer space

Figure 31.3 Statistics collected in this chapter.

The remainder of this chapter is divided into four sections:

- BPF interface structures,
- BPF device descriptors,
- BPF input processing, and
- BPF output processing.

31.3 `bpf_if` Structure

BPF keeps a list of the network interfaces that support BPF. Each interface is described by a `bpf_if` structure, and the global pointer `bpf_iflist` points to the first structure in the list. Figure 31.4 shows a BPF interface structure.

bpfdesc.h

```
67 struct bpf_if {
68     struct bpf_if *bif_next;     /* list of all interfaces */
69     struct bpf_d *bif_dlist;     /* descriptor list */
70     struct bpf_if **bif_driverp;     /* pointer into softc */
71     u_int   bif_dlt;             /* link layer type */
72     u_int   bif_hdrlen;          /* length of header (with padding) */
73     struct ifnet *bif_ifp;       /* correspoding interface */
74 };
```

bpfdesc.h

Figure 31.4 `bpf_if` structure.

67–69 `bif_next` points to the next BPF interface structure in the list. `bif_dlist` points to a list of BPF devices that have been opened and configured to tap this interface.

70 `bif_driverp` points to a `bpf_if` pointer stored in the `ifnet` structure of the tapped interface. When the interface is *not* tapped, `*bif_driverp` is null. When a BPF device is configured to tap an interface, `*bif_driverp` is changed to point back to the `bif_if` structure and tells the interface to begin passing packets to BPF.

71 The type of interface is saved in `bif_dlt`. The values for our example interfaces are shown in Figure 31.5.

`bif_dlt`	Description
DLT_EN10MB	10Mb Ethernet interface
DLT_SLIP	SLIP interface
DLT_NULL	loopback interface

Figure 31.5 `bif_dlt` values.

72–74 Each packet accepted by BPF has a BPF header prepended to it. `bif_hdrlen` is the size of the header. Finally, `bif_ifp` points to the `ifnet` structure for the associated interface.

Figure 31.6 shows the `bpf_hdr` structure that is prepended to every incoming packet.

bpf.h

```
122 struct bpf_hdr {
123     struct timeval bh_tstamp;    /* time stamp */
124     u_long  bh_caplen;           /* length of captured portion */
125     u_long  bh_datalen;          /* original length of packet */
126     u_short bh_hdrlen;           /* length of bpf header (this struct plus
127                                     alignment padding) */
128 };
```

bpf.h

Figure 31.6 `bpf_hdr` structure.

122–128 `bh_tstamp` records the time the packet was captured. `bh_caplen` is the number of bytes saved by BPF, and `bh_datalen` is the number of bytes in the original packet. `bh_headlen` is the size of the `bpf_hdr` structure plus any padding. This value should match `bif_hdrlen` for the receiving interface and is used by processes to interpret the packets read from the BPF device.

Figure 31.7 shows how `bpf_if` structures are connected to the `ifnet` structures for each of our three sample interfaces (`le_softc[0]`, `sl_softc[0]`, and `loif`).

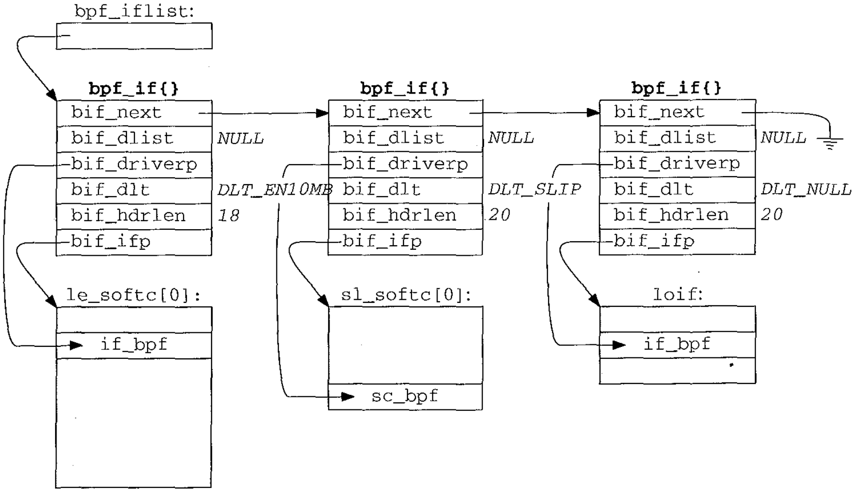


Figure 31.7 `bpf_if` and `ifnet` structures.

Notice that `bif_driverp` points to the `if_bpf` and `sc_bpf` pointers in the network interfaces and *not* to the interface structures.

> The SLIP device uses `sc_bpf`, instead of the `if_bpf` member. One reason might be that the SLIP BPF code was written before the `if_bpf` member was added to the `ifnet` structure. The `ifnet` structure in Net/2 does not include a `if_bpf` member.

The link-type and header-length members are initialized for all three interfaces according to the information passed by each driver in the call to `bpfattach`.

In Chapter 3 we saw that `bpfattach` was called by the Ethernet, SLIP, and loopback drivers. The linked list of BPF interface structures is built as each device driver calls `bpfattach` during initialization. The function is shown in Figure 31.8.

1053–1063 `bpfattach` is called by each device driver that supports BPF. The first argument is the pointer saved in `bif_driverp` (described with Figure 31.4). The second argument points to the `ifnet` structure of the interface. The third argument identifies the data-link type, and the fourth argument identifies the size of link-layer header passed with the packet. A new `bpf_if` structure is allocated for the interface.

bpf.c

```
1053 void
1054 bpfattach(driverp, ifp, dlt, hdrlen)
1055 caddr_t *driverp;
1056 struct ifnet *ifp;
1057 u_int   dlt, hdrlen;
1058 {
1059     struct bpf_if *bp;
1060     int     i;
1061     bp = (struct bpf_if *) malloc(sizeof(*bp), M_DEVBUF, M_DONTWAIT);
1062     if (bp == 0)
1063         panic("bpfattach");

1064     bp->bif_dlist = 0;
1065     bp->bif_driverp = (struct bpf_if **) driverp;
1066     bp->bif_ifp = ifp;
1067     bp->bif_dlt = dlt;

1068     bp->bif_next = bpf_iflist;
1069     bpf_iflist = bp;

1070     *bp->bif_driverp = 0;

1071     /*
1072      * Compute the length of the bpf header.  This is not necessarily
1073      * equal to SIZEOF_BPF_HDR because we want to insert spacing such
1074      * that the network layer header begins on a longword boundary (for
1075      * performance reasons and to alleviate alignment restrictions).
1076      */
1077     bp->bif_hdrlen = BPF_WORDALIGN(hdrlen + SIZEOF_BPF_HDR) - hdrlen;

1078     /*
1079      * Mark all the descriptors free if this hasn't been done.
1080      */
1081     if (!D_ISFREE(&bpf_dtab[0]))
1082         for (i = 0; i < NBPFILTER; ++i)
1083             D_MARKFREE(&bpf_dtab[i]);

1084     printf("bpf: %s%d attached\n", ifp->if_name, ifp->if_unit);
1085 }
```

bpf.c

Figure 31.8 `bpfattach` function.

Initialize `bpf_if` structure

1064–1070 The `bpf_if` structure is initialized from the arguments and inserted into the front of the BPF interface list, `bpf_iflist`.

Compute BPF header size

1071–1077 `bif_hdrlen` is set to force the *network-layer* header (e.g., the IP header) to start on a longword boundary. This improves performance and avoids unnecessary alignment restrictions for the BPF filter. Figure 31.9 shows the overall organization of the captured BPF packet for each of our three sample interfaces.

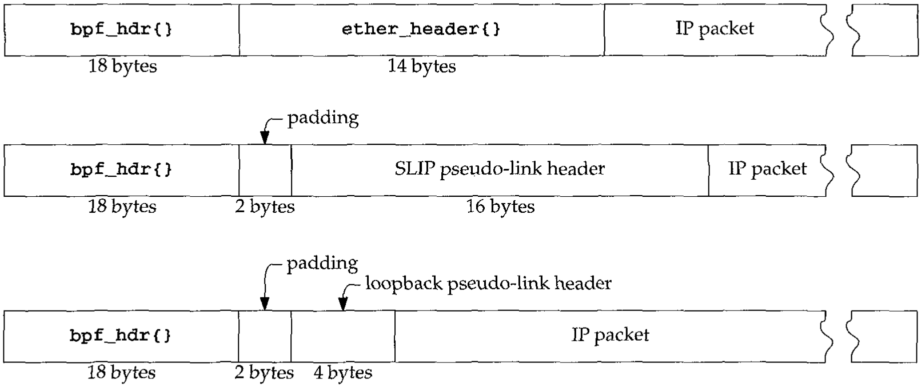


Figure 31.9 BPF packet organization.

The `ether_header` structure was described with Figure 4.10, the SLIP pseudo-link header was described with Figure 5.14, and the loopback pseudo-link header was described with Figure 5.28.

Notice that the SLIP and loopback packets require 2 bytes of padding to force the IP header to appear on a 4-byte boundary.

Initialize `bpf_dtab` table

1078–1083 This code initializes the BPF descriptor table, which is described with Figure 31.10. The initialization occurs the first time `bpfattach` is called and is skipped thereafter.

Print console message

1084–1085 A short message is printed to the console to announce that the interface has been configured for use by BPF.

31.4 `bpf_d` Structure

To begin tapping an interface, a process opens a BPF device and issues `ioctl` commands to select the interface, the read buffer size, and timeouts, and to specify a BPF filter. Each BPF device has an associated `bpf_d` structure, shown in Figure 31.10.

45–46 `bpf_d` structures are placed on a linked list when more than one BPF device is attached to the same network interface. `bd_next` points to the next structure in the list.

Packet buffers

47–52 Each `bpf_d` structure has two packet buffers associated with it. Incoming packets are always stored in the buffer attached to `bd_sbuf` (the store buffer). The other buffer is either attached to `bd_fbuf` (the free buffer), which means it is empty, or to `bd_hbuf` (the hold buffer), which means it contains packets that are being read by a process. `bd_slen` and `bd_hlen` record the number of bytes saved in the store and hold buffer respectively.

bpfdesc.h

```
45 struct bpf_d {
46     struct bpf_d *bd_next;      /* Linked list of descriptors */
47     caddr_t bd_sbuf;            /* store slot */
48     caddr_t bd_hbuf;            /* hold slot */
49     caddr_t bd_fbuf;            /* free slot */
50     int     bd_slen;            /* current length of store buffer */
51     int     bd_hlen;            /* current length of hold buffer */

52     int     bd_bufsize;         /* absolute length of buffers */

53     struct bpf_if *bd_bif;      /* interface descriptor */
54     u_long  bd_rtout;           /* Read timeout in 'ticks' */
55     struct bpf_insn *bd_filter; /* filter code */
56     u_long  bd_rcount;          /* number of packets received */
57     u_long  bd_dcount;          /* number of packets dropped */

58     u_char  bd_promisc;         /* true if listening promiscuously */
59     u_char  bd_state;           /* idle, waiting, or timed out */
60     u_char  bd_immediate;       /* true to return on packet arrival */
61     u_char  bd_pad;             /* explicit alignment */
62     struct selinfo bd_sel;      /* bsd select info */
63 };
```

bpfdesc.h

Figure 31.10 `bpf_d` structure.

When the store buffer becomes full, it is attached to `bd_hbuf` and the free buffer is attached to `bd_sbuf`. When the hold buffer is emptied, it is attached to `bd_fbuf`. The macro `ROTATE_BUFFERS` attaches the store buffer to `bd_hbuf`, attaches the free buffer to `bd_sbuf`, and clears `bd_fbuf`. It is called when the store buffer becomes full, or when the process doesn't want to wait for more packets.

`bd_bufsize` records the size of the two buffers associated with the device. It defaults to 4096 (`BPF_BUFSIZE`) bytes. The default value can be changed by patching the kernel, or `bd_bufsize` can be changed for a particular BPF device with the `BIOCSBLEN` `ioctl` command. The `BIOCGBLEN` command returns the current value of `bd_bufsize`, which can never exceed 32768 (`BPF_MAXBUFSIZE`) bytes. There is also a minimum size of 32 (`BPF_MINBUFSIZE`) bytes.

53–57 `bd_bif` points to the `bpf_if` structure associated with the BPF device. The `BIOCSETIF` command specifies the device. `bd_rtout` is the number of clock ticks to delay while waiting for packets to appear. `bd_filter` points to the BPF filter code for this device. Two statistics, which are available to a process through the `BIOCGSTATS` command, are kept in `bd_rcount` and `bd_dcount`.

58–63 `bd_promisc` is set with the `BIOCPROMISC` command and causes the interface to operate in promiscuous mode. `bd_state` is unused. `bd_immediate` is set with the `BIOCIMMEDIATE` command and causes the driver to return each packet as it is received instead of waiting for the hold buffer to fill. `bd_pad` pads the `bpf_d` structure to a longword boundary, and `bd_sel` holds the `selinfo` structure for the `select` system call. We don't describe the use of `select` with a BPF device, but `select` itself is described in Section 16.13.

`bpfopen` Function

When `open` is called for a BPF device, the call is routed to `bpfopen` (Figure 31.11) for processing.

bpf.c

```
256 int
257 bpfopen(dev, flag)
258 dev_t   dev;
259 int     flag;
260 {
261     struct bpf_d *d;

262     if (minor(dev) >= NBPFILTER)
263         return (ENXIO);
264     /*
265      * Each minor can be opened by only one process.  If the requested
266      * minor is in use, return EBUSY.
267      */
268     d = &bpf_dtab[minor(dev)];
269     if (!D_ISFREE(d))
270         return (EBUSY);

271     /* Mark "free" and do most initialization. */
272     bzero((char *) d, sizeof(*d));
273     d->bd_bufsize = bpf_bufsize;

274     return (0);
275 }
```

bpf.c

Figure 31.11 `bpfopen` function.

256–263 The number of BPF devices is limited at compile time to `NBPFILTER`. The minor device number specifies the device and `ENXIO` is returned if it is too large. This happens when the system administrator creates more `/dev/bpf`*x* entries than the value `NBPFILTER`.

Allocate `bpf_d` structure

264–275 Only one process is allowed access to a BPF device at a time. If the `bpf_d` structure is already active, `EBUSY` is returned. Programs such as `tcpdump` try the next device when this error is returned. If the device is available, the entry in the `bpf_dtab` table specified by the minor device number is cleared and the size of the packet buffers is set to the default value.

`bpfioctl` Function

Once the device is opened, it is configured with `ioctl` commands. Figure 31.12 summarizes the `ioctl` commands used with BPF devices. Figure 31.13 shows the `bpfioctl` function. Only the code for `BIOCSETF` and `BIOCSETIF` is shown. We have omitted the `ioctl` commands that are not discussed in this text.

Command	Third argument	Function	Description
FIONREAD	`u_int`	`bpfioctl`	return #bytes in hold buffer and store buffers.
BIOCGBLEN	`u_int`	`bpfioctl`	return size of packet buffers
BIOCSBLEN	`u_int`	`bpfioctl`	set size of packet buffers
BIOCSETF	`struct bpf_program`	`bpf_setf`	install BPF program
BIOCFLUSH		`reset_d`	discard pending packets
BIOCPROMISC		`ifpromisc`	enable promiscuous mode
BIOCGDLT	`u_int`	`bpfioctl`	return `bif_dlt`
BIOCGETIF	`struct ifreq`	`bpf_ifname`	return name of attached interface
BIOCSETIF	`struct ifreq`	`bpf_setif`	attach network interface to device
BIOCSRTIMEOUT	`struct timeval`	`bpfioctl`	set read timeout value
BIOCGRTIMEOUT	`struct timeval`	`bpfioctl`	return read timeout value
BIOCGSTATS	`struct bpf_stat`	`bpfioctl`	return BPF statistics
BIOCIMMEDIATE	`u_int`	`bpfioctl`	enable immediate mode
BIOCVERSION	`struct bpf_version`	`bpfioctl`	return BPF version information

Figure 31.12 BPF `ioctl` commands.

bpf.c

```
int
bpfioctl(dev, cmd, addr, flag)
dev_t   dev;
int     cmd;
caddr_t addr;
int     flag;
{
    struct bpf_d *d = &bpf_dtab[minor(dev)];
    int     s, error = 0;

    switch (cmd) {
        /*
         * Set link layer read filter.
         */
    case BIOCSETF:
        error = bpf_setf(d, (struct bpf_program *) addr);
        break;

        /*
         * Set interface.
         */
    case BIOCSETIF:
        error = bpf_setif(d, (struct ifreq *) addr);
        break;


                    /* other ioctl commands from Figure 31.12 */


    default:
        error = EINVAL;
        break;
    }
    return (error);
}
```

bpf.c

Figure 31.13 `bpfioctl` function.

501–509 As with `bpfopen`, the minor device number selects the `bpf_d` structure from the `bpf_dtab` table. The command is processed by the cases within the `switch`. We show two commands, `BIOCSETF` and `BIOCSETIF`, as well as the `default` case.

510–522 The `bpf_setf` function installs the filter passed in `addr`, and `bpf_setif` attaches the named interface to the `bpf_d` structure. We don't show the implementation of `bpf_setf` in this text.

668–673 If the command is not recognized, `EINVAL` is returned.

Figure 31.14 shows the `bpf_d` structure after `bpf_setif` has attached it to the LANCE interface in our example system.

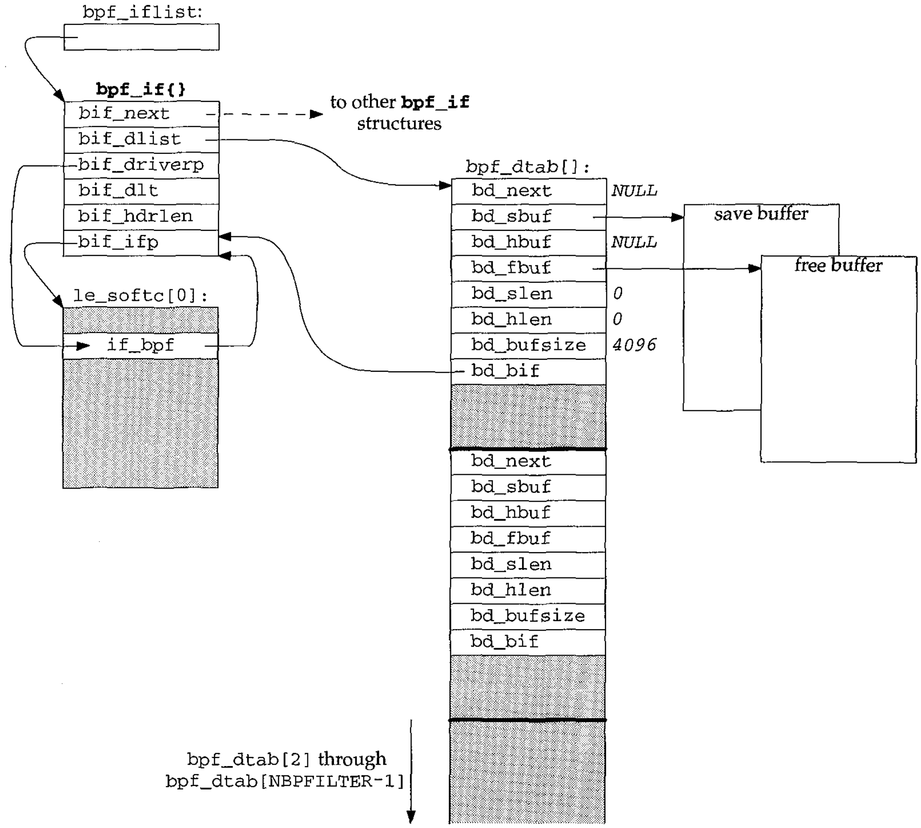


Figure 31.14 BPF device attached to the Ethernet interface.

In the figure, `bif_dlist` points to `bpf_dtab[0]`, the first and only descriptor in the descriptor list for the Ethernet interface. In `bpf_dtab[0]`, the `bd_sbuf` and `bd_hbuf` members point to the store and hold buffers. Each buffer is 4096 (`bd_bufsize`) bytes long. `bd_bif` points back to the `bpf_if` structure for the interface.

`if_bpf` in the `ifnet` structure (`le_softc[0]`) also points back to the `bpf_if` structure. As shown in Figures 4.19 and 4.11, when `if_bpf` is nonnull, the driver begins passing packets to the BPF device by calling `bpf_tap`.

Figure 31.15 shows the same structures after a second BPF device is opened and attached to the same Ethernet network interface as in Figure 31.10.

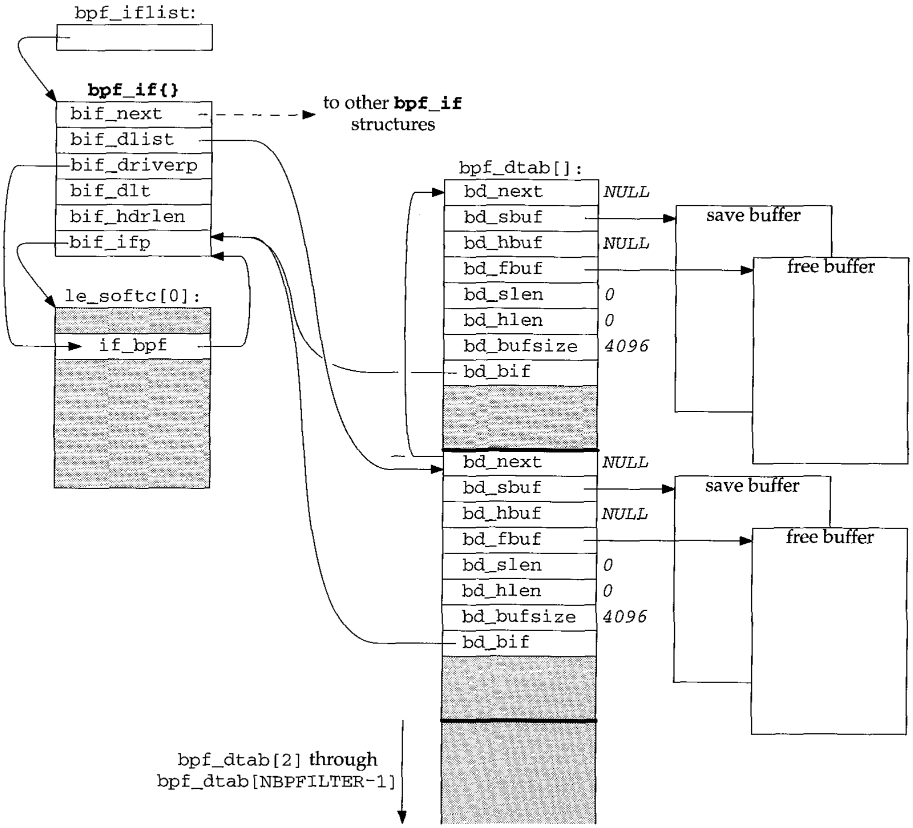


Figure 31.15 Two BPF devices attached to the Ethernet interface.

When the second BPF device is opened, a new `bpf_d` structure is allocated from the `bpf_dtab` table, in this case, `bpf_dtab[1]`. The second BPF device is also attached to the Ethernet interface, so `bif_dlist` points to `bpf_dtab[1]`, and `bpf_dtab[1].bd_next` points to `bpf_dtab[0]`, which is the first BPF descriptor attached to the Ethernet interface. Separate store and hold buffers are allocated and attached to the new descriptor structure.

`bpf_setif` Function

The `bpf_setif` function, which associates the BPF descriptor with a network interface, is shown in Figure 31.16.

bpf.c

```
static int
bpf_setif(d, ifr)
struct bpf_d *d;
struct ifreq *ifr;
{
    struct bpf_if *bp;
    char   *cp;
    int     unit, s, error;

    /*
     * Separate string into name part and unit number.  Put a null
     * byte at the end of the name part, and compute the number.
     * If the a unit number is unspecified, the default is 0,
     * as initialized above.  XXX This should be common code.
     */
    unit = 0;
    cp = ifr->ifr_name;
    cp[sizeof(ifr->ifr_name) - 1] = '\0';
    while (*cp++) {
        if (*cp >= '0' && *cp <= '9') {
            unit = *cp - '0';
            *cp++ = '\0';
            while (*cp)
                unit = 10 * unit + *cp++ - '0';
            break;
        }
    }
    /*
     * Look through attached interfaces for the named one.
     */
    for (bp = bpf_iflist; bp != 0; bp = bp->bif_next) {
        struct ifnet *ifp = bp->bif_ifp;

        if (ifp == 0 || unit != ifp->if_unit
            || strcmp(ifp->if_name, ifr->ifr_name) != 0)
            continue;
        /*
         * We found the requested interface.
         * If it's not up, return an error.
         * Allocate the packet buffers if we need to.
         * If we're already attached to requested interface,
         * just flush the buffer.
         */
        if ((ifp->if_flags & IFF_UP) == 0)
            return (ENETDOWN);
```

```
764         if (d->bd_sbuf == 0) {
765             error = bpf_allocbufs(d);
766             if (error != 0)
767                 return (error);
768         }
769         s = splimp();
770         if (bp != d->bd_bif) {
771             if (d->bd_bif)
772                 /*
773                  * Detach if attached to something else.
774                  */
775                 bpf_detachd(d);

776             bpf_attachd(d, bp);
777         }
778         reset_d(d);
779         splx(s);
780         return (0);
781     }
782     /* Not found. */
783     return (ENXIO);
784 }
```

bpf.c

Figure 31.16 `bpf_setif` function.

