value?

by the

on one
lticast

of the

,

nod to

multi-

ce has
)K?

tion is

ised to

13

IGMP: Internet Group Management Protocol

13.1 Introduction

IGMP conveys group membership information between hosts and routers on a local network. Routers periodically multicast IGMP queries to the all-hosts group. Hosts respond to the queries by multicasting IGMP report messages. The IGMP specification appears in RFC 1112. Chapter 13 of Volume 1 describes the specification of IGMP and provides some examples.

From an architecture perspective, IGMP is a transport protocol above IP. It has a protocol number (2) and its messages are carried in IP datagrams (as with ICMP). IGMP usually isn't accessed directly by a process but, as with ICMP, a process can send and receive IGMP messages through an IGMP socket. This feature enables multicast routing daemons to be implemented as user-level processes.

Figure 13.1 shows the overall organization of the IGMP protocol in Net/3.

The key to IGMP processing is the collection of in_multi structures shown in the center of Figure 13.1. An incoming IGMP query causes igmp_input to initialize a countdown timer for each in_multi structure. The timers are updated by igmp_fasttimo, which calls igmp_sendreport as each timer expires.

We saw in Chapter 12 that ip_setmoptions calls igmp_joingroup when a new in_multi structure is created. igmp_joingroup calls igmp_sendreport to announce the new group and enables the group's timer to schedule a second announcement a short time later. igmp_sendreport takes care of formatting an IGMP message and passing it to ip_output.

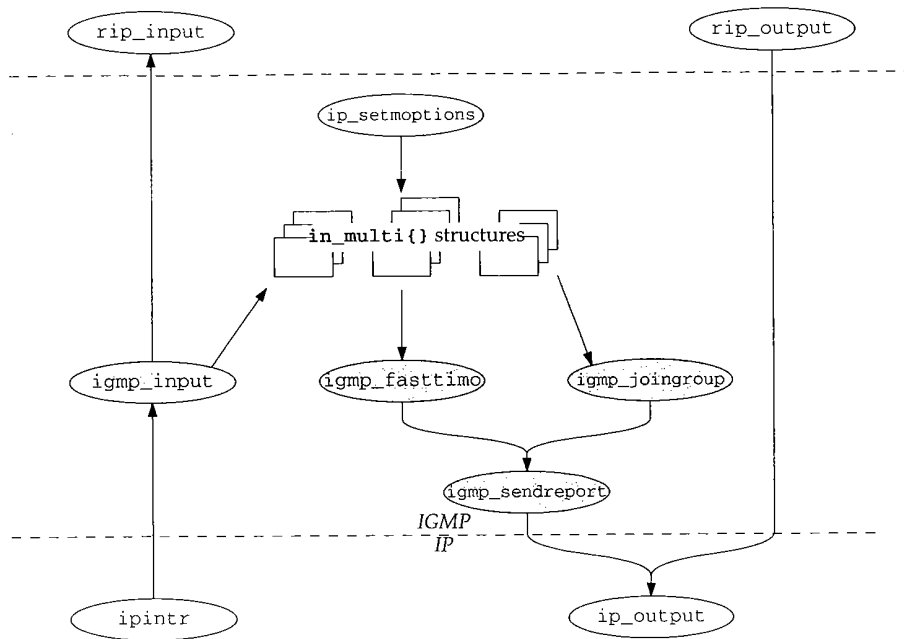On the left and right of Figure 13.1 we see that a raw socket can send and receive IGMP messages directly.

Figure 13.1   Summary of IGMP processing.

13.2 Code Introduction

The IGMP protocol is implemented in four files listed in Figure 13.2.

File	Description
netinet/igmp.h	IGMP protocol definitions
netinet/igmp_var.h	IGMP implementation definitions
netinet/in_var.h	IP multicast data structures
netinet/igmp.c	IGMP protocol implementation

Figure 13.2   Files discussed in this chapter.

Global Variables

Three new global variables, shown in Figure 13.3, are introduced in this chapter.

Statistics

IGMP statistics are maintained in the igmpstat variables shown in Figure 13.4.

Variable	Datatype	Description
`igmp_all_hosts_group`	`u_long`	all-hosts group address in network byte order
`igmp_timers_are_running`	`int`	true if any IGMP timer is active, false otherwise
`igmpstat`	`struct igmpstat`	IGMP statistics (Figure 13.4).

Figure 13.3 Global variables introduced in this chapter.

`igmpstat` member	Description
`igps_rcv_badqueries`	#messages received as invalid queries
`igps_rcv_badreports`	#messages received as invalid reports
`igps_rcv_badsum`	#messages received with bad checksum
`igps_rcv_ourreports`	#messages received as reports for local groups
`igps_rcv_queries`	#messages received as membership queries
`igps_rcv_reports`	#messages received as membership reports
`igps_rcv_tooshort`	#messages received with too few bytes
`igps_rcv_total`	total #IGMP messages received
`igps_snd_reports`	#messages sent as membership reports

Figure 13.4 IGMP statistics.

Figure 13.5 shows some sample output of these statistics, from the `netstat -p igmp` command on `vangogh.cs.berkeley.edu`.

`netstat -p igmp` output	`igmpstat` member
18774 messages received	`igps_rcv_total`
0 messages received with too few bytes	`igps_rcv_tooshort`
0 messages received with bad checksum	`igps_rcv_badsum`
18774 membership queries received	`igps_rcv_queries`
0 membership queries received with invalid field(s)	`igps_rcv_badqueries`
0 membership reports received	`igps_rcv_reports`
0 membership reports received with invalid field(s)	`igps_rcv_badreports`
0 membership reports received for groups to which we belong	`igps_rcv_ourreports`
0 membership reports sent	`igps_snd_reports`

Figure 13.5 Sample IGMP statistics.

From Figure 13.5 we can tell that `vangogh` is attached to a network where IGMP is being used, but that `vangogh` is not joining any multicast groups, since `igps_snd_reports` is 0.

SNMP Variables

There is no standard SNMP MIB for IGMP, but [McCloghrie and Farinacci 1994a] describes an experimental MIB for IGMP.

13.3 `igmp` Structure

An IGMP message is only 8 bytes long. Figure 13.6 shows the `igmp` structure used by Net/3.

```
                                                                     ──igmp.h
43 struct igmp {
44       u_char   igmp_type;        /* version & type of IGMP message  */
45       u_char   igmp_code;        /* unused, should be zero          */
46       u_short  igmp_cksum;       /* IP-style checksum               */
47       struct in_addr igmp_group; /* group address being reported    */
48 };                               /* (zero for queries)              */
                                                                     ──igmp.h
```

Figure 13.6 `igmp` structure.

43-44   A 4-bit version code and a 4-bit type code are contained within `igmp_type`. Figure 13.7 shows the standard values.

Version	Type	igmp_type	Description
1	1	0x11 (IGMP_HOST_MEMBERSHIP_QUERY)	membership query
1	2	0x12 (IGMP_HOST_MEMBERSHIP_REPORT)	membership report
1	3	0x13	DVMRP message (Chapter 14)

Figure 13.7 IGMP message types.

Only version 1 messages are used by Net/3. Multicast routers send type 1 (`IGMP_HOST_MEMBERSHIP_QUERY`) messages to solicit membership reports from hosts on the local network. The response to a type 1 IGMP message is a type 2 (`IGMP_HOST_MEMBERSHIP_REPORT`) message from the hosts reporting their multicast membership information. Type 3 messages transport multicast routing information between routers (Chapter 14). A host never processes type 3 messages. The remainder of this chapter discusses only type 1 and 2 messages.

45-46   `igmp_code` is unused in IGMP version 1, and `igmp_cksum` is the familiar IP checksum computed over all 8 bytes of the IGMP message.

47-48   `igmp_group` is 0 for queries. For replies, it contains the multicast group being reported.
Figure 13.8 shows the structure of an IGMP message relative to an IP datagram.

13.4 IGMP `protosw` Structure

Figure 13.9 describes the `protosw` structure for IGMP.

Although it is possible for a process to send raw IP packets through the IGMP `protosw` entry, in this chapter we are concerned only with how the kernel processes IGMP messages. Chapter 32 discusses how a process can access IGMP using a raw socket.
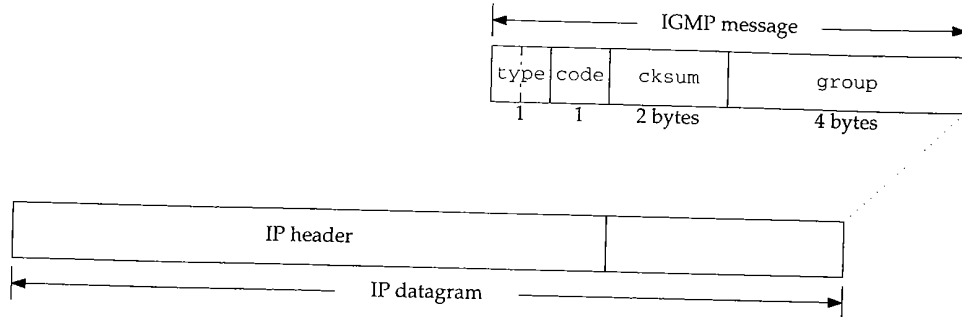
ed by

igmp.h

igmp.h

. Fig-

)

rpe 1
hosts
rpe 2
lticast
nation
under

iar IP

being

l.

IGMP
cesses
a raw

Figure 13.8 An IGMP message (igmp_ omitted).

Member	inetsw[5]	Description
pr_type	SOCK_RAW	IGMP provides raw packet services
pr_domain	&inetdomain	IGMP is part of the Internet domain
pr_protocol	IPPROTO_IGMP (2)	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	igmp_input	receives messages from IP layer
pr_output	rip_output	sends IGMP message to IP layer
pr_ctlinput	0	not used by IGMP
pr_ctloutput	rip_ctloutput	respond to administrative requests from a process
pr_usrreq	rip_usrreq	respond to communication requests from a process
pr_init	igmp_init	initialization for IGMP
pr_fasttimo	igmp_fasttimo	process pending membership reports
pr_slowtimo	0	not used by IGMP
pr_drain	0	not used by IGMP
pr_sysctl	0	not used by IGMP

Figure 13.9 The IGMP protosw structure.

There are three events that trigger IGMP processing:

- a local interface has joined a new multicast group (Section 13.5),
- an IGMP timer has expired (Section 13.6), and
- an IGMP query is received (Section 13.7).

There are also two events that trigger local IGMP processing but do not result in any messages being sent:

- an IGMP report is received (Section 13.7), and
- a local interface leaves a multicast group (Section 13.8).

These five events are discussed in the following sections.

13.5 Joining a Group: `igmp_joingroup` Function

We saw in Chapter 12 that `igmp_joingroup` is called by `in_addmulti` when a new `in_multi` structure is created. Subsequent requests to join the same group only increase the reference count in the `in_multi` structure; `igmp_joingroup` is not called. `igmp_joingroup` is shown in Figure 13.10

——— *igmp.c*

```
164 void
165 igmp_joingroup(inm)
166 struct in_multi *inm;
167 {
168     int     s = splnet();
169     if (inm->inm_addr.s_addr == igmp_all_hosts_group ||
170         inm->inm_ifp == &loif)
171         inm->inm_timer = 0;
172     else {
173         igmp_sendreport(inm);
174         inm->inm_timer = IGMP_RANDOM_DELAY(inm->inm_addr);
175         igmp_timers_are_running = 1;
176     }
177     splx(s);
178 }
```

——— *igmp.c*

Figure 13.10 `igmp_joingroup` function.

164–178   `inm` points to the new `in_multi` structure for the group. If the new group is the all-hosts group, or the membership request is for the loopback interface, `inm_timer` is disabled and `igmp_joingroup` returns. Membership in the all-hosts group is never reported, since every multicast host is assumed to be a member of the group. Sending a membership report to the loopback interface is unnecessary, since the local host is the only system on the loopback network and it already knows its membership status.

In the remaining cases, a report is sent immediately for the new group, and the group timer is set to a random value based on the group. The global flag `igmp_timers_are_running` is set to indicate that at least one timer is enabled. `igmp_fasttimo` (Section 13.6) examines this variable to avoid unnecessary processing.

When the timer for the new group expires, a second membership report is issued. The duplicate report is harmless, but it provides insurance in case the first report is lost or damaged. The report delay is computed by `IGMP_RANDOM_DELAY` (Figure 13.11).

59–73   According to RFC 1122, report timers should be set to a random time between 0 and 10 (`IGMP_MAX_HOST_REPORT_DELAY`) seconds. Since IGMP timers are decremented five (`PR_FASTHZ`) times per second, `IGMP_RANDOM_DELAY` must pick a random value between 1 and 50. If r is the random number computed by adding the total number of IP packets received, the host's primary IP address, and the multicast group, then

$$0 \leq (r \bmod 50) \leq 49$$

and

$$1 \leq (r \bmod 50) + 1 \leq 50$$

```
                                                                              ─── igmp_var.h
59  /*
60   * Macro to compute a random timer value between 1 and (IGMP_MAX_REPORTING_
61   * DELAY * countdown frequency).  We generate a "random" number by adding
62   * the total number of IP packets received, our primary IP address, and the
63   * multicast address being timed-out.  The 4.3 random() routine really
64   * ought to be available in the kernel!
65   */
66  #define IGMP_RANDOM_DELAY(multiaddr) \
67      /* struct in_addr multiaddr; */ \
68      ( (ipstat.ips_total + \
69          ntohl(IA_SIN(in_ifaddr)->sin_addr.s_addr) + \
70          ntohl((multiaddr).s_addr) \
71        ) \
72        % (IGMP_MAX_HOST_REPORT_DELAY * PR_FASTHZ) + 1 \
73      )
                                                                              ─── igmp_var.h
```

Figure 13.11   IGMP_RANDOM_DELAY function.

Zero is avoided because it would disable the timer and no report would be sent.

13.6   `igmp_fasttimo` Function

Before looking at `igmp_fasttimo`, we need to describe the mechanism used to traverse the `in_multi` structures.

To locate each `in_multi` structure, Net/3 must traverse the `in_multi` list for each interface. During a traversal, an `in_multistep` structure (shown in Figure 13.12) records the position.

```
                                                                              ─── in_var.h
123  struct in_multistep {
124      struct in_ifaddr *i_ia;
125      struct in_multi *i_inm;
126  };
                                                                              ─── in_var.h
```

Figure 13.12   in_multistep function.

123–126   `i_ia` points to the *next* `in_ifaddr` interface structure and `i_inm` points to the *next* `in_multi` structure for the *current* interface.

The `IN_FIRST_MULTI` and `IN_NEXT_MULTI` macros (shown in Figure 13.13) traverse the lists.

154–169   If the `in_multi` list has more entries, `i_inm` is advanced to the next entry. When `IN_NEXT_MULTI` reaches the end of a multicast list, `i_ia` is advanced to the next interface and `i_inm` to the first `in_multi` structure associated with the interface. If the interface has no multicast structures, the `while` loop continues to advance through the interface list until all interfaces have been searched.

170–177   The `in_multistep` array is initialized to point to the first `in_ifaddr` structure in the `in_ifaddr` list and `i_inm` is set to null. `IN_NEXT_MULTI` finds the first `in_multi` structure.

—————————————————————————————————————— *in_var.h*

```
147 /*
148  * Macro to step through all of the in_multi records, one at a time.
149  * The current position is remembered in "step", which the caller must
150  * provide.  IN_FIRST_MULTI(), below, must be called to initialize "step"
151  * and get the first record.  Both macros return a NULL "inm" when there
152  * are no remaining records.
153  */
154 #define IN_NEXT_MULTI(step, inm) \
155      /* struct in_multistep  step; */ \
156      /* struct in_multi *inm; */ \
157 { \
158      if (((inm) = (step).i_inm) != NULL) \
159          (step).i_inm = (inm)->inm_next; \
160      else \
161          while ((step).i_ia != NULL) { \
162              (inm) = (step).i_ia->ia_multiaddrs; \
163              (step).i_ia = (step).i_ia->ia_next; \
164              if ((inm) != NULL) { \
165                  (step).i_inm = (inm)->inm_next; \
166                  break; \
167              } \
168          } \
169 }

170 #define IN_FIRST_MULTI(step, inm) \
171      /* struct in_multistep step; */ \
172      /* struct in_multi *inm; */ \
173 { \
174      (step).i_ia = in_ifaddr; \
175      (step).i_inm = NULL; \
176      IN_NEXT_MULTI((step), (inm)); \
177 }
```

—————————————————————————————————————— *in_var.h*

Figure 13.13   IN_FIRST_MULTI and IN_NEXT_MULTI structures.

We know from Figure 13.9 that `igmp_fasttimo` is the fast timeout function for IGMP and is called five times per second. `igmp_fasttimo` (shown in Figure 13.14) decrements multicast report timers and sends a report when the timer expires.

187–198   If `igmp_timers_are_running` is false, `igmp_fasttimo` returns immediately instead of wasting time examining each timer.

199–213   `igmp_fasttimo` resets the running flag and then initializes `step` and `inm` with IN_FIRST_MULTI. The `igmp_fasttimo` function locates each `in_multi` structure with the `while` loop and the IN_NEXT_MULTI macro. For each structure:

- If the timer is 0, there is nothing to be done.
- If the timer is nonzero, it is decremented. If it reaches 0, an IGMP membership report is sent for the group.
- If the timer is still nonzero, then at least one timer is still running, so `igmp_timers_are_running` is set to 1.

```
187 void                                                                      igmp.c
188 igmp_fasttimo()
189 {
190     struct in_multi *inm;
191     int     s;
192     struct in_multistep step;

193     /*
194      * Quick check to see if any work needs to be done, in order
195      * to minimize the overhead of fasttimo processing.
196      */
197     if (!igmp_timers_are_running)
198         return;

199     s = splnet();
200     igmp_timers_are_running = 0;
201     IN_FIRST_MULTI(step, inm);
202     while (inm != NULL) {
203         if (inm->inm_timer == 0) {
204             /* do nothing */
205         } else if (--inm->inm_timer == 0) {
206             igmp_sendreport(inm);
207         } else {
208             igmp_timers_are_running = 1;
209         }
210         IN_NEXT_MULTI(step, inm);
211     }
212     splx(s);
213 }
```
—— igmp.c

Figure 13.14   igmp_fasttimo function.

igmp_sendreport Function

The igmp_sendreport function (shown in Figure 13.15) constructs and sends an IGMP report message for a single multicast group.

214–232   The single argument inm points to the in_multi structure for the group being reported. igmp_sendreport allocates a new mbuf and prepares it for an IGMP message. igmp_sendreport leaves room for a link-layer header and sets the length of the mbuf and packet to the length of an IGMP message.

233–245   The IP header and IGMP message is constructed one field at a time. The source address for the datagram is set to INADDR_ANY, and the destination address is the multicast group being reported. ip_output replaces INADDR_ANY with the unicast address of the outgoing interface. Every member of the group receives the report as does every multicast router (since multicast routers receive *all* IP multicasts).

246–260   Finally, igmp_sendreport constructs an ip_moptions structure to go along with the message sent to ip_output. The interface associated with the in_multi structure is selected as the outgoing interface; the TTL is set to 1 to keep the report on the local network; and, if the local system is configured as a router, multicast loopback is enabled for this request.

igmp.c

```
214 static void
215 igmp_sendreport(inm)
216 struct in_multi *inm;
217 {
218     struct mbuf *m;
219     struct igmp *igmp;
220     struct ip *ip;
221     struct ip_moptions *imo;
222     struct ip_moptions simo;

223     MGETHDR(m, M_DONTWAIT, MT_HEADER);
224     if (m == NULL)
225         return;
226     /*
227      * Assume max_linkhdr + sizeof(struct ip) + IGMP_MINLEN
228      * is smaller than mbuf size returned by MGETHDR.
229      */
230     m->m_data += max_linkhdr;
231     m->m_len = sizeof(struct ip) + IGMP_MINLEN;
232     m->m_pkthdr.len = sizeof(struct ip) + IGMP_MINLEN;

233     ip = mtod(m, struct ip *);
234     ip->ip_tos = 0;
235     ip->ip_len = sizeof(struct ip) + IGMP_MINLEN;
236     ip->ip_off = 0;
237     ip->ip_p = IPPROTO_IGMP;
238     ip->ip_src.s_addr = INADDR_ANY;
239     ip->ip_dst = inm->inm_addr;

240     igmp = (struct igmp *) (ip + 1);  ·
241     igmp->igmp_type = IGMP_HOST_MEMBERSHIP_REPORT;
242     igmp->igmp_code = 0;
243     igmp->igmp_group = inm->inm_addr;
244     igmp->igmp_cksum = 0;
245     igmp->igmp_cksum = in_cksum(m, IGMP_MINLEN);

246     imo = &simo;
247     bzero((caddr_t) imo, sizeof(*imo));
248     imo->imo_multicast_ifp = inm->inm_ifp;
249     imo->imo_multicast_ttl = 1;

250     /*
251      * Request loopback of the report if we are acting as a multicast
252      * router, so that the process-level routing demon can hear it.
253      */
254     {
255         extern struct socket *ip_mrouter;
256         imo->imo_multicast_loop = (ip_mrouter != NULL);
257     }
258     ip_output(m, NULL, NULL, 0, imo);

259     ++igmpstat.igps_snd_reports;
260 }
```

igmp.c

Figure 13.15   `igmp_sendreport` function.

The process-level multicast router must hear the membership reports. In Section 12.14 we saw that IGMP datagrams are always accepted when the system is configured as a multicast router. Through the normal transport demultiplexing code, the messages are passed to igmp_input, the pr_input function for IGMP (Figure 13.9).

13.7 Input Processing: igmp_input Function

In Section 12.14 we described the multicast processing portion of ipintr. We saw that a multicast router accepts *any* IGMP message, but a multicast host accepts only IGMP messages that arrive on an interface that is a member of the destination multicast group (i.e., queries and membership reports for which the receiving interface is a member).

The accepted messages are passed to igmp_input by the standard protocol demultiplexing mechanism. The beginning and end of igmp_input are shown in Figure 13.16. The code for each IGMP message type is described in following sections.

Validate IGMP message

52–96   The function ipintr passes m, a pointer to the received packet (stored in an mbuf), and iphlen, the size of the IP header in the datagram.

The datagram must be large enough to contain an IGMP message (IGMP_MINLEN), must be contained within a standard mbuf header (m_pullup), and must have a correct IGMP checksum. If any errors are found, they are counted, the datagram is silently discarded, and igmp_input returns.

The body of igmp_input processes the validated messages based on the code in igmp_type. Remember from Figure 13.6 that igmp_type includes a version code and a type code. The switch statement is based on the combined value stored in igmp_type (Figure 13.7). Each case is described separately in the following sections.

Pass IGMP messages to raw IP

157–163   There is no default case for the switch statement. Any valid message (i.e., one that is properly formed) is passed to rip_input where it is delivered to any process listening for IGMP messages. IGMP messages with versions or types that are unrecognized by the kernel can be processed or discarded by the listening processes.

> The mrouted program depends on this call to rip_input so that it receives membership queries and reports.

Membership Query: IGMP_HOST_MEMBERSHIP_QUERY

RFC 1075 recommends that multicast routers issue an IGMP membership query at least once every 120 seconds. The query is sent to group 224.0.0.1 (the all-hosts group). Figure 13.17 shows how the message is processed by a host.

igmp.c

```
52 void
53 igmp_input(m, iphlen)
54 struct mbuf *m;
55 int      iphlen;
56 {
57     struct igmp *igmp;
58     struct ip *ip;
59     int      igmplen;
60     struct ifnet *ifp = m->m_pkthdr.rcvif;
61     int      minlen;
62     struct in_multi *inm;
63     struct in_ifaddr *ia;
64     struct in_multistep step;

65     ++igmpstat.igps_rcv_total;

66     ip = mtod(m, struct ip *);
67     igmplen = ip->ip_len;

68     /*
69      * Validate lengths
70      */
71     if (igmplen < IGMP_MINLEN) {
72         ++igmpstat.igps_rcv_tooshort;
73         m_freem(m);
74         return;
75     }
76     minlen = iphlen + IGMP_MINLEN;
77     if ((m->m_flags & M_EXT || m->m_len < minlen) &&
78         (m = m_pullup(m, minlen)) == 0) {
79         ++igmpstat.igps_rcv_tooshort;
80         return;
81     }
82     /*
83      * Validate checksum
84      */
85     m->m_data += iphlen;
86     m->m_len -= iphlen;
87     igmp = mtod(m, struct igmp *);
88     if (in_cksum(m, igmplen)) {
89         ++igmpstat.igps_rcv_badsum;
90         m_freem(m);
91         return;
92     }
93     m->m_data -= iphlen;
94     m->m_len += iphlen;
95     ip = mtod(m, struct ip *);

96     switch (igmp->igmp_type) {


                              /* switch cases */


157     }
```

```
158     /*
159      * Pass all valid IGMP packets up to any process(es) listening
160      * on a raw IGMP socket.
161      */
162     rip_input(m);
163 }
```
                                                                    ── igmp.c

Figure 13.16 igmp_input function.


```
97      case IGMP_HOST_MEMBERSHIP_QUERY:                              ── igmp.c
98          ++igmpstat.igps_rcv_queries;

99          if (ifp == &loif)
100             break;

101         if (ip->ip_dst.s_addr != igmp_all_hosts_group) {
102             ++igmpstat.igps_rcv_badqueries;
103             m_freem(m);
104             return;
105         }
106         /*
107          * Start the timers in all of our membership records for
108          * the interface on which the query arrived, except those
109          * that are already running and those that belong to the
110          * "all-hosts" group.
111          */
112         IN_FIRST_MULTI(step, inm);
113         while (inm != NULL) {
114             if (inm->inm_ifp == ifp && inm->inm_timer == 0 &&
115                 inm->inm_addr.s_addr != igmp_all_hosts_group) {
116                 inm->inm_timer =
117                     IGMP_RANDOM_DELAY(inm->inm_addr);
118                 igmp_timers_are_running = 1;
119             }
120             IN_NEXT_MULTI(step, inm);
121         }

122         break;
```
                                                                    ── igmp.c

Figure 13.17 Input processing of the IGMP query message.

97–122 Queries that arrive on the loopback interface are silently discarded (Exercise 13.1). Queries by definition are sent to the all-hosts group. If a query arrives addressed to a different address, it is counted in igps_rcv_badqueries and discarded.

The receipt of a query message does not trigger an immediate flurry of IGMP membership reports. Instead, igmp_input resets the membership timers for each group associated with the interface on which the query was received to a random value with IGMP_RANDOM_DELAY. When the timer for a group expires, igmp_fasttimo sends a membership report. Meanwhile, the same activity is occurring on all the other hosts that received the IGMP query. As soon as the random timer for a particular group expires on one host, it is multicast to that group. This report cancels the timers on the

other hosts so that only one report is multicast to the network. The routers, as well as any other members of the group, receive the report.

The one exception to this scenario is the all-hosts group. A timer is never set for this group and a report is never sent.

Membership Report: `IGMP_HOST_MEMBERSHIP_REPORT`

The receipt of an IGMP membership report is one of the two events we mentioned in Section 13.1 that does not result in an IGMP message. The effect of the message is local to the interface on which it was received. Figure 13.18 shows the message processing.

```
                                                                              ───── igmp.c
123      case IGMP_HOST_MEMBERSHIP_REPORT:
124             ++igmpstat.igps_rcv_reports;

125          if (ifp == &loif)
126             break;

127          if (!IN_MULTICAST(ntohl(igmp->igmp_group.s_addr)) ||
128              igmp->igmp_group.s_addr != ip->ip_dst.s_addr) {
129              ++igmpstat.igps_rcv_badreports;
130              m_freem(m);
131              return;
132          }
133          /*
134           * KLUDGE: if the IP source address of the report has an
135           * unspecified (i.e., zero) subnet number, as is allowed for
136           * a booting host, replace it with the correct subnet number
137           * so that a process-level multicast routing demon can
138           * determine which subnet it arrived from.  This is necessary
139           * to compensate for the lack of any way for a process to
140           * determine the arrival interface of an incoming packet.
141           */
142          if ((ntohl(ip->ip_src.s_addr) & IN_CLASSA_NET) == 0) {
143              IFP_TO_IA(ifp, ia);
144              if (ia)
145                  ip->ip_src.s_addr = htonl(ia->ia_subnet);
146          }
147          /*
148           * If we belong to the group being reported, stop
149           * our timer for that group.
150           */
151          IN_LOOKUP_MULTI(igmp->igmp_group, ifp, inm);
152          if (inm != NULL) {
153              inm->inm_timer = 0;
154              ++igmpstat.igps_rcv_ourreports;
155          }
156          break;
                                                                              ───── igmp.c
```

Figure 13.18   Input processing of the IGMP report message.

123-156   Reports sent to the loopback interface are discarded, as are membership reports sent to the incorrect multicast group. That is, the message must be addressed to the group identified within the message.

The source address of an incompletely initialized host might not include a network or host number (or both). igmp_report looks at the class A network portion of the address, which can only be 0 when the network and subnet portions of the address are 0. If this is the case, the source address is set to the subnet address, which includes the network ID and subnet ID, of the receiving interface. The only reason for doing this is to inform a process-level daemon of the receiving interface, which is identified by the subnet number.

If the receiving interface belongs to the group being reported, the associated report timer is reset to 0. In this way the first report sent to the group stops any other hosts from issuing a report. It is only necessary for the router to know that at least one interface on the network is a member of the group. The router does not need to maintain an explicit membership list or even a counter.

13.8 Leaving a Group: igmp_leavegroup Function

We saw in Chapter 12 that in_delmulti calls igmp_leavegroup when the last reference count in the associated in_multi structure drops to 0.

```
                                                                    ─ igmp.c
179  void
180  igmp_leavegroup(inm)
181  struct in_multi *inm;
182  {
183      /*
184       * No action required on leaving a group.
185       */
186  }
                                                                    ─ igmp.c
```

Figure 13.19 igmp_leavegroup function.

179-186   As we can see, IGMP takes no action when an interface leaves a group. No explicit notification is sent—the next time a multicast router issues an IGMP query, the interface does not generate an IGMP report for this group. If no report is generated for a group, the multicast router assumes that all the interfaces have left the group and stops forwarding multicast packets for the group to the network.

If the interface leaves the group while a report is pending (i.e., the group's report timer is running), the report is never sent, since the timer is discarded by in_delmulti (Figure 12.36) along with the in_multi structure for the group when icmp_leavegroup returns.

13.9 Summary

In this chapter we described IGMP, which communicates IP multicast membership information between hosts and routers on a single network. IGMP membership reports are generated when an interface joins a group, and on demand when multicast routers issue an IGMP report query message.

The design of IGMP minimizes the number of messages required to communicate membership information:

- Hosts announce their membership when they join a group.
- Response to membership queries are delayed for a random interval, and the first response suppresses any others.
- Hosts are silent when they leave a group.
- Membership queries are sent no more than once per minute.

Multicast routers share the IGMP information they collect with each other (Chapter 14) to route multicast datagrams toward remote members of the multicast destination group.

Exercises

13.1   Why isn't it necessary to respond to an IGMP query on the loopback interface?

13.2   Verify the assumption stated on lines 226 to 229 in Figure 13.15.

13.3   Is it necessary to set random delays for membership queries that arrive on a point-to-point network interface?

r 13

;hip
orts
ters

.cate

first

:r 14)
ation

-point

14

IP Multicast Routing

14.1 Introduction

The previous two chapters discussed multicasting on a single network. In this chapter we look at multicasting across an entire internet. We describe the operation of the `mrouted` program, which computes the multicast routing tables, and the kernel functions that forward multicast datagrams between networks.

> Technically, multicast *packets* are forwarded. In this chapter we assume that every multicast packet contains an entire datagram (i.e., there are no fragments), so we use the term *datagram* exclusively. Net/3 forwards IP fragments as well as IP datagrams.

Figure 14.1 shows several versions of `mrouted` and how they correspond to the BSD releases. The `mrouted` releases include both the user-level daemons and the kernel-level multicast code.

`mrouted` version	Description
1.2	modifies the 4.3BSD Tahoe release
2.0	included with 4.4BSD and Net/3
3.3	modifies SunOS 4.1.3

Figure 14.1 `mrouted` and IP multicasting releases.

IP multicast technology is an active area of research and development. This chapter discusses version 2.0 of the multicast software, which is included in Net/3 but is considered an obsolete implementation. Version 3.3 was released too late to be discussed fully in this text, but we will point out various 3.3 features along the way.

397

Because commercial multicast routers are not widely deployed, multicast networks are often constructed using multicast *tunnels*, which connect two multicast routers over a standard IP unicast internet. Multicast tunnels are supported by Net/3 and are constructed with the Loose Source Record Route (LSRR) option (Section 9.6). An improved tunneling technique encapsulates the IP multicast datagram within an IP unicast datagram and is supported by version 3.3 of the multicast code but is not supported by Net/3.

As in Chapter 12, we use the generic term *transport protocols* to refer to the protocols that send and receive multicast datagrams, but UDP is the only Internet protocol that supports multicasting.

14.2 Code Introduction

The three files listed in Figure 14.2 are discussed in this chapter.

File	Description
netinet/ip_mroute.h	multicast structure definitions
netinet/ip_mroute.c	multicast routing functions
netinet/raw_ip.c	multicast routing options

Figure 14.2   Files discussed in this chapter.

Global Variables

The global variables used by the multicast routing code are shown in Figure 14.3.

Variable	Datatype	Description
cached_mrt	struct mrt	one-behind cache for multicast routing
cached_origin	u_long	multicast group for one-behind cache
cached_originmask	u_long	mask for multicast group for one-behind cache
mrtstat	struct mrtstat	multicast routing statistics
mrttable	struct mrt *[]	hash table of pointers to multicast routes
numvifs	vifi_t	number of enabled multicast interfaces
viftable	struct vif[]	array of virtual multicast interfaces

Figure 14.3   Global variables introduced in this chapter.

Statistics

All the statistics collected by the multicast routing code are found in the mrtstat structure described by Figure 14.4. Figure 14.5 shows some sample output of these statistics, from the netstat -gs command.

works
; over
: con-
roved
data-
ed by

tocols
l that

mrtstat member	Description	Used by SNMP
mrts_mrt_lookups	#multicast route lookups	
mrts_mrt_misses	#multicast route cache misses	
mrts_grp_lookups	#group address lookups	
mrts_grp_misses	#group address cache misses	
mrts_no_route	#multicast route lookup failures	
mrts_bad_tunnel	#packets with malformed tunnel options	
mrts_cant_tunnel	#packets with no room for tunnel options	

Figure 14.4   Statistics collected in this chapter.

netstat -gs output	mrtstat members

multicast routing:
  329569328 multicast route lookups
    9377023 multicast route cache misses
  242754062 group address lookups
  159317788 group address cache misses
      65648 datagrams with no route for origin
          0 datagrams with malformed tunnel options
          0 datagrams with no room for tunnel options
	
	mrts_mrt_lookups
	mrts_mrt_misses
	mrts_grp_lookups
	mrts_grp_misses
	mrts_no_route
	mrts_bad_tunnel
	mrts_cant_tunnel

Figure 14.5   Sample IP multicast routing statistics.

These statistics are from a system with two physical interfaces and one tunnel inter-face. These statistics show that the multicast route is found in the cache 98% of the time. The group address cache is less effective with only a 34% hit rate. The route cache is described with Figure 14.34 and the group address cache with Figure 14.21.

SNMP Variables

There is no standard SNMP MIB for multicast routing, but [McCloghrie and Farinacci 1994a] and [McCloghrie and Farinacci 1994b] describe some experimental MIBs for multicast routers.

14.3   Multicast Output Processing Revisited

In Section 12.15 we described how an interface is selected for an outgoing multicast datagram. We saw that ip_output is passed an explicit interface in the ip_moptions structure, or ip_output looks up the destination group in the routing tables and uses the interface returned in the route entry.

If, after selecting an outgoing interface, ip_output loops back the datagram, it is queued for input processing on the interface selected for *output* and is considered for forwarding when it is processed by ipintr. Figure 14.6 illustrates this process.
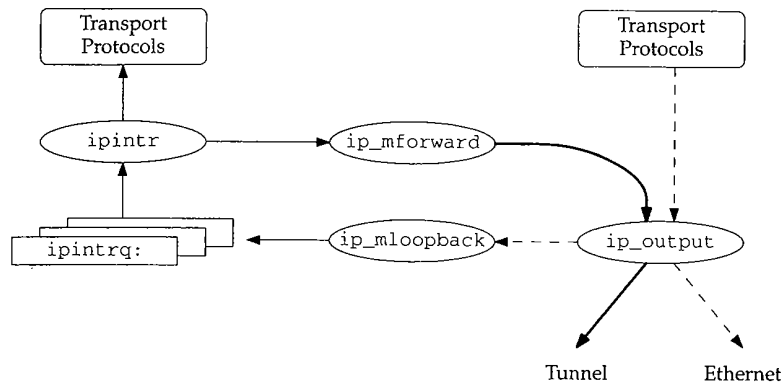
Figure 14.6   Multicast output processing with loopback.

In Figure 14.6 the dashed arrows represent the original outgoing datagram, which in this example is multicast on a local Ethernet. The copy created by `ip_mloopback` is represented by the thin arrows; this copy is passed to the transport protocols for input. The third copy is created when `ip_mforward` decides to forward the datagram through another interface on the system. The thickest arrows in Figure 14.6 represents the third copy, which in this example is sent on a multicast tunnel.

If the datagram is *not* looped back, `ip_output` passes it directly to `ip_mforward`, where it is duplicated and also processed as if it were received on the interface that `ip_output` selected. This process is shown in Figure 14.7.
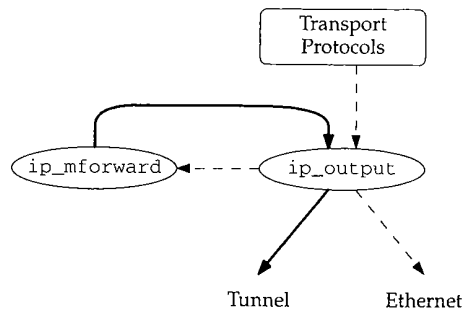
Figure 14.7   Multicast output processing with no loopback.

Whenever `ip_mforward` calls `ip_output` to send a multicast datagram, it sets the `IP_FORWARDING` flag so that `ip_output` does not pass the datagram back to `ip_mforward`, which would create an infinite loop.

`ip_mloopback` was described with Figure 12.42. `ip_mforward` is described in Section 14.8.

14.4 mrouted **Daemon**

Multicast routing is enabled and managed by a user-level process: the mrouted dae-
mon. mrouted implements the router portion of the IGMP protocol and communicates
with other multicast routers to implement multicast routing between networks. The
routing algorithms are implemented in mrouted, but the multicast routing tables are
maintained in the kernel, which forwards the datagrams.

In this text we describe only the kernel data structures and functions that support
mrouted—we do not describe mrouted itself. We describe the Truncated Reverse Path
Broadcast (TRPB) algorithm [Deering and Cheriton 1990], used to select routes for
multicast datagrams, and the Distance Vector Multicast Routing Protocol (DVMRP),
used to convey information between multicast routers, in enough detail to make sense
of the kernel multicast code.

RFC 1075 [Waitzman, Partridge, and Deering 1988] describes an old version of
DVMRP. mrouted implements a newer version of DVMRP, which is not yet docu-
mented in an RFC. The best documentation for the current algorithm and protocol is
the source code release for mrouted. Appendix B describes where the source code can
be obtained.

The mrouted daemon communicates with the kernel by setting options on an
IGMP socket (Chapter 32). The options are summarized in Figure 14.8.

optname	optval type	Function	Description
DVMRP_INIT		ip_mrouter_init	mrouted is starting
DVMRP_DONE		ip_mrouter_done	mrouted is shutting down
DVMRP_ADD_VIF	struct vifctl	add_vif	add virtual interface
DVMRP_DEL_VIF	vifi_t	del_vif	delete virtual interface
DVMRP_ADD_LGRP	struct lgrplctl	add_lgrp	add multicast group entry for an interface
DVMRP_DEL_LGRP	struct lgrplctl	del_lgrp	delete multicast group entry for an interface
DVMRP_ADD_MRT	struct mrtctl	add_mrt	add multicast route
DVMRP_DEL_MRT	struct in_addr	del_mrt	delete multicast route

Figure 14.8 Multicast routing socket options.

The socket options shown in Figure 14.8 are passed to rip_ctloutput (Section 32.8)
by the setsockopt system call. Figure 14.9 shows the portion of rip_ctloutput
that handles the DVMRP_*xxx* options.

173–187   When setsockopt is called, op equals PRCO_SETOPT and all the options are
passed to the ip_mrouter_cmd function. For the getsockopt system call, op equals
PRCO_GETOPT and EINVAL is returned for all the options.

Figure 14.10 shows the ip_mrouter_cmd function.

> These "options" are more like commands, since they cause the kernel to update various data
> structures. We use the term *command* throughout the rest of this chapter to emphasize this fact.

```
                                                                            raw_ip.c
173    case DVMRP_INIT:
174    case DVMRP_DONE:
175    case DVMRP_ADD_VIF:
176    case DVMRP_DEL_VIF:
177    case DVMRP_ADD_LGRP:
178    case DVMRP_DEL_LGRP:
179    case DVMRP_ADD_MRT:
180    case DVMRP_DEL_MRT:
181        if (op == PRCO_SETOPT) {
182            error = ip_mrouter_cmd(optname, so, *m);
183            if (*m)
184                (void) m_free(*m);
185        } else
186            error = EINVAL;
187        return (error);
                                                                            raw_ip.c
```

Figure 14.9   rip_ctloutput function: DVMRP_*xxx* socket options.

```
                                                                          ip_mroute.c
84  int
85  ip_mrouter_cmd(cmd, so, m)
86  int      cmd;
87  struct socket *so;
88  struct mbuf *m;
89  {
90      int      error = 0;

91      if (cmd != DVMRP_INIT && so != ip_mrouter)
92          error = EACCES;
93      else
94          switch (cmd) {

95          case DVMRP_INIT:
96              error = ip_mrouter_init(so);
97              break;

98          case DVMRP_DONE:
99              error = ip_mrouter_done();
100             break;

101         case DVMRP_ADD_VIF:
102             if (m == NULL || m->m_len < sizeof(struct vifctl))
103                     error = EINVAL;
104             else
105                 error = add_vif(mtod(m, struct vifctl *));
106             break;

107         case DVMRP_DEL_VIF:
108             if (m == NULL || m->m_len < sizeof(short))
109                     error = EINVAL;
110             else
111                 error = del_vif(mtod(m, vifi_t *));
112             break;
```

```
113         case DVMRP_ADD_LGRP:
114             if (m == NULL || m->m_len < sizeof(struct lgrplctl))
115                     error = EINVAL;
116             else
117                 error = add_lgrp(mtod(m, struct lgrplctl *));
118             break;

119         case DVMRP_DEL_LGRP:
120             if (m == NULL || m->m_len < sizeof(struct lgrplctl))
121                     error = EINVAL;
122             else
123                 error = del_lgrp(mtod(m, struct lgrplctl *));
124             break;

125         case DVMRP_ADD_MRT:
126             if (m == NULL || m->m_len < sizeof(struct mrtctl))
127                     error = EINVAL;
128             else
129                 error = add_mrt(mtod(m, struct mrtctl *));
130             break;

131         case DVMRP_DEL_MRT:
132             if (m == NULL || m->m_len < sizeof(struct in_addr))
133                     error = EINVAL;
134             else
135                 error = del_mrt(mtod(m, struct in_addr *));
136             break;

137         default:
138             error = EOPNOTSUPP;
139             break;
140         }
141     return (error);
142 }
```

——— *ip_mroute.c*

Figure 14.10 ip_mrouter_cmd function.

84–92   The first command issued by mrouted must be DVMRP_INIT. Subsequent commands must come from the same socket as the DVMRP_INIT command. EACCES is returned when other commands are issued on a different socket.

94–142   Each case in the switch checks to see if the right amount of data was included with the command and then calls the matching function. If the command is not recognized, EOPNOTSUPP is returned. Any error returned from the matching function is posted in error and returned at the end of the function.

Figure 14.11 shows ip_mrouter_init, which is called when mrouted issues the DVMRP_INIT command during initialization.

146–157   If the command is issued on something other than a raw IGMP socket, or if DVMRP_INIT has already been set, EOPNOTSUPP or EADDRINUSE are returned respectively. A pointer to the socket on which the initialization command is issued is saved in the global ip_mrouter. Subsequent commands must be issued on this socket. This prevents the concurrent operation of more than one instance of mrouted.

ip_mroute.c

```
146 static int
147 ip_mrouter_init(so)
148 struct socket *so;
149 {
150     if (so->so_type != SOCK_RAW ||
151         so->so_proto->pr_protocol != IPPROTO_IGMP)
152         return (EOPNOTSUPP);

153     if (ip_mrouter != NULL)
154         return (EADDRINUSE);

155     ip_mrouter = so;

156     return (0);
157 }
```

ip_mroute.c

Figure 14.11 ip_mrouter_init function: DVMRP_INIT command.

The remainder of the DVMRP_*xxx* commands are described in the following sections.

14.5 Virtual Interfaces

When operating as a multicast router, Net/3 accepts incoming multicast datagrams, duplicates them and forwards the copies through one or more interfaces. In this way, the datagram is forwarded to other multicast routers on the internet.

An outgoing interface can be a physical interface or it can be a multicast *tunnel*. Each end of the multicast tunnel is associated with a physical interface on a multicast router. Multicast tunnels allow two multicast routers to exchange multicast datagrams even when they are separated by routers that cannot forward multicast datagrams. Figure 14.12 shows two multicast routers connected by a multicast tunnel.
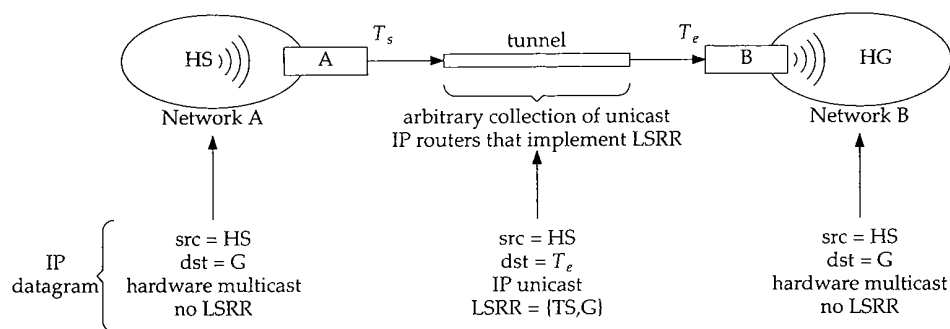
Figure 14.12 A multicast tunnel.

In Figure 14.12, the source host HS on network A is multicasting a datagram to group G. The only member of group G is on network B, which is connected to network A by a multicast tunnel. Router A receives the multicast (because multicast routers receive *all*

multicasts), consults its multicast routing tables, and forwards the datagram through the multicast tunnel.

The tunnel starts on the *physical* interface on router A identified by the IP unicast address T_s. The tunnel ends on the *physical* interface on router B identified by the IP unicast address, T_e. The tunnel itself is an arbitrarily complex collection of networks connected by IP unicast routers that implement the LSRR option. Figure 14.13 shows how an IP LSRR option implements the multicast tunnel.

System	IP header		Source route option		Description
	ip_src	ip_dst	*offset*	addresses	
HS	HS	G			on network A
T_s	HS	T_e	8	T_s • G	on tunnel
T_e	HS	G	12	T_s see text •	after ip_dooptions on router B
T_e	HS	G			after ip_mforward on router B

Figure 14.13   LSRR multicast tunnel options.

The first line of Figure 14.13 shows the datagram sent by HS as a multicast on network A. Router A receives the datagram because multicast routers receive all multicasts on their locally attached networks.

To send the datagram through the tunnel, router A inserts an LSRR option in the IP header. The second line shows the datagram as it leaves A on the tunnel. The first address in the LSRR option is the source address of the tunnel and the second address is the destination group. The destination of the datagram is T_e —the other end of the tunnel. The LSRR offset points to the *destination group*.

The tunneled datagram is forwarded through the internet until it reaches the other end of the tunnel on router B.

The third line of the figure shows the datagram after it is processed by ip_dooptions on router B. Recall from Chapter 9 that ip_dooptions processes the LSRR option before the destination address of the datagram is examined by ipintr. Since the destination address of the datagram (T_e) matches one of the interfaces on router B, ip_dooptions copies the address identified by the option offset (G in this example) into the destination field of the IP header. In the option, G is replaced with the address returned by ip_rtaddr, which normally selects the outgoing interface for the datagram based on the IP destination address (G in this case). This address is irrelevant, since ip_mforward discards the entire option. Finally, ip_dooptions advances the option offset.

The fourth line in Figure 14.13 shows the datagram after ipintr calls ip_mforward, where the LSRR option is recognized and removed from the datagram header. The resulting datagram looks like the original multicast datagram and is processed by ip_mforward, which in our example forwards it onto network B as a multicast datagram where it is received by HG.

Multicast tunnels constructed with LSRR options are obsolete. Since the March 1993 release of mrouted, tunnels have been constructed by prepending another IP header to the IP multicast datagram. The protocol in the new IP header is set to 4 to indicate that the contents of the packet is another IP packet. This value is documented

in RFC 1700 as the "IP in IP" protocol. LSRR tunnels are supported in newer versions of mrouted for backward compatibility.

Virtual Interface Table

For both physical interfaces and tunnel interfaces, the kernel maintains an entry in a *virtual interface* table, which contains information that is used only for multicasting. Each virtual interface is described by a vif structure (Figure 14.14). The global variable viftable is an array of these structures. An index to the table is stored in a vifi_t variable, which is an unsigned short integer.

```
                                                               ip_mroute.h
105 struct vif {
106     u_char  v_flags;            /* VIFF_ flags */
107     u_char  v_threshold;        /* min ttl required to forward on vif */
108     struct in_addr v_lcl_addr;  /* local interface address */
109     struct in_addr v_rmt_addr;  /* remote address (tunnels only) */
110     struct ifnet *v_ifp;        /* pointer to interface */
111     struct in_addr *v_lcl_grps; /* list of local grps (phyints only) */
112     int     v_lcl_grps_max;     /* malloc'ed number of v_lcl_grps */
113     int     v_lcl_grps_n;       /* used number of v_lcl_grps */
114     u_long  v_cached_group;     /* last grp looked-up (phyints only) */
115     int     v_cached_result;    /* last look-up result (phyints only) */
116 };
                                                               ip_mroute.h
```

Figure 14.14   vif structure.

105–110   The only flag defined for v_flags is VIFF_TUNNEL. When set, the interface is a tunnel to a remote multicast router. When not set, the interface is a physical interface on the local system. v_threshold is the multicast threshold, which we described in Section 12.9. v_lcl_addr is the unicast IP address of the local interface associated with this virtual interface. v_rmt_addr is the unicast IP address of the remote end of an IP multicast tunnel. Either v_lcl_addr or v_rmt_addr is nonzero, but never both. For physical interfaces, v_ifp is nonnull and points to the ifnet structure of the local interface. For tunnels, v_ifp is null.

111–116   The list of groups with members on the attached interface is kept as an array of IP multicast group addresses pointed to by v_lcl_grps, which is always null for tunnels. The size of the array is in v_lcl_grps_max, and the number of entries that are used is in v_lcl_grps_n. The array grows as needed to accommodate the group membership list. v_cached_group and v_cached_result implement a one-entry cache, which contain the group and result of the previous lookup.

Figure 14.15 illustrates the viftable, which has 32 (MAXVIFS) entries. viftable[2] is the last entry in use, so numvifs is 3. The size of the table is fixed when the kernel is compiled. Several members of the vif structure in the first entry of the table are shown. v_ifp points to an ifnet structure, v_lcl_grps points to an array of in_addr structures. The array has 32 (v_lcl_grps_max) entries, of which only 4 (v_lcl_grps_n) are in use.

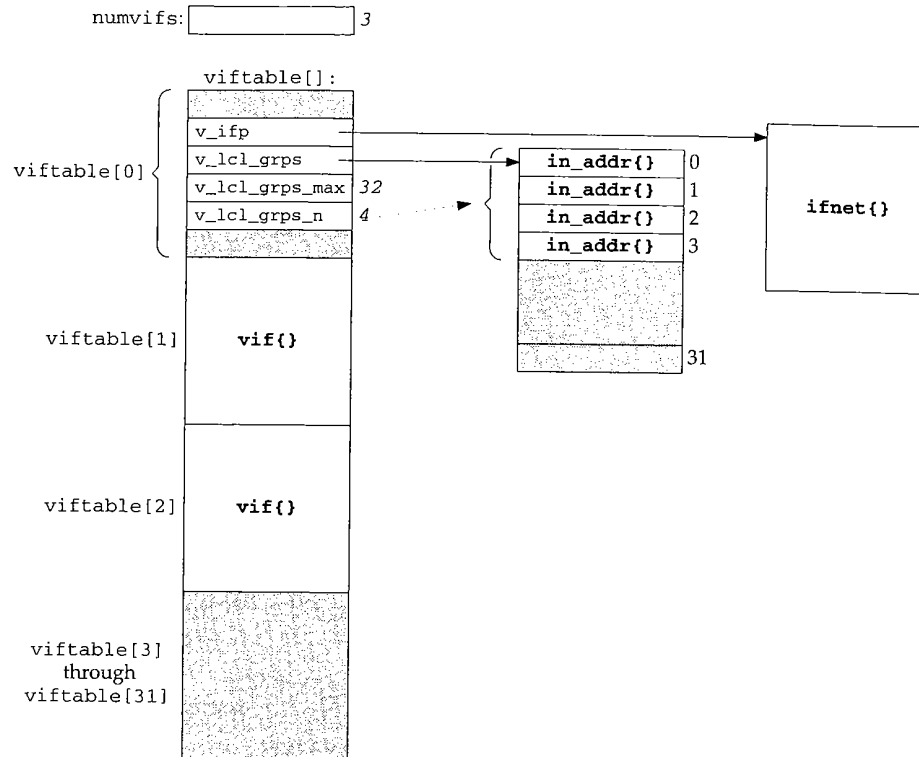**Figure 14.15** viftable array.

mrouted maintains viftable through the DVMRP_ADD_VIF and DVMRP_DEL_VIF commands. Normally all multicast-capable interfaces on the local system are added to the table when mrouted begins. Multicast tunnels are added when mrouted reads its configuration file, usually /etc/mrouted.conf. Commands in this file can also delete physical interfaces from the virtual interface table or change the multicast information associated with the interfaces.

A vifctl structure (Figure 14.16) is passed by mrouted to the kernel with the DVMRP_ADD_VIF command. It instructs the kernel to add an interface to the table of virtual interfaces.

```
                                                            ─────────────ip_mroute.h
76 struct vifctl {
77     vifi_t  vifc_vifi;          /* the index of the vif to be added */
78     u_char  vifc_flags;         /* VIFF_ flags (Figure 14.14) */
79     u_char  vifc_threshold;     /* min ttl required to forward on vif */
80     struct in_addr vifc_lcl_addr;   /* local interface address */
81     struct in_addr vifc_rmt_addr;   /* remote address (tunnels only) */
82 };
                                                            ─────────────ip_mroute.h
```

Figure 14.16 vifctl structure.

76–82   `vifc_vifi` identifies the index of the virtual interface within `viftable`. The remaining four members, `vifc_flags`, `vifc_threshold`, `vifc_lcl_addr`, and `vifc_rmt_addr`, are copied into the `vif` structure by the `add_vif` function.

`add_vif` Function

Figure 14.17 shows the `add_vif` function.

```
                                                                        ─ ip_mroute.c
202 static int
203 add_vif(vifcp)
204 struct vifctl *vifcp;
205 {
206     struct vif *vifp = viftable + vifcp->vifc_vifi;
207     struct ifaddr *ifa;
208     struct ifnet *ifp;
209     struct ifreq ifr;
210     int     error, s;
211     static struct sockaddr_in sin =
212     {sizeof(sin), AF_INET};

213     if (vifcp->vifc_vifi >= MAXVIFS)
214         return (EINVAL);
215     if (vifp->v_lcl_addr.s_addr != 0)
216         return (EADDRINUSE);

217     /* Find the interface with an address in AF_INET family */
218     sin.sin_addr = vifcp->vifc_lcl_addr;
219     ifa = ifa_ifwithaddr((struct sockaddr *) &sin);
220     if (ifa == 0)
221         return (EADDRNOTAVAIL);

222     s = splnet();

223     if (vifcp->vifc_flags & VIFF_TUNNEL)
224         vifp->v_rmt_addr = vifcp->vifc_rmt_addr;
225     else {
226         /* Make sure the interface supports multicast */
227         ifp = ifa->ifa_ifp;
228         if ((ifp->if_flags & IFF_MULTICAST) == 0) {
229             splx(s);
230             return (EOPNOTSUPP);
231         }
232         /*
233          * Enable promiscuous reception of all IP multicasts
234          * from the interface.
235          */
236         satosin(&ifr.ifr_addr)->sin_family = AF_INET;
237         satosin(&ifr.ifr_addr)->sin_addr.s_addr = INADDR_ANY;
238         error = (*ifp->if_ioctl) (ifp, SIOCADDMULTI, (caddr_t) & ifr);
239         if (error) {
240             splx(s);
241             return (error);
242         }
243     }
```

```
244        vifp->v_flags = vifcp->vifc_flags;
245        vifp->v_threshold = vifcp->vifc_threshold;
246        vifp->v_lcl_addr = vifcp->vifc_lcl_addr;
247        vifp->v_ifp = ifa->ifa_ifp;

248        /* Adjust numvifs up if the vifi is higher than numvifs */
249        if (numvifs <= vifcp->vifc_vifi)
250            numvifs = vifcp->vifc_vifi + 1;

251        splx(s);
252        return (0);
253 }
```
                                                                                    ─── *ip_mroute.c*

Figure 14.17 add_vif function: DVMRP_ADD_VIF command.

Validate index

202-216   If the table index specified by mrouted in vifc_vifi is too large, or the table entry is already in use, EINVAL or EADDRINUSE is returned respectively.

Locate physical interface

217-221   ifa_ifwithaddr takes the unicast IP address in vifc_lcl_addr and returns a pointer to the associated ifnet structure. This identifies the physical interface to be used for this virtual interface. If there is no matching interface, EADDRNOTAVAIL is returned.

Configure tunnel interface

222-224   For a tunnel, the remote end of the tunnel is copied from the vifctl structure to the vif structure in the interface table.

Configure physical interface

225-243   For a physical interface, the link-level driver must support multicasting. The SIOCADDMULTI command used with INADDR_ANY configures the interface to begin receiving *all* IP multicast datagrams (Figure 12.32) because it is a multicast router. Incoming datagrams are forwarded when ipintr passes them to ip_mforward.

Save multicast information

244-253   The remaining interface information is copied from the vifctl structure to the vif structure. If necessary, numvifs is updated to record the number of virtual interfaces in use.

del_vif Function

The function del_vif, shown in Figure 14.18, deletes entries from the virtual interface table. It is called when mrouted sets the DVMRP_DEL_VIF command.

Validate index

257-268   If the index passed to del_vif is greater than the largest index in use or it references an entry that is not in use, EINVAL or EADDRNOTAVAIL is returned respectively.

```
                                                                              ——ip_mroute.c
257 static int
258 del_vif(vifip)
259 vifi_t *vifip;
260 {
261     struct vif *vifp = viftable + *vifip;
262     struct ifnet *ifp;
263     int    i, s;
264     struct ifreq ifr;

265     if (*vifip >= numvifs)
266         return (EINVAL);
267     if (vifp->v_lcl_addr.s_addr == 0)
268         return (EADDRNOTAVAIL);

269     s = splnet();

270     if (!(vifp->v_flags & VIFF_TUNNEL)) {
271         if (vifp->v_lcl_grps)
272             free(vifp->v_lcl_grps, M_MRTABLE);
273         satosin(&ifr.ifr_addr)->sin_family = AF_INET;
274         satosin(&ifr.ifr_addr)->sin_addr.s_addr = INADDR_ANY;
275         ifp = vifp->v_ifp;
276         (*ifp->if_ioctl) (ifp, SIOCDELMULTI, (caddr_t) & ifr);
277     }
278     bzero((caddr_t) vifp, sizeof(*vifp));

279     /* Adjust numvifs down */
280     for (i = numvifs - 1; i >= 0; i--)
281         if (viftable[i].v_lcl_addr.s_addr != 0)
282             break;
283     numvifs = i + 1;

284     splx(s);
285     return (0);
286 }
                                                                              ——ip_mroute.c
```

Figure 14.18 del_vif function: DVMRP_DEL_VIF command.

Delete interface

269–278   For a physical interface, the local group table is released, and the reception of all multicast datagrams is disabled by SIOCDELMULTI. The entry in viftable is cleared by bzero.

Adjust interface count

279–286   The for loop searches for the first active entry in the table starting at the largest previously active entry and working back toward the first entry. For unused entries, the s_addr member of v_lcl_addr (an in_addr structure) is 0. numvifs is updated accordingly and the function returns.

14.6 IGMP Revisited

Chapter 13 focused on the host part of the IGMP protocol. `mrouted` implements the router portion of this protocol. For every physical interface, `mrouted` must keep track of which multicast groups have members on the attached network. `mrouted` multicasts an `IGMP_HOST_MEMBERSHIP_QUERY` datagram every 120 seconds and compiles the resulting `IGMP_HOST_MEMBERSHIP_REPORT` datagrams into a membership array associated with each network. This array is *not* the same as the membership list we described in Chapter 13.

From the information collected, `mrouted` constructs the multicast routing tables. The list of groups is also used to suppress multicasts to areas of the multicast internet that do not have members of the destination group.

The membership array is maintained only for physical interfaces. Tunnels are point-to-point interfaces to another multicast router, so no group membership information is needed.

We saw in Figure 14.14 that `v_lcl_grps` points to an array of IP multicast groups. `mrouted` maintains this list with the `DVMRP_ADD_LGRP` and `DVMRP_DEL_LGRP` commands. An `lgrplctl` (Figure 14.19) structure is passed with both commands.

```
                                                                    ─ip_mroute.h
87  struct lgrplctl {
88      vifi_t  lgc_vifi;
89      struct in_addr lgc_gaddr;
90  };
                                                                    ─ip_mroute.h
```

Figure 14.19 `lgrplctl` structure.

87–90   The {interface, group} pair is identified by `lgc_vifi` and `lgc_gaddr`. The interface index (`lgc_vifi`, an unsigned short) identifies a *virtual* interface, not a physical interface.

When an `IGMP_HOST_MEMBERSHIP_REPORT` datagram is received, the functions shown in Figure 14.20 are called.

add_lgrp Function

`mrouted` examines the source address of an incoming IGMP report to determine which subnet and therefore which interface the report arrived on. Based on this information, `mrouted` sets the `DVMRP_ADD_LGRP` command for the interface to update the membership table in the kernel. This information is also fed into the multicast routing algorithm to update the routing tables. Figure 14.21 shows the `add_lgrp` function.

Validate add request

291–301   If the request identifies an invalid interface, `EINVAL` is returned. If the interface is not in use or is a tunnel, `EADDRNOTAVAIL` is returned.

If needed, expand group array

302–326   If the new group won't fit in the current group array, a new array is allocated. The first time `add_lgrp` is called for an interface, an array is allocated to hold 32 groups.
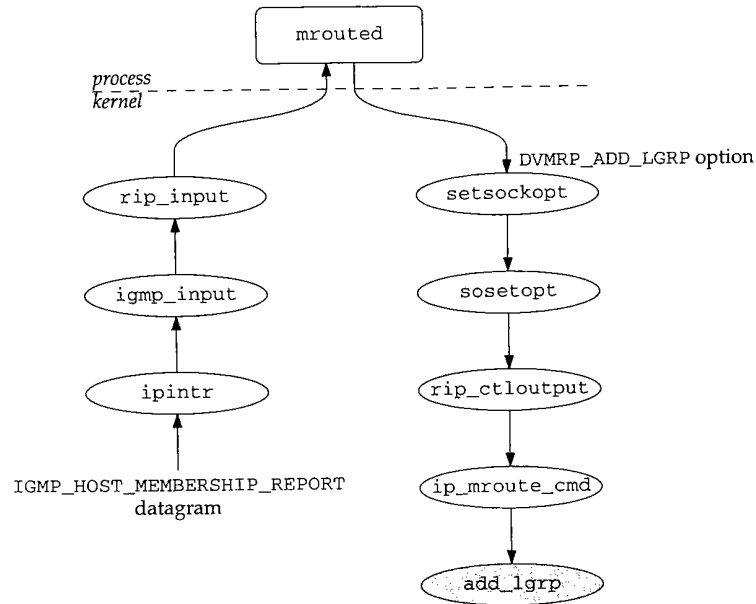
Figure 14.20   IGMP report processing.

Each time the array fills, add_lgrp allocates a new array of twice the previous size. The new array is allocated by malloc, cleared by bzero, and filled by copying the old array into the new one with bcopy. The maximum number of entries, v_lcl_grps_max, is updated, the old array (if any) is released, and the new array is attached to the vif entry with v_lcl_grps.

> The "paranoid" comment points out there is no guarantee that the memory allocated by malloc contains all 0s.

Add new group

327–332   The new group is copied into the next available entry and if the cache already contains the new group, the cache is marked as valid.

The lookup cache contains an address, v_cached_group, and a cached lookup result, v_cached_result. The grplst_member function always consults the cache before searching the membership array. If the given group matches v_cached_group, the cached result is returned; otherwise the membership array is searched.

del_lgrp Function

Group information is expired for each interface when no membership report has been received for the group within 270 seconds. mrouted maintains the appropriate timers and issues the DVMRP_DEL_LGRP command when the information expires. Figure 14.22 shows del_lgrp.

```
                                                                 ——————————— ip_mroute.c
291 static int
292 add_lgrp(gcp)
293 struct lgrplctl *gcp;
294 {
295     struct vif *vifp;
296     int     s;

297     if (gcp->lgc_vifi >= numvifs)
298         return (EINVAL);

299     vifp = viftable + gcp->lgc_vifi;
300     if (vifp->v_lcl_addr.s_addr == 0 || (vifp->v_flags & VIFF_TUNNEL))
301         return (EADDRNOTAVAIL);

302     /* If not enough space in existing list, allocate a larger one */
303     s = splnet();
304     if (vifp->v_lcl_grps_n + 1 >= vifp->v_lcl_grps_max) {
305         int     num;
306         struct in_addr *ip;

307         num = vifp->v_lcl_grps_max;
308         if (num <= 0)
309             num = 32;            /* initial number */
310         else
311             num += num;          /* double last number */
312         ip = (struct in_addr *) malloc(num * sizeof(*ip),
313                                       M_MRTABLE, M_NOWAIT);
314         if (ip == NULL) {
315             splx(s);
316             return (ENOBUFS);
317         }
318         bzero((caddr_t) ip, num * sizeof(*ip));      /* XXX paranoid */
319         bcopy((caddr_t) vifp->v_lcl_grps, (caddr_t) ip,
320               vifp->v_lcl_grps_n * sizeof(*ip));

321         vifp->v_lcl_grps_max = num;
322         if (vifp->v_lcl_grps)
323             free(vifp->v_lcl_grps, M_MRTABLE);
324         vifp->v_lcl_grps = ip;

325         splx(s);
326     }
327     vifp->v_lcl_grps[vifp->v_lcl_grps_n++] = gcp->lgc_gaddr;

328     if (gcp->lgc_gaddr.s_addr == vifp->v_cached_group)
329         vifp->v_cached_result = 1;

330     splx(s);
331     return (0);
332 }
                                                                 ——————————— ip_mroute.c
```

Figure 14.21   add_lgrp function: process DVMRP_ADD_LGRP command.

ip_mroute.c

```
337 static int
338 del_lgrp(gcp)
339 struct lgrplctl *gcp;
340 {
341     struct vif *vifp;
342     int     i, error, s;

343     if (gcp->lgc_vifi >= numvifs)
344         return (EINVAL);
345     vifp = viftable + gcp->lgc_vifi;
346     if (vifp->v_lcl_addr.s_addr == 0 || (vifp->v_flags & VIFF_TUNNEL))
347         return (EADDRNOTAVAIL);

348     s = splnet();

349     if (gcp->lgc_gaddr.s_addr == vifp->v_cached_group)
350         vifp->v_cached_result = 0;

351     error = EADDRNOTAVAIL;
352     for (i = 0; i < vifp->v_lcl_grps_n; ++i)
353         if (same(&gcp->lgc_gaddr, &vifp->v_lcl_grps[i])) {
354             error = 0;
355             vifp->v_lcl_grps_n--;
356             bcopy((caddr_t) & vifp->v_lcl_grps[i + 1],
357                     (caddr_t) & vifp->v_lcl_grps[i],
358                     (vifp->v_lcl_grps_n - i) * sizeof(struct in_addr));
359             error = 0;
360             break;
361         }
362     splx(s);
363     return (error);
364 }
```

ip_mroute.c

Figure 14.22   del_lgrp function: process DVMRP_DEL_LGRP command.

Validate interface index

337–347   If the request identifies an invalid interface, EINVAL is returned. If the interface is not in use or is a tunnel, EADDRNOTAVAIL is returned.

Update lookup cache

348–350   If the group to be deleted is in the cache, the lookup result is set to 0 (false).

Delete group

351–364   EADDRNOTAVAIL is posted in error in case the group is not found in the membership list. The for loop searches the membership array associated with the interface. If same (a macro that uses bcmp to compare the two addresses) is true, error is cleared and the group count is decremented. bcopy shifts the subsequent array entries down to delete the group and del_lgrp breaks out of the loop.

   If the loop completes without finding a match, EADDRNOTAVAIL is returned; otherwise 0 is returned.

grplst_member **Function**

During multicast forwarding, the membership array is consulted to avoid sending datagrams on a network when no member of the destination group is present. grplst_member, shown in Figure 14.23, searches the list looking for the given group address.

```
                                                                    ip_mroute.c
368 static int
369 grplst_member(vifp, gaddr)
370 struct vif *vifp;
371 struct in_addr gaddr;
372 {
373     int     i, s;
374     u_long  addr;

375     mrtstat.mrts_grp_lookups++;

376     addr = gaddr.s_addr;
377     if (addr == vifp->v_cached_group)
378         return (vifp->v_cached_result);

379     mrtstat.mrts_grp_misses++;

380     for (i = 0; i < vifp->v_lcl_grps_n; ++i)
381         if (addr == vifp->v_lcl_grps[i].s_addr) {
382             s = splnet();
383             vifp->v_cached_group = addr;
384             vifp->v_cached_result = 1;
385             splx(s);
386             return (1);
387         }
388     s = splnet();
389     vifp->v_cached_group = addr;
390     vifp->v_cached_result = 0;
391     splx(s);
392     return (0);
393 }
                                                                    ip_mroute.c
```

Figure 14.23 grplst_member function.

Check the cache

368–379   If the requested group is located in the cache, the cached result is returned and the membership array is not searched.

Search the membership array

380–393   A linear search determines if the group is in the array. If it is found, the cache is updated to record the match and one is returned. If it is not found, the cache is updated to record the miss and 0 is returned.

14.7 Multicast Routing

As we mentioned at the start of this chapter, we will not be presenting the TRPB algorithm implemented by `mrouted`, but we do need to provide a general overview of the mechanism to describe the multicast routing table and the multicast routing functions in the kernel. Figure 14.24 shows the sample multicast network that we use to illustrate the algorithms.
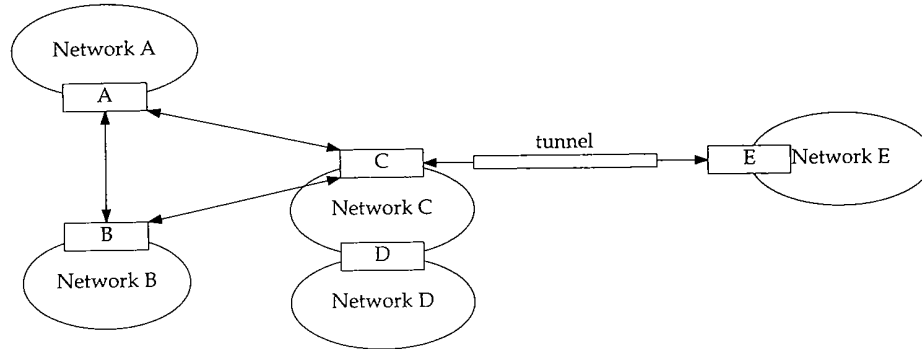

Figure 14.24   Sample multicast network.

In Figure 14.24, routers are shown as boxes and the ellipses are the multicast networks attached to the routers. For example, router D can multicast on network D and C. Router C can multicast to network C, to routers A and .B through point-to-point interfaces, and to E through a multicast tunnel.

The simplest approach to multicast routing is to select a subset of the internet topology that forms a *spanning tree*. If each router forwards multicasts along the spanning tree, every router eventually receives the datagram. Figure 14.25 shows one spanning tree for our sample network, where host S on network A represents the source of a multicast datagram.

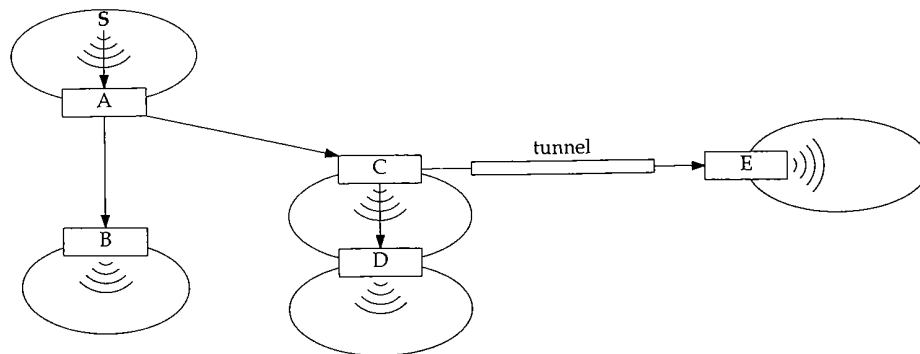For a discussion of spanning trees, see [Tanenbaum 1989] or [Perlman 1992].

Figure 14.25   Spanning tree for network A.

We constructed the tree based on the shortest *reverse path* from every network back to the source in network A. In Figure 14.25, the link between routers B and C is omitted to form the spanning tree. The arrows between the source and router A, and between router C and D, emphasize that the multicast network is part of the spanning tree.

If the same spanning tree were used to forward a datagram from network C, the datagram would be forwarded along a longer path than needed to get to a recipient on network B. The algorithm described in RFC 1075 computes a separate spanning tree for each potential source network to avoid this problem. The routing tables contain a network number and subnet mask for each route, so that a single route applies to any host within the source subnet.

Because each spanning tree is constructed to provide the shortest reverse path to the source of the datagram, and every network receives every multicast datagram, this process is called *reverse path broadcasting* or RPB.

The RPB protocol has no knowledge of multicast group membership, so many datagrams are unnecessarily forwarded to networks that have no members in the destination group. If, in addition to computing the spanning trees, the routing algorithm records which networks are *leaves* and is aware of the group membership on each network, then routers attached to leaf networks can avoid forwarding datagrams onto the network when there there is no member of the destination group present. This is called *truncated reverse path broadcasting* (TRPB), and is implemented by version 2.0 of `mrouted` with the help of IGMP to keep track of membership in the leaf networks.

Figure 14.26 shows TRPB applied to a multicast sent from a source on network C and with a member of the destination group on network B.
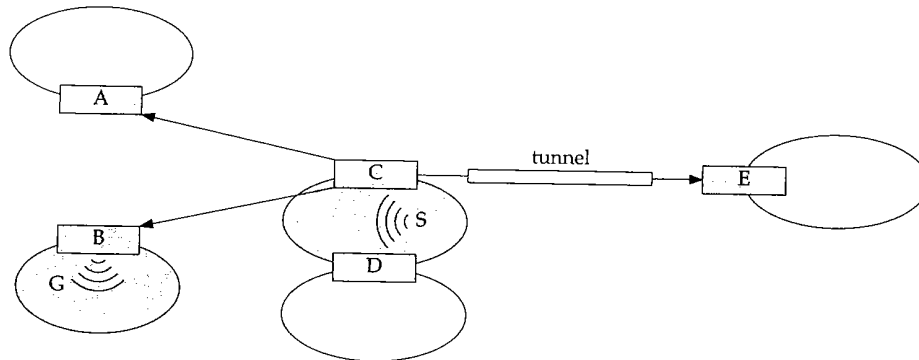

Figure 14.26   TRPB routing for network C.

We'll use Figure 14.26 to illustrate the terms used in the Net/3 multicast routing table. In this example, the shaded networks and routers receive a copy of the multicast datagram sent from the source on network C. The link between A and B is not part of the spanning tree and C does not have a link to D, since the multicast sent by the source is received directly by C and D.

In this figure, networks A, B, D, and E are leaf networks. Router C receives the multicast and forwards it through the interfaces attached to routers A, B, and E—even

though sending it to A and E is wasted effort. This is a major weakness of the TRPB algorithm.

The interface associated with network C on router C is called the *parent* because it is the interface on which router C expects to receive multicasts originating from network C. The interfaces from router C to routers A, B, and E, are *child* interfaces. For router A, the point-to-point interface is the parent for the source packets from C and the interface for network A is a child. Interfaces are identified as a parent or as a child relative to the source of the datagram. Multicast datagrams are forwarded only to the associated child interfaces, and never to the parent interface.

Continuing with the example, networks A, D, and E are not shaded because they are leaf networks without members of the destination group, so the spanning tree is truncated at the routers and the datagram is not forwarded onto these networks. Router B forwards the datagram onto network B, since there is a member of the destination group on the network. To implement the truncation algorithm, each multicast router that receives the datagram consults the group table associated with every virtual interface in the router's `viftable`.

The final refinement to the multicast routing algorithm is called *reverse path multicasting* (RPM). The goal of RPM is to *prune* each spanning tree and avoid sending datagrams along branches of the tree that do not contain a member of the destination group. In Figure 14.26, RPM would prevent router C from sending a datagram to A and E, since there is no member of the destination group in those branches of the tree. Version 3.3 of `mrouted` implements RPM.

Figure 14.27 shows our example network, but this time only the routers and networks reached when the datagram is routed by RPM are shaded.
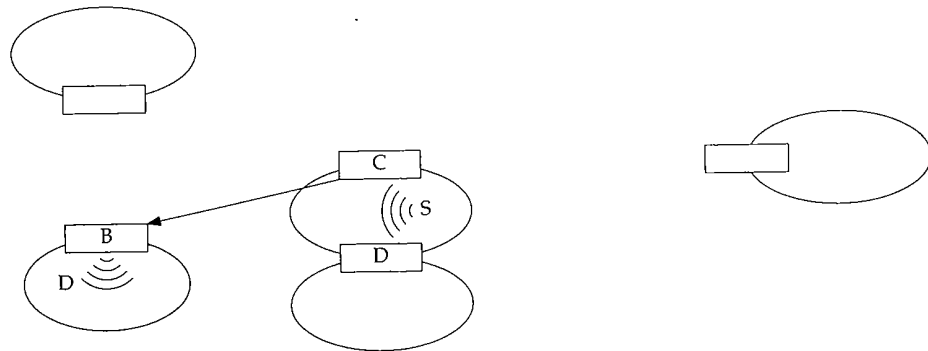

Figure 14.27   RPM routing for network C.

To compute the routing tables corresponding to the spanning trees we described, the multicast routers communicate with adjacent multicast routers to discover the multicast internet topology and the location of multicast group members. In Net/3, DVMRP is used for this communication. DVMRP messages are transmitted as IGMP datagrams and are sent to the multicast group 224.0.0.4, which is reserved for DVMRP communication (Figure 12.1).

In Figure 12.39, we saw that incoming IGMP packets are always accepted by a

multicast router. They are passed to `igmp_input`, to `rip_input`, and then read by `mrouted` on a raw IGMP socket. `mrouted` sends DVMRP messages to other multicast routers on the same raw IGMP socket.

For more information about RPB, TRPB, RPM, and the DVMRP messages that are needed to implement these algorithms, see [Deering and Cheriton 1990] and the source code release of `mrouted`.

There are other multicast routing protocols in use on the Internet. Proteon routers implement the MOSPF protocol described in RFC 1584 [Moy 1994]. PIM (Protocol Independent Multicasting) is implemented by Cisco routers, starting with Release 10.2 of their operating software. PIM is described in [Deering et al. 1994].

Multicast Routing Table

We can now describe the implementation of the multicast routing tables in Net/3. The kernel's multicast routing table is maintained as a hash table with 64 entries (`MRTHASHSIZ`). The table is kept in the global array `mrttable`, and each entry points to a linked list of `mrt` structures, shown in Figure 14.28.

```
                                                              ── ip_mroute.h
120  struct mrt {
121      struct in_addr mrt_origin;     /* subnet origin of multicasts */
122      struct in_addr mrt_originmask;  /* subnet mask for origin */
123      vifi_t   mrt_parent;           /* incoming vif */
124      vifbitmap_t mrt_children;      /* outgoing children vifs */
125      vifbitmap_t mrt_leaves;        /* subset of outgoing children vifs */
126      struct mrt *mrt_next;          /* forward link */
127  };
                                                              ── ip_mroute.h
```

Figure 14.28   `mrt` structure.

120–127   `mrtc_origin` and `mrtc_originmask` identify an entry in the table. `mrtc_parent` is the index of the virtual interface on which all multicast datagrams from the origin are expected. The outgoing interfaces are identified within `mrtc_children`, which is a bitmap. Outgoing interfaces that are also leaves in the multicast routing tree are identified in `mrtc_leaves`, which is also a bitmap. The last member, `mrt_next`, implements a linked list in case multiple routes hash to the same array entry.

Figure 14.29 shows the organization of the multicast routing table. Each `mrt` structure is placed in the hash chain that corresponds to return value from the `nethash` function shown in Figure 14.31.

The multicast routing table maintained by the kernel is a subset of the routing table maintained within `mrouted` and contains enough information to support multicast forwarding within the kernel. Updates to the kernel table are sent with the `DVMRP_ADD_MRT` command, which includes the `mrtctl` structure shown in Figure 14.30.
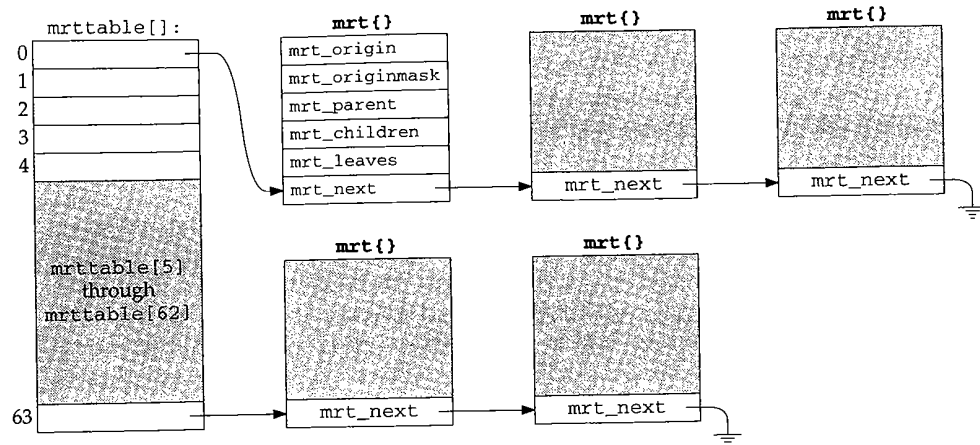
Figure 14.29 Multicast routing table.

ip_mroute.h
```
 95 struct mrtctl {
 96      struct in_addr mrtc_origin;  /* subnet origin of multicasts */
 97      struct in_addr mrtc_originmask;    /* subnet mask for origin */
 98      vifi_t  mrtc_parent;         /* incoming vif */
 99      vifbitmap_t mrtc_children;   /* outgoing children vifs */
100      vifbitmap_t mrtc_leaves;     /* subset of outgoing children vifs */
101 };
```
ip_mroute.h

Figure 14.30 mrtctl structure.

95–101   The five members of the mrtctl structure carry the information we have already described (Figure 14.28) between mrouted and the kernel.

The multicast routing table is keyed by the source IP address of the multicast datagram. nethash (Figure 14.31) implements the hashing algorithm used for the table. It accepts the source IP address and returns a value between 0 and 63 (MRTHASHSIZ − 1).

ip_mroute.c
```
398 static  u_long
399 nethash(in)
400 struct in_addr in;
401 {
402     u_long  n;

403     n = in_netof(in);
404     while ((n & 0xff) == 0)
405         n >>= 8;
406     return (MRTHASHMOD(n));
407 }
```
ip_mroute.c

Figure 14.31 nethash function.

398–407   in_netof returns in with the host portion set to all 0s leaving only the class A, B, or C network of the sending host in n. The result is shifted to the right until the low-order 8 bits are nonzero. MRTHASHMOD is

```
#define MRTHASHMOD(h)    ((h) & (MRTHASHSIZ - 1))
```

The low-order 8 bits are logically ANDed with 63, leaving only the low-order 6 bits, which is an integer in the range 0 to 63.

> Doing two function calls (nethash and in_netof) to calculate a hash value is an expensive algorithm to compute a hash for a 32-bit address.

del_mrt Function

The mrouted daemon adds and deletes entries in the kernel's multicast routing table through the DVMRP_ADD_MRT and DVMRP_DEL_MRT commands. Figure 14.32 shows the del_mrt function.

```
                                                              ip_mroute.c
451 static int
452 del_mrt(origin)
453 struct in_addr *origin;
454 {
455     struct mrt *rt, *prev_rt;
456     u_long  hash = nethash(*origin);
457     int     s;

458     for (prev_rt = rt = mrttable[hash]; rt; prev_rt = rt, rt = rt->mrt_next)
459         if (origin->s_addr == rt->mrt_origin.s_addr)
460             break;
461     if (!rt)
462         return (ESRCH);

463     s = splnet();

464     if (rt == cached_mrt)
465         cached_mrt = NULL;

466     if (prev_rt == rt)
467         mrttable[hash] = rt->mrt_next;
468     else
469         prev_rt->mrt_next = rt->mrt_next;
470     free(rt, M_MRTABLE);

471     splx(s);
472     return (0);
473 }
                                                              ip_mroute.c
```

Figure 14.32   del_mrt function: process DVMRP_DEL_MRT command.

Find route entry

451–462   The for loop starts at the entry identified by hash (initialized in its declaration from nethash). If the entry is not located, ESRCH is returned.

Delete route entry

463–473   If the entry was stored in the cache, the cache is invalidated. The entry is unlinked from the hash chain and released. The `if` statement is needed to handle the special case when the matched entry is at the front of the list.

add_mrt Function

The `add_mrt` function is shown in Figure 14.33.

―― *ip_mroute.c*
```
411 static int
412 add_mrt(mrtcp)
413 struct mrtctl *mrtcp;
414 {
415     struct mrt *rt;
416     u_long  hash;
417     int     s;

418     if (rt = mrtfind(mrtcp->mrtc_origin)) {
419         /* Just update the route */
420         s = splnet();
421         rt->mrt_parent = mrtcp->mrtc_parent;
422         VIFM_COPY(mrtcp->mrtc_children, rt->mrt_children);
423         VIFM_COPY(mrtcp->mrtc_leaves, rt->mrt_leaves);
424         splx(s);
425         return (0);
426     }
427     s = splnet();

428     rt = (struct mrt *) malloc(sizeof(*rt), M_MRTABLE, M_NOWAIT);
429     if (rt == NULL) {
430         splx(s);
431         return (ENOBUFS);
432     }
433     /*
434      * insert new entry at head of hash chain
435      */
436     rt->mrt_origin = mrtcp->mrtc_origin;
437     rt->mrt_originmask = mrtcp->mrtc_originmask;
438     rt->mrt_parent = mrtcp->mrtc_parent;
439     VIFM_COPY(mrtcp->mrtc_children, rt->mrt_children);
440     VIFM_COPY(mrtcp->mrtc_leaves, rt->mrt_leaves);
441     /* link into table */
442     hash = nethash(mrtcp->mrtc_origin);
443     rt->mrt_next = mrttable[hash];
444     mrttable[hash] = rt;

445     splx(s);
446     return (0);
447 }
```
―― *ip_mroute.c*

Figure 14.33   `add_mrt` function: process DVMRP_ADD_MRT command.

Update existing route

411–427   If the requested route is already in the routing table, the new information is copied
into the route and add_mrt returns.

Allocate new route

428–447   An mrt structure is constructed in a newly allocated mbuf with the information
from mrtctl structure passed with the add request. The hash index is computed from
mrtc_origin, and the new route is inserted as the first entry on the hash chain.

mrtfind Function

The multicast routing table is searched with the mrtfind function. The source of the
datagram is passed to mrtfind, which returns a pointer to the matching mrt structure,
or a null pointer if there is no match.

```
                                                                  ip_mroute.c
477 static struct mrt *
478 mrtfind(origin)
479 struct in_addr origin;
480 {
481     struct mrt *rt;
482     u_int   hash;
483     int     s;

484     mrtstat.mrts_mrt_lookups++;

485     if (cached_mrt != NULL &&
486         (origin.s_addr & cached_originmask) == cached_origin)
487         return (cached_mrt);

488     mrtstat.mrts_mrt_misses++;

489     hash = nethash(origin);
490     for (rt = mrttable[hash]; rt; rt = rt->mrt_next)
491         if ((origin.s_addr & rt->mrt_originmask.s_addr) ==
492             rt->mrt_origin.s_addr) {
493             s = splnet();
494             cached_mrt = rt;
495             cached_origin = rt->mrt_origin.s_addr;
496             cached_originmask = rt->mrt_originmask.s_addr;
497             splx(s);
498             return (rt);
499         }
500     return (NULL);
501 }
                                                                  ip_mroute.c
```

Figure 14.34 mrtfind function.

Check route lookup cache

477–488   The given source IP address (origin) is logically ANDed with the origin mask in
the cache. If the result matches cached_origin, the cached entry is returned.

Check the hash table

489-501   `nethash` returns the hash index for the route entry. The `for` loop searches the hash chain for a matching route. When a match is found, the cache is updated and a pointer to the route is returned. If a match is not found, a null pointer is returned.

14.8 Multicast Forwarding: `ip_mforward` Function

Multicast forwarding is implemented entirely in the kernel. We saw in Figure 12.39 that `ipintr` passes incoming multicast datagrams to `ip_mforward` when `ip_mrouter` is nonnull, that is, when `mrouted` is running.

We also saw in Figure 12.40 that `ip_output` can pass multicast datagrams that originate on the local host to `ip_mforward` to be routed to interfaces other than the one interface selected by `ip_output`.

Unlike unicast forwarding, each time a multicast datagram is forwarded to an interface, a copy is made. For example, if the local host is acting as a multicast router and is connected to three different networks, multicast datagrams originating on the system are duplicated and queued for *output* on all three interfaces. Additionally, the datagram may be duplicated and queued for *input* if the multicast loopback flag was set by the application or if any of the outgoing interfaces receive their own transmissions.

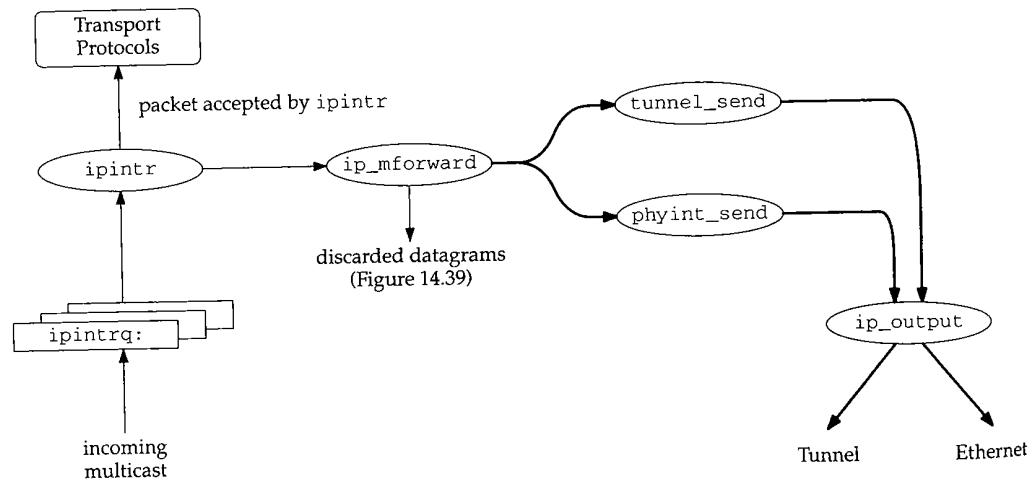Figure 14.35 shows a multicast datagram arriving on a physical interface.

Figure 14.35 Multicast datagram arriving on physical interface.

In Figure 14.35, the interface on which the datagram arrived is a member of the destination group, so the datagram is passed to the transport protocols for input processing. The datagram is also passed to `ip_mforward`, where it is duplicated and

forwarded to a physical interface and to a tunnel (the thick arrows), both of which must be different from the receiving interface.

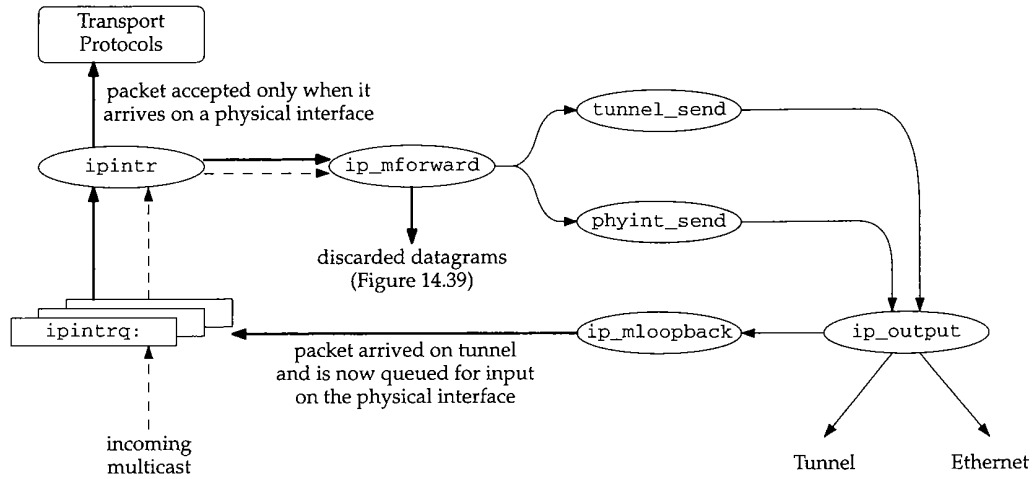Figure 14.36 shows a multicast datagram arriving on a tunnel.

Figure 14.36   Multicast datagram arriving on a multicast tunnel.

In Figure 14.36, the datagram arriving on a physical interface associated with the local end of the tunnel is represented by the dashed arrows. It is passed to `ip_mforward`, which as we'll see in Figure 14.37 returns a nonzero value because the packet arrived on a tunnel. This causes `ipintr` to not pass the packet to the transport protocols.

`ip_mforward` strips the tunnel options from the packet, consults the multicast routing table, and, in this example, forwards the packet on another tunnel and on the same *physical* interface on which it arrived, as shown by the thin arrows. This is OK because the multicast routing tables are based on the *virtual* interfaces, not the physical interfaces.

In Figure 14.36 we assume that the physical interface is a member of the destination group, so `ip_output` passes the datagram to `ip_mloopback`, which queues it for processing by `ipintr` (the thick arrows). The packet is passed to `ip_mforward` again, where it is discarded (Exercise 14.4). `ip_mforward` returns 0 this time (because the packet arrived on a physical interface), so `ipintr` considers and accepts the datagram for input processing.

We show the multicast forwarding code in three parts:

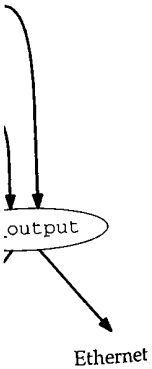- tunnel input processing (Figure 14.37),
- forwarding eligibility (Figure 14.39), and
- forward to outgoing interfaces (Figure 14.40).

                                                                                    ip_mroute.c
```
516 int
517 ip_mforward(m, ifp)
518 struct mbuf *m;
519 struct ifnet *ifp;
520 {
521     struct ip *ip = mtod(m, struct ip *);
522     struct mrt *rt;
523     struct vif *vifp;
524     int     vifi;
525     u_char *ipoptions;
526     u_long  tunnel_src;

527     if (ip->ip_hl < (IP_HDR_LEN + TUNNEL_LEN) >> 2 ||
528         (ipoptions = (u_char *) (ip + 1))[1] != IPOPT_LSRR) {
529         /* Packet arrived via a physical interface. */
530         tunnel_src = 0;
531     } else {
532         /*
533          * Packet arrived through a tunnel.
534          * A tunneled packet has a single NOP option and a
535          * two-element loose-source-and-record-route (LSRR)
536          * option immediately following the fixed-size part of
537          * the IP header.  At this point in processing, the IP
538          * header should contain the following IP addresses:
539          *
540          * original source        - in the source address field
541          * destination group      - in the destination address field
542          * remote tunnel end-point - in the first  element.of LSRR
543          * one of this host's addrs - in the second element of LSRR
544          *
545          * NOTE: RFC-1075 would have the original source and
546          * remote tunnel end-point addresses swapped.  However,
547          * that could cause delivery of ICMP error messages to
548          * innocent applications on intermediate routing
549          * hosts!  Therefore, we hereby change the spec.
550          */
551         /* Verify that the tunnel options are well-formed.  */
552         if (ipoptions[0] != IPOPT_NOP ||
553             ipoptions[2] != 11 ||   /* LSRR option length   */
554             ipoptions[3] != 12 ||   /* LSRR address pointer */
555             (tunnel_src = *(u_long *) (&ipoptions[4])) == 0) {
556             mrtstat.mrts_bad_tunnel++;
557             return (1);
558         }
559         /* Delete the tunnel options from the packet. */
560         ovbcopy((caddr_t) (ipoptions + TUNNEL_LEN), (caddr_t) ipoptions,
561                 (unsigned) (m->m_len - (IP_HDR_LEN + TUNNEL_LEN)));
562         m->m_len -= TUNNEL_LEN;
563         ip->ip_len -= TUNNEL_LEN;
564         ip->ip_hl -= TUNNEL_LEN >> 2;
565     }
```
                                                                                    ip_mroute.c

Figure 14.37   ip_mforward function: tunnel arrival.

tte.c

516–526   The two arguments to `ip_mforward` are a pointer to the mbuf chain containing the datagram; and a pointer to the `ifnet` structure of the receiving interface.

Arrival on physical interface

527–530   To distinguish between a multicast datagram arriving on a physical interface and a tunneled datagram arriving on the same physical interface, the IP header is examined for the characteristic LSRR option. If the header is too small to contain the option, or if the options don't start with a NOP followed by an LSRR option, it is assumed that the datagram arrived on a physical interface and `tunnel_src` is set to 0.

Arrival on a tunnel

531–558   If the datagram looks as though it arrived on a tunnel, the options are verified to make sure they are well formed. If the options are not well formed for a multicast tunnel, `ip_mforward` returns 1 to indicate that the datagram should be discarded. Figure 14.38 shows the organization of the tunnel options.
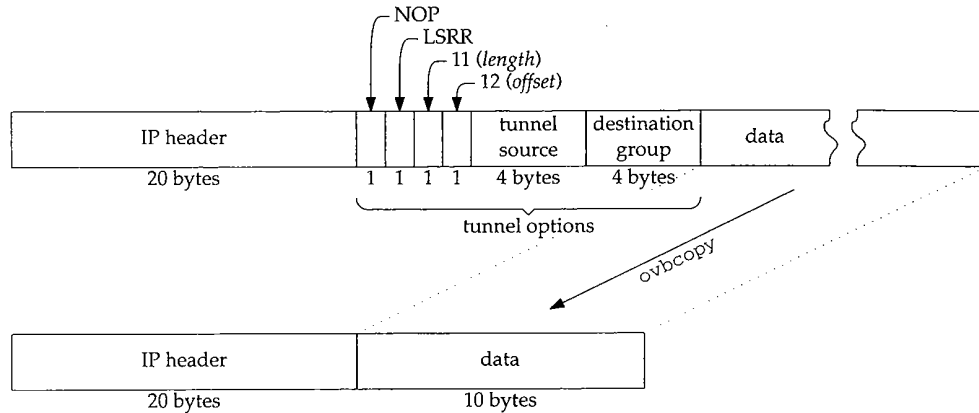

Figure 14.38   Multicast tunnel options.

In Figure 14.38 we assume there are no other options in the datagram, although that is not required. Any other IP options will appear after the LSRR option, which is always inserted before any other options by the multicast router at the start of the tunnel.

Delete tunnel options

559–565   If the options are OK, they are removed from the datagram by shifting the remaining options and data forward and adjusting `m_len` in the mbuf header and `ip_len` and `ip_hl` in the IP header (Figure 14.38).

`ip_mforward` often uses `tunnel_source` as its return value, which is only nonzero when the datagram arrives on a tunnel. When `ip_mforward` returns a nonzero value, the caller discards the datagram. For `ipintr` this means that a datagram that arrives on a tunnel is passed to `ip_mforward` and discarded by `ipintr`. The forwarding code strips out the tunnel information, duplicates the datagram, and sends the datagrams with `ip_output`, which calls `ip_mloopback` if the interface is a member of the destination group.

1route.c

The next part of ip_mforward, shown in Figure 14.39, discards the datagram if it is ineligible for forwarding.

———————————————————————————————————— *ip_mroute.c*

```
566     /*
567      * Don't forward a packet with time-to-live of zero or one,
568      * or a packet destined to a local-only group.
569      */
570     if (ip->ip_ttl <= 1 ||
571         ntohl(ip->ip_dst.s_addr) <= INADDR_MAX_LOCAL_GROUP)
572         return ((int) tunnel_src);

573     /*
574      * Don't forward if we don't have a route for the packet's origin.
575      */
576     if (!(rt = mrtfind(ip->ip_src))) {
577         mrtstat.mrts_no_route++;
578         return ((int) tunnel_src);
579     }
580     /*
581      * Don't forward if it didn't arrive from the parent vif for its origin.
582      */
583     vifi = rt->mrt_parent;
584     if (tunnel_src == 0) {
585         if ((viftable[vifi].v_flags & VIFF_TUNNEL) ||
586             viftable[vifi].v_ifp != ifp)
587             return ((int) tunnel_src);
588     } else {
589         if (!(viftable[vifi].v_flags & VIFF_TUNNEL) ||
590             viftable[vifi].v_rmt_addr.s_addr != tunnel_src)
591             return ((int) tunnel_src);
592     }
```
———————————————————————————————————— *ip_mroute.c*

Figure 14.39 ip_mforward function: forwarding eligibility checks.

Expired TTL or local multicast

566–572   If ip_ttl is 0 or 1, the datagram has reached the end of its lifetime and is not forwarded. If the destination group is less than or equal to INADDR_MAX_LOCAL_GROUP (the 224.0.0.x groups, Figure 12.1), the datagram is not allowed beyond the local network and is not forwarded. In either case, tunnel_src is returned to the caller.

> Version 3.3 of mrouted supports administrative scoping of certain destination groups. An interface can be configured to discard datagrams addressed to these groups, similar to the automatic scoping of the 224.0.0.x groups.

No route available

573–579   If mrtfind cannot locate a route based on the *source* address of the datagram, the function returns. Without a route, the multicast router cannot determine to which interfaces the datagram should be forwarded. This might occur, for example, when the multicast datagrams arrive before the multicast routing table has been updated by mrouted.

Arrived on unexpected interface

580–592   If the datagram arrived on a physical interface but was expected to arrive on a tunnel or on a different physical interface, `ip_mforward` returns. If the datagram arrived on a tunnel but was expected to arrive on a physical interface or on a different tunnel, `ip_mforward` returns. A datagram may arrive on an unexpected interface when the routing tables are in transition because of changes in the group membership or in the physical topology of the network.

The final part of `ip_mforward` (Figure 14.40) sends the datagram on each of the outgoing interfaces specified in the multicast route entry.

```
                                                                  ip_mroute.c
593     /*
594      * For each vif, decide if a copy of the packet should be forwarded.
595      * Forward if:
596      *       - the ttl exceeds the vif's threshold AND
597      *       - the vif is a child in the origin's route AND
598      *       - ( the vif is not a leaf in the origin's route OR
599      *            the destination group has members on the vif )
600      *
601      * (This might be speeded up with some sort of cache -- someday.)
602      */
603     for (vifp = viftable, vifi = 0; vifi < numvifs; vifp++, vifi++) {
604         if (ip->ip_ttl > vifp->v_threshold &&
605             VIFM_ISSET(vifi, rt->mrt_children) &&
606             (!VIFM_ISSET(vifi, rt->mrt_leaves) ||
607              grplst_member(vifp, ip->ip_dst))) {
608             if (vifp->v_flags & VIFF_TUNNEL)
609                 tunnel_send(m, vifp);
610             else
611                 phyint_send(m, vifp);
612         }
613     }

614     return ((int) tunnel_src);
615 }
                                                                  ip_mroute.c
```

Figure 14.40 `ip_mforward` function: forwarding.

593–615   For each interface in `viftable`, a datagram is sent on the interface if

- the datagram's TTL is greater than the multicast threshold for the interface,
- the interface is a child interface for the route, and
- the interface is not connected to a leaf network.

If the interface is a leaf, the datagram is output only if there is a member of the destination group on the network (i.e., `grplst_member` returns a nonzero value).

`tunnel_send` forwards the datagram on tunnel interfaces; `phyint_send` is used for physical interfaces.

`phyint_send` Function

To send a multicast datagram on a physical interface, `phyint_send` (Figure 14.41) specifies the output interface explicitly in the `ip_moptions` structure it passes to `ip_output`.

```
                                                                          ip_mroute.c
616 static void
617 phyint_send(m, vifp)
618 struct mbuf *m;
619 struct vif *vifp;
620 {
621     struct ip *ip = mtod(m, struct ip *);
622     struct mbuf *mb_copy;
623     struct ip_moptions *imo;
624     int     error;
625     struct ip_moptions simo;

626     mb_copy = m_copy(m, 0, M_COPYALL);
627     if (mb_copy == NULL)
628         return;

629     imo = &simo;
630     imo->imo_multicast_ifp = vifp->v_ifp;
631     imo->imo_multicast_ttl = ip->ip_ttl - 1;
632     imo->imo_multicast_loop = 1;

633     error = ip_output(mb_copy, NULL, NULL, IP_FORWARDING, imo);
634 }
                                                                          ip_mroute.c
```

Figure 14.41   `phyint_send` function.

616–634   `m_copy` duplicates the outgoing datagram. The `ip_moptions` structure is set to force the datagram to be transmitted on the selected interface. The TTL value is decremented, and multicast loopback is enabled.

The datagram is passed to `ip_output`. The `IP_FORWARDING` flag avoids an infinite loop, where `ip_output` calls `ip_mforward` again.
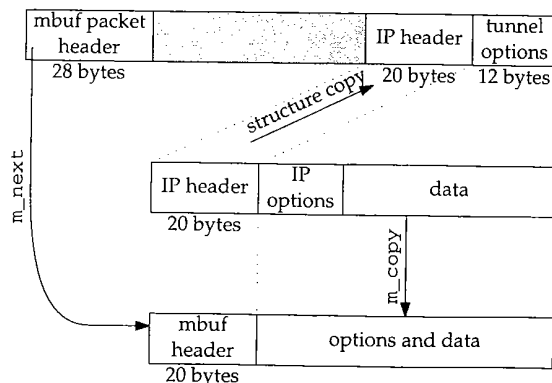

Figure 14.42   Inserting tunnel options.

`tunnel_send` Function

To send a datagram on a tunnel, `tunnel_send` (Figure 14.43) must construct the appropriate tunnel options and insert them in the header of the outgoing datagram. Figure 14.42 shows how `tunnel_send` prepares a packet for the tunnel.

```
                                                                        ip_mroute.c
635 static void
636 tunnel_send(m, vifp)
637 struct mbuf *m;
638 struct vif *vifp;
639 {
640     struct ip *ip = mtod(m, struct ip *);
641     struct mbuf *mb_copy, *mb_opts;
642     struct ip *ip_copy;
643     int     error;
644     u_char *cp;

645     /*
646      * Make sure that adding the tunnel options won't exceed the
647      * maximum allowed number of option bytes.
648      */
649     if (ip->ip_hl > (60 - TUNNEL_LEN) >> 2) {
650         mrtstat.mrts_cant_tunnel++;
651         return;
652     }
653     /*
654      * Get a private copy of the IP header so that changes to some
655      * of the IP fields don't damage the original header, which is
656      * examined later in ip_input.c.
657      */
658     mb_copy = m_copy(m, IP_HDR_LEN, M_COPYALL);
659     if (mb_copy == NULL)
660         return;
661     MGETHDR(mb_opts, M_DONTWAIT, MT_HEADER);
662     if (mb_opts == NULL) {
663         m_freem(mb_copy);
664         return;
665     }
666     /*
667      * Make mb_opts be the new head of the packet chain.
668      * Any options of the packet were left in the old packet chain head
669      */
670     mb_opts->m_next = mb_copy;
671     mb_opts->m_len = IP_HDR_LEN + TUNNEL_LEN;
672     mb_opts->m_data += MSIZE - mb_opts->m_len;
                                                                        ip_mroute.c
```

Figure 14.43   `tunnel_send` function: verify and allocate new header.

Will the tunnel options fit?

635–652   If there is no room in the IP header for the tunnel options, `tunnel_send` returns immediately and the datagram is not forwarded on the tunnel. It may be forwarded on other interfaces.

Duplicate the datagram and allocate mbuf for new header and tunnel options

653–672   In the call to m_copy, the starting offset for the copy is 20 (IP_HDR_LEN). The resulting mbuf chain contains the options and data for the datagram but not the IP header. mb_opts points to a new datagram header allocated by MGETHDR. The datagram header is prepended to mb_copy. Then m_len and m_data are adjusted to accommodate an IP header and the tunnel options.

The second half of tunnel_send, shown in Figure 14.44, modifies the headers of the outgoing packet and sends the packet.

```
                                                              ─── ip_mroute.c
673     ip_copy = mtod(mb_opts, struct ip *);
674     /*
675      * Copy the base ip header to the new head mbuf.
676      */
677     *ip_copy = *ip;
678     ip_copy->ip_ttl--;
679     ip_copy->ip_dst = vifp->v_rmt_addr;     /* remote tunnel end-point */
680     /*
681      * Adjust the ip header length to account for the tunnel options.
682      */
683     ip_copy->ip_hl += TUNNEL_LEN >> 2;
684     ip_copy->ip_len += TUNNEL_LEN;
685     /*
686      * Add the NOP and LSRR after the base ip header
687      */
688     cp = (u_char *) (ip_copy + 1);
689     *cp++ = IPOPT_NOP;
690     *cp++ = IPOPT_LSRR;
691     *cp++ = 11;                     /* LSRR option length */
692     *cp++ = 8;                      /* LSSR pointer to second element */
693     *(u_long *) cp = vifp->v_lcl_addr.s_addr;    /* local tunnel end-point */
694     cp += 4;
695     *(u_long *) cp = ip->ip_dst.s_addr;      /* destination group */

696     error = ip_output(mb_opts, NULL, NULL, IP_FORWARDING, NULL);
697 }
                                                              ─── ip_mroute.c
```

Figure 14.44   tunnel_send function: construct headers and send.

Modify IP header

673–679   The original IP header is copied from the original mbuf chain into the newly allocated mbuf header. The TTL in the header is decremented, and the destination is changed to be the other end of the tunnel.

Construct tunnel options

680–664   ip_hl and ip_len are adjusted to accommodate the tunnel options. The tunnel options are placed just after the IP header: a NOP, followed by the LSRR code, the length of the LSRR option (11 bytes), and a pointer to the *second* address in the option (8 bytes). The source route consists of the local tunnel end point followed by the destination group (Figure 14.13).

Send the tunneled datagram

665–697   ip_output sends the datagram, which now looks like a unicast datagram with an LSRR option since the destination address is the unicast address of the other end of the tunnel. When it reaches the other end of the tunnel, the tunnel options are stripped off and the datagram is forwarded at that point, possibly through additional tunnels.

14.9 Cleanup: ip_mrouter_done Function

When mrouted shuts down, it issues the DVMRP_DONE command, which is handled by the ip_mrouter_done function shown in Figure 14.45.

```
                                                                    ─── ip_mroute.c
161 int
162 ip_mrouter_done()
163 {
164     vifi_t  vifi;
165     int     i;
166     struct ifnet *ifp;
167     int     s;
168     struct ifreq ifr;
169     s = splnet();
170     /*
171      * For each phyint in use, free its local group list and
172      * disable promiscuous reception of all IP multicasts.
173      */
174     for (vifi = 0; vifi < numvifs; vifi++) {
175         if (viftable[vifi].v_lcl_addr.s_addr != 0 &&
176             !(viftable[vifi].v_flags & VIFF_TUNNEL)) {
177             if (viftable[vifi].v_lcl_grps)
178                 free(viftable[vifi].v_lcl_grps, M_MRTABLE);
179             satosin(&ifr.ifr_addr)->sin_family = AF_INET;
180             satosin(&ifr.ifr_addr)->sin_addr.s_addr = INADDR_ANY;
181             ifp = viftable[vifi].v_ifp;
182             (*ifp->if_ioctl) (ifp, SIOCDELMULTI, (caddr_t) & ifr);
183         }
184     }
185     bzero((caddr_t) viftable, sizeof(viftable));
186     numvifs = 0;
187     /*
188      * Free any multicast route entries.
189      */
190     for (i = 0; i < MRTHASHSIZ; i++)
191         if (mrttable[i])
192             free(mrttable[i], M_MRTABLE);
193     bzero((caddr_t) mrttable, sizeof(mrttable));
194     cached_mrt = NULL;
195     ip_mrouter = NULL;
196     splx(s);
197     return (0);
198 }
                                                                    ─── ip_mroute.c
```

Figure 14.45   ip_mrouter_done function: DVMRP_DONE command.

161–186   This function runs at `splnet` to avoid any interaction with the multicast forwarding code. For every physical multicast interface, the list of local groups is released and the `SIOCDELMULTI` command is issued to stop receiving multicast datagrams (Exercise 14.3). The entire `viftable` array is cleared by `bzero` and `numvifs` is set to 0.

187–198   Every active entry in the multicast routing table is released, the entire table is cleared with `bzero`, the cache is cleared, and `ip_mrouter` is reset.

> Each entry in the multicast routing table may be the first in a linked list of entries. This code introduces a memory leak by releasing only the first entry in the list.

14.10 Summary

In this chapter we described the general concept of internetwork multicasting and the specific functions within the Net/3 kernel that support it. We did not discuss the implementation of `mrouted`, but the source is readily available for the interested reader.

We described the virtual interface table and the differences between a physical interface and a tunnel, as well as the LSRR options used to implement tunnels in Net/3.

We illustrated the RPB, TRPB, and RPM algorithms and described the kernel tables used to forward multicast datagrams according to TRPB. The concept of parent and leaf networks was also discussed.

Exercises

14.1   In Figure 14.25, how many multicast routes are needed?

14.2   Why is the update to the group membership cache in Figure 14.23 protected by `splnet` and `splx`?

14.3   What happens when `SIOCDELMULTI` is issued for an interface that has explicitly joined a multicast group with the `IP_ADD_MEMBERSHIP` option?

14.4   When a datagram arrives on a tunnel and is accepted by `ip_mforward`, it may be looped back by `ip_output` when it is forwarded to a physical interface. Why does `ip_mforward` discard the looped-back packet when it arrives on the physical interface?

14.5   Redesign the group address cache to increase its effectiveness.

15

Socket Layer

15.1 Introduction

This chapter is the first of three that cover the socket-layer code in Net/3. The socket abstraction was introduced with the 4.2BSD release in 1983 to provide a uniform interface to network and interprocess communication protocols. The Net/3 release discussed here is based on the 4.3BSD Reno version of sockets, which is slightly different from the earlier 4.2 releases used by many Unix vendors.

As described in Section 1.7, the socket layer maps protocol-independent requests from a process to the protocol-specific implementation selected when the socket was created.

To allow standard Unix I/O system calls such as read and write to operate with network connections, the filesystem and networking facilities in BSD releases are integrated at the system call level. Network connections represented by sockets are accessed through a descriptor (a small integer) in the same way an open file is accessed through a descriptor. This allows the standard filesystem calls such as read and write, as well as network-specific system calls such as sendmsg and recvmsg, to work with a descriptor associated with a socket.

Our focus is on the implementation of sockets and the associated system calls and not on how a typical program might use the socket layer to implement network applications. For a detailed discussion of the process-level socket interface and how to program network applications see [Stevens 1990] and [Rago 1993].

Figure 15.1 shows the layering between the socket interface in a process and the protocol implementation in the kernel.
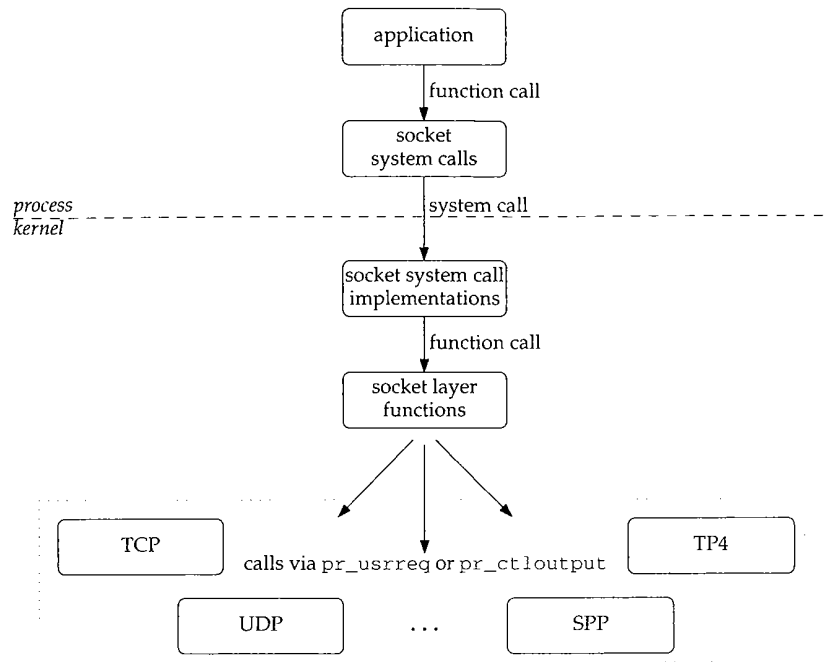
435

Figure 15.1   The socket layer converts generic requests to specific protocol operations.

`splnet` Processing

The socket layer contains many paired calls to `splnet` and `splx`. As discussed in Section 1.12, these calls protect code that accesses data structures shared between the socket layer and the protocol-processing layer. Without calls to `splnet`, a software interrupt that initiates protocol processing and changes the shared data structures will confuse the socket-layer code when it resumes.

We assume that readers understand these calls and we rarely point them out in our discussion.

15.2   Code Introduction

The three files listed in Figure 15.2 are described in this chapter.

Global Variables

The two global variable covered in this chapter are described in Figure 15.3.

File	Description
sys/socketvar.h	socket structure definitions
kern/uipc_syscalls.c kern/uipc_socket.c	system call implementation socket-layer functions

Figure 15.2 Files discussed in this chapter.

Variable	Datatype	Description
socketops sysent	struct fileops struct sysent[]	socket implementation of I/O system calls array of system call entries

Figure 15.3 Global variable introduced in this chapter.

15.3 socket Structure

A socket represents one end of a communication link and holds or points to all the information associated with the link. This information includes the protocol to use, state information for the protocol (which includes source and destination addresses), queues of arriving connections, data buffers, and option flags. Figure 15.5 shows the definition of a socket and its associated buffers.

41–42   so_type is specified by the process creating a socket and identifies the communication semantics to be supported by the socket and the associated protocol. so_type shares the same values as pr_type shown in Figure 7.8. For UDP, so_type would be SOCK_DGRAM and for TCP it would be SOCK_STREAM.

43   so_options is a collection of flags that modify the behavior of a socket. Figure 15.4 describes the flags.

so_options	Kernel only	Description
SO_ACCEPTCONN	•	socket accepts incoming connections
SO_BROADCAST		socket can send broadcast messages
SO_DEBUG		socket records debugging information
SO_DONTROUTE		output operations bypass routing tables
SO_KEEPALIVE		socket probes idle connections
SO_OOBINLINE		socket keeps out-of-band data inline
SO_REUSEADDR		socket can reuse a local address
SO_REUSEPORT		socket can reuse a local address and port
SO_USELOOPBACK		routing domain sockets only; sending process receives its own routing requests

Figure 15.4 so_options values.

A process can modify all the socket options with the getsockopt and setsockopt system calls except SO_ACCEPTCONN, which is set by the kernel when the listen system call is issued on the socket.

socketvar.h

```
41 struct socket {
42     short    so_type;            /* generic type, Figure 7.8 */
43     short    so_options;         /* from socket call, Figure 15.4 */
44     short    so_linger;          /* time to linger while closing */
45     short    so_state;           /* internal state flags, Figure 15.6 */
46     caddr_t so_pcb;              /* protocol control block */
47     struct protosw *so_proto;    /* protocol handle */
48 /*
49  * Variables for connection queueing.
50  * Socket where accepts occur is so_head in all subsidiary sockets.
51  * If so_head is 0, socket is not related to an accept.
52  * For head socket so_q0 queues partially completed connections,
53  * while so_q is a queue of connections ready to be accepted.
54  * If a connection is aborted and it has so_head set, then
55  * it has to be pulled out of either so_q0 or so_q.
56  * We allow connections to queue up based on current queue lengths
57  * and limit on number of queued connections for this socket.
58  */
59     struct socket *so_head;      /* back pointer to accept socket */
60     struct socket *so_q0;        /* queue of partial connections */
61     struct socket *so_q;         /* queue of incoming connections */
62     short    so_q0len;           /* partials on so_q0 */
63     short    so_qlen;            /* number of connections on so_q */
64     short    so_qlimit;          /* max number queued connections */
65     short    so_timeo;           /* connection timeout */
66     u_short so_error;            /* error affecting connection */
67     pid_t    so_pgid;            /* pgid for signals */
68     u_long  so_oobmark;          /* chars to oob mark */
69 /*
70  * Variables for socket buffering.
71  */
72     struct sockbuf {
73         u_long  sb_cc;           /* actual chars in buffer */
74         u_long  sb_hiwat;        /* max actual char count */
75         u_long  sb_mbcnt;        /* chars of mbufs used */
76         u_long  sb_mbmax;        /* max chars of mbufs to use */
77         long    sb_lowat;        /* low water mark */
78         struct mbuf *sb_mb;      /* the mbuf chain */
79         struct selinfo sb_sel;   /* process selecting read/write */
80         short    sb_flags;       /* Figure 16.5 */
81         short    sb_timeo;       /* timeout for read/write */
82     } so_rcv, so_snd;
83     caddr_t so_tpcb;             /* Wisc. protocol control block XXX */
84     void    (*so_upcall) (struct socket * so, caddr_t arg, int waitf);
85     caddr_t so_upcallarg;        /* Arg for above */
86 };
```

socketvar.h

Figure 15.5   struct socket definition.

44   so_linger is the time in clock ticks that a socket waits for data to drain while clos-
ing a connection (Section 15.15).

45   so_state represents the internal state and additional characteristics of the socket.
Figure 15.6 lists the possible values for so_state.

so_state	Kernel only	Description
SS_ASYNC		socket should send asynchronous notification of I/O events
SS_NBIO		socket operations should not block the process
SS_CANTRCVMORE	•	socket cannot receive more data from peer
SS_CANTSENDMORE	•	socket cannot send more data to peer
SS_ISCONFIRMING	•	socket is negotiating a connection request
SS_ISCONNECTED	•	socket is connected to a foreign socket
SS_ISCONNECTING	•	socket is connecting to a foreign socket
SS_ISDISCONNECTING	•	socket is disconnecting from peer
SS_NOFDREF	•	socket is not associated with a descriptor
SS_PRIV	•	socket was created by a process with superuser privileges
SS_RCVATMARK	•	process has consumed all data received before the most recent out-of-band data was received

Figure 15.6   so_state values.

In Figure 15.6, the middle column shows that SS_ASYNC and SS_NBIO can be
changed explicitly by a process by the fcntl and ioctl system calls. The other flags
are implicitly changed by the process during the execution of system calls. For exam-
ple, if the process calls connect, the SS_ISCONNECTED flag is set by the kernel when
the connection is established.

SS_NBIO and SS_ASYNC Flags

By default, a process blocks waiting for resources when it makes an I/O request. For
example, a read system call on a socket blocks if there is no data available from the net-
work. When the data arrives, the process is unblocked and read returns. Similarly,
when a process calls write, the kernel blocks the process until space is available in the
kernel for the data. If SS_NBIO is set, the kernel does not block a process during I/O on
the socket but instead returns the error code EWOULDBLOCK.

If SS_ASYNC is set, the kernel sends the SIGIO signal to the process or process
group specified by so_pgid when the status of the socket changes for one of the fol-
lowing reasons:

- a connection request has completed,
- a disconnect request has been initiated,
- a disconnect request has been completed,
- half of a connection has been shut down,
- data has arrived on a socket,
- data has been sent from a socket (i.e., the output buffer has free space), or
- an asynchronous error has occurred on a UDP or TCP socket.

46   so_pcb points to a protocol control block that contains protocol-specific state information and parameters for the socket. Each protocol defines its own control block structure, so so_pcb is defined to be a generic pointer. Figure 15.7 lists the control block structures that we discuss.

so_pcb never points to a tcpcb structure directly; see Figure 22.1.

Protocol	Control block	Reference
UDP	struct inpcb	Section 22.3
TCP	struct inpcb struct tcpcb	Section 22.3 Section 24.5
ICMP, IGMP, raw IP	struct inpcb	Section 22.3
Route	struct rawcb	Section 20.3

Figure 15.7   Protocol control blocks.

47   so_proto points to the protosw structure of the protocol selected by the process during the socket system call (Section 7.4).

48–64   Sockets with SO_ACCEPTCONN set maintain two connection queues. Connections that are not yet established (e.g., the TCP three-way handshake is not yet complete) are placed on the queue so_q0. Connections that are established and are ready to be accepted (e.g., the TCP three-way handshake is complete) are placed on the queue so_q. The lengths of the queues are kept in so_q0len and so_qlen. Each queued connection is represented by its own socket. so_head in each queued socket points to the original socket with SO_ACCEPTCONN set.

The maximum number of queued connections for a particular socket is controlled by so_qlimit, which is specified by a process when it calls listen. The kernel silently enforces an upper limit of 5 (SOMAXCONN, Figure 15.24) and a lower limit of 0. A somewhat obscure formula shown with Figure 15.29 uses so_qlimit to control the number of queued connections.

Figure 15.8 illustrates a queue configuration in which three connections are ready to be accepted and one connection is being established.

65   so_timeo is a *wait channel* (Section 15.10) used during accept, connect, and close processing.

66   so_error holds an error code until it can be reported to a process during the next system call that references the socket.

67   If SS_ASYNC is set for a socket, the SIGIO signal is sent to the process (if so_pgid is greater than 0) or to the progress group (if so_pgid is less than 0). so_pgid can be changed or examined with the SIOCSPGRP and SIOCGPGRP ioctl commands. For more information about process groups see [Stevens 1992].

68   so_oobmark identifies the point in the input data stream at which out-of-band data was most recently received. Section 16.11 discusses socket support for out-of-band data and Section 29.7 discusses the semantics of out-of-band data in TCP.

69–82   Each socket contains two data buffers, so_rcv and so_snd, used to buffer incoming and outgoing data. These are structures contained within the socket structure, not
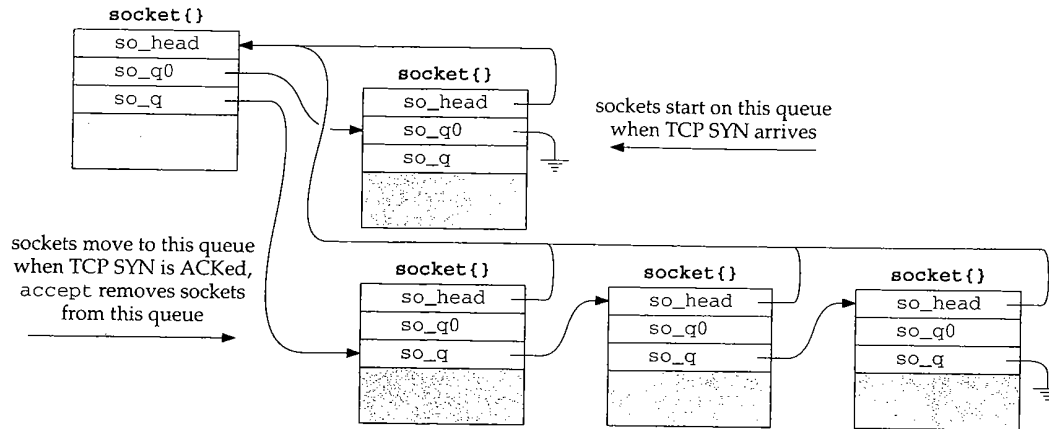
Figure 15.8 Socket connection queues.

pointers to structures. We describe the organization and use of the socket buffers in Chapter 16.

83–86   so_tpcb is not used by Net/3. so_upcall and so_upcallarg are used only by the NFS software in Net/3.

> NFS is unusual. In many ways it is a process-level application that has been moved into the kernel. The so_upcall mechanism triggers NFS input processing when data is added to a socket receive buffer. The tsleep and wakeup mechanism is inappropriate in this case, since the NFS protocol executes within the kernel, not as a process.

The files socketvar.h and uipc_socket2.c define several macros and functions that simplify the socket-layer code. Figure 15.9 summarizes them.

15.4 System Calls

A process interacts with the kernel through a collection of well-defined functions called *system calls*. Before showing the system calls that support networking, we discuss the system call mechanism itself.

The transfer of execution from a process to the protected environment of the kernel is machine- and implementation-dependent. In the discussion that follows, we use the 386 implementation of Net/3 to illustrate implementation specific operations.

In BSD kernels, each system call is numbered and the hardware is configured to transfer control to a single kernel function when the process executes a system call. The particular system call is identified as an integer argument to the function. In the 386 implementation, syscall is that function. Using the system call number, syscall indexes a table to locate the sysent structure for the requested system call. Each entry in the table is a sysent structure:

Name	Description
sosendallatonce	Does the protocol associated with *so* require each send system call to result in a single protocol request? int **sosendallatonce**(struct socket *so);
soisconnecting	Set the socket state to SS_ISCONNECTING. int **soisconnecting**(struct socket *so);
soisconnected	See Figure 15.30.
soreadable	Will a read on *so* return information without blocking? int **soreadable**(struct socket *so);
sowriteable	Will a write on *so* return without blocking? int **sowriteable**(struct socket *so);
socantsendmore	Set the SS_CANTSENDMORE flag. Wake up any processes sleeping on the send buffer. int **socantsendmore**(struct socket *so);
socantrcvmore	Set the SS_CANTRCVMORE flag. Wake up processes sleeping on the receive buffer. int **socantrcvmore**(struct socket *so);
sodisconnect	Issue the PRU_DISCONNECT request. int **sodisconnect**(struct socket *so);
soisdisconnecting	Clear the SS_ISCONNECTING flag. Set SS_ISDISCONNECTING, SS_CANTRCVMORE, and SS_CANTSENDMORE flags. Wake up any processes selecting on the socket. int **soisdisconnecting**(struct socket *so);
soisdisconnected	Clear the SS_ISCONNECTING, SS_ISCONNECTED, and SS_ISDISCONNECTING flags. Set the SS_CANTRCVMORE and SS_CANTSENDMORE flags. Wake up any processes selecting on the socket or waiting for close to complete. int **soisdisconnected**(struct socket *so);
soqinsque	Insert *so* on a queue associated with *head*. If *q* is 0, the socket is added to the end of so_q0, which holds incomplete connections. Otherwise, the socket is added to the end of so_q, which holds connections that are ready to be accepted. Net/1 incorrectly placed sockets at the front of the queue. int **soqinsque**(struct socket *head, struct socket *so, int *q*);
soqremque	Remove *so* from the queue identified by *q*. The socket queues are located by following *so->so_head*. int **soqremque**(struct socket *so, int *q*);

Figure 15.9 Socket macros and functions.

```
struct sysent {
    int sy_narg;           /* number of arguments */
    int (*sy_call) ();     /* implementing function */
};                         /* system call table entry */
```

Here are several entries from the sysent array, which is defined in kern/init_sysent.c.

```
struct sysent sysent[] = {
        /* ... */
        { 3, recvmsg },              /* 27 = recvmsg */
        { 3, sendmsg },              /* 28 = sendmsg */
        { 6, recvfrom },             /* 29 = recvfrom */
        { 3, accept },               /* 30 = accept */
        { 3, getpeername },          /* 31 = getpeername */
        { 3, getsockname },          /* 32 = getsockname */
        /* ... */
}
```

For example, the recvmsg system call is the 27th entry in the system call table, has three arguments, and is implemented by the recvmsg function in the kernel.

syscall copies the arguments from the calling process into the kernel and allocates an array to hold the results of the system call, which syscall returns to the process when the system call completes. syscall dispatches control to the kernel function associated with the system call. In the 386 implementation, this call looks like:

```
struct sysent *callp;
error = (*callp->sy_call)(p, args, rval);
```

where callp is a pointer to the relevant sysent structure, p is a pointer to the process table entry for the process that made the system call, args represents the arguments to the system call as an array of 32-bit words, and rval is an array of two 32-bit words to hold the return value of the system call. When we use the term *system call*, we mean the function within the kernel called by syscall, not the function within the process called by the application.

syscall expects the system call function (i.e., what sy_call points to) to return 0 if no errors occurred and a nonzero error code otherwise. If no error occurs, the kernel passes the values in rval back to the process as the return value of the system call (the one made by the application). If an error occurs, syscall ignores the values in rval and returns the error code to the process in a machine-dependent way so that the error is made available to the process in the external variable errno. The function called by the application returns −1 or a null pointer to indicate that errno should be examined.

The 386 implementation sets the carry bit to indicate that the value returned by syscall is an error code. The system call stub in the process stores the code in errno and returns −1 or a null pointer to the application. If the carry bit is not set, the value returned by syscall is returned by the stub.

To summarize, a function implementing a system call "returns" two values: one for the syscall function, and a second (found in rval) that syscall returns to the calling process when no error occurs.