721–746 The first part of `bpf_setif` separates the text portion of the name in the `ifreq` structure (Figure 4.23) from the numeric portion. The numeric portion is saved in `unit`. For example, if the first 4 bytes of `ifr_name` start is "`sl1\0`", after this code executes they are "`sl\0\0`" and `unit` is 1.

Locate matching `ifnet` structure

747–754 The `for` loop searches the interfaces that support BPF (the ones in `bpf_iflist`) for the one specified in the `ifreq` structure.

755–768 If the matching interface is not up `ENETDOWN` is returned. If the interface is up, `bpf_allocate` attaches the free and store buffers to the `bpf_d` structure, if they have not already been allocated.

Attach `bpf_d` structure

769–777 If no interface is attached to the BPF device, or if a different interface from the one specified in the `ifreq` structure is attached, `bpf_detachd` discards the previous interface (if any), and `bpf_attachd` attaches the new interface to the device.

778–784 `reset_d` resets the packet buffers, discarding any pending packets in the process. The function returns 0 to indicate success or returns `ENXIO` if the interface was not located.

`bpf_attachd` Function

The `bpf_attachd` function shown in Figure 31.17 associates a BPF descriptor structure with a BPF device and with a network interface.

bpf.c

```
189 static void
190 bpf_attachd(d, bp)
191 struct bpf_d *d;
192 struct bpf_if *bp;
193 {
194     /*
195      * Point d at bp, and add d to the interface's list of listeners.
196      * Finally, point the driver's bpf cookie at the interface so
197      * it will divert packets to bpf.
198      */
199     d->bd_bif = bp;
200     d->bd_next = bp->bif_dlist;
201     bp->bif_dlist = d;

202     *bp->bif_driverp = bp;
203 }
```

bpf.c

Figure 31.17 `bpf_attachd` function.

189–203 First, `bd_bif` is set to point to the BPF interface structure for the network device. Next, the `bpf_d` structure is inserted into the front of the list of `bpf_d` structures associated with the device. Finally, the BPF pointer within the network interface is changed to point to the BPF structure, which causes the interface to begin passing packets to the BPF device.

31.5 BPF Input

Once the BPF device is opened and configured, a process uses the `read` system call to receive packets from the interface. The BPF tap collects *copies* of the incoming packets so BPF does not interfere with normal network processing. Incoming packets are collected in the store and hold buffers associated with each BPF device.

`bpf_tap` Function

We described the call to `bpf_tap` by the LANCE device driver with Figure 4.11 and use this call to describe the `bpf_tap`. The call (from Figure 4.11) is:

```
bpf_tap(le->sc_if.if_bpf, buf, len + sizeof(struct ether_header));
```

The `bpf_tap` function is shown in Figure 31.18.

869–882 The first argument is a pointer to the `bpf_if` structure, which is set by `bpfattach`. The second argument is a pointer to the incoming packet, including the Ethernet header. The third argument is the number of bytes contained in the buffer, in this case, the size of the Ethernet header (14 bytes) plus the size of the data portion of the Ethernet frame.

bpf.c

```
869 void
870 bpf_tap(arg, pkt, pktlen)
871 caddr_t arg;
872 u_char *pkt;
873 u_int   pktlen;
874 {
875     struct bpf_if *bp;
876     struct bpf_d *d;
877     u_int   slen;
878     /*
879      * Note that the ipl does not have to be raised at this point.
880      * The only problem that could arise here is that if two different
881      * interfaces shared any data.  This is not the case.
882      */
883     bp = (struct bpf_if *) arg;
884     for (d = bp->bif_dlist; d != 0; d = d->bd_next) {
885         ++d->bd_rcount;
886         slen = bpf_filter(d->bd_filter, pkt, pktlen, pktlen);
887         if (slen != 0)
888             catchpacket(d, pkt, pktlen, slen, bcopy);
889     }
890 }
```

bpf.c

Figure 31.18 `bpf_tap` function.

Pass packet to one or more BPF devices

883–890 The `for` loop traverses the list of BPF devices attached to the interface. For each device, the packet is passed to `bpf_filter`. If the filter accepts the packet, it returns the number of bytes to capture and `catchpacket` saves a copy of the packet. If the filter rejects the packet, `slen` is 0 and the loop continues. When the loop completes, `bpf_tap` returns. This mechanism enables each BPF device to have a separate filter when multiple BPF devices are associated with the same network interface.

The loopback driver calls `bpf_mtap` to pass packets to BPF. This function is similar to `bpf_tap` but copies the packet from an mbuf chain instead of from a contiguous area of memory. This function is not described in this text.

`catchpacket` Function

In Figure 31.18 we saw that `catchpacket` is called when the filter accepts the packet. The function is shown in Figure 31.19.

946–955 The arguments to `catchpacket` are: `d`, a pointer to the BPF device structure; `pkt` a generic pointer to the incoming packet; `pktlen` the length of the packet as it was received; `snaplen` the number of bytes to save from the packet; and `cpfn` a pointer to a function that will copy the packet from `pkt` to a contiguous area of memory. When the packet is already in a contiguous area of memory, `cpfn` is `bcopy`. When the packet is stored in an mbuf (i.e., `pkt` points to the first mbuf in a chain such as with the loopback driver), `cpfn` is `bpf_mcopy`.

bpf.c

```
static void
catchpacket(d, pkt, pktlen, snaplen, cpfn)
struct bpf_d *d;
u_char *pkt;
u_int   pktlen, snaplen;
void    (*cpfn) (const void *, void *, u_int);
{
    struct bpf_hdr *hp;
    int     totlen, curlen;
    int     hdrlen = d->bd_bif->bif_hdrlen;
    /*
     * Figure out how many bytes to move.  If the packet is
     * greater or equal to the snapshot length, transfer that
     * much.  Otherwise, transfer the whole packet (unless
     * we hit the buffer size limit).
     */
    totlen = hdrlen + min(snaplen, pktlen);
    if (totlen > d->bd_bufsize)
        totlen = d->bd_bufsize;

    /*
     * Round up the end of the previous packet to the next longword.
     */
    curlen = BPF_WORDALIGN(d->bd_slen);
    if (curlen + totlen > d->bd_bufsize) {
        /*
         * This packet will overflow the storage buffer.
         * Rotate the buffers if we can, then wakeup any
         * pending reads.
         */
        if (d->bd_fbuf == 0) {
            /*
             * We haven't completed the previous read yet,
             * so drop the packet.
             */
            ++d->bd_dcount;
            return;
        }
        ROTATE_BUFFERS(d);
        bpf_wakeup(d);
        curlen = 0;
    } else if (d->bd_immediate)
        /*
         * Immediate mode is set.  A packet arrived so any
         * reads should be woken up.
         */
        bpf_wakeup(d);
    /*
     * Append the bpf header.
     */
    hp = (struct bpf_hdr *) (d->bd_sbuf + curlen);
    microtime(&hp->bh_tstamp);
    hp->bh_datalen = pktlen;
    hp->bh_hdrlen = hdrlen;
```

```
 999     /*
1000      * Copy the packet data into the store buffer and update its length.
1001      */
1002     (*cpfn) (pkt, (u_char *) hp + hdrlen, (hp->bh_caplen = totlen - hdrlen));
1003     d->bd_slen = curlen + totlen;
1004 }
```

bpf.c

Figure 31.19 `catchpacket` function.

956–964 In addition to the link-layer header and the packet, `catchpacket` appends a `bpf_hdr` to every packet. The number of bytes to save from the packet is the smaller of `snaplen` and `pktlen`. The resulting packet and `bpf_hdr` must fit within the packet buffers (`bd_bufsize` bytes).

Will the packet fit?

965–985 `curlen` is the number of bytes already in the store buffer plus enough bytes to align the next packet on a longword boundary. If the incoming packet doesn't fit in the remaining buffer space, the store buffer is full. If a free buffer is not available (i.e., a process is still reading data from the hold buffer), the incoming packet is discarded. If a free buffer is available, it is rotated into place by `ROTATE_BUFFERS` and any process waiting for incoming data is awakened by `bpf_wakeup`.

Immediate mode processing

986–991 If the device is operating in immediate mode, any waiting processes are awakened to process the incoming packet—there is no buffering of packets in the kernel.

Append BPF header

992–1004 The current time (`microtime`), the packet length, and the header length are saved in a `bpf_hdr`. The function pointed to by `cpfn` is called to copy the packet into the store buffer and the length of the store buffer is updated. Since `bpf_tap` is called directly from `leread` even before the packet is transferred from a device buffer to an mbuf chain, the receive timestamp is close to the actual reception time.

`bpfread` Function

The kernel routes a `read` on a BPF device to `bpfread`. BPF supports a timed read through the `BIOCSRTIMEOUT` command. This "feature" is easily emulated by the more general `select` system call, but `tcpdump`, for example, uses `BIOCSRTIMEOUT` and not `select`. The process must provide a read buffer that matches the size of the hold buffer for the device. The `BIOCGBLEN` command returns the size of the buffer. Normally, a read returns when the store buffer becomes full. The kernel rotates the store buffer to the hold buffer, which is copied to the buffer provided with the `read` system call while the BPF device continues collecting incoming packets in the store buffer. `bpfread` is shown in Figure 31.20.

bpf.c

```
int
bpfread(dev, uio)
dev_t   dev;
struct uio *uio;
{
    struct bpf_d *d = &bpf_dtab[minor(dev)];
    int     error;
    int     s;

    /*
     * Restrict application to use a buffer the same size as
     * as kernel buffers.
     */
    if (uio->uio_resid != d->bd_bufsize)
        return (EINVAL);

    s = splimp();
    /*
     * If the hold buffer is empty, then do a timed sleep, which
     * ends when the timeout expires or when enough packets
     * have arrived to fill the store buffer.
     */
    while (d->bd_hbuf == 0) {
        if (d->bd_immediate && d->bd_slen != 0) {
            /*
             * A packet(s) either arrived since the previous
             * read or arrived while we were asleep.
             * Rotate the buffers and return what's here.
             */
            ROTATE_BUFFERS(d);
            break;
        }
        error = tsleep((caddr_t) d, PRINET | PCATCH, "bpf", d->bd_rtout);
        if (error == EINTR || error == ERESTART) {
            splx(s);
            return (error);
        }
        if (error == EWOULDBLOCK) {
            /*
             * On a timeout, return what's in the buffer,
             * which may be nothing.  If there is something
             * in the store buffer, we can rotate the buffers.
             */
            if (d->bd_hbuf)
                /*
                 * We filled up the buffer in between
                 * getting the timeout and arriving
                 * here, so we don't need to rotate.
                 */
                break;
```

```
392             if (d->bd_slen == 0) {
393                 splx(s);
394                 return (0);
395             }
396             ROTATE_BUFFERS(d);
397             break;
398         }
399     }
400     /*
401      * At this point, we know we have something in the hold slot.
402      */
403     splx(s);

404     /*
405      * Move data from hold buffer into user space.
406      * We know the entire buffer is transferred since
407      * we checked above that the read buffer is bpf_bufsize bytes.
408      */
409     error = uiomove(d->bd_hbuf, d->bd_hlen, UIO_READ, uio);

410     s = splimp();
411     d->bd_fbuf = d->bd_hbuf;
412     d->bd_hbuf = 0;
413     d->bd_hlen = 0;
414     splx(s);

415     return (error);
416 }
```

bpf.c

Figure 31.20 `bpfread` function.

344–357 The minor device number selects the BPF device from the `bpf_dtab` table. If the read buffer doesn't match the size of the BPF device buffers, `EINVAL` is returned.

Wait for data

358–364 Since multiple processes may be reading from the same BPF device, the `while` loop forces the read to continue when some other process gets to the data first. If there is data in the hold buffer, the loop is skipped. This is different from two processes tapping the same network interface through two different BPF devices (Exercise 31.2).

Immediate mode

365–373 If the device is in immediate mode and there is some data in the store buffer, the buffers are rotated and the `while` loop terminates.

No packets available

374–384 If the device is not in the immediate mode, or there is no data in the store buffer, the process sleeps until a signal arrives, the read timer expires, or data arrives in the hold buffer. If a signal arrives, `EINTR` or `ERESTART` is returned.

> Remember that a process never sees the `ERESTART` error because the error is handled by the `syscall` function and never returned to a process.

Check hold buffer

385–391 If the timer expired and data is in the hold buffer, the loop terminates.

Check store buffer

392–399 If the timer expired and there is no data in the store buffer, the read returns 0. The process must handle this case when using a timed read. If the timer expired and there is data in the store buffer, it is rotated to the hold buffer and the loop terminates.

If `tsleep` returns without an error and data is present, the `while` loop test is false and the loop terminates.

Packets are available

400–416 At this point, there is data in the hold buffer. `uiomove` moves `bd_hlen` bytes of data from the hold buffer to the process. After the move, the hold buffer is moved to the free buffer, and the buffer counts are cleared before the function returns. The comment before `uiomove` indicates that `uiomove` will always be able to copy `bd_hlen` bytes into the process because the read buffer was checked to ensure it can hold the maximum number of bytes, `bd_bufsize`.

31.6 BPF Output

Finally, we describe how to add packets to the network interface output queues with BPF. An entire data-link frame must be constructed by the process. For Ethernet this includes the source and destination hardware addresses and the frame type (Figure 4.8). The kernel will not modify the frame before putting it on the interface's output queue.

`bpfwrite` Function

The frame is passed to the BPF device with the `write` system call, which the kernel routes to `bpfwrite`, shown in Figure 31.21.

Check device number

437–449 The minor device number selects the BPF device, which must be attached to a network interface. If it isn't, `ENXIO` is returned.

Copy data into mbuf chain

450–457 If the write specified 0 bytes, 0 is returned immediately. `bpf_movein` copies the data from the process into an mbuf chain. Based on the interface type passed from `bif_dlt`, it computes the length of the packet excluding the link-layer header and returns the value in `datlen`. It also returns an initialized `sockaddr` structure in `dst`. For Ethernet, the type of this address structure will be `AF_UNSPEC`, indicating that the mbuf chain contains the data-link header for the outgoing frame. If the packet is larger than the MTU of the interface, `EMSGSIZE` is returned.

Queue packet

458–465 The resulting mbuf chain is passed to the network interface using the `if_output` function specified in the `ifnet` structure. For Ethernet, `if_output` is `ether_output`.

bpf.c

```
int
bpfwrite(dev, uio)
dev_t   dev;
struct uio *uio;
{
    struct bpf_d *d = &bpf_dtab[minor(dev)];
    struct ifnet *ifp;
    struct mbuf *m;
    int     error, s;
    static struct sockaddr dst;
    int     datlen;

    if (d->bd_bif == 0)
        return (ENXIO);

    ifp = d->bd_bif->bif_ifp;

    if (uio->uio_resid == 0)
        return (0);

    error = bpf_movein(uio, (int) d->bd_bif->bif_dlt, &m, &dst, &datlen);
    if (error)
        return (error);

    if (datlen > ifp->if_mtu)
        return (EMSGSIZE);

    s = splnet();
    error = (*ifp->if_output) (ifp, m, &dst, (struct rtentry *) 0);
    splx(s);
    /*
     * The driver frees the mbuf.
     */
    return (error);
}
```

bpf.c

Figure 31.21 `bpfwrite` function.

31.7 Summary

In this chapter we showed how BPF devices are configured, how incoming frames are passed to BPF devices, and how outgoing frames can be transmitted on a BPF device.

We showed that a single network interface can have multiple BPF taps, each with a separate filter. The store and hold buffers minimize the number of read system calls required to process incoming frames.

We focused only on the major features of BPF in this chapter. For a more detailed description of the filtering code and the other features of the BPF device, the interested reader should examine the source code and the Net/3 manual pages.

Exercises

31.1 Why is it OK to call `bpf_wakeup` in `catchpacket` before the packet is stored in the BPF buffers?

31.2 With Figure 31.20, we noted that two processes may be waiting for data from the same BPF device. With Figure 31.11, we noted that only one process at a time can open a particular BPF device. How can both of these statements be true?

31.3 What happens if the device named in the `BIOCSETIF` command does not support BPF?

32

Raw IP

32.1 Introduction

A process accesses the raw IP layer by creating a socket of type `SOCK_RAW` in the Internet domain. There are three uses for raw sockets:

1. Raw sockets allow a process to send and receive ICMP and IGMP messages.

   The Ping program uses this type of socket to send ICMP echo requests and to receive ICMP echo replies.

   Some routing daemons use this feature to track ICMP redirects that are processed by the kernel. We saw in Section 19.7 that Net/3 generates an `RTM_REDIRECT` message on a routing socket when a redirect is processed, obviating the need for this use of raw sockets.

   This feature is also used to implement protocols based on ICMP, such as router advertisement and router solicitation (Section 9.6 of Volume 1), which use ICMP but are better implemented as user processes than within the kernel.

   The multicast routing daemon uses a raw IGMP socket to send and receive IGMP messages.
2. Raw sockets let a process build its own IP headers. The Traceroute program uses this feature to build its own UDP datagrams, including the IP and UDP headers.

3. Raw sockets let a process read and write IP datagrams with an IP protocol type that the kernel doesn't support.

   The `gated` program uses this to support three routing protocols that are built directly on IP: EGP, HELLO, and OSPF.

   This type of raw socket can also be used to experiment with new transport layers on top of IP, instead of adding support to the kernel. It is usually much easier to debug code within a user process than it is within the kernel.

This chapter examines the implementation of raw IP sockets.

32.2 Code Introduction

There are five raw IP functions in a single C file, shown in Figure 32.1.

File	Description
`netinet/raw_ip.c`	raw IP functions

Figure 32.1 File discussed in this chapter.

Figure 32.2 shows the relationship of the five raw IP functions to other kernel functions.

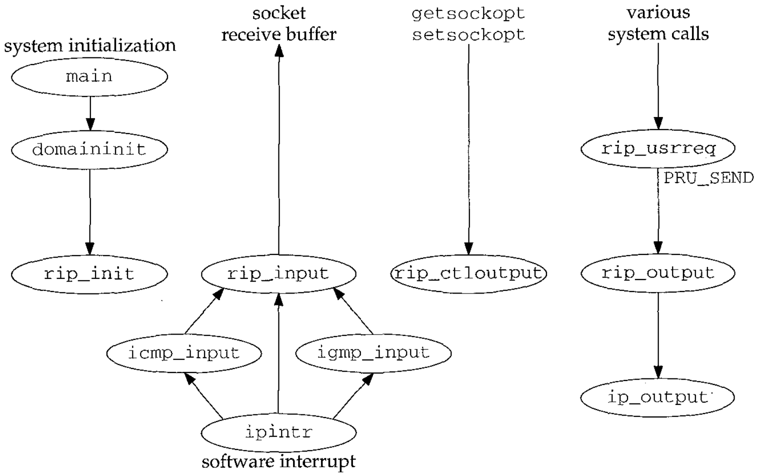


Figure 32.2 Relationship of raw IP functions to rest of kernel.

The shaded ellipses are the five functions that we cover in this chapter. Be aware that the "rip" prefix used within the raw IP functions stands for "raw IP" and not the "Routing Information Protocol," whose common acronym is RIP.

Global Variables

Four global variables are introduced in this chapter, which are shown in Figure 32.3.

Variable	Datatype	Description
`rawinpcb`	`struct inpcb`	head of the raw IP Internet PCB list
`ripsrc`	`struct sockaddr_in`	contains sender's IP address on input
`rip_recvspace`	`u_long`	default size of socket receive buffer, 8192 bytes
`rip_sendspace`	`u_long`	default size of socket send buffer, 8192 bytes

Figure 32.3 Global variables introduced in this chapter.

Statistics

Raw IP maintains two of the counters in the `ipstat` structure (Figure 8.4). We describe these in Figure 32.4.

`ipstat` member	Description	Used by SNMP
`ips_noproto`	#packets with an unknown or unsupported protocol	•
`ips_rawout`	total #raw ip packets generated	

Figure 32.4 Raw IP statistics maintained in the `ipstat` structure.

The use of the `ips_noproto` counter with SNMP is shown in Figure 8.6. Figure 8.5 shows some sample output of these two counters.

32.3 Raw IP `protosw` Structure

Unlike all other protocols, raw IP is accessed through multiple entries in the `inetsw` array. There are four entries in this structure with a socket type of `SOCK_RAW`, each with a different protocol value:

- `IPPROTO_ICMP` (protocol value of 1),
- `IPPROTO_IGMP` (protocol value of 2),
- `IPPROTO_RAW` (protocol value of 255), and
- raw wildcard entry (protocol value of 0).

The first two entries for ICMP and IGMP were described earlier (Figures 11.12 and 13.9). The difference in these four entries can be summarized as follows:

- If the process creates a raw socket (`SOCK_RAW`) with a nonzero protocol value (the third argument to `socket`), and if that value matches `IPPROTO_ICMP`, `IPPROTO_IGMP`, or `IPPROTO_RAW`, then the corresponding `protosw` entry is used.

- If the process creates a raw socket with a nonzero protocol value that is not known to the kernel, the wildcard entry with a protocol of 0 is matched by `pffindproto`. This allows a process to handle any IP protocol that is not known to the kernel, without making kernel modifications.

We saw in Section 7.8 that all entries in the `ip_protox` array that are unknown are set to point to the entry for `IPPROTO_RAW`, whose protocol switch entry we show in Figure 32.5.

Member	`inetsw[3]`	Description
`pr_type`	*SOCK_RAW*	raw socket
`pr_domain`	*&inetdomain*	raw IP is part of the Internet domain
`pr_protocol`	*IPPROTO_RAW (255)*	appears in the `ip_p` field of the IP header
`pr_flags`	*PR_ATOMIC | PR_ADDR*	socket layer flags, not used by protocol processing
`pr_input`	*rip_input*	receives messages from IP layer
`pr_output`	*0*	not used by raw IP
`pr_ctlinput`	*0*	not used by raw IP
`pr_ctloutput`	*rip_ctloutput*	respond to administrative requests from a process
`pr_usrreq`	*rip_usrreq*	respond to communication requests from a process
`pr_init`	*0*	not used by raw IP
`pr_fasttimo`	*0*	not used by raw IP
`pr_slowtimo`	*0*	not used by raw IP
`pr_drain`	*0*	not used by raw IP
`pr_sysctl`	*0*	not used by raw IP

Figure 32.5 The raw IP `protosw` structure.

We describe the three functions that begin with `rip_` in this chapter. We also cover the function `rip_output`, which is not in the protocol switch entry but is called by `rip_usrreq` when a raw IP datagram is output.

The fifth raw IP function, `rip_init`, is contained only in the wildcard entry. The initialization function must be called only once, so it could appear in either the `IPPROTO_RAW` entry or in the wildcard entry.

What Figure 32.5 doesn't show, however, is that other protocols (ICMP and IGMP) also reference some of the raw IP functions in their `protosw` entries. Figure 32.6 compares the relevant fields in the `protosw` entries for the four `SOCK_RAW` protocols. To highlight the differences, values in these rows are in a bolder font when they differ.

`protosw` entry	`SOCK_RAW` protocol type			
	`IPPROTO_ICMP` (1)	`IPPROTO_IGMP` (2)	`IPPROTO_RAW` (255)	wildcard (0)
`pr_input`	***icmp_input***	***igmp_input***	***rip_input***	***rip_input***
`pr_output`	*rip_output*	*rip_output*	*rip_output*	*rip_output*
`pr_ctloutput`	*rip_ctloutput*	*rip_ctloutput*	*rip_ctloutput*	*rip_ctloutput*
`pr_usrreq`	*rip_usrreq*	*rip_usrreq*	*rip_usrreq*	*rip_usrreq*
`pr_init`	*0*	***igmp_init***	*0*	***rip_init***
`pr_sysctl`	***icmp_sysctl***	*0*	*0*	*0*
`pr_fasttimo`	*0*	***igmp_fasttimo***	*0*	*0*

Figure 32.6 Comparison of protocol switch values for raw sockets.

The implementation of raw sockets has changed with the different BSD releases. The entry with a protocol of `IPPROTO_RAW` has always been used as the wildcard entry in the `ip_protox` table for unknown IP protocols. The entry with a protocol of 0 has always been the default entry, to allow processes to read and write IP datagrams with a protocol that the kernel doesn't support.

Usage of the `IPPROTO_RAW` entry by a process started when Traceroute was developed by Van Jacobson, because Traceroute was the first process that needed to write its own IP headers (to change the TTL field). The kernel patches to 4.3BSD and Net/1 to support Traceroute included a change to `rip_output` so that if the protocol was `IPPROTO_RAW`, it was assumed the process had passed a complete IP datagram, including the IP header. This was changed with Net/2 when the `IP_HDRINCL` socket option was introduced, removing this overloading of the `IPPROTO_RAW` protocol and allowing a process to send its own IP header with the wildcard entry.

32.4 `rip_init` Function

The `domaininit` function calls the raw IP initialization function `rip_init` (Figure 32.7) at system initialization time.

raw_ip.c

```
void
rip_init()
{

    rawinpcb.inp_next = rawinpcb.inp_prev = &rawinpcb;
}
```

raw_ip.c

Figure 32.7 `rip_init` function.

The only action performed by this function is to set the next and previous pointers in the head PCB (`rawinpcb`) to point to itself. This is an empty doubly linked list.

Whenever a socket of type `SOCK_RAW` is created by the `socket` system call, we'll see that the raw IP `PRU_ATTACH` function creates an Internet PCB and puts it onto the `rawinpcb` list.

32.5 `rip_input` Function

Since all entries in the `ip_protox` array for unknown protocols are set to point to the entry for `IPPROTO_RAW` (Section 7.8), and since the `pr_input` function for this protocol is `rip_input` (Figure 32.6), this function is called for all IP datagrams that have a protocol value that the kernel doesn't recognize. But from Figure 32.2 we see that both ICMP and IGMP also call `rip_input`. This happens under the following conditions:

- `icmp_input` calls `rip_input` for all unknown ICMP message types and for all ICMP messages that are not reflected.
- `igmp_input` calls `rip_input` for all IGMP packets.

One reason for calling `rip_input` in these two cases is to allow a process with a raw socket to handle new ICMP and IGMP messages that might not be supported by the kernel.

Figure 32.8 shows the `rip_input` function.

raw_ip.c

```
void
rip_input(m)
struct mbuf *m;
{
    struct ip *ip = mtod(m, struct ip *);
    struct inpcb *inp;
    struct socket *last = 0;

    ripsrc.sin_addr = ip->ip_src;
    for (inp = rawinpcb.inp_next; inp != &rawinpcb; inp = inp->inp_next) {
        if (inp->inp_ip.ip_p && inp->inp_ip.ip_p != ip->ip_p)
            continue;
        if (inp->inp_laddr.s_addr &&
            inp->inp_laddr.s_addr == ip->ip_dst.s_addr)
            continue;
        if (inp->inp_faddr.s_addr &&
            inp->inp_faddr.s_addr == ip->ip_src.s_addr)
            continue;
        if (last) {
            struct mbuf *n;
            if (n = m_copy(m, 0, (int) M_COPYALL)) {
                if (sbappendaddr(&last->so_rcv, &ripsrc,
                                n, (struct mbuf *) 0) == 0)
                    /* should notify about lost packet */
                    m_freem(n);
                else
                    sorwakeup(last);
            }
        }
        last = inp->inp_socket;
    }
    if (last) {
        if (sbappendaddr(&last->so_rcv, &ripsrc,
                        m, (struct mbuf *) 0) == 0)
            m_freem(m);
        else
            sorwakeup(last);
    } else {
        m_freem(m);
        ipstat.ips_noproto++;
        ipstat.ips_delivered--;
    }
}
```

raw_ip.c

Figure 32.8 `rip_input` function.

Save source IP address

59–66 The source address from the IP datagram is put into the global variable `ripsrc`, which becomes an argument to `sbappendaddr` whenever a matching PCB is found. Unlike UDP, there is no concept of a port number with raw IP, so the `sin_port` field in the `sockaddr_in` structure is always 0.

Search all raw IP PCBs for one or more matching entries

67–88 Raw IP handles its list of PCBs differently from UDP and TCP. We saw that these two protocols maintain a pointer to the PCB for the most recently received datagram (a one-behind cache) and call the generic function `in_pcblookup` to search for a single "best" match when the received datagram does not equal the cache entry. Raw IP has completely different criteria for a matching PCB, so it searches the PCB list itself. `in_pcblookup` cannot be used because a raw IP datagram can be delivered to multiple sockets, so every PCB on the raw PCB list must be scanned. This is similar to UDP's handling of a received datagram destined for a broadcast or multicast address (Figure 23.26).

Compare protocols

68–69 If the protocol field in the PCB is nonzero, and if it doesn't match the protocol field in the IP header, the PCB is ignored. This implies that a raw socket with a protocol value of 0 (the third argument to `socket`) can match any received raw IP datagram.

Compare local and foreign IP addresses

70–75 If the local address in the PCB is nonzero, and if it doesn't match the destination IP address in the IP header, the PCB is ignored. If the foreign address in the PCB is nonzero, and if it doesn't match the source IP address in the IP header, the PCB is ignored.

These three tests imply that a process can create a raw socket with a protocol of 0, not bind a local address, and not connect to a foreign address, and the process receives *all* datagrams processed by `rip_input`.

Lines 71 and 74 both contain the same bug: the test for equality should be a test for inequality.

Pass copy of received datagram to processes

76–94 `sbappendaddr` passes a copy of the received datagram to the process. The use of the variable `last` is similar to what we saw in Figure 23.26: since `sbappendaddr` releases the mbuf after placing it onto the appropriate queue, if more than one process receives a copy of the datagram, `rip_input` must make a copy by calling `m_copy`. But if only one process receives the datagram, there's no need to make a copy.

Undeliverable datagram

95–99 If no matching sockets are found for the datagram, the mbuf is released, `ips_noproto` is incremented, and `ips_delivered` is decremented. This latter counter was incremented by IP just before calling the `rip_input` (Figure 8.15). It must be decremented so that the two SNMP counters, `ipInDiscards` and `ipInDelivers` (Figure 8.6) are correct, since the datagram was not really delivered to a transport layer.

At the beginning of this section we mentioned that `icmp_input` calls `rip_input` for unknown message types and for messages that are not reflected. This means that the receipt of an ICMP host unreachable causes `ips_noproto` to be incremented if there are no raw listeners whose PCB is matched by `rip_input`. That's one reason this counter has such a large value in Figure 8.5. The description of this counter as being "unknown or unsupported protocols" is not entirely accurate.

Net/3 does not generate an ICMP destination unreachable message with code 2 (protocol unreachable) when an IP datagram is received with a protocol field that is not handled by either the kernel or some process through a raw socket. RFC 1122 says an implementation should generate this ICMP error. (See Exercise 32.4.)

32.6 `rip_output` Function

We saw in Figure 32.6 that `rip_output` is called for output for raw sockets by ICMP, IGMP, and raw IP. Output occurs when the application calls one of the five write functions: `send`, `sendto`, `sendmsg`, `write`, or `writev`. If the socket is connected, any of the five functions can be called, although a destination address cannot be specified with `sendto` or `sendmsg`. If the socket is unconnected, only `sendto` and `sendmsg` can be called, and a destination address must be specified.

The function `rip_output` is shown in Figure 32.9.

Kernel fills in IP header

119–128 If the `IP_HDRINCL` socket option is not defined, `M_PREPEND` allocates room for an IP header, and fields in the IP header are filled in. The fields that are not filled in here are left for `ip_output` to initialize (Figure 8.22). The protocol field is set to the value stored in the PCB, which we'll see in Figure 32.10 is the third argument to the `socket` system call.

The TOS is set to 0 and the TTL to 255. These values are always used for a raw socket when the kernel fills in the header. This differs from UDP and TCP where the process had the capability of setting the `IP_TTL` and `IP_TOS` socket options.

129 Any IP options set by the process with the `IP_OPTIONS` socket options are passed to `ip_output` through the `opts` variable.

Caller fills in IP header: `IP_HDRINCL` socket option

130–133 If the `IP_HDRINCL` socket option is set, the caller supplies a completed IP header at the front of the datagram. The only modification made to this IP header is to set the ID field if the value supplied by the process is 0. The ID field of an IP datagram can be 0. The assignment of the ID field here by `rip_output` is just a convention that allows the process to set it to 0, asking the kernel to assign an ID value based on the kernel's current `ip_id` variable.

134–136 The `opts` variable is set to a null pointer, which ignores any IP options the process may have set with the `IP_OPTIONS` socket option. The convention here is that if the caller builds its own IP header, that header includes any IP options the caller might want. The `flags` variable must also include the `IP_RAWOUTPUT` flag, telling `ip_output` to leave the header alone.

raw_ip.c

```
105 int
106 rip_output(m, so, dst)
107 struct mbuf *m;
108 struct socket *so;
109 u_long  dst;
110 {
111     struct ip *ip;
112     struct inpcb *inp = sotoinpcb(so);
113     struct mbuf *opts;
114     int     flags = (so->so_options & SO_DONTROUTE) | IP_ALLOWBROADCAST;

115     /*
116      * If the user handed us a complete IP packet, use it.
117      * Otherwise, allocate an mbuf for a header and fill it in.
118      */
119     if ((inp->inp_flags & INP_HDRINCL) == 0) {
120         M_PREPEND(m, sizeof(struct ip), M_WAIT);
121         ip = mtod(m, struct ip *);
122         ip->ip_tos = 0;
123         ip->ip_off = 0;
124         ip->ip_p = inp->inp_ip.ip_p;
125         ip->ip_len = m->m_pkthdr.len;
126         ip->ip_src = inp->inp_laddr;
127         ip->ip_dst.s_addr = dst;
128         ip->ip_ttl = MAXTTL;
129         opts = inp->inp_options;
130     } else {
131         ip = mtod(m, struct ip *);
132         if (ip->ip_id == 0)
133             ip->ip_id = htons(ip_id++);
134         opts = NULL;
135         /* XXX prevent ip_output from overwriting header fields */
136         flags |= IP_RAWOUTPUT;
137         ipstat.ips_rawout++;
138     }
139     return (ip_output(m, opts, &inp->inp_route, flags, inp->inp_moptions));
140 }
```

raw_ip.c

Figure 32.9 rip_output function.

137 The counter `ips_rawout` is incremented. Running Traceroute causes this variable to be incremented by 1 for each datagram sent by Traceroute.

> The operation of `rip_output` has changed over time. When the `IP_HDRINCL` socket option is used in Net/3, the only change made to the IP header by `rip_output` is to set the ID field, if the process sets it to 0. The Net/3 `ip_output` function does nothing to the IP header fields because the `IP_RAWOUTPUT` flag is set. Net/2, however, always set certain fields in the IP header, even if the `IP_HDRINCL` socket option was set: the IP version was set to 4, the fragment offset was set to 0, and the more-fragments flag was cleared.

32.7 `rip_usrreq` Function

The protocol's user-request function is called for a variety of operations. As with the UDP and TCP user-request functions, `rip_usrreq` is a large `switch` statement, with one `case` for each `PRU_`*xxx* request.

The `PRU_ATTACH` request, shown in Figure 32.10, is from the `socket` system call.

raw_ip.c

```
194 int
195 rip_usrreq(so, req, m, nam, control)
196 struct socket *so;
197 int     req;
198 struct mbuf *m, *nam, *control;
199 {
200     int     error = 0;
201     struct inpcb *inp = sotoinpcb(so);
202     extern struct socket *ip_mrouter;
203     switch (req) {

204     case PRU_ATTACH:
205         if (inp)
206             panic("rip_attach");
207         if ((so->so_state & SS_PRIV) == 0) {
208             error = EACCES;
209             break;
210         }
211         if ((error = soreserve(so, rip_sendspace, rip_recvspace)) ||
212             (error = in_pcballoc(so, &rawinpcb)))
213             break;
214         inp = (struct inpcb *) so->so_pcb;
215         inp->inp_ip.ip_p = (int) nam;
216         break;
```

raw_ip.c

Figure 32.10 `rip_usrreq` function: `PRU_ATTACH` request.

194–206 Since the `socket` function creates a new `socket` structure each time it is called, that structure cannot point to an Internet PCB.

Verify superuser

207–210 Only the superuser can create a raw socket. This is to prevent random users from writing their own IP datagrams to the network.

Create Internet PCB and reserve buffer space

211–215 Space is reserved for input and output queues, and `in_pcballoc` allocates a new Internet PCB. The PCB is added to the raw IP PCB list (`rawinpcb`). The PCB is linked to the `socket` structure. The `nam` argument to `rip_usrreq` is the third argument to the `socket` system call: the protocol. It is stored in the PCB since it is used by `rip_input` to demultiplex received datagrams, and its value is placed into the protocol field of outgoing datagrams by `rip_output` (if `IP_HDRINCL` is not set).

A raw IP socket can be connected to a foreign IP address similar to a UDP socket being connected to a foreign IP address. This fixes the foreign IP address from which the raw socket receives datagrams, as we saw in `rip_input`. Since raw IP is a connectionless protocol like UDP, a `PRU_DISCONNECT` request can occur in two cases:

1. When a connected raw socket is closed, `PRU_DISCONNECT` is called before `PRU_DETACH`.
2. When a `connect` is issued on an already-connected raw socket, `soconnect` issues the `PRU_DISCONNECT` request before the `PRU_CONNECT` request.

Figure 32.11 shows the `PRU_DISCONNECT`, `PRU_ABORT`, and `PRU_DETACH` requests.

raw_ip.c

```
217      case PRU_DISCONNECT:
218          if ((so->so_state & SS_ISCONNECTED) == 0) {
219              error = ENOTCONN;
220              break;
221          }
222          /* FALLTHROUGH */

223      case PRU_ABORT:
224          soisdisconnected(so);
225          /* FALLTHROUGH */

226      case PRU_DETACH:
227          if (inp == 0)
228              panic("rip_detach");
229          if (so == ip_mrouter)
230              ip_mrouter_done();
231          in_pcbdetach(inp);
232          break;
```

raw_ip.c

Figure 32.11 `rip_usrreq` function: `PRU_DISCONNECT`, `PRU_ABORT`, and `PRU_DETACH` requests.

217–222 The socket must already be connected to disconnect or else an error is returned.

223–225 A `PRU_ABORT` abort should never be issued for a raw IP socket, but this `case` also handles the fall through from `PRU_DISCONNECT`. The socket is marked as disconnected.

226–230 The `close` system call issues the `PRU_DETACH` request, and this `case` also handles the fall through from the `PRU_DISCONNECT` request. If the `socket` structure is the one used for multicast routing (`ip_mrouter`), multicast routing is disabled by calling `ip_mrouter_done`. Normally the `mrouted`(8) daemon issues the `DVMRP_DONE` socket option to disable multicast routing, so this check handles the case of the router daemon terminating (i.e., crashing) without issuing the socket option.

231 The Internet PCB is released by `in_pcbdetach`, which also removes the PCB from the list of raw IP PCBs (`rawinpcb`).

A raw IP socket can be bound to a local IP address with the `PRU_BIND` request, shown in Figure 32.12. We saw in `rip_input` that the socket will receive only datagrams sent to this IP address.

233–250 The process fills in a `sockaddr_in` structure with the local IP address. The following three conditions must all be true, or else the error `EADDRNOTAVAIL` is returned:

raw_ip.c

```
    case PRU_BIND:
        {
            struct sockaddr_in *addr = mtod(nam, struct sockaddr_in *);

            if (nam->m_len != sizeof(*addr)) {
                error = EINVAL;
                break;
            }
            if ((ifnet == 0) ||
                ((addr->sin_family != AF_INET) &&
                 (addr->sin_family != AF_IMPLINK)) ||
                (addr->sin_addr.s_addr &&
                 ifa_ifwithaddr((struct sockaddr *) addr) == 0)) {
                error = EADDRNOTAVAIL;
                break;
            }
            inp->inp_laddr = addr->sin_addr;
            break;
        }
```

raw_ip.c

Figure 32.12 `rip_usrreq` function: `PRU_BIND` request.

1. at least one interface must be configured,
2. the address family must be `AF_INET` (or `AF_IMPLINK`, a historical artifact), and
3. if the IP address being bound is not 0.0.0.0, it must correspond to a local interface. For the call to `ifa_ifwithaddr` to succeed, the port number in the caller's `sockaddr_in` must be 0.

The local IP address is stored in the PCB.

A process can also connect a raw IP socket to a particular foreign IP address. We saw in `rip_input` that this restricts the process so that it receives only IP datagrams with a source IP address equal to the connected IP address. A process has the option of calling `bind`, `connect`, both, or neither, depending on the type of filtering it wants `rip_input` to place on received datagrams. Figure 32.13 shows the `PRU_CONNECT` request.

251–270 If the caller's `sockaddr_in` is initialized correctly and at least one IP interface is configured, the specified foreign IP address is stored in the PCB. Notice that this process differs from the connection of a UDP socket to a foreign address. In the UDP case, `in_pcbconnect` acquires a route to the foreign address and also stores the outgoing interface as the local address (Figure 22.9). With raw IP, only the foreign IP address is stored in the PCB, and unless the process also calls `bind`, only the foreign address is compared by `rip_input`.

raw_ip.c

```
251     case PRU_CONNECT:
252         {
253             struct sockaddr_in *addr = mtod(nam, struct sockaddr_in *);

254             if (nam->m_len != sizeof(*addr)) {
255                 error = EINVAL;
256                 break;
257             }
258             if (ifnet == 0) {
259                 error = EADDRNOTAVAIL;
260                 break;
261             }
262             if ((addr->sin_family != AF_INET) &&
263                 (addr->sin_family != AF_IMPLINK)) {
264                 error = EAFNOSUPPORT;
265                 break;
266             }
267             inp->inp_faddr = addr->sin_addr;
268             soisconnected(so);
269             break;
270         }
```

raw_ip.c

Figure 32.13 `rip_usrreq` function: `PRU_CONNECT` request.

A call to `shutdown` specifying that the process has finished sending data generates the `PRU_SHUTDOWN` request, although it is rare for a process to issue this system call for a raw IP socket. Figure 32.14 shows the `PRU_CONNECT2` and `PRU_SHUTDOWN` requests.

raw_ip.c

```
271     case PRU_CONNECT2:
272         error = EOPNOTSUPP;
273         break;

274         /*
275          * Mark the connection as being incapable of further input.
276          */
277     case PRU_SHUTDOWN:
278         socantsendmore(so);
279         break;
```

raw_ip.c

Figure 32.14 `rip_usrreq` function: `PRU_CONNECT2` and `PRU_SHUTDOWN` requests.

271–273 The `PRU_CONNECT2` request is not supported for a raw IP socket.

274–279 `socantsendmore` sets the socket's flags to prevent any future output.

In Figure 23.14 we showed how the five write functions call the protocol's `pr_usrreq` function with a `PRU_SEND` request. We show this request in Figure 32.15.

raw_ip.c

```
280         /*
281          * Ship a packet out.  The appropriate raw output
282          * routine handles any massaging necessary.
283          */
284     case PRU_SEND:
285         {
286             u_long  dst;

287             if (so->so_state & SS_ISCONNECTED) {
288                 if (nam) {
289                     error = EISCONN;
290                     break;
291                 }
292                 dst = inp->inp_faddr.s_addr;
293             } else {
294                 if (nam == NULL) {
295                     error = ENOTCONN;
296                     break;
297                 }
298                 dst = mtod(nam, struct sockaddr_in *)->sin_addr.s_addr;
299             }
300             error = rip_output(m, so, dst);
301             m = NULL;
302             break;
303         }
```

raw_ip.c

Figure 32.15 `rip_usrreq` function: `PRU_SEND` request.

280–303 If the socket state is connected, the caller cannot specify a destination address (the `nam` argument). Likewise, if the state is unconnected, a destination address is required. If all is OK, in either state, `dst` is set to the destination IP address. `rip_output` sends the datagram. The mbuf pointer `m` is set to a null pointer, to prevent it from being released at the end of the function. This is because the interface output routine will release the mbuf after it has been output. (Remember that `rip_output` passes the mbuf chain to `ip_output`, who appends it to the interface's output queue.)

The final part of `rip_usrreq` is shown in Figure 32.16. The `PRU_SENSE` request, generated by the `fstat` system call, returns nothing. The `PRU_SOCKADDR` and `PRU_PEERADDR` requests are from the `getsockname` and `getpeername` system calls, respectively. The remaining requests are not supported.

319–324 The functions `in_setsockaddr` and `in_setpeeraddr` fetch the information from the PCB, storing the result in the `nam` argument.

raw_ip.c

```
    case PRU_SENSE:
        /*
         * fstat: don't bother with a blocksize.
         */
        return (0);

        /*
         * Not supported.
         */
    case PRU_RCVOOB:
    case PRU_RCVD:
    case PRU_LISTEN:
    case PRU_ACCEPT:
    case PRU_SENDOOB:
        error = EOPNOTSUPP;
        break;

    case PRU_SOCKADDR:
        in_setsockaddr(inp, nam);
        break;

    case PRU_PEERADDR:
        in_setpeeraddr(inp, nam);
        break;

    default:
        panic("rip_usrreq");
    }
    if (m != NULL)
        m_freem(m);
    return (error);
}
```

raw_ip.c

Figure 32.16 `rip_usrreq` function: remaining requests.

32.8 `rip_ctloutput` Function

The `setsockopt` and `getsockopt` system calls invoke the `rip_ctloutput` function. Only one IP socket option is handled here, along with eight socket options related to multicast routing.

Figure 32.17 shows the first part of the `rip_ctloutput` function.

144–172 The size of the mbuf that contains either the new value of the option or will hold the current value of the option must be at least as large as an integer. For the `setsockopt` system call, the flag is set if the integer value in the mbuf is nonzero, or cleared otherwise. For the `getsockopt` system call, the value returned in the mbuf is either 0 or the nonzero value of the flag. The function returns, to avoid the processing at the end of the `switch` statement for other IP options.

raw_ip.c

```
int
rip_ctloutput(op, so, level, optname, m)
int     op;
struct socket *so;
int     level, optname;
struct mbuf **m;
{
    struct inpcb *inp = sotoinpcb(so);
    int     error;

    if (level != IPPROTO_IP)
        return (EINVAL);

    switch (optname) {

    case IP_HDRINCL:
        if (op == PRCO_SETOPT || op == PRCO_GETOPT) {
            if (m == 0 || *m == 0 || (*m)->m_len < sizeof(int))
                        return (EINVAL);
            if (op == PRCO_SETOPT) {
                if (*mtod(*m, int *))
                            inp->inp_flags |= INP_HDRINCL;
                else
                    inp->inp_flags &= ~INP_HDRINCL;
                (void) m_free(*m);
            } else {
                (*m)->m_len = sizeof(int);
                *mtod(*m, int *) = inp->inp_flags & INP_HDRINCL;
            }
            return (0);
        }
        break;
```

raw_ip.c

Figure 32.17 `rip_usrreq` function: process `IP_HDRINCL` socket option.

raw_ip.c

```
    case DVMRP_INIT:
    case DVMRP_DONE:
    case DVMRP_ADD_VIF:
    case DVMRP_DEL_VIF:
    case DVMRP_ADD_LGRP:
    case DVMRP_DEL_LGRP:
    case DVMRP_ADD_MRT:
    case DVMRP_DEL_MRT:


                           /* shown in Figure 14.9 */


    }
    return (ip_ctloutput(op, so, level, optname, m));
}
```

raw_ip.c

Figure 32.18 `rip_usrreq` function: process multicast routing socket option.

Figure 32.18 shows the last portion of the `rip_ctloutput` function. It handles eight multicast routing socket options.

173–188 These eight socket options are valid only for the `setsockopt` system call. They are processed by the `ip_mrouter_cmd` function as discussed with Figure 14.9.

189 Any other IP socket options, such as `IP_OPTIONS` to set the IP options, are processed by `ip_ctloutput`.

32.9 Summary

Raw sockets provide three capabilities for an IP host.

1. They are used to send and receive ICMP and IGMP messages.
2. They allow a process to build its own IP headers.
3. They allow additional IP-based protocols to be supported in a user process.

We saw that raw IP output is simple—it just fills in a few fields in the IP header—but it allows a process to supply its own IP header. This allows diagnostic programs to create any type of IP datagram.

Raw IP input provides three types of filtering for incoming IP datagrams. The process chooses to receive datagrams based on (1) the protocol field, (2) the source IP address (set by `connect`), and (3) the destination IP address (set by `bind`). The process chooses which combination of these three filters (if any) to apply.

Exercises

32.1 Assume the `IP_HDRINCL` socket option is not set. What value will `rip_output` place into the IP header protocol field (ip_p) when the third argument to `socket` is 0? What value will `rip_output` place into this field when the third argument to `socket` is `IPPROTO_RAW` (255)?

32.2 A process creates a raw socket with a protocol value of `IPPROTO_RAW` (255). What type of IP datagrams will the process receive on this socket?

32.3 A process creates a raw socket with a protocol value of 0. What type of IP datagrams will the process receive on this socket?

32.4 Modify `rip_input` to send an ICMP destination unreachable with code 2 (protocol unreachable) when appropriate. Be careful not to generate the error for received ICMP and IGMP packets for which `rip_input` is called.

32.5 If a process wants to write its own IP datagrams with its own IP header, what are the differences in using a raw IP socket with the `IP_HDRINCL` option, and using BPF (Chapter 31)?

32.6 When would a process read from a raw IP socket, and when would it read from BPF?

Epilogue

"We have come a long way. Nine chapters stuffed with code is a lot to negotiate. If you didn't assimilate all of it the first time through, don't worry—you weren't really expected to. Even the best of code takes time to absorb, and you seldom grasp all the implications until you try to use and modify the program. Much of what you learn about programming comes only from working with the code: reading, revising and rereading."

From the Epilogue of *Software Tools* [Kernighan and Plauger 1976].

"In fact, this RFC will argue that modularity is one of the chief villains in attempting to obtain good performance, so that the designer is faced with a delicate and inevitable tradeoff between good structure and good performance."

From RFC 817 [Clark 1982].

This text has provided a long and detailed examination of a significant piece of a real operating system. Versions of the code presented in the text are shipped as part of the Unix kernel with most flavors of Unix today, along with many non-Unix systems.

The code that we've examined is not perfect and it is not the only way to write a TCP/IP protocol stack. It has been modified, enhanced, tweaked, and maligned over the past 15 years by many people. Large portions of the code that we've presented weren't even written at the U. C. Berkeley Computer Systems Research Group: the multicasting code was written by Steve Deering, the long fat pipe support was added by Thomas Skibo, portions of the TCP code were written by Van Jacobson, and so on. The code contains `goto`s (221 to be exact), many large functions (e.g., `tcp_input` and `tcp_output`), and numerous examples of questionable coding style. (We tried to note these items when discussing the code.) Nevertheless, the code is unquestionably "industrial strength" and continues to be the base upon which new features are added and the standard upon which other implementations are measured.

The Berkeley networking code was designed on VAXes when a VAX-11/780 with 4 megabytes of memory was a big system. For that reason some of the design features (e.g., mbufs) emphasized memory savings over higher performance. This would change if the code were rewritten from scratch today.

There has been a strong push over the last few years toward higher performance of networking software, as the underlying networks become faster (e.g., FDDI and ATM) and as high-bandwidth applications become more prevalent (e.g., voice and video). Whenever designing networking software within the kernel of an operating system, clarity normally gives way to speed [Clark 1982]. This will continue in any real-world implementation.