Example

The prototype for the `socket` system call is:

```
int socket(int domain, int type, int protocol);
```

The prototype for the kernel function that implements the system call is

```
struct socket_args {
    int domain;
    int type;
    int protocol;
};
socket(struct proc *p, struct socket_args *uap, int *retval);
```

When an application calls `socket`, the process passes three separate integers to the kernel with the system call mechanism. `syscall` copies the arguments into an array of 32-bit values and passes a pointer to the array as the second argument to the kernel version of `socket`. The kernel version of `socket` treats the second argument as a pointer to an `socket_args` structure. Figure 15.10 illustrates this arrangement.
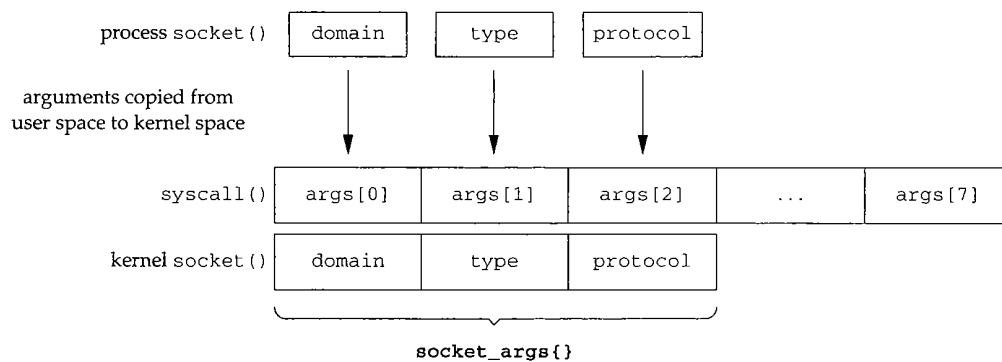

Figure 15.10   `socket` argument processing.

As illustrated by `socket`, each kernel function that implements a system call declares `args` not as a pointer to an array of 32-bit words, but as as a pointer to a structure specific to the system call.

> The implicit cast is legal only in traditional K&R C or in ANSI C when a prototype is not in effect. If a prototype is in effect, the compiler generates a warning.

`syscall` prepares the return value of 0 before executing the kernel system call function. If no error occurs, the system call function can return without clearing `*retval` and `syscall` returns 0 to the process.

System Call Summary

Figure 15.11 summarizes the system calls relevant to networking.

Category	Name	Function
setup	socket	create a new unnamed socket within a specified communication domain
	bind	assign a local address to a socket
server	listen	prepare a socket to accept incoming connections
	accept	wait for and accept connections
client	connect	establish a connection to a foreign socket
input	read	receive data into a single buffer
	readv	receive data into multiple buffers
	recv	receive data specifying options
	recvfrom	receive data and address of sender
	recvmsg	receive data into multiple buffers, control information, and receive the address of sender; specify receive options
output	write	send data from a single buffer
	writev	send data from multiple buffers
	send	send data specifying options
	sendto	send data to specified address
	sendmsg	send data from multiple buffers and control information to a specified address; specify send options
I/O	select	wait for I/O conditions
termination	shutdown	terminate connection in one or both directions
	close	terminate connection and release socket
administration	fcntl	modify I/O semantics
	ioctl	miscellaneous socket operations
	setsockopt	set socket or protocol options
	getsockopt	get socket or protocol options
	getsockname	get local address assigned to socket
	getpeername	get foreign address assigned to socket

Figure 15.11   Networking system calls in Net/3.

We present the setup, server, client, and termination calls in this chapter. The input and output system calls are discussed in Chapter 16 and the administrative calls in Chapter 17.

Figure 15.12 shows the sequence in which an application might use the calls. The I/O system calls in the large box can be called in any order. This is not a complete state diagram as some valid transitions are not included; just the most common ones are shown.

15.5 Processes, Descriptors, and Sockets

Before describing the socket system calls, we need to discuss the data structures that tie together processes, descriptors, and sockets. Figure 15.13 shows the structures and members relevant to our discussion. A more complete explanation of the file structures can be found in [Leffler et al. 1989].
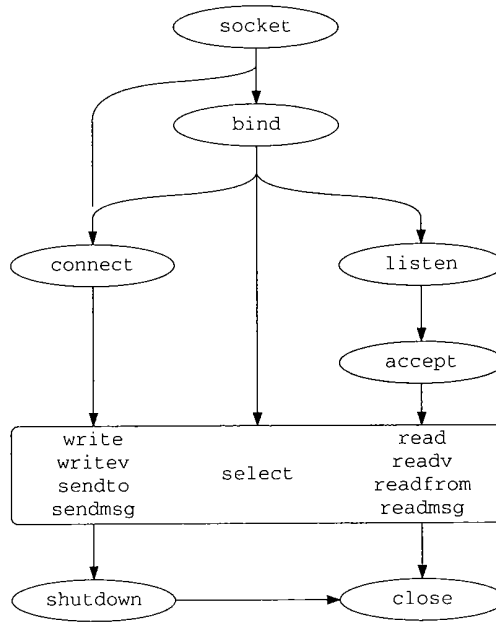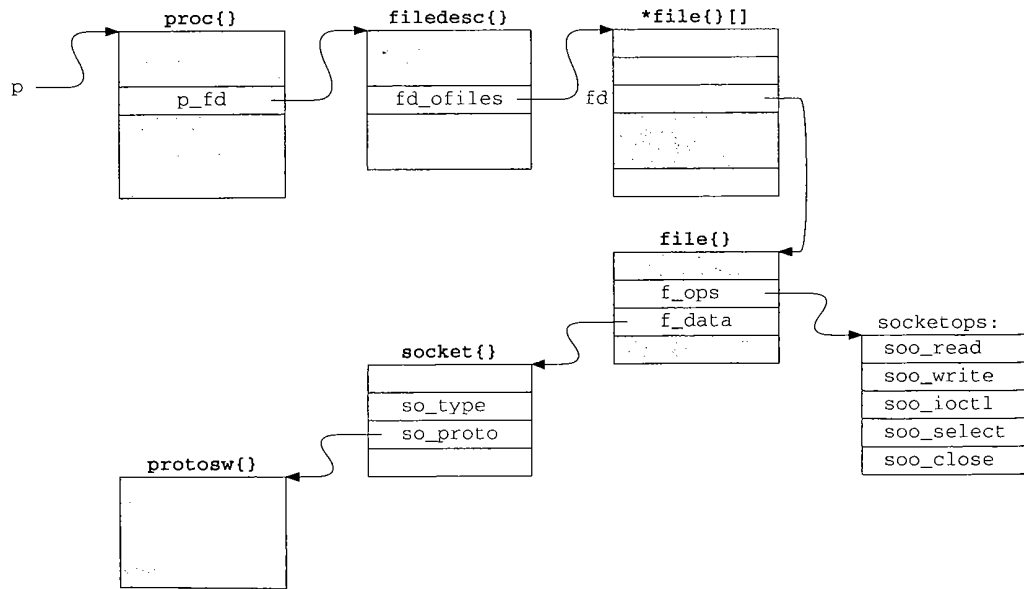
Figure 15.12   Network system call flowchart.

Figure 15.13   Process, file, and socket structures.

The first argument to a function implementing a system call is always p, a pointer to the proc structure of the calling process. The proc structure represents the kernel's notion of a process. Within the proc structure, p_fd points to a filedesc structure, which manages the descriptor table pointed to by fd_ofiles. The descriptor table is dynamically sized and consists of an array of pointers to file structures. Each file structure describes a single open file and can be shared between multiple processes.

Only a single file structure is shown in Figure 15.13. It is accessed by p->p_fd->fd_ofiles[fd]. Within the file structure, two members are of interest to us: f_ops and f_data. The implementation of I/O system calls such as read and write varies according to what type of I/O object is associated with a descriptor. f_ops points to a fileops structure containing a list of function pointers that implement the read, write, ioctl, select, and close system calls for the associated I/O object. Figure 15.13 shows f_ops pointing to a global fileops structure, socketops, which contains pointers to the functions for sockets.

f_data points to private data used by the associated I/O object. For sockets, f_data points to the socket structure associated with the descriptor. Finally, we see that so_proto in the socket structure points to the protosw structure for the protocol selected when the socket is created. Recall that each protosw structure is shared by all sockets associated with the protocol.

We now proceed to discuss the system calls.

15.6 socket **System Call**

The socket system call creates a new socket and associates it with a protocol as specified by the domain, type, and protocol arguments specified by the process. The function (shown in Figure 15.14) allocates a new descriptor, which identifies the socket in future system calls, and returns the descriptor to the process.

42–55   Before each system call a structure is defined to describe the arguments passed from the process to the kernel. In this case, the arguments are passed within a socket_args structure. All the socket-layer system calls have three arguments: p, a pointer to the proc structure for the calling process; uap, a pointer to a structure containing the arguments passed by the process to the system call; and retval, a value–result argument that points to the return value for the system call. Normally, we ignore the p and retval arguments and refer to the contents of the structure pointed to by uap as the arguments to the system call.

56–60   falloc allocates a new file structure and slot in the fd_ofiles array (Figure 15.13). fp points to the new structure and fd is the index of the structure in the fd_ofiles array. socket enables the file structure for read and write access and marks it as a socket. socketops, a global fileops structure shared by all sockets, is attached to the file structure by f_ops. The socketops variable is initialized at compile time as shown in Figure 15.15.

60–69   socreate allocates and initializes a socket structure. If socreate fails, the error code is posted in error, the file structure is released, and the descriptor slot cleared. If socreate succeeds, f_data is set to point to the socket structure and establishes

uipc_syscalls.c

```
42 struct socket_args {
43     int     domain;
44     int     type;
45     int     protocol;
46 };

47 socket(p, uap, retval)
48 struct proc *p;
49 struct socket_args *uap;
50 int     *retval;
51 {
52     struct filedesc *fdp = p->p_fd;
53     struct socket *so;
54     struct file *fp;
55     int     fd, error;

56     if (error = falloc(p, &fp, &fd))
57         return (error);
58     fp->f_flag = FREAD | FWRITE;
59     fp->f_type = DTYPE_SOCKET;
60     fp->f_ops = &socketops;
61     if (error = socreate(uap->domain, &so, uap->type, uap->protocol)) {
62         fdp->fd_ofiles[fd] = 0;
63         ffree(fp);
64     } else {
65         fp->f_data = (caddr_t) so;
66         *retval = fd;
67     }
68     return (error);
69 }
```

uipc_syscalls.c

Figure 15.14   socket system call.

Member	Value
fo_read	soo_read
fo_write	soo_write
fo_ioctl	soo_ioctl
fo_select	soo_select
fo_close	soo_close

Figure 15.15   socketops: the global fileops structure for sockets.

the association between the descriptor and the socket. fd is returned to the process through *retval. socket returns 0 or the error code returned by socreate.

socreate Function

Most socket system calls are divided into at least two functions, in the same way that socket and socreate are. The first function retrieves from the process all the data

required, calls the second so*xxx* function to do the work, and then returns any results to the process. This split is so that the second function can be called directly by kernel-based network protocols, such as NFS. socreate is shown in Figure 15.16.

```
                                                                    uipc_socket.c
43 socreate(dom, aso, type, proto)
44 int     dom;
45 struct socket **aso;
46 int     type;
47 int     proto;
48 {
49      struct proc *p = curproc;    /* XXX */
50      struct protosw *prp;
51      struct socket *so;
52      int     error;

53      if (proto)
54          prp = pffindproto(dom, proto, type);
55      else
56          prp = pffindtype(dom, type);
57      if (prp == 0 || prp->pr_usrreq == 0)
58          return (EPROTONOSUPPORT);
59      if (prp->pr_type != type)
60          return (EPROTOTYPE);
61      MALLOC(so, struct socket *, sizeof(*so), M_SOCKET, M_WAIT);
62      bzero((caddr_t) so, sizeof(*so));
63      so->so_type = type;
64      if (p->p_ucred->cr_uid == 0)
65          so->so_state = SS_PRIV;
66      so->so_proto = prp;
67      error =
68          (*prp->pr_usrreq) (so, PRU_ATTACH,
69              (struct mbuf *) 0, (struct mbuf *) proto, (struct mbuf *) 0);
70      if (error) {
71          so->so_state |= SS_NOFDREF;
72          sofree(so);
73          return (error);
74      }
75      *aso = so;
76      return (0);
77 }
                                                                    uipc_socket.c
```

Figure 15.16 socreate function.

43–52   The four arguments to socreate are: dom, the requested protocol domain (e.g., PF_INET); aso, in which a pointer to a new socket structure is returned; type, the requested socket type (e.g., SOCK_STREAM); and proto, the requested protocol .

Find protocol switch table

53–60   If proto is nonzero, pffindproto looks for the specific protocol requested by the process. If proto is 0, pffindtype looks for a protocol within the specified domain with the semantics specified by type. Both functions return a pointer to a protosw structure of the matching protocol or a null pointer (Section 7.6).

Allocate and initialize `socket` structure

61-66   `socreate` allocates a new `socket` structure, fills it with 0s, records the `type`, and, if the calling process has superuser privileges, turns on `SS_PRIV` in the socket structure.

`PRU_ATTACH` request

67-69   The first example of the protocol-independent socket layer making a protocol-specific request appears in `socreate`. Recall from Section 7.4 and Figure 15.13 that `so->so_proto->pr_usrreq` is a pointer to the user-request function of the protocol associated with socket `so`. Every protocol provides this function in order to handle communication requests from the socket layer. The prototype for the function is:

```
int pr_usrreq(struct socket *so, int req, struct mbuf *m0, *m1, *m2);
```

The first argument, *so*, is a pointer to the relevant socket and *req* is a constant identifying the particular request. The next three arguments (*m0*, *m1*, and *m2*) are different for each request. They are always passed as pointers to `mbuf` structures, even if they have another type. Casts are used when necessary to avoid warnings from the compiler.

Figure 15.17 shows the requests available through the `pr_usrreq` function. The semantics of each request depend on the particular protocol servicing the request.

Request	Arguments			Description
	m0	*m1*	*m2*	
PRU_ABORT				abort any existing connection
PRU_ACCEPT		*address*		wait for and accept a connection
PRU_ATTACH		*protocol*		a new socket has been created
PRU_BIND		*address*		bind the address to the socket
PRU_CONNECT		*address*		establish association or connection to address
PRU_CONNECT2		*socket2*		connect two sockets together
PRU_DETACH				socket is being closed
PRU_DISCONNECT				break association between socket and foreign address
PRU_LISTEN				begin listening for connections
PRU_PEERADDR		*buffer*		return foreign address associated with socket
PRU_RCVD		*flags*		process has accepted some data
PRU_RCVOOB	*buffer*	*flags*		receive OOB data
PRU_SEND	*data*	*address*	*control*	send regular data
PRU_SENDOOB	*data*	*address*	*control*	send OOB data
PRU_SHUTDOWN				end communication with foreign address
PRU_SOCKADDR		*buffer*		return local address associated with socket

Figure 15.17 `pr_usrreq` requests.

PRU_CONNECT2 is supported only within the Unix domain, where it connects two local sockets to each other. Unix pipes are implemented in this way.

Cleanup and return

70-77   Returning to `socreate`, the function attaches the protocol switch table to the new socket and issues the `PRU_ATTACH` request to notify the protocol of the new end point. This request causes most protocols, including TCP and UDP, to allocate and initialize any structures required to support the new end point.

Superuser Privileges

Figure 15.18 summarizes the networking operations that require superuser access.

| Function | Superuser | | Description | Reference |
	Process	Socket		
in_control		•	interface address, netmask, and destination address assignment	Figure 6.14
in_control		•	broadcast address assignment	Figure 6.22
in_pcbbind	•		binding to an Internet port less than 1024	Figure 22.22
ifioctl	•		interface configuration changes	Figure 4.29
ifioctl	•		multicast address configuration (see text)	Figure 12.11
rip_usrreq	•		creating an ICMP, IGMP, or raw IP socket	Figure 32.10
slopen	•		associating a SLIP device with a tty device	Figure 5.9

Figure 15.18   Superuser privileges in Net/3.

The multicast ioctl commands (SIOCADDMULTI and SIOCDELMULTI) are accessible to non-superuser processes when they are invoked indirectly by the IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP socket options (Sections 12.11 and 12.12).

In Figure 15.18, the "Process" column identifies requests that must be made by a superuser process, and the "Socket" column identifies requests that must be issued on a socket *created* by a superuser process (i.e., the process does not need superuser privileges if it has access to the socket, Exercise 15.1). In Net/3, the suser function determines if the calling process has superuser privileges, and the SS_PRIV flag determines if the socket was created by a superuser process.

Since rip_usrreq tests SS_PRIV immediately after creating the socket with socreate, we show this function as accessible only from a superuser process.

15.7 getsock and sockargs Functions

These functions appear repeatedly in the implementation of the socket system calls. getsock maps a descriptor to a file table entry and sockargs copies arguments from the process to a newly allocated mbuf in the kernel. Both functions check for invalid arguments and return a nonzero error code accordingly.

Figure 15.19 shows the getsock function.

754–767   The function selects the file table entry specified by the descriptor fdes with fdp, a pointer to the filedesc structure. getsock returns a pointer to the open file structure in fpp or an error if the descriptor is out of the valid range, does not point to an open file, or does not have a socket associated with it.

Figure 15.20 shows the sockargs function.

768–783   The mechanism described in Section 15.4 copies pointer arguments for a system call from the process to the kernel but does not copy the data referenced by the pointers, since the semantics of each argument are known only by the specific system call and not

————————————————————————————————— *uipc_syscalls.c*

```
754 getsock(fdp, fdes, fpp)
755 struct filedesc *fdp;
756 int     fdes;
757 struct file **fpp;
758 {
759     struct file *fp;

760     if ((unsigned) fdes >= fdp->fd_nfiles ||
761         (fp = fdp->fd_ofiles[fdes]) == NULL)
762         return (EBADF);
763     if (fp->f_type != DTYPE_SOCKET)
764         return (ENOTSOCK);
765     *fpp = fp;
766     return (0);
767 }
```
————————————————————————————————— *uipc_syscalls.c*

Figure 15.19   getsock function.

————————————————————————————————— *uipc_syscalls.c*

```
768 sockargs(mp, buf, buflen, type)
769 struct mbuf **mp;
770 caddr_t buf;
771 int     buflen, type;
772 {
773     struct sockaddr *sa;
774     struct mbuf *m;
775     int     error;

776     if ((u_int) buflen > MLEN) {
777         return (EINVAL);
778     }
779     m = m_get(M_WAIT, type);
780     if (m == NULL)
781         return (ENOBUFS);
782     m->m_len = buflen;
783     error = copyin(buf, mtod(m, caddr_t), (u_int) buflen);
784     if (error)
785         (void) m_free(m);
786     else {
787         *mp = m;
788         if (type == MT_SONAME) {
789             sa = mtod(m, struct sockaddr *);
790             sa->sa_len = buflen;
791         }
792     }
793     return (error);
794 }
```
————————————————————————————————— *uipc_syscalls.c*

Figure 15.20   sockargs function.

by the generic system call mechanism. Several system calls use sockargs to follow the
pointer arguments and copy the referenced data from the process into a newly allocated
mbuf within the kernel. For example, sockargs copies the local socket address
pointed to by bind's second argument from the process to an mbuf.

If the data does not fit in a single mbuf or an mbuf cannot be allocated, `sockargs` returns `EINVAL` or `ENOBUFS`. Note that a standard mbuf is used and not a packet header mbuf. `copyin` copies the data from the process into the mbuf. The most common error from `copyin` is `EACCES`, returned when the process provides an invalid address.

784–785   When an error occurs, the mbuf is discarded and the error code is returned. If there is no error, a pointer to the mbuf is returned in `mp`, and `sockargs` returns 0.

786–794   If `type` is `MT_SONAME`, the process is passing in a `sockaddr` structure. `sockargs` sets the internal length, `sa_len`, to the length of the argument just copied. This ensures that the size contained within the structure is correct even if the process did not initialize the structure correctly.

> Net/3 does include code to support applications compiled on a pre-4.3BSD Reno system, which did not have an `sa_len` member in the `sockaddr` structure, but that code is not shown in Figure 15.20.

15.8 bind System Call

The `bind` system call associates a local network transport address with a socket. A process acting as a client usually does not care what its local address is. In this case, it isn't necessary to call `bind` before the process attempts to communicate; the kernel selects and implicitly binds a local address to the socket as needed.

A server process almost always needs to bind to a specific well-known address. If so, the process must call `bind` before accepting connections (TCP) or receiving datagrams (UDP), because the clients establish connections or send datagrams to the well-known address.

A socket's foreign address is specified by `connect` or by one of the write calls that allow specification of foreign addresses (`sendto` or `sendmsg`).

Figure 15.21 shows `bind`.

70–82   The arguments to `bind` (passed within a `bind_args` structure) are: s, the socket descriptor; `name`, a pointer to a buffer containing the transport address (e.g., a `sockaddr_in` structure); and `namelen`, the size of the buffer.

83–90   `getsock` returns the `file` structure for the descriptor, and `sockargs` copies the local address from the process into an mbuf, `sobind` associates the address specified by the process with the socket. Before `bind` returns `sobind`'s result, the mbuf holding the address is released.

> Technically, a descriptor such as s identifies a `file` structure with an associated `socket` structure and is not itself a `socket` structure. We refer to such a descriptor as a socket to simplify our discussion.

We will see this pattern many times: arguments specified by the process are copied into an mbuf and processed as necessary, and then the mbuf is released before the system call returns. Although mbufs were designed explicitly to facilitate processing of network data packets, they are also effective as a general-purpose dynamic memory allocation mechanism.

uipc_syscalls.c

```
70 struct bind_args {
71     int     s;
72     caddr_t name;
73     int     namelen;
74 };

75 bind(p, uap, retval)
76 struct proc *p;
77 struct bind_args *uap;
78 int     *retval;
79 {
80     struct file *fp;
81     struct mbuf *nam;
82     int     error;

83     if (error = getsock(p->p_fd, uap->s, &fp))
84         return (error);
85     if (error = sockargs(&nam, uap->name, uap->namelen, MT_SONAME))
86         return (error);
87     error = sobind((struct socket *) fp->f_data, nam);
88     m_freem(nam);
89     return (error);
90 }
```

uipc_syscalls.c

Figure 15.21   bind function.

Another pattern illustrated by bind is that retval is unused in many system calls. In Section 15.4 we mentioned that retval is always initialized to 0 before syscall dispatches control to a system call. If 0 is the appropriate return value, the system calls do not need to change retval.

sobind Function

sobind, shown in Figure 15.22, is a wrapper that issues the PRU_BIND request to the protocol associated with the socket.

uipc_socket.c

```
78 sobind(so, nam)
79 struct socket *so;
80 struct mbuf *nam;
81 {
82     int     s = splnet();
83     int     error;

84     error =
85         (*so->so_proto->pr_usrreq) (so, PRU_BIND,
86                             (struct mbuf *) 0, nam, (struct mbuf *) 0);
87     splx(s);
88     return (error);
89 }
```

uipc_socket.c

Figure 15.22   sobind function.

15

's.c

'ls.c

lls.
.11
ills

the

:et.c

;

'et.c

Section 15.9   listen System Call   455

78–89   sobind issues the PRU_BIND request. The local address, nam, is associated with the socket if the request succeeds; otherwise the error code is returned.

15.9 listen System Call

The listen system call, shown in Figure 15.23, notifies a protocol that the process is prepared to accept incoming connections on the socket. It also specifies a limit on the number of connections that can be queued on the socket, after which the socket layer refuses to queue additional connection requests. When this occurs, TCP ignores incoming connection requests. Queued connections are made available to the process when it calls accept (Section 15.11).

```
                                                                ─── uipc_syscalls.c
 91 struct listen_args {
 92      int     s;
 93      int     backlog;
 94 };

 95 listen(p, uap, retval)
 96 struct proc *p;
 97 struct listen_args *uap;
 98 int     *retval;
 99 {
100      struct file *fp;
101      int     error;

102      if (error = getsock(p->p_fd, uap->s, &fp))
103            return (error);
104      return (solisten((struct socket *) fp->f_data, uap->backlog));
105 }
                                                                ─── uipc_syscalls.c
```

Figure 15.23 listen system call.

91–98   The two arguments passed to listen specify the socket descriptor and the connection queue limit.

99–105   getsock returns the file structure for the descriptor, s, and solisten passes the listen request to the protocol layer.

solisten Function

This function, shown in Figure 15.24, issues the PRU_LISTEN request and prepares the socket to receive connections.

90–109   After solisten issues the PRU_LISTEN request and pr_usrreq returns, the socket is marked as ready to accept connections. SS_ACCEPTCONN is not set if a connection is queued when pr_usrreq returns.

The maximum queue size for incoming connections is computed and saved in so_qlimit. Here Net/3 silently enforces a lower limit of 0 and an upper limit of 5 (SOMAXCONN) backlogged connections.

―― *uipc_socket.c*
```
 90 solisten(so, backlog)
 91 struct socket *so;
 92 int     backlog;
 93 {
 94     int     s = splnet(), error;

 95     error =
 96         (*so->so_proto->pr_usrreq) (so, PRU_LISTEN,
 97                 (struct mbuf *) 0, (struct mbuf *) 0, (struct mbuf *) 0);
 98     if (error) {
 99         splx(s);
100         return (error);
101     }
102     if (so->so_q == 0)
103         so->so_options |= SO_ACCEPTCONN;
104     if (backlog < 0)
105         backlog = 0;
106     so->so_qlimit = min(backlog, SOMAXCONN);
107     splx(s);
108     return (0);
109 }
```
―― *uipc_socket.c*

Figure 15.24   solisten function.

15.10 `tsleep` and `wakeup` Functions

When a process executing within the kernel cannot proceed because a kernel resource is unavailable, it waits for the resource by calling `tsleep`, which has the following prototype:

```
int tsleep(caddr_t chan, int pri, char *mesg, int timeo);
```

The first argument to `tsleep`, *chan*, is called the *wait channel*. It identifies the particular resource or event such as an incoming network connection, for which the process is waiting. Many processes can be sleeping on a single wait channel. When the resource becomes available or when the event occurs, the kernel calls `wakeup` with the wait channel as the single argument. The prototype for `wakeup` is:

```
void wakeup(caddr_t chan);
```

All processes waiting for the channel are awakened and set to the run state. The kernel arranges for `tsleep` to return when each of the processes resumes execution.

The *pri* argument specifies the priority of the process when it is awakened, as well as several optional control flags for `tsleep`. By setting the PCATCH flag in *pri*, `tsleep` also returns when a signal arrives. *mesg* is a string identifying the call to `tsleep` and is included in debugging messages and in `ps` output. *timeo* sets an upper bound on the sleep period and is measured in clock ticks.

Figure 15.25 summarizes the return values from `tsleep`.

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

tsleep()	Description
0	The process was awakened by a matching call to wakeup.
EWOULDBLOCK	The process was awakened after sleeping for *timeo* clock ticks and before the matching call to wakeup.
ERESTART	A signal was handled by the process during the sleep and the pending system call should be restarted.
EINTR	A signal was handled by the process during the sleep and the pending system call should fail.

Figure 15.25   tsleep return values.

Because all processes sleeping on a wait channel are awakened by wakeup, we always see a call to tsleep within a tight loop. Every process must determine if the resource is available before proceeding because another awakened process may have claimed the resource first. If the resource is not available, the process calls tsleep once again.

It is unusual for multiple processes to be sleeping on a single socket, so a call to wakeup usually causes only one process to be awakened by the kernel.

For a more detailed discussion of the sleep and wakeup mechanism see [Leffler et al. 1989].

Example

One use of multiple processes sleeping on the same wait channel is to have multiple server processes reading from a UDP socket. Each server calls recvfrom and, as long as no data is available, the calls block in tsleep. When a datagram arrives on the socket, the socket layer calls wakeup and each server is placed on the run queue. The first server to run receives the datagram while the others call tsleep again. In this way, incoming datagrams are distributed to multiple servers without the cost of starting a new process for each datagram. This technique can also be used to process incoming connection requests in TCP by having multiple processes call accept on the same socket. This technique is described in [Comer and Stevens 1993].

15.11 accept System Call

After calling listen, a process waits for incoming connections by calling accept, which returns a descriptor that references a new socket connected to a client. The original socket, s, remains unconnected and ready to receive additional connections. accept returns the address of the foreign system if name points to a valid buffer.

The connection-processing details are handled by the protocol associated with the socket. For TCP, the socket layer is notified when a connection has been established (i.e., when TCP's three-way handshake has completed). For other protocols, such as OSI's TP4, tsleep returns when a connection request has arrived. The connection is completed when explicitly confirmed by the process by reading or writing on the socket.

Figure 15.26 shows the implementation of accept.

```
                                                                        ──── uipc_syscalls.c ────
106 struct accept_args {
107     int     s;
108     caddr_t name;
109     int     *anamelen;
110 };

111 accept(p, uap, retval)
112 struct proc *p;
113 struct accept_args *uap;
114 int     *retval;
115 {
116     struct file *fp;
117     struct mbuf *nam;
118     int     namelen, error, s;
119     struct socket *so;

120     if (uap->name && (error = copyin((caddr_t) uap->anamelen,
121                                     (caddr_t) & namelen, sizeof(namelen))))
122         return (error);
123     if (error = getsock(p->p_fd, uap->s, &fp))
124         return (error);
125     s = splnet();
126     so = (struct socket *) fp->f_data;
127     if ((so->so_options & SO_ACCEPTCONN) == 0) {
128         splx(s);
129         return (EINVAL);
130     }
131     if ((so->so_state & SS_NBIO) && so->so_qlen == 0) {
132         splx(s);
133         return (EWOULDBLOCK);
134     }
135     while (so->so_qlen == 0 && so->so_error == 0) {
136         if (so->so_state & SS_CANTRCVMORE) {
137             so->so_error = ECONNABORTED;
138             break;
139         }
140         if (error = tsleep((caddr_t) & so->so_timeo, PSOCK | PCATCH,
141                             netcon, 0)) {
142             splx(s);
143             return (error);
144         }
145     }
146     if (so->so_error) {
147         error = so->so_error;
148         so->so_error = 0;
149         splx(s);
150         return (error);
151     }
152     if (error = falloc(p, &fp, retval)) {
153         splx(s);
154         return (error);
155     }
```

```
156        { struct socket *aso = so->so_q;
157          if (soqremque(aso, 1) == 0)
158            panic("accept");
159          so = aso;
160        }

161        fp->f_type = DTYPE_SOCKET;
162        fp->f_flag = FREAD | FWRITE;
163        fp->f_ops = &socketops;
164        fp->f_data = (caddr_t) so;
165        nam = m_get(M_WAIT, MT_SONAME);
166        (void) soaccept(so, nam);
167        if (uap->name) {
168            if (namelen > nam->m_len)
169                namelen = nam->m_len;
170            /* SHOULD COPY OUT A CHAIN HERE */
171            if ((error = copyout(mtod(nam, caddr_t), (caddr_t) uap->name,
172                                 (u_int) namelen)) == 0)
173                error = copyout((caddr_t) & namelen,
174                                (caddr_t) uap->anamelen, sizeof(*uap->anamelen));
175        }
176        m_freem(nam);
177        splx(s);
178        return (error);
179 }
```
                                                                              uipc_syscalls.c

Figure 15.26 accept system call.

106–114   The three arguments to accept (in the accept_args structure) are: s, the socket descriptor; name, a pointer to a buffer to be filled in by accept with the transport address of the foreign host; and anamelen, a pointer to the size of the buffer.

Validate arguments

116–134   accept copies the size of the buffer (*anamelen) into namelen, and getsock returns the file structure for the socket. If the socket is not ready to accept connections (i.e., listen has not been called) or nonblocking I/O has been requested and no connections are queued, EINVAL or EWOULDBLOCK are returned respectively.

Wait for a connection

135–145   The while loop continues until a connection is available, an error occurs, or the socket can no longer receive data. accept is not automatically restarted after a signal is caught (tsleep returns EINTR). The protocol layer wakes up the process when it inserts a new connection on the queue with sonewconn.

Within the loop, the process waits in tsleep, which returns 0 when a connection is available. If tsleep is interrupted by a signal or the socket is set for nonblocking semantics, accept returns EINTR or EWOULDBLOCK (Figure 15.25).

Asynchronous errors

146–151   If an error occurred on the socket during the sleep, the error code is moved from the socket to the return value for accept, the socket error is cleared, and accept returns.

It is common for asynchronous events to change the state of a socket. The protocol processing layer notifies the socket layer of the change by setting `so_error` and waking any process waiting on the socket. Because of this, the socket layer must always examine `so_error` after waking to see if an error occurred while the process was sleeping.

Associate socket with descriptor

152–164   `falloc` allocates a descriptor for the new connection; the socket is removed from the accept queue by `soqremque` and attached to the `file` structure. Exercise 15.4 discusses the call to `panic`.

Protocol processing

167–179   `accept` allocates a new mbuf to hold the foreign address and calls `soaccept` to do protocol processing. The allocation and queueing of new sockets created during connection processing is described in Section 15.12. If the process provided a buffer to receive the foreign address, `copyout` copies the address from `nam` and the length from `namelen` to the process. If necessary, `copyout` silently truncates the name to fit in the process's buffer. Finally, the mbuf is released, protocol processing enabled, and `accept` returns.

Because only one mbuf is allocated for the foreign address, transport addresses must fit in one mbuf. Unix domain addresses, which are pathnames in the filesystem (up to 1023 bytes in length), may encounter this limit, but there is no problem with the 16-byte `sockaddr_in` structure for the Internet domain. The comment on line 170 indicates that this limitation could be removed by allocating and copying an mbuf chain.

`soaccept` Function

`soaccept`, shown in Figure 15.27, calls the protocol layer to retrieve the client's address for the new connection.

```
                                                                  ── uipc_socket.c
184 soaccept(so, nam)
185 struct socket *so;
186 struct mbuf *nam;
187 {
188     int     s = splnet();
189     int     error;

190     if ((so->so_state & SS_NOFDREF) == 0)
191         panic("soaccept: !NOFDREF");
192     so->so_state &= ~SS_NOFDREF;
193     error = (*so->so_proto->pr_usrreq) (so, PRU_ACCEPT,
194                         (struct mbuf *) 0, nam, (struct mbuf *) 0);
195     splx(s);
196     return (error);
197 }
                                                                  ── uipc_socket.c
```

Figure 15.27   `soaccept` function.

184-197   soaccept ensures that the socket is associated with a descriptor and issues the
PRU_ACCEPT request to the protocol. After pr_usrreq returns, nam contains the name
of the foreign socket.

15.12 sonewconn and soisconnected Functions

In Figure 15.26 we saw that accept waits for the protocol layer to process incoming
connection requests and to make them available through so_q. Figure 15.28 uses TCP
to illustrate this process.

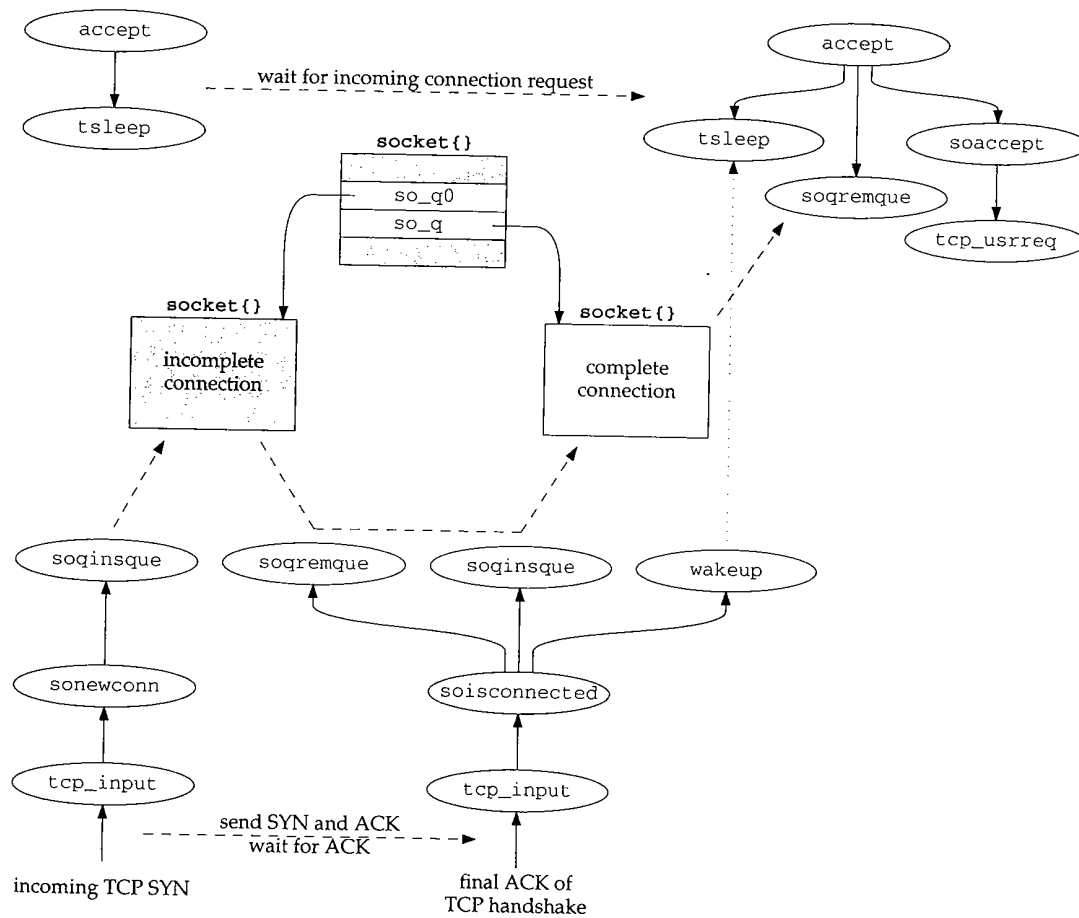**Figure 15.28** Incoming TCP connection processing.

In the upper left corner of Figure 15.28, accept calls tsleep to wait for incoming
connections. In the lower left, tcp_input processes an incoming TCP SYN by calling
sonewconn to create a socket for the new connection (Figure 28.7). sonewconn queues
the socket on so_q0, since the three-way handshake is not yet complete.

When the final ACK of the TCP handshake arrives, `tcp_input` calls `soisconnected` (Figure 29.2), which updates the new socket, moves it from `so_q0` to `so_q`, and wakes up any processes that had called `accept` to wait for incoming connections.

The upper right corner of the figure shows the functions we described with Figure 15.26. When `tsleep` returns, `accept` takes the connection off `so_q` and issues the `PRU_ATTACH` request. The socket is associated with a new file descriptor and returned to the calling process.

Figure 15.29 shows the `sonewconn` function.

—————————————————————————————————— uipc_socket2.c

```
123 struct socket *
124 sonewconn(head, connstatus)
125 struct socket *head;
126 int     connstatus;
127 {
128     struct socket *so;
129     int     soqueue = connstatus ? 1 : 0;

130     if (head->so_qlen + head->so_q0len > 3 * head->so_qlimit / 2)
131         return ((struct socket *) 0);
132     MALLOC(so, struct socket *, sizeof(*so), M_SOCKET, M_DONTWAIT);
133     if (so == NULL)
134         return ((struct socket *) 0);
135     bzero((caddr_t) so, sizeof(*so));
136     so->so_type = head->so_type;
137     so->so_options = head->so_options & ~SO_ACCEPTCONN;
138     so->so_linger = head->so_linger;
139     so->so_state = head->so_state | SS_NOFDREF;
140     so->so_proto = head->so_proto;
141     so->so_timeo = head->so_timeo;
142     so->so_pgid = head->so_pgid;
143     (void) soreserve(so, head->so_snd.sb_hiwat, head->so_rcv.sb_hiwat);
144     soqinsque(head, so, soqueue);
145     if ((*so->so_proto->pr_usrreq) (so, PRU_ATTACH,
146                 (struct mbuf *) 0, (struct mbuf *) 0, (struct mbuf *) 0)) {
147         (void) soqremque(so, soqueue);
148         (void) free((caddr_t) so, M_SOCKET);
149         return ((struct socket *) 0);
150     }
151     if (connstatus) {
152         sorwakeup(head);
153         wakeup((caddr_t) & head->so_timeo);
154         so->so_state |= connstatus;
155     }
156     return (so);
157 }
```
—————————————————————————————————— uipc_socket2.c

Figure 15.29   `sonewconn` function.

123–129   The protocol layer passes `head`, a pointer to the socket that is accepting the incoming connection, and `connstatus`, a flag to indicate the state of the new connection. For TCP, `connstatus` is always 0.

For TP4, connstatus is always SS_ISCONFIRMING. The connection is implicitly confirmed when a process begins reading from or writing to the socket.

Limit incoming connections

130–131   sonewconn prohibits additional connections when the following inequality is true:

$$so\_qlen + so\_q0len > \frac{3 \times so\_qlimit}{2}$$

This formula provides a fudge factor for connections that never complete and guarantees that listen(fd, 0) allows one connection. See Figure 18.23 in Volume 1 for an additional discussion of this formula.

Allocate new socket

132–143   A new socket structure is allocated and initialized. If the process calls setsockopt for the listening socket, the connected socket inherits several socket options because so_options, so_linger, so_pgid, and the sb_hiwat values are copied into the new socket structure.

Queue connection

144   soqueue was set from connstatus on line 129. The new socket is inserted onto so_q0 if soqueue is 0 (e.g., TCP connections) or onto so_q if connstatus is nonzero (e.g., TP4 connections).

Protocol processing

145–150   The PRU_ATTACH request is issued to perform protocol layer processing on the new connection. If this fails, the socket is dequeued and discarded, and sonewconn returns a null pointer.

Wakeup processes

151–157   If connstatus is nonzero, any processes sleeping in accept or selecting for readability on the socket are awakened. connstatus is logically ORed with so_state. This code is never executed for TCP connections, since connstatus is always 0 for TCP.

Protocols, such as TCP, that put incoming connections on so_q0 first, call soisconnected when the connection establishment phase completes. For TCP, this happens when the second SYN is ACKed on the connection.

Figure 15.30 shows soisconnected.

Queue incomplete connections

78–87   The socket state is changed to show that the connection has completed. When soisconnected is called for incoming connections, (i.e., when the local process is calling accept), head is nonnull.

If soqremque returns 1, the socket is queued on so_q and sorwakeup wakes up any processes using select to monitor the socket for connection arrival by testing for readability. If a process is blocked in accept waiting for the connection, wakeup causes the matching tsleep to return.

————————————————————————————————————— uipc_socket2.c
```
78 soisconnected(so)
79 struct socket *so;
80 {
81     struct socket *head = so->so_head;

82     so->so_state &= ~(SS_ISCONNECTING | SS_ISDISCONNECTING | SS_ISCONFIRMING);
83     so->so_state |= SS_ISCONNECTED;
84     if (head && soqremque(so, 0)) {
85         soqinsque(head, so, 1);
86         sorwakeup(head);
87         wakeup((caddr_t) & head->so_timeo);
88     } else {
89         wakeup((caddr_t) & so->so_timeo);
90         sorwakeup(so);
91         sowwakeup(so);
92     }
93 }
```
————————————————————————————————————— uipc_socket2.c

Figure 15.30   soisconnected function.

Wakeup processes waiting for new connection

88–93   If head is null, soqremque is not called since the process initiated the connection
with the connect system call and the socket is not on a queue. If head is nonnull and
soqremque returns 0, the socket is already on so_q. This happens with protocols such
as TP4, which place connections on so_q before they are complete. wakeup awakens
any process blocked in connect, and sorwakeup and sowwakeup take care of any
processes that are using select to wait for the connection to complete.

15.13 connect System call

A server process calls the listen and accept system calls to wait for a remote process
to initiate a connection. If the process wants to initiate a connection itself (i.e., a client),
it calls connect.

For connection-oriented protocols such as TCP, connect establishes a connection to
the specified foreign address. The kernel selects and implicitly binds an address to the
local socket if the process has not already done so with bind.

For connectionless protocols such as UDP or ICMP, connect records the foreign
address for use in sending future datagrams. Any previous foreign address is replaced
with the new address.

Figure 15.31 shows the functions called when connect is used for UDP or TCP.

The left side of the figure shows connect processing for connectionless protocols,
such as UDP. In this case the protocol layer calls soisconnected and the connect
system call returns immediately.

The right side of the figure shows connect processing for connection-oriented pro-
tocols, such as TCP. In this case, the protocol layer begins the connection establishment
and calls soisconnecting to indicate that the connection will complete some time in
the future. Unless the socket is nonblocking, soconnect calls tsleep to wait for the
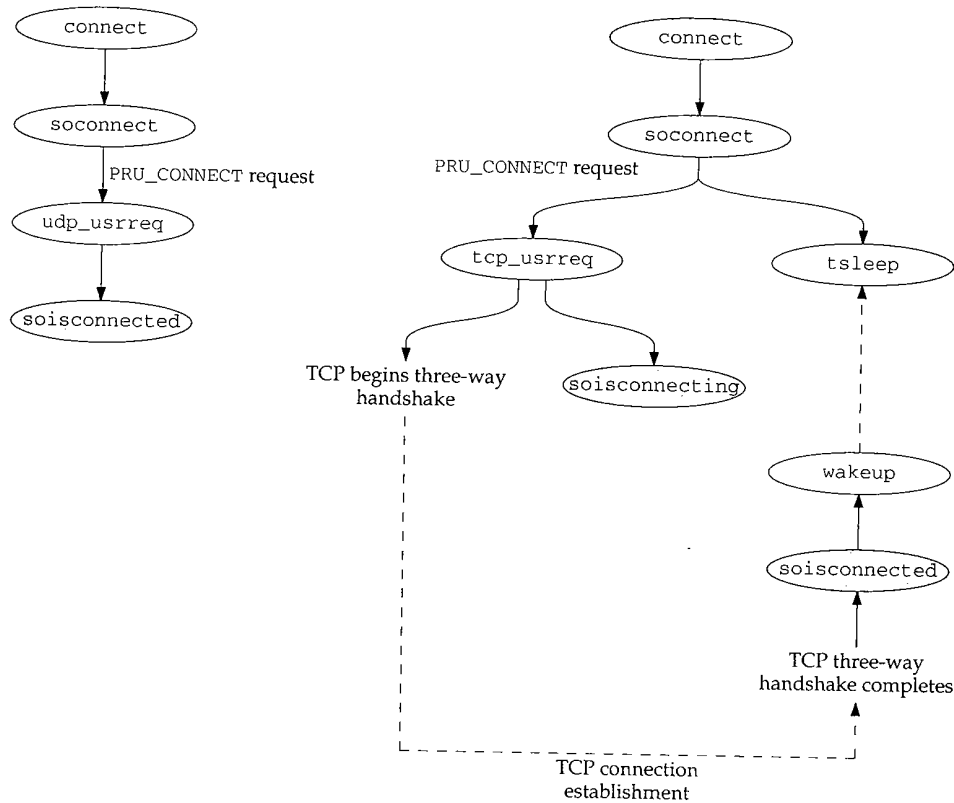
Figure 15.31   connect processing.

connection to complete. For TCP, when the three-way handshake is complete, the protocol layer calls soisconnected to mark the socket as connected and then calls wakeup to awaken the process and complete the connect system call.

Figure 15.32 shows the connect system call.

180–188   The three arguments to connect (in the connect_args structure) are: s, the socket descriptor; name, a pointer to a buffer containing the foreign address; and namelen, the length of the buffer.

189–200   getsock returns the socket as usual. A connection request may already be pending on a nonblocking socket, in which case EALREADY is returned. sockargs copies the foreign address from the process into the kernel.

Start connection processing

201–208   The connection attempt is started by calling soconnect. If soconnect reports an error, connect jumps to bad. If a connection has not yet completed by the time soconnect returns and nonblocking I/O is enabled, EINPROGRESS is returned immediately to avoid waiting for the connection to complete. Since connection establishment

uipc_syscalls.c

```
180 struct connect_args {
181     int     s;
182     caddr_t name;
183     int     namelen;
184 };

185 connect(p, uap, retval)
186 struct proc *p;
187 struct connect_args *uap;
188 int     *retval;
189 {
190     struct file *fp;
191     struct socket *so;
192     struct mbuf *nam;
193     int     error, s;

194     if (error = getsock(p->p_fd, uap->s, &fp))
195         return (error);
196     so = (struct socket *) fp->f_data;
197     if ((so->so_state & SS_NBIO) && (so->so_state & SS_ISCONNECTING))
198         return (EALREADY);
199     if (error = sockargs(&nam, uap->name, uap->namelen, MT_SONAME))
200         return (error);

201     error = soconnect(so, nam);
202     if (error)
203         goto bad;
204     if ((so->so_state & SS_NBIO) && (so->so_state & SS_ISCONNECTING)) {
205         m_freem(nam);
206         return (EINPROGRESS);
207     }
208     s = splnet();
209     while ((so->so_state & SS_ISCONNECTING) && so->so_error == 0)
210         if (error = tsleep((caddr_t) & so->so_timeo, PSOCK | PCATCH,
211                         netcon, 0))
212             break;
213     if (error == 0) {
214         error = so->so_error;
215         so->so_error = 0;
216     }
217     splx(s);
218  bad:
219     so->so_state &= ~SS_ISCONNECTING;
220     m_freem(nam);
221     if (error == ERESTART)
222         error = EINTR;
223     return (error);
224 }
```

uipc_syscalls.c

Figure 15.32   connect system call.

normally involves exchanging several packets with the remote system, it may take a while to complete. Further calls to connect return EALREADY until the connection completes. EISCONN is returned when the connection is complete.

Wait for connection establishment

208–217   The while loop continues until the connection is established or an error occurs. splnet prevents connect from missing a wakeup between testing the state of the socket and the call to tsleep. After the loop, error contains 0, the error code from tsleep, or the error from the socket.

218–224   The SS_ISCONNECTING flag is cleared since the connection has completed or the attempt has failed. The mbuf containing the foreign address is released and any error is returned.

soconnect Function

This function ensures that the socket is in a valid state for a connection request. If the socket is not connected or a connection is not pending, then the connection request is always valid. If the socket is already connected or a connection is pending, the new connection request is rejected for connection-oriented protocols such as TCP. For connectionless protocols such as UDP, multiple connection requests are OK but each new request replaces the previous foreign address.

Figure 15.33 shows the soconnect function.

```
                                                                ─── uipc_socket.c
198 soconnect(so, nam)
199 struct socket *so;
200 struct mbuf *nam;
201 {
202     int     s;
203     int     error;

204     if (so->so_options & SO_ACCEPTCONN)
205         return (EOPNOTSUPP);
206     s = splnet();
207     /*
208      * If protocol is connection-based, can only connect once.
209      * Otherwise, if connected, try to disconnect first.
210      * This allows user to disconnect by connecting to, e.g.,
211      * a null address.
212      */
213     if (so->so_state & (SS_ISCONNECTED | SS_ISCONNECTING) &&
214         ((so->so_proto->pr_flags & PR_CONNREQUIRED) ||
215          (error = sodisconnect(so))))
216         error = EISCONN;
217     else
218         error = (*so->so_proto->pr_usrreq) (so, PRU_CONNECT,
219                             (struct mbuf *) 0, nam, (struct mbuf *) 0);
220     splx(s);
221     return (error);
222 }
                                                                ─── uipc_socket.c
```

Figure 15.33 soconnect function.

198–222   `soconnect` returns `EOPNOTSUPP` if the socket is marked to accept connections, since a process cannot initiate connections if `listen` has already been called for the socket. `EISCONN` is returned if the protocol is connection oriented and a connection has already been initiated. For a connectionless protocol, any existing association with a foreign address is broken by `sodisconnect`.

The `PRU_CONNECT` request starts the appropriate protocol processing to establish the connection or the association.

Breaking a Connectionless Association

For connectionless protocols, the foreign address associated with a socket can be discarded by calling `connect` with an invalid `name` such as a pointer to a structure filled with 0s or a structure with an invalid size. `sodisconnect` removes a foreign address associated with the socket, and `PRU_CONNECT` returns an error such as `EAFNOSUPPORT` or `EADDRNOTAVAIL`, leaving the socket with no foreign address. This is a useful, although obscure, way of breaking the association between a connectionless socket and a foreign address without replacing it.

15.14 `shutdown` System Call

The `shutdown` system call, shown in Figure 15.34, closes the write-half, read-half, or both halves of a connection. For the read-half, `shutdown` discards any data the process hasn't yet read and any data that arrives after the call to `shutdown`. For the write-half, `shutdown` lets the protocol specify the semantics. For TCP, any remaining data will be sent followed by a FIN. This is TCP's half-close feature (Section 18.5 of Volume 1).

To destroy the socket and release the descriptor, `close` must be called. `close` can also be called directly without first calling `shutdown`. As with all descriptors, `close` is called by the kernel for sockets that have not been closed when a process terminates.

```
                                                                     ────── uipc_syscalls.c
550 struct shutdown_args {
551     int     s;
552     int     how;
553 };

554 shutdown(p, uap, retval)
555 struct proc *p;
556 struct shutdown_args *uap;
557 int     *retval;
558 {
559     struct file *fp;
560     int     error;

561     if (error = getsock(p->p_fd, uap->s, &fp))
562         return (error);
563     return (soshutdown((struct socket *) fp->f_data, uap->how));
564 }
                                                                     ────── uipc_syscalls.c
```

Figure 15.34   `shutdown` system call.

550–557 In the shutdown_args structure, s is the socket descriptor and how specifies which halves of the connection are to be closed. Figure 15.35 shows the expected values for how and how++ (which is used in Figure 15.36).

how	how++	Description
0	FREAD	shut down the read-half of the connection
1	FWRITE	shut down the write-half of the connection
2	FREAD|FWRITE	shut down both halves of the connection

Figure 15.35 shutdown system call options.

Notice that there is an implicit numerical relationship between how and the constants FREAD and FWRITE.

558–564 shutdown is a wrapper function for soshutdown. The socket associated with the descriptor is returned by getsock, soshutdown is called, and its value is returned.

soshutdown and sorflush Functions

The shut down of the read-half of a connection is handled in the socket layer by sorflush, and the shut down of the write-half of a connection is processed by the PRU_SHUTDOWN request in the protocol layer. The soshutdown function is shown in Figure 15.36.

```
                                                          uipc_socket.c
720 soshutdown(so, how)
721 struct socket *so;
722 int     how;
723 {
724     struct protosw *pr = so->so_proto;

725     how++;
726     if (how & FREAD)
727         sorflush(so);
728     if (how & FWRITE)
729         return ((*pr->pr_usrreq) (so, PRU_SHUTDOWN,
730                 (struct mbuf *) 0, (struct mbuf *) 0, (struct mbuf *) 0));
731     return (0);
732 }
                                                          uipc_socket.c
```

Figure 15.36 soshutdown function.

720–732 If the read-half of the socket is being closed, sorflush, shown in Figure 15.37, discards the data in the socket's receive buffer and disables the read-half of the connection. If the write-half of the socket is being closed, the PRU_SHUTDOWN request is issued to the protocol.

733–747 The process waits for a lock on the receive buffer. Because of SB_NOINTR, sblock does not return when an interrupt occurs. splimp blocks network interrupts and protocol processing while the socket is modified, since the receive buffer may be accessed by the protocol layer as it processes incoming packets.

———————————————————————————————— *uipc_socket.c*
```
733 sorflush(so)
734 struct socket *so;
735 {
736     struct sockbuf *sb = &so->so_rcv;
737     struct protosw *pr = so->so_proto;
738     int     s;
739     struct sockbuf asb;

740     sb->sb_flags |= SB_NOINTR;
741     (void) sblock(sb, M_WAITOK);
742     s = splimp();
743     socantrcvmore(so);
744     sbunlock(sb);
745     asb = *sb;
746     bzero((caddr_t) sb, sizeof(*sb));
747     splx(s);

748     if (pr->pr_flags & PR_RIGHTS && pr->pr_domain->dom_dispose)
749         (*pr->pr_domain->dom_dispose) (asb.sb_mb);
750     sbrelease(&asb);
751 }
```
———————————————————————————————— *uipc_socket.c*

Figure 15.37 sorflush function.

socantrcvmore marks the socket to reject incoming packets. A copy of the sockbuf structure is saved in asb to be used after interrupts are restored by splx. The original sockbuf structure is cleared by bzero, so that the receive queue appears to be empty.

Release control mbufs

748–751   Some kernel resources may be referenced by control information present in the receive queue when shutdown was called. The mbuf chain is still available through sb_mb in the copy of the sockbuf structure.

If the protocol supports access rights and has registered a dom_dispose function, it is called here to release these resources.

> In the Unix domain it is possible to pass descriptors between processes with control messages. These messages contain pointers to reference counted data structures. The dom_dispose function takes care of discarding the references and the data structures if necessary to avoid creating an unreferenced structure and introducing a memory leak in the kernel. For more information on passing file descriptors within the Unix domain, see [Stevens 1990] and [Leffler et al. 1989].

Any input data pending when shutdown is called is discarded when sbrelease releases any mbufs on the receive queue.

Notice that the shut down of the read-half of the connection is processed entirely by the socket layer (Exercise 15.6) and the shut down of the write-half of the connection is handled by the protocol through the PRU_SHUTDOWN request. TCP responds to the PRU_SHUTDOWN by sending all queued data and then a FIN to close the write-half of the TCP connection.

15.15 `close` System Call

The `close` system call works with any type of descriptor. When `fd` is the last descriptor that references the object, the object-specific `close` function is called:

```
error = (*fp->f_ops->fo_close)(fp, p);
```

As shown in Figure 15.13, `fp->f_ops->fo_close` for a socket is the function `soo_close`.

`soo_close` Function

This function, shown in Figure 15.38, is a wrapper for the `soclose` function.

```
                                                                  ─── sys_socket.c
152 soo_close(fp, p)
153 struct file *fp;
154 struct proc *p;
155 {
156     int       error = 0;

157     if (fp->f_data)
158         error = soclose((struct socket *) fp->f_data);
159     fp->f_data = 0;
160     return (error);
161 }
                                                                  ─── sys_socket.c
```

Figure 15.38   `soo_close` function.

152–161   If a `socket` structure is associated with the `file` structure, `soclose` is called, `f_data` is cleared, and any posted error is returned.

`soclose` Function

This function aborts any connections that are pending on the socket (i.e., that have not yet been accepted by a process), waits for data to be transmitted to the foreign system, and releases the data structures that are no longer needed.

`soclose` is shown in Figure 15.39.

Discard pending connections

129–141   If the socket was accepting connections, `soclose` traverses the two connection queues and calls `soabort` for each pending connection. If the protocol control block is null, the protocol has already been detached from the socket and `soclose` jumps to the cleanup code at `discard`.

> `soabort` issues the `PRU_ABORT` request to the socket's protocol and returns the result. `soabort` is not shown in this text. Figures 23.38 and 30.7 discuss how UDP and TCP handle this request.

─── *uipc_socket.c*

```
129 soclose(so)
130 struct socket *so;
131 {
132     int     s = splnet();          /* conservative */
133     int     error = 0;

134     if (so->so_options & SO_ACCEPTCONN) {
135         while (so->so_q0)
136             (void) soabort(so->so_q0);
137         while (so->so_q)
138             (void) soabort(so->so_q);
139     }
140     if (so->so_pcb == 0)
141         goto discard;
142     if (so->so_state & SS_ISCONNECTED) {
143         if ((so->so_state & SS_ISDISCONNECTING) == 0) {
144             error = sodisconnect(so);
145             if (error)
146                 goto drop;
147         }
148         if (so->so_options & SO_LINGER) {
149             if ((so->so_state & SS_ISDISCONNECTING) &&
150                 (so->so_state & SS_NBIO))
151                 goto drop;
152             while (so->so_state & SS_ISCONNECTED)
153                 if (error = tsleep((caddr_t) & so->so_timeo,
154                                    PSOCK | PCATCH, netcls, so->so_linger))
155                     break;
156         }
157     }
158 drop:
159     if (so->so_pcb) {
160         int     error2 =
161         (*so->so_proto->pr_usrreq) (so, PRU_DETACH,
162                 (struct mbuf *) 0, (struct mbuf *) 0, (struct mbuf *) 0);
163         if (error == 0)
164             error = error2;
165     }
166 discard:
167     if (so->so_state & SS_NOFDREF)
168         panic("soclose: NOFDREF");
169     so->so_state |= SS_NOFDREF;
170     sofree(so);
171     splx(s);
172     return (error);
173 }
```
─── *uipc_socket.c*

Figure 15.39 soclose function.

Break established connection or association

142–157   If the socket is not connected, execution continues at drop; otherwise the socket must be disconnected from its peer. If a disconnect is not in progress, sodisconnect starts the disconnection process. If the SO_LINGER socket option is set, soclose may need to wait for the disconnect to complete before returning. A nonblocking socket never waits for a disconnect to complete, so soclose jumps immediately to drop in that case. Otherwise, the connection termination is in progress and the SO_LINGER option indicates that soclose must wait some time for it to complete. The while loop continues until the disconnect completes, the linger time (so_linger) expires, or a signal is delivered to the process.

> If the linger time is set to 0, tsleep returns only when the disconnect completes (perhaps because of an error) or a signal is delivered.

Release data structures

158–173   If the socket still has an attached protocol, the PRU_DETACH request breaks the connection between this socket and the protocol. Finally the socket is marked as not having an associated file descriptor, which allows sofree to release the socket.

The sofree function is shown in Figure 15.40.

```
                                                                    ── uipc_socket.c
110 sofree(so)
111 struct socket *so;
112 {
113     if (so->so_pcb || (so->so_state & SS_NOFDREF) == 0)
114         return;
115     if (so->so_head) {
116         if (!soqremque(so, 0) && !soqremque(so, 1))
117             panic("sofree dq");
118         so->so_head = 0;
119     }
120     sbrelease(&so->so_snd);
121     sorflush(so);
122     FREE(so, M_SOCKET);
123 }
                                                                    ── uipc_socket.c
```

Figure 15.40 sofree function.

Return if socket still in use

110–114   If a protocol is still associated with the socket, or if the socket is still associated with a descriptor, sofree returns immediately.

Remove from connection queues

115–119   If the socket is on a connection queue (so_head is nonnull), soqremque is called to remove the socket. An attempt is made to remove the socket from the incomplete connection queue and if this fails, then from the completed connection queue. One of the removals must succeed or the kernel panics, since so_head was nonnull. so_head is cleared.

Discard send and receive queues

120–123   `sbrelease` discards any buffers in the send queue and `sorflush` discards any buffers in the receive queue. Finally, the socket itself is released.

15.16 Summary

In this chapter we looked at all the system calls related to network operations. The system call mechanism was described, and we traced the calls until they entered the protocol processing layer through the `pr_usrreq` function.

While looking at the socket layer, we avoided any discussion of address formats, protocol semantics, or protocol implementations. In the upcoming chapters we tie together the link-layer processing and socket-layer processing by looking in detail at the implementation of the Internet protocols in the protocol processing layer.

Exercises

15.1   How can a process *without* superuser privileges gain access to a socket created by a superuser process?

15.2   How can a process determine if the `sockaddr` buffer it provides to `accept` was too small to hold the foreign address returned by the call?

15.3   A feature proposed for IPv6 sockets is to have `accept` and `recvfrom` return a source route as an array of 128-bit IPv6 addresses instead of a single peer address. Since the array will not fit in a single mbuf, modify `accept` and `recvfrom` to handle an mbuf chain from the protocol layer instead of a single mbuf. Will the existing code work if the protocol layer returns the array in an mbuf cluster instead of a chain of mbufs?

15.4   Why is `panic` called when `soqremque` returns a null pointer in Figure 15.26?

15.5   Why does `sorflush` make a copy of the receive buffer?

15.6   What happens when additional data is received after `sorflush` has zeroed the socket's receive buffer? Read Chapter 16 before attempting this exercise.

16

Socket I/O

16.1 Introduction

In this chapter we discuss the system calls that read and write data on a network connection. The chapter is divided into three parts.

The first part covers the four system calls for sending data: `write`, `writev`, `sendto`, and `sendmsg`. The second part covers the four system calls for receiving data: `read`, `readv`, `recvfrom`, and `recvmsg`. The third part of the chapter covers the `select` system call, which provides a standard way to monitor the status of descriptors in general and sockets in particular.

The core of the socket layer is the `sosend` and `soreceive` functions. They handle all I/O between the socket layer and the protocol layer. As we'll see, the semantics of the various types of protocols overlap in these functions, making the functions long and complex.

16.2 Code Introduction

The three headers and four C files listed in Figure 16.1 are covered in this chapter.

Global Variables

The first two global variables shown in Figure 16.2 are used by the `select` system call. The third global variable controls the amount of memory allocated to a socket.

475

File	Description
sys/socket.h sys/socketvar.h sys/uio.h	structures and macro for sockets API socket structure and macros uio structure definition
kern/uipc_syscalls.c kern/uipc_socket.c kern/sys_generic.c kern/sys_socket.c	socket system calls socket layer processing select system call select processing for sockets

Figure 16.1   Files discussed in this chapter.

Variable	Datatype	Description
selwait	int	wait channel for select
nselcoll	int	flag used to avoid race conditions in select
sb_max	u_long	maximum number of bytes to allocate for a socket receive or send buffer

Figure 16.2   Global variables introduced in this chapter.

16.3 Socket Buffers

Section 15.3 showed that each socket has an associated send and receive buffer. The sockbuf structure definition from Figure 15.5 is repeated in Figure 16.3.

```
                                                                      socketvar.h
72      struct sockbuf {
73          u_long  sb_cc;         /* actual chars in buffer */
74          u_long  sb_hiwat;      /* max actual char count */
75          u_long  sb_mbcnt;      /* chars of mbufs used */
76          u_long  sb_mbmax;      /* max chars of mbufs to use */
77          long    sb_lowat;      /* low water mark */
78          struct mbuf *sb_mb;    /* the mbuf chain */
79          struct selinfo sb_sel; /* process selecting read/write */
80          short   sb_flags;      /* Figure 16.5 */
81          short   sb_timeo;      /* timeout for read/write */
82      } so_rcv, so_snd;
                                                                      socketvar.h
```

Figure 16.3   sockbuf structure.

72–78   Each buffer contains control information as well as pointers to data stored in mbuf chains. sb_mb points to the first mbuf in the chain, and sb_cc is the total number of data bytes contained within the mbufs. sb_hiwat and sb_lowat regulate the socket flow control algorithms. sb_mbcnt is the total amount of memory allocated to the mbufs in the buffer.

Recall that each mbuf may store from 0 to 2048 bytes of data (if an external cluster is used). sb_mbmax is an upper bound on the amount of memory to be allocated as

mbufs for each socket buffer. Default limits are specified by each protocol when the PRU_ATTACH request is issued by the socket system call. The high-water and low-water marks may be modified by the process as long as the kernel-enforced hard limit of 262,144 bytes per socket buffer (sb_max) is not exceeded. The buffering algorithms are described in Sections 16.7 and 16.12. Figure 16.4 shows the default settings for the Internet protocols.

Protocol	so_snd			so_rcv		
	sb_hiwat	sb_lowat	sb_mbmax	sb_hiwat	sb_lowat	sb_mbmax
UDP	9×1024	2048 (ignored)	$2 \times$ sb_hiwat	$40 \times (1024 + 16)$	1	$2 \times$ sb_hiwat
TCP	8×1024	2048	$2 \times$ sb_hiwat	8×1024	1	$2 \times$ sb_hiwat
raw IP ICMP IGMP	8×1024	2048 (ignored)	$2 \times$ sb_hiwat	8×1024	1	$2 \times$ sb_hiwat

Figure 16.4   Default socket buffer limits for the Internet protocols.

Since the source address of each incoming UDP datagram is queued with the data (Section 23.8), the default UDP value for sb_hiwat is set to accommodate 40 1K datagrams and their associated sockaddr_in structures (16 bytes each).

79   sb_sel is a selinfo structure used to implement the select system call (Section 16.13).

80   Figure 16.5 lists the possible values for sb_flags.

sb_flags	Description
SB_LOCK SB_WANT SB_WAIT	a process has locked the socket buffer a process is waiting to lock the buffer a process is waiting for data (receive) or space (send) in this buffer
SB_SEL SB_ASYNC SB_NOINTR	one or more processes are selecting on this buffer generate asynchronous I/O signal for this buffer signals do not cancel a lock request
SB_NOTIFY	(SB_WAIT | SB_SEL | SB_ASYNC) a process is waiting for changes to the buffer and should be notified by wakeup when any changes occur

Figure 16.5   sb_flags values.

81–82   sb_timeo is measured in clock ticks and limits the time a process blocks during a read or write call. The default value of 0 causes the process to wait indefinitely. sb_timeo may be changed or retrieved by the SO_SNDTIMEO and SO_RCVTIMEO socket options.

Socket Macros and Functions

There are many macros and functions that manipulate the send and receive buffers associated with each socket. The macros and functions in Figure 16.6 handle buffer locking and synchronization.

Name	Description
sblock	Acquires a lock for *sb*. If *wf* is M_WAITOK, the process sleeps waiting for the lock; otherwise EWOULDBLOCK is returned if the buffer cannot be locked immediately. EINTR or ERESTART is returned if the sleep is interrupted by a signal; 0 is returned otherwise. int **sblock**(struct sockbuf *sb, int *wf*);
sbunlock	Releases the lock on sb. Any other process waiting to lock *sb* is awakened. void **sbunlock**(struct sockbuf *sb);
sbwait	Calls tsleep to wait for protocol activity on *sb*. Returns result of tsleep. int **sbwait**(struct sockbuf *sb);
sowakeup	Notifies socket of protocol activity. Wakes up matching call to sbwait or to tsleep if any processes are selecting on *sb*. void **sowakeup**(struct socket *so, struct sockbuf *sb);
sorwakeup	Wakes up any process waiting for read events on *so* and sends the SIGIO signal if a process requested asynchronous notification of I/O. void **sorwakeup**(struct socket *so);
sowwakeup	Wakes up any process waiting for write events on *so* and sends the SIGIO signal if a process requested asynchronous notification of I/O. void **sowwakeup**(struct socket *so);

Figure 16.6   Macros and functions for socket buffer locking and synchronization.

Figure 16.7 includes the macros and functions used to set the resource limits for socket buffers and to append and delete data from the buffers. In the table, *m, m0, n*, and *control* are all pointers to mbuf chains. *sb* points to the send or receive buffer for a socket.

Name	Description
sbspace	The number of bytes that may be added to *sb* before it is considered full: min((sb_hiwat - sb_cc), (sb_mbmax - sb_mbcnt)). long **sbspace**(struct sockbuf *sb);
sballoc	*m* has been added to *sb*. Adjust sb_cc and sb_mbcnt in *sb* accordingly. void **sballoc**(struct sockbuf *sb, struct mbuf *m);
sbfree	m has been removed from sb. Adjust sb_cc and sb_mbcnt in sb accordingly. int **sbfree**(struct sockbuf *sb, struct mbuf *m);

Name	Description
sbappend	Append the mbufs in *m* to the end of the last record in *sb*. Call sbcompress. int **sbappend**(struct sockbuf *sb, struct mbuf *m);
sbappendrecord	Append the record in *m0* after the last record in *sb*. Call sbcompress. int **sbappendrecord**(struct sockbuf *sb, struct mbuf *m0);
sbappendaddr	Put address from *asa* in an mbuf. Concatenate address, *control*, and *m0*. Append the resulting mbuf chain after the last record in *sb*. int **sbappendaddr**(struct sockbuf *sb, struct sockaddr *asa, struct mbuf *m0, struct mbuf *control);
sbappendcontrol	Concatenate *control* and *m0*. Append the resulting mbuf chain after the last record in *sb*. int **sbappendcontrol**(struct sockbuf *sb, struct mbuf *m0, struct mbuf *control);
sbinsertoob	Insert *m0* before first record in *sb* without out-of-band data. Call sbcompress. int **sbinsertoob**(struct sockbuf *sb, struct mbuf *m0);
sbcompress	Append *m* to *n* squeezing out any unused space. void **sbcompress**(struct sockbuf *sb, struct mbuf *m, struct mbuf *n);
sbdrop	Discard *len* bytes from the front of *sb*. void **sbdrop**(struct sockbuf *sb, int len);
sbdroprecord	Discard the first record in *sb*. Move the next record to the front. void **sbdroprecord**(struct sockbuf *sb);
sbrelease	Call sbflush to release all mbufs in *sb*. Reset sb_hiwat and sb_mbmax values to 0. void **sbrelease**(struct sockbuf *sb);
sbflush	Release all mbufs in *sb*. void **sbflush**(struct sockbuf *sb);
soreserve	Set high-water and low-water marks. For the send buffer, call sbreserve with *sndcc*. For the receive buffer, call sbreserve with *rcvcc*. Initialize sb_lowat in both buffers to default values, Figure 16.4. ENOBUFS is returned if any limits are exceeded. int **soreserve**(struct socket *so, int sndcc, int rcvcc);
sbreserve	Set high-water mark for *sb* to *cc*. Also drop low-water mark to *cc*. No memory is allocated by this function. int **sbreserve**(struct sockbuf *sb, int cc);

Figure 16.7   Macros and functions for socket buffer allocation and manipulation.

16.4 `write`, `writev`, `sendto`, and `sendmsg` System Calls

These four system calls, which we refer to collectively as the *write system calls*, send data on a network connection. The first three system calls are simpler interfaces to the most general request, `sendmsg`.

All the write system calls, directly or indirectly, call `sosend`, which does the work of copying data from the process to the kernel and passing data to the protocol associated with the socket. Figure 16.8 summarizes the flow of control.
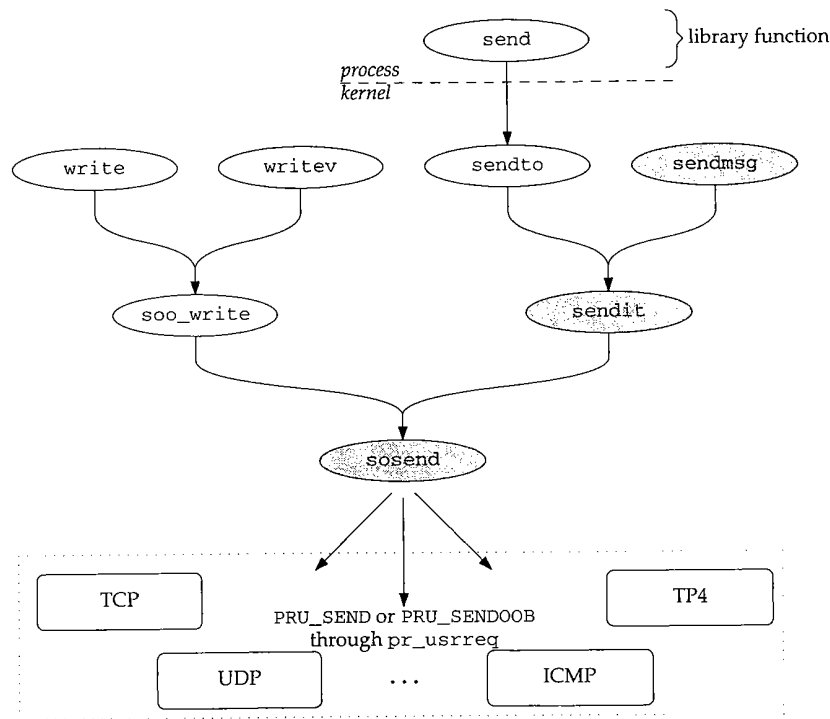

Figure 16.8   All socket output is handled by `sosend`.

In the following sections, we discuss the functions shaded in Figure 16.8. The other four system calls and `soo_write` are left for readers to investigate on their own.

Figure 16.9 shows the features of these four system calls and a related library function (`send`).

> In Net/3, `send` is implemented as a library function that calls `sendto`. For binary compatibility with previously compiled programs, the kernel maps the old `send` system call to the function `osend`, which is not discussed in this text.

From the second column in Figure 16.9 we see that the `write` and `writev` system calls are valid with any descriptor, but the remaining system calls are valid only with socket descriptors.

Function	Type of descriptor	Number of buffers	Specify destination address?	Flags?	Control information?
`write`	any	1			
`writev`	any	[1..`UIO_MAXIOV`]			
`send`	socket only	1		•	
`sendto`	socket only	1	•	•	
`sendmsg`	socket only	[1..`UIO_MAXIOV`]	•	•	•

Figure 16.9 Write system calls.

The third column shows that `writev` and `sendmsg` accept data from multiple buffers. Writing from multiple buffers is called *gathering*. The analogous read operation is called *scattering*. In a gather operation the kernel accepts, in order, data from each buffer specified in an array of `iovec` structures. The array can have a maximum of `UIO_MAXIOV` elements. The structure is shown in Figure 16.10.

```
                                                                  uio.h
41  struct iovec {
42      char   *iov_base;        /* Base address */
43      size_t  iov_len;         /* Length */
44  };
                                                                  uio.h
```

Figure 16.10 `iovec` structure.

41–44   `iov_base` points to the start of a buffer of `iov_len` bytes.

Without this type of interface, a process would have to copy buffers into a single larger buffer or make multiple write system calls to send data from multiple buffers. Both alternatives are less efficient than passing an array of `iovec` structures to the kernel in a single call. With datagram protocols, the result of one `writev` is one datagram, which cannot be emulated with multiple writes.

Figure 16.11 illustrates the structures as they are used by `writev`, where `iovp` points to the first element of the array and `iovcnt` is the size of the array.
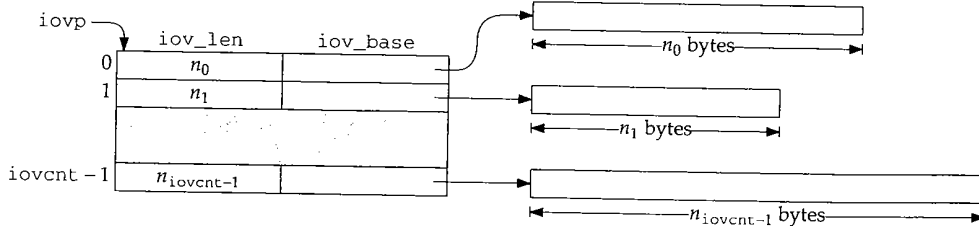

Figure 16.11 `iovec` arguments to `writev`.

Datagram protocols require a destination address to be associated with each write call. Since `write`, `writev`, and `send` do not accept an explicit destination, they may be called only after a destination has been associated with a connectionless socket by calling `connect`. A destination must be provided with `sendto` or `sendmsg`, or `connect` must have been previously called.

The fifth column in Figure 16.9 shows that the send*xxx* system calls accept optional control flags, which are described in Figure 16.12.

flags	Description	Reference
MSG_DONTROUTE	bypass routing tables for this message	Figure 16.23
MSG_DONTWAIT	do not wait for resources during this message	Figure 16.22
MSG_EOR	data marks the end of a logical record	Figure 16.25
MSG_OOB	send as out-of-band data	Figure 16.26

Figure 16.12   send*xxx* system calls: flags values.

As indicated in the last column of Figure 16.9, only the sendmsg system call supports control information. The control information and several other arguments to sendmsg are specified within a msghdr structure (Figure 16.13) instead of being passed separately.

-- *socket.h*
```
228 struct msghdr {
229     caddr_t msg_name;          /* optional address */
230     u_int   msg_namelen;       /* size of address */
231     struct iovec *msg_iov;     /* scatter/gather array */
232     u_int   msg_iovlen;        /* # elements in msg_iov */
233     caddr_t msg_control;       /* ancillary data, see below */
234     u_int   msg_controllen;    /* ancillary data buffer len */
235     int     msg_flags;         /* Figure 16.33 */
236 };
```
-- *socket.h*

Figure 16.13   msghdr structure.

msg_name should be declared as a pointer to a sockaddr structure, since it contains a network address.

228–236   The msghdr structure contains a destination address (msg_name and msg_namelen), a scatter/gather array (msg_iov and msg_iovlen), control information (msg_control and msg_controllen), and receive flags (msg_flags). The control information is formatted as a cmsghdr structure shown in Figure 16.14.

-- *socket.h*
```
251 struct cmsghdr {
252     u_int   cmsg_len;          /* data byte count, including hdr */
253     int     cmsg_level;        /* originating protocol */
254     int     cmsg_type;         /* protocol-specific type */
255 /* followed by  u_char  cmsg_data[]; */
256 };
```
-- *socket.h*

Figure 16.14   cmsghdr structure.

251–256   The control information is not interpreted by the socket layer, but the messages are typed (cmsg_type) and they have an explicit length (cmsg_len). Multiple control messages may appear in the control information mbuf.

Example

Figure 16.15 shows how a fully specified `msghdr` structure might look during a call to `sendmsg`.
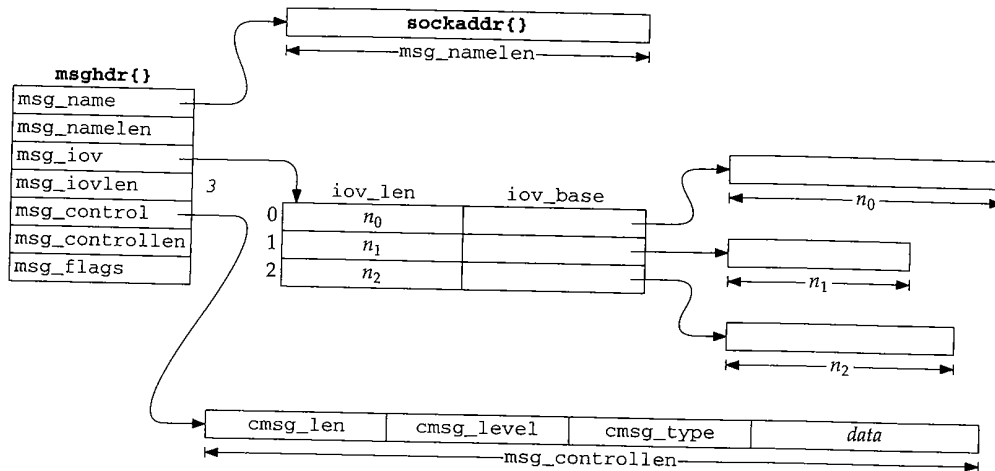

Figure 16.15   `msghdr` structure for `sendmsg` system call.

16.5   `sendmsg` System Call

Only the `sendmsg` system call provides access to all the features of the sockets API associated with output. The `sendmsg` and `sendit` functions prepare the data structures needed by `sosend`, which passes the message to the appropriate protocol. For `SOCK_DGRAM` protocols, a message is a datagram. For `SOCK_STREAM` protocols, a message is a sequence of bytes. For `SOCK_SEQPACKET` protocols, a message could be an entire record (implicit record boundaries) or part of a larger record (explicit record boundaries). A message is always an entire record (implicit record boundaries) for `SOCK_RDM` protocols.

> Even though the general `sosend` code handles `SOCK_SEQPACKET` and `SOCK_RDM` protocols, there are no such protocols in the Internet domain.

Figure 16.16 shows the `sendmsg` code.

307–321   There are three arguments to `sendmsg`: the socket descriptor; a pointer to a `msghdr` structure; and several control flags. The `copyin` function copies the `msghdr` structure from user space to the kernel.

Copy `iov` array

322–334   An `iovec` array with eight entries (`UIO_SMALLIOV`) is allocated automatically on the stack. If this is not large enough, `sendmsg` calls `MALLOC` to allocate a larger array. If

---

uipc_syscalls.c
```
307 struct sendmsg_args {
308     int     s;
309     caddr_t msg;
310     int     flags;
311 };

312 sendmsg(p, uap, retval)
313 struct proc *p;
314 struct sendmsg_args *uap;
315 int    *retval;
316 {
317     struct msghdr msg;
318     struct iovec aiov[UIO_SMALLIOV], *iov;
319     int     error;

320     if (error = copyin(uap->msg, (caddr_t) & msg, sizeof(msg)))
321         return (error);
322     if ((u_int) msg.msg_iovlen >= UIO_SMALLIOV) {
323         if ((u_int) msg.msg_iovlen >= UIO_MAXIOV)
324             return (EMSGSIZE);
325         MALLOC(iov, struct iovec *,
326               sizeof(struct iovec) * (u_int) msg.msg_iovlen, M_IOV,
327               M_WAITOK);
328     } else
329         iov = aiov;
330     if (msg.msg_iovlen &&
331         (error = copyin((caddr_t) msg.msg_iov, (caddr_t) iov,
332                     (unsigned) (msg.msg_iovlen * sizeof(struct iovec)))))
333             goto done;
334     msg.msg_iov = iov;
335     error = sendit(p, uap->s, &msg, uap->flags, retval);
336 done:
337     if (iov != aiov)
338         FREE(iov, M_IOV);
339     return (error);
340 }
```
uipc_syscalls.c

Figure 16.16   sendmsg system call.

the process specifies an array with more than 1024 (UIO_MAXIOV) entries, EMSGSIZE is returned. copyin places a copy of the iovec array from user space into either the array on the stack or the larger, dynamically allocated, array.

> This technique avoids the relatively expensive call to malloc in the most common case of eight or fewer entries.

sendit and cleanup

335–340   When sendit returns, the data has been delivered to the appropriate protocol or an error has occurred. sendmsg releases the iovec array (if it was dynamically allocated) and returns sendit's result.

16.6 `sendit` Function

`sendit` is the common function called by `sendto` and `sendmsg`. `sendit` initializes a `uio` structure and copies control and address information from the process into the kernel. Before discussing `sosend`, we must explain the `uiomove` function and the `uio` structure.

`uiomove` Function

The prototype for this function is:

```
int uiomove(caddr_t cp, int n, struct uio *uio);
```

The `uiomove` function moves *n* bytes between a single buffer referenced by *cp* and the multiple buffers specified by an *iovec* array in *uio*. Figure 16.17 shows the definition of the `uio` structure, which controls and records the actions of the `uiomove` function.

```
                                                                          ──uio.h
45 enum uio_rw {
46     UIO_READ, UIO_WRITE
47 };

48 enum uio_seg {              /* Segment flag values */
49     UIO_USERSPACE,          /* from user data space */
50     UIO_SYSSPACE,           /* from system space */
51     UIO_USERISPACE          /* from user instruction space */
52 };

53 struct uio {
54     struct iovec *uio_iov;  /* an array of iovec structures */
55     int     uio_iovcnt;     /* size of iovec array */
56     off_t   uio_offset;     /* starting position of transfer */
57     int     uio_resid;      /* remaining bytes to transfer */
58     enum uio_seg uio_segflg; /* location of buffers */
59     enum uio_rw uio_rw;     /* direction of transfer */
60     struct proc *uio_procp; /* the associated process */
61 };
                                                                          ──uio.h
```

Figure 16.17 `uio` structure.

45–61   In the `uio` structure, `uio_iov` points to an array of `iovec` structures, `uio_offset` counts the number of bytes transferred by `uiomove`, and `uio_resid` counts the number of bytes remaining to be transferred. Each time `uiomove` is called, `uio_offset` increases by *n* and `uio_resid` decreases by *n*. `uiomove` adjusts the base pointers and buffer lengths in the `uio_iov` array to exclude any bytes that `uiomove` transfers each time it is called. Finally, `uio_iov` is advanced through each entry in the array as each buffer is transferred. `uio_segflg` indicates the location of the buffers specified by the base pointers in the `uio_iov` array and `uio_rw` indicates the direction of the transfer. The buffers may be located in the user data space, user instruction space, or kernel data space. Figure 16.18 summarizes the operation of `uiomove`. The descriptions use the argument names shown in the `uiomove` prototype.

uio_segflg	uio_rw	Description
UIO_USERSPACE	*UIO_READ*	scatter *n* bytes from a kernel buffer *cp* to process buffers
UIO_USERISPACE		
UIO_USERSPACE	*UIO_WRITE*	gather *n* bytes from process buffers into the kernel buffer *cp*
UIO_USERISPACE		
UIO_SYSSPACE	*UIO_READ*	scatter *n* bytes from the kernel buffer *cp* to multiple kernel buffers
	UIO_WRITE	gather *n* bytes from multiple kernel buffers into the kernel buffer *cp*

Figure 16.18   uiomove operation.

Example

Figure 16.19 shows a uio structure before uiomove is called.

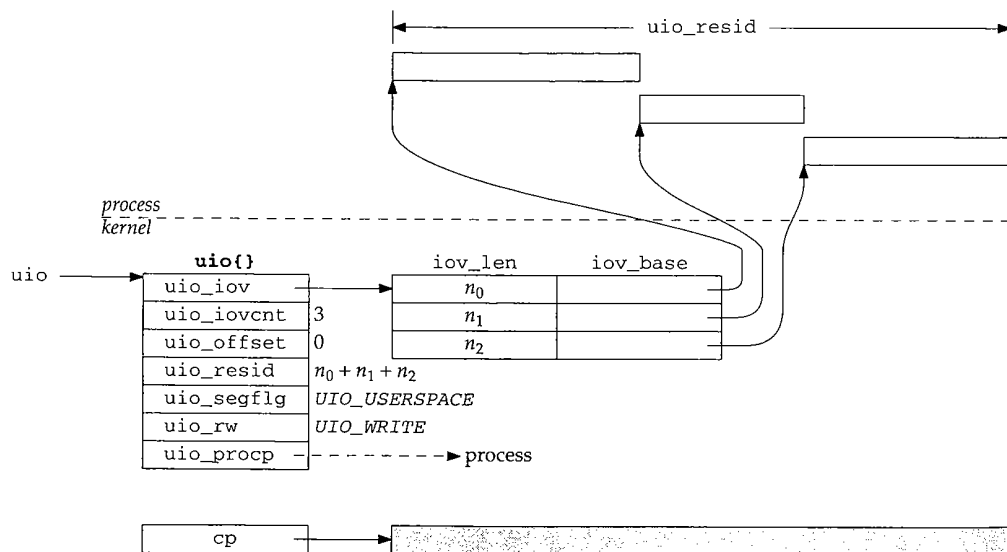**Figure 16.19**   uiomove: before.

uio_iov points to the first entry in the iovec array. Each of the iov_base pointers point to the start of their respective buffer in the address space of the process. uio_offset is 0, and uio_resid is the sum of size of the three buffers. cp points to a buffer within the kernel, typically the data area of an mbuf. Figure 16.20 shows the same data structures after

```
uiomove(cp, n, uio);
```

is executed where n includes all the bytes from the first buffer and only some of the bytes from the second buffer (i.e., $n_0 < n < n_0 + n_1$).
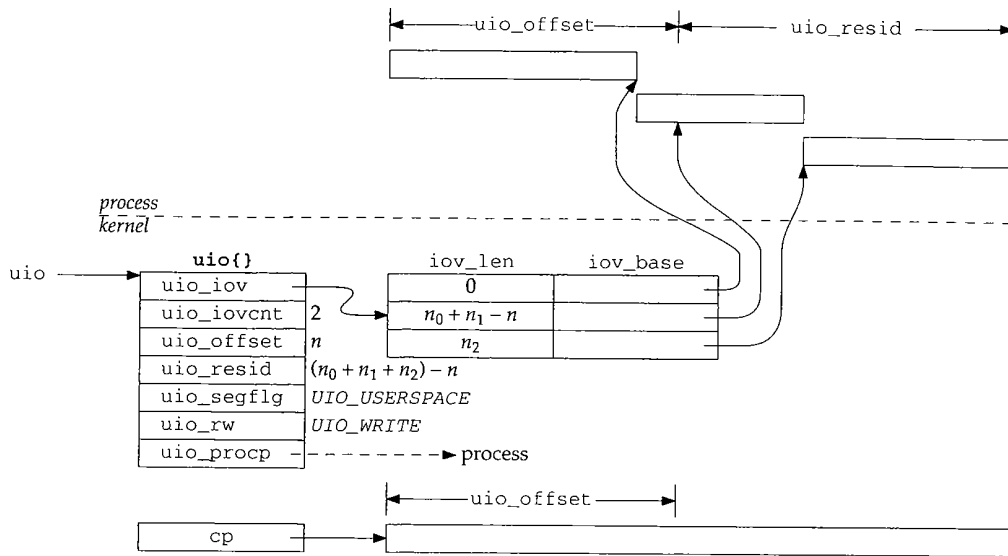
Figure 16.20 uiomove: after.

After uiomove, the first buffer has a length of 0 and its base pointer has been advanced to the end of the buffer. uio_iov now points to the second entry in the iovec array. The pointer in this entry has been advanced and the length decreased to reflect the transfer of some of the bytes in the buffer. uio_offset has been increased by n and uio_resid has been decreased by n. The data from the buffers in the process has been moved into the kernel's buffer because uio_rw was UIO_WRITE.

sendit Code

We can now discuss the sendit code shown in Figure 16.21.

Initialize auio

341–368   sendit calls getsock to get the file structure associated with the descriptor s and initializes the uio structure to gather the output buffers specified by the process into mbufs in the kernel. The length of the transfer is calculated by the for loop as the sum of the buffer lengths and saved in uio_resid. The first if within the loop ensures that the buffer length is nonnegative. The second if ensures that uio_resid does not overflow, since uio_resid is a signed integer and iov_len is guaranteed to be nonnegative.

Copy address and control information from the process

369–385   sockargs makes copies of the destination address and control information into mbufs if they are provided by the process.

```
341 sendit(p, s, mp, flags, retsize)
342 struct proc *p;
343 int     s;
344 struct msghdr *mp;
345 int     flags, *retsize;
346 {
347     struct file *fp;
348     struct uio auio;
349     struct iovec *iov;
350     int     i;
351     struct mbuf *to, *control;
352     int     len, error;
353     if (error = getsock(p->p_fd, s, &fp))
354         return (error);
355     auio.uio_iov = mp->msg_iov;
356     auio.uio_iovcnt = mp->msg_iovlen;
357     auio.uio_segflg = UIO_USERSPACE;
358     auio.uio_rw = UIO_WRITE;
359     auio.uio_procp = p;
360     auio.uio_offset = 0;            /* XXX */
361     auio.uio_resid = 0;
362     iov = mp->msg_iov;
363     for (i = 0; i < mp->msg_iovlen; i++, iov++) {
364         if (iov->iov_len < 0)
365             return (EINVAL);
366         if ((auio.uio_resid += iov->iov_len) < 0)
367             return (EINVAL);
368     }
369     if (mp->msg_name) {
370         if (error = sockargs(&to, mp->msg_name, mp->msg_namelen,
371                             MT_SONAME))
372             return (error);
373     } else
374         to = 0;
375     if (mp->msg_control) {
376         if (mp->msg_controllen < sizeof(struct cmsghdr)
377         ) {
378             error = EINVAL;
379             goto bad;
380         }
381         if (error = sockargs(&control, mp->msg_control,
382                             mp->msg_controllen, MT_CONTROL))
383             goto bad;
384     } else
385         control = 0;
386     len = auio.uio_resid;
387     if (error = sosend((struct socket *) fp->f_data, to, &auio,
388                         (struct mbuf *) 0, control, flags)) {
389         if (auio.uio_resid != len && (error == ERESTART ||
390                                 error == EINTR || error == EWOULDBLOCK))
391             error = 0;
392         if (error == EPIPE)
393             psignal(p, SIGPIPE);
```

```
394        }
395        if (error == 0)
396            *retsize = len - auio.uio_resid;
397    bad:
398        if (to)
399            m_freem(to);
400        return (error);
401    }
```
                                                                    —— uipc_syscalls.c

Figure 16.21 sendit function.

Send data and cleanup

386–401   uio_resid is saved in len so that the number of bytes transferred can be calcu-
lated if sosend does not accept all the data. The socket, destination address, uio struc-
ture, control information, and flags are all passed to sosend. When sosend returns,
sendit responds as follows:

- If sosend transfers some data and is interrupted by a signal or a blocking condi-
  tion, the error is discarded and the partial transfer is reported.

- If sosend returns EPIPE, the SIGPIPE signal is sent to the process. error is
  not set to 0, so if a process catches the signal and the signal handler returns, or if
  the process ignores the signal, the write call returns EPIPE.

- If no error occurred (or it was discarded), the number of bytes transferred is cal-
  culated and saved in *retsize. Since sendit returns 0, syscall (Sec-
  tion 15.4) returns *retsize to the process instead of returning the error code.

- If any other error occurs, the error code is returned to the process.

Before returning, sendit releases the mbuf containing the destination address.
sosend is responsible for releasing the control mbuf.

16.7 sosend Function

sosend is one of the most complicated functions in the socket layer. Recall from Fig-
ure 16.8 that all five write calls eventually call sosend. It is sosend's responsibility to
pass the data and control information to the pr_usrreq function of the protocol associ-
ated with the socket according to the semantics supported by the protocol and the buff-
er limits specified by the socket. sosend never places data in the send buffer; it is the
protocol's responsibility to store and remove the data.

The interpretation of the send buffer's sb_hiwat and sb_lowat values by
sosend depends on whether the associated protocol implements reliable or unreliable
data transfer semantics.

Reliable Protocol Buffering

For reliable protocols, the send buffer holds both data that has not yet been transmitted and data that has been sent, but has not been acknowledged. sb_cc is the number of bytes of data that reside in the send buffer, and $0 \leq$ sb_cc $\leq$ sb_hiwat.

sb_cc may temporarily exceed sb_hiwat when out-of-band data is sent.

It is sosend's responsibility to ensure that there is enough space in the send buffer before passing any data to the protocol layer through the pr_usrreq function. The protocol layer adds the data to the send buffer. sosend transfers data to the protocol in one of two ways:

- If PR_ATOMIC is set, sosend must preserve the message boundaries between the process and the protocol layer. In this case, sosend waits for enough space to become available to hold the entire message. When the space is available, an mbuf chain containing the entire message is constructed and passed to the protocol in a single call through the pr_usrreq function. RDP and SPP are examples of this type of protocol.

- If PR_ATOMIC is not set, sosend passes the message to the protocol one mbuf at a time and may pass a partial mbuf to avoid exceeding the high-water mark. This method is used with SOCK_STREAM protocols such as TCP and SOCK_SEQPACKET protocols such as TP4. With TP4, record boundaries are indicated explicitly with the MSG_EOR flag (Figure 16.12), so it is not necessary for the message boundaries to be preserved by sosend.

TCP applications have no control over the size of outgoing TCP segments. For example, a message of 4096 bytes sent on a TCP socket will be split by the socket layer into two mbufs with external clusters, containing 2048 bytes each, assuming there is enough space in the send buffer for 4096 bytes. Later, during protocol processing, TCP will segment the data according to the maximum segment size for the connection, which is normally less than 2048.

When a message is too large to fit in the available buffer space and the protocol allows messages to be split, sosend still does not pass data to the protocol until the free space in the buffer rises above sb_lowat. For TCP, sb_lowat defaults to 2048 (Figure 16.4), so this rule prevents the socket layer from bothering TCP with small chunks of data when the send buffer is nearly full.

Unreliable Protocol Buffering

With unreliable protocols (e.g., UDP), no data is ever stored in the send buffer and no acknowledgment is ever expected. Each message is passed immediately to the protocol where it is queued for transmission on the appropriate network device. In this case, sb_cc is always 0, and sb_hiwat specifies the maximum size of each write and indirectly the maximum size of a datagram.

Figure 16.4 shows that sb_hiwat defaults to 9216 (9×1024) for UDP. Unless the process changes sb_hiwat with the SO_SNDBUF socket option, an attempt to write a datagram larger than 9216 bytes returns with an error. Even then, other limitations of the protocol implementation may prevent a process from sending large datagrams. Section 11.10 of Volume 1 discusses these defaults and limits in other TCP/IP implementations.

> 9216 is large enough for a NFS write, which often defaults to 8192 bytes of data plus protocol headers.

sosend Code

Figure 16.22 shows an overview of the sosend function. We discuss the four shaded sections separately.

271–278   The arguments to sosend are: so, a pointer to the relevant socket; addr, a pointer to a destination address; uio, a pointer to a uio structure describing the I/O buffers in user space; top, an mbuf chain that holds data to be sent; control, an mbuf that holds control information to be sent; and flags, which contains options for this write call.

Normally, a process provides data to the socket layer through the uio mechanism and top is null. When the kernel itself is using the socket layer (such as with NFS), the data is passed to sosend as an mbuf chain pointed to by top, and uio is null.

279–304   The initialization code is described separately.

Lock send buffer

305–308   sosend's main processing loop starts at restart, where it obtains a lock on the send buffer with sblock before proceeding. The lock ensures orderly access to the socket buffer by multiple processes.

If MSG_DONTWAIT is set in flags, then SBLOCKWAIT returns M_NOWAIT, which tells sblock to return EWOULDBLOCK if the lock is not available immediately.

> MSG_DONTWAIT is used only by NFS in Net/3.

The main loop continues until sosend transfers all the data to the protocol (i.e., resid == 0).

Check for space

309–341   Before any data is passed to the protocol, various error conditions are checked and sosend implements the flow control and resource control algorithms described earlier. If sosend blocks waiting for more space to appear in the output buffer, it jumps back to restart before continuing.

Use data from top

342–350   Once space becomes available and sosend has obtained a lock on the send buffer, the data is prepared for delivery to the protocol layer. If uio is null (i.e., the data is in the mbuf chain pointed to by top), sosend checks MSG_EOR and sets M_EOR in the chain to mark the end of a logical record. The mbuf chain is ready for the protocol layer.

—— uipc_socket.c

```
271 sosend(so, addr, uio, top, control, flags)
272 struct socket *so;
273 struct mbuf *addr;
274 struct uio *uio;
275 struct mbuf *top;
276 struct mbuf *control;
277 int      flags;
278 {

                    /* initialization (Figure 16.23) */

305  restart:
306    if (error = sblock(&so->so_snd, SBLOCKWAIT(flags)))
307        goto out;
308    do {                           /* main loop, until resid == 0 */

                 /* wait for space in send buffer (Figure 16.24) */

342        do {
343            if (uio == NULL) {
344                /*
345                 * Data is prepackaged in "top".
346                 */
347                resid = 0;
348                if (flags & MSG_EOR)
349                    top->m_flags |= M_EOR;
350            } else
351                do {

                 /* fill a single mbuf or an mbuf chain (Figure 16.25) */

396                } while (space > 0 && atomic);

                 /* pass mbuf chain to protocol (Figure 16.26) */

412        } while (resid && space > 0);
413    } while (resid);

414  release:
415    sbunlock(&so->so_snd);
416  out:
417    if (top)
418        m_freem(top);
419    if (control)
420        m_freem(control);
421    return (error);
422 }
```

—— uipc_socket.c

Figure 16.22   sosend function: overview.

Copy data from process

351–396   When uio is not null, sosend must transfer the data from the process. When PR_ATOMIC is set (e.g., UDP), this loop continues until all the data has been stored in a single mbuf chain. A break, which is not shown in Figure 16.22, causes the loop to terminate when all the data has been copied from the process, and sosend passes the entire chain to the protocol.

When PR_ATOMIC is not set (e.g., TCP), this loop is executed only once, filling a single mbuf with data from uio. In this case, the mbufs are passed one at a time to the protocol.

Pass data to the protocol

397–413   For PR_ATOMIC protocols, after the mbuf chain is passed to the protocol, resid is always 0 and control falls through the two loops to release. When PR_ATOMIC is not set, sosend continues filling individuals mbufs while there is more data to send and while there is still space in the buffer. If the buffer fills and there is still data to send, sosend loops back and waits for more space before filling the next mbuf. If all the data is sent, both loops terminate.

Cleanup

414–422   After all the data has been passed to the protocol, the socket buffer is unlocked, any remaining mbufs are discarded, and sosend returns.

The detailed description of sosend is shown in four parts:

- initialization (Figure 16.23),
- error and resource checking (Figure 16.24),
- data transfer (Figure 16.25), and
- protocol dispatch (Figure 16.26).

The first part of sosend shown in Figure 16.23 initializes various variables.

Compute transfer size and semantics

279–284   atomic is set if sosendallatonce is true (any protocol for which PR_ATOMIC is set) or the data has been passed to sosend as an mbuf chain in top. This flag controls whether data is passed to the protocol as a single mbuf chain or in separate mbufs.

285–297   resid is the number of bytes in the iovec buffers or the number of bytes in the top mbuf chain. Exercise 16.1 discusses why resid might be negative.

If requested, disable routing

298–303   dontroute is set when the routing tables should be bypassed for *this* message only. clen is the number of bytes in the optional control mbuf.

304   The macro snderr posts the error code, reenables protocol processing, and jumps to the cleanup code at out. This macro simplifies the error handling within the function.

Figure 16.24 shows the part of sosend that checks for error conditions and waits for space to appear in the send buffer.

—————————————————————————————— *uipc_socket.c*
```
279      struct proc *p = curproc;    /* XXX */
280      struct mbuf **mp;
281      struct mbuf *m;
282      long    space, len, resid;
283      int     clen = 0, error, s, dontroute, mlen;
284      int     atomic = sosendallatonce(so) || top;

285      if (uio)
286          resid = uio->uio_resid;
287      else
288          resid = top->m_pkthdr.len;
289      /*
290       * In theory resid should be unsigned.
291       * However, space must be signed, as it might be less than 0
292       * if we over-committed, and we must use a signed comparison
293       * of space and resid.  On the other hand, a negative resid
294       * causes us to loop sending 0-length segments to the protocol.
295       */
296      if (resid < 0)
297          return (EINVAL);
298      dontroute =
299          (flags & MSG_DONTROUTE) && (so->so_options & SO_DONTROUTE) == 0 &&
300          (so->so_proto->pr_flags & PR_ATOMIC);
301      p->p_stats->p_ru.ru_msgsnd++;
302      if (control)
303          clen = control->m_len;
304 #define snderr(errno)    { error = errno; splx(s); goto release; }
```
—————————————————————————————— *uipc_socket.c*

Figure 16.23 sosend function: initialization.

309   Protocol processing is suspended to prevent the buffer from changing while it is being examined. Before each transfer, sosend checks several conditions:

310–311   • If output from the socket is prohibited (e.g., the write-half of a TCP connection has been closed), EPIPE is returned.

312–313   • If the socket is in an error state (e.g., an ICMP port unreachable may have been generated by a previous datagram), so_error is returned. sendit discards the error if some data has been sent before the error occurs (Figure 16.21, line 389).

314–318   • If the protocol requires connections and a connection has not been established or a connection attempt has not been started, ENOTCONN is returned. sosend permits a write consisting of control information and no data even when a connection has not been established.

> The Internet protocols do not use this feature, but it is used by TP4 to send data with a connection request, to confirm a connection request, and to send data with a disconnect request.

319–321   • If a destination address is not specified for a connectionless protocol (e.g., the process calls send without establishing a destination with connect), EDESTADDREQ is returned.

uipc_socket.c

```
309        s = splnet();
310        if (so->so_state & SS_CANTSENDMORE)
311            snderr(EPIPE);
312        if (so->so_error)
313            snderr(so->so_error);
314        if ((so->so_state & SS_ISCONNECTED) == 0) {
315            if (so->so_proto->pr_flags & PR_CONNREQUIRED) {
316                if ((so->so_state & SS_ISCONFIRMING) == 0 &&
317                    !(resid == 0 && clen != 0))
318                    snderr(ENOTCONN);
319            } else if (addr == 0)
320                snderr(EDESTADDRREQ);
321        }
322        space = sbspace(&so->so_snd);
323        if (flags & MSG_OOB)
324            space += 1024;
325        if (atomic && resid > so->so_snd.sb_hiwat ||
326            clen > so->so_snd.sb_hiwat)
327            snderr(EMSGSIZE);
328        if (space < resid + clen && uio &&
329            (atomic || space < so->so_snd.sb_lowat || space < clen)) {
330            if (so->so_state & SS_NBIO)
331                snderr(EWOULDBLOCK);
332            sbunlock(&so->so_snd);
333            error = sbwait(&so->so_snd);
334            splx(s);
335            if (error)
336                goto out;
337            goto restart;
338        }
339        splx(s);
340        mp = &top;
341        space -= clen;
```

uipc_socket.c

Figure 16.24   sosend function: error and resource checking.

Compute available space

322–324   sbspace computes the amount of free space remaining in the send buffer. This is an administrative limit based on the buffer's high-water mark, but is also limited by sb_mbmax to prevent many small messages from consuming too many mbufs (Figure 16.6). sosend gives out-of-band data some priority by relaxing the limits on the buffer size by 1024 bytes.

Enforce message size limit

325–327   If atomic is set and the message is larger than the high-water mark, EMSGSIZE is returned; the message is too large to be accepted by the protocol—even if the buffer were empty. If the control information is larger than the high-water mark, EMSGSIZE is also returned. This is the test that limits the size of a datagram or record.

Wait for more space?

328–329   If there is not enough space in the send buffer, the data is from a process (versus from the kernel in top), and one of the following conditions is true, then sosend must wait for additional space before continuing:

- the message must be passed to protocol in a single request (atomic is set), or
- the message may be split, but the free space has dropped below the low-water mark, or
- the message may be split, but the control information does not fit in the available space.

When the data is passed to sosend in top (i.e., when uio is null), the data is already located in mbufs. Therefore sosend ignores the high- and low-water marks since no additional mbuf allocations are required to pass the data to the protocol.

If the send buffer low-water mark is not used in this test, an interesting interaction occurs between the socket layer and the transport layer that leads to performance degradation. [Crowcroft et al. 1992] provides details on this scenario.

Wait for space

330–338   If sosend must wait for space and the socket is nonblocking, EWOULDBLOCK is returned. Otherwise, the buffer lock is released and sosend waits with sbwait until the status of the buffer changes. When sbwait returns, sosend reenables protocol processing and jumps back to restart to obtain a lock on the buffer and to check the error and space conditions again before continuing.

By default, sbwait blocks until data can be sent. By changing sb_timeo in the buffer through the SO_SNDTIMEO socket option, the process selects an upper bound for the wait time. If the timer expires, sbwait returns EWOULDBLOCK. Recall from Figure 16.21 that this error is discarded by sendit if some data has already been transferred to the protocol. This timer does not limit the length of the entire call, just the inactivity time between filling mbufs.

339–341   At this point, sosend has determined that some data may be passed to the protocol. splx enables interrupts since they should not be blocked during the relatively long time it takes to copy data from the process to the kernel. mp holds a pointer used to construct the mbuf chain. The size of the control information (clen) is subtracted from the space available before sosend transfers any data from the process.

Figure 16.25 shows the section of sosend that moves data from the process to one or more mbufs in the kernel.

Allocate packet header or standard mbuf

351–360   When atomic is set, this code allocates a packet header during the first iteration of the loop and standard mbufs afterwards. When atomic is not set, this code always allocates a packet header since top is always cleared before entering the loop.

─── *uipc_socket.c*

```
351                do {
352                    if (top == 0) {
353                        MGETHDR(m, M_WAIT, MT_DATA);
354                        mlen = MHLEN;
355                        m->m_pkthdr.len = 0;
356                        m->m_pkthdr.rcvif = (struct ifnet *) 0;
357                    } else {
358                        MGET(m, M_WAIT, MT_DATA);
359                        mlen = MLEN;
360                    }

361                    if (resid >= MINCLSIZE && space >= MCLBYTES) {
362                        MCLGET(m, M_WAIT);
363                        if ((m->m_flags & M_EXT) == 0)
364                            goto nopages;
365                        mlen = MCLBYTES;
366                        if (atomic && top == 0) {
367                            len = min(MCLBYTES - max_hdr, resid);
368                            m->m_data += max_hdr;
369                        } else
370                          . len = min(MCLBYTES, resid);
371                        space -= MCLBYTES;
372                    } else {
373                      nopages:
374                        len = min(min(mlen, resid), space);
375                        space -= len;
376                        /*
377                         * For datagram protocols, leave room
378                         * for protocol headers in first mbuf.
379                         */
380                        if (atomic && top == 0 && len < mlen)
381                            MH_ALIGN(m, len);
382                    }

383                    error = uiomove(mtod(m, caddr_t), (int) len, uio);
384                    resid = uio->uio_resid;
385                    m->m_len = len;
386                    *mp = m;
387                    top->m_pkthdr.len += len;
388                    if (error)
389                        goto release;
390                    mp = &m->m_next;
391                    if (resid <= 0) {
392                        if (flags & MSG_EOR)
393                            top->m_flags |= M_EOR;
394                        break;
395                    }
396                } while (space > 0 && atomic);
```
─── *uipc_socket.c*

Figure 16.25   sosend function: data transfer.

If possible, use a cluster

361–371   If the message is large enough to make a cluster allocation worthwhile and `space` is greater than or equal to `MCLBYTES`, a cluster is attached to the mbuf by `MCLGET`. When `space` is less than `MCLBYTES`, the extra 2048 bytes will break the allocation limit for the buffer since the entire cluster is allocated even if `resid` is less than `MCLBYTES`.

If `MCLGET` fails, `sosend` jumps to `nopages` and uses a standard mbuf instead of an external cluster.

> The test against `MINCLSIZE` should use >, not >=, since a write of 208 (`MINCLSIZE`) bytes fits within two mbufs.

When `atomic` is set (e.g., UDP), the mbuf chain represents a datagram or record and `max_hdr` bytes are reserved at the front of the *first* cluster for protocol headers. Subsequent clusters are part of the same chain and do not need room for the headers.

If `atomic` is not set (e.g., TCP), no space is reserved since `sosend` does not know how the protocol will segment the outgoing data.

Notice that `space` is decremented by the size of the cluster (2048 bytes) and not by `len`, which is the number of data bytes to be placed in the cluster (Exercise 16.2).

Prepare the mbuf

372–382   If a cluster was not used, the number of bytes stored in the mbuf is limited by the smaller of: (1) the space in the mbuf, (2) the number of bytes in the message, or (3) the space in the buffer.

When `atomic` is set, `MH_ALIGN` locates the data at the end of the buffer for the first buffer in the chain. `MH_ALIGN` is skipped if the data completely fills the mbuf. This may or may not leave enough room for protocol headers, depending on how much data is placed in the mbuf. When `atomic` is not set, no space is set aside for the headers.

Get data from the process

383–395   `uiomove` copies `len` bytes of data from the process to the mbuf. After the transfer, the mbuf length is updated, the previous mbuf is linked to the new mbuf (or `top` points to the first mbuf), and the length of the mbuf chain is updated. If an error occurred during the transfer, `sosend` jumps to `release`.

When the last byte is transferred from the process, `M_EOR` is set in the packet if the process set `MSG_EOR`, and `sosend` breaks out of this loop.

> `MSG_EOR` applies only to protocols with explicit record boundaries such as TP4, from the OSI protocol suite. TCP does not support logical records and ignores the `MSG_EOR` flag.

Fill another buffer?

396   If `atomic` is set, `sosend` loops back and begins filling another mbuf.

> The test for `space > 0` appears to be extraneous. `space` is irrelevant when `atomic` is not set since the mbufs are passed to the protocol one at a time. When `atomic` is set, this loop is entered only when there is enough space for the entire message. See also Exercise 16.2.

The last section of `sosend`, shown in Figure 16.26, passes the data and control mbufs to the protocol associated with the socket.

uipc_socket.c

```
397              if (dontroute)
398                  so->so_options |= SO_DONTROUTE;
399              s = splnet();        /* XXX */
400              error = (*so->so_proto->pr_usrreq) (so,
401                                (flags & MSG_OOB) ? PRU_SENDOOB : PRU_SEND,
402                                                   top, addr, control);
403              splx(s);
404              if (dontroute)
405                  so->so_options &= ~SO_DONTROUTE;
406              clen = 0;
407              control = 0;
408              top = 0;
409              mp = &top;
410              if (error)
411                  goto release;
412          } while (resid && space > 0);
413      } while (resid);
```
uipc_socket.c

Figure 16.26   sosend function: protocol dispatch.

397–405   The socket's SO_DONTROUTE option is toggled if necessary before and after passing the data to the protocol layer to bypass the routing tables on this message. This is the only option that can be enabled for a single message and, as described with Figure 16.23, it is controlled by the MSG_DONTROUTE flag during a write.

pr_usrreq is bracketed with splnet and splx to block interrupts while the protocol is processing the message. This is a paranoid assumption since some protocols (such as UDP) may be able to do output processing without blocking interrupts, but this information is not available at the socket layer.

If the process tagged this message as out-of-band data, sosend issues the PRU_SENDOOB request; otherwise it issues the PRU_SEND request. Address and control mbufs are also passed to the protocol at this time.

406–413   clen, control, top, and mp are reset, since control information is passed to the protocol only once and a new mbuf chain is constructed for the next part of the message. resid is nonzero only when atomic is not set (e.g., TCP). In that case, if space remains in the buffer, sosend loops back to fill another mbuf. If there is no more space, sosend loops back to wait for more space (Figure 16.24).

We'll see in Chapter 23 that unreliable protocols, such as UDP, immediately queue the data for transmission on the network. Chapter 26 describes how reliable protocols, such as TCP, add the data to the socket's send buffer where it remains until it is sent to, and acknowledged by, the destination.

sosend Summary

sosend is a complex function. It is 142 lines long, contains three nested loops, one loop implemented with goto, two code paths based on whether PR_ATOMIC is set or not, and two concurrency locks. As with much software, some of the complexity has accumulated over the years. NFS added the MSG_DONTWAIT semantics and the possibility

of receiving data from an mbuf chain instead of the buffers in a process. The SS_ISCONFIRMING state and MSG_EOR flag were introduced to handle the connection and record semantics of the OSI protocols.

A cleaner approach would be to implement a separate sosend function for each type of protocol and dispatch through a pr_send pointer in the protosw entry. This idea is suggested and implemented for UDP in [Partridge and Pink 1993].

Performance Considerations

As described in Figure 16.25, sosend, when possible, passes message in mbuf-sized chunks to the protocol layer. While this results in more calls to the protocol than building and passing an entire mbuf chain, [Jacobson 1988a] reports that it improves performance by increasing parallelism.

Transferring one mbuf at a time (up to 2048 bytes) allows the CPU to prepare a packet while the network hardware is transmitting. Contrast this to sending a large mbuf chain: while the chain is being constructed, the network and the receiving system are idle. On the system described in [Jacobson 1988a], this change resulted in a 20% increase in network throughput.

It is important to make sure the send buffer is always larger than the bandwidth-delay product of a connection (Section 20.7 of Volume 1). For example, if TCP discovers that the connection can hold 20 segments before an acknowledgment is received, the send buffer must be large enough to hold the 20 unacknowledged segments. If it is too small, TCP will run out of data to send before the first acknowledgment is returned and the connection will be idle for some period of time.

16.8 read, readv, recvfrom, and recvmsg System Calls

These four system calls, which we refer to collectively as *read system calls*, receive data from a network connection. The first three system calls are simpler interfaces to the most general read system call, recvmsg. Figure 16.27 summarizes the features of the four read system calls and one library function (recv).

Function	Type of descriptor	Number of buffers	Return sender's address?	Flags?	Return control information?
read	any	1			
readv	any	[1..UIO_MAXIOV]			
recv	sockets only	1		•	
recvfrom	sockets only	1	•	•	
recvmsg	sockets only	[1..UIO_MAXIOV]	•	•	•

Figure 16.27 Read system calls.

In Net/3, recv is implemented as a library function that calls recvfrom. For binary compatibility with previously compiled programs, the kernel maps the old recv system call to the function orecv. We discuss only the kernel implementation of recvfrom.

The `read` and `readv` system calls are valid with any descriptor, but the remaining calls are valid only with socket descriptors.

As with the write calls, multiple buffers are specified by an array of `iovec` structures. For datagram protocols, `recvfrom` and `recvmsg` return the source address associated with each incoming datagram. For connection-oriented protocols, `getpeername` returns the address associated with the other end of the connection. The flags associated with the receive calls are shown in Section 16.11.

As with the write calls, the receive calls utilize a common function, in this case `soreceive`, to do all the work. Figure 16.28 illustrates the flow of control for the read system calls.
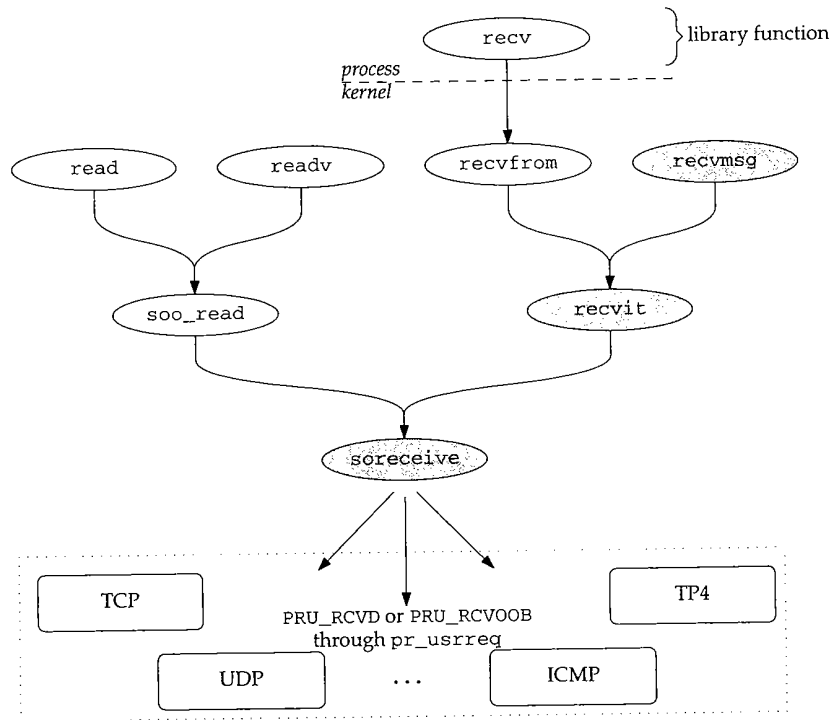

Figure 16.28   All socket input is processed by `soreceive`.

We discuss only the three shaded functions in Figure 16.28. The remaining functions are left for readers to investigate on their own.

16.9 `recvmsg` System Call

The `recvmsg` function is the most general read system call. Addresses, control information, and receive flags may be discarded without notification if a process uses one of the other read system calls while this information is pending. Figure 16.29 shows the `recvmsg` function.

```
                                                                  —— uipc_syscalls.c
433 struct recvmsg_args {
434     int     s;
435     struct msghdr *msg;
436     int     flags;
437 };

438 recvmsg(p, uap, retval)
439 struct proc *p;
440 struct recvmsg_args *uap;
441 int     *retval;
442 {
443     struct msghdr msg;
444     struct iovec aiov[UIO_SMALLIOV], *uiov, *iov;
445     int     error;

446     if (error = copyin((caddr_t) uap->msg, (caddr_t) & msg, sizeof(msg)))
447         return (error);
448     if ((u_int) msg.msg_iovlen >= UIO_SMALLIOV) {
449         if ((u_int) msg.msg_iovlen >= UIO_MAXIOV)
450             return (EMSGSIZE);
451         MALLOC(iov, struct iovec *,
452                 sizeof(struct iovec) * (u_int) msg.msg_iovlen, M_IOV,
453                 M_WAITOK);
454     } else
455         iov = aiov;
456     msg.msg_flags = uap->flags;
457     uiov = msg.msg_iov;
458     msg.msg_iov = iov;
459     if (error = copyin((caddr_t) uiov, (caddr_t) iov,
460                         (unsigned) (msg.msg_iovlen * sizeof(struct iovec))))
461             goto done;
462     if ((error = recvit(p, uap->s, &msg, (caddr_t) 0, retval)) == 0) {
463         msg.msg_iov = uiov;
464         error = copyout((caddr_t) & msg, (caddr_t) uap->msg, sizeof(msg));
465     }
466 done:
467     if (iov != aiov)
468         FREE(iov, M_IOV);
469     return (error);
470 }
                                                                  —— uipc_syscalls.c
```

Figure 16.29   recvmsg system call.

433–445   The three arguments to recvmsg are: the socket descriptor; a pointer to a msghdr structure; and several control flags.

Copy iov array

446–461   As with sendmsg, recvmsg copies the msghdr structure into the kernel, allocates a larger iovec array if the automatic array aiov is too small, and copies the array entries from the process into the kernel array pointed to by iov (Section 16.4). The flags provided as the third argument are copied into the msghdr structure.

recvit and cleanup

462-470   After recvit has received data, the msghdr structure is copied back into the process with the updated buffer lengths and flags. If a larger iovec structure was allocated, it is released before recvmsg returns.

16.10 recvit Function

The recvit function shown in Figures 16.30 and 16.31 is called from recv, recvfrom, and recvmsg. It prepares a uio structure for processing by soreceive based on the msghdr structure prepared by the recvxxx calls.

uipc_syscalls.c

```
471 recvit(p, s, mp, namelenp, retsize)
472 struct proc *p;
473 int     s;
474 struct msghdr *mp;
475 caddr_t namelenp;
476 int     *retsize;
477 {
478     struct file *fp;
479     struct uio auio;
480     struct iovec *iov;
481     int     i;
482     int     len, error;
483     struct mbuf *from = 0, *control = 0;

484     if (error = getsock(p->p_fd, s, &fp))
485         return (error);
486     auio.uio_iov = mp->msg_iov;
487     auio.uio_iovcnt = mp->msg_iovlen;
488     auio.uio_segflg = UIO_USERSPACE;
489     auio.uio_rw = UIO_READ;
490     auio.uio_procp = p;
491     auio.uio_offset = 0;          /* XXX */
492     auio.uio_resid = 0;
493     iov = mp->msg_iov;
494     for (i = 0; i < mp->msg_iovlen; i++, iov++) {
495         if (iov->iov_len < 0)
496             return (EINVAL);
497         if ((auio.uio_resid += iov->iov_len) < 0)
498             return (EINVAL);
499     }
500     len = auio.uio_resid;
```

uipc_syscalls.c

Figure 16.30   recvit function: initialize uio structure.

471-500   getsock returns the file structure for the descriptor s, and then recvit initializes the uio structure to describe a read transfer from the kernel to the process. The number of bytes to transfer is computed by summing the msg_iovlen members of the iovec array. The total is saved in uio_resid and in len.

The second half of recvit, shown in Figure 16.31, calls soreceive and copies the results back to the process.

——————————————————————————————— uipc_syscalls.c

```
501    if (error = soreceive((struct socket *) fp->f_data, &from, &auio,
502      (struct mbuf **) 0, mp->msg_control ? &control : (struct mbuf **) 0,
503                        &mp->msg_flags)) {
504        if (auio.uio_resid != len && (error == ERESTART ||
505                                error == EINTR || error == EWOULDBLOCK))
506            error = 0;
507    }
508    if (error)
509        goto out;
510    *retsize = len - auio.uio_resid;
511    if (mp->msg_name) {
512        len = mp->msg_namelen;
513        if (len <= 0 || from == 0)
514            len = 0;
515        else {
516            if (len > from->m_len)
517                len = from->m_len;
518            /* else if len < from->m_len ??? */
519            if (error = copyout(mtod(from, caddr_t),
520                            (caddr_t) mp->msg_name, (unsigned) len))
521                goto out;
522        }
523        mp->msg_namelen = len;
524        if (namelenp &&
525            (error = copyout((caddr_t) & len, namelenp, sizeof(int)))) {
526            goto out;
527        }
528    }
529    if (mp->msg_control) {
530        len = mp->msg_controllen;
531        if (len <= 0 || control == 0)
532            len = 0;
533        else {
534            if (len >= control->m_len)
535                len = control->m_len;
536            else
537                mp->msg_flags |= MSG_CTRUNC;
538            error = copyout((caddr_t) mtod(control, caddr_t),
539                            (caddr_t) mp->msg_control, (unsigned) len);
540        }
541        mp->msg_controllen = len;
542    }
543  out:
544    if (from)
545        m_freem(from);
546    if (control)
547        m_freem(control);
548    return (error);
549 }
```

——————————————————————————————— uipc_syscalls.c

Figure 16.31 recvit function: return results.

Section

501–510

511–542

543–549

16.11

Out-of-E

Call soreceive

501-510   soreceive implements the complex semantics of receiving data from the socket buffers. The number of bytes transferred is saved in *retsize and returned to the process. When an signal arrives or a blocking condition occurs after some data has been copied to the process (len is not equal to uio_resid), the error is discarded and the partial transfer is reported.

Copy address and control information to the process

511-542   If the process provided a buffer for an address or control information or both, the buffers are filled and their lengths adjusted according to what soreceive returned. An address may be truncated if the buffer is too small. This can be detected by the process if it saves the buffer length before the read call and compares it with the value returned by the kernel in the namelenp variable (or in the length field of the sockaddr structure). Truncation of control information is reported by setting MSG_CTRUNC in msg_flags. See also Exercise 16.7.

Cleanup

543-549   At out, the mbufs allocated for the source address and the control information are released.

16.11 soreceive Function

This function transfers data from the receive buffer of the socket to the buffers specified by the process. Some protocols provide an address specifying the sender of the data, and this can be returned along with additional control information that may be present. Before examining the code, we need to discuss the semantics of a receive operation, out-of-band data, and the organization of a socket's receive buffer.

Figure 16.32 lists the flags that are recognized by the kernel during soreceive.

flags	Description	Reference
MSG_DONTWAIT	do not wait for resources during this call	Figure 16.38
MSG_OOB	receive out-of-band data instead of regular data	Figure 16.39
MSG_PEEK	receive a copy of the data without consuming it	Figure 16.43
MSG_WAITALL	wait for data to fill buffers before returning	Figure 16.50

Figure 16.32   recvxxx system calls: flag values passed to kernel.

recvmsg is the only read system call that returns flags to the process. In the other calls, the information is discarded by the kernel before control returns to the process. Figure 16.33 lists the flags that recvmsg can set in the msghdr structure.

Out-of-Band Data

Out-of-band (OOB) data semantics vary widely among protocols. In general, protocols expedite OOB data along a previously established communication link. The OOB data might not remain in sequence with previously sent regular data. The socket layer

msg_flags	Description	Reference
MSG_CTRUNC	the control information received was larger than the buffer provided	Figure 16.31
MSG_EOR	the data received marks the end of a logical record	Figure 16.48
MSG_OOB	the buffer(s) contains out-of-band data	Figure 16.45
MSG_TRUNC	the message received was larger than the buffer(s) provided	Figure 16.51

Figure 16.33   recvmsg system call: msg_flag values returned by kernel.

supports two mechanisms to facilitate handling OOB data in a protocol-independent way: tagging and synchronization. In this chapter we describe the abstract OOB mechanisms implemented by the socket layer. UDP does not support OOB data. The relationship between TCP's urgent data mechanism and the socket OOB mechanism is described in the TCP chapters.

A sending process tags data as OOB data by setting the MSG_OOB flag in any of the sendxxx calls. sosend passes this information to the socket's protocol, which provides any special services, such as expediting the data or using an alternate queueing strategy.

When a protocol receives OOB data, the data is set aside instead of placing it in the socket's receive buffer. A process receives the pending OOB data by setting the MSG_OOB flag in one of the recvxxx calls. Alternatively, the receiving process can ask the protocol to place OOB data inline with the regular data by setting the SO_OOBINLINE socket option (Section 17.3). When SO_OOBINLINE is set, the protocol places incoming OOB data in the receive buffer with the regular data. In this case, MSG_OOB is not used to receive the OOB data. Read calls return either all regular data or all OOB data. The two types are never mixed in the input buffers of a single input system call. A process that uses recvmsg to receive data can examine the MSG_OOB flag to determine if the returned data is regular data or OOB data that has been placed inline.

The socket layer supports synchronization of OOB and regular data by allowing the protocol layer to mark the point in the regular data stream at which OOB data was received. The receiver can determine when it has reached this mark by using the SIOCATMARK ioctl command after each read system call. When receiving regular data, the socket layer ensures that only the bytes preceding the mark are returned in a single message so that the receiver does not inadvertently pass the mark. If additional OOB data is received before the receiver reaches the mark, the mark is silently advanced.

Example

Figure 16.34 illustrates the two methods of receiving out-of-band data. In both examples, bytes A through I have been received as regular data, byte J as out-of-band data, and bytes K and L as regular data. The receiving process has accepted all data up to but not including byte A.

In the first example, the process can read bytes A through I or, if MSG_OOB is set, byte J. Even if the length of the read request is more than 9 bytes (A–I), the socket layer
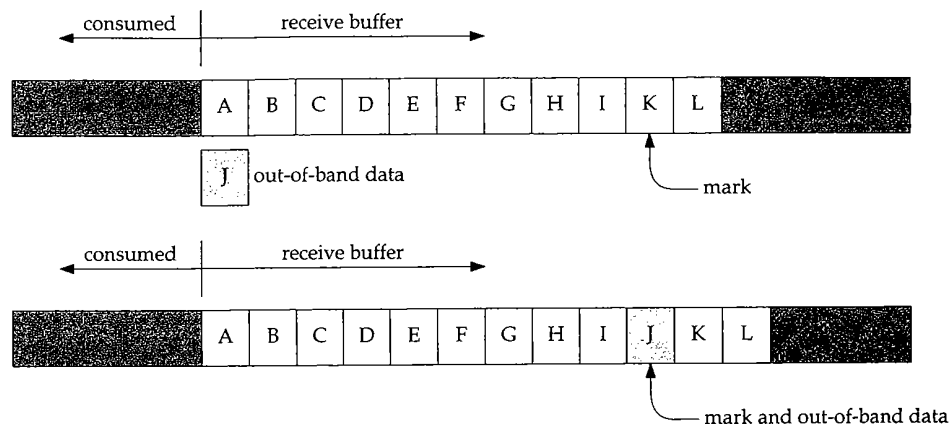
Figure 16.34   Receiving out-of-band data.

returns only 9 bytes to avoid passing the out-of-band synchronization mark. When byte I is consumed, SIOCATMARK is true; it is not necessary to consume byte J for the process to reach the out-of-band mark.

In the second example, the process can read only bytes A through I, at which point SIOCATMARK is true. A second call can read bytes J through L.

In Figure 16.34, byte J is *not* the byte identified by TCP's urgent pointer. The urgent pointer in this example would point to byte K. See Section 29.7 for details.

Other Receive Options

A process can set the MSG_PEEK flag to retrieve data without consuming it. The data remains on the receive queue until a read system call without MSG_PEEK is processed.

The MSG_WAITALL flag indicates that the call should not return until enough data can be returned to fulfill the entire request. Even if soreceive has some data that can be returned to the process, it waits until additional data has been received.

When MSG_WAITALL is set, soreceive can return without filling the buffer in the following cases:

- the read-half of the connection is closed,
- the socket's receive buffer is smaller than the size of the read,
- an error occurs while the process is waiting for additional data,
- out-of-band data becomes available, or
- the end of a logical record occurs before the read buffer is filled.

NFS is the only software in Net/3 that uses the MSG_WAITALL and MSG_DONTWAIT flags. MSG_DONTWAIT can be set by a process to issue a nonblocking read system call without selecting nonblocking I/O with ioctl or fcntl.

Receive Buffer Organization: Message Boundaries

For protocols that support message boundaries, each message is stored in a single chain of mbufs. Multiple messages in the receive buffer are linked together by m_nextpkt to form a queue of mbufs (Figure 2.21). The protocol processing layer adds data to the receive queue and the socket layer removes data from the receive queue. The high-water mark for a receive buffer restricts the amount of data that can be stored in the buffer.