The research implementation of the Internet protocols described in [Partridge 1993] and [Jacobson 1993] is a move toward much higher performance. [Jacobson 1993] reports the code is 10 to 100 times faster than the implementation described in this book. Mbufs, software interrupts, and much of the protocol layering evident in BSD systems are gone. If widely released, this implementation could become the standard that others are measured against in the future.

In July 1994 the successor to IP version 4, IP version 6 (IPv6), was announced. It uses 128-bit (16-byte) addresses. Many changes will take place with the IP and ICMP protocols, but the transport layers, UDP and TCP, will remain virtually the same. (There is talk of a TCPng, the next generation of TCP, but the authors think just upgrading IP will provide enough of a challenge for the hundreds of vendors and millions of users across the world to put off any changes to TCP.) It will take a year or two for vendor-supported implementations to appear, and many years after that for end users to migrate their hosts and routers to IPv6. Research implementations of IPv6 based on the code in this text should appear in early 1995.

To continue your understanding of the Berkeley networking code, the best course of action at this point is to obtain the source code, and modify it. The source code is easily obtainable (Appendix B) and numerous exercises throughout the text suggest modifications.

Appendix A

Solutions to Selected Exercises

Chapter 1

1.2 SLIP drivers execute at `spltty` (Figure 1.13), which must be a priority lower than or equal to `splimp` and must be a priority higher than `splnet`. Therefore the SLIP drivers are blocked from interrupting.

Chapter 2

2.1 The `M_EXT` flag is a property of the mbuf itself, not a property of the packet described by the mbuf.

2.2 The caller asks for more than 100 (`MHLEN`) contiguous bytes.

2.3 This is infeasible since clusters can be pointed to by multiple mbufs (Section 2.9). Also, there is no room in a cluster for a back pointer (Exercise 2.4).

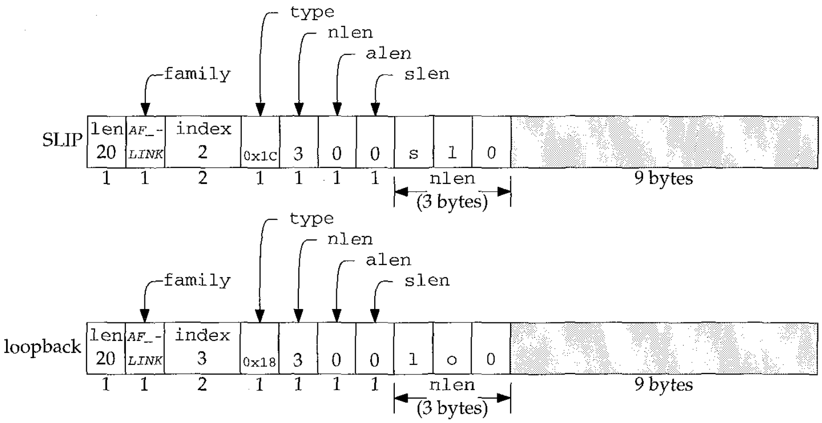
2.4 In the macros `MCLALLOC` and `MCLFREE` in `<sys/mbuf.h>` we see that the reference count is an array named `mclrefcnt`. This array is allocated when the kernel is initialized in the file `machdep.c`.

Chapter 3

3.3 A large interactive queue would defeat the purpose of the queue by delaying new interactive traffic behind the existing interactive data.

3.4 Since the `sl_softc` structures are all declared as global variables, they are initialized to 0 when the kernel starts.

3.5



Chapter 4

4.1 `leread` must examine the packet to decide if it needs to be discarded after it is passed to BPF. Since a BPF tap can enable promiscuous mode on the interface, packets may be addressed to some other system on the Ethernet and must be discarded after BPF has processed them.

When the interface is not tapped, the tests must be done in `ether_input`.

4.2 If the tests were reversed, the broadcast flag would never be set.

If the second `if` wasn't preceded by an `else`, every broadcast packet would also have the multicast flag set.

Chapter 5

5.1 The loopback interface does not need an input function because all its packets are received directly from `looutput`, which performs the "input" functions.

5.2 The stack allocation is faster than dynamic memory allocation. Performance is important for BPF processing, since the code is executed for each incoming packet.

5.5 The first character that overflows the buffer is discarded, `SC_ERROR` is set, and `slinput` resets the cluster pointers to begin collecting characters at the start of the buffer. Because `SC_ERROR` is set, `slinput` discards the frame when it receives the SLIP END character.

5.6 IP discards the packet when the checksum is found to be invalid or when it notices that the IP header length does not match the physical packet size.

5.7 Since `ifp` points to the first member of a `le_softc` structure,

```
sc = (struct le_softc *)ifp;
```

initializes `sc` correctly.

5.8 This is very hard to do. Some routers may send ICMP source quench messages when they begin discarding packets but Net/3 discards these messages for UDP sockets (Figure 23.30). An application would have to begin using the same techniques used by TCP: estimation of the available bandwidth and delay on roundtrip times for acknowledged datagrams.

Chapter 6

6.1 Before IP subnetting (RFC 950 [Mogul and Postel 1985]), the network and host portions of IP addresses always appeared on byte boundaries. The definition of an `in_addr` structure was

```
struct in_addr {
        union {
                struct { u_char s_b1, s_b2, s_b3, s_b4; } S_un_b;
                struct { u_short s_w1, s_w2; } S_un_w;
                u_long S_addr;
        } S_un;
#define s_addr  S_un.S_addr         /* should be used for all code */
#define s_host  S_un.S_un_b.s_b2    /* OBSOLETE: host on imp */
#define s_net   S_un.S_un_b.s_b1    /* OBSOLETE: network */
#define s_imp   S_un.S_un_w.s_w2    /* OBSOLETE: imp */
#define s_impno S_un.S_un_b.s_b4    /* OBSOLETE: imp # */
#define s_lh    S_un.S_un_b.s_b3    /* OBSOLETE: logical host */
};
```

The Internet address could be accessed as 8-bit bytes, 16-bit words, or a single 32-bit address. The macros `s_host`, `s_net`, `s_imp`, and so on have names that correspond to the physical structure of early TCP/IP networks.

The use of subnetting and supernetting makes the byte and word divisions obsolete.

6.2 A pointer to the structure labeled `sl_softc[0]` is returned.

6.3 The interface output functions, such as `ether_output`, have a pointer only to the `ifnet` structure for the interface, and not to an `ifaddr` structure. Using the IP address in the `arpcom` structure (which is the last IP address assigned to the interface) avoids having to select an address from the `ifaddr` address list.

6.4 Only a superuser process can create a raw IP socket. By using a UDP socket, any process can examine the interface configurations but the kernel can still require superuser privileges to modify the interface addresses.

6.5 Three functions loop through a netmask 1 byte at a time. These are `ifa_ifwithnet`, `ifaof_ifpforaddr`, and `rt_maskedcopy`. A shorter mask improves the performance of these functions.

6.6 The Telnet connection is established with the remote system. Net/2 systems shouldn't forward these packets, and other systems should never accept loopback packets that arrive on any interface other than the loopback interface.

Chapter 7

7.1 The following call returns a pointer to `inetsw[6]`:

```
pffindproto(PF_INET, 0, SOCK_RAW);
```

Chapter 8

8.1 Probably not. The system could not respond to any broadcasts since it would have no source address to use in the reply.

8.4 Since the packet has been damaged, there is no way of knowing if the addresses in the header are correct or not.

8.5 If an application selects a source address that differs from the address of the selected outgoing interface, redirects from the selected next-hop router fail. The next-hop router sees a source address different from that of the subnetwork on which it was transmitted and does not send a redirect message. This is a consequence of implementing the weak end system model and is noted in RFC 1122.

8.6 The new host thinks the broadcast packet is the address of some other host in the unsubnetted network and trys to send it back out on the network. The network interface begins broadcasting ARP requests for the broadcast address, which are never answered.

8.7 The decrement of the TTL is done after the comparison for less than or equal to 1 to avoid the potential error of decrementing a received TTL of 0 to become 255.

8.8 If two routers each consider the other the best next-hop for a packet, a routing loop exists. Until the loop is removed, the original packet bounces between the two routers and each one sends an ICMP redirect back to the source host if that host is on the same network as the routers. Loops may exist when the routing tables are temporarily inconsistent during a routing update.

The TTL of the original packet eventually reaches 0 and the packet is discarded. This is one of the primary reasons why the TTL field exists.

8.9 The four Ethernet broadcast addresses would not be checked because they do not belong to the receiving interface. The limited-broadcast addresses would be checked. This implies that a system on a SLIP link can communicate with the system on the other end without knowing the other system's address by utilizing the limited-broadcast address.

8.10 ICMP error messages are generated only for the initial fragment of a datagram, which always has an offset of 0. The host and network forms for 0 are the same, so no conversion is necessary.

Chapter 9

9.1 RFC 1122 says that the behavior is implementation dependent when conflicting options appear in a packet. Net/3 processes the first source route option correctly, but since this updates `ip_dst` in the packet header, the second source route processing will be incorrect.

9.2 The host within the network can be used as a relay to access other hosts within the network. To communicate with an otherwise-blocked host, the source host need only construct packets with a loose route to the relay host and then to the final destination host. The router does not drop the packets because the destination address is the relay host, which will process the route and forward the packet to the final destination host. The destination host reverses the route and uses the relay host to return packets.

9.3 The same principle from the previous exercise applies. We pick a relay router that can communicate with the source and destination hosts and construct source routes to pass through the relay and to the destination. The relay router must be on the same network as the destination host so that a default route is not required for communication.

This technique can be extended to allow two hosts to communicate even if they do not have routes to each other, as long as they can find willing relay hosts.

9.4 If the source route is the only IP option, the NOP option causes all the IP addresses to be on a 4-byte boundary in the IP header. This can optimize memory references to these addresses on many architectures. This alignment technique also works when multiple options are present if each option is padded with NOPs to a 4-byte boundary.

9.5 A nonstandard time value cannot be confused with a standard value since the largest standard time value is 86,399,999 ($24 \times 60 \times 60 \times 1000 - 1$) and this value can be represented in 28 bits, which avoids any conflict with the high-order bit since time values are 32 bits long.

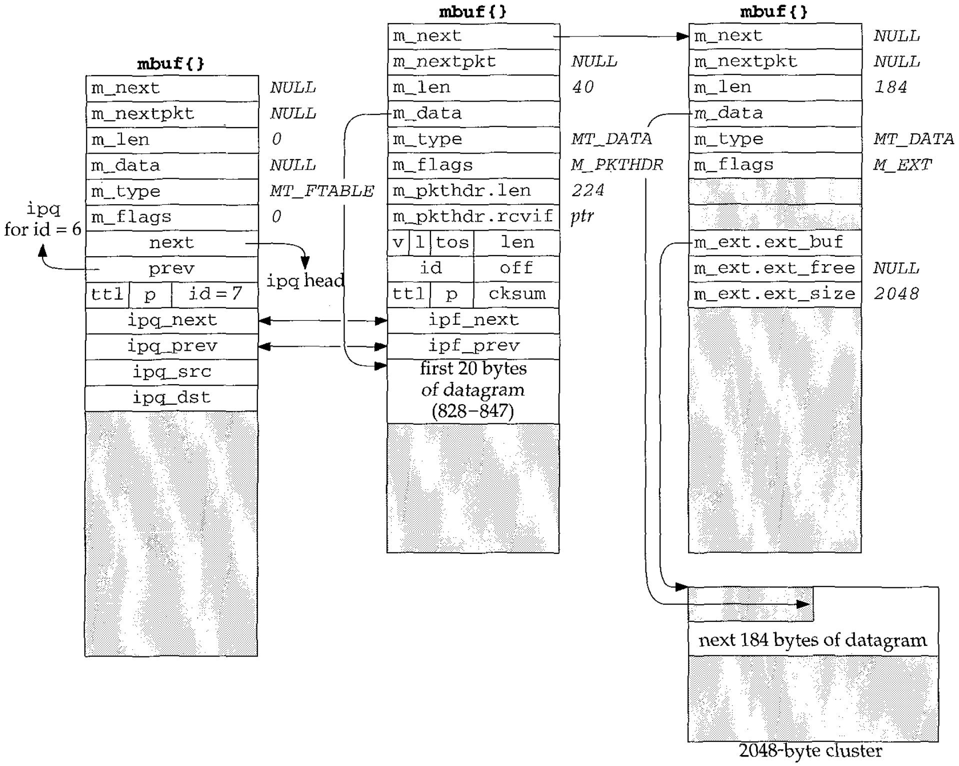
9.6 The source route option code may change `ip_dst` in the packet during processing. The destination is saved so that the timestamp processing code uses the original destination.

Chapter 10

10.2 After reassembly, only the options from the initial fragment are available to the transport protocols.

10.3 The fragment is read into a cluster since the data length (204 + 20) is greater than 208 (Figure 2.16).

`m_pullup` in Figure 10.11 moves the first 40 bytes into a separate mbuf as in Figure 2.18.



10.5 The average number of received fragments per datagram is

$$\frac{72,786 - 349}{16,557} = 4.4$$

The average number of fragments created for an outgoing datagram is

$$\frac{796,084}{260,484} = 3.1$$

10.6 In Figure 10.11 the packet is initially processed as a fragment. The reserved bit is discarded when `ip_off` is left shifted. The resulting packet is processed as a fragment or as a complete datagram, depending on the values of the MF and offset bits.

Chapter 11

11.1 The outgoing reply uses the source address of the interface on which the request was received. Hosts are not required to recognize 0.0.0.0 as a valid broadcast address, so the request may be ignored. The recommended broadcast address is 255.255.255.255.

11.2 Assume that a host sends link-level broadcasts packets with the IP source address of another host and the packet contains errors such as an improperly formed option. Every host receives and detects the error because of the link-level broadcast and because options are processed before a final destination check. Many hosts that detect the error try to send an ICMP message back to the IP source of the packet even though the original packet was sent as a link-level broadcast. The unfortunate host will begin receiving many bogus ICMP error messages. This is one reason why ICMP errors must not be sent in response to link-level broadcasts.

11.3 In the first case, such a redirect message can fool the host into sending packets to an arbitrary host on an alternate subnetwork. This host may be masquerading as a router but recording the traffic it receives instead. RFC 1009 requires that routers only generate redirect messages for other routers on the same subnet. Even if the host ignores these messages to redirect packets to a new subnetwork, a host on the same subnetwork can fool the host. The second case guards against this by requiring that the host only accept the redirect advice from the original router that it had (erroneously) selected to receive the traffic. Presumably this incorrect router was a default router specified by an administrator.

11.4 By passing the message to `rip_input`, a process-level daemon could respond and old systems that relied on this behavior could continue to be supported.

11.5 ICMP errors are sent only for the initial fragment of an IP datagram. Since the offset value of an initial fragment is always 0, the byte ordering of the field is unimportant.

11.6 If the ICMP request was received on an interface that was not yet configured with an IP address, `ia` would be null and no reply could be generated.

11.7 Net/3 reflects the data along with the timestamp reply.

11.10 The high-order bit is reserved and must be 0. If it is sent, `icmp_error` will discard the packet.

11.11 The return value is discarded because `icmp_send` does not return an error, but more significantly, errors generated during ICMP processing are discarded to avoid generating an endless series of error messages.

Chapter 12

12.1 On an Ethernet, the IP broadcast address 255.255.255.255 translates to the Ethernet broadcast address `ff:ff:ff:ff:ff:ff` and is received by *every* Ethernet interface on the network. Systems that aren't running IP software must actively receive and discard each of these broadcast packets.

A packet sent to the IP all-hosts multicast group 224.0.0.1 translates to the Ethernet multicast address `01:00:5e:00:00:01` and is received only by systems that have explicitly instructed their interfaces to receive IP multicast datagrams. Systems that aren't running IP or that aren't level-2 compliant never receive these datagrams, as they are discarded by the Ethernet interface hardware itself.

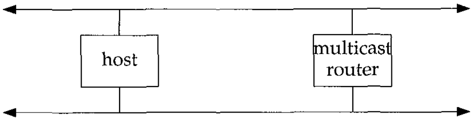
12.2 One alternative would be to specify interfaces by their text name as with the `ifreq` structure and the `ioctl` commands for accessing interface information. `ip_setmoptions` and `ip_getmoptions` would have to call `ifunit` instead of `INADDR_TO_IFP` to locate the pointer to the interface's `ifnet` structure.

12.3 The high-order 4 bits of a multicast group are always 1110, so only 5 significant bits are discarded by the mapping function.

12.4 The entire `ip_moptions` structure must fit within an mbuf, which limits the size of the structure to 108 bytes (remember the 20-byte mbuf header). `IP_MAX_MEMBERSHIPS` can be larger but must be less than or equal to 25. $(4+1+1+2+(4\times 25)=108)$

12.5 The datagram is duplicated and two copies appear on the IP input queue. A multicast application must be prepared to discard duplicate datagrams.

12.6



12.8 The process could create a second socket and request another `IP_MAX_MEMBERSHIPS` through the second socket.

12.9 Define a new mbuf flag `M_LOCAL` for the `m_flags` member of the mbuf header. The flag can be set on loopback packets by `ip_output` instead of computing the checksum. `ipintr` can skip the checksum verification if the flag is on. SunOS 5.X has an option to do this (`ip_local_cksum`, page 531, Volume 1).

12.10 There are $2^{23}-1$ (8,388,607) unique Ethernet IP multicast addresses. Remember that IP group 224.0.0.0 is reserved.

12.11 This assumption is correct since `in_addmulti` rejects all add requests if the interface does not have an `ioctl` function, and this implies that `in_delmulti` is never called if `if_ioctl` is null.

12.12 The mbuf is never released. It appears that `ip_getmoptions` contains a memory leak. `ip_getmoptions` is called from `ip_ctloutput`, which allows a call such as:

```
ip_getmoptions(IP_ADD_MEMBERSHIP, 0, mp)
```

which exercises the bug in `ip_getmoptions`.

Chapter 13

13.1 Responding to an IGMP query from the loopback interface is unnecessary since the local host is the only system on the loopback network and it already knows its membership status.

13.2 `max_linkhdr` + `sizeof(struct ip)` + `IGMP_MINLEN` = $16 + 20 + 8 = 44 < 100$

13.3 The primary reason for the random delay in reporting memberships is to minimize (ideally to 1) the number of reports that appear on a multicast network. A point-to-point network consists only of two interfaces, so the delay is not necessary to minimize the response to the query. One interface (presumably a multicast router) generates the query, and the other interface responds.

There is another reason not to flood the interface's output queue with all the membership reports. The output queue may have a packet or byte limit that could be exceeded by many IGMP membership reports. For example, in the SLIP driver, if the output queue is full or the device is too busy, the entire queue of pending packets is discarded (Figure 5.16).

Chapter 14

14.1 Five. One each for networks A through E.

14.2 `grplst_member` is called only by `ip_mforward`, but `ip_mforward` can be called by `ipintr` during protocol processing, or by `ip_output`, which can be called indirectly from the socket layer. The cache is a shared data structure that must be protected while it is being updated. The membership list itself is protected by `splx` calls in `add_lgrp` and `del_lgrp`, where it is modified.

14.3 The `SIOCDELMULTI` command affects only the Ethernet multicast list for the interface. The IP multicast group list remains unchanged, so the interface remains a member of the group. The interface continues accepting multicast datagrams for any groups that are still on the IP group membership list for the interface. Specifically, when `ether_delmulti` returns `ENETRESET` to `leioctl`, the function `lereset` is called to reconfigure the interface (Figure 12.31).

14.4 Only one virtual interface is considered to be the parent interface for a multicast spanning tree. If the packet is accepted on the tunnel, then the physical interface cannot be the parent and `ip_mforward` discards the packet.

Chapter 15

15.1 The socket could be shared across a fork or passed to a process through a Unix domain socket ([Stevens 1990]).

15.2 The `sa_len` member of the structure is larger than the size of the buffer after `accept` returns. This is usually not a problem with the fixed-length Internet address, but it can be when using variable-length addresses supported by the OSI protocols, for example.

15.4 The call to `soqremque` is only made when `so_qlen` is not equal to 0. If `soqremque` returns a null pointer there must be an error in the socket queueing code so the kernel panics.

15.5 The copy is made so that `bzero` can clear the structure while it is locked and so that `dom_dispose` and `sbrelease` can be called after `splx`. This minimizes the amount of time the CPU is kept at `splimp` and therefore the amount of time that network interrupts are blocked.

15.6 The `sbspace` macro will return 0. As a result, the `sbappendaddr` and `sbappendcontrol` functions (used by UDP) will refuse to queue additional packets. TCP uses `sbappend`, which assumes that the caller has checked for space first. TCP calls `sbappend` even when `sbspace` returns 0. The data placed in the receive queue is not available to a process because the `SS_CANTRCVMORE` flag prevents the read system calls from returning any data.

Chapter 16

16.1 When the value is assigned to `uio_resid` in the `uio` structure it becomes a large negative number. `sosend` rejects the message with `EINVAL`.

> Net/2 did *not* check for a negative value. This problem is described by the comment at the start of `sosend` (Figure 16.23).

16.2 No. The only time the cluster is ever filled with less than `MCLBYTES` is at the end of a message when less than `MCLBYTES` remain. `resid` is 0 at this time and the loop is terminated by the `break` on line 394 before reaching the test for `space` > 0.

16.5 The process blocks until the buffer is unlocked. In this case the lock exists only while another process is examining the buffer or passing data to the protocol layer, and not when a process must wait for space in the buffer, which may take an indefinite amount of time.

16.6 If the send buffer contained many mbufs, each of which contained only a few bytes of data, `sb_cc` may be well below the limit specified by `sb_hiwat` while a large amount of memory would be allocated for the mbufs. If the kernel didn't limit the number of mbufs attached to each buffer, a process could easily create a memory shortage.

16.7 `recvit` is called from `recvfrom` and `recvmsg`. Only `recvmsg` handles control information. The entire `msghdr` structure, including the length of the control message, is copied back to the process by `recvmsg`. For address information, `recvmsg` sets the `namelenp` argument to null because it expects the length in `msg_namelen`. When `recvfrom` calls `recvit`, the `namelenp` is nonnull because it expects the length in `*namelenp`.

16.8 `MSG_EOR` is cleared by `soreceive` so that it is not inadvertently returned by `soreceive` before an `M_EOR` mbuf is processed.

16.9 There would be a race condition while `select` examined the descriptors. If a selectable event occurred after `selscan` examined the descriptor but before `select` called `tsleep`, it would not be detected and the process would sleep until another selectable event occurred.

Chapter 17

17.1 This simplifies the code that copies data between the kernel and the process. `copyin` and `copyout` can be used for a single mbuf, but `uiomove` is needed to handle multiple mbufs.

17.2 The code works correctly because the first member of a `linger` structure is the expected integer flag.

Chapter 18

18.1 Write eight rows, one for each possible combination of the bits from the search key, the routing table key, and the routing table mask.

row	1 search key	2 table key	3 table mask	1 & 3	2 == 4?	1 ^ 2	6 & 3
1	0	0	0	0	yes	0	0=yes
2	0	0	1	0	yes	0	0=yes
3	0	1	0	0	no	1	0=yes
4	0	1	1	0	no	1	1=no
5	1	0	0	0	yes	1	0=yes
6	1	0	1	1	no	1	1=no
7	1	1	0	0	no	0	0=yes
8	1	1	1	1	yes	0	0=yes

The column "2 == 4?" should equal the final column "6 & 3." On first glance they are not the same, but we can ignore rows 3 and 7 because in these two rows the routing table bit is 1 while the same bit in the routing table mask is 1. When the routing table is built the key is logically ANDed with the mask, guaranteeing that for every bit of 0 in the mask, the corresponding bit in the key is also 0.

Another way to look at the exclusive OR and logical AND in Figure 18.40 is that the exclusive OR becomes 1 only if the the search key bit differs from the bit in the routing table key. The logical AND then ignores any differences that correspond to a bit that's 0 in the mask. If the result is still nonzero, the search key does not match the routing table key.

18.2 The size of an `rtentry` structure is 120 bytes, which includes the two `radix_node` structures. Each entry also requires two `sockaddr_in` structures (Figure 18.28), for 152 bytes per routing table entry. The total is about 3 megabytes.

18.3 Since `rn_b` is a short integer, assuming 16 bits for a short imposes a limit of 32767 bits per key (4095 bytes).

Chapter 19

19.1 The `RTF_DYNAMIC` flag is set in Figure 19.15 when the route is created by a redirect, and the `RTF_MODIFIED` flag is set when the gateway field of an existing route is modified by a redirect. If a route is created by a redirect and then later modified by another redirect, both flags will be set.

19.2 A host route is created for each host accessed through the default route. TCP can then maintain and update routing metrics for each individual host (Figure 27.3).

19.3 Each `rt_msghdr` structure requires 76 bytes. Two `sockaddr_in` structures are present for a host route (destination and gateway) giving a message size of 108 bytes. The message size for each ARP entry is 112 bytes: one `sockaddr_in` and one `sockaddr_dl`. The total size is then $(15 \times 112 + 20 \times 108)$ or 3840 bytes. A network route (instead of a host route) requires an additional 8 bytes for the network mask (116 bytes for the message instead of 108), so if the 20 routes are all network routes, the total size is 4000 bytes.

Chapter 20

20.1 The return value is returned in the `rtm_errno` member of the message (Figure 20.14) and also as the return value from `write` (Figure 20.22). The latter is more reliable since the former may run into mbuf starvation, causing the reply message to be discarded (Figure 20.17).

20.2 For a `SOCK_RAW` socket, the `pffindproto` function (Figure 7.20) returns the entry with a protocol of 0 (the wildcard) if an exact match isn't found.