When PR_ATOMIC is not set, the protocol layer stores as much data in the buffer as possible and discards the portion of the incoming data that does not fit. For TCP, this means that any data that arrives and is outside the receive window is discarded. When PR_ATOMIC is set, the entire message must fit within the buffer. If the message does not fit, the protocol layer discards the entire message. For UDP, this means that incoming datagrams are discarded when the receive buffer is full, probably because the process is not reading datagrams fast enough.

Protocols with PR_ADDR set use sbappendaddr to construct an mbuf chain and add it to the receive queue. The chain contains an mbuf with the source address of the message, 0 or more control mbufs, followed by 0 or more mbufs containing the data.

For SOCK_SEQPACKET and SOCK_RDM protocols, the protocol builds an mbuf chain for each record and calls sbappendrecord to append the record to the end of the receive buffer if PR_ATOMIC is set. If PR_ATOMIC is not set (OSI's TP4), a new record is started with sbappendrecord. Additional data is added to the record with sbappend.

> It is not correct to assume that PR_ATOMIC indicates the buffer organization. For example, TP4 does not have PR_ATOMIC set, but supports record boundaries with the M_EOR flag.

Figure 16.35 illustrates the organization of a UDP receive buffer consisting of 3 mbuf chains (i.e., three datagrams). The m_type value for each mbuf is included.

In the figure, the third datagram has some control information associated with it. Three UDP socket options can cause control information to be placed in the receive buffer. See Figure 22.5 and Section 23.7 for details.

For PR_ATOMIC protocols, sb_lowat is ignored while data is being received. When PR_ATOMIC is not set, sb_lowat is the smallest number of bytes returned in a read system call. There are some exceptions to this rule, discussed with Figure 16.41.

Receive Buffer Organization: No Message Boundaries

When the protocol does not maintain message boundaries (i.e., SOCK_STREAM protocols such as TCP), incoming data is appended to the end of the last mbuf chain in the buffer with sbappend. Incoming data is trimmed to fit within the receive buffer, and sb_lowat puts a lower bound on the number of bytes returned by a read system call.

Figure 16.36 illustrates the organization of a TCP receive buffer, which contains only regular data.

socket{}
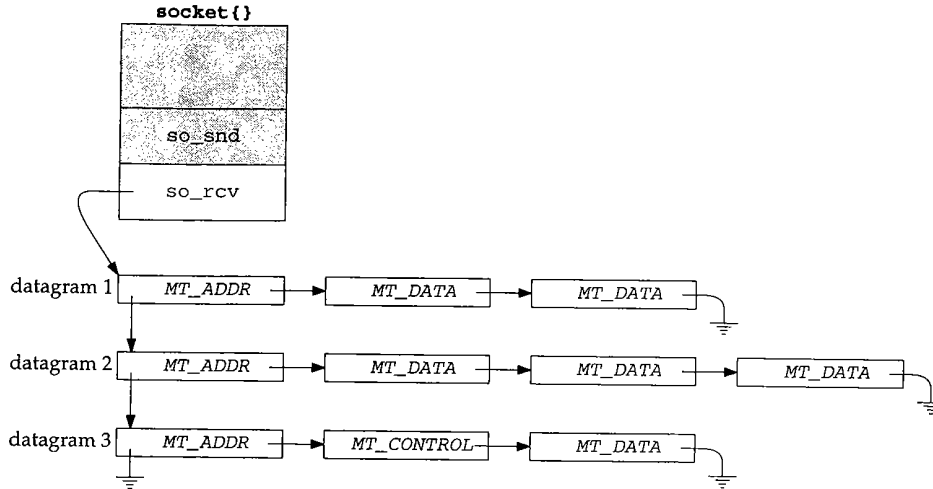
Figure 16.35   UDP receive buffer consisting of three datagrams.

socket{}
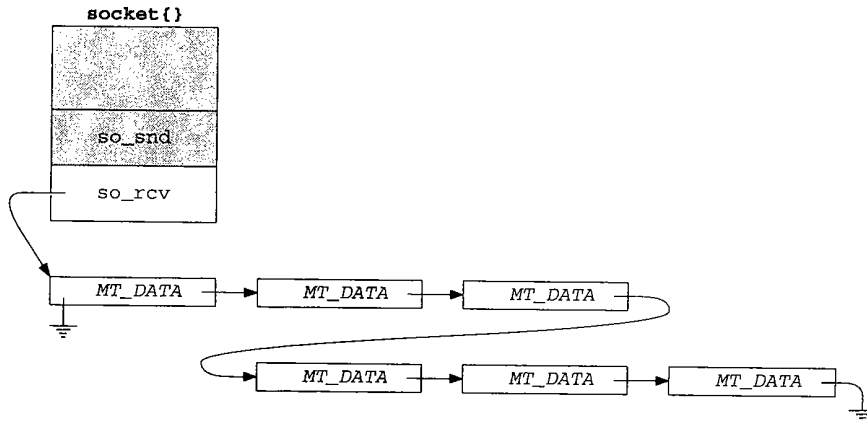
Figure 16.36   so_rcv buffer for TCP.

Control Information and Out-of-band Data

Unlike TCP, some stream protocols support control information and call
sbappendcontrol to append the control information and the associated data as a new
mbuf chain in the receive buffer. If the protocol supports inline OOB data,
sbinsertoob inserts a new mbuf chain just after any mbuf chain that contains OOB
data, but before any mbuf chain with regular data. This ensures that incoming OOB
data is queued ahead of any regular data.

Figure 16.37 illustrates the organization of a receive buffer that contains control information and OOB data.

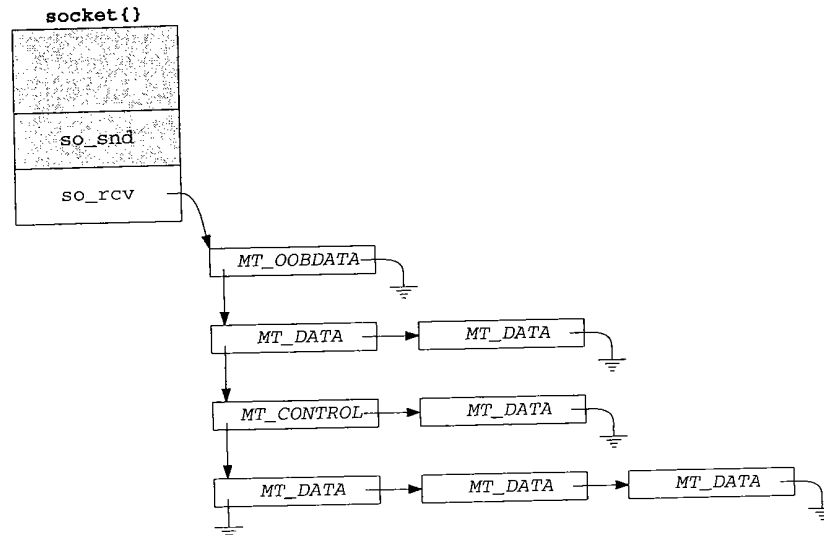**Figure 16.37**   so_rcv buffer with control and OOB data.

The Unix domain stream protocol supports control information and the OSI TP4 protocol supports MT_OOBDATA mbufs. TCP does not support control data nor does it support the MT_OOBDATA form of out-of-band data. If the byte identified by TCP's urgent pointer is stored inline (SO_OOBINLINE is set), it appears as regular data, not OOB data. TCP's handling of the urgent pointer and the associated byte is described in Section 29.7.

16.12 soreceive Code

We now have enough background information to discuss soreceive in detail. While receiving data, soreceive must respect message boundaries, handle addresses and control information, and handle any special semantics identified by the read flags (Figure 16.32). The general rule is that soreceive processes one record per call and tries to return the number of bytes requested. Figure 16.38 shows an overview of the function.

439–446   soreceive has six arguments. so is a pointer to the socket. A pointer to an mbuf to receive address information is returned in *paddr. If mp0 points to an mbuf pointer, soreceive transfers the receive buffer data to an mbuf chain pointed to by *mp0. In this case, the uio structure is used only for the count in uio_resid. If mp0 is null, soreceive copies the data into buffers described by the uio structure. A pointer to the mbuf containing control information is returned in *controlp, and soreceive returns the flags described in Figure 16.33 in *flagsp.

447–453   soreceive starts by setting pr to point to the socket's protocol switch structure and saving uio_resid (the size of the receive request) in orig_resid. If control information or addressing information is copied from the kernel to the process, orig_resid is set to 0. If data is copied, uio_resid is updated. In either case, orig_resid will not equal uio_resid. This fact is used at the end of soreceive (Figure 16.51).

454–461   *paddr and *controlp are cleared. The flags passed to soreceive in *flagsp are saved in flags after the MSG_EOR flag is cleared (Exercise 16.8). flagsp is a value–result argument, but only the recvmsg system call can receive the result flags. If flagsp is null, flags is set to 0.

483–487   Before accessing the receive buffer, sblock locks the buffer. soreceive waits for the lock unless MSG_DONTWAIT is set in flags.

> This is another side effect of supporting calls to the socket layer from NFS within the kernel.

Protocol processing is suspended, so soreceive is not interrupted while it examines the buffer. m is the first mbuf on the first chain in the receive buffer.

If necessary, wait for data

488–541   soreceive checks several conditions and if necessary waits for more data to arrive in the buffer before continuing. If soreceive sleeps in this code, it jumps back to restart when it wakes up to see if enough data has arrived. This continues until the request can be satisfied.

542–545   soreceive jumps to dontblock when it has enough data to satisfy the request. A pointer to the second chain in the receive buffer is saved in nextrecord.

Process address and control information

546–590   Address information and control information are processed before any other data is transferred from the receive buffer.

Setup data transfer

591–597   Since only OOB data or regular data is transferred in a single call to soreceive, this code remembers the type of data at the front of the queue so soreceive can stop the transfer when the type changes.

Mbuf data transfer loop

598–692   This loop continues as long as there are mbufs in the buffer (m is not null), the requested number of bytes has not been transferred (uio_resid > 0), and no error has occurred.

Cleanup

693–719   The remaining code updates various pointers, flags, and offsets; releases the socket buffer lock; enables protocol processing; and returns.

```
439 soreceive(so, paddr, uio, mp0, controlp, flagsp)
440 struct socket *so;
441 struct mbuf **paddr;
442 struct uio *uio;
443 struct mbuf **mp0;
444 struct mbuf **controlp;
445 int    *flagsp;
446 {
447     struct mbuf *m, **mp;
448     int     flags, len, error, s, offset;
449     struct protosw *pr = so->so_proto;
450     struct mbuf *nextrecord;
451     int     moff, type;
452     int     orig_resid = uio->uio_resid;

453     mp = mp0;
454     if (paddr)
455         *paddr = 0;
456     if (controlp)
457         *controlp = 0;
458     if (flagsp)
459         flags = *flagsp & ~MSG_EOR;
460     else
461         flags = 0;

                                    /* MSG_OOB processing and */
                                /* implicit connection confirmation */


483 restart:
484     if (error = sblock(&so->so_rcv, SBLOCKWAIT(flags)))
485         return (error);
486     s = splnet();
487     m = so->so_rcv.sb_mb;

                            /* if necessary, wait for data to arrive */


542 dontblock:
543     if (uio->uio_procp)
544         uio->uio_procp->p_stats->p_ru.ru_msgrcv++;
545     nextrecord = m->m_nextpkt;

                            /* process address and control information */


591     if (m) {
592         if ((flags & MSG_PEEK) == 0)
593             m->m_nextpkt = nextrecord;
594         type = m->m_type;
595         if (type == MT_OOBDATA)
596             flags |= MSG_OOB;
597     }
```

```
                          /* process data */

693       }                       /* while more data and more space to fill */


                          /* cleanup */

715    release:
716       sbunlock(&so->so_rcv);
717       splx(s);
718       return (error);
719 }
```
—— *uipc_socket.c*

Figure 16.38   soreceive function: overview.

In Figure 16.39, soreceive handles requests for OOB data.

—— *uipc_socket.c*
```
462    if (flags & MSG_OOB) {
463        m = m_get(M_WAIT, MT_DATA);
464        error = (*pr->pr_usrreq) (so, PRU_RCVOOB,
465                m, (struct mbuf *) (flags & MSG_PEEK), (struct mbuf *) 0);
466        if (error)
467            goto bad;
468        do {
469            error = uiomove(mtod(m, caddr_t),
470                             (int) min(uio->uio_resid, m->m_len), uio);
471            m = m_free(m);
472        } while (uio->uio_resid && error == 0 && m);
473        bad:
474        if (m)
475            m_freem(m);
476        return (error);
477    }
```
—— *uipc_socket.c*

Figure 16.39   soreceive function: out-of-band data.

Receive OOB data

462–477   Since OOB data is not stored in the receive buffer, soreceive allocates a standard mbuf and issues the PRU_RCVOOB request to the protocol. The while loop copies any data returned by the protocol to the buffers specified by uio. After the copy, soreceive returns 0 or the error code.

UDP always returns EOPNOTSUPP for the PRU_RCVOOB request. See Section 30.2 for details regarding TCP urgent processing. In Figure 16.40, soreceive handles connection confirmation.

```
                                                                            uipc_socket.c
478     if (mp)
479         *mp = (struct mbuf *) 0;
480     if (so->so_state & SS_ISCONFIRMING && uio->uio_resid)
481         (*pr->pr_usrreq) (so, PRU_RCVD, (struct mbuf *) 0,
482                           (struct mbuf *) 0, (struct mbuf *) 0);
                                                                            uipc_socket.c
```

Figure 16.40 soreceive function: connection confirmation.

Connection confirmation

478–482   If the data is to be returned in an mbuf chain, *mp is initialized to null. If the socket is in the SO_ISCONFIRMING state, the PRU_RCVD request notifies the protocol that the process is attempting to receive data.

> The SO_ISCONFIRMING state is used only by the OSI stream protocol, TP4. In TP4, a connection is not considered complete until a user-level process has confirmed the connection by attempting to send or receive data. The process can reject a connection by calling shutdown or close, perhaps after calling getpeername to determine where the connection came from.

Figure 16.38 showed that the receive buffer is locked before it is examined by the code in Figure 16.41. This part of soreceive determines if the read system call can be satisfied by the data that is already in the receive buffer.

```
                                                                            uipc_socket.c
488     /*
489      * If we have less data than requested, block awaiting more
490      * (subject ·to any timeout) if:
491      *   1. the current count is less than the low water mark, or
492      *   2. MSG_WAITALL is set, and it is possible to do the entire
493      *   receive operation at once if we block (resid <= hiwat).
494      *   3. MSG_DONTWAIT is not set
495      *
496      * If MSG_WAITALL is set but resid is larger than the receive buffer,
497      * we have to do the receive in sections, and thus risk returning
498      * a short count if a timeout or signal occurs after we start.
499      */
500     if (m == 0 || ((flags & MSG_DONTWAIT) == 0 &&
501                     so->so_rcv.sb_cc < uio->uio_resid) &&
502         (so->so_rcv.sb_cc < so->so_rcv.sb_lowat ||
503         ((flags & MSG_WAITALL) && uio->uio_resid <= so->so_rcv.sb_hiwat)) &&
504         m->m_nextpkt == 0 && (pr->pr_flags & PR_ATOMIC) == 0) {
                                                                            uipc_socket.c
```

Figure 16.41 soreceive function: enough data?

Can the call be satisfied now?

488–504   The general rule for soreceive is that it waits until enough data is in the receive buffer to satisfy the entire read. There are several conditions that cause an error or less data than was requested to be returned.

   If any of the following conditions are true, the process is put to sleep to wait for more data to arrive so the call can be satisfied:

_socket.c

_socket.c

e socket
that the

a connec-
iection by
;hutdown
ne from.

1 by the
ll can be

c_socket.c

ffer,
g

t)) &&

)c_socket.c

e receive
)r or less

wait for

- There is no data in the receive buffer (m equals 0).

- There is not enough data to satisfy the entire read (sb_cc < uio_resid and MSG_DONTWAIT is not set), the minimum amount of data is *not* available (sb_cc < sb_lowat), and more data can be appended to this chain when it arrives (m_nextpkt is 0 and PR_ATOMIC is *not* set).

- There is not enough data to satisfy the entire read, a minimum amount of data *is* available, data can be added to this chain, but MSG_WAITALL indicates that soreceive should wait until the entire read can be satisfied.

If the conditions in the last case are met but the read is too large to be satisfied without blocking (uio_resid > sb_hiwat), soreceive continues without waiting for more data.

If there is some data in the buffer and MSG_DONTWAIT is set, soreceive does not wait for more data.

There are several reasons why waiting for more data may not be appropriate. In Figure 16.42, soreceive checks for these conditions and returns, or waits for more data to arrive.

Wait for more data?

505–534   At this point, soreceive has determined that it must wait for additional data to arrive before the read can be satisfied. Before waiting it checks for several additional conditions:

505–512   - If the socket is in an error state and *empty* (m is null), soreceive returns the error code. If there is an error and the receive buffer also contains data (m is nonnull), the data is returned and a subsequent read returns the error when there is no more data. If MSG_PEEK is set, the error is not cleared, since a read system call with MSG_PEEK set should not change the state of the socket.

513–518   - If the read-half of the connection has been closed and data remains in the receive buffer, sosend does not wait and returns the data to the process (at dontblock). If the receive buffer is empty, soreceive jumps to release and the read system call returns 0, which indicates that the read-half of the connection is closed.

519–523   - If the receive buffer contains out-of-band data or the end of a logical record, soreceive does not wait for additional data and jumps to dontblock.

524–528   - If the protocol requires a connection and it does not exist, ENOTCONN is posted and the function jumps to release.

529–534   - If the read is for 0 bytes or nonblocking semantics have been selected, the function jumps to release and returns 0 or EWOULDBLOCK, respectively.

Yes, wait for more data

535–541   soreceive has now determined that it must wait for more data, and that it is reasonable to do so (i.e., some data will arrive). The receive buffer is unlocked while the process sleeps in sbwait. If sbwait returns because of an error or a signal,

—— *uipc_socket.c*
```
505            if (so->so_error) {
506                if (m)
507                    goto dontblock;
508                error = so->so_error;
509                if ((flags & MSG_PEEK) == 0)
510                    so->so_error = 0;
511                goto release;
512            }
513            if (so->so_state & SS_CANTRCVMORE) {
514                if (m)
515                    goto dontblock;
516                else
517                    goto release;
518            }
519            for (; m; m = m->m_next)
520                if (m->m_type == MT_OOBDATA || (m->m_flags & M_EOR)) {
521                    m = so->so_rcv.sb_mb;
522                    goto dontblock;
523                }
524            if ((so->so_state & (SS_ISCONNECTED | SS_ISCONNECTING)) == 0 &&
525                (so->so_proto->pr_flags & PR_CONNREQUIRED)) {
526                error = ENOTCONN;
527                goto release;
528            }
529            if (uio->uio_resid == 0)
530                goto release;
531            if ((so->so_state & SS_NBIO) || (flags & MSG_DONTWAIT)) {
532                error = EWOULDBLOCK;
533                goto release;
534            }
535            sbunlock(&so->so_rcv);
536            error = sbwait(&so->so_rcv);
537            splx(s);
538            if (error)
539                return (error);
540            goto restart;
541        }
```
—— *uipc_socket.c*

Figure 16.42   soreceive function: wait for more data?

soreceive returns the error; otherwise the function jumps to restart to determine if
the read can be satisfied now that more data has arrived.

As in sosend, a process can enable a receive timer for sbwait with the
SO_RCVTIMEO socket option. If the timer expires before any data arrives, sbwait
returns EWOULDBLOCK.

> The effect of this timer is not what one would expect. Since the timer gets reset every time
> there is activity on the socket buffer, the timer never expires if at least 1 byte arrives within the
> timeout interval. This can delay the return of the read system call for more than the value of
> the timer. sb_timeo is an inactivity timer and does not put an upper bound on the amount of
> time that may be required to satisfy the read system call.

At this point, `soreceive` is prepared to transfer some data from the receive buffer. Figure 16.43 shows the transfer of any address information.

```
                                                            ─── uipc_socket.c
542    dontblock:
543        if (uio->uio_procp)
544            uio->uio_procp->p_stats->p_ru.ru_msgrcv++;
545        nextrecord = m->m_nextpkt;
546        if (pr->pr_flags & PR_ADDR) {
547            orig_resid = 0;
548            if (flags & MSG_PEEK) {
549                if (paddr)
550                    *paddr = m_copy(m, 0, m->m_len);
551                m = m->m_next;
552            } else {
553                sbfree(&so->so_rcv, m);
554                if (paddr) {
555                    *paddr = m;
556                    so->so_rcv.sb_mb = m->m_next;
557                    m->m_next = 0;
558                    m = so->so_rcv.sb_mb;
559                } else {
560                    MFREE(m, so->so_rcv.sb_mb);
561                    m = so->so_rcv.sb_mb;
562                }
563            }
564        }
                                                            ─── uipc_socket.c
```

Figure 16.43   `soreceive` function: return address information.

dontblock

542–545   `nextrecord` maintains a reference to the next record that appears in the receive buffer. This is used at the end of `soreceive` to attach the remaining mbufs to the socket buffer after the first chain has been discarded.

Return address information

546–564   If the protocol provides addresses, such as UDP, the mbuf containing the address is removed from the mbuf chain and returned in `*paddr`. If `paddr` is null, the address is discarded.

Throughout `soreceive`, if `MSG_PEEK` is set, the data is not removed from the buffer.

The code in Figure 16.44 processes any control mbufs that are in the buffer.

Return control information

565–590   Each control mbuf is removed from the buffer (or copied if `MSG_PEEK` is set) and attached to `*controlp`. If `controlp` is null, the control information is discarded.

If the process is prepared to receive control information, the protocol has a `dom_externalize` function defined, and if the control mbuf contains a `SCM_RIGHTS` (access rights) message, the `dom_externalize` function is called. This function takes any kernel action associated with receiving the access rights. Only the Unix protocol

─── *uipc_socket.c*

```
565        while (m && m->m_type == MT_CONTROL && error == 0) {
566            if (flags & MSG_PEEK) {
567                if (controlp)
568                    *controlp = m_copy(m, 0, m->m_len);
569                m = m->m_next;
570            } else {
571                sbfree(&so->so_rcv, m);
572                if (controlp) {
573                    if (pr->pr_domain->dom_externalize &&
574                        mtod(m, struct cmsghdr *)->cmsg_type ==
575                        SCM_RIGHTS)
576                            error = (*pr->pr_domain->dom_externalize) (m);
577                    *controlp = m;
578                    so->so_rcv.sb_mb = m->m_next;
579                    m->m_next = 0;
580                    m = so->so_rcv.sb_mb;
581                } else {
582                    MFREE(m, so->so_rcv.sb_mb);
583                    m = so->so_rcv.sb_mb;
584                }
585            }
586            if (controlp) {
587                orig_resid = 0;
588                controlp = &(*controlp)->m_next;
589            }
590        }
```
─── *uipc_socket.c*

Figure 16.44   soreceive function: control information.

domain supports access rights, as discussed in Section 7.3. If the process is not prepared to receive control information (controlp is null) the mbuf is discarded.

The loop continues while there are more mbufs with control information and no error has occurred.

> For the Unix protocol domain, the dom_externalize function implements the semantics of passing file descriptors by modifying the file descriptor table of the receiving process.

After the control mbufs are processed, m points to the next mbuf on the chain. If the chain does not contain any mbufs after the address, or after the control information, m is null. This occurs, for example, when a 0-length UDP datagram is queued in the receive buffer. In Figure 16.45 soreceive prepares to transfer the data from the mbuf chain.

Prepare to transfer data

591–597   After the control mbufs have been processed, the chain should contain regular, out-of-band data mbufs or no mbufs at all. If m is null, soreceive is finished with this chain and control drops to the bottom of the while loop. If m is not null, any remaining chains (nextrecord) are reattached to m and the type of the next mbuf is saved in type. If the next mbuf contains OOB data, MSG_OOB is set in flags, which is later

```
591    if (m) {                                                        ── uipc_socket.c
592        if ((flags & MSG_PEEK) == 0)
593            m->m_nextpkt = nextrecord;
594        type = m->m_type;
595        if (type == MT_OOBDATA)
596            flags |= MSG_OOB;
597    }
```
── uipc_socket.c

Figure 16.45 soreceive function: mbuf transfer setup.

returned to the process. Since TCP does not support the MT_OOBDATA form of out-of-band data, MSG_OOB will never be returned for reads on TCP sockets.

Figure 16.47 shows the first part of the mbuf transfer loop. Figure 16.46 lists the variables updated within the loop.

Variable	Description
moff	the offset of the next byte to transfer when MSG_PEEK is set
offset	the offset of the OOB mark when MSG_PEEK is set
uio_resid	the number of bytes remaining to be transferred
len	the number of bytes to be transferred from this mbuf; may be less than m_len if uio_resid is small, or if the OOB mark is near

Figure 16.46 soreceive function: loop variables.

598–600   During each iteration of the while loop, the data in a single mbuf is transferred to the output chain or to the uio buffers. The loop continues while there are more mbufs, the process's buffers are not full, and no error has occurred.

Check for transition between OOB and regular data

600–605   If, while processing the mbuf chain, the type of the mbuf changes, the transfer stops. This ensures that regular and out-of-band data are not both returned in the same message. This check does not apply to TCP.

Update OOB mark

606–611   The distance to the oobmark is computed and limits the size of the transfer, so the byte before the mark is the last byte transferred. The size of the transfer is also limited by the size of the mbuf. This code does apply to TCP.

612–625   If the data is being returned to the uio buffers, uiomove is called. If the data is being returned as an mbuf chain, uio_resid is adjusted to reflect the number of bytes moved.

To avoid suspending protocol processing for a long time, protocol processing is enabled during the call to uiomove. Additional data may appear in the receive buffer because of protocol processing while uiomove is running.

The code in Figure 16.48 adjusts all the pointers and offsets to prepare for the next mbuf.

```
                                                                              ── uipc_socket.c
598     moff = 0;
599     offset = 0;
600     while (m && uio->uio_resid > 0 && error == 0) {
601         if (m->m_type == MT_OOBDATA) {
602             if (type != MT_OOBDATA)
603                 break;
604         } else if (type == MT_OOBDATA)
605             break;
606         so->so_state &= ~SS_RCVATMARK;
607         len = uio->uio_resid;
608         if (so->so_oobmark && len > so->so_oobmark - offset)
609             len = so->so_oobmark - offset;
610         if (len > m->m_len - moff)
611             len = m->m_len - moff;
612         /*
613          * If mp is set, just pass back the mbufs.
614          * Otherwise copy them out via the uio, then free.
615          * Sockbuf must be consistent here (points to current mbuf,
616          * it points to next record) when we drop priority;
617          * we must note any additions to the sockbuf when we
618          * block interrupts again.
619          */
620         if (mp == 0) {
621             splx(s);
622             error = uiomove(mtod(m, caddr_t) + moff, (int) len, uio);
623             s = splnet();
624         } else
625             uio->uio_resid -= len;
                                                                              ── uipc_socket.c
```

Figure 16.47   soreceive function: uiomove.

Finished with mbuf?

626–646   If all the bytes in the mbuf have been transferred, the mbuf must be discarded or the pointers advanced. If the mbuf contained the end of a logical record, MSG_EOR is set. If MSG_PEEK is set, soreceive skips to the next buffer. If MSG_PEEK is not set, the buffer is discarded if the data was copied by uiomove, or appended to mp if the data is being returned in an mbuf chain.

More data to process

647–657   There may be more data to process in the mbuf if the request didn't consume all the data, if so_oobmark cut the request short, or if additional data arrived during uiomove. If MSG_PEEK is set, moff is updated. If the data is to be returned on an mbuf chain, len bytes are copied and attached to the chain. The mbuf pointers and the receive buffer byte count are updated by the amount of data that was transferred.

Figure 16.49 contains the code that handles the OOB offset and the MSG_EOR processing.

```
                                                            ── uipc_socket.c
626         if (len == m->m_len - moff) {
627             if (m->m_flags & M_EOR)
628                 flags |= MSG_EOR;
629             if (flags & MSG_PEEK) {
630                 m = m->m_next;
631                 moff = 0;
632             } else {
633                 nextrecord = m->m_nextpkt;
634                 sbfree(&so->so_rcv, m);
635                 if (mp) {
636                     *mp = m;
637                     mp = &m->m_next;
638                     so->so_rcv.sb_mb = m = m->m_next;
639                     *mp = (struct mbuf *) 0;
640                 } else {
641                     MFREE(m, so->so_rcv.sb_mb);
642                     m = so->so_rcv.sb_mb;
643                 }
644                 if (m)
645                     m->m_nextpkt = nextrecord;
646             }
647         } else {
648             if (flags & MSG_PEEK)
649                 moff += len;
650             else {
651                 if (mp)
652                     *mp = m_copym(m, 0, len, M_WAIT);
653                 m->m_data += len;
654                 m->m_len -= len;
655                 so->so_rcv.sb_cc -= len;
656             }
657         }
                                                            ── uipc_socket.c
```

Figure 16.48   soreceive function: update buffer.

```
                                                            ── uipc_socket.c
658         if (so->so_oobmark) {
659             if ((flags & MSG_PEEK) == 0) {
660                 so->so_oobmark -= len;
661                 if (so->so_oobmark == 0) {
662                     so->so_state |= SS_RCVATMARK;
663                     break;
664                 }
665             } else {
666                 offset += len;
667                 if (offset == so->so_oobmark)
668                     break;
669             }
670         }
671         if (flags & MSG_EOR)
672             break;
                                                            ── uipc_socket.c
```

Figure 16.49   soreceive function: out-of-band data mark.

Update OOB mark

658–670   If the out-of-band mark is nonzero, it is decremented by the number of bytes transferred. If the mark has been reached, SS_RCVATMARK is set and soreceive breaks out of the while loop. If MSG_PEEK is set, offset is updated instead of so_oobmark.

End of logical record

671–672   If the end of a logical record has been reached, soreceive breaks out of the mbuf processing loop so data from the next logical record is not returned with this message.

The loop in Figure 16.50 waits for more data to arrive when MSG_WAITALL is set and the request is not complete.

```
                                                                  ── uipc_socket.c
673          /*
674           * If the MSG_WAITALL flag is set (for non-atomic socket),
675           * we must not quit until "uio->uio_resid == 0" or an error
676           * termination.  If a signal/timeout occurs, return
677           * with a short count but without error.
678           * Keep sockbuf locked against other readers.
679           */
680          while (flags & MSG_WAITALL && m == 0 && uio->uio_resid > 0 &&
681                  !sosendallatonce(so) && !nextrecord) {
682              if (so->so_error || so->so_state & SS_CANTRCVMORE)
683                  break;
684              error = sbwait(&so->so_rcv);
685              if (error) {
686                  sbunlock(&so->so_rcv);
687                  splx(s);
688                  return (0);
689              }
690              if (m = so->so_rcv.sb_mb)
691                  nextrecord = m->m_nextpkt;
692          }
693      }                                   /* while more data and more space to fill */
                                                                  ── uipc_socket.c
```

Figure 16.50   soreceive function: MSG_WAITALL processing.

MSG_WAITALL

673–681   If MSG_WAITALL is set, there is no more data in the receive buffer (m equals 0), the caller wants more data, sosendallatonce is false, and this is the last record in the receive buffer (nextrecord is null), then soreceive must wait for additional data.

Error or no more data will arrive

682–683   If an error is pending or the connection is closed, the loop is terminated.

Wait for data to arrive

684–689   sbwait returns when the receive buffer is changed by the protocol layer. If the wait was interrupted by a signal (error is nonzero), sosend returns immediately.

Synchronize m and nextrecord with receive buffer

690–692   m and nextrecord are updated, since the receive buffer has been modified by the protocol layer. If data arrived in the mbuf, m will be nonzero and the while loop terminates.

Process next mbuf

693   This is the end of the mbuf processing loop. Control returns to the loop starting on line 600 (Figure 16.47). As long as there is data in the receive buffer, more space to fill, and no error has occurred, the loop continues.

When soreceive stops copying data, the code in Figure 16.51 is executed.

```
                                                                ——— uipc_socket.c
694     if (m && pr->pr_flags & PR_ATOMIC) {
695         flags |= MSG_TRUNC;
696         if ((flags & MSG_PEEK) == 0)
697             (void) sbdroprecord(&so->so_rcv);
698     }
699     if ((flags & MSG_PEEK) == 0) {
700         if (m == 0)
701             so->so_rcv.sb_mb = nextrecord;
702         if (pr->pr_flags & PR_WANTRCVD && so->so_pcb)
703             (*pr->pr_usrreq) (so, PRU_RCVD, (struct mbuf *) 0,
704                             (struct mbuf *) flags, (struct mbuf *) 0,
705                             (struct mbuf *) 0);
706     }
707     if (orig_resid == uio->uio_resid && orig_resid &&
708         (flags & MSG_EOR) == 0 && (so->so_state & SS_CANTRCVMORE) == 0) {
709         sbunlock(&so->so_rcv);
710         splx(s);
711         goto restart;
712     }
713     if (flagsp)
714         *flagsp |= flags;
                                                                ——— uipc_socket.c
```

Figure 16.51   soreceive function: cleanup.

Truncated message

694–698   If the process received a partial message (a datagram or a record) because its receive buffer was too small, the process is notified by setting MSG_TRUNC and the remainder of the message is discarded. MSG_TRUNC (as with all receive flags) is available only to a process through the recvmsg system call, even though soreceive always sets the flags.

End of record processing

699–706   If MSG_PEEK is not set, the next mbuf chain is attached to the receive buffer and, if required, the protocol is notified that the receive operation has been completed by issuing the PRU_RCVD protocol request. TCP uses this feature to update the receive window for the connection.

Nothing transferred

707-712   If soreceive runs to completion, no data is transferred, the end of a record is not reached, and the read-half of the connection is still active, then the buffer is unlocked and soreceive jumps back to restart to continue waiting for data.

713-714   Any flags set during soreceive are returned in *flagsp, the buffer is unlocked, and soreceive returns.

Analysis

soreceive is a complex function. Much of the complication is because of the intricate manipulation of pointers and the multiple types of data (out-of-band, address, control, regular) and multiple destinations (process buffers, mbuf chain).

Similar to sosend, soreceive has collected features over the years. A specialized receive function for each protocol would blur the boundary between the socket layer and the protocol layer, but it would simplify the code considerably.

[Partridge and Pink 1993] describe the creation of a custom soreceive function for UDP to checksum datagrams while they are copied from the receive buffer to the process. They note that modifying the generic soreceive function to support this feature would "make the already complicated socket routines even more complex."

16.13 select System Call

In the following discussion we assume that the reader is familiar with the basic operation and semantics of select. For a detailed discussion of the application interface to select see [Stevens 1992].

Figure 16.52 shows the conditions detected by using select to monitor a socket.

Description	Detected by selecting for:		
	reading	writing	exceptions
data available for reading read-half of connection is closed listen socket has queued connection socket error is pending	• • • •		
space available for writing and a connection exists or is not required write-half of connection is closed socket error is pending		• • •	
OOB synchronization mark is pending			•

Figure 16.52   select system call: socket events.

We start with the first half of the select system call, shown in Figure 16.53.

Validation and setup

390–410   Two arrays of three descriptor sets are allocated on the stack: ibits and obits. They are cleared by bzero. The first argument, nd, must be no larger than the maximum number of descriptors associated with the process. If nd is more than the number of descriptors currently allocated to the process, it is reduced to the current allocation. ni is set to the number of bytes needed to store a bit mask with nd bits (1 bit for each descriptor). For example, if the maximum number of descriptors is 256 (FD_SETSIZE), fd_set is represented as an array of 32-bit integers (NFDBITS), and nd is 65, then:

$$ni = howmany(65,32) \times 4 = 3 \times 4 = 12$$

where howmany(x,y) returns the number of y-bit objects required to store x bits.

Copy file descriptor sets from process

411–418   The getbits macro uses copyin to transfer the file descriptor sets from the process to the three descriptor sets in ibits. If a descriptor set pointer is null, nothing is copied from the process.

Setup timeout value

419–438   If tv is null, timo is set to 0 and select will wait indefinitely. If tv is not null, the timeout value is copied into the kernel and rounded up to the resolution of the hardware clock by itimerfix. The current time is added to the timeout value by timevaladd. The number of clock ticks until the timeout is computed by hzto and saved in timo. If the resulting timeout is 0, timo is set to 1. This prevents select from blocking and implements the nonblocking semantics of an all-0s timeval structure.

The second half of select, shown in Figure 16.54, scans the file descriptors indicated by the process and returns when one or more become ready, or the timer expires, or a signal occurs.

Scan file descriptors

439–442   The loop that starts at retry continues until select can return. The current value of the global integer nselcoll is saved and the P_SELECT flag is set in the calling process's control block. If either of these change while selscan (Figure 16.55) is checking the file descriptors, it indicates that the status of a descriptor has changed because of interrupt processing and select must rescan the descriptors. selscan looks at every descriptor set in the three input descriptor sets and sets the matching descriptor in the output set if the descriptor is ready.

Error or some descriptors are ready

443–444   Return immediately if an error occurred or if a descriptor is ready.

Timeout expired?

445–451   If the process supplied a time limit and the current time has advanced beyond the timeout value, return immediately.

—— sys_generic.c

```
390 struct select_args {
391     u_int   nd;
392     fd_set *in, *ou, *ex;
393     struct timeval *tv;
394 };

395 select(p, uap, retval)
396 struct proc *p;
397 struct select_args *uap;
398 int     *retval;
399 {
400     fd_set  ibits[3], obits[3];
401     struct timeval atv;
402     int     s, ncoll, error = 0, timo;
403     u_int   ni;

404     bzero((caddr_t) ibits, sizeof(ibits));
405     bzero((caddr_t) obits, sizeof(obits));
406     if (uap->nd > FD_SETSIZE)
407         return (EINVAL);
408     if (uap->nd > p->p_fd->fd_nfiles)
409         uap->nd = p->p_fd->fd_nfiles;   /* forgiving; slightly wrong */
410     ni = howmany(uap->nd, NFDBITS) * sizeof(fd_mask);

411 #define getbits(name, x) \
412     if (uap->name && \
413         (error = copyin((caddr_t)uap->name, (caddr_t)&ibits[x], ni))) \
414         goto done;
415     getbits(in, 0);
416     getbits(ou, 1);
417     getbits(ex, 2);
418 #undef  getbits

419     if (uap->tv) {
420         error = copyin((caddr_t) uap->tv, (caddr_t) & atv,
421                        sizeof(atv));
422         if (error)
423             goto done;
424         if (itimerfix(&atv)) {
425             error = EINVAL;
426             goto done;
427         }
428         s = splclock();
429         timevaladd(&atv, (struct timeval *) &time);
430         timo = hzto(&atv);
431         /*
432          * Avoid inadvertently sleeping forever.
433          */
434         if (timo == 0)
435             timo = 1;
436         splx(s);
437     } else
438         timo = 0;
```

—— sys_generic.c

Figure 16.53   select function: initialization.

```
                                                            ———————— sys_generic.c
439    retry:
440        ncoll = nselcoll;
441        p->p_flag |= P_SELECT;
442        error = selscan(p, ibits, obits, uap->nd, retval);
443        if (error || *retval)
444            goto done;
445        s = splhigh();
446        /* this should be timercmp(&time, &atv, >=) */
447        if (uap->tv && (time.tv_sec > atv.tv_sec ||
448                    time.tv_sec == atv.tv_sec && time.tv_usec >= atv.tv_usec)) {
449            splx(s);
450            goto done;
451        }
452        if ((p->p_flag & P_SELECT) == 0 || nselcoll != ncoll) {
453            splx(s);
454            goto retry;
455        }
456        p->p_flag &= ~P_SELECT;
457        error = tsleep((caddr_t) & selwait, PSOCK | PCATCH, "select", timo);
458        splx(s);
459        if (error == 0)
460            goto retry;
461    done:
462        p->p_flag &= ~P_SELECT;
463        /* select is not restarted after signals... */
464        if (error == ERESTART)
465            error = EINTR;
466        if (error == EWOULDBLOCK)
467            error = 0;
468    #define putbits(name, x) \
469        if (uap->name && \
470            (error2 = copyout((caddr_t)&obits[x], (caddr_t)uap->name, ni))) \
471            error = error2;
472        if (error == 0) {
473            int     error2;

474            putbits(in, 0);
475            putbits(ou, 1);
476            putbits(ex, 2);
477    #undef putbits
478        }
479        return (error);
480    }
                                                            ———————— sys_generic.c
```

Figure 16.54 select function: second half.

Status changed during `selscan`

452–455   `selscan` can be interrupted by protocol processing. If the socket is modified during the interrupt, `P_SELECT` and `nselcoll` are changed and `select` must rescan the descriptors.

Wait for buffer changes

456–460   All processes calling `select` use `selwait` as the wait channel when they call `tsleep`. With Figure 16.60 we show that this causes some inefficiencies if more than one process is waiting for the same socket buffer. If `tsleep` returns without an error, `select` jumps to `retry` to rescan the descriptors.

Ready to return

461–480   At `done`, `P_SELECT` is cleared, `ERESTART` is changed to `EINTR`, and `EWOULDBLOCK` is changed to 0. These changes ensure that `EINTR` is returned when a signal occurs during `select` and 0 is returned when a timeout occurs.
          The output descriptor sets are copied back to the process and `select` returns.

`selscan` Function

The heart of `select` is the `selscan` function shown in Figure 16.55. For every bit set in one of the three descriptor sets, `selscan` computes the descriptor associated with the bit and dispatches control to the `fo_select` function associated with the descriptor. For sockets, this is the `soo_select` function.

Locate descriptors to be monitored

481–496   The first `for` loop iterates through each of the three descriptor sets: read, write, and exception. The second `for` loop interates within each descriptor set. This loop is executed once for every 32 bits (`NFDBITS`) in the set.
          The inner `while` loop checks all the descriptors identified by the 32-bit mask extracted from the current descriptor set and stored in `bits`. The function `ffs` returns the position within `bits` of the first 1 bit, starting at the low-order bit. For example, if `bits` is `1000` (with 28 leading 0s), `ffs(bits)` is 4.

Poll descriptor

497–500   From `i` and the return value of `ffs`, the descriptor associated with the bit is computed and stored in `fd`. The bit is cleared in `bits` (but not in the input descriptor set), the `file` structure associated with the descriptor is located, and `fo_select` is called.
          The second argument to `fo_select` is one of the elements in the `flag` array. `msk` is the index of the outer `for` loop. So the first time through the loop, the second argument is `FREAD`, the second time it is `FWRITE`, and the third time it is 0. `EBADF` is returned if the descriptor is not valid.

Descriptor is ready

501–504   When a descriptor is found to be ready, the matching bit is set in the output descriptor set and `n` (the number of matches) is incremented.

505–510   The loops continue until all the descriptors are polled. The number of ready descriptors is returned in `*retval`.

```
                                                                   sys_generic.c
481 selscan(p, ibits, obits, nfd, retval)
482 struct proc *p;
483 fd_set *ibits, *obits;
484 int     nfd, *retval;
485 {
486     struct filedesc *fdp = p->p_fd;
487     int     msk, i, j, fd;
488     fd_mask bits;
489     struct file *fp;
490     int     n = 0;
491     static int flag[3] =
492     {FREAD, FWRITE, 0};

493     for (msk = 0; msk < 3; msk++) {
494         for (i = 0; i < nfd; i += NFDBITS) {
495             bits = ibits[msk].fds_bits[i / NFDBITS];
496             while ((j = ffs(bits)) && (fd = i + --j) < nfd) {
497                 bits &= ~(1 << j);
498                 fp = fdp->fd_ofiles[fd];
499                 if (fp == NULL)
500                     return (EBADF);
501                 if ((*fp->f_ops->fo_select) (fp, flag[msk], p)) {
502                     FD_SET(fd, &obits[msk]);
503                     n++;
504                 }
505             }
506         }
507     }
508     *retval = n;
509     return (0);
510 }
                                                                   sys_generic.c
```

Figure 16.55   selscan function.

soo_select Function

For every descriptor that selscan finds in the input descriptor sets, it calls the function referenced by the fo_select pointer in the fileops structure (Section 15.5) associated with the descriptor. In this text, we are interested only in socket descriptors and the soo_select function shown in Figure 16.56.

105–112   Each time soo_select is called, it checks the status of only one descriptor. If the descriptor is ready relative to the conditions specified in which, the function returns 1 immediately. If the descriptor is not ready, selrecord marks either the socket's receive or send buffer to indicate that a process is selecting on the buffer and then soo_select returns 0.

Figure 16.52 showed the read, write, and exceptional conditions for sockets. Here we see that the macros soreadable and sowriteable are consulted by soo_select. These macros are defined in sys/socketvar.h.

sys_socket.c

```
105 soo_select(fp, which, p)
106 struct file *fp;
107 int      which;
108 struct proc *p;
109 {
110     struct socket *so = (struct socket *) fp->f_data;
111     int      s = splnet();

112     switch (which) {

113     case FREAD:
114         if (soreadable(so)) {
115             splx(s);
116             return (1);
117         }
118         selrecord(p, &so->so_rcv.sb_sel);
119         so->so_rcv.sb_flags |= SB_SEL;
120         break;

121     case FWRITE:
122         if (sowriteable(so)) {
123             splx(s);
124             return (1);
125         }
126         selrecord(p, &so->so_snd.sb_sel);
127         so->so_snd.sb_flags |= SB_SEL;
128         break;

129     case 0:
130         if (so->so_oobmark || (so->so_state & SS_RCVATMARK)) {
131             splx(s);
132             return (1);
133         }
134         selrecord(p, &so->so_rcv.sb_sel);
135         so->so_rcv.sb_flags |= SB_SEL;
136         break;
137     }
138     splx(s);
139     return (0);
140 }
```

sys_socket.c

Figure 16.56   soo_select function.

Is socket readable?

113–120   The soreadable macro is:

```
#define soreadable(so) \
    ((so)->so_rcv.sb_cc >= (so)->so_rcv.sb_lowat || \
    ((so)->so_state & SS_CANTRCVMORE) || \
    (so)->so_qlen || (so)->so_error)
```

Since the receive low-water mark for UDP and TCP defaults to 1 (Figure 16.4), the socket is readable if any data is in the receive buffer, if the read-half of the connection is closed, if any connections are ready to be accepted, or if there is an error pending.

Is socket writeable?

121–128   The `sowriteable` macro is:

```
#define sowriteable(so) \
    (sbspace(&(so)->so_snd) >= (so)->so_snd.sb_lowat && \
    (((so)->so_state&SS_ISCONNECTED) || \
      ((so)->so_proto->pr_flags&PR_CONNREQUIRED)==0) || \
    ((so)->so_state & SS_CANTSENDMORE) || \
    (so)->so_error)
```

The default send low-water mark for UDP and TCP is 2048. For UDP, `sowriteable` is always true because `sbspace` is always equal to `sb_hiwat`, which is always greater than or equal to `sb_lowat`, and a connection is not required.

For TCP, the socket is not writeable when the free space in the send buffer is less than 2048 bytes. The other cases are described in Figure 16.52.

Are there any exceptional conditions pending?

129–140   For exceptions, `so_oobmark` and the `SS_RCVATMARK` flags are examined. An exceptional condition exists until the process has read past the synchronization mark in the data stream.

`selrecord` Function

Figure 16.57 shows the definition of the `selinfo` structure stored with each send and receive buffer (the `sb_sel` member from Figure 16.3).

```
                                                              ──── select.h
41 struct selinfo {
42     pid_t   si_pid;          /* process to be notified */
43     short   si_flags;        /* 0 or SI_COLL */
44 };
                                                              ──── select.h
```

Figure 16.57   `selinfo` structure.

41–44   When only one process has called `select` for a given socket buffer, `si_pid` is the process ID of the waiting process. When additional processes call select on the same buffer, `SI_COLL` is set in `si_flags`. This is called a *collision*. This is the only flag currently defined for `si_flags`.

The `selrecord` function shown in Figure 16.58 is called when `soo_select` finds a descriptor that is not ready. The function records enough information so that the process is awakened by the protocol processing layer when the buffer changes.

Already selecting on this descriptor

522–531   The first argument to `selrecord` points to the `proc` structure for the selecting process. The second argument points to the `selinfo` record to update (`so_snd.sb_sel` or `so_rcv.sb_sel`). If this process is already recorded in the `selinfo` record for this socket buffer, the function returns immediately. For example, the process called `select` with the read and exception bits set for the same descriptor.

sys_generic.c

```
522 void
523 selrecord(selector, sip)
524 struct proc *selector;
525 struct selinfo *sip;
526 {
527     struct proc *p;
528     pid_t    mypid;

529     mypid = selector->p_pid;
530     if (sip->si_pid == mypid)
531         return;
532     if (sip->si_pid && (p = pfind(sip->si_pid)) &&
533         p->p_wchan == (caddr_t) & selwait)
534         sip->si_flags |= SI_COLL;
535     else
536         sip->si_pid = mypid;
537 }
```

sys_generic.c

Figure 16.58   selrecord function.

Select collision with another process?

532–534   If another process is already selecting on this buffer, SI_COLL is set.

No collision

535–537   If there is no other process already selecting on this buffer, si_pid is 0 so the ID of
the current process is saved in si_pid.

selwakeup **Function**

When protocol processing changes the state of a socket buffer and only one process is
selecting on the buffer, Net/3 can immediately put that process on the run queue based
on the information it finds in the selinfo structure.

When the state changes and there is more than one process selecting on the buffer
(SI_COLL is set), Net/3 has no way of determining the set of processes interested in the
buffer. When we discussed the code in Figure 16.54, we pointed out that *every* process
that calls select uses selwait as the wait channel when calling tsleep. This means
the corresponding wakeup will schedule *all* the processes that are blocked in
select—even those that are not interested in activity on the buffer.

Figure 16.59 shows how selwakeup is called.

The protocol processing layer is responsible for notifying the socket layer by calling
one of the functions listed at the bottom of Figure 16.59 when an event occurs that
changes the state of a socket. The three functions shown at the bottom of Figure 16.59
cause selwakeup to be called and any process selecting on the socket to be scheduled
to run.

selwakeup is shown in Figure 16.60.

541–548   If si_pid is 0, there is no process selecting on the buffer and the function returns
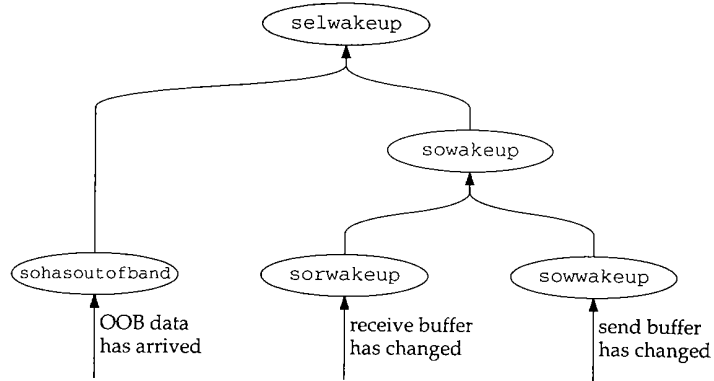immediately.

Figure 16.59 selwakeup processing.

```
                                                                     ── sys_generic.c
541 void
542 selwakeup(sip)
543 struct selinfo *sip;
544 {
545     struct proc *p;
546     int     s;

547     if (sip->si_pid == 0)
548         return;
549     if (sip->si_flags & SI_COLL) {
550         nselcoll++;
551         sip->si_flags &= ~SI_COLL;
552         wakeup((caddr_t) & selwait);
553     }
554     p = pfind(sip->si_pid);
555     sip->si_pid = 0;
556     if (p != NULL) {
557         s = splhigh();
558         if (p->p_wchan == (caddr_t) & selwait) {
559             if (p->p_stat == SSLEEP)
560                 setrunnable(p);
561             else
562                 unsleep(p);
563         } else if (p->p_flag & P_SELECT)
564             p->p_flag &= ~P_SELECT;
565         splx(s);
566     }
567 }
                                                                     ── sys_generic.c
```

Figure 16.60 selwakeup function.

Wake all processes during a collision

549–553   If more than one process is selecting on the affected socket, nselcoll is incremented, the collision flag is cleared, and every process blocked in select is awakened. As mentioned with Figure 16.54, nselcoll forces select to rescan the descriptors if the buffers change before the process has blocked in tsleep (Exercise 16.9).

554–567   If the process identified by si_pid is waiting on selwait, it is scheduled to run. If the process is waiting on some other wait channel, the P_SELECT flag is cleared. The process can be waiting on some other wait channel if selrecord is called for a valid descriptor and then selscan finds a bad file descriptor in one of the descriptor sets. selscan returns EBADF, but the previously modified selinfo record is not reset. Later, when selwakeup runs, selwakeup may find the process identified by sel_pid is no longer waiting on the socket buffer so the selinfo information is ignored.

Only one process is awakened during selwakeup unless multiple processes are sharing the same descriptor (i.e., the same socket buffers), which is rare. On the machines to which the authors had access, nselcoll was always 0, which confirms the statement that select collisions are rare.

16.14 Summary

In this chapter we looked at the read, write, and select system calls for sockets.

We saw that sosend handles all output between the socket layer and the protocol processing layer and that soreceive handles all input.

The organization of the send buffer and receive buffers was described, as well as the default values and semantics of the high-water and low-water marks for the buffers.

The last part of the chapter discussed the implementation of select. We showed that when only one process is selecting on a descriptor, the protocol processing layer will awaken only the process identified in the selinfo structure. When there is a collision and more than one process is selecting on a descriptor, the protocol layer has no choice but to awaken every process that is selecting on *any* descriptor.

Exercises

16.1   What happens to resid in sosend when an unsigned integer larger than the maximum positive signed integer is passed in the write system call?

16.2   When sosend puts less than MCLBYTES of data in a cluster, space is reduced by the full MCLBYTES and may become negative, which terminates the loop that fills mbufs for atomic protocols. Is this a problem?

16.3   Datagram and stream protocols have very different semantics. Divide the sosend and soreceive functions each into two functions, one to handle messages, and one to handle streams. Other than making the code clearer, what are the advantages of making this change?

16.4   For PR_ATOMIC protocols, each write call specifies an implicit message boundary. The

socket layer delivers the message as a single unit to the protocol. The MSG_EOR flag allows a process to specify explicit message boundaries. Why is the implicit technique insufficient?

16.5   What happens when sosend cannot immediately acquire a lock on the send buffer when the socket descriptor is marked as nonblocking and the process does not specify MSG_DONTWAIT?

16.6   Under what circumstances would sb_cc < sb_hiwat yet sbspace would report no free space? Why should a process be blocked in this case?

16.7   Why isn't the length of a control message copied back to the process by recvit as is the name length?

16.8   Why does soreceive clear MSG_EOR?

16.9   What might happen if the nselcoll code were removed from select and selwakeup?

16.10 Modify the select system call to return the time remaining in the timer when select returns.

17

Socket Options

17.1 Introduction

We complete our discussion of the socket layer in this chapter by discussing several system calls that modify the behavior of sockets.

The setsockopt and getsockopt system calls were introduced in Section 8.8, where we described the options that provide access to IP features. In this chapter we show the implementation of these two system calls and the socket-level options that are controlled through them.

The ioctl function was introduced in Section 4.4, where we described the protocol-independent ioctl commands for network interface configuration. In Section 6.7 we described the IP specific ioctl commands used to assign network masks as well as unicast, broadcast, and destination addresses. In this chapter we describe the implementation of ioctl and the related features of the fcntl function.

Finally, we describe the getsockname and getpeername system calls, which return address information for sockets and connections.

Figure 17.1 shows the functions that implement the socket option system calls. The shaded functions are described in this chapter.
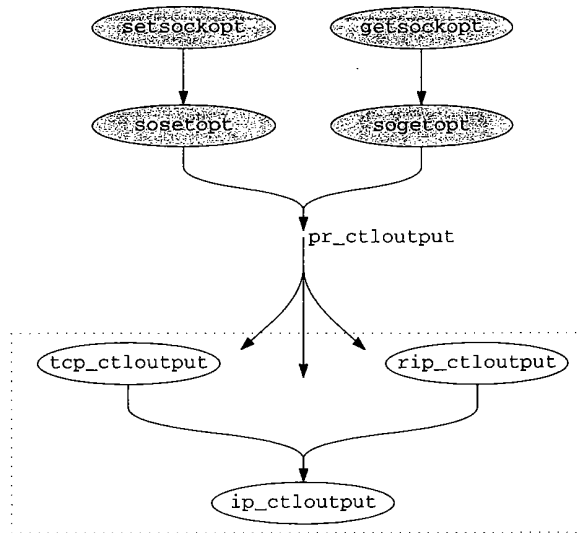
537

Figure 17.1   setsockopt and getsockopt system calls.

17.2 Code Introduction

The code in this chapter comes from the four files listed in Figure 17.2.

File	Description
kern/kern_descrip.c	fcntl system call
kern/uipc_syscalls.c	setsockopt, getsockopt, getsockname, and getpeername system calls
kern/uipc_socket.c	socket layer processing for setsockopt and getsockopt
kern/sys_socket.c	ioctl system call for sockets

Figure 17.2   Files discussed in this chapter.

Global Variables and Statistics

No new global variables are introduced and no statistics are collected by the system calls we describe in this chapter.

17.3 `setsockopt` System Call

Figure 8.29 listed the different protocol levels that can be accessed with this function (and with `getsockopt`). In this chapter we focus on the `SOL_SOCKET` level options, which are listed in Figure 17.3.

optname	*optval* type	Variable	Description
SO_SNDBUF	int	so_snd.sb_hiwat	send buffer high-water mark
SO_RCVBUF	int	so_rcv.sb_hiwat	receive buffer high-water mark
SO_SNDLOWAT	int	so_snd.sb_lowat	send buffer low-water mark
SO_RCVLOWAT	int	so_rcv.sb_lowat	receive buffer low-water mark
SO_SNDTIMEO	struct timeval	so_snd.sb_timeo	send timeout
SO_RCVTIMEO	struct timeval	so_rcv.sb_timeo	receive timeout
SO_DEBUG	int	so_options	record debugging information for this socket
SO_REUSEADDR	int	so_options	socket can reuse a local address
SO_REUSEPORT	int	so_options	socket can reuse a local port
SO_KEEPALIVE	int	so_options	protocol probes idle connections
SO_DONTROUTE	int	so_options	bypass routing tables
SO_BROADCAST	int	so_options	socket allows broadcast messages
SO_USELOOPBACK	int	so_options	routing domain sockets only; sending process receives its own routing messages
SO_OOBINLINE	int	so_options	protocol queues out-of-band data inline
SO_LINGER	struct linger	so_linger	socket lingers on close
SO_ERROR	int	so_error	get error status and clear; getsockopt only
SO_TYPE	int	so_type	get socket type; getsockopt only
other			ENOPROTOOPT returned

Figure 17.3 `setsockopt` and `getsockopt` options.

The prototype for `setsockopt` is

```
int setsockopt(int s, int level, int optname, void *optval, int optlen);
```

Figure 17.4 shows the code for this system call.

565–597   `getsock` locates the `file` structure for the socket descriptor. If `val` is nonnull, `valsize` bytes of data are copied from the process into an mbuf allocated by `m_get`. The data associated with an option can be no more than `MLEN` bytes in length, so if `valsize` is larger than `MLEN`, then `EINVAL` is returned. `sosetopt` is called and its value is returned.

uipc_syscalls.c
```
565 struct setsockopt_args {
566     int     s;
567     int     level;
568     int     name;
569     caddr_t val;
570     int     valsize;
571 };

572 setsockopt(p, uap, retval)
573 struct proc *p;
574 struct setsockopt_args *uap;
575 int     *retval;
576 {
577     struct file *fp;
578     struct mbuf *m = NULL;
579     int     error;

580     if (error = getsock(p->p_fd, uap->s, &fp))
581         return (error);
582     if (uap->valsize > MLEN)
583         return (EINVAL);
584     if (uap->val) {
585         m = m_get(M_WAIT, MT_SOOPTS);
586         if (m == NULL)
587             return (ENOBUFS);
588         if (error = copyin(uap->val, mtod(m, caddr_t),
589                            (u_int) uap->valsize)) {
590             (void) m_free(m);
591             return (error);
592         }
593         m->m_len = uap->valsize;
594     }
595     return (sosetopt((struct socket *) fp->f_data, uap->level,
596                      uap->name, m));
597 }
```
uipc_syscalls.c

Figure 17.4   setsockopt system call.

sosetopt Function

This function processes all the socket-level options and passes any other options to the pr_ctloutput function for the protocol associated with the socket. Figure 17.5 shows an overview of the function.

752–764   If the option is not for the socket level (SOL_SOCKET), the PRCO_SETOPT request is issued to the underlying protocol. Note that the protocol's pr_ctloutput function is being called and not its pr_usrreq function. Figure 17.6 shows which function is called for the Internet protocols.

765   The switch statement handles the socket-level options.

841–844   An unrecognized option causes ENOPROTOOPT to be returned after the mbuf holding the option is released.

uipc_socket.c

```
752 sosetopt(so, level, optname, m0)
753 struct socket *so;
754 int     level, optname;
755 struct mbuf *m0;
756 {
757     int     error = 0;
758     struct mbuf *m = m0;

759     if (level != SOL_SOCKET) {
760         if (so->so_proto && so->so_proto->pr_ctloutput)
761             return ((*so->so_proto->pr_ctloutput)
762                     (PRCO_SETOPT, so, level, optname, &m0));
763         error = ENOPROTOOPT;
764     } else {
765         switch (optname) {


                                    /* socket option processing */


841         default:
842             error = ENOPROTOOPT;
843             break;
844         }
845         if (error == 0 && so->so_proto && so->so_proto->pr_ctloutput) {
846             (void) ((*so->so_proto->pr_ctloutput)
847                     (PRCO_SETOPT, so, level, optname, &m0));
848             m = NULL;           /* freed by protocol */
849         }
850     }
851 bad:
852     if (m)
853         (void) m_free(m);
854     return (error);
855 }
```

uipc_socket.c

Figure 17.5 sosetopt function.

Protocol	pr_ctloutput Function	Reference
UDP	ip_ctloutput	Section 8.8
TCP	tcp_ctloutput	Section 30.6
ICMP IGMP raw IP	rip_ctloutput and ip_ctloutput	Section 8.8 and Section 32.8

Figure 17.6 pr_ctloutput functions.

845–855   Unless an error occurs, control always falls through the switch, where the option is passed to the associated protocol in case the protocol layer needs to respond to the request as well as the socket layer. None of the Internet protocols expect to process the socket-level options.

Notice that the return value from the call to the `pr_ctloutput` function is explicitly discarded in case the option is not expected by the protocol. `m` is set to null to avoid the call to `m_free`, since the protocol layer is responsible for releasing the mbuf.

Figure 17.7 shows the `linger` option and the options that set a single flag in the socket structure.

```
                                                                    ——— uipc_socket.c
766          case SO_LINGER:
767              if (m == NULL || m->m_len != sizeof(struct linger)) {
768                  error = EINVAL;
769                  goto bad;
770              }
771              so->so_linger = mtod(m, struct linger *)->l_linger;
772              /* fall thru... */

773          case SO_DEBUG:
774          case SO_KEEPALIVE:
775          case SO_DONTROUTE:
776          case SO_USELOOPBACK:
777          case SO_BROADCAST:
778          case SO_REUSEADDR:
779          case SO_REUSEPORT:
780          case SO_OOBINLINE:
781              if (m == NULL || m->m_len < sizeof(int)) {
782                  error = EINVAL;
783                  goto bad;
784              }
785              if (*mtod(m, int *))
786                  so->so_options |= optname;
787              else
788                  so->so_options &= ~optname;
789              break;
                                                                    ——— uipc_socket.c
```

Figure 17.7   `sosetopt` function: `linger` and flag options.

766–772   The linger option expects the process to pass a `linger` structure:

```
struct linger {
    int    l_onoff;     /* option on/off */
    int    l_linger;    /* linger time in seconds */
};
```

After making sure the process has passed data and it is the size of a `linger` structure, the `l_linger` member is copied into `so_linger`. The option is enabled or disabled after the next set of `case` statements. `so_linger` was described in Section 15.15 with the `close` system call.

773–789   These options are boolean flags set when the process passes a nonzero value and cleared when 0 is passed. The first check makes sure an integer-sized object (or larger) is present in the mbuf and then sets or clears the appropriate option.

Figure 17.8 shows the socket buffer options.

```
                                                            ——— uipc_socket.c
790        case SO_SNDBUF:
791        case SO_RCVBUF:
792        case SO_SNDLOWAT:
793        case SO_RCVLOWAT:
794            if (m == NULL || m->m_len < sizeof(int)) {
795                error = EINVAL;
796                goto bad;
797            }
798            switch (optname) {

799            case SO_SNDBUF:
800            case SO_RCVBUF:
801                if (sbreserve(optname == SO_SNDBUF ?
802                            &so->so_snd : &so->so_rcv,
803                            (u_long) * mtod(m, int *)) == 0) {
804                    error = ENOBUFS;
805                    goto bad;
806                }
807                break;

808            case SO_SNDLOWAT:
809                so->so_snd.sb_lowat = *mtod(m, int *);
810                break;
811            case SO_RCVLOWAT:
812                so->so_rcv.sb_lowat = *mtod(m, int *);
813                break;
814            }
815            break;
                                                            ——— uipc_socket.c
```

Figure 17.8   sosetopt function: socket buffer options.

790–815   This set of options changes the size of the send and receive buffers in a socket. The first test makes sure the required integer has been provided for all four options. For SO_SNDBUF and SO_RCVBUF, sbreserve adjusts the high-water mark but does no buffer allocation. For SO_SNDLOWAT and SO_RCVLOWAT, the low-water marks are adjusted.

Figure 17.9 shows the timeout options.

816–824   The timeout value for SO_SNDTIMEO and SO_RCVTIMEO is specified by the process in a timeval structure. If the right amount of data is not available, EINVAL is returned.

825–830   The time interval stored in the timeval structure must be small enough so that when it is represented as clock ticks, it fits within a short integer, since sb_timeo is a short integer.

The code on line 826 is incorrect. The time interval cannot be represented as a short integer if:

uipc_socket.c

```
816              case SO_SNDTIMEO:
817              case SO_RCVTIMEO:
818                  {
819                      struct timeval *tv;
820                      short    val;

821                      if (m == NULL || m->m_len < sizeof(*tv)) {
822                          error = EINVAL;
823                          goto bad;
824                      }
825                      tv = mtod(m, struct timeval *);
826                      if (tv->tv_sec > SHRT_MAX / hz - hz) {
827                          error = EDOM;
828                          goto bad;
829                      }
830                      val = tv->tv_sec * hz + tv->tv_usec / tick;

831                      switch (optname) {

832                      case SO_SNDTIMEO:
833                          so->so_snd.sb_timeo = val;
834                          break;
835                      case SO_RCVTIMEO:
836                          so->so_rcv.sb_timeo = val;
837                          break;
838                      }
839                      break;
840                  }
```

uipc_socket.c

Figure 17.9   sosetopt function: timeout options.

$$\texttt{tv\_sec} \times \texttt{hz} + \frac{\texttt{tv\_usec}}{\texttt{tick}} > \texttt{SHRT\_MAX}$$

where

$$\texttt{tick} = \frac{1,000,000}{\texttt{hz}} \text{ and SHRT\_MAX} = 32767$$

So EDOM should be returned if

$$\texttt{tv\_sec} > \frac{\texttt{SHRT\_MAX}}{\texttt{hz}} - \frac{\texttt{tv\_usec}}{\texttt{tick} \times \texttt{hz}} = \frac{\texttt{SHRT\_MAX}}{\texttt{hz}} - \frac{\texttt{tv\_usec}}{1,000,000}$$

The last term in this equation is not hz as specified in the code. The correct test is

```
if (tv->tv_sec*hz + tv->tv_usec/tick > SHRT_MAX)
```

but see Exercise 17.3 for more discussion.

831–840   The converted time, val, is saved in the send or receive buffer as requested. sb_timeo limits the amount of time a process will wait for data in the receive buffer or space in the send buffer. See Sections 16.7 and 16.12 for details.

> The timeout values are passed as the last argument to tsleep, which expects an integer, so the process is limited to 65535 ticks. At 100 Hz, this less than 11 minutes.

17.4 getsockopt **System Call**

getsockopt returns socket and protocol options as requested. The prototype for this system call is

    int getsockopt(int s, int level, int name, caddr_t val, int *valsize);

The code is shown in Figure 17.10.

```
                                                                  ── uipc_syscalls.c
598 struct getsockopt_args {
599     int     s;
600     int     level;
601     int     name;
602     caddr_t val;
603     int     *avalsize;
604 };

605 getsockopt(p, uap, retval)
606 struct proc *p;
607 struct getsockopt_args *uap;
608 int     *retval;
609 {
610     struct file *fp;
611     struct mbuf *m = NULL;
612     int     valsize, error;

613     if (error = getsock(p->p_fd, uap->s, &fp))
614         return (error);
615     if (uap->val) {
616         if (error = copyin((caddr_t) uap->avalsize, (caddr_t) & valsize,
617                            sizeof(valsize)))
618             return (error);
619     } else
620         valsize = 0;
621     if ((error = sogetopt((struct socket *) fp->f_data, uap->level,
622             uap->name, &m)) == 0 && uap->val && valsize && m != NULL) {
623         if (valsize > m->m_len)
624             valsize = m->m_len;
625         error = copyout(mtod(m, caddr_t), uap->val, (u_int) valsize);
626         if (error == 0)
627             error = copyout((caddr_t) & valsize,
628                             (caddr_t) uap->avalsize, sizeof(valsize));
629     }
630     if (m != NULL)
631         (void) m_free(m);
632     return (error);
633 }
                                                                  ── uipc_syscalls.c
```

Figure 17.10 getsockopt system call.

598–633   The code should look pretty familiar by now. getsock locates the socket, the size of the option buffer is copied into the kernel, and sogetopt is called to get the value of the requested option. The data returned by sogetopt is copied out to the buffer in the process along with the possibly new length of the buffer. It is possible that the data will

be silently truncated if the process did not provide a large enough buffer. As usual, the mbuf holding the option data is released before the function returns.

sogetopt Function

As with `sosetopt`, the `sogetopt` function handles the socket-level options and passes any other options to the protocol associated with the socket. The beginning and end of the function are shown in Figure 17.11.

─── uipc_socket.c

```
856 sogetopt(so, level, optname, mp)
857 struct socket *so;
858 int     level, optname;
859 struct mbuf **mp;
860 {
861     struct mbuf *m;

862     if (level != SOL_SOCKET) {
863         if (so->so_proto && so->so_proto->pr_ctloutput) {
864             return ((*so->so_proto->pr_ctloutput)
865                     (PRCO_GETOPT, so, level, optname, mp));
866         } else
867             return (ENOPROTOOPT);
868     } else {
869         m = m_get(M_WAIT, MT_SOOPTS);
870         m->m_len = sizeof(int);

871         switch (optname) {
```

```
                        /* socket option processing */
```

```
918         default:
919             (void) m_free(m);
920             return (ENOPROTOOPT);
921         }
922         *mp = m;
923         return (0);
924     }
925 }
```
─── uipc_socket.c

Figure 17.11 `sogetopt` function: overview.

856–871   As with `sosetopt`, options that do not pertain to the socket level are immediately passed to the protocol level through the `PRCO_GETOPT` protocol request. The protocol returns the requested option in the mbuf pointed to by `*mp`.

For socket-level options, a standard mbuf is allocated to hold the option value, which is normally an integer, so `m_len` is set to the size of an integer. The appropriate option is copied into the mbuf by the code in the `switch` statement.

918–925   If the `default` case is taken by the `switch`, the mbuf is released and `ENOPROTOOPT` returned. Otherwise, after the `switch` statement, the pointer to the

mbuf is saved in *mp. When this function returns, getsockopt copies the option from the mbuf to the process and releases the mbuf.

In Figure 17.12 the linger option and the options that are implemented as boolean flags are processed.

```
                                                                  uipc_socket.c
872        case SO_LINGER:
873            m->m_len = sizeof(struct linger);
874            mtod(m, struct linger *)->l_onoff =
875                    so->so_options & SO_LINGER;
876            mtod(m, struct linger *)->l_linger = so->so_linger;
877            break;

878        case SO_USELOOPBACK:
879        case SO_DONTROUTE:
880        case SO_DEBUG:
881        case SO_KEEPALIVE:
882        case SO_REUSEADDR:
883        case SO_REUSEPORT:
884        case SO_BROADCAST:
885        case SO_OOBINLINE:
886            *mtod(m, int *) = so->so_options & optname;
887            break;
                                                                  uipc_socket.c
```

Figure 17.12   sogetopt function: SO_LINGER and boolean options.

872–877   The SO_LINGER option requires two copies, one for the flag into l_onoff and a second for the linger time into l_linger.

878–887   The remaining options are implemented as boolean flags. so_options is masked with optname, which results in a nonzero value if the option is on and 0 if the option is off. Notice that the return value is not necessarily 1 when the flag is on.

In the next part of sogetopt (Figure 17.13), the integer-valued options are copied into the mbuf.

```
                                                                  uipc_socket.c
888        case SO_TYPE:
889            *mtod(m, int *) = so->so_type;
890            break;

891        case SO_ERROR:
892            *mtod(m, int *) = so->so_error;
893            so->so_error = 0;
894            break;

895        case SO_SNDBUF:
896            *mtod(m, int *) = so->so_snd.sb_hiwat;
897            break;

898        case SO_RCVBUF:
899            *mtod(m, int *) = so->so_rcv.sb_hiwat;
900            break;
```

```
901             case SO_SNDLOWAT:
902                 *mtod(m, int *) = so->so_snd.sb_lowat;
903                 break;

904             case SO_RCVLOWAT:
905                 *mtod(m, int *) = so->so_rcv.sb_lowat;
906                 break;                                              uipc_socket.c
```

Figure 17.13   sogetopt function: integer valued options.

888–906   Each option is copied as an integer into the mbuf. Notice that some of the options are stored as shorts in the kernel (e.g., the high-water and low-water marks) but returned as integers. Also, so_error is cleared once the value is copied into the mbuf. This is the only time that a call to getsockopt changes the state of the socket.

The fourth and last part of sogetopt is shown in Figure 17.14, where the SO_SNDTIMEO and SO_RCVTIMEO options are handled.

```
                                                                     uipc_socket.c
907             case SO_SNDTIMEO:
908             case SO_RCVTIMEO:
909                 {
910                     int     val = (optname == SO_SNDTIMEO ?
911                                     so->so_snd.sb_timeo : so->so_rcv.sb_timeo);

912                     m->m_len = sizeof(struct timeval);
913                     mtod(m, struct timeval *)->tv_sec = val / hz;
914                     mtod(m, struct timeval *)->tv_usec =
915                                 (val % hz) / tick;
916                     break;
917                 }                                               uipc_socket.c
```

Figure 17.14   sogetopt function: timeout options.

907–917   The sb_timeo value from the send or receive buffer is copied into val. A timeval structure is constructed in the mbuf based on the clock ticks in val.

There is a bug in the calculation of tv_usec. The expression should be "(val % hz) * tick".

17.5   fcntl and ioctl System Calls

Due more to history than intent, several features of the sockets API can be accessed from either ioctl or fcntl. We have already discussed many of the ioctl commands and have mentioned fcntl several times.

Figure 17.15 highlights the functions described in this chapter.
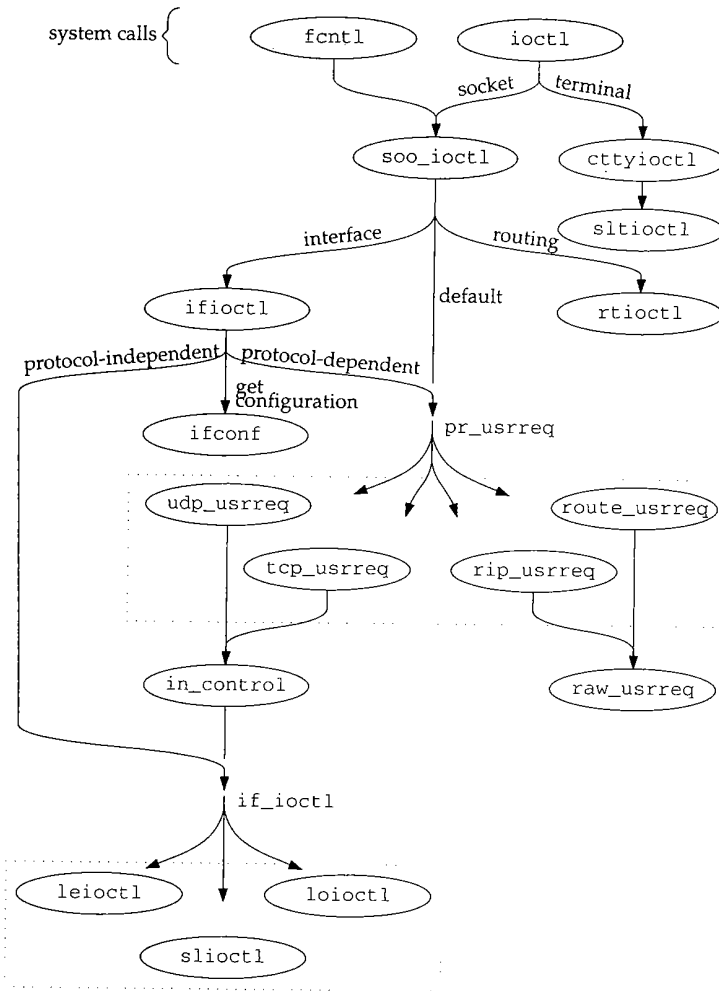
Figure 17.15   fcntl and ioctl functions.

The prototypes for ioctl and fcntl are:

```
int ioctl(int fd, unsigned long result, char *argp);

int fcntl(int fd, int cmd, ... /* int arg */);
```

Figure 17.16 summarizes the features of these two system calls as they relate to sockets. We show the traditional constants in Figure 17.16, since they appear in the code. For Posix compatibility, O_NONBLOCK can be used instead of FNONBLOCK, and O_ASYNC can be used instead of FASYNC.

Description	fcntl	ioctl
enable or disable nonblocking semantics by turning SS_NBIO on or off in so_state	FNONBLOCK file status flag	FIONBIO command
enable or disable asynchronous notification by turning SB_ASYNC on or off in sb_flags	FASYNC file status flag	FIOASYNC command
set or get so_pgid, which is the target process or process group for SIGIO and SIGURG signals	F_SETOWN or F_GETOWN	SIOCSPGRP or SIOCGPGRP commands
get number of bytes in receive buffer; return so_rcv.sb_cc		FIONREAD
return OOB synchronization mark; the SS_RCVATMARK flag in so_state		SIOCATMARK

Figure 17.16   fcntl and ioctl commands.

fcntl Code

Figure 17.17 shows an overview of the fcntl function.

————————————————————————— kern_descrip.c

```
133 struct fcntl_args {
134     int     fd;
135     int     cmd;
136     int     arg;
137 };
138 /* ARGSUSED */
139 fcntl(p, uap, retval)
140 struct proc *p;
141 struct fcntl_args *uap;
142 int     *retval;
143 {
144     struct filedesc *fdp = p->p_fd;
145     struct file *fp;
146     struct vnode *vp;
147     int     i, tmp, error, flg = F_POSIX;
148     struct flock fl;
149     u_int   newmin;

150     if ((unsigned) uap->fd >= fdp->fd_nfiles ||
151         (fp = fdp->fd_ofiles[uap->fd]) == NULL)
152         return (EBADF);
153     switch (uap->cmd) {

                                          /* command processing */


253     default:
254         return (EINVAL);
255     }
256     /* NOTREACHED */
257 }
```

————————————————————————— kern_descrip.c

Figure 17.17   fcntl system call: overview.

133–153   After verifying that the descriptor refers to an open file, the `switch` statement processes the requested command.

253–257   If the command is not recognized, `fcntl` returns `EINVAL`.

Figure 17.18 shows only the cases from `fcntl` that are relevant to sockets.

—— kern_descrip.c

```
168     case F_GETFL:
169          *retval = OFLAGS(fp->f_flag);
170          return (0);

171     case F_SETFL:
172          fp->f_flag &= ~FCNTLFLAGS;
173          fp->f_flag |= FFLAGS(uap->arg) & FCNTLFLAGS;

174          tmp = fp->f_flag & FNONBLOCK;
175          error = (*fp->f_ops->fo_ioctl) (fp, FIONBIO, (caddr_t) & tmp, p);
176          if (error)
177               return (error);

178          tmp = fp->f_flag & FASYNC;
179          error = (*fp->f_ops->fo_ioctl) (fp, FIOASYNC, (caddr_t) & tmp, p);
180          if (!error)
181               return (0);

182          fp->f_flag &= ~FNONBLOCK;
183          tmp = 0;
184          (void) (*fp->f_ops->fo_ioctl) (fp, FIONBIO, (caddr_t) & tmp, p);
185          return (error);

186     case F_GETOWN:
187          if (fp->f_type == DTYPE_SOCKET) {
188               *retval = ((struct socket *) fp->f_data)->so_pgid;
189               return (0);
190          }
191          error = (*fp->f_ops->fo_ioctl)
192               (fp, (int) TIOCGPGRP, (caddr_t) retval, p);
193          *retval = -*retval;
194          return (error);

195     case F_SETOWN:
196          if (fp->f_type == DTYPE_SOCKET) {
197               ((struct socket *) fp->f_data)->so_pgid = uap->arg;
198               return (0);
199          }
200          if (uap->arg <= 0) {
201               uap->arg = -uap->arg;
202          } else {
203               struct proc *p1 = pfind(uap->arg);
204               if (p1 == 0)
205                    return (ESRCH);
206               uap->arg = p1->p_pgrp->pg_id;
207          }
208          return ((*fp->f_ops->fo_ioctl)
209               (fp, (int) TIOCSPGRP, (caddr_t) & uap->arg, p));
```
—— kern_descrip.c

Figure 17.18   `fcntl` system call: socket processing.

168–185   F_GETFL returns the current file status flags associated with the descriptor and F_SETFL sets the flags. The new settings for FNONBLOCK and FASYNC are passed to the associated socket by calling `fo_ioctl`, which for sockets is the `soo_ioctl` function described with Figure 17.20. The third call to `fo_ioctl` is made only if the second call fails. It clears the FNONBLOCK flag, but should instead restore the flag to its original setting.

186–209   F_GETOWN returns so_pgid, the process or process group associated with the socket. For a descriptor other than a socket, the TIOCGPGRP ioctl command is passed to the associated `fo_ioctl` function. F_SETOWN assigns a new value to so_pgid.

For a descriptor other than a socket, the process group is checked in this function, but for sockets, the value is checked just before a signal is sent in sohasoutofband and in sowakeup.

`ioctl` Code

We skip the `ioctl` system call itself and start with `soo_ioctl` in Figure 17.20, since most of the code in `ioctl` duplicates the code we described with Figure 17.17. We've already shown that this function sends routing commands to `rtioctl`, interface commands to `ifioctl`, and any remaining commands to the `pr_usrreq` function of the underlying protocol.

55–68   A few commands are handled by `soo_ioctl` directly. FIONBIO turns on nonblocking semantics if *data is nonzero, and turns them off otherwise. As we have seen, this flag affects the accept, connect, and close system calls as well as the various read and write system calls.

69–79   FIOASYNC enables or disables asynchronous I/O notification. Whenever there is activity on a socket, sowakeup gets called and if SS_ASYNC is set, the SIGIO signal is sent to the process or process group.

80–88   FIONREAD returns the number of bytes available in the receive buffer. SIOCSPGRP sets the process group associated with the socket, and SIOCGPGRP gets it. so_pgid is used as a target for the SIGIO signal as we just described and for the SIGURG signal when out-of-band data arrives for a socket. The signal is sent when the protocol layer calls the sohasoutofband function.

89–92   SIOCATMARK returns true if the socket is at the out-of-band synchronization mark, false otherwise.

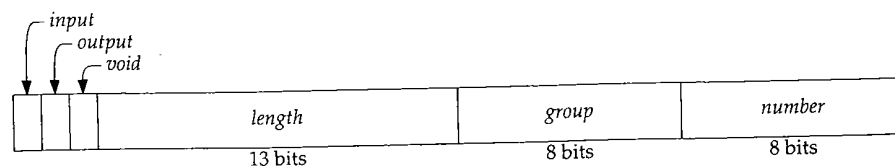ioctl commands, the FIO*xxx* and SIO*xxx* constants, have an internal structure illustrated in Figure 17.19.

Figure 17.19   The structure of an `ioctl` command.

sys_socket.c

```
55  soo_ioctl(fp, cmd, data, p)
56  struct file *fp;
57  int     cmd;
58  caddr_t data;
59  struct proc *p;
60  {
61      struct socket *so = (struct socket *) fp->f_data;
62      switch (cmd) {
63      case FIONBIO:
64          if (*(int *) data)
65              so->so_state |= SS_NBIO;
66          else
67              so->so_state &= ~SS_NBIO;
68          return (0);
69      case FIOASYNC:
70          if (*(int *) data) {
71              so->so_state |= SS_ASYNC;
72              so->so_rcv.sb_flags |= SB_ASYNC;
73              so->so_snd.sb_flags |= SB_ASYNC;
74          } else {
75              so->so_state &= ~SS_ASYNC;
76              so->so_rcv.sb_flags &= ~SB_ASYNC;
77              so->so_snd.sb_flags &= ~SB_ASYNC;
78          }
79          return (0);
80      case FIONREAD:
81          *(int *) data = so->so_rcv.sb_cc;
82          return (0);
83      case SIOCSPGRP:
84          so->so_pgid = *(int *) data;
85          return (0);
86      case SIOCGPGRP:
87          *(int *) data = so->so_pgid;
88          return (0);
89      case SIOCATMARK:
90          *(int *) data = (so->so_state & SS_RCVATMARK) != 0;
91          return (0);
92      }
93      /*
94       * Interface/routing/protocol specific ioctls:
95       * interface and routing ioctls should have a
96       * different entry since a socket's unnecessary
97       */
98      if (IOCGROUP(cmd) == 'i')
99          return (ifioctl(so, cmd, data, p));
100     if (IOCGROUP(cmd) == 'r')
101         return (rtioctl(cmd, data, p));
102     return ((*so->so_proto->pr_usrreq) (so, PRU_CONTROL,
103             (struct mbuf *) cmd, (struct mbuf *) data, (struct mbuf *) 0));
104 }
```

sys_socket.c

Figure 17.20 soo_ioctl function.

If the third argument to ioctl is used as input, *input* is set. If the argument is used as output, *output* is set. If the argument is unused, *void* is set. *length* is the size of the argument in bytes. Related commands are in the same *group* but each command has its own *number* within the group. The macros in Figure 17.21 extract the components of an ioctl command.

Macro	Description
IOCPARM_LEN(cmd)	the *length* from *cmd*
IOCBASECMD(cmd)	the command with *length* set to 0
IOCGROUP(cmd)	the *group* from *cmd*

Figure 17.21   ioctl command macros.

93–104   The macro IOCGROUP extracts the 8-bit *group* from the command. Interface commands are handled by ifioctl. Routing commands are processed by rtioctl. All other commands are passed to the socket's protocol through the PRU_CONTROL request.

> As we describe in Chapter 19, Net/2 introduced a new interface to the routing tables in which messages are passed to the routing subsystem through a socket created in the PF_ROUTE domain. This method replaces the ioctl method shown here. rtioctl always returns ENOTSUPP in kernels that do not have compatibility code compiled in.

17.6   getsockname System Call

The prototype for this system call is:

    int getsockname(int *fd*, caddr_t *asa*, int *alen*);

getsockname retrieves the local address bound to the socket *fd* and places it in the buffer pointed to by *asa*. This is useful when the kernel has selected an address during an implicit bind or when the process specified a wildcard address (Section 22.5) during an explicit call to bind. The getsockname system call is shown in Figure 17.22.

682–715   getsock locates the file structure for the descriptor. The size of the buffer specified by the process is copied from the process into len. This is the first call to m_getclr that we've seen—it allocates a standard mbuf and clears it with bzero. The protocol processing layer is responsible for returning the local address in m when the PRU_SOCKADDR request is issued.

If the address is larger than the buffer specified by the process, it is silently truncated when it is copied out to the process. *alen is updated to the number of bytes copied out to the process. Finally, the mbuf is released and getsockname returns.

17.7   getpeername System Call

The prototype for this system call is:

    int getpeername(int *fd*, caddr_t *asa*, int *alen*);

uipc_syscalls.c

```
682  struct getsockname_args {
683      int      fdes;
684      caddr_t asa;
685      int     *alen;
686  };

687  getsockname(p, uap, retval)
688  struct proc *p;
689  struct getsockname_args *uap;
690  int    *retval;
691  {
692      struct file *fp;
693      struct socket *so;
694      struct mbuf *m;
695      int     len, error;

696      if (error = getsock(p->p_fd, uap->fdes, &fp))
697          return (error);
698      if (error = copyin((caddr_t) uap->alen, (caddr_t) & len, sizeof(len)))
699          return (error);
700      so = (struct socket *) fp->f_data;
701      m = m_getclr(M_WAIT, MT_SONAME);
702      if (m == NULL)
703          return (ENOBUFS);
704      if (error = (*so->so_proto->pr_usrreq) (so, PRU_SOCKADDR, 0, m, 0))
705          goto bad;
706      if (len > m->m_len)
707          len = m->m_len;
708      error = copyout(mtod(m, caddr_t), (caddr_t) uap->asa, (u_int) len);
709      if (error == 0)
710          error = copyout((caddr_t) & len, (caddr_t) uap->alen,
711                              sizeof(len));
712  bad:
713      m_freem(m);
714      return (error);
715  }
```

uipc_syscalls.c

Figure 17.22 getsockname system call.

The getpeername system call returns the address of the remote end of the connection associated with the specified socket. This function is often called when a server is invoked through a fork and exec by the process that calls accept (i.e., any server started by inetd). The server doesn't have access to the peer address returned by accept and must use getpeername. The returned address is often checked against an access list for the application, and the connection is closed if the address is not on the list.

Some protocols, such as TP4, utilize this function to determine if an incoming connection should be rejected or confirmed. In TP4, the connection associated with a socket returned by accept is not yet complete and must be confirmed before the connection completes. Based on the address returned by getpeername, the server can close the connection or implicitly confirm the connection by sending or receiving data. This

feature is irrelevant for TCP, since TCP doesn't make a connection available to accept until the three-way handshake is complete. Figure 17.23 shows the getpeername function.

```
                                                                  ─── uipc_syscalls.c
719 struct getpeername_args {
720     int      fdes;
721     caddr_t asa;
722     int     *alen;
723 };

724 getpeername(p, uap, retval)
725 struct proc *p;
726 struct getpeername_args *uap;
727 int     *retval;
728 {
729     struct file *fp;
730     struct socket *so;
731     struct mbuf *m;
732     int      len, error;

733     if (error = getsock(p->p_fd, uap->fdes, &fp))
734         return (error);
735     so = (struct socket *) fp->f_data;
736     if ((so->so_state & (SS_ISCONNECTED | SS_ISCONFIRMING)) == 0)
737         return (ENOTCONN);
738     if (error = copyin((caddr_t) uap->alen, (caddr_t) & len, sizeof(len)))
739         return (error);
740     m = m_getclr(M_WAIT, MT_SONAME);
741     if (m == NULL)
742         return (ENOBUFS);
743     if (error = (*so->so_proto->pr_usrreq) (so, PRU_PEERADDR, 0, m, 0))
744         goto bad;
745     if (len > m->m_len)
746         len = m->m_len;
747     if (error = copyout(mtod(m, caddr_t), (caddr_t) uap->asa, (u_int) len))
748         goto bad;
749     error = copyout((caddr_t) & len, (caddr_t) uap->alen, sizeof(len));
750 bad:
751     m_freem(m);
752     return (error);
753 }
                                                                  ─── uipc_syscalls.c
```

Figure 17.23   getpeername system call.

719–753   The code here is almost identical to the getsockname code. getsock locates the socket and ENOTCONN is returned if the socket is not yet connected to a peer or if the connection is not in a confirmation state (e.g., TP4). If it is connected, the size of the buffer is copied in from the process and an mbuf is allocated to hold the address. The PRU_PEERADDR request is issued to get the remote address from the protocol layer. The address and the length of the address are copied from the kernel mbuf to the buffer in the process. The mbuf is released and the function returns.

17.8 Summary

In this chapter we discussed the six functions that modify the semantics of a socket. Socket options are processed by setsockopt and getsockopt. Additional options, some of which are not unique to sockets, are handled by fcntl and ioctl. Finally, connection information is available through getsockname and getpeername.

Exercises

17.1 Why do you think options are limited to the size of a standard mbuf (MHLEN, 128 bytes)?

17.2 Why does the code at the end of Figure 17.7 work for the SO_LINGER option?

17.3 There is a problem with the suggested code used to test the timeval structure in Figure 17.9 since tv->tv_sec * hz may cause an overflow. Suggest a change to the code to solve this problem.

18

Radix Tree Routing Tables

18.1 Introduction

The routing performed by IP, when it searches the routing table and decides which interface to send a packet out on, is a *routing mechanism*. This differs from a *routing policy*, which is a set of rules that decides which routes go into the routing table. The Net/3 kernel implements the routing mechanism while a routing daemon, typically routed or gated, implements the routing policy. The structure of the routing table must recognize that the packet forwarding occurs frequently—hundreds or thousands of times a second on a busy system—while routing policy changes are less frequent.

Routing is a detailed issue and we divide our discussion into three chapters.

- This chapter looks at the structure of the radix tree routing tables used by the Net/3 packet forwarding code. The tables are consulted by IP every time a packet is sent (since IP must determine which local interface receives the packet) and every time a packet is forwarded.

- Chapter 19 looks at the functions that interface between the kernel and the radix tree functions, and also at the routing messages that are exchanged between the kernel and routing processes—normally the routing daemons that implement the routing policy. These messages allow a process to modify the kernel's routing table (add a route, delete a route, etc.) and let the kernel notify the daemons when an asynchronous event occurs that might affect the routing policy (a redirect is received, an interface goes down, and so on).

- Chapter 20 presents the routing sockets that are used to exchange routing messages between the kernel and a process.

559

18.2 Routing Table Structure

Before looking at the internal structure of the Net/3 routing table, we need to under-
stand the type of information contained in the table. Figure 18.1 is the bottom half of
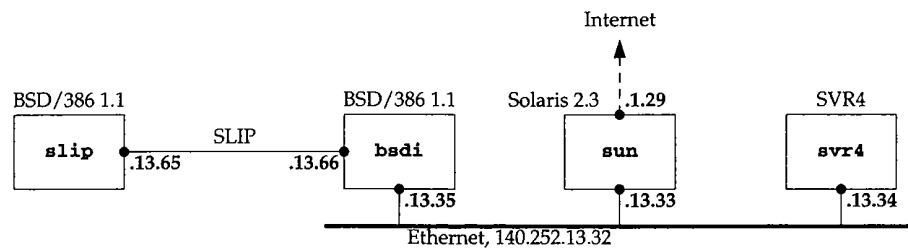Figure 1.17: the four systems on the author's Ethernet.

Figure 18.1   Subnet used for routing table example.

Figure 18.2 shows the routing table for bsdi in Figure 18.1.

```
bsdi $ netstat -rn
Routing tables

Internet:
Destination        Gateway            Flags      Refs      Use  Interface
default            140.252.13.33      UG S          0        3  le0
127                127.0.0.1          UG S   R      0        2  lo0
127.0.0.1          127.0.0.1          U H           1       55  lo0
128.32.33.5        140.252.13.33      UGHS          2       16  le0
140.252.13.32      link#1             U   C         0        0  le0
140.252.13.33      8:0:20:3:f6:42     U H   L      11    55146  le0
140.252.13.34      0:0:c0:c2:9b:26    U H   L       0        3  le0
140.252.13.35      0:0:c0:6f:2d:40    U H   L       1       12  lo0
140.252.13.65      140.252.13.66      U H           0       41  sl0
224                link#1             U   C         0        0  le0
224.0.0.1          link#1             U H   L       0        5  le0
```

Figure 18.2   Routing table on the host bsdi.

We have modified the "Flags" column from the normal netstat output, making it eas-
ier to see which flags are set for the various entries.

The routes in this table were entered as follows. Steps 1, 3, 5, 8, and 9 are performed
at system initialization when the /etc/netstart shell script is executed.

1. A default route is added by the route command to the host sun
    (140.252.13.33), which contains a PPP link to the Internet.

2. The entry for network 127 is typically created by a routing daemon such as
    gated, or it can be entered with the route command in the /etc/netstart
    file. This entry causes all packets sent to this network, other than references to
    the host 127.0.0.1 (which are covered by the more specific route entered in the
    next step), to be rejected by the loopback driver (Figure 5.27).

3. The entry for the loopback interface (127.0.0.1) is configured by `ifconfig`.

4. The entry for `vangogh.cs.berkeley.edu` (128.32.33.5) was created by hand using the `route` command. It specifies the same router as the default route (140.252.13.33), but having a host-specific route, instead of using the default route for this host, allows routing metrics to be stored in this entry. These metrics can optionally be set by the administrator, are used by TCP each time a connection is established to the destination host, and are updated by TCP when the connection is closed. We describe these metrics in more detail with Figure 27.3.

5. The interface `le0` is initialized using the `ifconfig` command. This causes the entry for network 140.252.13.32 to be entered into the routing table.

6. The entries for the other two hosts on the Ethernet, `sun` (140.252.13.33) and `svr4` (140.252.13.34), were created by ARP, as we describe in Chapter 21. These are temporary entries that are removed if they are not used for a certain period of time.

7. The entry for the local host, 140.252.13.35, is created the first time the host's own IP address is referenced. The interface is the loopback, meaning any IP datagrams sent to the host's own IP address are looped back internally. The automatic creation of this entry is new with 4.4BSD, as we describe in Section 21.13.

8. The entry for the host 140.252.13.65 is created when the SLIP interface is configured by `ifconfig`.

9. The `route` command adds the route to network 224 through the Ethernet interface.

10. The entry for the multicast group 224.0.0.1 (the all-hosts group) was created by running the Ping program, pinging the address 224.0.0.1. This is also a temporary entry that is removed if not used for a certain period of time.

The "Flags" column in Figure 18.2 needs a brief explanation. Figure 18.25 provides a list of all the possible flags.

U The route is up.

G The route is to a gateway (router). This is called an *indirect route*. If this flag is not set, the destination is directly connected; this is called a *direct route*.

H The route is to a host, that is, the destination is a complete host address. If this flag is *not* set, the route is to a network, and the destination is a network address: a network ID, or a combination of a network ID and a subnet ID. The `netstat` command doesn't show it, but each network route also contains a network mask. A host route has an implied mask of all one bits.

S The route is static. The three entries created by the `route` command in Figure 18.2 are static.

C The route is cloned to create new routes. Two entries in this routing table have this flag set: (1) the route for the local Ethernet (140.252.13.32), which is cloned by ARP to create the host-specific routes of other hosts on the Ethernet, and (2) the route for multicast groups (224), which is cloned to create specific multicast group routes such as 224.0.0.1

L The route contains a link-layer address. The host routes that ARP clones from the Ethernet network routes all have the link flag set. This applies to unicast and multicast addresses.

R The loopback driver (the normal interface for routes with this flag) rejects all datagrams that use this route.

> The ability to enter a route with the "reject" flag was provided in Net/2. It provides a simple way of preventing datagrams destined to network 127 from appearing outside the host. See also Exercise 6.6.

Before 4.3BSD Reno, two distinct routing tables were maintained by the kernel for IP addresses: one for host routes and one for network routes. A given route was entered into one table or the other, based on the type of route. The default route was stored in the network routing table with a destination address of 0.0.0.0. There was an implied hierarchy: a search was made for a host route first, and if not found a search was made for a network route, and if still not found, a search was made for a default route. Only if all three searches failed was the destination unreachable. Section 11.5 of [Leffler et al. 1989] describes the hash table with linked lists used for the host and network routing tables in Net/1.

Major changes took place in the internal representation of the routing table with 4.3BSD Reno [Sklower 1991]. These changes allow the same routing table functions to access a routing table for other protocol suites, notably the OSI protocols, which use variable-length addresses, unlike the fixed-length 32-bit Internet addresses. The internal structure was also changed, to provide faster lookups.

The Net/3 routing table uses a Patricia tree structure [Sedgewick 1990] to represent both host addresses and network addresses. (Patricia stands for "Practical Algorithm to Retrieve Information Coded in Alphanumeric.") The address being searched for and the addresses in the tree are considered as sequences of bits. This allows the same functions to maintain and search one tree containing fixed-length 32-bit Internet addresses, another tree containing fixed-length 48-bit XNS addresses, and another tree containing variable-length OSI addresses.

> The idea of using Patricia trees for the routing table is attributed to Van Jacobson in [Sklower 1991]. These are actually binary radix tries with one-way branching removed.

An example is the easiest way to describe the algorithm. The goal of routing lookup is to find the most specific address that matches the given destination: the search key. The term *most specific* implies that a host address is preferred over a network address, which is preferred over a default address.

Each entry has an associated network mask, although no mask is stored with a host route; instead host routes have an implied mask of all one bits. An entry in the routing table matches a search key if the search key logically ANDed with the network mask of

the entry equals the entry itself. A given search key might match multiple entries in the routing table, so with a single table for both network route and host routes, the table must be organized so that more-specific routes are considered before less-specific routes.

Consider the examples in Figure 18.3. The two search keys are 127.0.0.1 and 127.0.0.2, which we show in hexadecimal since the logical ANDing is easier to illustrate. The two routing table entries are the host entry for 127.0.0.1 (with an implied mask of `0xffffffff`) and the network entry for 127.0.0.0 (with a mask of `0xff000000`).

		search key = 127.0.0.1		search key = 127.0.0.2	
		host route	net route	host route	net route
1	search key	7f000001	7f000001	7f000002	7f000002
2	routing table key	7f000001	7f000000	7f000001	7f000000
3	routing table mask	ffffffff	ff000000	ffffffff	ff000000
4	logical AND of 1 and 3	7f000001	7f000000	7f000002	7f000000
	2 and 4 equal?	yes	yes	no	yes

Figure 18.3   Example routing table lookups for the two search keys 127.0.0.1 and 127.0.0.2.

Since the search key 127.0.0.1 matches both routing table entries, the routing table must be organized so that the more-specific entry (127.0.0.1) is tried first.

Figure 18.4 shows the internal representation of the Net/3 routing table corresponding to Figure 18.2. This table was built from the output of the netstat command with the -A flag, which dumps the tree structure of the routing tables.
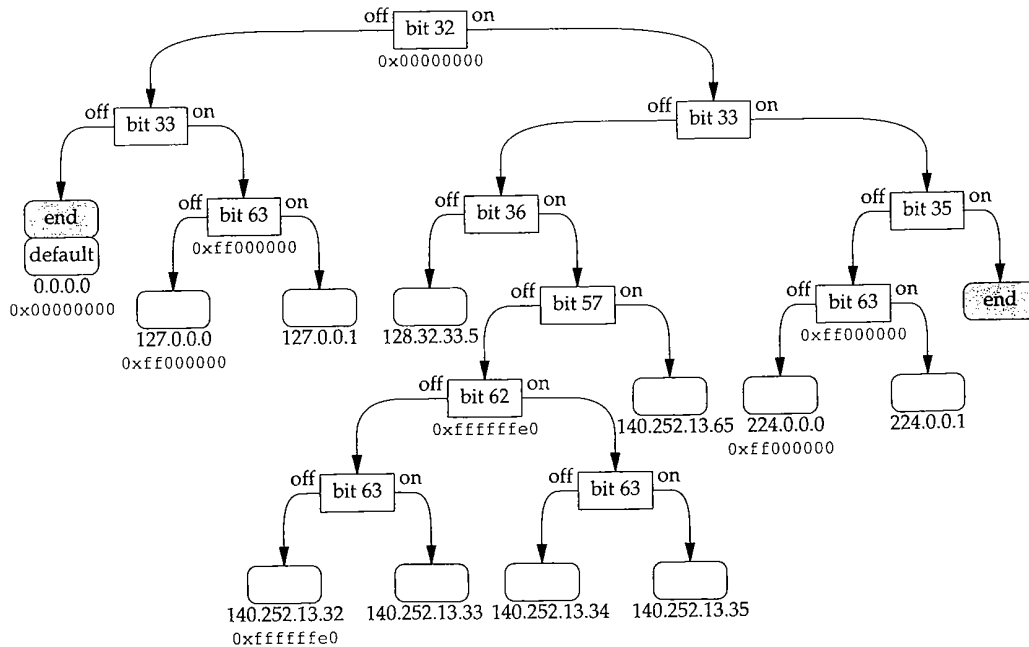

Figure 18.4   Net/3 routing table corresponding to Figure 18.2.

The two shaded boxes labeled "end" are leaves with special flags denoting the end of the tree. The left one has a key of all zero bits and the right one has a key of all one bits. The two boxes stacked together at the left, labeled "end" and "default," are a special representation used for duplicate keys, which we describe in Section 18.9.

The square-cornered boxes are called *internal nodes* or just *nodes*, and the boxes with rounded corners are called *leaves*. Each internal node corresponds to a bit to test in the search key, and a branch is made to the left or the right. Each leaf corresponds to either a host address or a network address. If there is a hexadecimal number beneath a leaf, that leaf is a network address and the number specifies the network mask for the leaf. The absence of a hexadecimal mask beneath a leaf node implies that the leaf is a host address with an implied mask of `0xffffffff`.

Some of the internal nodes also contain network masks, and we'll see how these are used in backtracking. Not shown in this figure is that every node also contains a pointer to its parent, to facilitate backtracking, deletion, and nonrecursive walks of the tree.

The bit comparisons are performed on socket address structures, so the bit positions given in Figure 18.4 are from the start of the socket address structure. Figure 18.5 shows the bit positions for a `sockaddr_in` structure.
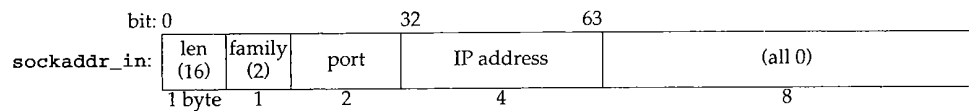

Figure 18.5   Bit offsets in Internet socket address structure.

The highest-order bit of the IP address is at bit position 32 and the lowest-order bit is at bit position 63. We also show the length as 16 and the address family as 2 (`AF_INET`), as we'll encounter these two values throughout our examples.

To work through the examples we also need to show the bit representations of the various IP addresses in the tree. These are shown in Figure 18.6 along with some other IP addresses that are used in the examples that follow. The bit positions used in Figure 18.4 as branching points are shown in a bolder font.

We now provide some specific examples of how the routing table searches are performed.

Example—Host Match

Assume the host address 127.0.0.1 is the search key—the destination address being looked up. Bit 32 is off, so the left branch is made from the top of the tree. Bit 33 is on, so the right branch is made from the next node. Bit 63 is on, so the right branch is made from the next node. This next node is a leaf, so the search key (127.0.0.1) is compared to the address in the leaf (127.0.0.1). They match exactly so this routing table entry is returned by the lookup function.

	32-bit IP address (bits 32–63)							dotted-decimal
bit:	**3333** **2345**	**3333** **6789**	**4444** **0123**	**4444** **4567**	**4455** **8901**	**5555** **2345**	**5555** **6789**	**6666** **0123**	
	0000	1010	0000	0001	0000	0010	0000	0011	10.1.2.3
	0111	0000	0000	0000	0000	0000	0000	0001	112.0.0.1
	0111	1111	0000	0000	0000	0000	0000	0000	127.0.0.0
	0111	1111	0000	0000	0000	0000	0000	0001	127.0.0.1
	0111	1111	0000	0000	0000	0000	0000	0011	127.0.0.3
	1000	0000	0010	0000	0010	0001	0000	0101	128.32.33.5
	1000	0000	0010	0000	0010	0001	0000	0110	128.32.33.6
	1000	1100	1111	1100	0000	1101	0010	0000	140.252.13.32
	1000	1100	1111	1100	0000	1101	0010	0001	140.252.13.33
	1000	1100	1111	1100	0000	1101	0010	0010	140.252.13.34
	1000	1100	1111	1100	0000	1101	0010	0011	140.252.13.35
	1000	1100	1111	1100	0000	1101	0100	0001	140.252.13.65
	1110	0000	0000	0000	0000	0000	0000	0000	224.0.0.0
	1110	0000	0000	0000	0000	0000	0000	0001	224.0.0.1

Figure 18.6   Bit representations of the IP addresses in Figures 18.2 and 18.4.

Example—Host Match

Next assume the search key is the address 140.252.13.35. Bit 32 is on, so the right branch is made from the top of the tree. Bit 33 is off, bit 36 is on, bit 57 is off, bit 62 is on, and bit 63 is on, so the search ends at the leaf on the bottom labeled 140.252.13.35. The search key matches the routing table key exactly.

Example—Network Match

The search key is 127.0.0.2. Bit 32 is off, bit 33 is on, and bit 63 is off so the search ends up at the leaf labeled 127.0.0.0. The search key and the routing table key don't match exactly, so a network match is tried. The search key is logically ANDed with the network mask ($0xff000000$) and since the result equals the routing table key, this entry is considered a match.

Example—Default Match

The search key is 10.1.2.3. Bit 32 is off and bit 33 is off, so the search ends up at the leaf with the duplicate keys labeled "end" and "default." The routing table key that is duplicated in these two leaves is 0.0.0.0. The search key and the routing table key don't match exactly, so a network match is tried. This match is tried for all duplicate keys that have a network mask. The first key (the end marker) doesn't have a network mask, so it is skipped. The next key (the default entry) has a mask of $0x00000000$. The search key is logically ANDed with this mask and since the result equals the routing table key (0), this entry is considered a match. The default route is used.

Example—Network Match with Backtracking

The search key is 127.0.0.3. Bit 32 is off, bit 33 is on, and bit 63 is on, so the search ends up at the leaf labeled 127.0.0.1. The search key and the routing table key don't match exactly. A network match cannot be attempted since this leaf does not have a network mask. Backtracking now takes place.

The backtracking algorithm is to move up the tree, one level at a time. If an internal node is encountered that contains a mask, the search key is logically ANDed with the mask and another search is made of the subtree starting at the node with the mask, looking for a match with the ANDed key. If a match isn't found, the backtrack keeps moving up the tree, until the top is reached.

In this example the search moves up one level to the node for bit 63 and this node contains a mask. The search key is logically ANDed with the mask (0xff000000), giving a new search key of 127.0.0.0. Another search is made starting at this node for 127.0.0.0. Bit 63 is off, so the left branch is taken to the leaf labeled 127.0.0.0. The new search key is compared to the routing table key and since they're equal, this leaf is the match.

Example—Backtracking Multiple Levels

The search key is 112.0.0.1. Bit 32 is off, bit 33 is on, and bit 63 is on, so the search ends up at the leaf labeled 127.0.0.1. The keys are not equal and the routing table entry does not have a network mask, so backtracking takes place.

The search moves up one level to the node for bit 63, which contains a mask. The search key is logically ANDed with the mask of 0xff000000 and another search is made starting at that node. Bit 63 is off in the new search key, so the left branch is made to the leaf labeled 127.0.0.0. A comparison is made but the ANDed search key (112.0.0.0) doesn't equal the search key in the table.

Backtracking continues up one level from the bit-63 node to the bit-33 node. But this node does not have a mask, so the backtracking continues upward. The next level is the top of the tree (bit 32) and it has a mask. The search key (112.0.0.1) is logically ANDed with the mask (0x00000000) and a new search started from that point. Bit 32 is off in the new search key, as is bit 33, so the search ends up at the leaf labeled "end" and "default." The list of duplicate keys is traversed and the default key matches the new search key, so the default route is used.

As we can see in this example, if a default route is present in the routing table, when the backtrack ends up at the top node in the tree, its mask is all zero bits, which causes the search to proceed to the leftmost leaf in the tree for a match with the default.

Example—Host Match with Backtracking and Cloning

The search key is 224.0.0.5. Bit 32 is on, bit 33 is on, bit 35 is off, and bit 63 is on, so the search ends up at the leaf labeled 224.0.0.1. This routing table key does not equal the search key, and the routing table entry does not contain a network mask, so backtracking takes place.

The backtrack moves one level up to the node that tests bit 63. This node contains the mask 0xff000000, so the search key ANDed with the mask yields a new search key of 224.0.0.0. Another search is made, starting at this node. Since bit 63 is off in the ANDed key, the left branch is taken to the leaf labeled 224.0.0.0. This routing table key matches the ANDed search key, so this entry is a match.

This route has the "clone" flag set (Figure 18.2), so a new leaf is created for the address 224.0.0.5. The new routing table entry is

```
Destination      Gateway          Flags     Refs     Use  Interface
224.0.0.5        link#1           UHL          0       0  le0
```

and Figure 18.7 shows the new arrangement of the right side of the routing table tree from Figure 18.4, starting with the node for bit 35. Notice that whenever a new leaf is added to the tree, two nodes are needed: one for the leaf and one for the internal node specifying the bit to test.
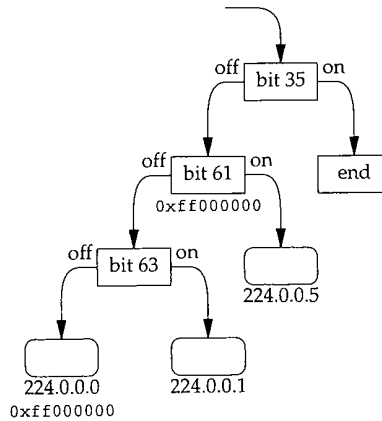

Figure 18.7   Modification of Figure 18.4 after inserting entry for 224.0.0.5.

This newly created entry is the one returned to the caller who was searching for 224.0.0.5.

The Big Picture

Figure 18.8 shows a bigger picture of all the data structures involved. The bottom portion of this figure is from Figure 3.32.

There are numerous points about this figure that we'll note now and describe in detail later in this chapter.

- rt_tables is an array of pointers to radix_node_head structures. There is one entry in the array for each address family. rt_tables[AF_INET] points to the top of the Internet routing table tree.
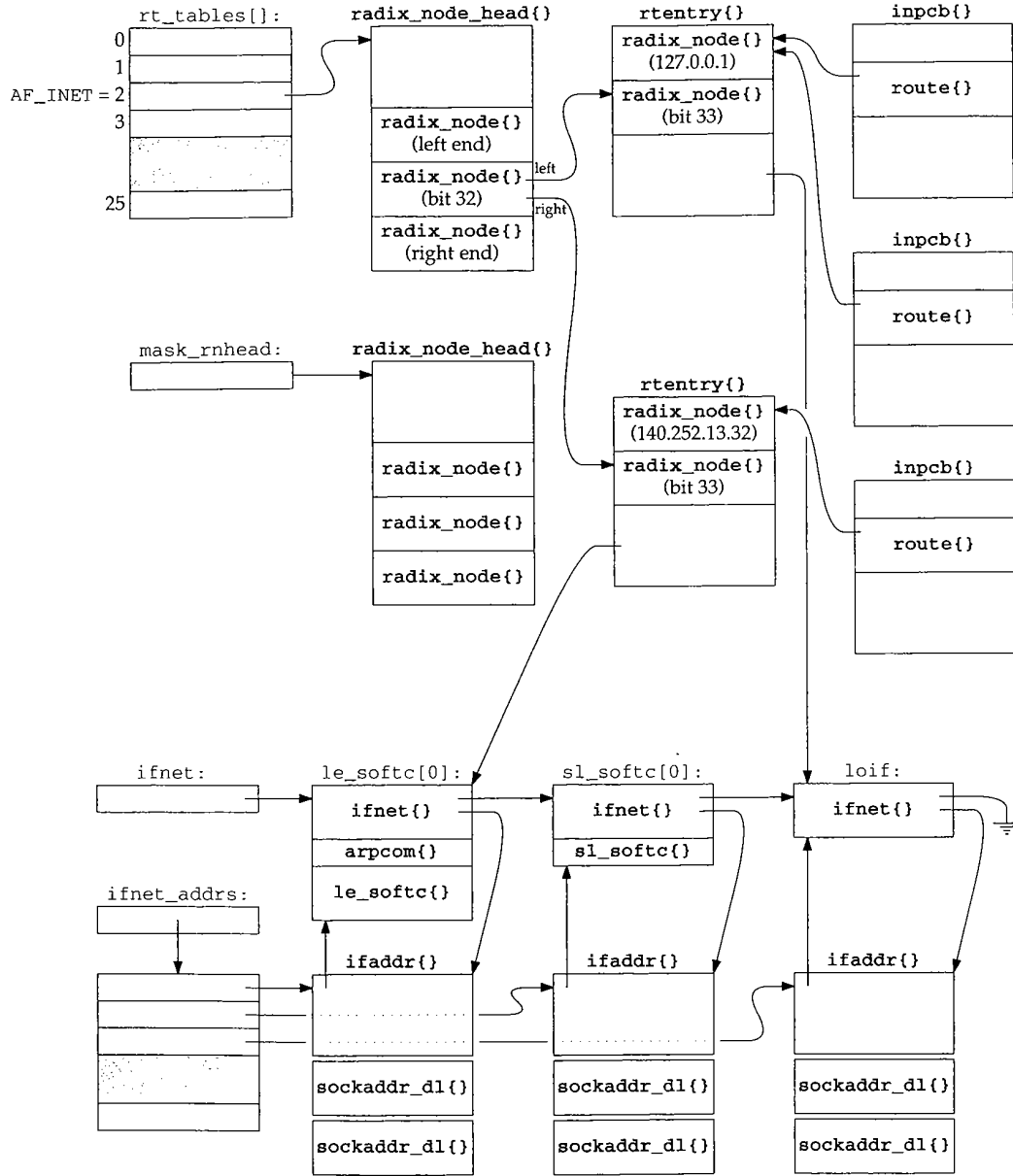
Figure 18.8   Data structures involved with routing tables.

- The `radix_node_head` structure contains three `radix_node` structures. These structures are built when the tree is initialized and the middle of the three is the top of the tree. This corresponds to the top box in Figure 18.4, labeled "bit 32." The first of the three `radix_node` structures is the leftmost leaf in Figure 18.4 (the shared duplicate with the default route) and the third of the three is the rightmost leaf. An empty routing table consists of just these three `radix_node` structures; we'll see how it is constructed by the `rn_inithead` function.

- The global `mask_rnhead` also points to a `radix_node_head` structure. This is the head of a separate tree of all the masks. Notice in Figure 18.4 that of the eight masks shown, one is duplicated four times and two are duplicated once. By keeping a separate tree for the masks, only one copy of each unique mask is maintained.

- The routing table tree is built from `rtentry` structures, and we show two of these in Figure 18.8. Each `rtentry` structure contains two `radix_node` structures, because each time a new entry is inserted into the tree, two nodes are required: an internal node corresponding to a bit to be tested, and a leaf node corresponding to a host route or a network route. In each `rtentry` structure we also show which bit test the internal node corresponds to and the address contained in the leaf node.

  The remainder of the `rtentry` structure is the focal point of information for this route. We show only a single pointer from this structure to the corresponding `ifnet` structure for the route, but this structure also contains a pointer to the `ifaddr` structure, the flags for the route, a pointer to another `rtentry` structure if this entry is an indirect route, the metrics for the route, and so on.

- Protocol control blocks (Chapter 22), of which one exists for each UDP and TCP socket (Figure 22.1), contain a `route` structure that points to an `rtentry` structure. The UDP and TCP output functions both pass a pointer to the `route` structure in a PCB as the third argument to `ip_output`, each time an IP datagram is sent. PCBs that use the same route point to the same routing table entry.

18.3 Routing Sockets

When the routing table changes were made with 4.3BSD Reno, the interaction of processes with the routing subsystem also changed—the concept of routing sockets was introduced. Prior to 4.3BSD Reno, fixed-length `ioctl`s were issued by a process (such as the `route` command) to modify the routing table. 4.3BSD Reno changed this to a more generalized message-passing scheme using the new `PF_ROUTE` domain. A process creates a raw socket in the `PF_ROUTE` domain and can send routing messages to the kernel, and receives routing messages from the kernel (e.g., redirects and other asynchronous notifications from the kernel).

Figure 18.9 shows the 12 different types of routing messages. The message type is the `rtm_type` field in the `rt_msghdr` structure, which we describe in Figure 19.16. Only five of the messages can be issued by a process (a write to a routing socket), but all 12 can be received by a process.

We'll defer our discussion of these routing messages until Chapter 19.

rtm_type	To kernel?	From kernel?	Description	Structure type
RTM_ADD	•	•	add route	rt_msghdr
RTM_CHANGE	•	•	change gateway, metrics, or flags	rt_msghdr
RTM_DELADDR		•	address being removed from interface	ifa_msghdr
RTM_DELETE	•	•	delete route	rt_msghdr
RTM_GET	•	•	report metrics and other route information	rt_msghdr
RTM_IFINFO		•	interface going up, down, etc.	if_msghdr
RTM_LOCK	•	•	lock specified metrics	rt_msghdr
RTM_LOSING		•	kernel suspects route is failing	rt_msghdr
RTM_MISS		•	lookup failed on this address	rt_msghdr
RTM_NEWADDR		•	address being added to interface	ifa_msghdr
RTM_REDIRECT		•	kernel told to use different route	rt_msghdr
RTM_RESOLVE		•	request to resolve destination to link-layer address	rt_msghdr

Figure 18.9   Types of messages exchanged across a routing socket.

18.4   Code Introduction

Three headers and five C files define the various structures and functions used for routing. These are summarized in Figure 18.10.

File	Description
net/radix.h	radix node definitions
net/raw_cb.h	routing control block definitions
net/route.h	routing structures
net/radix.c	radix node (Patricia tree) functions
net/raw_cb.c	routing control block functions
net/raw_usrreq.c	routing control block functions
net/route.c	routing functions
net/rtsock.c	routing socket functions

Figure 18.10   Files discussed in this chapter.

In general, the prefix rn_ denotes the radix node functions that search and manipulate the Patricia trees, the raw_ prefix denotes the routing control block functions, and the three prefixes route_, rt_, and rt denote the general routing functions.

> We use the term *routing control blocks* instead of *raw control blocks* in all the routing chapters, even though the files and functions begin with the prefix raw. This is to avoid confusion with the raw IP control blocks and functions, which we discuss in Chapter 32. Although the raw control blocks and their associated functions are used for more than just routing sockets in Net/3 (one of the raw OSI protocols uses these structures and functions), our use in this text is only with routing sockets in the PF_ROUTE domain.

Figure 18.11 shows the primary routing functions and their relationships. The shaded ellipses are the ones we cover in this chapter and the next two. We also show where each of the 12 routing message types are generated.
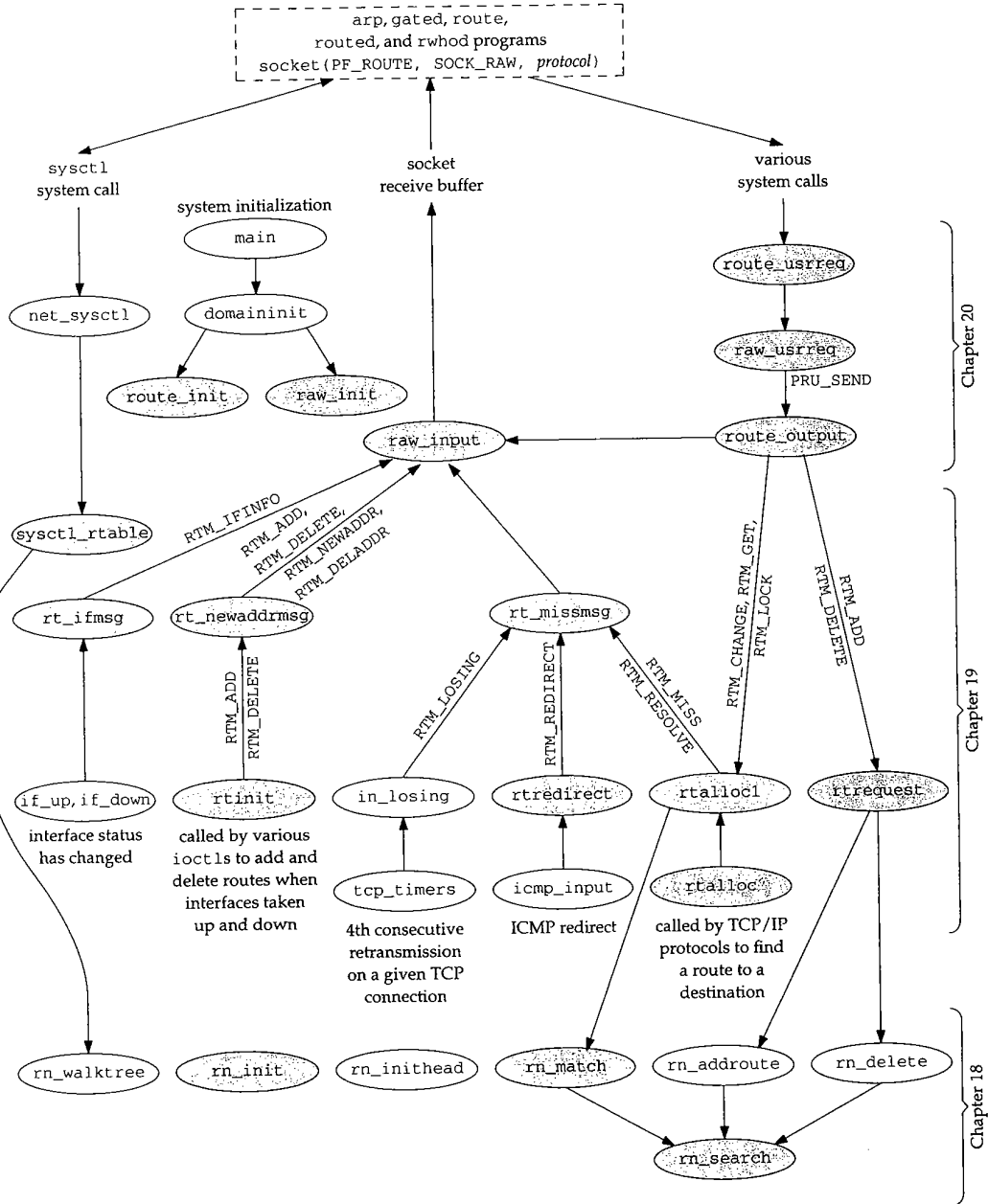
Figure 18.11   Relationships between the various routing functions.

rtalloc is the function called by the Internet protocols to look up routes to destinations. We've already encountered rtalloc in the ip_rtaddr, ip_forward, ip_output, and ip_setmoptions functions. We'll also encounter it later in the in_pcbconnect and tcp_mss functions.

We also show in Figure 18.11 that five programs typically create sockets in the routing domain:

- arp manipulates the ARP cache, which is stored in the IP routing table in Net/3 (Chapter 21),
- gated and routed are routing daemons that communicate with other routers and manipulate the kernel's routing table as the routing environment changes (routers and links go up or down),
- route is a program typically executed by start-up scripts or by the system administrator to add or delete routes, and
- rwhod issues a routing sysctl on start-up to determine the attached interfaces.

Naturally, any process (with superuser privilege) can open a routing socket to send and receive messages to and from the routing subsystem; we show only the common system programs in Figure 18.11.

Global Variables

The global variables introduced in the three routing chapters are shown in Figure 18.12.

Variable	Datatype	Description
rt_tables mask_rnhead rn_mkfreelist	struct radix_node_head * [] struct radix_node_head * struct radix_mask *	array of pointers to heads of routing tables pointer to head of mask table head of linked list of available radix_mask structures
max_keylen rn_zeros rn_ones maskedKey	int char * char * char *	longest routing table key, in bytes array of all zero bits, of length max_keylen array of all one bits, of length max_keylen array for masked search key, of length max_keylen
rtstat rttrash	struct rtstat int	routing statistics (Figure 18.13) #routes not in table but not freed
rawcb raw_recvspace raw_sendspace	struct rawcb u_long u_long	head of doubly linked list of routing control blocks default size of routing socket receive buffer, 8192 bytes default size of routing socket send buffer, 8192 bytes
route_cb route_dst route_src route_proto	struct route_cb struct sockaddr struct sockaddr struct sockproto	#routing socket listeners, per protocol, and total temporary for destination of routing message temporary for source of routing message temporary for protocol of routing message

Figure 18.12   Global variables in the three routing chapters.

Statistics

Some routing statistics are maintained in the global structure `rtstat`, described in Figure 18.13.

`rtstat` member	Description	Used by SNMP
`rts_badredirect`	#invalid redirect calls	
`rts_dynamic`	#routes created by redirects	
`rts_newgateway`	#routes modified by redirects	
`rts_unreach`	#lookups that failed	
`rts_wildcard`	#lookups matched by wildcard (never used)	

Figure 18.13 Routing statistics maintained in the `rtstat` structure.

We'll see where these counters are incremented as we proceed through the code. None are used by SNMP.

Figure 18.14 shows some sample output of these statistics from the `netstat -rs` command, which displays this structure.

`netstat -rs` output	`rtstat` member
1029 bad routing redirects	`rts_badredirect`
0 dynamically created routes	`rts_dynamic`
0 new gateways due to redirects	`rts_newgateway`
0 destinations found unreachable	`rts_unreach`
0 uses of a wildcard route	`rts_wildcard`

Figure 18.14 Sample routing statistics.

SNMP Variables

Figure 18.15 shows the IP routing table, named `ipRouteTable`, and the kernel variables that supply the corresponding value.

For `ipRouteType`, if the `RTF_GATEWAY` flag is set in `rt_flags`, the route is remote (4); otherwise the route is direct (3). For `ipRouteProto`, if either the `RTF_DYNAMIC` or `RTF_MODIFIED` flag is set, the route was created or modified by ICMP (4), otherwise the value is other (1). Finally, if the `rt_mask` pointer is null, the returned mask is all one bits (i.e., a host route).

18.5 Radix Node Data Structures

In Figure 18.8 we see that the head of each routing table is a `radix_node_head` and all the nodes in the routing tree, both the internal nodes and the leaves, are `radix_node` structures. The `radix_node_head` structure is shown in Figure 18.16.

IP routing table, index = < *ipRouteDest* >		
SNMP variable	Variable	Description
ipRouteDest	rt_key	Destination IP address. A value of 0.0.0.0 indicates a default entry.
ipRouteIfIndex	rt_ifp.if_index	Interface number: ifIndex.
ipRouteMetric1	−1	Primary routing metric. The meaning of the metric depends on the routing protocol (ipRouteProto). A value of −1 means it is not used.
ipRouteMetric2	−1	Alternative routing metric.
ipRouteMetric3	−1	Alternative routing metric.
ipRouteMetric4	−1	Alternative routing metric.
ipRouteNextHop	rt_gateway	IP address of next-hop router.
ipRouteType	(see text)	Route type: 1 = other, 2 = invalidated route, 3 = direct, 4 = indirect.
ipRouteProto	(see text)	Routing protocol: 1 = other, 4 = ICMP redirect, 8 = RIP, 13 = OSPF, 14 = BGP, and others.
ipRouteAge	(not implemented)	Number of seconds since route was last updated or determined to be correct.
ipRouteMask	rt_mask	Mask to be logically ANDed with destination IP address before being compared with ipRouteDest.
ipRouteMetric5	−1	Alternative routing metric.
ipRouteInfo	NULL	Reference to MIB definitions specific to this particular routing protocol.

Figure 18.15   IP routing table: ipRouteTable.

——— radix.h

```
 91 struct radix_node_head {
 92     struct radix_node *rnh_treetop;
 93     int     rnh_addrsize;          /* (not currently used) */
 94     int     rnh_pktsize;           /* (not currently used) */
 95     struct radix_node *(*rnh_addaddr)   /* add based on sockaddr */
 96             (void *v, void *mask,
 97              struct radix_node_head * head, struct radix_node nodes[]);
 98     struct radix_node *(*rnh_addpkt)    /* add based on packet hdr */
 99             (void *v, void *mask,
100              struct radix_node_head * head, struct radix_node nodes[]);
101     struct radix_node *(*rnh_deladdr)   /* remove based on sockaddr */
102             (void *v, void *mask, struct radix_node_head * head);
103     struct radix_node *(*rnh_delpkt)    /* remove based on packet hdr */
104             (void *v, void *mask, struct radix_node_head * head);
105     struct radix_node *(*rnh_matchaddr)      /* locate based on sockaddr */
106             (void *v, struct radix_node_head * head);
107     struct radix_node *(*rnh_matchpkt)  /* locate based on packet hdr */
108             (void *v, struct radix_node_head * head);
109     int     (*rnh_walktree)      /* traverse tree */
110             (struct radix_node_head * head, int (*f) (), void *w);

111     struct radix_node rnh_nodes[3];      /* top and end nodes */
112 };
```
——— radix.h

Figure 18.16   radix_node_head structure: the top of each routing tree.

92   rnh_treetop points to the top radix_node structure for the routing tree. Notice that three of these structures are allocated at the end of the radix_node_head, and the middle one of these is initialized as the top of the tree (Figure 18.8).

93-94   rnh_addrsize and rnh_pktsize are not currently used.

> rnh_addrsize is to facilitate porting the routing table code to systems that don't have a length byte in the socket address structure. rnh_pktsize is to allow using the radix node machinery to examine addresses in packet headers without having to copy the address into a socket address structure.

95-110   The seven function pointers, rnh_addaddr through rnh_walktree, point to functions that are called to operate on the tree. Only four of these pointers are initialized by rn_inithead and the other three are never used by Net/3, as shown in Figure 18.17.

Member	Initialized to (by rn_inithead)
rnh_addaddr	*rn_addroute*
rnh_addpkt	*NULL*
rnh_deladdr	*rn_delete*
rnh_delpkt	*NULL*
rnh_matchaddr	*rn_match*
rnh_matchpkt	*NULL*
rnh_walktree	*rn_walktree*

Figure 18.17   The seven function pointers in the radix_node_head structure.

111-112   Figure 18.18 shows the radix_node structure that forms the nodes of the tree. In Figure 18.8 we see that three of these are allocated in the radix_node_head and two are allocated in each rtentry structure.