Chapter 21

21.1 It is assumed that the `ifnet` structure is at the beginning of the `arpcom` structure, which it is (Figure 3.20).

21.2 Sending the ICMP echo request does not require ARP, since the destination address is the broadcast address. But the ICMP echo replies are normally unicast, so each sender uses ARP to determine the destination Ethernet address. When the local host receives each ARP request, `in_arpinput` replies and creates an entry for the other host.

21.3 When a new ARP entry is created, the `rt_gateway` value, a `sockaddr_dl` structure in this case, is copied from the entry being cloned by `rtrequest` in Figure 19.8. In Figure 21.1 we see that the `sdl_alen` member of this entry is 0.

21.4 With Net/3, if the caller of `arpresolve` supplies a pointer to a routing table entry, `arplookup` is not called, and the corresponding Ethernet address is available through the `rt_gateway` pointer (assuming it hasn't expired). This avoids any type of lookup in the common case. In Chapter 22 we'll see that TCP and UDP store a pointer to their routing table entry in their protocol control block, avoiding a search of the routing table in the case of TCP (where the destination IP address never changes for a connection) and in the case of UDP when the destination doesn't change.

21.5 The timeout of an incomplete ARP entry occurs between 0 and 5 minutes after the entry is created. `arpresolve` sets `rt_expire` to the current time when the ARP request is sent. The next time `arptimer` runs, if that entry is not resolved, it is deleted (assuming its reference count is 0).

21.6 `ether_output` returns `EHOSTUNREACH` instead of `EHOSTDOWN`, causing an ICMP host unreachable error to be sent to the sending host by `ip_forward`.

21.7 The value for 140.252.13.32 is set in Figure 21.28 to the current time when the entry is created. It never changes.

The values for 140.252.13.33 and 140.252.13.34 are copied from the entry for 140.252.13.32 when these two entries are cloned by `rtrequest`. They are then set to the time at which an ARP request is sent by `arpresolve`, and finally set by `in_arpinput` to the time at which an ARP reply is received, plus 20 minutes.

The value for 140.252.13.35 is also copied from the entry for 140.252.13.32 when the entry is cloned, but then set to 0 by the code at the end of Figure 21.29.

21.8 Change the call to `arplookup` at the beginning of Figure 21.19 to always specify a second argument of 1 (the create flag).

21.9 The first datagram was sent *after* the halfway mark to the next second. Therefore both the first and second datagrams caused ARP requests to be sent, about 500 ms apart, since the kernel's `time.tv_sec` variable had different values when these two datagrams were sent.

21.10 Each packet to send is an mbuf chain. The `m_nextpkt` pointer in the first mbuf in each chain could be used to form a list of mbufs awaiting transmission.

Chapter 22

22.1 An infinite loop occurs, waiting for a port to become available. This assumes the process is allowed to open enough descriptors to tie up all ephemeral ports.

22.2 Few, if any, servers support this option. [Cheswick and Bellovin 1994] mention how this would be nice for implementing firewall systems.

22.4 The `udb` structure is initialized to 0 so `udb.inp_lport` starts at 0. The first time through `in_pcbbind` it is incremented to 1, which is less than 1024, so it is set to 1024.

22.5 Normally the caller sets the address family (`sa_family`) to `AF_INET`, but we saw in Figure 22.20 that the test for this is commented out. The caller can set the length member (`sa_len`), but we saw in Figure 15.20 that the function `sockargs` always sets this to the third argument to `bind`, which for a `sockaddr_in` structure is specified as 16, normally using C's `sizeof` operator.

The local IP address (`sin_addr`) can be specified as a wildcard address or as a local IP address. The local port number (`sin_port`), can be either 0 (telling the kernel to choose an ephemeral port) or nonzero if the process wants a particular port. Normally a TCP or UDP server specifies a wildcard IP address and a nonzero port, and a UDP client often specifies a wildcard IP address and a port number of 0.

22.6 A process is allowed to `bind` a local broadcast address, because the call to `ifa_ifwithaddr` in Figure 22.22 succeeds. That address is used as the source address for IP datagrams sent on the socket. As noted in Section C.2, this behavior is not allowed by RFC 1122.

An attempt to `bind` 255.255.255.255, however, fails, since that address is not acceptable to `ifa_ifwithaddr`.

Chapter 23

23.1 `sosend` places the user data into a single mbuf if the size is less than or equal to 100 bytes; into two mbufs if the size is less than or equal to 207 bytes; or into one or more mbufs, each with a cluster, otherwise. Furthermore, `sosend` calls `MH_ALIGN` if the size is less than 100 bytes, which, it is hoped, will allow room at the beginning of the mbuf for the protocol headers. Since `udp_output` calls `M_PREPEND`, the following five scenarios are possible: (1) If the size of the user data is less than or equal to 72 bytes, a single mbuf contains the IP header, UDP header, and data. (2) If the size is between 73 and 100 bytes, one mbuf is allocated by `sosend` for the data and another is allocated by `M_PREPEND` for the IP and UDP headers. (3) If the size is between 101 and 207 bytes, two mbufs are allocated by `sosend` for the data and another by `M_PREPEND` for the IP and UDP headers. (4) If the size is between 208 and `MCLBYTES`, one mbuf with a cluster is allocated by `sosend` for the data and another by `M_PREPEND` for the IP and UDP headers. (5) Beyond this size, `sosend` allocates as many mbufs with clusters as necessary to hold the data (up to 64 for a maximum data size of 65507 bytes with 1024-byte clusters), and one mbuf is allocated by `M_PREPEND` for the IP and UDP headers.

23.2 IP options are passed to `ip_output`, which calls `ip_insertoptions` to insert the options into the outgoing IP datagram. This function in turn allocates a new mbuf to hold the IP header including options if the first mbuf in the chain points to a cluster (which never happens with UDP output) or if there is not enough room at the beginning of the first mbuf in the chain for the options. In scenario 1 from the previous solution, the size of the options determines whether another mbuf is allocated by `ip_insertoptions`: if the size of the user data is less than $100 - 28 - optlen$, (where *optlen* is the number of bytes of IP options), there is room in the mbuf for the IP header with options, the UDP header, and the data.

In scenarios 2, 3, 4, and 5, the first mbuf in the chain is always allocated by `M_PREPEND` just for the IP and UDP headers. `M_PREPEND` calls `m_prepend`, which calls `MH_ALIGN`, moving the 28 bytes of headers to the end of the mbuf, hence there is always room for the maximum of 40 bytes of IP options in this first mbuf in the chain.

23.3 No. The function `in_pcbconnect` is called, either when the application calls `connect` or when the first datagram is sent on an unconnected UDP socket. Since the local address is a wildcard and the local port is 0, `in_pcbconnect` sets the local port to an ephemeral port (by calling `in_pcbbind`) and sets the local address based on the route to the destination.

23.4 The processor priority level is left at `splnet`; it is not restored to the saved value. This is a bug.

23.5 No. `in_pcbconnect` will not allow a connection to port 0. Even if the process doesn't call `connect` directly, an implicit `connect` is performed, so `in_pcbconnect` is called regardless.

23.6 The application must call `ioctl` with the `SIOCGIFCONF` command to return information on all configured IP interfaces. The destination address in the received UDP datagram must then be compared against all the IP addresses and broadcast addresses in the list returned by `ioctl`. (As an alternative to `ioctl`, the `sysctl` system call described in Section 19.14 can also be used to obtain the information on all the configured interfaces.)

23.7 `recvit` releases the mbuf with the control information.

23.8 To disconnect a connected UDP socket, call `connect` with an invalid address, such as 0.0.0.0, and a port of 0. Since the socket is already connected, `soconnect` calls `sodisconnect`, which calls `udp_usrreq` with a `PRU_DISCONNECT` request. This sets the foreign address to 0.0.0.0 and the foreign port to 0, allowing a subsequent call to `sendto` that specifies a destination address to succeed. Specifying the invalid address causes the `PRU_CONNECT` request from `sodisconnect` to fail. We don't want the `connect` to succeed, we just want the `PRU_DISCONNECT` request executed and this back door through `connect` is the only way to execute this request, since the sockets API doesn't provide a `disconnect` function.

The manual page for `connect`(2) usually contains the following note that hints at this: "Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address." What this note fails to mention is that the call to `connect` for the invalid address is expected to return an error. The term *null address* is also vague: it means the IP address 0.0.0.0, not a null pointer for the second argument to `bind`.

23.9 Since an unconnected UDP socket is temporarily connected to the foreign IP address by `in_pcbconnect`, the scenario is the same as if the process calls `connect`: the datagram is sent out the primary interface with a destination IP address corresponding to the broadcast address of that interface.

23.10 The server must set the `IP_RECVDSTADDR` socket option and use `recvmsg` to obtain the destination IP address from the client's request. For this address to be the source IP address of the reply requires that this IP address be bound to the socket. Since you cannot `bind` a socket more than once, the server must create a brand new socket for each reply.

23.11 Notice in `ip_output` (Figure 8.22) that IP does not modify the DF bit supplied by the caller. A new socket option could be defined to cause `udp_output` to set the DF bit before passing datagrams to IP.

23.12 No. It is used only in the `udp_input` function and should be local to that function.

Chapter 24

24.1 The total number of ESTABLISHED connections is 126,820. Dividing this into the total number of bytes transmitted and received yields an average of about 30,000 bytes in each direction.

24.2 In `tcp_output`, the mbuf obtained for the IP and TCP headers also contains room for the link-layer headers (`max_linkhdr`). The IP and TCP header prototype is copied into the mbuf using `bcopy`, which won't work if the 40-byte header were split between two mbufs. Although the 40-byte headers must fit into one mbuf, the link-layer header need not. But a performance penalty would occur later (`ether_output`) because a separate mbuf would be required for the link-layer header.

24.3 On the author's system `bsdi`, the count was 16, 15 of which were standard system daemons (Telnet, Rlogin, FTP, etc.). On `vangogh.cs.berkeley.edu`, a medium-sized multiuser system with around 20 users, the count was 60. On a large multiuser system (`world.std.com`) with around 150 users, the count was 417 TCP end points and 809 UDP end points.

Chapter 25

25.1 In Figure 24.5 there were 531,285 delayed ACKs over 2,592,000 seconds (30 days). This is an average of about one delayed ACK every 5 seconds, or one delayed ACK every 25 times `tcp_fasttimo` is called. This means 96% of the time (24 times out of every 25) *every* TCP control block is checked for the delayed-ACK flag, when not one is set. On the large multiuser system in the solution to Exercise 24.3, this involves looking at over 400 control blocks, 5 times a second.

One alternative implementation would be to set a global flag when a delayed ACK is needed and only go through the list of control blocks when the flag is set. Alternatively, another list could be maintained that contains only the control blocks that require a delayed ACK. See, for example, the variable `igmp_timers_are_running` in Figure 13.14.

25.2 This allows the variable `tcp_keepintvl` to be patched in the running kernel, which then changes the value of `tcp_maxidle` the next time `tcp_slowtimo` is called.

25.3 `t_idle` actually counts the time since a segment was last received or transmitted. This is because TCP output must be acknowledged by the other end and the receipt of the ACK clears `t_idle`, as does the receipt of a data segment (Figure 28.8).

25.4 Here is one way to rewrite the code:

```
case TCPT_2MSL:
    if (tp->t_state == TCPS_TIME_WAIT)
        tp = tcp_close(tp);
    else {
        if (tp->t_idle <= tcp_maxidle)
            tp->t_timer[TCPT_2MSL] = tcp_keepintvl;
        else
            tp = tcp_close(tp);
    }
    break;
```

25.5 When the duplicate ACK is received, `t_idle` is 150, but it is reset to 0. When the FIN_WAIT_2 timer expires, `t_idle` will be 1048 (1198 − 150), so the timer is set to 150 ticks. When the timer expires the next time, `t_idle` will be 1198, so the timer is set to 150 ticks. When the timer expires the next time, `t_idle` will be 1198 + 150, so the connection is closed. The duplicate ACK extends the time until the connection is closed.

25.6 The first keepalive probe will be sent 1 hour in the future. When the process sets the option, nothing happens other than setting the `SO_KEEPALIVE` option in the `socket` structure. When the timer expires 1 hour in the future, since the option is enabled, the code in Figure 25.16 sends the first probe.

25.7 The value of `tcp_rttdflt` initializes the RTT estimators for every TCP connection. A site can change the default of 3, if desired, by patching the global variable. If the value were a `#define` constant, it could be changed only by recompiling the kernel.

Chapter 26

26.1 The counter `t_idle` is always running for a connection, whereas TCP does not measure the amount of time since the last segment was sent on a connection.

26.2 In Figure 25.26 `snd_nxt` is set to `snd_una`, giving a value of 0 for `len`.

26.3 If you're running a Net/3 system and encounter a peer that can't handle either of these two newer options (i.e., that peer refuses to establish the connection, even though a host is required to ignore options it doesn't understand), this global can be patched in the kernel to disable one or both of these options.

26.4 The timestamp option would have updated the RTT estimators each time an ACK was received for new data: 16 times, twice the number of times without the option. The value calculated when the ACK of 6145 was received at time 217.944, however, would have been bogus—either the data segment with bytes

5633 through 6144 that was sent at time 3.740, or the received ACK of 6145, was delayed somewhere for about 200 seconds.

26.5 There is no guarantee that the 2-byte MSS value is correctly aligned for such a memory reference.

26.6 (This solution is from Dave Borman.) The maximum amount of TCP data in a segment is 65495 bytes, which is 65535 minus the minimum IP and TCP headers (40). Hence there are 39 values of the urgent offset that make no sense: 65496 through and including 65535. Whenever the sender has a 32-bit urgent offset that exceeds 65495, 65535 is sent as the urgent offset instead, and the URG flag is set. This puts the receiver into urgent mode and tells the receiver that the urgent offset points to data that has not been sent yet. The special value of 65535 continues to be sent as the urgent offset (with the URG flag set) until the urgent offset is less than or equal to 65495, at which point the real urgent offset is sent.

26.7 We've mentioned that data segments are transmitted reliably (i.e., the retransmission timer is set) but ACKs are not. RST segments are not transmitted reliably either. RST segments are generated when a bogus segment arrives (either a segment that is wrong for a connection, or a segment for a nonexistent connection). If the RST segment is discarded by `ip_output`, when the other end retransmits the segment that caused the RST to be generated, another RST will be generated.

26.8 The application does eight writes of 1024 bytes. The first four times `sosend` is called, `tcp_output` is called, and a segment is sent. Since these four segments each contain the final bytes of data in the send buffer, the PSH flag is set for each segment (Figure 26.25). The send buffer is also full, so the next write by the process puts the process to sleep in `sosend`. When the ACK is returned with an advertised window of 0, the 4096 bytes of data in the send buffer have been acknowledged and are discarded, and the process wakes up and continues filling the send buffer with the next four writes. But nothing can be sent until a nonzero window is advertised by the receiver. When this happens, the next four segments are sent, but only the final segment contains the PSH flag, since the first three segments do not empty the send buffer.

26.9 The `tp` argument to `tcp_respond` can be a null pointer if the segment being sent does not correspond to a connection. The code should check the value of `tp` and use the default only if the pointer is null.

26.10 `tcp_output` always allocates an mbuf just to contain the IP and TCP headers, by calling `MGETHDR` in Figures 26.25 and 26.26. This code allocates room at the front of the new mbuf only for the link-layer header (`max_linkhdr`). If IP options are in use and the size of the options exceeds `max_linkhdr`, another mbuf is allocated by `ip_insertoptions`. If the size of the IP options is less than or equal to `max_linkhdr`, then even though `ip_insertoptions` will use the space at the beginning of the mbuf, this will cause `ether_output` to allocate another mbuf for the link-layer header (assuming Ethernet output).

To try to avoid the extra mbuf, Figures 26.25 and 26.26 could call `MH_ALIGN` if the segment will contain IP options.

26.11 About 80 lines of C code, assuming RFC 1323 timestamps are in use and the segment is timed.

The macro `MGETHDR` invokes the macro `MALLOC`, which might call the function `malloc`. The function `m_copy` is also called, but a full-sized segment will be in a cluster, so the mbuf is not copied, a reference is made to the cluster. The call to `MGET` by `m_copy` might call `malloc`. The function `bcopy` copies the header template and `in_cksum` calculates the TCP checksum.

26.12 Nothing changes with `writev` because of the logic in `sosend`. Since the total size of the data (150) is less than `MINCLSIZE` (208), one mbuf is allocated for the first 100 bytes, and since the protocol is not atomic, the `PRU_SEND` request is issued. Another mbuf is allocated for the next 50 bytes, and another `PRU_SEND` is issued. TCP still generates two segments. (`writev` only generates a single "record," that is, a single `PRU_SEND` request, for `PR_ATOMIC` protocols such as UDP.)

With two buffers of length 200 and 300 the total size now exceeds `MINCLSIZE`. An mbuf cluster is allocated and only one `PRU_SEND` is issued. One 500-byte segment is generated by TCP.

Chapter 27

27.1 The first six rows of the table are asynchronous errors that are generated by the receipt of a segment or the expiration of a timer. By storing the nonzero error code in `so_error`, the process receives the error on the next read or write. The call from `tcp_disconnect`, however, occurs when the process calls `close`, or when the descriptor is closed automatically on process termination. In either case of the descriptor being closed, the process won't issue a read or write call to fetch the error. Also, since the process had to set the socket option explicitly to force the RST, returning an error provides no useful information to the process.

27.2 Assuming a 32-bit `u_long`, the maximum value is just under 4298 seconds (1.2 hours).

27.3 The statistics in the routing table are updated by `tcp_close` and it is called only when the connection enters the CLOSED state. Since the sending of data to the other end is terminated by the FTP client (it does the active close), the local end point enters the TIME_WAIT state. The routing table statistics won't be updated until twice the MSL has elapsed.

Chapter 28

28.1 0, 1, 2, and 3.

28.2 34.9 Mbits/sec. For higher speeds, larger buffers are required on both ends.

28.3 In the general case, `tcp_dooptions` doesn't know whether the two timestamp values are aligned on 32-bit boundaries or not. The special code in Figure 28.4, however, knows that the values are on 32-bit boundaries, and avoids calling `bcopy`.

28.4 The "options prediction" code in Figure 28.4 handles only the recommended format, so systems that send other than the recommended format cause the slower processing of `tcp_dooptions` to occur for every received segment.

28.5 If `tcp_template` were called every time a socket were created, instead of every time a connection is established, each listening server on a system would have one allocated, which it would never use.

28.6 The timestamp clock frequency should be between 1 bit/ms and 1 bit/sec. (Net/3 uses 2 bits/sec.) With the highest frequency of 1 bit/ms, a 32-bit timestamp wraps its sign bit in $2^{31}/(24 \times 60 \times 60 \times 1000)$ days, which is 24.8 days.

28.7 With a frequency of 1 bit per 500 ms, a 32-bit timestamp wraps its sign bit in $2^{31}/(24 \times 60 \times 60 \times 2)$ days, which is 12,427 days, or about 34 years, longer than the uptime of current computer systems.

28.8 The cleanup function of an RST should take precedence over timestamps, and it is recommended that RSTs not carry timestamps (which is enforced by `tcp_input` in Figure 26.24).

28.9 Since the client is in the ESTABLISHED state, processing ends up in Figure 28.24. `todrop` is 1 because `rcv_nxt` was incremented over the SYN when it was first received. The SYN flag is cleared (since it is a duplicate), `ti_seq` is incremented, and `todrop` is decremented to 0. The `if` statement at the top of Figure 28.25 is executed since `todrop` and `ti_len` are both 0. The next `if` statement is skipped, and processing continues with the call to `m_adj`. But `tcp_output` is not called in the continuation of `tcp_input` in the next chapter, therefore the client does not respond to the duplicate SYN/ACK. The server will time out and resend the SYN/ACK (recall the timer set in Figure 28.17 when a passive socket receives a SYN), which will also be ignored. This is another bug in the code in Figure 28.25 and this one is also fixed with the code shown in Figure 28.30.

28.10 The client's SYN arrives at the server and is delivered to the socket in the TIME_WAIT state. The code in Figure 28.24 turns off the SYN flag and the code in Figure 28.25 jumps to `dropafterack`, dropping the segment but generating an ACK with an acknowledgment field of `rcv_nxt` (Figure 26.27). This is called a *resynchronization ACK* because its purpose is to tell the other end what sequence number it expects. When this ACK is received at the client (which is in the SYN_SENT state), its acknowledgment field is not the expected value (Figure 28.18), causing an RST to be sent. The sequence number of the RST is the acknowledgment field from the resynchronization ACK, and the ACK flag of the RST segment is off (Figure 29.28). When the server receives the RST, its TIME_WAIT state is prematurely terminated and the socket is closed on the server's host (Figure 28.36). The client times out after 6 seconds and retransmits its SYN. Assuming a listening server process is running on the server host, the new connection is established. Because of this form of TIME_WAIT assassination, a new connection is established not only when a SYN arrives with a higher sequence number (as checked for in Figure 28.29), but also when a SYN with a lower sequence number arrives.

Chapter 29

29.1 Assume a 2-second RTT. The server has a passive open pending and the client issues its active open at time 0. The server receives the SYN at time 1 and responds with its own SYN and an ACK of the client's SYN. The client receives this segment at time 2, and the code in Figure 28.20 completes the active open with the call to `soisconnected` (waking up the client process) and an ACK will be sent back to the server. The server receives the ACK at time 3, and the code in Figure 29.2 completes the server's passive open, returning control to the server process. In general, the client process receives control about one-half RTT before the server.

29.2 Assume the sequence number of the SYN is 1000 and the 50 bytes of data are numbered 1001–1050. When the SYN is processed by `tcp_input`, first the `case` starting in Figure 28.15 is executed, which sets `rcv_nxt` to 1001, and then a jump is made to `step6`. Figure 29.22 calls `tcp_reass` and the data is placed onto the socket's reassembly queue. But the data cannot be appended to the socket's receive buffer yet (Figure 27.23) so `rcv_nxt` is left at 1001. When `tcp_output` is called to generate the immediate ACK, `rcv_nxt` (1001) is sent as the acknowledgment field. In summary, the SYN is acknowledged, but not the 50 bytes of data. Since the client will retransmit the 50 bytes of data, there is no advantage in sending data with a SYN generated by an active open.

29.3 The server's socket is in the SYN_RCVD state when the client's ACK/FIN arrives, so `tcp_input` ends up processing the ACK in Figure 29.2. The connection moves to the ESTABLISHED state and `tcp_reass` appends the already-queued data to the socket's receive buffer. `rcv_nxt` is incremented to 1051. `tcp_input` continues and the FIN is handled in Figure 29.24 where the `TF_ACKNOW` flag is set and `rcv_nxt` becomes 1052. `socantrcvmore` sets the socket's state so that after the server reads the 50 bytes of data, the server will receive an end-of-file. The server's socket also moves to the CLOSE_WAIT state. `tcp_output` will be called to ACK the client's FIN (since `rcv_nxt` equals 1052). Assuming the server process closes its socket when it reads the end-of-file, the server will then send a FIN for the client to ACK.

In this example six segments requiring three round trips are required to pass the 50 bytes from the client to server. To reduce the number of segments requires the TCP extensions for transactions [Braden 1994].

29.4 The client's socket is in the SYN_SENT state when the server's response is received. Figure 28.20 processes the segment and moves the connection to the ESTABLISHED state. A jump is made to `step6` and the data is processed in Figure 29.22. `TCP_REASS` appends the data to the socket's receive buffer and `rcv_nxt` is incremented to acknowledge the data. The FIN is then processed in Figure 29.24, incrementing `rcv_nxt` again and moving the connection to the CLOSE_WAIT state. When `tcp_output` is called, the acknowledgment field ACKs the SYN, the 50 bytes of data, and the FIN. The client process then reads the 50 bytes of data, followed by the end-of-file, and then probably closes its socket. This moves the connection to the LAST_ACK state and causes a FIN to be sent by the client, which the server should acknowledge.

29.5 The bug is in the entry `tcp_outflags[TCPS_CLOSING]` shown in Figure 24.16. It specifies the `TH_FIN` flag, whereas the state transition diagram (Figure 24.15) doesn't specify that the FIN should be retransmitted. To fix this, remove `TH_FIN` from the `tcp_outflags` entry for this state. The bug is relatively harmless—it just causes two extra segments to be exchanged—and a simultaneous close or a close following a self-connect is rare.

29.6 No. An OK return from a write system call only means the data has been copied into the socket buffer. Net/3 does not notify the process when that data is acknowledged by the other end. An application-level acknowledgment is required to obtain this information.

29.7 RFC 1323 timestamps defeat header compression because whenever the timestamps change, the TCP options change, and the segment is sent uncompressed. The window scale option has no effect because the value in the TCP header is still a 16-bit value.

29.8 IP assigns the ID field from a global variable that is incremented each time *any* IP datagram is sent. This increases the probability that two consecutive TCP segments sent on the same connection will have ID values that differ by more than 1. A difference other than 1 causes the Δ*ipid* field in Figure 29.34 to be transmitted, increasing the size of the compressed header. A better scheme would be for TCP to maintain its own counter for assigning IDs.

Chapter 30

30.2 Yes, the RST is still sent. Part of process termination is the closing of all open descriptors. The same function (`soclose`) is eventually called, regardless of whether the process explicitly closes the socket descriptor or implicitly closes it (by terminating first).

30.3 No. The only use of this constant is when a listening socket sets the `SO_LINGER` socket option with a linger time of 0. Normally this causes an RST to be sent when the connection is closed (Figure 30.12), but Figure 30.2 changes this value of 0 to 120 (clock ticks) for a listening socket that receives a connection request.