―― *radix.h*
```
40 struct radix_node {
41     struct radix_mask *rn_mklist;   /* list of masks contained in subtree */
42     struct radix_node *rn_p;      /* parent pointer */
43     short    rn_b;               /* bit offset; -1-index(netmask) */
44     char     rn_bmask;           /* node: mask for bit test */
45     u_char   rn_flags;           /* Figure 18.20 */
46     union {
47         struct {                 /* leaf only data: rn_b < 0 */
48             caddr_t rn_Key;      /* object of search */
49             caddr_t rn_Mask;     /* netmask, if present */
50             struct radix_node *rn_Dupedkey;
51         } rn_leaf;
52         struct {                 /* node only data: rn_b >= 0 */
53             int      rn_Off;     /* where to start compare */
54             struct radix_node *rn_L;    /* left pointer */
55             struct radix_node *rn_R;    /* right pointer */
56         } rn_node;
57     } rn_u;
58 };

59 #define rn_dupedkey rn_u.rn_leaf.rn_Dupedkey
60 #define rn_key      rn_u.rn_leaf.rn_Key
```

```
61 #define rn_mask      rn_u.rn_leaf.rn_Mask
62 #define rn_off       rn_u.rn_node.rn_Off
63 #define rn_l         rn_u.rn_node.rn_L
64 #define rn_r         rn_u.rn_node.rn_R
```
—— *radix.h*

Figure 18.18 `radix_node` structure: the nodes of the routing tree.

41–45   The first five members are common to both internal nodes and leaves, followed by a union defining three members if the node is a leaf, or a different three members if the node is internal. As is common throughout the Net/3 code, a set of `#define` statements provide shorthand names for the members in the union.

41–42   `rn_mklist` is the head of a linked list of masks for this node. We describe this field in Section 18.9. `rn_p` points to the parent node.

43   If `rn_b` is greater than or equal to 0, the node is an internal node, else the node is a leaf. For the internal nodes, `rn_b` is the bit number to test: for example, its value is 32 in the top node of the tree in Figure 18.4. For leaves, `rn_b` is negative and its value is –1 minus the *index of the network mask*. This index is the first bit number where a 0 occurs. Figure 18.19 shows the indexes of the masks from Figure 18.4.

	32-bit IP mask (bits 32–63)								index	rn_b
	3333 2345	3333 6789	4444 0123	4444 4567	4455 8901	5555 2345	5555 6789	6666 0123		
00000000:	0000	0000	0000	0000	0000	0000	0000	0000	0	–1
ff000000:	1111	1111	0000	0000	0000	0000	0000	0000	40	–41
fffffe0:	1111	1111	1111	1111	1111	1111	1110	0000	59	–60

Figure 18.19 Example of mask indexes.

As we can see, the index of the all-zero mask is handled specially: its index is 0, not 32.

44   `rn_bmask` is a 1-byte mask used with the internal nodes to test whether the corresponding bit is on or off. Its value is 0 in leaves. We'll see how this member is used with the `rn_off` member shortly.

45   Figure 18.20 shows the three values for the `rn_flags` member.

Constant	Description
RNF_ACTIVE	this node is alive (for `rtfree`)
RNF_NORMAL	leaf contains normal route (not currently used)
RNF_ROOT	node is in the `radix_node_head` structure

Figure 18.20 `rn_flags` values.

The `RNF_ROOT` flag is set only for the three radix nodes in the `radix_node_head` structure: the top of the tree and the left and right end nodes. These three nodes can never be deleted from the routing tree.

48-49   For a leaf, `rn_key` points to the socket address structure and `rn_mask` points to a socket address structure containing the mask. If `rn_mask` is null, the implied mask is all one bits (i.e., this route is to a host, not to a network).

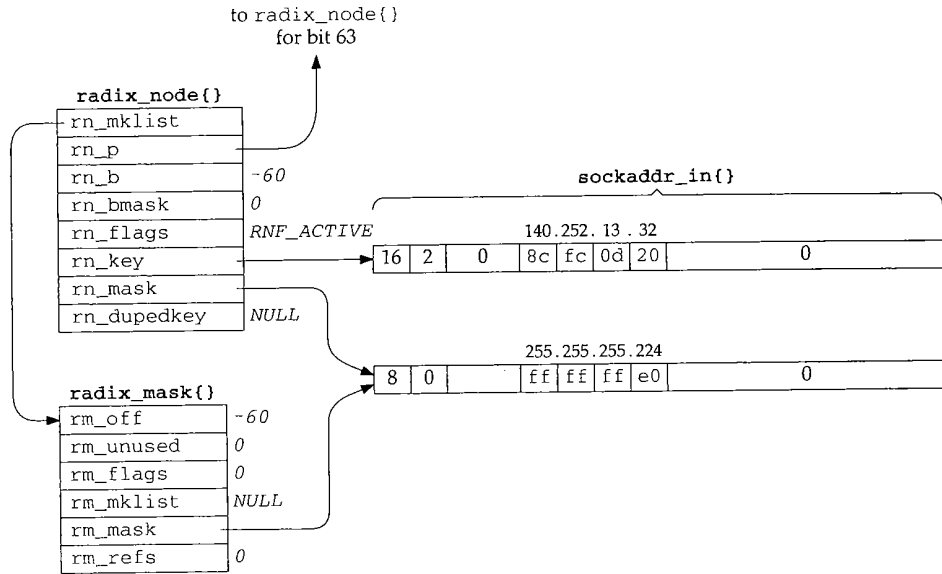Figure 18.21 shows an example corresponding to the leaf for 140.252.13.32 in Figure 18.4.

Figure 18.21   `radix_node` structure corresponding to leaf for 140.252.13.32 in Figure 18.4.

This example also shows a `radix_mask` structure, which we describe in Figure 18.22. We draw this latter structure with a smaller width, to help distinguish it as a different structure from the `radix_node`; we'll encounter both structures in many of the figures that follow. We describe the reason for the `radix_mask` structure in Section 18.9.

The `rn_b` of −60 corresponds to an index of 59. `rn_key` points to a `sockaddr_in`, with a length of 16 and an address family of 2 (`AF_INET`). The mask structure pointed to by `rn_mask` and `rm_mask` has a length of 8 and a family of 0 (this family is `AF_UNSPEC`, but it is never even looked at).

50-51   The `rn_dupedkey` pointer is used when there are multiple leaves with the same key. We describe these in Section 18.9.

52-58   We describe `rn_off` in Section 18.8. `rn_l` and `rn_r` are the left and right pointers for the internal node.

Figure 18.22 shows the `radix_mask` structure.

—— *radix.h*
```
76 extern struct radix_mask {
77     short   rm_b;                /* bit offset; -1-index(netmask) */
78     char    rm_unused;           /* cf. rn_bmask */
79     u_char  rm_flags;            /* cf. rn_flags */
80     struct radix_mask *rm_mklist;   /* more masks to try */
81     caddr_t rm_mask;             /* the mask */
82     int     rm_refs;             /* # of references to this struct */
83 }       *rn_mkfreelist;
```
—— *radix.h*

Figure 18.22   radix_mask structure.

76–83   Each of these structures contains a pointer to a mask: rm_mask, which is really a
pointer to a socket address structure containing the mask. Each radix_node structure
points to a linked list of radix_mask structures, allowing multiple masks per node:
rn_mklist points to the first, and then each rm_mklist points to the next. This structure definition also declares the global rn_mkfreelist, which is the head of a linked
list of available structures.

18.6 Routing Structures

The focal points of access to the kernel's routing information are

1. the rtalloc function, which searches for a route to a destination,

2. the route structure that is filled in by this function, and

3. the rtentry structure that is pointed to by the route structure.

Figure 18.8 showed that the protocol control blocks (PCBs) used by UDP and TCP
(Chapter 22) contain a route structure, which we show in Figure 18.23.

—— *route.h*
```
46 struct route {
47     struct rtentry *ro_rt;       /* pointer to struct with information */
48     struct sockaddr ro_dst;      /* destination of this route */
49 };
```
—— *route.h*

Figure 18.23   route structure.

ro_dst is declared as a generic socket address structure, but for the Internet protocols
it is a sockaddr_in. Notice that unlike most references to this type of structure,
ro_dst is the structure itself, not a pointer to one.

At this point it is worth reviewing Figure 8.24, which shows the use of these routes
every time an IP datagram is output.

• If the caller passes a pointer to a route structure, that structure is used. Otherwise a local route structure is used and it is set to 0, setting ro_rt to a null
  pointer. UDP and TCP pass a pointer to the route structure in their PCB to
  ip_output.

- If the `route` structure points to an `rtentry` structure (the `ro_rt` pointer is nonnull), and if the referenced interface is still up, and if the destination address in the `route` structure equals the destination address of the IP datagram, that route is used. Otherwise the socket address structure `ro_dst` is filled in with the destination IP address and `rtalloc` is called to locate a route to that destination. For a TCP connection the destination address of the datagram never changes from the destination address of the route, but a UDP application can send a datagram to a different destination with each `sendto`.

- If `rtalloc` returns a null pointer in `ro_rt`, a route was not found and `ip_output` returns an error.

- If the `RTF_GATEWAY` flag is set in the `rtentry` structure, the route is indirect (the G flag in Figure 18.2). The destination address (`dst`) for the interface output function becomes the IP address of the gateway, the `rt_gateway` member, not the destination address of the IP datagram.

Figure 18.24 shows the `rtentry` structure.

```
                                                              route.h
83 struct rtentry {
84     struct radix_node rt_nodes[2];   /* a leaf and an internal node */

85     struct sockaddr *rt_gateway;      /* value associated with rn_key */
86     short   rt_flags;                 /* Figure 18.25 */
87     short   rt_refcnt;                /* #held references */
88     u_long  rt_use;                   /* raw #packets sent */
89     struct ifnet *rt_ifp;             /* interface to use */
90     struct ifaddr *rt_ifa;            /* interface address to use */
91     struct sockaddr *rt_genmask;      /* for generation of cloned routes */
92     caddr_t rt_llinfo;                /* pointer to link level info cache */
93     struct rt_metrics rt_rmx;         /* metrics: Figure 18.26 */
94     struct rtentry *rt_gwroute;       /* implied entry for gatewayed routes */
95 };

96 #define rt_key(r)    ((struct sockaddr *)((r)->rt_nodes->rn_key))
97 #define rt_mask(r)   ((struct sockaddr *)((r)->rt_nodes->rn_mask))
                                                              route.h
```

Figure 18.24   rtentry structure.

83–84   Two `radix_node` structures are contained within this structure. As we noted in the example with Figure 18.7, each time a new leaf is added to the routing tree a new internal node is also added. `rt_nodes[0]` contains the leaf entry and `rt_nodes[1]` contains the internal node. The two `#define` statements at the end of Figure 18.24 provide a shorthand access to the key and mask of this leaf node.

86   Figure 18.25 shows the various constants stored in `rt_flags` and the corresponding character output by `netstat` in the "Flags" column (Figure 18.2).

The `RTF_BLACKHOLE` flag is not output by `netstat` and the two with lowercase flag characters, `RTF_DONE` and `RTF_MASK`, are used in routing messages and not normally stored in the routing table entry.

85   If the `RTF_GATEWAY` flag is set, `rt_gateway` contains a pointer to a socket address structure containing the address (e.g., the IP address) of that gateway. Also,

Constant	netstat flag	Description
RTF_BLACKHOLE		discard packets without error (loopback driver: Figure 5.27)
RTF_CLONING	C	generate new routes on use (used by ARP)
RTF_DONE	d	kernel confirmation that message from process was completed
RTF_DYNAMIC	D	created dynamically (by redirect)
RTF_GATEWAY	G	destination is a gateway (indirect route)
RTF_HOST	H	host entry (else network entry)
RTF_LLINFO	L	set by ARP when rt_llinfo pointer valid
RTF_MASK	m	subnet mask present (not used)
RTF_MODIFIED	M	modified dynamically (by redirect)
RTF_PROTO1	1	protocol-specific routing flag
RTF_PROTO2	2	protocol-specific routing flag (ARP uses)
RTF_REJECT	R	discard packets with error (loopback driver: Figure 5.27)
RTF_STATIC	S	manually added entry (route program)
RTF_UP	U	route usable
RTF_XRESOLVE	X	external daemon resolves name (used with X.25)

Figure 18.25   rt_flags values.

rt_gwroute points to the rtentry for that gateway. This latter pointer was used in ether_output (Figure 4.15).

87   rt_refcnt counts the "held" references to this structure. We describe this counter at the end of Section 19.3. This counter is output as the "Refs" column in Figure 18.2.

88   rt_use is initialized to 0 when the structure is allocated; we saw it incremented in Figure 8.24 each time an IP datagram was output using the route. This counter is also the value printed in the "Use" column in Figure 18.2.

89–90   rt_ifp and rt_ifa point to the interface structure and the interface address structure, respectively. Recall from Figure 6.5 that a given interface can have multiple addresses, so minimally the rt_ifa is required.

92   The rt_llinfo pointer allows link-layer protocols to store pointers to their protocol-specific structures in the routing table entry. This pointer is normally used with the RTF_LLINFO flag. Figure 21.1 shows how ARP uses this pointer.

```
                                                                  route.h
54 struct rt_metrics {
55     u_long  rmx_locks;      /* bitmask for values kernel leaves alone */
56     u_long  rmx_mtu;        /* MTU for this path */
57     u_long  rmx_hopcount;   /* max hops expected */
58     u_long  rmx_expire;     /* lifetime for route, e.g. redirect */
59     u_long  rmx_recvpipe;   /* inbound delay-bandwith product */
60     u_long  rmx_sendpipe;   /* outbound delay-bandwith product */
61     u_long  rmx_ssthresh;   /* outbound gateway buffer limit */
62     u_long  rmx_rtt;        /* estimated round trip time */
63     u_long  rmx_rttvar;     /* estimated RTT variance */
64     u_long  rmx_pksent;     /* #packets sent using this route */
65 };
                                                                  route.h
```

Figure 18.26   rt_metrics structure.

93   Figure 18.26 shows the `rt_metrics` structure, which is contained within the `rtentry` structure. Figure 27.3 shows that TCP uses six members in this structure.

54–65   `rmx_locks` is a bitmask telling the kernel which of the eight metrics that follow must not be modified. The values for this bitmask are shown in Figure 20.13.

`rmx_expire` is used by ARP (Chapter 21) as a timer for each ARP entry. Contrary to the comment with `rmx_expire`, it is not used for redirects.

Figure 18.28 summarizes the structures that we've described, their relationships, and the various types of socket address structures they reference. The `rtentry` that we show is for the route to 128.32.33.5 in Figure 18.2. The other `radix_node` contained in the `rtentry` is for the bit 36 test right above this node in Figure 18.4. The two `sockaddr_dl` structures pointed to by the first `ifaddr` were shown in Figure 3.38. Also note from Figure 6.5 that the `ifnet` structure is contained within an `le_softc` structure, and the second `ifaddr` structure is contained within an `in_ifaddr` structure.

18.7 Initialization: `route_init` and `rtable_init` Functions

The initialization of the routing tables is somewhat obscure and takes us back to the `domain` structures in Chapter 7. Before outlining the function calls, Figure 18.27 shows the relevant fields from the `domain` structure (Figure 7.5) for various protocol families.

Member	OSI value	Internet value	Routing value	Unix value	XNS value	Comment
dom_family	AF_ISO	AF_INET	PF_ROUTE	AF_UNIX	AF_NS	
dom_init	0	0	route_init	0	0	
dom_rtattach	rn_inithead	rn_inithead	0	0	rn_inithead	
dom_rtoffset	48	32	0	0	16	in bits
dom_maxrtkey	32	16	0	0	16	in bytes

Figure 18.27   Members of `domain` structure relevant to routing.

The `PF_ROUTE` domain is the only one with an initialization function. Also, only the domains that require a routing table have a `dom_rtattach` function, and it is always `rn_inithead`. The routing domain and the Unix domain protocols do not require a routing table.

The `dom_rtoffset` member is the offset, in bits, (from the beginning of the domain's socket address structure) of the first bit to be examined for routing. The size of this structure in bytes is given by `dom_maxrtkey`. We saw earlier in this chapter that the offset of the IP address in the `sockaddr_in` structure is 32 bits. The `dom_maxrtkey` member is the size in bytes of the protocol's socket address structure: 16 for `sockaddr_in`.

Figure 18.29 outlines the steps involved in initializing the routing tables.
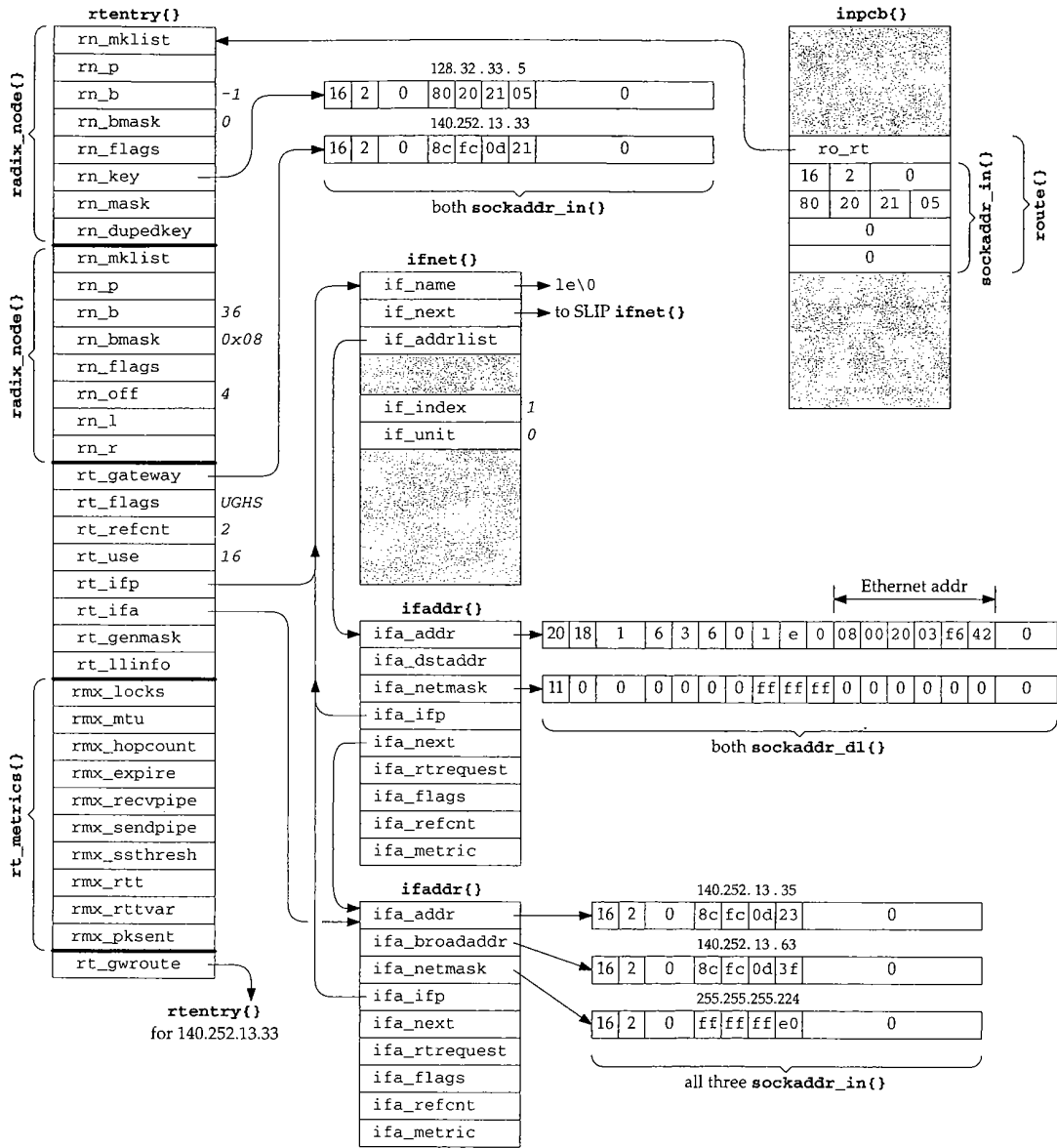
Figure 18.28   Summary of routing structures.

```
main()              /* kernel initialization */
{

    ...
    ifinit();
   ─domaininit();
    ...

}/

domaininit()      /* Figure 7.15 */
{

    ...
    ADDDOMAIN(unix);
    ADDDOMAIN(route);
    ADDDOMAIN(inet);
    ADDDOMAIN(osi);

    ...
    for ( dp = all domains ) {
         ─────────(*dp->dom_init)();
            for ( pr = all protocols for this domain )
                 ─(*pr->pr_init)();
        }
}

raw_init()         /* pr_init() function for SOCK_RAW/PF_ROUTE protocol */
{
    initialize head of routing protocol control blocks;

}

route_init()      /* dom_init() function for PF_ROUTE domain */
{
   ─rn_init();
   ──rtable_init();
}

rn_init()
{

    for ( dp = all domains )
         if (dp->dom_maxrtkey > max_keylen)
             max_keylen = dp->dom_maxrtkey;
    allocate and initialize rn_zeros, rn_ones, masked_key;
    ─rn_inithead(&mask_rnhead);   /* allocate and init tree for masks */
}

rtable_init()
{
    for ( dp = all domains )
     ─(*dp->dom_rtattach)(&rt_tables[dp->dom_family]);
}

rn_inithead()    /* dom_rtattach() function for all protocol families */
{
    allocate and initialize one radix_node_head structure;
}
```

Figure 18.29 Steps involved in initialization of routing tables.

domaininit is called once by the kernel's main function when the system is initialized. The linked list of domain structures is built by the ADDDOMAIN macro and the linked list is traversed, calling each domain's dom_init function, if defined. As we saw in Figure 18.27, the only dom_init function is route_init, which is shown in Figure 18.30.

―― route.c
```
49 void
50 route_init()
51 {
52     rn_init();   /* initialize all zeros, all ones, mask table */

53     rtable_init((void **) rt_tables);
54 }
```
――― route.c

Figure 18.30   route_init function.

The function rn_init, shown in Figure 18.32, is called only once.

The function rtable_init, shown in Figure 18.31, is also called only once. It in turn calls all the dom_rtattach functions, which initialize a routing table tree for that domain.

―― route.c
```
39 void
40 rtable_init(table)
41 void  **table;
42 {
43     struct domain *dom;
44     for (dom = domains; dom; dom = dom->dom_next)
45         if (dom->dom_rtattach)
46             dom->dom_rtattach(&table[dom->dom_family],
47                               dom->dom_rtoffset);
48 }
```
――― route.c

Figure 18.31   rtable_init function: call each domain's dom_rtattach function.

We saw in Figure 18.27 that the only dom_rtattach function is rn_inithead, which we describe in the next section.

18.8   Initialization: rn_init and rn_inithead Functions

The function rn_init, shown in Figure 18.32, is called once by route_init to initialize some of the globals used by the radix functions.

―― radix.c
```
750 void
751 rn_init()
752 {
753     char    *cp, *cplim;
754     struct domain *dom;
```

```
755        for (dom = domains; dom; dom = dom->dom_next)
756            if (dom->dom_maxrtkey > max_keylen)
757                max_keylen = dom->dom_maxrtkey;
758        if (max_keylen == 0) {
759            printf("rn_init: radix functions require max_keylen be set\n");
760            return;
761        }
762        R_Malloc(rn_zeros, char *, 3 * max_keylen);
763        if (rn_zeros == NULL)
764            panic("rn_init");
765        Bzero(rn_zeros, 3 * max_keylen);
766        rn_ones = cp = rn_zeros + max_keylen;
767        maskedKey = cplim = rn_ones + max_keylen;
768        while (cp < cplim)
769            *cp++ = -1;

770        if (rn_inithead((void **) &mask_rnhead, 0) == 0)
771            panic("rn_init 2");
772    }
```
── *radix.c*

Figure 18.32 rn_init function.

Determine max_keylen

750–761   All the domain structures are examined and the global max_keylen is set to the largest value of dom_maxrtkey. In Figure 18.27 the largest value is 32 for AF_ISO, but in a typical system that excludes the OSI and XNS protocols, max_keylen is 16, the size of a sockaddr_in structure.

Allocate and initialize rn_zeros, rn_ones, and maskedKey

762–769   A buffer three times the size of max_keylen is allocated and the pointer stored in the global rn_zeros. R_Malloc is a macro that calls the kernel's malloc function, specifying a type of M_RTABLE and M_DONTWAIT. We'll also encounter the macros Bcmp, Bcopy, Bzero, and Free, which call kernel functions of similar names, with the arguments appropriately type cast.

This buffer is divided into three pieces, and each piece is initialized as shown in Figure 18.33.
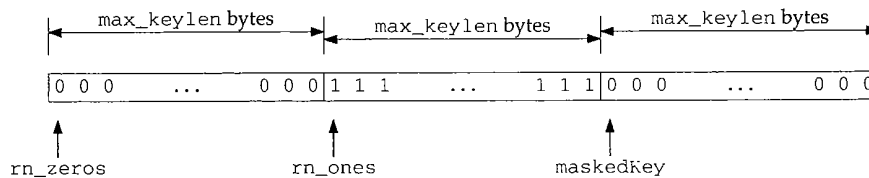

Figure 18.33 rn_zeros, rn_ones, and maskedKey arrays.

rn_zeros is an array of all zero bits, rn_ones is an array of all one bits, and maskedKey is an array used to hold a temporary copy of a search key that has been masked.

Initialize tree of masks

770–772   The function rn_inithead is called to initialize the head of the routing tree for the address masks; the radix_node_head structure pointed to by the global mask_rnhead in Figure 18.8.

From Figure 18.27 we see that rn_inithead is also the dom_attach function for all the protocols that require a routing table. Instead of showing the source code for this function, Figure 18.34 shows the radix_node_head structure that it builds for the Internet protocols.
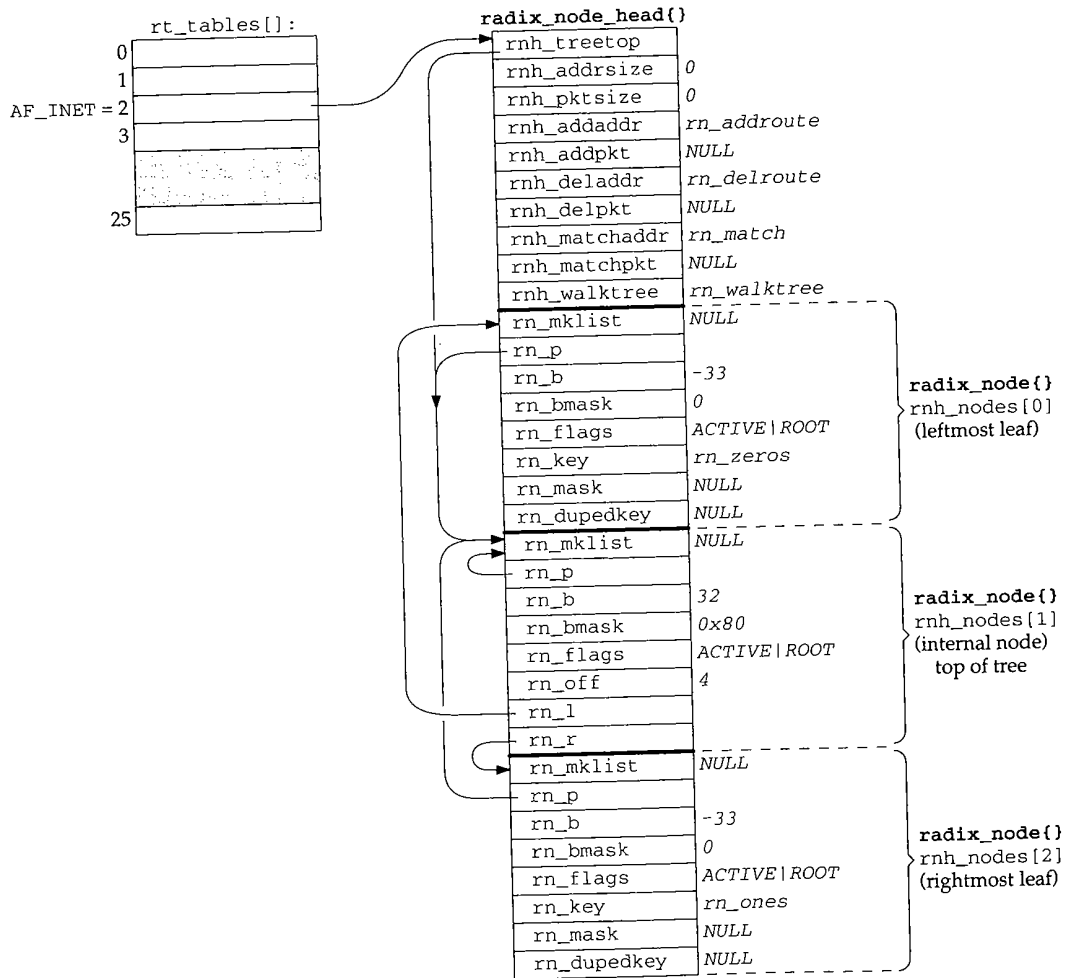

Figure 18.34   radix_node_head structure built by rn_inithead for Internet protocols.

The three radix_node structures form a tree: the middle of the three is the top (it is pointed to by rnh_treetop), the first of the three is the leftmost leaf of the tree, and

the last of the three is the rightmost leaf of the tree. The parent pointer of all three nodes (rn_p) points to the middle node.

The value 32 for rnh_nodes[1].rn_b is the bit position to test. It is from the dom_rtoffset member of the Internet domain structure (Figure 18.27). Instead of performing shifts and masks during forwarding, the byte offset and corresponding byte mask are precomputed. The byte offset from the start of a socket address structure is in the rn_off member of the radix_node structure (4 in this case) and the byte mask is in the rn_bmask member (0x80 in this case). These values are computed whenever a radix_node structure is added to the tree, to speed up the comparisons during forwarding. As additional examples, the offset and byte mask for the two nodes that test bit 33 in Figure 18.4 would be 4 and 0x40, respectively. The offset and byte mask for the two nodes that test bit 63 would be 7 and 0x01.

The value of −33 for the rn_b member of both leaves is negative one minus the index of the leaf.

The key of the leftmost node is all zero bits (rn_zeros) and the key of the rightmost node is all one bits (rn_ones).

All three nodes have the RNF_ROOT flag set. (We have omitted the RNF_ prefix.) This indicates that the node is one of the three original nodes used to build the tree. These are the only nodes with this flag.

> One detail we have not mentioned is that the Network File System (NFS) also uses the routing table functions. For each mount point on the local host a radix_node_head structure is allocated, along with an array of pointers to these structures (indexed by the protocol family), similar to the rt_tables array. Each time this mount point is exported, the protocol address of the host that can mount this filesystem is added to the appropriate tree for the mount point.

18.9 Duplicate Keys and Mask Lists

Before looking at the source code that looks up entries in a routing table we need to understand two fields in the radix_node structure: rn_dupedkey, which forms a linked list of additional radix_node structures containing duplicate keys, and rn_mklist, which starts a linked list of radix_mask structures containing network masks.

We first return to Figure 18.4 and the two boxes on the far left of the tree labeled "end" and "default." These are duplicate keys. The leftmost node with the RNF_ROOT flag set (rnh_nodes[0] in Figure 18.34) has a key of all zero bits, but this is the same key as the default route. We would have the same problem with the rightmost end node in the tree, which has a key of all one bits, if an entry were created for 255.255.255.255, but this is the limited broadcast address, which doesn't appear in the routing table. In general, the radix node functions in Net/3 allow any key to be duplicated, if each occurrence has a unique mask.

Figure 18.35 shows the two nodes with a duplicate key of all zero bits. In this figure we have removed the RNF_ prefix for the rn_flags and omit nonnull parent, left, and right pointers, which add nothing to the discussion.
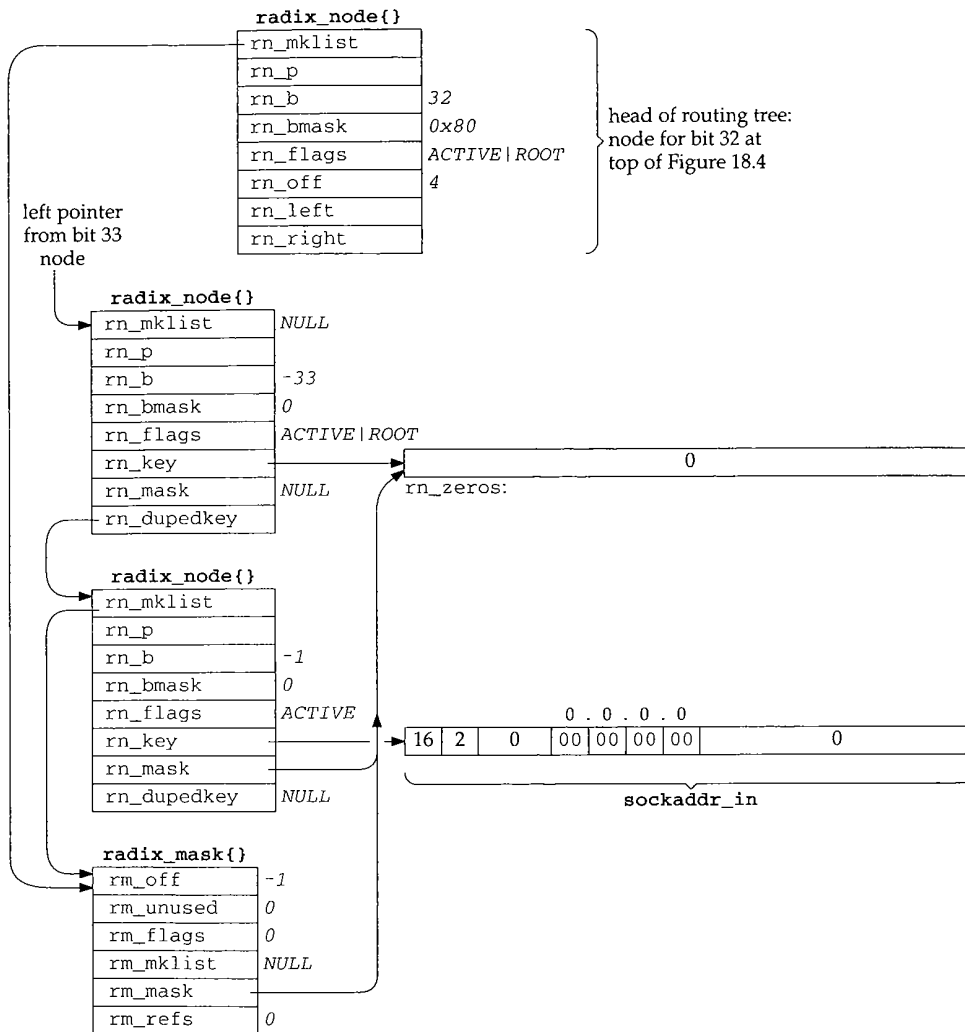
Figure 18.35   Duplicated nodes with a key of all zero bits.

The top node is the top of the routing tree—the node for bit 32 at the top of Figure 18.4. The next two nodes are leaves (their `rn_b` values are negative) with the `rn_dupedkey` member of the first pointing to the second. The first of these two leaves is the `rnh_nodes[0]` structure from Figure 18.34, which is the left end marker of the tree—its `RNF_ROOT` flag is set. Its key was explicitly set by `rn_inithead` to `rn_zeros`.

The second of these leaves is the entry for the default route. Its `rn_key` points to a `sockaddr_in` with the value 0.0.0.0, and it has a mask of all zero bits. Its `rn_mask` points to `rn_zeros`, since equivalent masks in the mask table are shared.

Normally keys are not shared, let alone shared with masks. The rn_key pointers of the two end markers (those with the RNF_ROOT flag) are special since they are built by rn_inithead (Figure 18.34). The key of the left end marker points to rn_zeros and the key of the right end marker points to rn_ones.

The final structure is a radix_mask structure and is pointed to by both the top node of the tree and the leaf for the default route. The list from the top node of the tree is used with the backtracking algorithm when the search is looking for a network mask. The list of radix_mask structures with an internal node specifies the masks that apply to subtrees starting at that node. In the case of duplicate keys, a mask list also appears with the leaves, as we'll see in the following example.

We now show a duplicate key that is added to the routing tree intentionally and the resulting mask list. In Figure 18.4 we have a host route for 127.0.0.1 and a network route for 127.0.0.0. The default mask for the class A network route is 0xff000000, as we show in the figure. If we divide the 24 bits following the class A network ID into a 16-bit subnet ID and an 8-bit host ID, we can add a route for the subnet 127.0.0 with a mask of 0xffffff00:

```
bsdi $ route add 127.0.0.0 -netmask 0xffffff00 140.252.13.33
```

Although it makes little practical sense to use network 127 in this fashion, our interest is in the resulting routing table structure. Although duplicate keys are not common with the Internet protocols (other than the previous example with the default route), duplicate keys are required to provide routes to subnet 0 of any network.

There is an implied priority in these three entries with a network ID of 127. If the search key is 127.0.0.1 it matches all three entries, but the host route is selected because it is the *most specific*: its mask (0xffffffff) has the most one bits. If the search key is 127.0.0.2 it matches both network routes, but the route for subnet 0, with a mask of 0xffffff00, is more specific than the route with a mask of 0xff000000. The search key 127.1.2.3 matches only the entry with a mask of 0xff000000.

Figure 18.36 shows the resulting tree structure, starting at the internal node for bit 33 from Figure 18.4. We show two boxes for the entry with the key of 127.0.0.0 since there are two leaves with this duplicate key.
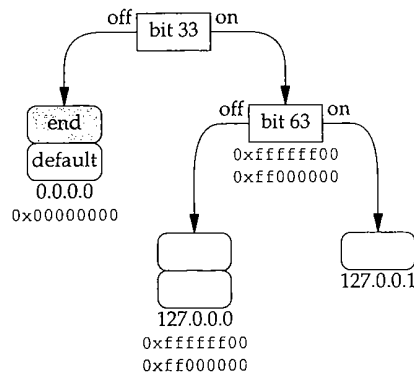

Figure 18.36   Routing tree showing duplicate keys for 127.0.0.0.

Figure 18.37 shows the resulting `radix_node` and `radix_mask` structures.
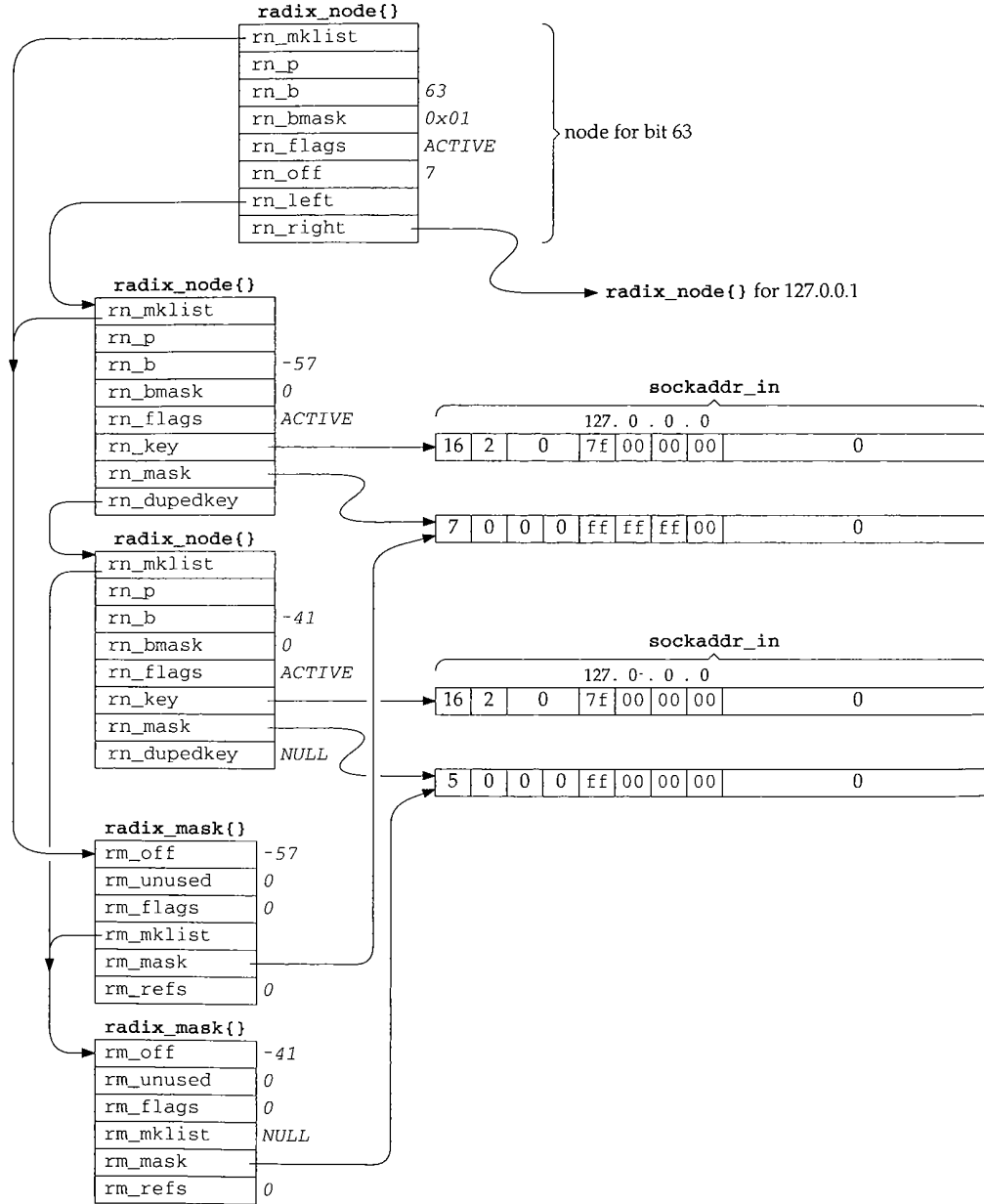

Figure 18.37   Example routing table structures for the duplicate keys for network 127.0.0.0.

First look at the linked list of `radix_mask` structures for each `radix_node`. The mask list for the top node (bit 63) consists of the entry for `0xffffff00` followed by `0xff000000`. The more-specific mask comes first in the list so that it is tried first. The mask list for the second `radix_node` (the one with the `rn_b` of –57) is the same as that of the first. But the list for the third `radix_node` consists of only the entry with a mask of `0xff000000`.

Notice that masks with the same value are shared but keys with the same value are not. This is because the masks are maintained in their own routing tree, explicitly to be shared, because equal masks are so common (e.g., every class C network route has the same mask of `0xffffff00`), while equal keys are infrequent.

18.10 `rn_match` Function

We now show the `rn_match` function, which is called as the `rnh_matchaddr` function for the Internet protocols. We'll see that it is called by the `rtalloc1` function, which is called by the `rtalloc` function. The algorithm is as follows:

1. Start at the top of the tree and go to the leaf corresponding to the bits in the search key. Check the leaf for an exact match (Figure 18.38).

2. Check the leaf for a network match (Figure 18.40).

3. Backtrack (Figure 18.43).

Figure 18.38 shows the first part of `rn_match`.

```
                                                                        ── radix.c
135 struct radix_node *
136 rn_match(v_arg, head)
137 void    *v_arg;
138 struct radix_node_head *head;
139 {
140     caddr_t v = v_arg;
141     struct radix_node *t = head->rnh_treetop, *x;
142     caddr_t cp = v, cp2, cp3;
143     caddr_t cplim, mstart;
144     struct radix_node *saved_t, *top = t;
145     int     off = t->rn_off, vlen = *(u_char *) cp, matched_off;

146     /*
147      * Open code rn_search(v, top) to avoid overhead of extra
148      * subroutine call.
149      */
150     for (; t->rn_b >= 0;) {
151         if (t->rn_bmask & cp[t->rn_off])
152             t = t->rn_r;        /* right if bit on */
153         else
154             t = t->rn_l;        /* left if bit off */
155     }
```

```
156     /*
157      * See if we match exactly as a host destination
158      */
159     cp += off;
160     cp2 = t->rn_key + off;
161     cplim = v + vlen;
162     for (; cp < cplim; cp++, cp2++)
163         if (*cp != *cp2)
164             goto on1;
165     /*
166      * This extra grot is in case we are explicitly asked
167      * to look up the default.  Ugh!
168      */
169     if ((t->rn_flags & RNF_ROOT) && t->rn_dupedkey)
170         t = t->rn_dupedkey;
171     return t;
172  on1:
```
——— *radix.c*

Figure 18.38   `rn_match` function: go down tree, check for exact host match.

135–145   The first argument `v_arg` is a pointer to a socket address structure, and the second argument `head` is a pointer to the `radix_node_head` structure for the protocol. All protocols call this function (Figure 18.17) but each calls it with a different `head` argument.

In the assignment statements, `off` is the `rn_off` member of the top node of the tree (4 for Internet addresses, from Figure 18.34), and `vlen` is the length field from the socket address structure of the search key (16 for Internet addresses).

Go down the tree to the corresponding leaf

146–155   This loop starts at the top of the tree and moves down the left and right branches until a leaf is encountered (`rn_b` is less than 0). Each test of the appropriate bit is made using the precomputed byte mask in `rn_bmask` and the corresponding precomputed offset in `rn_off`. For Internet addresses, `rn_off` will be 4, 5, 6, or 7.

Check for exact match

156–164   When the leaf is encountered, a check is first made for an exact match. *All* bytes of the socket address structure, starting at the `rn_off` value for the protocol family, are compared. This is shown in Figure 18.39 for an Internet socket address structure.
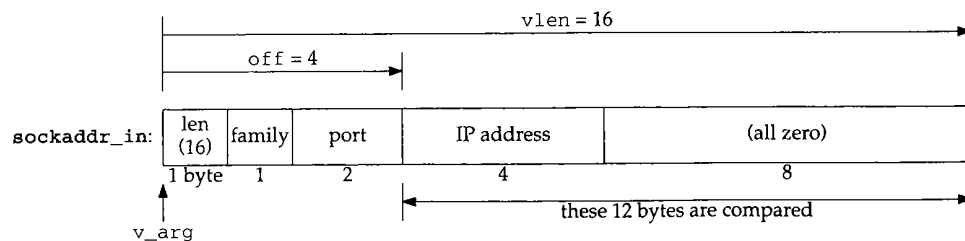

Figure 18.39   Variables during comparison of `sockaddr_in` structures.

As soon as a mismatch is found, a jump is made to `on1`.

> Normally the final 8 bytes of the sockaddr_in are 0 but proxy ARP (Section 21.12) sets one of these bytes nonzero. This allows two routing table entries for a given IP address: one for the normal IP address (with the final 8 bytes of 0) and a proxy ARP entry for the same IP address (with one of the final 8 bytes nonzero).

The length byte in Figure 18.39 was assigned to vlen at the beginning of the function, and we'll see that rtalloc1 uses the family member to select the routing table to search. The port is never used by the routing functions.

Explicit check for default

165–172   Figure 18.35 showed that the default route is stored as a duplicate leaf with a key of 0. The first of the duplicate leaves has the RNF_ROOT flag set. Hence if the RNF_ROOT flag is set in the matching node and the leaf contains a duplicate key, the value of the pointer rn_dupedkey is returned (i.e., the pointer to the node containing the default route in Figure 18.35). If a default route has not been entered and the search matches the left end marker (a key of all zero bits), or if the search encounters the right end marker (a key of all one bits), the returned pointer t points to a node with the RNF_ROOT flag set. We'll see that rtalloc1 explicitly checks whether the matching node has this flag set, and considers such a match an error.

At this point in rn_match a leaf has been reached but it is not an exact match with the search key. The next part of the function, shown in Figure 18.40, checks whether the leaf is a network match.

—— radix.c
```
173        matched_off = cp - v;
174        saved_t = t;
175        do {
176            if (t->rn_mask) {
177                /*
178                 * Even if we don't match exactly as a host;
179                 * we may match if the leaf we wound up at is
180                 * a route to a net.
181                 */
182                cp3 = matched_off + t->rn_mask;
183                cp2 = matched_off + t->rn_key;
184                for (; cp < cplim; cp++)
185                    if ((*cp2++ ^ *cp) & *cp3++)
186                        break;
187                if (cp == cplim)
188                    return t;
189                cp = matched_off + v;
190            }
191        } while (t = t->rn_dupedkey);
192        t = saved_t;
```
—— radix.c

Figure 18.40   rn_match function: check for network match.

173–174   cp points to the unequal byte in the search key. matched_off is set to the offset of this byte from the start of the socket address structure.

175–183   The do while loop iterates through all duplicate leaves and each one with a network mask is compared. Let's work through the code with an example. Assume we're

looking up the IP address 140.252.13.60 in the routing table in Figure 18.4. The search will end up at the node labeled 140.252.13.32 (bits 62 and 63 are both off), which contains a network mask. Figure 18.41 shows the structures when the for loop in Figure 18.40 starts executing.
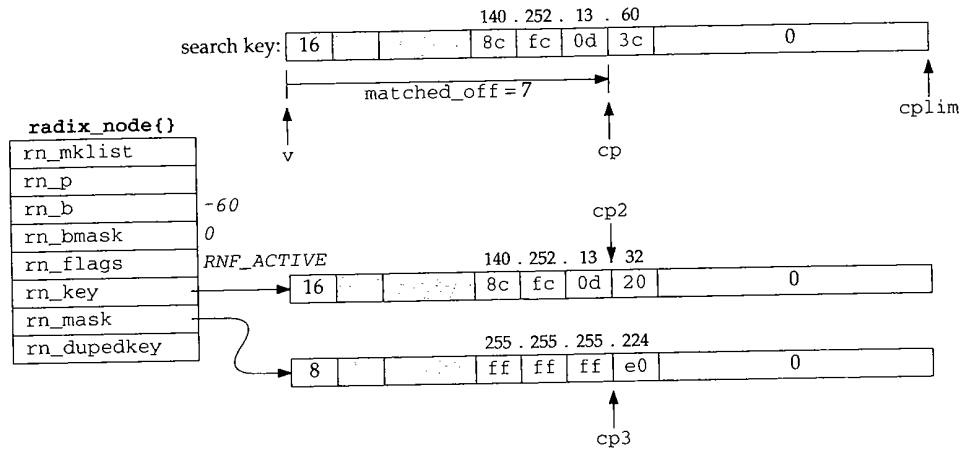

Figure 18.41   Example for network mask comparison.

The search key and the routing table key are both sockaddr_in structures, but the length of the mask is different. The mask length is the minimum number of bytes containing nonzero values. All the bytes past this point, up through max_keylen, are 0.

184–190   The search key is exclusive ORed with the routing table key, and the result logically ANDed with the network mask, one byte at a time. If the resulting byte is ever nonzero, the loop terminates because they don't match (Exercise 18.1). If the loop terminates normally, however, the search key ANDed with the network mask matches the routing table entry. The pointer to the routing table entry is returned.

Figure 18.42 shows how this example matches, and how the IP address 140.252.13.188 does not match, looking at just the fourth byte of the IP address. The search for both IP addresses ends up at this node since both addresses have bits 57, 62, and 63 off.

	search key = 140.252.13.60	search key = 140.252.13.188
search key byte (*cp):	0011 1100 = 3c	1011 1100 = bc
routing table key byte (*cp2):	0010 0000 = 20	0010 0000 = 20
exclusive OR:	0001 1100	1001 1100
network mask byte (*cp3):	1110 0000 = e0	1110 0000 = e0
logical AND:	0000 0000	1000 0000

Figure 18.42   Example of search key match using network mask.

The first example (140.252.13.60) matches since the result of the logical AND is 0 (and all the remaining bytes in the address, the key, and the mask are all 0). The other example does not match since the result of the logical AND is nonzero.

191   If the routing table entry has duplicate keys, the loop is repeated for each key.

The final portion of rn_match, shown in Figure 18.43, backtracks up the tree, looking for a network match or a match with the default.

```
                                                                        ──────── radix.c
193      /* start searching up the tree */
194      do {
195          struct radix_mask *m;
196          t = t->rn_p;
197          if (m = t->rn_mklist) {
198              /*
199               * After doing measurements here, it may
200               * turn out to be faster to open code
201               * rn_search_m here instead of always
202               * copying and masking.
203               */
204              off = min(t->rn_off, matched_off);
205              mstart = maskedKey + off;
206              do {
207                  cp2 = mstart;
208                  cp3 = m->rm_mask + off;
209                  for (cp = v + off; cp < cplim;)
210                      *cp2++ = *cp++ & *cp3++;
211                  x = rn_search(maskedKey, t);
212                  while (x && x->rn_mask != m->rm_mask)
213                      x = x->rn_dupedkey;
214                  if (x &&
215                      (Bcmp(mstart, x->rn_key + off,
216                          vlen - off) == 0))
217                      return x;
218              } while (m = m->rm_mklist);
219          }
220      } while (t != top);
221      return 0;
222  };
                                                                        ──────── radix.c
```

Figure 18.43   rn_match function: backtrack up the tree.

193–195   The do while loop continues up the tree, checking each level, until the top has been checked.

196   The pointer t is replaced with the pointer to the parent node, moving up one level. Having the parent pointer in each node simplifies backtracking.

197–210   Each level is checked only if the internal node has a nonnull list of masks. rn_mklist is a pointer to a linked list of radix_mask structures, each containing a mask that applies to the subtree starting at that node. The inner do while loop iterates through each radix_mask structure on the list.

Using the previous example, 140.252.13.188, Figure 18.44 shows the various data structures when the innermost for loop starts. This loop logically ANDs each byte of the search key with each byte of the mask, storing the result in the global maskedKey. The mask value is 0xfffffffe0 and the search would have backtracked from the leaf for 140.252.13.32 in Figure 18.4 two levels to the node that tests bit 62.
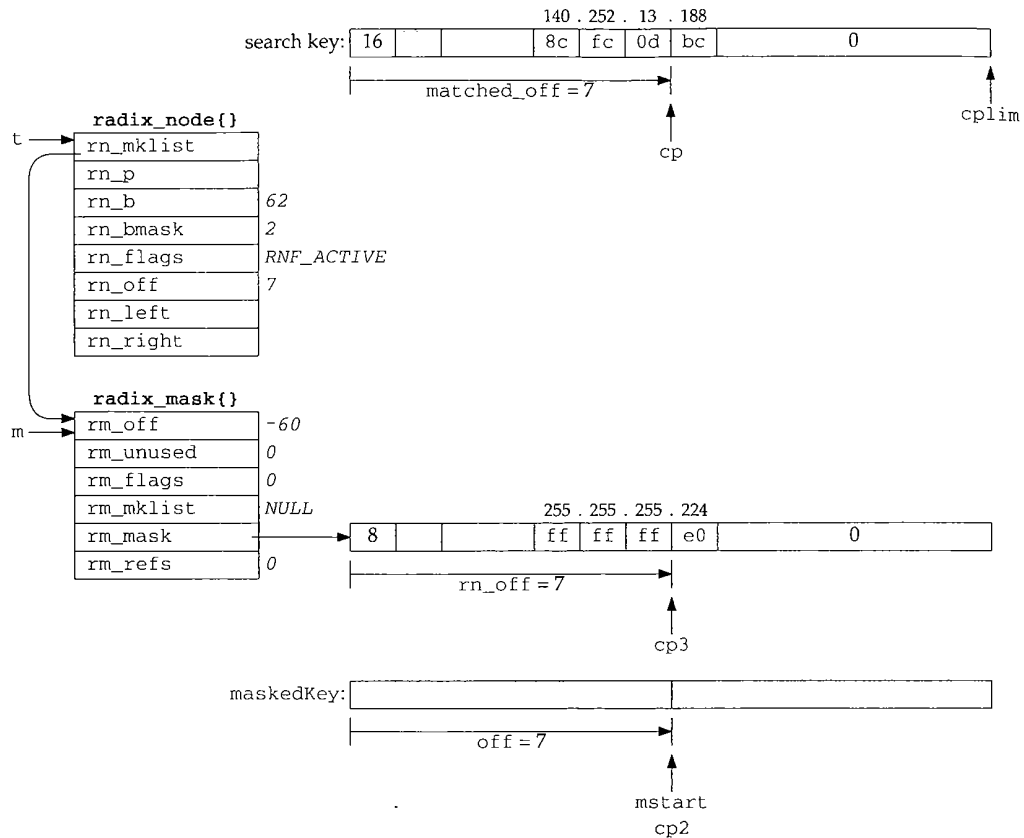
Figure 18.44   Preparation to search again using masked search key.

Once the `for` loop completes, the masking is complete, and `rn_search` (shown in Figure 18.48) is called with `maskedKey` as the search key and the pointer `t` as the top of the subtree to search. Figure 18.45 shows the value of `maskedKey` for our example.
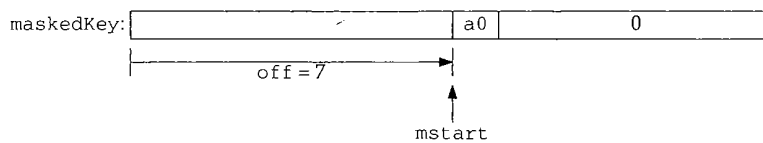

Figure 18.45   `maskedKey` when `rn_search` is called.

The byte `0xa0` is the logical AND of `0xbc` (188, the search key) and `0xe0` (the mask). `rn_search` proceeds down the tree from its starting point, branching right or left depending on the key, until a leaf is reached. In this example the search key is the 9 bytes shown in Figure 18.45 and the leaf that's reached is the one labeled 140.252.13.32 in Figure 18.4, since bits 62 and 63 are off in the byte `0xa0`. Figure 18.46 shows the data structures when `Bcmp` is called to check if a match has been found.
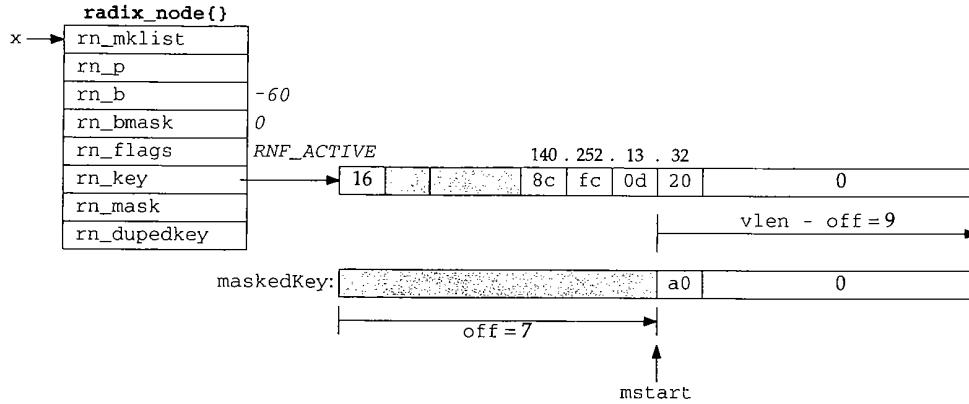
Figure 18.46 Comparison of maskedKey and new leaf.

Since the 9-byte strings are not the same, the comparison fails.

212–221   This while loop handles duplicate keys, each with a different mask. The only key of the duplicates that is compared is the one whose rn_mask pointer equals m->rm_mask. As an example, recall Figures 18.36 and 18.37. If the search starts at the node for bit 63, the first time through the inner do while loop m points to the radix_mask structure for 0xffffff00. When rn_search returns the pointer to the first of the duplicate leaves for 127.0.0.0, the rm_mask of this leaf equals m->rm_mask, so Bcmp is called. If the comparison fails, m is replaced with the pointer to the next radix_mask structure on the list (the one with a mask of 0xff000000) and the do while loop iterates around again with the new mask. rn_search again returns the pointer to the first of the duplicate leaves for 127.0.0.0, but its rn_mask does not equal m->rm_mask. The while steps to the next of the duplicate leaves and its rn_mask is the right one.

Returning to our example with the search key of 140.252.13.188, since the search from the node that tests bit 62 failed, the backtracking continues up the tree until the top is reached, which is the next node up the tree with a nonnull rn_mklist.

Figure 18.47 shows the data structures when the top node of the tree is reached. At this point maskedKey is computed (it is all zero bits) and rn_search starts at this node (the top of the tree) and continues down the two left branches to the leaf labeled "default" in Figure 18.4.

When rn_search returns, x points to the radix_node with an rn_b of –33, which is the first leaf encountered after the two left branches from the top of the tree. But x->rn_mask (which is null) does not equal m->rm_mask, so x is replaced with x->rn_dupedkey. The test of the while loop occurs again, but now x->rn_mask equals m->rm_mask, so the while loop terminates. Bcmp compares the 12 bytes of 0 starting at mstart with the 12 bytes of 0 stating at x->rn_key plus 4, and since they're equal, the function returns the pointer x, which points to the entry for the default route.
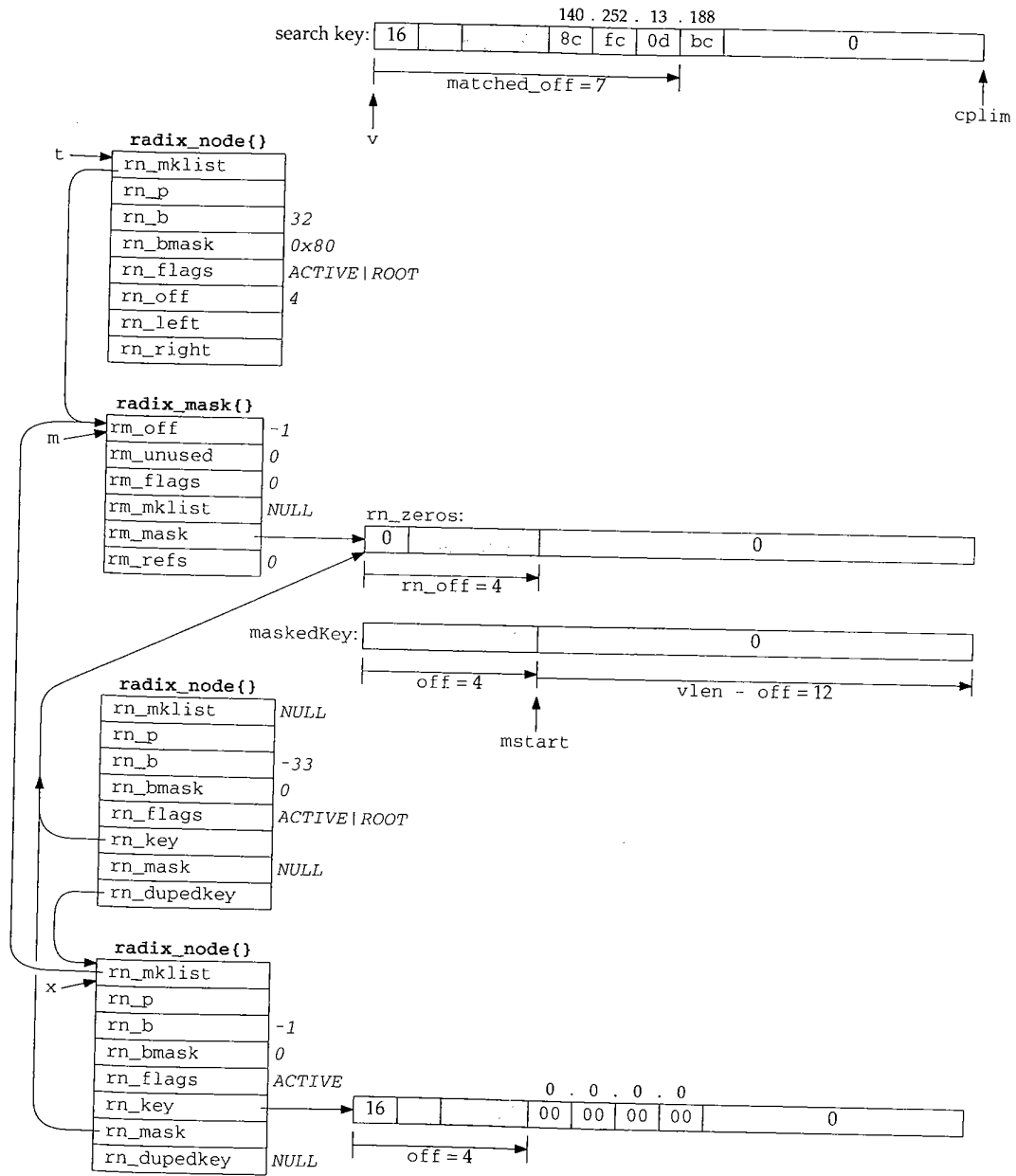
Figure 18.47   Backtrack to top of tree and `rn_search` that locates default leaf.

18.11 `rn_search` Function

`rn_search` was called in the previous section from `rn_match` to search a subtree of the routing table.

——— *radix.c*
```
79 struct radix_node *
80 rn_search(v_arg, head)
81 void    *v_arg;
82 struct  radix_node *head;
83 {
84     struct radix_node *x;
85     caddr_t v;

86     for (x = head, v = v_arg; x->rn_b >= 0;) {
87         if (x->rn_bmask & v[x->rn_off])
88             x = x->rn_r;        /* right if bit on */
89         else
90             x = x->rn_l;        /* left if bit off */
91     }
92     return (x);
93 };
```
——— *radix.c*

Figure 18.48   `rn_search` function.

This loop is similar to the one in Figure 18.38. It compares one bit in the search key at each node, branching left if the bit is off or right if the bit is on, terminating when a leaf is encountered. The pointer to that leaf is returned.

18.12 Summary

Each routing table entry is identified by a key: the destination IP address in the case of the Internet protocols, which is either a host address or a network address with an associated network mask. Once the entry is located by searching for the key, additional information in the entry specifies the IP address of a router to which datagrams should be sent for the destination, a pointer to the interface to use, metrics, and so on.

The information maintained by the Internet protocols is the `route` structure, composed of just two elements: a pointer to a routing table entry and the destination address. We'll encounter one of these `route` structures in each of the Internet protocol control blocks used by UDP, TCP, and raw IP.

The Patricia tree data structure is well suited to routing tables. Routing table lookups occur much more frequently than adding or deleting routes, so from a performance standpoint using Patricia trees for the routing table makes sense. Patricia trees provide fast lookups at the expense of additional work in adding and deleting. Measurements in [Sklower 1991] comparing the radix tree approach to the Net/1 hash table show that the radix tree method is about two times faster in building a test tree and four times faster in searching.

Exercises

18.1   We said with Figure 18.3 that the general condition for matching a routing table entry is that the search key logically ANDed with the routing table mask equal the routing table key. But in Figure 18.40 a different test is used. Build a logic truth table showing that the two tests are the same.

18.2   Assume a Net/3 system needs a routing table with 20,000 entries (IP addresses). Approximately how much memory is required for this, ignoring the space required for the masks?

18.3   What is the limit imposed on the length of a routing table key by the `radix_node` structure?

1

:er 18

try is
table
at the

ɔroxi-
.sks?

struc-

19

Routing Requests and
Routing Messages

19.1 Introduction

The various protocols within the kernel don't access the routing trees directly, using the functions from the previous chapter, but instead call a few functions that we describe in this chapter: `rtalloc` and `rtalloc1` are two that perform routing table lookups, `rtrequest` adds and deletes routing table entries, and `rtinit` is called by most interfaces when the interface goes up or down.

Routing messages communicate information in two directions. A process such as the `route` command or one of the routing daemons (`routed` or `gated`) writes routing messages to a routing socket, causing the kernel to add a new route, delete an existing route, or modify an existing route. The kernel also generates routing messages that can be read by any routing socket when events occur in which the processes might be interested: an interface has gone down, a redirect has been received, and so on. In this chapter we cover the formats of these routing messages and the information contained therein, and we save our discussion of routing sockets until the next chapter.

Another interface provided by the kernel to the routing tables is through the `sysctl` system call, which we describe at the end of this chapter. This system call allows a process to read the entire routing table or a list of all the configured interfaces and interface addresses.

19.2 `rtalloc` and `rtalloc1` Functions

`rtalloc` and `rtalloc1` are the functions normally called to look up an entry in the routing table. Figure 19.1 shows `rtalloc`.