30.4 Two if this is the first use of the default route; otherwise one. When the socket is created the Internet PCB is set to 0 by `in_pcballoc`. This sets the `route` structure in the PCB to 0. When the first segment is sent (the SYN), `tcp_output` calls `ip_output`. Since the `ro_rt` pointer is null, `ro_dst` is filled in with the destination address of the IP datagram and `rtalloc` is called. The pointer to the default route is saved in the `ro_rt` member of the `route` structure within the PCB for this connection. When `ether_output` is called by `ip_output`, it checks whether the `rt_gwroute` member of the routing table entry is null, and, if so, `rtalloc1` is called. Assuming the route doesn't change, each time `tcp_output` is called for this connection, the cached `ro_rt` pointer is used, avoiding any additional routing table lookups.

Chapter 31

31.1 Because `catchpacket` will always run to completion before any sleeping processes are awakened by the `bpf_wakeup` call.

31.2 A process that opens a BPF device may call `fork` resulting in multiple processes with access to the same BPF device.

31.3 Only supported devices are on the BPF interface list (`bpf_iflist`), so `bpf_setif` returns `ENXIO` when the interface is not found.

Chapter 32

32.1 0 in the first example, and 255 in the second. Both of these values are reserved in RFC 1700 [Reynolds and Postel 1994] and should not appear in datagrams. This means, for example, that a socket created with a protocol of `IPPROTO_RAW` should always have the `IP_HDRINCL` socket option set, and datagrams written to the socket should have a valid protocol value.

32.2 Since the IP protocol value of 255 is reserved, datagrams should never appear on the wire with this protocol value. Since this is a nonzero protocol value, the first of the three tests in `rip_input` will ignore every received datagram that does not have this protocol value. Therefore the process should not receive any datagrams on the socket.

32.3 Even though this protocol value is reserved and datagrams should never appear on the wire with this value, the first of the three tests in `rip_input` allows datagrams with any protocol value to be received by sockets of this type. The only input filtering that occurs for this type of raw socket is based on the source and destination IP addresses, if the process calls either `connect` or `bind`, or both.

32.4 Since the array `ip_protox` array (Figure 7.22) contains information about which protocol the kernel supports, the ICMP error should be generated only when there are no raw listeners for the protocol and the pointer `inetsw[ip_protox[ip->ip_p]].pr_input` equals `rip_input`.

32.5 In both cases the process must build its own IP header, in addition to whatever follows the IP header (UDP datagram, TCP segment, or whatever). With a raw IP socket, output is normally done using `sendto` specifying the destination address as an Internet socket address structure containing an IP address. `ip_output` is called and normal IP routing is done based on the destination IP address.

BPF requires the process to supply a complete data-link header, such as an Ethernet header. Output is normally done by calling `write`, since a destination address cannot be specified. The packet is passed directly to the interface output function, bypassing `ip_output` (Figure 31.20). The process selects the outgoing interface using the `BIOCSETIF ioctl` (Figure 31.16). Since IP routing is not performed, the destination of the packet is limited to another system on an attached network (unless the process duplicates the IP routing function and sends the packet to a router on an attached network, for the router to forward based on the destination IP address).

32.6 A raw IP socket receives only IP datagrams destined for an IP protocol that the kernel does not process itself. A process cannot receive TCP segments or UDP datagrams on a raw socket, for example.

BPF can receive *all* frames received on a specified interface, regardless of whether they are IP datagrams or not. The `BIOCPROMISC ioctl` can put the interface into a promiscuous mode, to receive datagrams that are not even destined for this host.

Appendix B

Source Code Availability

URLs: Uniform Resource Locators

This text uses URLs to specify the location and method of access of resources on the Internet. For example, the common "anonymous FTP" technique is designated as

```
ftp://ftp.cdrom.com/pub/bsd-sources/4.4BSD-Lite.tar.gz
```

This specifies anonymous FTP to the host `ftp.cdrom.com`. The filename is `4.4BSD-Lite.tar.gz` in the directory `pub/bsd-sources`. The suffix `.tar` implies the standard Unix `tar`(1) format, and the additional `.gz` suffix implies that the file has been compressed with the GNU `gzip`(1) program.

4.4BSD-Lite

There are numerous ways to obtain the 4.4BSD-Lite release. The entire 4.4BSD-Lite release is available from Walnut Creek CD-ROM as

```
ftp://ftp.cdrom.com/pub/bsd-sources/4.4BSD-Lite.tar.gz
```

You can also obtain this release on CD-ROM. Contact 1 800 786 9907 or +1 510 674 0783.

O'Reilly & Associates publishes the entire set of 4.4BSD manuals along with the 4.4BSD-Lite release on CD-ROM. Contact 1 800 889 8969 or +1 707 829 0515.

Operating Systems that Run the 4.4BSD-Lite Networking Software

The 4.4BSD-Lite release is *not* a complete operating system. To experiment with the networking software described in this text you need an operating system that is built from the 4.4BSD-Lite release or an environment that supports the 4.4BSD-Lite networking code.

The operating system used by the authors is commercially available from Berkeley Software Design, Inc. Contact 1 800 ITS BSD8, +1 719 260 8114, or `info@bsdi.com` for additional information.

There are also freely available operating systems built on 4.4BSD-Lite. These are known by the names NetBSD, 386BSD, and FreeBSD. Additional information is available from Walnut Creek CD-ROM (`ftp.cdrom.com`) or on the various `comp.os.386bsd` Usenet newsgroups.

RFCs

All RFCs are available at no charge through electronic mail or by using anonymous FTP across the Internet. Sending electronic mail as shown here:

```
To: rfc-info@ISI.EDU
Subject: getting rfcs

help: ways_to_get_rfcs
```

returns a detailed listing of various ways to obtain the RFCs using either email or anonymous FTP.

Remember that the starting place is to obtain the current index and look up the RFC that you want in the index. This entry tells you if that RFC has been made obsolete or updated by a newer RFC.

GNU Software

The GNU Indent program was used to format all the source code presented in the text, and the GNU Gzip program is often used on the Internet to compress files. These programs are available as

```
ftp://prep.ai.mit.edu/pub/gnu/indent-1.9.1.tar.gz
ftp://prep.ai.mit.edu/pub/gnu/gzip-1.2.2.tar
```

The numbers in the filenames will change as newer versions are released. There are also versions of the Gzip program for other operating systems, such as MS-DOS.

There are many sites around the world that also provide the GNU archives, and the FTP greeting on `prep.ai.mit.edu` displays their names.

PPP Software

There are several freely available implementations of PPP. Part 5 of the `comp.protocols.ppp` FAQ is a good place to start:

```
http://cs.uni-bonn.de/ppp/part5.html
```

`mrouted` Software

Current releases of the `mrouted` software as well as other multicast applications can be found at the Xerox Palo Alto Research Center:

```
ftp://parcftp.xerox.com/pub/net-research/
```

ISODE Software

An SNMP agent implementation compatible with Net/3 is part of the ISODE software package. For more information, start with the ISODE Consortium's World Wide Web page at

```
http://www.isode.com/
```

Appendix C

RFC 1122 Compliance

This appendix summarizes the compliance of the Net/3 implementation with RFC 1122 [Braden 1989a]. This RFC summarizes these requirements in four categories

- link layer
- internet layer
- UDP
- TCP

We have chosen to present these requirements in the same breakdown and order as the chapters of this text.

C.1 Link-Layer Requirements

This section summarizes the link-layer requirements from Section 2.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

- *May* support trailer encapsulation.
  Partially: Net/3 does not send IP datagrams with trailer encapsulation but some Net/3 device drivers may be able to receive such datagrams. We have omitted all the trailer encapsulation code in this text. Interested readers are referred to RFC 893 and Section 11.8 of [Leffler et al. 1989] for additional details.

- *Must* not send trailers by default without negotiation.
  Not applicable: Net/2 would negotiate the use of trailers but Net/3 ignores requests to send trailers and does not request trailers itself.

- *Must* be able to send and receive RFC 894 Ethernet encapsulation.
  Yes: Net/3 supports RFC 894 Ethernet encapsulation.

- *Should* be able to receive RFC 1042 (IEEE 802) encapsulation.
  No: Net/3 processes packets received with 802.3 encapsulation but only for use with OSI protocols. IP packets that arrive with 802.3 encapsulation are discarded by `ether_input` (Figure 4.13).

- *May* send RFC 1042 encapsulation, in which case there must be a software configuration switch to select the encapsulation method and RFC 894 must be the default.
  No: Net/3 does not send IP packets in RFC 1042 encapsulation.

- *Must* report link-layer broadcasts to the IP layer.
  Yes: The link layer reports link-layer broadcasts by setting the `M_BCAST` flag (or the `M_MCAST` flag for multicasts) in the mbuf packet header.

- *Must* pass the IP TOS value to the link layer.
  Yes: The TOS value is not passed explicitly, but is part of the IP header available to the link layer.

C.2 IP Requirements

This section summarizes the IP requirements from Section 3.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

- *Must* implement IP and ICMP.
  Yes: `inetsw[0]` implements the IP protocol and `inetsw[4]` implements ICMP.

- *Must* handle remote multihoming in application layer.
  Yes: The kernel is unaware of communication to remote multihomed hosts and neither hinders nor supports such communication by an application.

- *May* support local multihoming.
  Yes: Net/3 supports multiple IP interfaces with the `ifnet` list and multiple addresses per IP interface with the `ifaddr` list for each `ifnet` structure.

- *Must* meet router specifications if forwarding datagrams.
  Partially: See Chapter 18 for a discussion of the router requirements.

- *Must* provide configuration switch for embedded router functionality. The switch must default to host operation.
  Yes: The `ipforwarding` variable defaults to false and controls the IP packet forwarding mechanism in Net/3.

- *Must not* enable routing based on number of interfaces.
  Yes: The `if_attach` function does not modify `ipforwarding` according to the number of interfaces configured at system initialization time.

- *Should* log discarded datagrams, including the contents of the datagram, and record the event in a statistics counter.
  Partially: Net/3 does not provide a mechanism for logging the contents of discarded datagrams but maintains a variety of statistics counters.

- *Must* silently discard datagrams that arrive with an IP version other than 4.
  Yes: `ipintr` implements this requirement.

- *Must* verify IP checksum and silently discard an invalid datagram.
  Yes: `ipintr` calls `ip_cksum` and implements this requirement.

- *Must* support subnet addressing (RFC 950).
  Yes: Every IP address has an associated subnet mask in the `in_ifaddr` structure.

- *Must* transmit packets with host's own IP address as the source address.
  Partially: When the transport layer sends an IP datagram with all-0 bits as the source address, IP inserts the IP address of the outgoing interface in its place. A process can bind one of the local IP broadcast addresses to the local socket, and IP will transmit it as an invalid source address.

- *Must* silently discard datagrams not destined for the host.
  Yes: If the system is not configured as a router, `ipintr` discards datagrams that arrive with a bad destination address (i.e., an unrecognized unicast, broadcast, or multicast address).

- *Must* silently discard datagrams with bad source address (nonunicast address).
  No: `ipintr` does not examine the source address of incoming datagrams before delivering the datagram to the transport protocols.

- *Must* support reassembly.
  Yes: `ip_reass` implements reassembly.

- *May* retain same ID field in identical datagrams.
  No: `ip_output` assigns a new ID to every outgoing datagram and does not allow the ID to be specified by the transport protocols. See Chapter 32.

- *Must* allow the transport layer to set TOS.
  Yes: `ip_output` accepts any TOS value set in the IP header by the transport protocols. The transport layer must default TOS to all 0s. The TOS value for a particular datagram or connection may be set by the application through the `IP_TOS` socket option.

- *Must* pass received TOS up to transport layer.
  Yes: Net/3 preserves the TOS field during input processing. The entire IP header is made available to the transport layer when IP calls the `pr_input` function for the receiving protocol. Unfortunately, the UDP and TCP transport layers ignore it.

- *Should not* use RFC 795 [Postel 1981d] link-layer mappings for TOS.
  Yes: Net/3 does not use these mappings.

- *Must not* send packet with TTL of 0.
  Partially: The IP layer (`ip_output`) in Net/3 does not check this requirement and relies on the transport layers not to construct an IP header with a TTL of 0. UDP, TCP, ICMP, and IGMP all select a nonzero TTL default value. The default value can be overridden by the `IP_TTL` option.

- *Must not* discard received packets with a TTL less than 2.
  Yes: If the system is the final destination of the packet, `ipintr` accepts it regardless of the TTL value. The TTL is examined only when the packet is being forwarded.

- *Must* allow transport layer to set TTL.
  Yes: The transport layer must set TTL before calling `ip_output`.

- *Must* enable configuration of a fixed TTL.
  Yes: The default TTL is specified by the global integer `ip_defttl`, which defaults to 64 (`IPDEFTTL`). Both UDP and TCP use this value unless the `IP_TTL` socket option has specified a different value for a particular socket. `ip_defttl` can be modified through the `IPCTL_DEFTTL` name for `sysctl`.

Multihoming

- *Should* select, as the source address for a reply, the specific address received as the destination address of the request.
  Yes: Responses generated by the kernel (ICMP reply messages) include the correct source address (Section C.5). Responses generated by the transport protocols are described in their respective chapters.

- *Must* allow application to choose local IP address.
  Yes: An application can bind a socket to a specific local IP address (Section 15.8).

- *May* silently discard datagrams addressed to an interface other than the one on which it is received.
  No: Net/3 implements the weak end system model and `ipintr` accepts such packets.

- *May* require packets to exit the system through the interface with an IP address that corresponds to the source address of the packet. This requirement pertains only to packets that are not source routed.

No: Net/3 allows packets to exit the system through any interface—another weak end system characteristic.

Broadcast

- *Must* not select an IP broadcast address as a source address.
  Partially: If an application explicitly selects a source address, the IP layer does not override the selection. Otherwise, IP selects as a source address the specific IP address associated with the outgoing interface.

- *Should* accept an all-0s or all-1s broadcast address.
  Yes: `ipintr` accepts packets sent to either address.

- *May* support a configurable option to send all 0s or all 1s as the broadcast address on an interface. If provided, the configurable broadcast address *must* default to all 1s.
  No: A process must explicitly send to either the all-0s (`INADDR_ANY`) or all-1s broadcast address (`INADDR_BROADCAST`). There is no configurable default.

- *Must* recognize all broadcast address formats.
  Yes: `ipintr` recognizes the limited (all-1s and all-0s) and the network-directed and subnet-directed broadcast addresses.

- *Must* use an IP broadcast or IP multicast destination address in a link-layer broadcast.
  Yes: `ip_output` enables the link-layer multicast or broadcast flags only when the destination is an IP multicast or broadcast address.

- *Should* silently discard link-layer broadcasts when the packet does not specify an IP broadcast address as its destination.
  No: There is no explicit test for the `M_BCAST` or `M_MCAST` flags on incoming packets in Net/3, but `ip_forward` will discard these packets before forwarding them.

- *Should* use limited broadcast address for connected networks.
  Partially: The decision to use the limited broadcast address (versus a subnet-directed or network-directed broadcast) is left to the application level by Net/3.

IP Interface

- *Must* allow transport layer to use all IP mechanisms (e.g., IP options, TTL, TOS).
  Yes: All the IP mechanisms are available to the transport layer in Net/3.

- *Must* pass interface identification up to transport layer.
  Yes: The `m_pkthdr.rcvif` member of each mbuf containing an incoming packet points to the `ifnet` structure of the interface that received the packet.

- *Must* pass all IP options to transport layer.
  Yes: The entire IP header, including options, is present in the packet passed to the `pr_input` function of the receiving transport protocol by `ipintr`.

- *Must* allow transport layer to send ICMP port unreachable and any of the ICMP query messages.
  Yes: The transport layer may send any ICMP error messages by calling `icmp_error` or may format and send any type of IP datagram by calling the `ip_output` function.

- *Must* pass the following ICMP messages to the transport layer: destination unreachable, source quench, echo reply, timestamp reply, and time exceeded.
  Yes: These messages are distributed by ICMP to other transport protocols or to any waiting processes using the raw IP socket mechanism.

- *Must* include contents of ICMP message (IP header plus the data bytes present) in ICMP message passed to the transport layer.
  Yes: `icmp_input` passes the portion of the original IP packet contained within the ICMP message to the transport layers.

- *Should* be able to leap tall buildings at a single bound.
  No: The next version of IP may meet this requirement.

C.3 IP Options Requirements

This section summarizes the IP option processing requirements from Section 3.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

- *Must* allow transport layer to send IP options.
  Yes: The second argument to `ip_output` is a list of IP options to include in the outgoing IP datagram.

- *Must* pass all IP options received to higher layer.
  Yes: The IP header and options are passed to the `pr_input` function of the receiving transport protocol.

- *Must* silently ignore unknown options.
  Yes: The `default` case in `ip_dooptions` skips over unknown options.

- *May* support the security option.
  No: Net/3 does not support the IP security option.

- *Should not* send the stream identifier option and *must ignore* it in received datagrams.
  Yes: Net/3 does not support the stream identifier option and ignores it on incoming datagrams.

- *May* support the record route option.
  Yes: Net/3 supports the record route option.

- *May* support the timestamp option.
  Partially: Net/3 supports the timestamp option but does not implement it exactly as specified. The originating host does not insert a timestamp when required but the destination host records a timestamp before passing the datagram to the transport layer. The timestamp value follows the rules regarding standard values as specified in Section 3.2.2.8 of RFC 1122 for the ICMP timestamp message.

- *Must* support originating a source route and *must* be able to act as the final destination of a source route.
  Yes: A source route may be included in the options passed to `ip_output`, and `ip_dooptions` correctly terminates a source route and saves it for use in constructing return routes.

- *Must* pass a datagram with completed source route up to the transport layer.
  Yes: The source route option is passed up with any other options that may have appeared in the datagram.

- *Must* build correct (nonredundant) return route.
  No: Net/3 blindly reverses the source route and does not check or correct for a route that was built incorrectly with a redundant hop for the original source host.

- *Must* not send multiple source route options in one header.
  No: The IP layer in Net/3 does not prohibit a transport protocol from constructing and sending multiple source route options in a single datagram.

Source Route Forwarding

- *May* support packet forwarding with the source route option.
  Yes: Net/3 supports the source route options. `ip_dooptions` does all the work.

- *Must* obey corresponding router rules while processing source routes.
  Yes: Net/3 follows the router rules whether or not the packet contains a source route.

- *Must* update TTL according to gateway rules.
  Yes: `ip_forward` implements this requirement.

- *Must* generate ICMP error codes 4 and 5 (fragmentation required and source route failed).
  Yes: `ip_output` is able to generate a fragmentation required message, and `ip_dooptions` is able to generate the source route failed message.

- *Must* allow the IP source address of a source routed packet to not be an IP address of the forwarding host.
  Yes: `ip_output` transmits such packets.

  RFC 1122 lists this as a *may* requirement because the addresses *may* be different, which *must* be allowed.

- *Must* update timestamp and record route options.
  Yes: `ip_dooptions` processes these options for source routed packets.

- *Must* support a configurable switch for *nonlocal source routing*. The switch *must* default to off.
  No: Net/3 always allows nonlocal source routing and does not provide a switch to disable this function. Nonlocal source routing is routing packets between two different interfaces instead of receiving and sending the packet on the same interface.

- *Must* satisfy gateway access rules for nonlocal source routing.
  Yes: Net/3 follows the forwarding rules for nonlocal source routing.

- *Should* send an ICMP destination unreachable error (source route failed) if a source routed packet cannot be forwarded (except for ICMP error messages).
  Yes: `ip_dooptions` sends the ICMP destination unreachable error. `icmp_error` discards it if the original datagram was an ICMP error message.

C.4 IP Fragmentation and Reassembly Requirements

This section summarizes the IP fragmentation and reassembly requirements from Section 3.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

- *Must* be able to reassemble incoming datagrams of at least 576 bytes.
  Yes: `ip_reass` supports reassembly of datagrams of indefinite size.

- *Should* support a configurable or indefinite maximum size for incoming datagrams.
  Yes: Net/3 supports an indefinite maximum size for incoming datagrams.

- *Must* provide a mechanism for the transport layer to learn the maximum datagram size to receive.
  Not applicable: Net/3 has an indefinite limit based on available memory.

- *Must* send ICMP time exceeded error on reassembly timeout.
  No: Net/3 does not send an ICMP time exceeded error. See Figure 10.30 and Exercise 10.1.

- *Should* support a fixed reassembly timeout value. The remaining TTL value in a received IP fragment *should not* be used as a reassembly timeout value.
  Yes: Net/3 uses a compile-time value of 30 seconds (`IPFRAGTTL` is 60 slow-timeout intervals, which equals 30 seconds).

- *Must* provide the MMS_S (maximum message size to send) to higher layers.
  Partially: TCP derives the MMS_S from the maximum MTU found in the route entry for the destination or from the MTU of the outgoing interface. A UDP application does not have access to this information.

- *May* support local fragmentation of outgoing packets.
  Yes: `ip_output` fragments an outgoing packet if it is too large for the selected interface.

- *Must* not allow transport layer to send a message larger than MMS_S if local fragmentation is not supported.
  Not applicable: This is a transport-level requirement that does not apply to Net/3 since local fragmentation is supported.

- *Should not* send messages larger than 576 bytes to a remote destination in the absence of other information regarding the minimum path MTU to the destination.
  Partially: Net/3 TCP defaults to a segment size of 552 (512 data bytes + 40 header bytes). Net/3 UDP applications cannot determine if a destination is local or remote and so they often restrict their messages to 540 bytes (512 + 20 + 8). There is no kernel mechanism that prohibits sending larger messages.

- *May* support an all-subnets-MTU configuration flag.
  Yes: The global integer `subnetsarelocal` defaults to true. TCP uses this flag to select a larger segment size (the size of the outgoing interface's MTU) instead of the default segment size for destinations on a subnet of the local network.

C.5 ICMP Requirements

This section summarizes the ICMP requirements from Section 3.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

- *Must* silently discard ICMP messages with unknown type.
  Partially: `icmp_input` ignores these messages and passes them to `rip_input`, which delivers the message to any waiting processes or silently discards the message if no process is prepared to receive the message.

- *May* include more than 8 bytes of the original datagram.
  No: The `icmp_error` function returns only a maximum of 8 bytes of the original datagram in the ICMP error message, Exercise 11.9.

- *Must* return the header and data unchanged from the received datagram.
  Partially: Net/3 converts the ID, offset, and length fields of an IP packet from network byte order to host byte order in `ipintr`. This facilitates processing the packet, but Net/3 neglects to return the offset and length fields to network byte order before including the header in an ICMP error message. If the system operates with the same byte ordering as the network, this error is harmless. If it operates with a different ordering, the IP header contained within the ICMP error message has incorrect offset and length values.

  > The authors found that an Intel implementation of SVR4 and AIX 3.2 (Net/2 based) both return the length byte-swapped. Implementations other than Net/2 or Net/3 that were tried (Cisco, NetBlazer, VM, and Solaris 2.3) did not have this bug.
  >
  > Another error occurs when an ICMP port unreachable error is sent from the UDP code: the header length of the received datagram is changed incorrectly (Section 23.7). The authors found this error in Net/2 and Net/3 implementations. Net/1, however, did not have the bug.

- *Must* demultiplex received ICMP error message to transport protocol.
  Yes: `icmp_error` uses the protocol field from the original header to select the appropriate transport protocol to respond to the error.

- *Should* send ICMP error messages with a TOS field of 0.
  Yes: All ICMP error messages are constructed with a TOS of 0 by `icmp_error`.

- *Must not* send an ICMP error message caused by a previous ICMP error message.
  Partially: `icmp_error` sends an error for an ICMP redirect message, which Section 3.2.2 of RFC 1122 classifies as an ICMP error message.

- *Must not* send an ICMP error message caused by an IP broadcast or IP multicast datagram.
  No: `icmp_error` does not check for this case.

  > The `icmp_error` function from the original Deering multicast code for BSD checks for this case.

- *Must not* send an ICMP error message caused by an link-layer broadcast.
  Yes: `icmp_error` discards ICMP messages that arrive as link-layer broadcasts or multicasts.

- *Must not* send an ICMP error message caused by an noninitial fragment.
  Yes: `icmp_error` discards errors generated in this case.

- *Must not* send an ICMP error message caused by an datagram with nonunique source address.

Yes: `icmp_reflect` checks for experimental and multicast addresses. `ip_output` discards messages sent from a broadcast address.

- *Must* return ICMP error messages when not prohibited.
  Partially: In general, Net/3 sends appropriate ICMP error messages. It fails to send an ICMP reassembly timeout message at the appropriate time (Exercise 10.1).

- *Should* generate ICMP destination unreachable (protocol and port).
  No: Datagrams for unsupported protocols are delivered to `rip_input` where they are silently discarded if there are no processes registered to accept the datagrams. UDP generates an ICMP port unreachable error.

- *Must* pass ICMP destination unreachable to higher layer.
  Yes: `icmp_input` passes the message to the `pr_ctlinput` function defined for the protocol (`udp_ctlinput` and `tcp_ctlinput` for UDP and TCP, respectively).

- *Should* respond to destination unreachable error.
  See Sections 23.9 and 27.6.

- *Must* interpret destination unreachable as only a hint, as it may indicate a transient condition.
  See Sections 23.9 and 27.6.

- *Must not* send an ICMP redirect when configured as a host.
  Yes: `ip_forward`, the only function that detects and sends redirects, is not called unless the system is configured as a router.

- *Must* update route cache when an ICMP redirect is received.
  Yes: `ipintr` calls `rtredirect` to process the message.

- *Must* handle both host and network redirects. Furthermore, network redirects must be treated as host redirects.
  Yes: `ipintr` calls `rtredirect` for both types of messages.

- *Should* discard illegal redirects.
  Yes: `rtredirect` discards illegal redirects (Section 19.7).

- *May* send source quench if memory is unavailable.
  Yes: `ip_forward` sends a source quench if `ip_output` returns `ENOBUFS`. This occurs when there is a shortage of mbufs or when an interface output queue is full.

- *Must* pass source quench to higher layer.
  Yes: `icmp_input` passes source quench errors to the transport layers.

- *Should* respond to source quench in higher layer.
  See Sections 23.9 and 27.6 for UDP and TCP processing. Neither ICMP nor IGMP

accept ICMP error messages (they don't define a `pr_ctlinput` function), in which case they are discarded by IP.

- *Must* pass time exceeded error to transport layer.
  Yes: `icmp_input` passes this message to the transport layers.

- *Should* send parameter problem errors.
  Yes: `ip_dooptions` complains about incorrectly formed options.

- *Must* pass parameter problem errors to transport layer.
  Yes: `icmp_input` passes parameter problem errors to the transport layer.

- *May* report parameter problem errors to process.
  See Sections 23.9 and 27.6 for UDP and TCP processing. Neither ICMP nor IGMP accept ICMP error messages.

- *Must* support an echo server and *should* support an echo client.
  Yes: `icmp_input` implements the echo server and the `ping` program implements the echo client using a raw IP socket.

- *May* discard echo requests to a broadcast address.
  No: The reply is sent by `icmp_reflect`.

- *May* discard echo request to multicast address.
  No: Net/3 responds to multicast echo requests. Both `icmp_reflect` and `ip_output` permit multicast destination addresses.

- *Must* use specific destination address as echo reply source.
  Yes: `icmp_reflect` converts a broadcast or multicast destination to the specific address of the receiving interface and uses the result as the source address for the echo reply.

- *Must* return echo request data in echo reply.
  Yes: The data portion of the echo request is not altered by `icmp_reflect`.

- *Must* pass echo reply to higher layer.
  Yes: ICMP echo replies are passed to `rip_input` for receipt by registered processes.

- *Must* reflect record route and timestamp options in ICMP echo request message.
  Yes: `icmp_reflect` includes the record route and timestamp options in the echo reply message.

- *Must* reverse and reflect source route option.
  Yes: `icmp_reflect` retrieves the reversed source route with `ip_srcroute` and includes it in the outgoing echo reply.

- *Should not* support the ICMP information request or reply.
  Partially: The kernel does not generate or respond to either message, but a process may send or receive the messages through the raw IP mechanism.

- *May* implement the ICMP timestamp request and timestamp reply messages.
  Yes: `icmp_input` implements the timestamp server functionality. The timestamp client may be implemented through the raw IP mechanism.

- *Must* minimize timestamp delay variability (if implementing the timestamp messages).
  Partially: The receive timestamp is applied after the message is taken off the IP input queue and the transmit timestamp is applied before the message is placed in the interface output queue.

- *May* silently discard broadcast timestamp request.
  No: `icmp_input` responds to broadcast timestamp requests.

- *May* silently discard multicast timestamp requests.
  No: `icmp_input` responds to broadcast timestamp requests.

- *Must* use specific destination address as timestamp reply source address.
  Yes: `icmp_reflect` converts a broadcast or multicast destination to the specific address of the receiving interface and uses the result as the source address for the timestamp reply.

- *Should* reflect record route and timestamp options in an ICMP timestamp request.
  Yes: `icmp_reflect` includes the record route and timestamp options in the timestamp reply message.

- *Must* reverse and reflect source route option in ICMP timestamp request.
  Yes: `icmp_reflect` retrieves the reversed source route with `ip_srcroute` and includes it in the outgoing timestamp reply.

- *Must* pass timestamp reply to higher layer.
  Yes: ICMP timestamp replies are passed to `rip_input` for receipt by registered processes.

- *Must* obey rules for standard timestamp value.
  Yes: `icmp_input` calls `iptime`, which returns a standard time value.

- *Must* provide a configurable method for selecting the address mask selection method for an interface.
  No: Net/3 supports only static configuration of address masks through the `ifconfig` program.

- *Must* support static configuration of address mask.
  Yes: This is accomplished indirectly by specifying static information when the `ifconfig` program configures an interface during system initialization, typically in the `/etc/netstart` start-up script.

- *May* get address mask dynamically during system initialization.
  No: Net/3 does not support the use of BOOTP or DHCP to acquire address mask information.

- *May* get address with an ICMP address mask request and reply messages.
  No: Net/3 does not support the use ICMP messages to acquire address mask information.

- *Must* retransmit address mask request if no reply.
  Not Applicable: Not required since this method is not implemented by Net/3.

- *Should* assume default mask if no reply is received.
  Not Applicable: Not required since this method is not implemented by Net/3.

- *Must* update address mask from first reply only.
  Not Applicable: Not required since this method is not implemented by Net/3.

- *Should* perform reasonableness check on any installed address mask.
  No: Net/3 performs no reasonableness check on address masks.

- *Must not* send unauthorized address mask reply messages and *must* be explicitly configured to be agent.
  Yes: `icmp_input` only responds to address mask requests if `icmpmaskrepl` is nonzero (it defaults to 0).

- *Should* support an associated address mask authority flag with each static address mask configuration.
  No: Net/3 consults a global authority flag (`icmpmaskrepl`) to determine if it should send address mask replies for *any* interface.

- *Must* broadcast address mask reply when initialized.
  No: Net/3 does not broadcast an address mask reply when an interface is configured.

C.6 Multicasting Requirements

This section summarizes the IP multicast requirements from Section 3.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

- *Should* support local IP multicasting (RFC 1112).
  Yes: Net/3 supports IP multicasting.

- *Should* join the all-hosts group at start-up.
  Yes: `in_ifinit` joins the all-hosts group while initializing an interface.

- *Should* provide a mechanism for higher layers to discover an interface's IP multicast capability.
  Yes: The `IFF_MULTICAST` flag in the interface's `ifnet` structure is available directly to kernel code and by the `SIOCGIFFLAGS` command for processes.

C.7 IGMP Requirements

This section summarizes the IGMP requirements from Section 3.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

- *May* support IGMP (RFC 1112).
  Yes: Net/3 supports IGMP.

C.8 Routing Requirements

This section summarizes the routing requirements from Section 3.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements. Be aware that the requirements of this RFC apply to a host and not necessarily the kernel implementation. Some items are not explicitly handled by the kernel routing functions in Net/3, but they are expected to be provided by a routing daemon such as `routed` or `gated`.

- *Must* use address mask in determining whether a datagram's destination is on a connected network.
  Yes: When an interface for a connected network such as an Ethernet is configured, its address mask is specified (or a default is chosen based on the class of IP address) and stored in the routing table entry. This mask is used by `rn_match` when it checks a leaf for a network match.

- *Must* operate correctly in a minimal environment when there are no routers (all networks are directly connected).
  Yes: The system administrator must not configure a default route in this case.

- *Must* keep a "route cache" of mappings to next-hop routers.
  Yes: The routing table is the cache.

- *Should* treat a received network redirect the same as a host redirect.
  Yes, as described in Section 19.7.

- *Must* use a default router when no entry exists for the destination in the routing table.
  Yes, if a default route has been entered into the routing table.

- *Must* support multiple default routers.
  Multiple defaults are not supported by the kernel. Instead, this should be provided by a routing daemon.

- *May* implement a table of static routes.
  Yes: These can be created at system initialization time with the `route` command.

- *May* include a flag with each static route specifying whether or not the route can be overridden by a redirect.
  No.

- *May* allow the routing table key to be a complete host address and not just a network address.
  Yes: Host routes take priority over a network route to the same network.

- *Should* include the TOS in the routing table entry.
  No: There is a TOS field in the `sockaddr_inarp` that we describe in Chapter 21, but it is not currently used.

- *Must* be able to detect the failure of a next-hop router that appears as the gateway field in the routing table and be able to choose an alternate next-hop router.
  Negative advice, the `RTM_LOSING` message generated by `in_losing`, is passed to any processes reading from a routing socket, which allows the process (e.g., a routing daemon) to handle this event.

- *Should not* assume that a route is good forever.
  Yes: There are no timeouts on routing table entries in the kernel other than those created by ARP. Again, the standard Unix routing daemons time out routes and replace them with alternatives when possible.

- *Must not* ping routers continuously (ICMP echo request).
  Yes: The Net/3 kernel does not do this. The routing daemons don't generate ICMP echo requests either.

- *Must* use pinging of a router only when traffic is being sent to that router.
  The Net/3 kernel never generates pings to a next-hop router.

- *Should* allow higher and lower layers to give positive and negative advice.
  Partially: The only information passed by other layers to the Net/3 routing functions

is by `in_losing`, which is called only from TCP. The only action performed by the routing layer is to generate the `RTM_LOSING` message.

- *Must* switch to another default router when the existing default fails.
  Yes, although the Net/3 kernel does not do this, it is supported by the routing daemons.

- *Must* allow the following information to be configured manually in the routing table: IP address, network mask, list of defaults.
  Yes, but only one default is supported in the kernel.

C.9 ARP Requirements

This section summarizes the ARP requirements from Section 2.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

- *Must* provide a mechanism to flush out-of-date ARP entries. If this mechanism involves a timeout, it *should* be configurable.
  Yes and yes: `arptimer` provides this mechanism. The timeout is configurable (the `arpt_prune` and `arpt_keep` globals) but the only ways to change their values are to recompile the kernel or modify the kernel with a debugger.
- *Must* include a mechanism to prevent ARP flooding.
  Yes, as we described with Figure 21.24.
- *Should* save (rather than discard) at least one (the latest) packet of each set of packets destined to the same unresolved IP address, and transmit the saved packet when the address has been resolved.
  Yes: This is the purpose of the `la_hold` member of the `llinfo_arp` structure.

C.10 UDP Requirements

This section summarizes the UDP requirements from Section 4.1.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

- *Should* send ICMP port unreachable.
  Yes: `udp_input` does this.
- *Must* pass received IP options to application.
  No: The code to do this is commented out in `udp_input`. This means that a process that receives a UDP datagram with a source route option cannot send a reply using the reversed route.
- *Must* allow application to specify IP options to send.
  Yes: The `IP_OPTIONS` socket option does this. The options are saved in the PCB and placed into the outgoing IP datagram by `ip_output`.

- *Must* pass IP options down to IP layer.
  Yes: As mentioned above, IP places the options into the IP datagram.
- *Must* pass received ICMP messages to application.
  Yes: We must look at the exact wording from the RFC: "A UDP-based application that wants to receive ICMP error messages is responsible for maintaining the state necessary to demultiplex these messages when they arrive; for example, the application may keep a pending receive operation for this purpose." The state required by Berkeley-derived systems is that the socket be connected to the foreign address and port. As the comments at the beginning of Figure 23.26 indicate, some applications create both a connected and an unconnected socket for a given foreign port, using the connected socket to receive asynchronous errors.
- *Must* be able to generate and verify UDP checksum.
  Yes: This is done by `udp_input`, based on the global integer `udpcksum`.
- *Must* silently discard datagrams with bad checksum.
  Yes: This is done only if `udpcksum` is nonzero. As we mentioned earlier, this variable controls both the sending of checksums and the verification of received checksums. If this variable is 0, the kernel does not verify a received nonzero checksum.
- *May* allow sending application to specify whether outgoing checksum is calculated, but *must* default to on.
  No: The application has no control over UDP checksums. Regarding the default, UDP checksums are generated unless the kernel is compiled with 4.2BSD compatibility defined, or unless the administrator has disabled UDP checksums using `sysctl`(8).
- *May* allow receiving application to specify whether received UDP datagrams without a checksum (i.e., the received checksum is 0) are discarded or passed to the application.
  No: Received datagrams with a checksum field of 0 are passed to the receiving process.
- *Must* pass destination IP address to application.
  Yes: The application must call `recvmsg` and specify the `IP_RECVDSTADDR` socket option. Also recall our discussion following Figure 23.25 noting that 4.4BSD broke this option when the destination address is a multicast or broadcast address.
- *Must* allow application to specify local IP address to be used when sending a UDP datagram.
  Yes: The application can call `bind` to set the local IP address. Recall our discussion at the end of Section 22.8 about the difference between the source IP address and the IP address of the outgoing interface. Net/3 does not allow the application to choose the outgoing interface—that is done by `ip_output`, based on the route to the destination IP address.
- *Must* allow application to specify wildcard local IP address.
  Yes: If the IP address `INADDR_ANY` is specified in the call to `bind`, the local IP address is chosen by `in_pcbconnect`, based on the route to the destination.

- *Should* allow application to learn of the local address that was chosen.
  Yes: The application must call `connect`. When a datagram is sent on an unconnected socket with a wildcard local address, `ip_output` chooses the outgoing interface, which also becomes the source address. The `inp_laddr` member of the PCB, however, is restored to the wildcard address at the end of `udp_output` before `sendto` returns. Therefore, `getsockname` cannot return the value. But the application can `connect` a UDP socket to the destination, causing `in_pcbconnect` to determine the local interface and store the address in the PCB. The application can then call `getsockname` to fetch the IP address of the local interface.
- *Must* silently discard a received UDP datagram with an invalid source IP address (broadcast or multicast).
  No: A received UDP datagram with an invalid source address is delivered to a socket, if a socket is bound to the destination port.
- *Must* send a valid IP source address.
  Yes: If the local IP address is set by `bind`, it checks the validity of the address. If the local IP address is wildcarded, `ip_output` chooses the local address.
- *Must* provide the full IP interface from Section 3.4 of RFC 1122.
  Refer to Section C.2.
- *Must* allow application to specify TTL, TOS, and IP options for output datagrams.
  Yes: The application can use the `IP_TTL`, `IP_TOS`, and `IP_OPTIONS` socket options.
- *May* pass received TOS to application.
  No: There is no way for the application to receive this value from the IP header. Notice that a `getsockopt` of `IP_TOS` returns the value used in outgoing datagrams, not the value from a received datagram. The received `ip_tos` value is available to `udp_input`, but is discarded along with the entire IP header.

C.11 TCP Requirements

This section summarizes the TCP requirements from Section 4.2.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

PSH Flag

- *May* aggregate data sent by the user without the PSH flag.
  Yes and no: Net/3 does not give the process a way to specify the PSH flag with a write operation, but Net/3 does aggregate data sent by the user in separate write operations.
- *May* queue data received without the PSH flag.
  No: The absence or presence of a PSH flag in a received datagram makes no difference. Received data is placed onto the socket's received queue when it is processed.

- Sender *should* collapse successive PSH flags when it packetizes data.
  No.
- *May* implement PSH flag on write calls.
  No: This is not part of the sockets API.
- Since the PSH flag is not part of the write calls, *must not* buffer data indefinitely and *must* set the PSH flag in the last buffered segment.
  Yes: This is the method used by Berkeley-derived implementations.
- *May* pass received PSH flag to application.
  No: This is not part of the sockets API.
- *Should* send maximum-sized segment whenever possible, to improve performance.
  Yes.

Window

- *Must* treat window size as an unsigned number. *Should* treat window size as 32-bit value.
  Yes: All the window sizes in Figure 24.13 are `unsigned longs`, which is also required by the window scale option of RFC 1323.
- Receiver *must not* shrink the window (move the right edge to the left).
  Yes, in Figure 26.29.
- Sender *must* be robust against window shrinking.
  Yes, in Figure 29.15.
- *May* keep offered receive window closed indefinitely.
  Yes.
- Sender *must* probe a zero window.
  Yes, this is the purpose of the persist timer.
- *Should* send first zero-window probe when the window has been closed for the RTO.
  No: Net/3 sets a lower bound for the persist timer of 5 seconds, which is normally greater than the RTO.
- *Should* exponentially increase the interval between successive probes.
  Yes, as shown in Figure 25.14.
- *Must* allow peer's window to stay closed indefinitely.
  Yes, TCP never gives up probing a closed window.
- Sender *must not* timeout a connection just because the other end keeps advertising a zero window.
  Yes.

Urgent Data

- *Must* have urgent pointer point to last byte of urgent data.
  No: Berkeley-derived implementations continue to interpret the urgent pointer as pointing just beyond the last byte of urgent data.

- *Must* support a sequence of urgent data of any length.
  Yes, with the bug fix discussed in Exercise 26.6.
- *Must* inform the receiving process (1) when TCP receives an urgent pointer and there was no previously pending urgent data, or (2) when the urgent pointer advances in the data stream.
  Yes, in Figure 29.17.
- *Must* be a way for the process to determine how much urgent data remains, or at least whether more urgent data remains to be read.
  Yes, this is the purpose of the out-of-band mark, the `SIOCATMARK` `ioctl`.

TCP Options

- *Must* be able to receive TCP options in any segment.
  Yes.
- *Must* ignore any options not supported.
  Yes, in Section 28.3.
- *Must* cope with an illegal option length.
  Yes, in Section 28.3.
- *Must* implement both sending and receiving the MSS option.
  Yes, a received MSS option is handled in Figure 28.10, and Figure 26.23 always sends an MSS option with a SYN.
- *Should* send an MSS option in every SYN when its receive MSS differs from 536, and *may* send it always.
  Yes, as mentioned earlier, an MSS option is always sent by Net/3 with a SYN.
- If an MSS option is not received with a SYN, *must* assume a default MSS of 536.
  No: The default MSS is 512, not 536.

  This is probably a historical artifact because VAXes had a physical page size of 512 bytes and trailer protocols working only with data that is a multiple of 512.
- *Must* calculate the "effective send MSS."
  Yes, in Section 27.5.

TCP Checksums

- *Must* generate a TCP checksum in outgoing segments and *must* verify received checksums.
  Yes, TCP checksums are always calculated and verified.

Initial Sequence Number Selection

- *Must* use the specified clock-driven selection from RFC 793.
  No: RFC 793 specifies a clock that changes by 125,000 every half-second, whereas

the Net/3 ISN (the global variable `tcp_iss`) is incremented by 64,000 every half-second, about one-half the specified rate.

Opening Connections

- *Must* support simultaneous open attempts.
  Yes, although Berkeley-derived systems prior to 4.4BSD did not support this, as described in Section 28.9.
- *Must* keep track of whether it reached the SYN_RCVD state from the LISTEN or SYN_SENT states.
  Yes, same result, different technique. The purpose of this requirement is to allow a passive open that receives an RST to return to the LISTEN state (as shown in Figure 24.15), but force an active open that ends up in SYN_RCVD and then receives an RST to be aborted. This is described following Figure 28.36.
- A passive open *must not* affect previously created connections.
  Yes.
- *Must* allow a listening socket with a given local port at the same time that another socket with the same local port is in the SYN_SENT or SYN_RCVD state.
  Yes: The stated purpose of this requirement is to allow a given application to accept multiple connection attempts at about the same time. This is done in Berkeley-derived implementations by cloning new connections from the socket in the LISTEN state when the incoming SYN arrives.
- *Must* ask IP to select a local IP address to be used as the source IP address when the source IP address is not specified by the process performing an active open on a multihomed host.
  Yes, done by `in_pcbconnect`.
- *Must* continue to use the same source IP address for all segments sent on a connection.
  Yes: Once `in_pcbconnect` selects the source address, it doesn't change.
- *Must not* allow an active open for a broadcast or multicast foreign address.
  Yes and no: TCP will not send segments to a broadcast address because the call to `ip_output` in Figure 26.32 does not specify the `SO_BROADCAST` option. Net/3, however, allows connection attempts to multicast addresses.
- *Must* ignore incoming SYNs with an invalid source address.
  Yes: The code in Figure 28.16 checks for these invalid source addresses.

Closing Connections

- *Should* allow an RST to contain data.
  No: The RST processing in Figure 28.36 ends up jumping to `drop`, which skips the processing of any segment data in Figure 29.22.
- *Must* inform process whether other end closed the connection normally (e.g., sent a FIN) or aborted the connection with an RST.

Yes: The read system calls return 0 (end-of-file) when the FIN is processed, but –1 with an error of `ECONNRESET` when an RST is received.

- *May* implement a half-close.
  Yes: The process calls `shutdown` with a second argument of 1 to send a FIN. The process can still read from the connection.
- If the process completely closes a connection (i.e., not a half-close) and received data is still pending in TCP, or if new data arrives after the close, TCP *should* send an RST to indicate data was lost.
  No and yes: If a process calls `close` and unread data is in the socket's receive buffer, an RST is not sent. But if data arrives after a socket is closed, an RST is returned to the sender.
- *Must* linger in TIME_WAIT state for twice the MSL.
  Yes, although the Net/3 MSL of 30 seconds is much smaller than the RFC 793 recommended value of 2 minutes.
- *May* accept a new SYN from a peer to reopen a connection directly from the TIME_WAIT state.
  Yes, as shown in Figure 28.29.

Retransmissions

- *Must* implement Van Jacobson's slow start and congestion avoidance.
  Yes.
- *May* reuse the same IP identifier field when a retransmission is identical to the original packet.
  No: The IP identifier is assigned by `ip_output` from the global variable `ip_id`, which increments each time an IP datagram is sent. It is not assigned by TCP.
- *Must* implement Jacobson's algorithm for calculating the RTO and Karn's algorithm for selecting the RTT measurements.
  Yes, but realize that when RFC 1323 timestamps are present, the retransmission ambiguity problem is gone, obviating half of Karn's algorithm, as we discussed with Figure 29.6.
- *Must* include an exponential backoff for successive RTO values.
  Yes, as described with Figure 25.22.
- Retransmission of SYN segments *should* use the same algorithm as data segments.
  Yes, as shown in Figure 25.15.
- *Should* initialize estimation parameters to calculate an initial RTO of 3 seconds.
  No: The initial value of `t_rxtcur` calculated by `tcp_newtcpcb` is 6 seconds. This is also seen in Figure 25.15.
- *Should* have a lower bound on the RTO measured in fractions of a second and an upper bound of twice the MSL.
  No: The lower bound is 1 second and the upper bound is 64 seconds (Figure 25.3).

Generating ACKs

- *Should* queue out-of-order segments.
  Yes, done by `tcp_reass`.
- *Must* process all queued segments before sending any ACKs.
  Yes, but only for in-order segments. `ipintr` calls `tcp_input` for each queued datagram that is a TCP segment. For in-order segments, `tcp_input` schedules a delayed ACK and returns to `ipintr`. If there are additional TCP segments on IP's input queue, `tcp_input` is called by `ipintr` for each one. Only when `ipintr` finds no more IP datagrams on its input queue and returns can `tcp_fasttimo` be called to generate a delayed ACK. This ACK will contain the highest acknowledgment number in all the segments processed by `tcp_input`.

  The problem is with out-of-order segments: `tcp_input` calls `tcp_output` itself, before returning to `ipintr`, to generate the ACK for the out-of-order segment. If there are additional segments on IP's input queue that would have made the out-of-order segment be in order, they are processed after the immediate ACK is sent.
- *May* generate an immediate ACK for an out-of-order segment.
  Yes, this is needed for the fast retransmit and fast recovery algorithms (Section 29.4).
- *Should* implement delayed ACKs and the delay *must* be less than 0.5 seconds.
  Yes: The `TF_DELACK` flag is checked by the `tcp_fasttimo` function every 200 ms.
- *Should* send an ACK for at least every second segment.
  Yes, the code in Figure 26.9 generates an ACK for every second segment. We also discussed that this happens only if the process receiving the data reads the data as it arrives, since the calls to `tcp_output` that cause every other segment to be acknowledged are driven by the `PRU_RCVD` request.
- *Must* include silly window syndrome avoidance in the receiver.
  Yes, as seen in Figure 26.29.

Sending Data

- The TTL value for TCP segments *must* be configurable.
  Yes: The TTL is initialized to 64 (`IPDEFTTL`) by `tcp_newtcpcb`, but can then be changed by a process using the `IP_TTL` socket option.
- *Must* include sender silly window syndrome avoidance.
  Yes, in Figure 26.8.
- *Should* implement the Nagle algorithm.
  Yes, in Figure 26.8.
- *Must* allow a process to disable the Nagle algorithm on a given connection.
  Yes, with the `TCP_NODELAY` socket option.

Connection Failures

- *Must* pass negative advice to IP when the number of retransmissions for a given segment exceeds some value R1.
  Yes: The value of R1 is 4, and in Figure 25.26, when the number of retransmissions exceeds 4, `in_losing` is called.
- *Must* close a connection when the number of retransmissions for a given segment exceeds some value R2.
  Yes: The value of R2 is 12 (Figure 25.26).
- *Must* allow process to set the value of R2.
  No: The value 12 is hardcoded in Figure 25.26.
- *Should* inform the process when R1 is reached and before R2 is reached.
  No.
- *Should* default R1 to at least 3 retransmissions and R2 to at least 100 seconds.
  Yes: R1 is 4 retransmissions, and with a minimum RTO of 1 second, the `tcp_backoff` array (Section 25.9) guarantees a minimum value of R2 of over 500 seconds.
- *Must* handle SYN retransmissions in the same general way as data retransmissions.
  Yes, but R1 is normally not reached for the retransmission of a SYN (Figure 25.15).
- *Must* set R2 to at least 3 minutes for a SYN.
  No: R2 for a SYN is limited to 75 seconds by the connection-establishment timer (Figure 25.15).

Keepalive Packets

- *May* provide keepalives.
  Yes, they are provided.
- *Must* allow process to turn keepalives on or off, and *must* default to off.
  Yes: Default is off and process must turn them on with the `SO_KEEPALIVE` socket option.
- *Must* send keepalives only when connection is idle for a given period.
  Yes.
- *Must* allow the keepalive interval to be configurable and *must* default to no less than 2 hours.
  No and yes: The idle time before sending keepalive probes is not easily configurable, but it defaults to 2 hours. If the default idle time is changed (by changing the global variable `tcp_keepidle`), it affects all users of the keepalive option on the host—it cannot be configured on a per-connection basis as many users would like.
- *Must not* interpret the failure to respond to any given probe as a dead connection.
  Yes: Nine probes are sent before the connection is considered dead.

IP Options

- *Must* ignore received IP options it doesn't understand.
  Yes: This is done by the IP layer.
- *May* support the timestamp and record route options in received segments.
  No: Net/3 only reflects these options for ICMP packets that are reflected back to the sender (`icmp_reflect`). `tcp_input` discards any received IP options by calling `ip_stripoptions` in Figure 28.2.
- *Must* allow process to specify a source route when a connection is actively opened, and this route must take precedence over a source route received for this connection.
  Yes: The source route is specified with the `IP_OPTIONS` socket option. `tcp_input` never looks at a received source route when the connection is actively opened.
- *Must* save a received source route in a connection that is passively opened and use the return route for all segments sent on this connection. If a different source route arrives in a later segment, the later route *should* override the earlier one.
  Yes and no: Figure 28.7 calls `ip_srcroute`, but only when the SYN arrives for a listening socket. If a different source route arrives later, it is not used.

Receiving ICMP Messages from IP

- Receipt of an ICMP source quench *should* trigger slow start.
  Yes: The function `tcp_quench` is called by `tcp_ctlinput`.
- Receipt of a network unreachable, host unreachable, or source route failed *must not* cause TCP to abort the connection and the process *should* be informed.
  Yes and no: As described following Figure 27.12, Net/3 now completely ignores host unreachable and network unreachable errors for an established connection.
- Receipt of a protocol unreachable, port unreachable, or fragmentation required and DF bit set *should* abort an existing connection.
  No: `tcp_notify` records these ICMP error in `t_softerror`, which is reported to the process if the connection is eventually dropped.
- *Should* handle time exceeded and parameter problem errors the same as required previously for network and host unreachable.
  Yes: ICMP parameter problem errors are just recorded in `t_softerror` by `tcp_notify`. ICMP time exceeded errors are ignored by `tcp_ctlinput`. Neither type of ICMP error causes the connection to be aborted.

Application Programming Interface

- *Must* be a method for reporting soft errors to the process, normally in an asynchronous fashion.
  No: Soft errors are returned to the process if the connection is aborted.

- *Must* allow process to specify TOS for segments sent on a connection. *Should* let application change this during a connection's lifetime.
  Yes to both, with the `IP_TOS` socket option.
- *May* pass most recently received TOS to process.
  No: There is no way to do this with the sockets API. Calling `getsockopt` for `IP_TOS` returns only the current value being sent; it does not return the most recently received value.
- *May* implement a "flush" call.
  No: TCP sends the data from the process as quickly as it can.
- *Must* allow process to specify local IP address before either an active open or a passive open.
  Yes: This is done by calling `bind` before either `connect` or `accept`.

Bibliography

All the RFCs are available at no charge through electronic mail or by using anonymous FTP across the Internet as described in Appendix B.

Whenever the authors were able to locate an electronic copy of papers and reports referenced in this bibliography, its URL (Uniform Resource Locator, Appendix B) is included.

Almquist, P. 1992. "Type of Service in the Internet Protocol Suite," RFC 1349, 28 pages (July).

Almquist, P., and Kastenholz, F. J. 1994. "Towards Requirements for IP Routers," RFC 1716, 186 pages (Nov.).

This RFC is an intermediate step to replace RFC 1009 [Braden and Postel 1987].

Auerbach, K. 1994. "Max IP Packet Length and MTU," Message-ID <karl.3.000A4DD7 @cavebear.com>, Usenet, comp.protocols.tcp-ip Newsgroup (July).

Boggs, D. R. 1982. "Internet Broadcasting," Xerox PARC CSL-83-3, Stanford University, Palo Alto, Calif. (Jan.).

Braden, R. T., ed. 1989a. "Requirements for Internet Hosts—Communication Layers," RFC 1122, 116 pages (Oct.).

The first half of the Host Requirements RFC. This half covers the link layer, IP, TCP, and UDP.

Braden, R. T., ed. 1989b. "Requirements for Internet Hosts—Application and Support," RFC 1123, 98 pages (Oct.).

The second half of the Host Requirements RFC. This half covers Telnet, FTP, TFTP, SMTP, and the DNS.

Braden, R. T. 1989c. "Perspective on the Host Requirements RFCs," RFC 1127, 20 pages (Oct.).

An informal summary of the discussions and conclusions of the IETF working group that developed the Host Requirements RFC.

Braden, R. T. 1992. "TIME-WAIT Assassination Hazards in TCP," RFC 1337, 11 pages (May).

Shows how the receipt of an RST while in the TIME_WAIT state can lead to problems.

Braden, R. T. 1993. "TCP Extensions for High Performance: An Update," Internet Draft, 10 pages (June).

This is an update to RFC 1323 [Jacobson, Braden, and Borman 1992].

`http://www.noao.edu/~rstevens/tcplw-extensions.txt`

Braden, R. T. 1994. "T/TCP—TCP Extensions for Transactions, Functional Specification," RFC 1644, 38 pages (July).

Braden, R. T., Borman, D. A., and Partridge, C. 1988. "Computing the Internet Checksum," RFC 1071, 24 pages (Sept.).

Provides techniques and algorithms for calculating the checksum used by IP, ICMP, IGMP, UDP, and TCP.

Braden, R. T., and Postel, J. B. 1987. "Requirements for Internet Gateways," RFC 1009, 55 pages (June).

The equivalent of the Host Requirements RFC for routers. This RFC is being replaced by RFC 1716 [Almquist and Kastenholz 1994].

Brakmo, L. S., O'Malley, S. W., and Peterson, L. L. 1994. "TCP Vegas: New Techniques for Congestion Detection and Avoidance," *Computer Communication Review*, vol. 24, no. 4, pp. 24–35 (Oct.).

Describes modifications to the 4.3BSD Reno TCP implementation to improve throughput and reduce retransmissions.

`ftp://ftp.cs.arizona.edu/xkernel/Papers/vegas.ps`

Carlson, J. 1993. "Re: Bug in Many Versions of TCP," Message-ID <1993Jul12.130854.26176@xylogics.com>, Usenet, comp.protocols.tcp-ip Newsgroup (July).

Casner, S., *Frequently Asked Questions (FAQ) on the Multicast Backbone (MBONE)*, 1993.

`ftp://ftp.isi.edu/mbone/faq.txt`

Cheswick, W. R., and Bellovin, S. M. 1994. *Firewalls and Internet Security: Repelling the Wily Hacker.* Addison-Wesley, Reading, Mass.

Describes how to set up and administer a firewall gateway and the security issues involved.

Clark, D. D. 1982. "Modularity and Efficiency in Protocol Implementation," RFC 817, 26 pages (July).

Comer, D. E., and Lin, J. C. 1994. "TCP Buffering and Performance Over an ATM Network," Purdue Technical Report CSD-TR 94-026, Purdue University, West Lafayette, Ind. (Mar.).

`ftp://gwen.cs.purdue.edu/pub/lin/TCP.atm.ps.Z`

Comer, D. E., and Stevens, D. L. 1993. *Internetworking with TCP/IP: Vol. III: Client–Server Programming and Applications, BSD Socket Version.* Prentice-Hall, Englewood Cliffs, N.J.

Croft, W., and Gilmore, J. 1985. "Bootstrap Protocol (BOOTP)," RFC 951, 12 pages (Sept.).

Crowcroft, J., Wakeman, I., Wang, Z., and Sirovica, D. 1992. "Is Layering Harmful?," *IEEE Network*, vol. 6, no. 1, pp. 20–24 (Jan.).

The seven missing figures from this paper appear in the next issue, vol. 6, no. 2 (March).

Dalton, C., Watson, G., Banks, D., Calamvokis, C., Edwards, A., and Lumley, J. 1993. "Afterburner," *IEEE Network*, vol. 7, no. 4, pp. 36–43 (July).

Describes how to speed up TCP by reducing the number of data copies performed, and a special-purpose interface card that supports this design.

Deering, S. E. 1989. "Host Extensions for IP Multicasting," RFC 1112, 17 pages (Aug.).

The specification of IP multicasting and IGMP.

Deering, S. E., ed. 1991a. "ICMP Router Discovery Messages," RFC 1256, 19 pages (Sept.).

Deering, S. E. 1991b. "Multicast Routing in a Datagram Internetwork," STAN-CS-92-1415, Stanford University, Palo Alto, Calif. (Dec.).

`ftp://gregorio.stanford.edu/vmtp-ip/sdthesis.part1.ps.Z`

Deering, S. E., and Cheriton, D. P. 1990. "Multicast Routing in Datagram Internetworks and Extended LANs," *ACM Transactions on Computer Systems*, vol. 8, no. 2, pp. 85–110 (May).

Proposes extensions to common routing techniques to support multicasting.

Deering, S., Estrin, D., Farinacci, D., Jacobson, V., Liu, C., and Wei, L. 1994. "An Architecture for Wide-Area Multicast Routing," *Computer Communication Review*, vol. 24, no. 4, pp. 126–135 (Oct.).

Droms, R. 1993. "Dynamic Host Configuration Protocol," RFC 1541, 39 pages (Oct.).

Finlayson, R., Mann, T., Mogul, J. C., and Theimer, M. 1984. "A Reverse Address Resolution Protocol," RFC 903, 4 pages (June).

Floyd, S. 1994. Private Communication.

Forgie, J. 1979. "ST—A Proposed Internet Stream Protocol," IEN 119, MIT Lincoln Laboratory (Sept.).

Fuller, V., Li, T., Yu, J. Y., and Varadhan, K. 1993. "Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy," RFC 1519, 24 pages (Sept.).

Hornig, C. 1984. "Standard for the Transmission of IP Datagrams over Ethernet Networks," RFC 894, 3 pages (Apr.).

Hutchinson, N. C., and Peterson, L. L. 1991. "The x-Kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, vol. 17, no. 1, pp. 64–76 (Jan.).

`ftp://ftp.cs.arizona.edu/xkernel/Papers/architecture.ps`

Itano, W. M., and Ramsey, N. F. 1993. "Accurate Measurement of Time," *Scientific American*, vol. 269, p. 56 (July).

Overview of historical and current methods for accurate timekeeping. Includes a short discussion of international time scales including International Atomic Time (TAI) and Coordinated Universal Time (UTC).

Jacobson, V. 1988a. "Some Interim Notes on the BSD Network Speedup," Message-ID <8807200426.AA01221@helios.ee.lbl.gov>, Usenet, comp.protocols.tcp-ip Newsgroup (July).

Jacobson, V. 1988b. "Congestion Avoidance and Control," *Computer Communication Review*, vol. 18, no. 4, pp. 314–329 (Aug.).

A classic paper describing the slow start and congestion avoidance algorithms for TCP.

`ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z`

Jacobson, V. 1990a. "Compressing TCP/IP Headers for Low-Speed Serial Links," RFC 1144, 43 pages (Feb.).

Describes CSLIP, a version of SLIP with the TCP and IP headers compressed.

Jacobson, V. 1990b. "4BSD TCP Header Prediction," *Computer Communication Review*, vol. 20, no. 2, pp. 13–15 (Apr.).

Jacobson, V. 1990c. "Modified TCP Congestion Avoidance Algorithm," April 30, 1990, end2end-interest mailing list (Apr.).

Describes the fast retransmit and fast recovery algorithms.

`ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail`

Jacobson, V. 1990d. "Berkeley TCP Evolution from 4.3-Tahoe to 4.3-Reno," *Proceedings of the Eighteenth Internet Engineering Task Force*, p. 365 (Sept.), University of British Columbia, Vancouver, B.C.

Jacobson, V. 1993. "Some Design Issues for High-Speed Networks," *Networkshop '93* (Nov.), Melbourne, Australia.

A set of 21 overheads.

`ftp://ftp.ee.lbl.gov/talks/vj-nws93-1.ps.Z`

Jacobson, V., and Braden, R. T. 1988. "TCP Extensions for Long-Delay Paths," RFC 1072, 16 pages (Oct.).

Describes the selective acknowledgment option for TCP, which was removed from the later RFC 1323, and the echo options, which were replaced with the timestamp option in RFC 1323.

Jacobson, V., Braden, R. T., and Borman, D. A. 1992. "TCP Extensions for High Performance," RFC 1323, 37 pages (May).

Describes the window scale option, the timestamp option, and the PAWS algorithm, along with the reasons these modifications are needed. [Braden 1993] updates this RFC.

Jain, R., and Routhier, S. A. 1986. "Packet Trains: Measurements and a New Model for Computer Network Traffic," *IEEE Journal on Selected Areas in Communications*, vol. 4, pp. 1162–1167.

Karels, M. J., and McKusick, M. K. 1986. "Network Performance and Management with 4.3BSD and IP/TCP," *Proceedings of the 1986 Summer USENIX Conference*, pp. 182–188, Atlanta, Ga.

Describes the changes made from 4.2BSD to 4.3BSD with regard to TCP/IP.

Karn, P., and Partridge, C. 1987. "Improving Round-Trip Time Estimates in Reliable Transport Protocols," *Computer Communication Review*, vol. 17, no. 5, pp. 2–7 (Aug.).

Details of Karn's algorithm to handle the retransmission timeout for segments that have been retransmitted.

`ftp://sics.se/users/craig/karn-partridge.ps`

Kay, J., and Pasquale, J. 1993. "The Importance of Non-Data Touching Processing Overheads in TCP/IP," *Computer Communication Review*, vol. 23, no. 4, pp. 259–268 (Sept.).

Kent, C. A., and Mogul, J. C. 1987. "Fragmentation Considered Harmful," *Computer Communication Review*, vol. 17, no. 5, pp. 390–401 (Aug.).

Kernighan, B. W., and Plauger, P. J. 1976. *Software Tools.* Addison-Wesley, Reading, Mass.

Krol, E. 1994. *The Whole Internet, Second Edition.* O'Reilly & Associates, Sebastopol, Calif.

An introduction into the Internet, common Internet applications, and various resources available on the Internet.

Krol, E., and Hoffman, E. 1993. "FYI on 'What is the Internet?'," RFC 1462, 11 pages (May).

Lanciani, D. 1993. "Re: Bug in Many Versions of TCP," Message-ID <1993Jul10.015938.15951@burrhus.harvard.edu>, Usenet, comp.protocols.tcp-ip Newsgroup (July).

Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. 1989. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, Reading, Mass.

An entire book on the 4.3BSD Unix system. This book describes the Tahoe release of 4.3BSD.

Lynch, D. C. 1993. "Historical Perspective," in *Internet System Handbook,* eds. D. C. Lynch and M. T. Rose, pp. 3–14. Addison-Wesley, Reading, Mass.

A historical overview of the Internet and its precursor, the ARPANET.

Mallory, T., and Kullberg, A. 1990. "Incremental Updating of the Internet Checksum," RFC 1141, 2 pages (Jan.).

This RFC is updated by RFC 1624 [Rijsinghani 1994].

Mano, M. M. 1993. *Computer System Architecture, Third Edition.* Prentice-Hall, Englewood Cliffs, N.J.

McCanne, S., and Jacobson, V. 1993. "The BSD Packet Filter: A New Architecture for User-Level Packet Capture," *Proceedings of the 1993 Winter USENIX Conference,* pp. 259–269, San Diego, Calif.

A detailed description of the BSD Packet Filter (BPF) and comparisons with Sun's Network Interface Tap (NIT).

`ftp://ftp.ee.lbl.gov/papers/bpf-usenix93.ps.Z`

McCloghrie, K., and Farinacci, D. 1994a. "Internet Group Management Protocol MIB," Internet Draft, 12 pages (Jul.).

McCloghrie, K., and Farinacci, D. 1994b. "IP Multicast Routing MIB," Internet Draft, 15 pages (Jul.).

McCloghrie, K., and Rose, M. T. 1991. "Management Information Base for Network Management of TCP/IP-based Internets: MIB-II," RFC 1213 (Mar.).

McGregor, G. 1992. "PPP Internet Protocol Control Protocol (IPCP)," RFC 1332, 12 pages (May).

McKenney, P. E., and Dove, K. F. 1992. "Efficient Demultiplexing of Incoming TCP Packets," *Computer Communication Review,* vol. 22, no. 4, pp. 269–279 (Oct.).

Mogul, J. C. 1991. "Network Locality at the Scale of Processes," *Computer Communication Review,* vol. 21, no. 4, pp. 273–284 (Sept.).

Mogul, J. C. 1993. "IP Network Performance," in *Internet System Handbook,* eds. D. C. Lynch and M. T. Rose, pp. 575–675. Addison-Wesley, Reading, Mass.

Covers numerous topics in the Internet protocols that are candidates for tuning to obtain optimal performance.

Mogul, J. C., and Deering, S. E. 1990. "Path MTU Discovery," RFC 1191, 19 pages (Apr.).

Mogul, J. C., and Postel, J. B. 1985. "Internet Standard Subnetting Procedure," RFC 950, 18 pages (Aug.).

Moy, J. 1994. "Multicast Extensions to OSPF," RFC 1584, 102 pages (Mar.).

Olivier, G. 1994. "What is the Diameter of the Internet?," Message-ID <1994Jan22.094832@mines.u-nancy.fr>, Usenet, comp.unix.wizards Newsgroup (Jan.).

Partridge, C. 1987. "Implementing the Reliable Data Protocol (RDP)," *Proceedings of the 1987 Summer USENIX Conference,* pp. 367–379, Phoenix, Ariz.

Partridge, C. 1993. "Jacobson on TCP in 30 Instructions," Message-ID <1993Sep8.213239.28992@sics.se>, Usenet, comp.protocols.tcp-ip Newsgroup (Sept.).

Describes a research implementation of TCP/IP being developed by Van Jacobson that reduces TCP receive packet processing down to 30 instructions on a RISC system.

Partridge, C., and Hinden, R. 1990. "Version 2 of the Reliable Data Protocol (RDP)," RFC 1151, 4 pages (Apr.).

Partridge, C., Mendez, T., and Milliken, W. 1993. "Host Anycasting Service," RFC 1546, 9 pages (Nov.).

Partridge, C., and Pink, S. 1993. "A Faster UDP," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 429–440 (Aug.).

Describes implementation improvements to the Berkeley sources to speed up UDP performance about 30%.

Paxson, V. 1994. Private Communication.

Perlman, R. 1992. *Interconnections: Bridges and Routers.* Addison-Wesley, Reading, Mass.

Piscitello, D. M., and Chapin, A. L. 1993. *Open Systems Networking: TCP/IP and OSI.* Addison-Wesley, Reading, Mass.

Plummer, D. C. 1982. "An Ethernet Address Resolution Protocol," RFC 826, 10 pages (Nov.).

Postel, J. B., ed. 1981a. "Internet Protocol," RFC 791, 45 pages (Sept.).

Postel, J. B. 1981b. "Internet Control Message Protocol," RFC 792, 21 pages (Sept.).

Postel, J. B., ed. 1981c. "Transmission Control Protocol," RFC 793, 85 pages (Sept.).

Postel, J. B. 1981d. "Service Mappings," RFC 795, 4 pages (Sept.).

Postel, J. B., and Reynolds, J. K. 1988. "Standard for the Transmission of IP Datagrams over IEEE 802 Networks," RFC 1042, 15 pages (Apr.).

Rago, S. A. 1993. *UNIX System V Network Programming.* Addison-Wesley, Reading, Mass.

Reynolds, J. K., and Postel, J. B. 1994. "Assigned Numbers," RFC 1700, 230 pages (Oct.).

Rijsinghani, A. 1994. "Computation of the Internet Checksum via Incremental Update," RFC 1624, 6 pages (May).

An update to RFC 1141 [Mallory and Kullberg 1990].

Romkey, J. L. 1988. "A Nonstandard for Transmission of IP Datagrams Over Serial Lines: SLIP," RFC 1055, 6 pages (June).

Rose, M. T. 1990. *The Open Book: A Practical Perspective on OSI.* Prentice-Hall, Englewood Cliffs, N.J.

Salus, P. H. 1994. *A Quarter Century of Unix.* Addison-Wesley, Reading, Mass.

Sedgewick, R. 1990. *Algorithms in C.* Addison-Wesley, Reading, Mass.

Simpson, W. A. 1993. "The Point-to-Point Protocol (PPP)," RFC 1548, 53 pages (Dec.).

Sklower, K. 1991. "A Tree-Based Packet Routing Table for Berkeley Unix," *Proceedings of the 1991 Winter USENIX Conference*, pp. 93–99, Dallas, Tex.

Stallings, W. 1987. *Handbook of Computer-Communications Standards, Volume 2: Local Network Standards.* Macmillan, New York.

Stallings, W. 1993. *Networking Standards: A Guide to OSI, ISDN, LAN, and MAN Standards.* Addison-Wesley, Reading, Mass.

Stevens, W. R. 1990. *UNIX Network Programming.* Prentice-Hall, Englewood Cliffs, N.J.

Stevens, W. R. 1992. *Advanced Programming in the UNIX Environment.* Addison-Wesley, Reading, Mass.

Stevens, W. R. 1994. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley, Reading, Mass.

The first volume in this series, which provides a complete introduction to the Internet protocols.

Tanenbaum, A. S. 1989. *Computer Networks, Second Edition.* Prentice-Hall, Englewood Cliffs, N.J.

Topolcic, C. 1990. "Experimental Stream Protocol, Version 2 (SY-II)," RFC 1190, 148 pages (Oct.).

Torek, C. 1992. "Re: A Problem in Bind System Call," Message-ID <27240@dog.ee.lbl.gov>, Usenet, comp.unix.internals Newsgroup (Nov.).

Waitzman, D., Partridge, C., and Deering, S. E. 1988. "Distance Vector Multicast Routing Protocol," RFC 1075, 24 pages (Nov.).

Index

Rather than provide a separate glossary (with most of the entries being acronyms), this index also serves as a glossary for all the acronyms used in the book. The primary entry for the acronym appears under the acronym name. For example, all references to the Address Resolution Protocol appear under ARP. The entry under the compound term "Address Resolution Protocol" refers back to the main entry under ARP.

The two end papers at the back of the book contain a list of all the functions and macros presented or described in the text, along with the starting page number of the source code. Similarly one front end paper contains a list of all the structures presented in the text. These end papers should be the starting point to locate the definition of a function or structure.

The various functions, constants, variables, and the like that appear in this index refer to their appearance in the text. We have not attempted to index all these names when they appear in source code files that are included in the text. The definitive answer to a question such as "where are all the references to the constant `IP_RECVOPTS`" can only be obtained by obtaining the Net/3 source code (Appendix B) and using a tool such as `grep`.

The entries in this index for RFCs refer only to the reference for that RFC in the Bibliography. This is to help locate an RFC if you encounter a reference to it by number within the text.

Function and Macro Definitions

Function and Macro Definitions

Networking, TCP/IP

TCP/IP Illustrated, Volume 2

"Teaching Cisco internetworking classes around the country, I get to hear about every new data communications book of note because my students bring them to class for reference. Stevens' *TCP/IP Illustrated, Volume 1* arrived like a bomb going off ... students raved about the book. I've just read *TCP/IP Illustrated, Volume 2*. If *Volume 1* was like a bomb, then *Volume 2* is the Big Bang!

— Greg Gentile, Director of Training
Protocol Interface

"After reading Stevens' *TCP/IP Illustrated, Volume 1*, I thought it would be hard to come up with another book as useful, but *Volume 2* is just that. Some of you might ask how this book applies to you if you aren't a full-time network programmer. Since my earliest days of using UNIX®, the standard answer to 'How does this really work?' has been 'Look at the source.' With this book, you can not only look at the source, but also have one of the clearest explanations as to how it all fits together."

— Greg Merrell, Systems Engineer
International Network Services

TCP/IP Illustrated, an ongoing series covering the many facets of TCP/IP, brings a highly-effective visual approach to learning about this networking protocol suite.

TCP/IP Illustrated, Volume 2 contains a thorough explanation of how TCP/IP protocols are implemented. There isn't a more practical or up-to-date book — this volume is the only one to cover the de facto standard implementation from the 4.4BSD-Lite release, the foundation for TCP/IP implementations run daily on hundreds of thousands of systems worldwide.

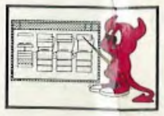
Combining 500 illustrations with 15,000 lines of real working code, *TCP/IP Illustrated, Volume 2* uses a teach-by-example approach to help you master TCP/IP implementation. You will learn about such topics as the relationship between the sockets API and the protocol suite, and the differences between a host implementation and a router. In addition, the book covers the newest features of the 4.4BSD-Lite release, including multicasting, long fat pipe support, window scale, timestamp options, protection against wrapped sequence numbers, and many other topics.

Comprehensive in scope, based on a working standard, and thoroughly illustrated, this book is an indispensable resource for anyone working with TCP/IP.

Gary R. Wright has worked with TCP/IP for more than eight years. He is President of Connix, a Connecticut-based company providing Internet access and consulting services. **W. Richard Stevens** is the highly-respected author of three best-selling books, *TCP/IP Illustrated, Volume 1* (Addison-Wesley, 1994), *Advanced Programming in the UNIX Environment* (Addison-Wesley, 1992), and *UNIX Network Programming* (Prentice-Hall). He is also a popular tutorials instructor and consultant.

Cover art by Fine Line Design
♻ Text printed on recycled paper
Corporate & Professional Publishing Group
Addison-Wesley Publishing Company

90000
9 780201 633542
ISBN 0-201-63354-X



TCP/IP Illustrated, Volume 2

Wright
Stevens

Addison
Wesley
63354