601

```
58 void
59 rtalloc(ro)                                                              ──── route.c
60 struct route *ro;
61 {
62     if (ro->ro_rt && ro->ro_rt->rt_ifp && (ro->ro_rt->rt_flags & RTF_UP))
63         return;                              /* XXX */
64     ro->ro_rt = rtalloc1(&ro->ro_dst, 1);
65 }
```
──── route.c

Figure 19.1 rtalloc function.

58–65   The argument `ro` is often the pointer to a `route` structure contained in an Internet PCB (Chapter 22) which is used by UDP and TCP. If `ro` already points to an `rtentry` structure (`ro_rt` is nonnull), and that structure points to an interface structure, and the route is up, the function returns. Otherwise `rtalloc1` is called with a second argument of 1. We'll see the purpose of this argument shortly.

`rtalloc1`, shown in Figure 19.2, calls the `rnh_matchaddr` function, which is always `rn_match` (Figure 18.17) for Internet addresses.

66–76   The first argument is a pointer to a socket address structure containing the address to search for. The `sa_family` member selects the routing table to search.

Call rn_match

77–78   If the following three conditions are met, the search is successful.

1. A routing table exists for the protocol family,
2. `rn_match` returns a nonnull pointer, and
3. the matching `radix_node` does not have the `RNF_ROOT` flag set.

Remember that the two leaves that mark the end of the tree both have the `RNF_ROOT` flag set.

Search fails

94–101   If the search fails because any one of the three conditions is not met, the statistic `rts_unreach` is incremented and if the second argument to `rtalloc1` (report) is nonzero, a routing message is generated that can be read by any interested processes on a routing socket. The routing message has the type `RTM_MISS`, and the function returns a null pointer.

79   If all three of the conditions are met, the lookup succeeded and the pointer to the matching `radix_node` is stored in `rt` and `newrt`. Notice that in the definition of the `rtentry` structure (Figure 18.24) the two `radix_node` structures are at the beginning, and, as shown in Figure 18.8, the first of these two structures contains the leaf node. Therefore the pointer to a `radix_node` structure returned by `rn_match` is really a pointer to an `rtentry` structure, which is the matching leaf node.

```
                                                                               ──── route.c
66  struct rtentry *
67  rtalloc1(dst, report)
68  struct sockaddr *dst;
69  int      report;
70  {
71      struct radix_node_head *rnh = rt_tables[dst->sa_family];
72      struct rtentry *rt;
73      struct radix_node *rn;
74      struct rtentry *newrt = 0;
75      struct rt_addrinfo info;
76      int      s = splnet(), err = 0, msgtype = RTM_MISS;

77      if (rnh && (rn = rnh->rnh_matchaddr((caddr_t) dst, rnh)) &&
78              ((rn->rn_flags & RNF_ROOT) == 0)) {
79          newrt = rt = (struct rtentry *) rn;
80          if (report && (rt->rt_flags & RTF_CLONING)) {
81              err = rtrequest(RTM_RESOLVE, dst, SA(0),
82                                  SA(0), 0, &newrt);
83              if (err) {
84                  newrt = rt;
85                  rt->rt_refcnt++;
86                  goto miss;
87              }
88              if ((rt = newrt) && (rt->rt_flags & RTF_XRESOLVE)) {
89                  msgtype = RTM_RESOLVE;
90                  goto miss;
91              }
92          } else
93              rt->rt_refcnt++;
94      } else {
95          rtstat.rts_unreach++;
96        miss:if (report) {
97              bzero((caddr_t) & info, sizeof(info));
98              info.rti_info[RTAX_DST] = dst;
99              rt_missmsg(msgtype, &info, 0, err);
100         }
101     }
102     splx(s);
103     return (newrt);
104 }
                                                                               ──── route.c
```

Figure 19.2   rtalloc1 function.

Create clone entries

80–82   If the caller specified a nonzero second argument, and if the RTF_CLONING flag is set, rtrequest is called with a command of RTM_RESOLVE to create a new rtentry structure that is a clone of the one that was located. This feature is used by ARP and for multicast addresses.

Clone creation fails

83–87   If `rtrequest` returns an error, `newrt` is set back to the entry returned by `rn_match` and its reference count is incremented. A jump is made to `miss` where an `RTM_MISS` message is generated.

Check for external resolution

88–91   If `rtrequest` succeeds but the newly cloned entry has the `RTF_XRESOLVE` flag set, a jump is made to `miss`, this time to generate an `RTM_RESOLVE` message. The intent of this message is to notify a user process when the route is created, and it could be used with the conversion of IP addresses to X.121 addresses.

Increment reference count for normal successful search

92–93   When the search succeeds but the `RTF_CLONING` flag is not set, this statement increments the entry's reference count. This is the normal flow through the function, which then returns the nonnull pointer.

For a small function, `rtalloc1` has many options in how it operates. There are seven different flows through the function, summarized in Figure 19.3.

	report argument	RTF_-CLONING flag	RTM_-RESOLVE return	RTF_-XRESOLVE flag	routing message generated	rt_refcnt	return value
entry not found	0						null
	1				RTM_MISS		null
entry found		0				++	ptr
	0					++	ptr
	1	1	OK	0		++	ptr
	1	1	OK	1	RTM_RESOLVE	++	ptr
	1	1	error		RTM_MISS	++	ptr

Figure 19.3   Summary of operation of `rtalloc1`.

We note that the first two rows (entry not found) are impossible if a default route exists. Also we show `rt_refcnt` being incremented in the fifth and sixth rows when the call to `rtrequest` with a command of `RTM_RESOLVE` is OK. The increment is done by `rtrequest`.

19.3   `RTFREE` Macro and `rtfree` Function

The `RTFREE` macro, shown in Figure 19.4, calls the `rtfree` function only if the reference count is less than or equal to 1, otherwise it just decrements the reference count.

209–213   The `rtfree` function, shown in Figure 19.5, releases an `rtentry` structure when there are no more references to it. We'll see in Figure 22.7, for example, that when a protocol control block is released, if it points to a routing entry, `rtfree` is called.

```
209 #define RTFREE(rt) \                                          route.h
210     if ((rt)->rt_refcnt <= 1) \
211         rtfree(rt); \
212     else \
213         (rt)->rt_refcnt--;        /* no need for function call */
                                                                  route.h
```

Figure 19.4 RTFREE macro.

```
105 void                                                          route.c
106 rtfree(rt)
107 struct rtentry *rt;
108 {
109     struct ifaddr *ifa;

110     if (rt == 0)
111         panic("rtfree");
112     rt->rt_refcnt--;
113     if (rt->rt_refcnt <= 0 && (rt->rt_flags & RTF_UP) == 0) {
114         if (rt->rt_nodes->rn_flags & (RNF_ACTIVE | RNF_ROOT))
115             panic("rtfree 2");
116         rttrash--;
117         if (rt->rt_refcnt < 0) {
118             printf("rtfree: %x not freed (neg refs)\n", rt);
119             return;
120         }
121         ifa = rt->rt_ifa;
122         IFAFREE(ifa);
123         Free(rt_key(rt));
124         Free(rt);
125     }
126 }
                                                                  route.c
```

Figure 19.5 rtfree function: release an rtentry structure.

105–115 The entry's reference count is decremented and if it is less than or equal to 0 and the route is not usable, the entry can be released. If either of the flags RNF_ACTIVE or RNF_ROOT are set, this is an internal error. If RNF_ACTIVE is set, this structure is still part of the routing table tree. If RNF_ROOT is set, this structure is one of the end markers built by rn_inithead.

116 rttrash is a debugging counter of the number of routing entries not in the routing tree, but not released. It is incremented by rtrequest when it begins deleting a route, and then decremented here. Its value should normally be 0.

Release interface reference

117–122 A check is made that the reference count is not negative, and then IFAFREE decrements the reference count for the ifaddr structure and releases it by calling ifafree when it reaches 0.

Release routing memory

123–124   The memory occupied by the routing entry key and its gateway is released. We'll
see in `rt_setgate` that the memory for both is allocated in one contiguous chunk,
allowing both to be released with a single call to `Free`. Finally the `rtentry` structure
itself is released.

Routing Table Reference Counts

The handling of the routing table reference count, `rt_refcnt`, differs from most other
reference counts. We see in Figure 18.2 that most routes have a reference count of 0, yet
the routing table entries without any references are not deleted. We just saw the reason
in `rtfree`: an entry with a reference count of 0 is not deleted unless the entry's `RTF_UP`
flag is not set. The only time this flag is cleared is by `rtrequest` when a route is
deleted from the routing tree.
    Most routes are used in the following fashion.

- If the route is created automatically as a route to an interface when the interface
  is configured (which is typical for Ethernet interfaces, for example), then
  `rtinit` calls `rtrequest` with a command of `RTM_ADD`, creating the new entry
  and setting the reference count to 1. `rtinit` then decrements the reference
  count to 0 before returning.

  A point-to-point interface follows a similar procedure, so the route starts with a
  reference count of 0.

  If the route is created manually by the `route` command or by a routing daemon,
  a similar procedure occurs, with `route_output` calling `rtrequest` with a
  command of `RTM_ADD`, setting the reference count to 1. This is then decre-
  mented by `route_output` to 0 before it returns.

  Therefore all newly created routes start with a reference count of 0.

- When an IP datagram is sent on a socket, be it TCP or UDP, we saw that
  `ip_output` calls `rtalloc`, which calls `rtalloc1`. In Figure 19.3 we saw that
  the reference count is incremented by `rtalloc1` if the route is found.

  The located route is called a *held route,* since a pointer to the routing table entry
  is being held by the protocol, normally in a `route` structure contained within a
  protocol control block. An `rtentry` structure that is being held by someone
  else cannot be deleted, which is why `rtfree` doesn't release the structure until
  its reference count reaches 0.

- A protocol releases a held route by calling `RTFREE` or `rtfree`. We saw this in
  Figure 8.24 when `ip_output` detects a change in the destination address. We'll
  encounter it in Chapter 22 when a protocol control block that holds a route is
  released.

    Part of the confusion we'll encounter in the code that follows is that `rtalloc1` is
often called to look up a route in order to verify that a route to the destination exists, but

when the caller doesn't want to hold the route. Since `rtalloc1` increments the counter, the caller immediately decrements it.

Consider a route being deleted by `rtrequest`. The `RTF_UP` flag is cleared, and if no one is holding the route (its reference count is 0), `rtfree` should be called. But `rtfree` considers it an error for the reference count to go below 0, so `rtrequest` checks whether its reference count is less than or equal to 0, and, if so, increments it and calls `rtfree`. Normally this sets the reference count to 1 and `rtfree` decrements it to 0 and deletes the route.

19.4 `rtrequest` Function

The `rtrequest` function is the focal point for adding and deleting routing table entries. Figure 19.6 shows some of the other functions that call it.
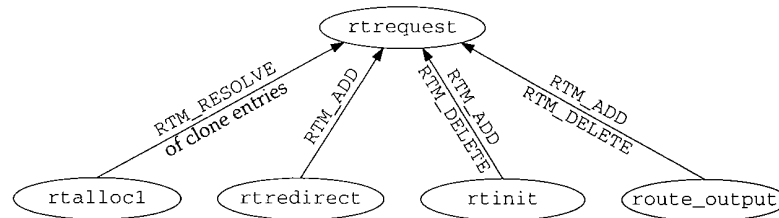

Figure 19.6 Summary of functions that call `rtrequest`.

`rtrequest` is a `switch` statement with one `case` per command: `RTM_ADD`, `RTM_DELETE`, and `RTM_RESOLVE`. Figure 19.7 shows the start of the function and the `RTM_DELETE` command.

route.c

```
290 int
291 rtrequest(req, dst, gateway, netmask, flags, ret_nrt)
292 int      req, flags;
293 struct sockaddr *dst, *gateway, *netmask;
294 struct rtentry **ret_nrt;
295 {
296     int      s = splnet();
297     int      error = 0;
298     struct rtentry *rt;
299     struct radix_node *rn;
300     struct radix_node_head *rnh;
301     struct ifaddr *ifa;
302     struct sockaddr *ndst;
303 #define senderr(x) { error = x ; goto bad; }

304     if ((rnh = rt_tables[dst->sa_family]) == 0)
305         senderr(ESRCH);
306     if (flags & RTF_HOST)
307         netmask = 0;
```

```
308        switch (req) {
309        case RTM_DELETE:
310            if ((rn = rnh->rnh_deladdr(dst, netmask, rnh)) == 0)
311                senderr(ESRCH);
312            if (rn->rn_flags & (RNF_ACTIVE | RNF_ROOT))
313                panic("rtrequest delete");
314            rt = (struct rtentry *) rn;
315            rt->rt_flags &= ~RTF_UP;
316            if (rt->rt_gwroute) {
317                rt = rt->rt_gwroute;
318                RTFREE(rt);
319                (rt = (struct rtentry *) rn)->rt_gwroute = 0;
320            }
321            if ((ifa = rt->rt_ifa) && ifa->ifa_rtrequest)
322                ifa->ifa_rtrequest(RTM_DELETE, rt, SA(0));
323            rttrash++;
324            if (ret_nrt)
325                *ret_nrt = rt;
326            else if (rt->rt_refcnt <= 0) {
327                rt->rt_refcnt++;
328                rtfree(rt);
329            }
330            break;
```
―― *route.c*

Figure 19.7   `rtrequest` function: RTM_DELETE command.

290–307   The second argument, `dst`, is a socket address structure specifying the key to be added or deleted from the routing table. The `sa_family` from this key selects the routing table. If the `flags` argument indicates a host route (instead of a route to a network), the `netmask` pointer is set to null, ignoring any value the caller may have passed.

Delete from routing tree

309–315   The `rnh_deladdr` function (`rn_delete` from Figure 18.17) deletes the entry from the routing table tree and returns a pointer to the corresponding `rtentry` structure. The RTF_UP flag is cleared.

Remove reference to gateway routing table entry

316–320   If the entry is an indirect route through a gateway, RTFREE decrements the `rt_refcnt` member of the gateway's entry and deletes it if the count reaches 0. The `rt_gwroute` pointer is set to null and `rt` is set back to point to the entry that was deleted.

Call interface request function

321–322   If an `ifa_rtrequest` function is defined for this entry, that function is called. This function is used by ARP, for example, in Chapter 21 to delete the corresponding ARP entry.

Return pointer or release reference

323–330   The `rttrash` global is incremented because the entry may not be released in the code that follows. If the caller wants the pointer to the `rtentry` structure that was

deleted from the routing tree (if `ret_nrt` is nonnull), then that pointer is returned, but the entry cannot be released: it is the caller's responsibility to call `rtfree` when it is finished with the entry. If `ret_nrt` is null, the entry can be released: if the reference count is less than or equal to 0, it is incremented, and `rtfree` is called. The `break` causes the function to return.

Figure 19.8 shows the next part of the function, which handles the `RTM_RESOLVE` command. This function is called with this command only from `rtalloc1`, when a new entry is to be created from an entry with the `RTF_CLONING` flag set.

```
                                                                        route.c
331     case RTM_RESOLVE:
332         if (ret_nrt == 0 || (rt = *ret_nrt) == 0)
333             senderr(EINVAL);
334         ifa = rt->rt_ifa;
335         flags = rt->rt_flags & ~RTF_CLONING;
336         gateway = rt->rt_gateway;
337         if ((netmask = rt->rt_genmask) == 0)
338             flags |= RTF_HOST;
339         goto makeroute;
                                                                        route.c
```

Figure 19.8   rtrequest function: RTM_RESOLVE command.

331–339   The final argument, `ret_nrt`, is used differently for this command: it contains the pointer to the entry with the `RTF_CLONING` flag set (Figure 19.2). The new entry will have the same `rt_ifa` pointer, the same flags (with the `RTF_CLONING` flag cleared), and the same `rt_gateway`. If the entry being cloned has a null `rt_genmask` pointer, the new entry has its `RTF_HOST` flag set, because it is a host route; otherwise the new entry is a network route and the network mask of the new entry is copied from the `rt_genmask` value. We give an example of cloned routes with a network mask at the end of this section. This `case` continues at the label `makeroute`, which is in the next figure.

Figure 19.9 shows the `RTM_ADD` command.

Locate corresponding interface

340–342   The function `ifa_ifwithroute` finds the appropriate local interface for the destination (`dst`), returning a pointer to its `ifaddr` structure.

Allocate memory for routing table entry

343–348   An `rtentry` structure is allocated. Recall that this structure contains both the two `radix_node` structures for the routing tree and the other routing information. The structure is zeroed and the `rt_flags` are set from the caller's flags, including the `RTF_UP` flag.

Allocate and copy gateway address

349–352   The `rt_setgate` function (Figure 19.11) allocates memory for both the routing table key (`dst`) and its `gateway`. It then copies `gateway` into the new memory and sets the pointers `rt_key`, `rt_gateway`, and `rt_gwroute`.

```
340      case RTM_ADD:                                                    route.c
341          if ((ifa = ifa_ifwithroute(flags, dst, gateway)) == 0)
342              senderr(ENETUNREACH);

343      makeroute:
344          R_Malloc(rt, struct rtentry *, sizeof(*rt));
345          if (rt == 0)
346              senderr(ENOBUFS);
347          Bzero(rt, sizeof(*rt));
348          rt->rt_flags = RTF_UP | flags;
349          if (rt_setgate(rt, dst, gateway)) {
350              Free(rt);
351              senderr(ENOBUFS);
352          }
353          ndst = rt_key(rt);
354          if (netmask) {
355              rt_maskedcopy(dst, ndst, netmask);
356          } else
357              Bcopy(dst, ndst, dst->sa_len);

358          rn = rnh->rnh_addaddr((caddr_t) ndst, (caddr_t) netmask,
359                              rnh, rt->rt_nodes);
360          if (rn == 0) {
361              if (rt->rt_gwroute)
362                  rtfree(rt->rt_gwroute);
363              Free(rt_key(rt));
364              Free(rt);
365              senderr(EEXIST);
366          }
367          ifa->ifa_refcnt++;
368          rt->rt_ifa = ifa;
369          rt->rt_ifp = ifa->ifa_ifp;
370          if (req == RTM_RESOLVE)
371              rt->rt_rmx = (*ret_nrt)->rt_rmx;     /* copy metrics */
372          if (ifa->ifa_rtrequest)
373              ifa->ifa_rtrequest(req, rt, SA(ret_nrt ? *ret_nrt : 0));
374          if (ret_nrt) {
375              *ret_nrt = rt;
376              rt->rt_refcnt++;
377          }
378          break;
379      }
380  bad:
381      splx(s);
382      return (error);
383  }
                                                                         route.c
```

Figure 19.9   rtrequest function: RTM_ADD command.

Copy destination address

353–357   The destination address (the routing table key dst) must now be copied into the memory pointed to by rn_key. If a network mask is supplied, rt_maskedcopy logically ANDs dst and netmask, forming the new key. Otherwise dst is copied into the

new key. The reason for logically ANDing `dst` and `netmask` is to guarantee that the key in the table has already been ANDed with its mask, so when a search key is compared against the key in the table only the search key needs to be ANDed. For example, the following command adds another IP address (an alias) to the Ethernet interface `le0`, with subnet 12 instead of 13:

```
bsdi $ ifconfig le0 inet 140.252.12.63 netmask 0xfffffffe0 alias
```

The problem is that we've incorrectly specified all one bits for the host ID. Nevertheless, when the key is stored in the routing table we can verify with `netstat` that the address is first logically ANDed with the mask:

```
Destination       Gateway          Flags     Refs     Use  Interface
140.252.12.32     link#1           U C        0        0  le0
```

Add entry to routing tree

358–366   The `rnh_addaddr` function (`rn_addroute` from Figure 18.17) adds this `rtentry` structure, with its destination and mask, to the routing table tree. If an error occurs, the structures are released and `EEXIST` returned (i.e., the entry is already in the routing table).

Store interface pointers

367–369   The `ifaddr` structure's reference count is incremented and the pointers to its `ifaddr` and `ifnet` structures are stored.

Copy metrics for newly cloned route

370–371   If the command was `RTM_RESOLVE` (not `RTM_ADD`), the entire metrics structure is copied from the cloned entry into the new entry. If the command was `RTM_ADD`, the caller can set the metrics after this function returns.

Call interface request function

372–373   If an `ifa_rtrequest` function is defined for this entry, that function is called. ARP uses this to perform additional processing for both the `RTM_ADD` and `RTM_RESOLVE` commands (Section 21.13).

Return pointer and increment reference count

374–378   If the caller wants a copy of the pointer to the new structure, it is returned through `ret_nrt` and the `rt_refcnt` reference count is incremented from 0 to 1.

Example: Cloned Routes with Network Masks

oute.c

o the
logi-
o the

The only use of the `rt_genmask` value is with cloned routes created by the `RTM_RESOLVE` command in `rtrequest`. If an `rt_genmask` pointer is nonnull, then the socket address structure pointed to by this pointer becomes the network mask of the newly created route. In our routing table, Figure 18.2, the cloned routes are for the local Ethernet and for multicast addresses. The following example from [Sklower 1991] provides a different use of cloned routes. Another example is in Exercise 19.2.

Consider a class B network, say 128.1, that is behind a point-to-point link. The subnet mask is `0xffffff00`, the typical value that uses 8 bits for the subnet ID and 8 bits

for the host ID. We need a routing table entry for all possible 254 subnets, with a gateway value of a router that is directly connected to our host and that knows how to reach the link to which the 128.1 network is connected.

The easiest solution, assuming the gateway router isn't our default router, is a single entry with a destination of 128.1.0.0 and a mask of 0xffff0000. Assume, however, that the topology of the 128.1 network is such that each of the possible 254 subnets can have different operational characteristics: RTTs, MTUs, delays, and so on. If a separate routing table entry were used for each subnet, we would see that whenever a connection is closed, TCP would update the routing table entry with statistics about that route—its RTT, RTT variance, and so on (Figure 27.3). While we could create up to 254 entries by hand using the route command, one per subnet, a better solution is to use the cloning feature.

One entry is created by the system administrator with a destination of 128.1.0.0 and a network mask of 0xffff0000. Additionally, the RTF_CLONING flag is set and the genmask is set to 0xffffff00, which differs from the network mask. If the routing table is searched for 128.1.2.3, and an entry does not exist for the 128.1.2 subnet, the entry for 128.1 with the mask of 0xffff0000 is the best match. A new entry is created (since the RTF_CLONING flag is set) with a destination of 128.1.2 and a network mask of 0xffffff00 (the genmask value). The next time any host on this subnet is referenced, say 128.1.2.88, it will match this newly created entry.

19.5   rt_setgate **Function**

Each leaf in the routing tree has a key (rt_key, which is just the rn_key member of the radix_node structure contained at the beginning of the rtentry structure), and an associated gateway (rt_gateway). Both are socket address structures specified when the routing table entry is created. Memory is allocated for both structures by rt_setgate, as shown in Figure 19.10.

This example shows two of the entries from Figure 18.2, the ones with keys of 127.0.0.1 and 140.252.13.33. The former's gateway member points to an Internet socket address structure, while the latter's points to a data-link socket address structure that contains an Ethernet address. The former was entered into the routing table by the route system when the system was initialized, and the latter was created by ARP.

We purposely show the two structures pointed to by rt_key one right after the other, since they are allocated together by rt_setgate, which we show in Figure 19.11.

Set lengths from socket address structures

384–391   dlen is the length of the destination socket address structure, and glen is the length of the gateway socket address structure. The ROUNDUP macro rounds the value up to the next multiple of 4 bytes, but the size of most socket address structures is already a multiple of 4.

rtentry{}
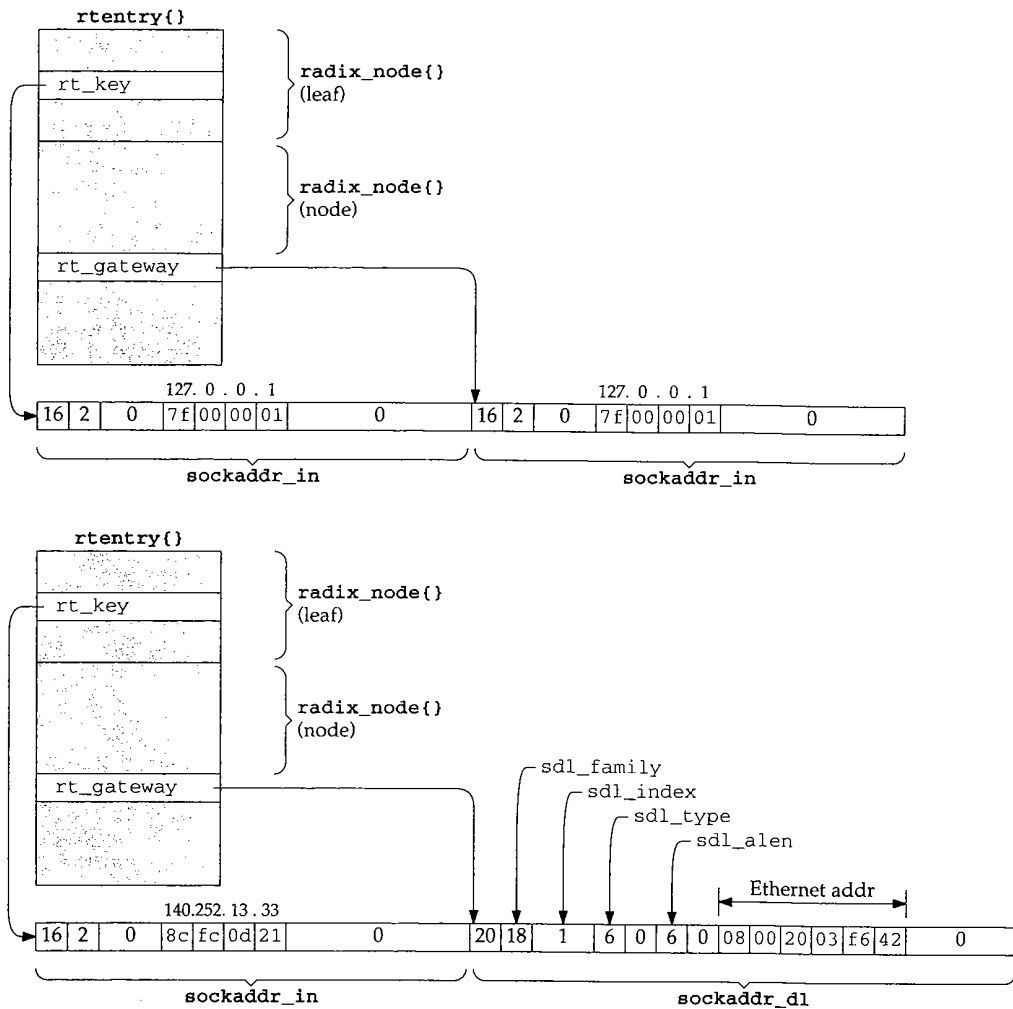
Figure 19.10   Example of routing table keys and associated gateways.

Allocate memory

392–397   If memory has not been allocated for this routing table key and gateway yet, or if glen is greater than the current size of the structure pointed to by rt_gateway, a new piece of memory is allocated and rn_key is set to point to the new memory.

Use memory already allocated for key and gateway

398–401   An adequately sized piece of memory is already allocated for the key and gateway, so new is set to point to this existing memory.

```
                                                                            route.c
384  int
385  rt_setgate(rt0, dst, gate)
386  struct rtentry *rt0;
387  struct sockaddr *dst, *gate;
388  {
389      caddr_t new, old;
390      int      dlen = ROUNDUP(dst->sa_len), glen = ROUNDUP(gate->sa_len);
391      struct rtentry *rt = rt0;

392      if (rt->rt_gateway == 0 || glen > ROUNDUP(rt->rt_gateway->sa_len)) {
393          old = (caddr_t) rt_key(rt);
394          R_Malloc(new, caddr_t, dlen + glen);
395          if (new == 0)
396              return 1;
397          rt->rt_nodes->rn_key = new;
398      } else {
399          new = rt->rt_nodes->rn_key;
400          old = 0;
401      }
402      Bcopy(gate, (rt->rt_gateway = (struct sockaddr *) (new + dlen)), glen);
403      if (old) {
404          Bcopy(dst, new, dlen);
405          Free(old);
406      }
407      if (rt->rt_gwroute) {
408          rt = rt->rt_gwroute;
409          RTFREE(rt);
410          rt = rt0;
411          rt->rt_gwroute = 0;
412      }
413      if (rt->rt_flags & RTF_GATEWAY) {
414          rt->rt_gwroute = rtalloc1(gate, 1);
415      }
416      return 0;
417  }
                                                                            route.c
```

Figure 19.11   rt_setgate function.

Copy new gateway

402   The new gateway structure is copied and `rt_gateway` is set to point to the socket address structure.

Copy key from old memory to new memory

403–406   If a new piece of memory was allocated, the routing table key (`dst`) is copied right before the gateway field that was just copied. The old piece of memory is released.

Release gateway routing pointer

407–412   If the routing table entry contains a nonnull `rt_gwroute` pointer, that structure is released by RTFREE and the `rt_gwroute` pointer is set to null.

Locate and store new gateway routing pointer

413–415   If the routing table entry is an indirect route, rtalloc1 locates the entry for the new gateway, which is stored in rt_gwroute. If an invalid gateway is specified for an indirect route, an error is not returned by rt_setgate, but the rt_gwroute pointer will be null.

19.6 rtinit **Function**

There are four calls to rtinit from the Internet protocols to add or delete routes associated with interfaces.

- in_control calls rtinit twice when the destination address of a point-to-point interface is set (Figure 6.21). The first call specifies RTM_DELETE to delete any existing route to the destination; the second call specifies RTM_ADD to add the new route.

- in_ifinit calls rtinit to add a network route for a broadcast network or a host route for a point-to-point link (Figure 6.19). If the route is for an Ethernet interface, the RTF_CLONING flag is automatically set by in_ifinit.

- in_ifscrub calls rtinit to delete an existing route for an interface.

Figure 19.12 shows the first part of the rtinit function. The cmd argument is always RTM_ADD or RTM_DELETE.

Get destination address for route

452   If the route is to a host, the destination address is the other end of the point-to-point link. Otherwise we're dealing with a network route and the destination address is the unicast address of the interface (masked with ifa_netmask).

Mask network address with network mask

453–459   If a route is being deleted, the destination must be looked up in the routing table to locate its routing table entry. If the route being deleted is a network route and the interface has an associated network mask, an mbuf is allocated and the destination address is copied into the mbuf by rt_maskedcopy, logically ANDing the caller's address with the mask. dst is set to point to the masked copy in the mbuf, and that is the destination looked up in the next step.

Search for routing table entry

460–469   rtalloc1 searches the routing table for the destination address. If the entry is found, its reference count is decremented (since rtalloc1 incremented the reference count). If the pointer to the interface's ifaddr in the routing table does not equal the caller's argument, an error is returned.

Process request

470–473   rtrequest executes the command, either RTM_ADD or RTM_DELETE. When it returns, if an mbuf was allocated earlier, it is released.

route.c

```
441 int
442 rtinit(ifa, cmd, flags)
443 struct ifaddr *ifa;
444 int     cmd, flags;
445 {
446     struct rtentry *rt;
447     struct sockaddr *dst;
448     struct sockaddr *deldst;
449     struct mbuf *m = 0;
450     struct rtentry *nrt = 0;
451     int     error;

452     dst = flags & RTF_HOST ? ifa->ifa_dstaddr : ifa->ifa_addr;
453     if (cmd == RTM_DELETE) {
454         if ((flags & RTF_HOST) == 0 && ifa->ifa_netmask) {
455             m = m_get(M_WAIT, MT_SONAME);
456             deldst = mtod(m, struct sockaddr *);
457             rt_maskedcopy(dst, deldst, ifa->ifa_netmask);
458             dst = deldst;
459         }
460         if (rt = rtalloc1(dst, 0)) {
461             rt->rt_refcnt--;
462             if (rt->rt_ifa != ifa) {
463                 if (m)
464                     (void) m_free(m);
465                 return (flags & RTF_HOST ? EHOSTUNREACH
466                         : ENETUNREACH);
467             }
468         }
469     }
470     error = rtrequest(cmd, dst, ifa->ifa_addr, ifa->ifa_netmask,
471                       flags | ifa->ifa_flags, &nrt);
472     if (m)
473         (void) m_free(m);
```

route.c

Figure 19.12   `rtinit` function: call `rtrequest` to handle command.

Figure 19.13 shows the second half of `rtinit`.

Generate routing message on successful delete

474–480   If a route was deleted, and `rtrequest` returned 0 along with a pointer to the `rtentry` structure that was deleted (in `nrt`), a routing socket message is generated by `rt_newaddrmsg`. If the reference count is less than or equal to 0, it is incremented and the route is released by `rtfree`.

Successful add

481–482   If a route was added, and `rtrequest` returned 0 along with a pointer to the `rtentry` structure that was added (in `nrt`), the reference count is decremented (since `rtrequest` incremented it).

```
                                                              ──── route.c
474    if (cmd == RTM_DELETE && error == 0 && (rt = nrt)) {
475        rt_newaddrmsg(cmd, ifa, error, nrt);
476        if (rt->rt_refcnt <= 0) {
477            rt->rt_refcnt++;
478            rtfree(rt);
479        }
480    }
481    if (cmd == RTM_ADD && error == 0 && (rt = nrt)) {
482        rt->rt_refcnt--;
483        if (rt->rt_ifa != ifa) {
484            printf("rtinit: wrong ifa (%x) was (%x)\n", ifa,
485                rt->rt_ifa);
486            if (rt->rt_ifa->ifa_rtrequest)
487                rt->rt_ifa->ifa_rtrequest(RTM_DELETE, rt, SA(0));
488            IFAFREE(rt->rt_ifa);
489            rt->rt_ifa = ifa;
490            rt->rt_ifp = ifa->ifa_ifp;
491            ifa->ifa_refcnt++;
492            if (ifa->ifa_rtrequest)
493                ifa->ifa_rtrequest(RTM_ADD, rt, SA(0));
494        }
495        rt_newaddrmsg(cmd, ifa, error, nrt);
496    }
497    return (error);
498 }
                                                              ──── route.c
```

Figure 19.13   rtinit function: second half.

Incorrect interface

483–494   If the pointer to the interface's ifaddr in the new routing table entry does not equal the caller's argument, an error occurred. Recall that rtrequest determines the ifa pointer that is stored in the new entry by calling ifa_ifwithroute (Figure 19.9). When this error occurs the following steps take place: an error message is output to the console, the ifa_rtrequest function is called (if defined) with a command of RTM_DELETE, the ifaddr structure is released, the rt_ifa pointer is set to the value specified by the caller, the interface reference count is incremented, and the new interface's ifa_rtrequest function (if defined) is called with a command of RTM_ADD.

Generate routing message

495   A routing socket message is generated by rt_newaddrmsg for the RTM_ADD command.

19.7   rtredirect Function

When an ICMP redirect is received, icmp_input calls rtredirect and then calls pfctlinput (Figure 11.27). This latter function calls udp_ctlinput and tcp_ctlinput, which go through all the UDP and TCP protocol control blocks. If the

PCB is connected to the foreign address that has been redirected, and if the PCB holds a route to that foreign address, the route is released by rtfree. The next time any of these control blocks is used to send an IP datagram to that foreign address, rtalloc will be called and the destination will be looked up in the routing table, possibly finding a new (redirected) route.

The purpose of rtredirect, the first half of which is shown in Figure 19.14, is to validate the information in the redirect, update the routing table immediately, and then generate a routing socket message.

———————————————————————————— route.c

```
147 int
148 rtredirect(dst, gateway, netmask, flags, src, rtp)
149 struct sockaddr *dst, *gateway, *netmask, *src;
150 int      flags;
151 struct rtentry **rtp;
152 {
153     struct rtentry *rt;
154     int      error = 0;
155     short    *stat = 0;
156     struct rt_addrinfo info;
157     struct ifaddr *ifa;

158     /* verify the gateway is directly reachable */
159     if ((ifa = ifa_ifwithnet(gateway)) == 0) {
160         error = ENETUNREACH;
161         goto out;
162     }
163     rt = rtalloc1(dst, 0);
164     /*
165      * If the redirect isn't from our current router for this dst,
166      * it's either old or wrong.  If it redirects us to ourselves,
167      * we have a routing loop, perhaps as a result of an interface
168      * going down recently.
169      */
170 #define equal(a1, a2) (bcmp((caddr_t)(a1), (caddr_t)(a2), (a1)->sa_len) == 0)
171     if (!(flags & RTF_DONE) && rt &&
172          (!equal(src, rt->rt_gateway) || rt->rt_ifa != ifa))
173         error = EINVAL;
174     else if (ifa_ifwithaddr(gateway))
175         error = EHOSTUNREACH;
176     if (error)
177         goto done;
178     /*
179      * Create a new entry if we just got back a wildcard entry
180      * or if the lookup failed.  This is necessary for hosts
181      * which use routing redirects generated by smart gateways
182      * to dynamically build the routing tables.
183      */
184     if ((rt == 0) || (rt_mask(rt) && rt_mask(rt)->sa_len < 2))
185         goto create;
```
———————————————————————————— route.c

Figure 19.14   rtredirect function: validate received redirect.

147–157   The arguments are dst, the destination IP address of the datagram that caused the redirect (HD in Figure 8.18); gateway, the IP address of the router to use as the new gateway field for the destination (R2 in Figure 8.18); netmask, which is a null pointer; flags, which is RTF_GATEWAY and RTF_HOST; src, the IP address of the router that sent the redirect (R1 in Figure 8.18); and rtp, which is a null pointer. We indicate that netmask and rtp are both null pointers when called by icmp_input, but these arguments might be nonnull when called from other protocols.

New gateway must be directly connected

158–162   The new gateway must be directly connected or the redirect is invalid.

Locate routing table entry for destination and validate redirect

163–177   rtalloc1 searches the routing table for a route to the destination. The following conditions must all be true, or the redirect is invalid and an error is returned. Notice that icmp_input ignores any error return from rtredirect. ICMP does not generate an error in response to an invalid redirect—it just ignores it.

- the RTF_DONE flag must not be set;
- rtalloc must have located a routing table entry for dst;
- the address of the router that sent the redirect (src) must equal the current rt_gateway for the destination;
- the interface for the new gateway (the ifa returned by ifa_ifwithnet) must equal the current interface for the destination (rt_ifa), that is, the new gateway must be on the same network as the current gateway; and
- the new gateway cannot redirect this host to itself, that is, there cannot exist an attached interface with a unicast address or a broadcast address equal to gateway.

Must create a new route

178–185   If a route to the destination was not found, or if the routing table entry that was located is the default route, a new entry is created for the destination. As the comment indicates, a host with access to multiple routers can use this feature to learn of the correct router when the default is not correct. The test for finding the default route is whether the routing table entry has an associated mask and if the length field of the mask is less than 2, since the mask for the default route is rn_zeros (Figure 18.35).

Figure 19.15 shows the second half of this function.

Create new host route

186–195   If the current route to the destination is a network route and the redirect is a host redirect and not a network redirect, a new host route is created for the destination and the existing network route is left alone. We mentioned that the flags argument always specifies RTF_HOST since the Net/3 ICMP considers all received redirects as host redirects.

─── *route.c*
```
186     /*
187      * Don't listen to the redirect if it's
188      * for a route to an interface.
189      */
190     if (rt->rt_flags & RTF_GATEWAY) {
191         if (((rt->rt_flags & RTF_HOST) == 0) && (flags & RTF_HOST)) {
192             /*
193              * Changing from route to net => route to host.
194              * Create new route, rather than smashing route to net.
195              */
196           create:
197             flags |= RTF_GATEWAY | RTF_DYNAMIC;
198             error = rtrequest((int) RTM_ADD, dst, gateway,
199                                 netmask, flags,
200                                 (struct rtentry **) 0);
201             stat = &rtstat.rts_dynamic;
202         } else {
203             /*
204              * Smash the current notion of the gateway to
205              * this destination.  Should check about netmask!!!
206              */
207             rt->rt_flags |= RTF_MODIFIED;
208             flags |= RTF_MODIFIED;
209             stat = &rtstat.rts_newgateway;
210             rt_setgate(rt, rt_key(rt), gateway);
211         }
212     } else
213         error = EHOSTUNREACH;
214 done:
215     if (rt) {
216         if (rtp && !error)
217             *rtp = rt;
218         else
219             rtfree(rt);
220     }
221 out:
222     if (error)
223         rtstat.rts_badredirect++;
224     else if (stat != NULL)
225         (*stat)++;

226     bzero((caddr_t) & info, sizeof(info));
227     info.rti_info[RTAX_DST] = dst;
228     info.rti_info[RTAX_GATEWAY] = gateway;
229     info.rti_info[RTAX_NETMASK] = netmask;
230     info.rti_info[RTAX_AUTHOR] = src;
231     rt_missmsg(RTM_REDIRECT, &info, flags, error);
232 }
```
─── *route.c*

Figure 19.15 rtredirect function: second half.

Create route

196-201   rtrequest creates the new route, setting the RTF_GATEWAY and RTF_DYNAMIC flags. The netmask argument is a null pointer, since the new route is a host route with an implied mask of all one bits. stat points to a counter that is incremented later.

Modify existing host route

202-211   This code is executed when the current route to the destination is already a host route. A new entry is not created, but the existing entry is modified. The RTF_MODIFIED flag is set and rt_setgate changes the rt_gateway field of the routing table entry to the new gateway address.

Ignore if destination is directly connected

212-213   If the current route to the destination is a direct route (the RTF_GATEWAY flag is not set), it is a redirect for a destination that is already directly connected. EHOSTUNREACH is returned.

Return pointer and increment statistic

214-225   If a routing table entry was located, it is either returned (if rtp is nonnull and there were no errors) or released by rtfree. The appropriate statistic is incremented.

Generate routing message

226-232   An rt_addrinfo structure is cleared and a routing socket message is generated by rt_missmsg. This message is sent by raw_input to any processes interested in the redirect.

19.8 Routing Message Structures

Routing messages consist of a fixed-length header followed by up to eight socket address structures. The fixed-length header is one of the following three structures:

- rt_msghdr
- if_msghdr
- ifa_msghdr

Figure 18.11 provided an overview of which functions generated the different messages and Figure 18.9 showed which structure is used by each message type. The first three members of the three structures have the same data type and meaning: the message length, version, and type. This allows the receiver of the message to decode the message. Also, each structure has a member that encodes which of the eight potential socket address structures follow the structure (a bitmask): the rtm_addrs, ifm_addrs, and ifam_addrs members.

Figure 19.16 shows the most common of the structures, rt_msghdr. The RTM_IFINFO message uses an if_msghdr structure, shown in Figure 19.17. The RTM_NEWADDR and RTM_DELADDR messages use an ifa_msghdr structure, shown in Figure 19.18.

route.h
```
139 struct rt_msghdr {
140     u_short rtm_msglen;        /* to skip over non-understood messages */
141     u_char  rtm_version;       /* future binary compatibility */
142     u_char  rtm_type;          /* message type */

143     u_short rtm_index;         /* index for associated ifp */
144     int     rtm_flags;         /* flags, incl. kern & message, e.g. DONE */
145     int     rtm_addrs;         /* bitmask identifying sockaddrs in msg */
146     pid_t   rtm_pid;           /* identify sender */
147     int     rtm_seq;           /* for sender to identify action */
148     int     rtm_errno;         /* why failed */
149     int     rtm_use;           /* from rtentry */
150     u_long  rtm_inits;         /* which metrics we are initializing */
151     struct rt_metrics rtm_rmx; /* metrics themselves */
152 };
```
route.h

Figure 19.16 rt_msghdr structure.

if.h
```
235 struct if_msghdr {
236     u_short ifm_msglen;        /* to skip over non-understood messages */
237     u_char  ifm_version;       /* future binary compatability */
238     u_char  ifm_type;          /* message type */

239     int     ifm_addrs;         /* like rtm_addrs */
240     int     ifm_flags;         /* value of if_flags */
241     u_short ifm_index;         /* index for associated ifp */
242     struct if_data ifm_data;   /* statistics and other data about if */
243 };
```
if.h

Figure 19.17 if_msghdr structure.

if.h
```
248 struct ifa_msghdr {
249     u_short ifam_msglen;       /* to skip over non-understood messages */
250     u_char  ifam_version;      /* future binary compatability */
251     u_char  ifam_type;         /* message type */

252     int     ifam_addrs;        /* like rtm_addrs */
253     int     ifam_flags;        /* value of ifa_flags */
254     u_short ifam_index;        /* index for associated ifp */
255     int     ifam_metric;       /* value of ifa_metric */
256 };
```
if.h

Figure 19.18 ifa_msghdr structure.

Note that the first three members across the three different structures have the same data types and meanings.

The three variables rtm_addrs, ifm_addrs, and ifam_addrs are bitmasks defining which socket address structures follow the header. Figure 19.19 shows the constants used with these bitmasks.

Bitmask		Array index		Name in rtsock.c	Description
Constant	Value	Constant	Value		
RTA_DST	0x01	RTAX_DST	0	dst	destination socket address structure
RTA_GATEWAY	0x02	RTAX_GATEWAY	1	gate	gateway socket address structure
RTA_NETMASK	0x04	RTAX_NETMASK	2	netmask	netmask socket address structure
RTA_GENMASK	0x08	RTAX_GENMASK	3	genmask	cloning mask socket address structure
RTA_IFP	0x10	RTAX_IFP	4	ifpaddr	interface name socket address structure
RTA_IFA	0x20	RTAX_IFA	5	ifaaddr	interface address socket address structure
RTA_AUTHOR	0x40	RTAX_AUTHOR	6		socket address structure for author of redirect
RTA_BRD	0x80	RTAX_BRD	7	brdaddr	broadcast or point-to-point destination address
		RTAX_MAX	8		#elements in an rti_info[] array

Figure 19.19   Constants used to refer to members of rti_info array.

The bitmask value is always the constant 1 left shifted by the number of bits specified by the array index. For example, 0x20 (RTA_IFA) is 1 left shifted by five bits (RTAX_IFA). We'll see this fact used in the code.

The socket address structures that are present always occur in order of increasing array index, one right after the other. For example, if the bitmask is 0x87, the first socket address structure contains the destination, followed by the gateway, followed by the network mask, followed by the broadcast address.

The array indexes in Figure 19.19 are used within the kernel to refer to its rt_addrinfo structure, shown in Figure 19.20. This structure holds the same bitmask that we described, indicating which addresses are present, and pointers to those socket address structures.

-- *route.h*
```
199 struct rt_addrinfo {
200     int     rti_addrs;              /* bitmask, same as rtm_addrs */
201     struct sockaddr *rti_info[RTAX_MAX];
202 };
```
-- *route.h*

Figure 19.20   rt_addrinfo structure: encode which addresses are present and pointers to them.

For example, if the RTA_GATEWAY bit is set in the rti_addrs member, then the member rti_info[RTAX_GATEWAY] is a pointer to a socket address structure containing the gateway's address. In the case of the Internet protocols, the socket address structure is a sockaddr_in containing the gateway's IP address.

The fifth column in Figure 19.19 shows the names used for the corresponding members of an rti_info array throughout the file rtsock.c. These definitions look like

        #define dst   info.rti_info[RTAX_DST]

We'll encounter these names in many of the source files later in this chapter. The RTAX_AUTHOR element is not assigned a name because it is never passed from a process to the kernel.

We've already encountered this rt_addrinfo structure twice: in rtalloc1 (Figure 19.2) and rtredirect (Figure 19.14). Figure 19.21 shows the format of this

structure when built by `rtalloc1`, after a routing table lookup fails, when `rt_missmsg` is called.
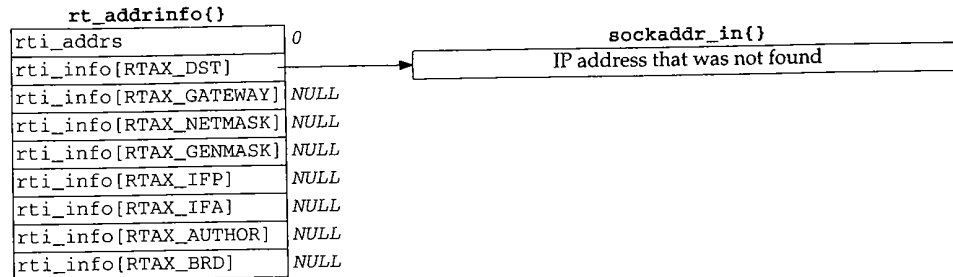
rt_addrinfo{}

rti_addrs	0
rti_info[RTAX_DST]	
rti_info[RTAX_GATEWAY]	NULL
rti_info[RTAX_NETMASK]	NULL
rti_info[RTAX_GENMASK]	NULL
rti_info[RTAX_IFP]	NULL
rti_info[RTAX_IFA]	NULL
rti_info[RTAX_AUTHOR]	NULL
rti_info[RTAX_BRD]	NULL

sockaddr_in{}

IP address that was not found

Figure 19.21   `rt_addrinfo` structure passed by `rtalloc1` to `rt_missmsg`.

All the unused pointers are null because the structure is set to 0 before it is used. Also note that the `rti_addrs` member is not initialized with the appropriate bitmask because when this structure is used within the kernel, a null pointer in the `rti_info` array indicates a nonexistent socket address structure. The bitmask is needed only for messages between a process and the kernel.

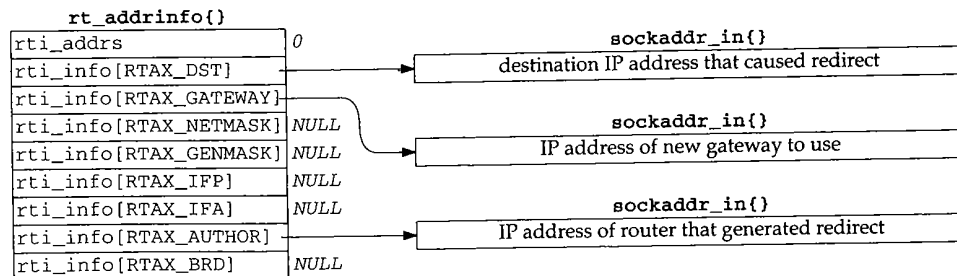Figure 19.22 shows the format of the structure built by `rtredirect` when it calls `rt_missmsg`.

rt_addrinfo{}

rti_addrs	0
rti_info[RTAX_DST]	
rti_info[RTAX_GATEWAY]	
rti_info[RTAX_NETMASK]	NULL
rti_info[RTAX_GENMASK]	NULL
rti_info[RTAX_IFP]	NULL
rti_info[RTAX_IFA]	NULL
rti_info[RTAX_AUTHOR]	
rti_info[RTAX_BRD]	NULL

sockaddr_in{}

destination IP address that caused redirect

sockaddr_in{}

IP address of new gateway to use

sockaddr_in{}

IP address of router that generated redirect

Figure 19.22   `rt_addrinfo` structure passed by `rtredirect` to `rt_missmsg`.

The following sections show how these structures are placed into the messages sent to a process.

Figure 19.23 shows the `route_cb` structure, which we'll encounter in the following sections. It contains four counters; one each for the IP, XNS, and OSI protocols, and an "any" counter. Each counter is the number of routing sockets currently in existence for that domain.

203–208   By keeping track of the number of routing socket listeners, the kernel avoids building a routing message and calling `raw_input` to send the message when there aren't any processes waiting for a message.

route.h
```
203 struct route_cb {
204     int      ip_count;            /* IP */
205     int      ns_count;            /* XNS */
206     int      iso_count;           /* ISO */
207     int      any_count;           /* sum of above three counters */
208 };
```
route.h

Figure 19.23 route_cb structure: counters of routing socket listeners.

19.9 `rt_missmsg` Function

The function `rt_missmsg`, shown in Figure 19.24, takes the structures shown in Figures 19.21 and 19.22, calls `rt_msg1` to build a corresponding variable-length message for a process in an mbuf chain, and then calls `raw_input` to pass the mbuf chain to all appropriate routing sockets.

rtsock.c
```
516 void
517 rt_missmsg(type, rtinfo, flags, error)
518 int      type, flags, error;
519 struct rt_addrinfo *rtinfo;
520 {
521     struct rt_msghdr *rtm;
522     struct mbuf *m;
523     struct sockaddr *sa = rtinfo->rti_info[RTAX_DST];

524     if (route_cb.any_count == 0)
525         return;

526     m = rt_msg1(type, rtinfo);
527     if (m == 0)
528         return;

529     rtm = mtod(m, struct rt_msghdr *);
530     rtm->rtm_flags = RTF_DONE | flags;
531     rtm->rtm_errno = error;
532     rtm->rtm_addrs = rtinfo->rti_addrs;

533     route_proto.sp_protocol = sa ? sa->sa_family : 0;
534     raw_input(m, &route_proto, &route_src, &route_dst);
535 }
```
*rtsock.c*

Figure 19.24 rt_missmsg function.

516–525   If there aren't any routing socket listeners, the function returns immediately.

Build message in mbuf chain

526–528   `rt_msg1` (Section 19.12) builds the appropriate message in an mbuf chain, and returns the pointer to the chain. Figure 19.25 shows an example of the resulting mbuf chain, using the `rt_addrinfo` structure from Figure 19.22. The information needs to be in an mbuf chain because `raw_input` calls `sbappendaddr` to append the mbuf chain to a socket's receive buffer.
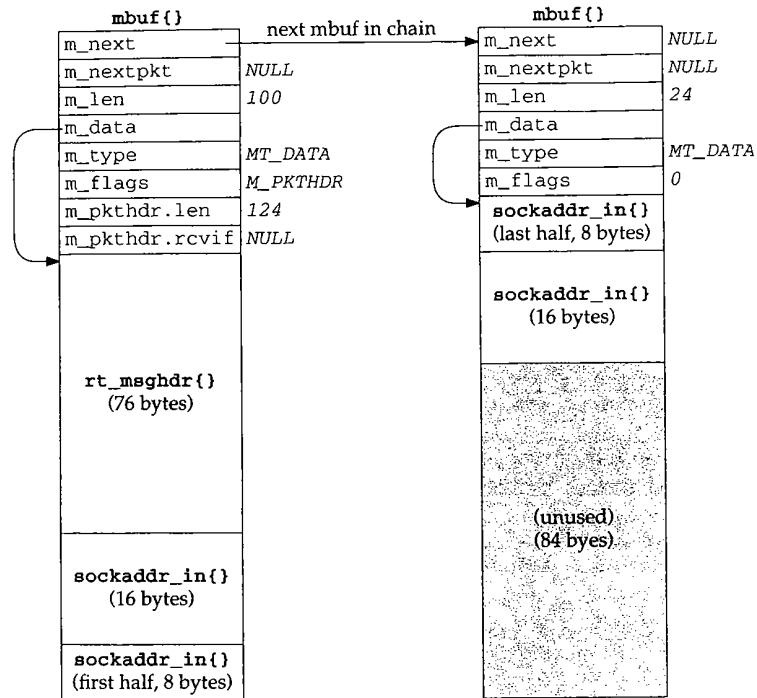
Figure 19.25   Mbuf chain built by `rt_msg1` corresponding to Figure 19.22.

Finish building message

529–532   The two members `rtm_flags` and `rtm_errno` are set to the values passed by the caller. The `rtm_addrs` member is copied from the `rti_addrs` value. We showed this value as 0 in Figures 19.21 and 19.22, but `rt_msg1` calculates and stores the appropriate bitmask, based on which pointers in the `rti_info` array are nonnull.

Set protocol of message, call `raw_input`

533–534   The final three arguments to `raw_input` specify the protocol, source, and destination of the routing message. These three structures are initialized as

```
struct  sockaddr  route_dst = { 2, PF_ROUTE, };
struct  sockaddr  route_src = { 2, PF_ROUTE, };
struct  sockproto route_proto = { PF_ROUTE, };
```

The first two structures are never modified by the kernel. The `sockproto` structure, shown in Figure 19.26, is one we haven't seen before.

―― *socket.h*
```
128 struct sockproto {
129     u_short sp_family;      /* address family */
130     u_short sp_protocol;    /* protocol */
131 };
```
―― *socket.h*

Figure 19.26   `sockproto` structure.

The family is never changed from its initial value of PF_ROUTE, but the protocol is set each time raw_input is called. When a process creates a routing socket by calling socket, the third argument (the protocol) specifies the protocol in which the process is interested. The caller of raw_input sets the sp_protocol member of the route_proto structure to the protocol of the routing message. In the case of rt_missmsg, it is set to the sa_family of the destination socket address structure (if specified by the caller), which in Figures 19.21 and 19.22 would be AF_INET.

19.10 rt_ifmsg Function

In Figure 4.30 we saw that if_up and if_down both call rt_ifmsg, shown in Figure 19.27, to generate a routing socket message when an interface goes up or down.

```
                                                                ─────── rtsock.c
540 void
541 rt_ifmsg(ifp)
542 struct ifnet *ifp;
543 {
544     struct if_msghdr *ifm;
545     struct mbuf *m;
546     struct rt_addrinfo info;

547     if (route_cb.any_count == 0)
548         return;

549     bzero((caddr_t) & info, sizeof(info));
550     m = rt_msg1(RTM_IFINFO, &info);
551     if (m == 0)
552         return;

553     ifm = mtod(m, struct if_msghdr *);
554     ifm->ifm_index = ifp->if_index;
555     ifm->ifm_flags = ifp->if_flags;
556     ifm->ifm_data = ifp->if_data;    /* structure assignment */
557     ifm->ifm_addrs = 0;

558     route_proto.sp_protocol = 0;
559     raw_input(m, &route_proto, &route_src, &route_dst);
560 }
                                                                ─────── rtsock.c
```

Figure 19.27 rt_ifmsg function.

547–548   If there aren't any routing socket listeners, the function returns immediately.

Build message in mbuf chain

549–552   An rt_addrinfo structure is set to 0 and rt_msg1 builds an appropriate message in an mbuf chain. Notice that all socket address pointers in the rt_addrinfo structure are null, so only the fixed-length if_msghdr structure becomes the routing message; there are no addresses.

Finish building message

553–557   The interface's index, flags, and `if_data` structure are copied into the message in the mbuf and the `ifm_addrs` bitmask is set to 0.

Set protocol of message, call `raw_input`

558–559   The protocol of the routing message is set to 0 because this message can apply to all protocol suites. It is a message about an interface, not about some specific destination. `raw_input` delivers the message to the appropriate listeners.

19.11 `rt_newaddrmsg` Function

In Figure 19.13 we saw that `rtinit` calls `rt_newaddrmsg` with a command of `RTM_ADD` or `RTM_DELETE` when an interface has an address added or deleted. Figure 19.28 shows the first half of the function.

―― rtsock.c
```
569 void
570 rt_newaddrmsg(cmd, ifa, error, rt)
571 int      cmd, error;
572 struct ifaddr *ifa;
573 struct rtentry *rt;
574 {
575      struct rt_addrinfo info;
576      struct sockaddr *sa;
577      int      pass;
578      struct mbuf *m;
579      struct ifnet *ifp = ifa->ifa_ifp;

580      if (route_cb.any_count == 0)
581          return;

582      for (pass = 1; pass < 3; pass++) {
583          bzero((caddr_t) & info, sizeof(info));
584          if ((cmd == RTM_ADD && pass == 1) ||
585              (cmd == RTM_DELETE && pass == 2)) {
586              struct ifa_msghdr *ifam;
587              int      ncmd = cmd == RTM_ADD ? RTM_NEWADDR : RTM_DELADDR;

588              ifaaddr = sa = ifa->ifa_addr;
589              ifpaddr = ifp->if_addrlist->ifa_addr;
590              netmask = ifa->ifa_netmask;
591              brdaddr = ifa->ifa_dstaddr;
592              if ((m = rt_msg1(ncmd, &info)) == NULL)
593                  continue;
594              ifam = mtod(m, struct ifa_msghdr *);
595              ifam->ifam_index = ifp->if_index;
596              ifam->ifam_metric = ifa->ifa_metric;
597              ifam->ifam_flags = ifa->ifa_flags;
598              ifam->ifam_addrs = info.rti_addrs;
599          }
```
―― rtsock.c

Figure 19.28   `rt_newaddrmsg` function: first half: create `ifa_msghdr` message.

580–581 If there aren't any routing socket listeners, the function returns immediately.

Generate two routing messages

582 The `for` loop iterates twice because two messages are generated. If the command is RTM_ADD, the first message is of type RTM_NEWADDR and the second message is of type RTM_ADD. If the command is RTM_DELETE, the first message is of type RTM_DELETE and the second message is of type RTM_DELADDR. The RTM_NEWADDR and RTM_DELADDR messages are built from an `ifa_msghdr` structure, while the RTM_ADD and RTM_DELETE messages are built from an `rt_msghdr` structure. The function generates two messages because one message provides information about the interface and the other about the addresses.

583 An `rt_addrinfo` structure is set to 0.

Generate message with up to four addresses

588–591 Pointers to four socket address structures containing information about the interface address that has been added or deleted are stored in the `rti_info` array. Recall from Figure 19.19 that `ifaaddr`, `ifpaddr`, `netmask`, and `brdaddr` reference elements in the `rti_info` array in `info`. `rt_msg1` builds the appropriate message in an mbuf chain. Notice that `sa` is set to point to the `ifa_addr` structure, and we'll see at the end of the function that the family of this socket address structure becomes the protocol of the routing message.

Remaining members of the `ifa_msghdr` structure are filled in with the interface's index, metric, and flags, along with the bitmask set by `rt_msg1`.

Figure 19.29 shows the second half of `rt_newaddrmsg`, which creates an `rt_msghdr` message with information about the routing table entry that was added or deleted.

Build message

600–609 Pointers to three socket address structures are stored in the `rti_info` array: the `rt_mask`, `rt_key`, and `rt_gateway` structures. `sa` is set to point to the destination address, and its family becomes the protocol of the routing message. `rt_msg1` builds the appropriate message in an mbuf chain.

Additional fields in the `rt_msghdr` structure are filled in, including the bitmask set by `rt_msg1`.

Set protocol of message, call `raw_input`

616–619 The protocol of the routing message is set and `raw_input` passes the message to the appropriate listeners. The function returns after two iterations through the loop.

—— *rtsock.c*
```
600            if ((cmd == RTM_ADD && pass == 2) ||
601                (cmd == RTM_DELETE && pass == 1)) {
602                struct rt_msghdr *rtm;

603                if (rt == 0)
604                    continue;
605                netmask = rt_mask(rt);
606                dst = sa = rt_key(rt);
607                gate = rt->rt_gateway;
608                if ((m = rt_msg1(cmd, &info)) == NULL)
609                    continue;
610                rtm = mtod(m, struct rt_msghdr *);
611                rtm->rtm_index = ifp->if_index;
612                rtm->rtm_flags |= rt->rt_flags;
613                rtm->rtm_errno = error;
614                rtm->rtm_addrs = info.rti_addrs;
615            }
616            route_proto.sp_protocol = sa ? sa->sa_family : 0;
617            raw_input(m, &route_proto, &route_src, &route_dst);
618        }
619 }
```
—— *rtsock.c*

Figure 19.29   rt_newaddrmsg function: second half, create rt_msghdr message.

19.12 rt_msg1 Function

The functions described in the previous three sections each called rt_msg1 to build the appropriate routing message. In Figure 19.25 we showed the mbuf chain that was built by rt_msg1 from the rt_msghdr and rt_addrinfo structures in Figure 19.22. Figure 19.30 shows the function.

Get mbuf and determine fixed size of message

399–422   An mbuf with a packet header is obtained and the length of the fixed-size message is stored in len. Two of the message types in Figure 18.9 use an ifa_msghdr structure, one uses an if_msghdr structure, and the remaining nine use an rt_msghdr structure.

Verify structure fits in mbuf

423–424   The size of the fixed-length structure must fit entirely within the data portion of the packet header mbuf, because the mbuf pointer is cast to a structure pointer using mtod and the structure is then referenced through the pointer. The largest of the three structures is if_msghdr, which at 84 bytes is less than MHLEN (100).

Initialize mbuf packet header and zero structure

425–428   The two fields in the packet header are initialized and the structure in the mbuf is set to 0.

——————————————————————————————————————— rtsock.c

```
399 static struct mbuf *
400 rt_msg1(type, rtinfo)
401 int     type;
402 struct rt_addrinfo *rtinfo;
403 {
404     struct rt_msghdr *rtm;
405     struct mbuf *m;
406     int     i;
407     struct sockaddr *sa;
408     int     len, dlen;

409     m = m_gethdr(M_DONTWAIT, MT_DATA);
410     if (m == 0)
411         return (m);
412     switch (type) {

413     case RTM_DELADDR:
414     case RTM_NEWADDR:
415         len = sizeof(struct ifa_msghdr);
416         break;

417     case RTM_IFINFO:
418         len = sizeof(struct if_msghdr);
419         break;

420     default:
421         len = sizeof(struct rt_msghdr);
422     }
423     if (len > MHLEN)
424         panic("rt_msg1");
425     m->m_pkthdr.len = m->m_len = len;
426     m->m_pkthdr.rcvif = 0;
427     rtm = mtod(m, struct rt_msghdr *);
428     bzero((caddr_t) rtm, len);

429     for (i = 0; i < RTAX_MAX; i++) {
430         if ((sa = rtinfo->rti_info[i]) == NULL)
431             continue;
432         rtinfo->rti_addrs |= (1 << i);
433         dlen = ROUNDUP(sa->sa_len);
434         m_copyback(m, len, dlen, (caddr_t) sa);
435         len += dlen;
436     }
437     if (m->m_pkthdr.len != len) {
438         m_freem(m);
439         return (NULL);
440     }
441     rtm->rtm_msglen = len;
442     rtm->rtm_version = RTM_VERSION;
443     rtm->rtm_type = type;
444     return (m);
445 }
```

——————————————————————————————————————— rtsock.c

Figure 19.30   rt_msg1 function: obtain and initialize mbuf.

Copy socket address structures into mbuf chain

429–436   The caller passes a pointer to an `rt_addrinfo` structure. The socket address structures corresponding to all the nonnull pointers in the `rti_info` are copied into the mbuf by `m_copyback`. The value 1 is left shifted by the RTAX_*xxx* index to generate the corresponding RTA_*xxx* bitmask (Figure 19.19), and each individual bitmask is logically ORed into the `rti_addrs` member, which the caller can store on return into the corresponding member of the message structure. The ROUNDUP macro rounds the size of each socket address structure up to the next multiple of 4 bytes.

437–440   If, when the loop terminates, the length in the mbuf packet header does not equal `len`, the function `m_copyback` wasn't able to obtain a required mbuf.

Store length, version, and type

441–445   The length, version, and message type are stored in the first three members of the message structure. Again, all three *xxx*_msghdr structures start with the same three members, so this code works with all three structures even though the pointer `rtm` is a pointer to an `rt_msghdr` structure.

19.13 `rt_msg2` Function

`rt_msg1` constructs a routing message in an mbuf chain, and the three functions that called it then called `raw_input` to append the mbuf chain to one or more socket's receive buffer. `rt_msg2` is different—it builds a routing message in a memory buffer, not an mbuf chain, and has as an argument a pointer to a `walkarg` structure that is used when `rt_msg2` is called by the two functions that handle the `sysctl` system call for the routing domain. `rt_msg2` is called in two different scenarios:

1. from `route_output` to process the RTM_GET command, and

2. from `sysctl_dumpentry` and `sysctl_iflist` to process a `sysctl` system call.

Before looking at `rt_msg2`, Figure 19.31 shows the `walkarg` structure that is used in scenario 2. We go through all these members as we encounter them.

—— *rtsock.c*
```
41 struct walkarg {
42      int      w_op;          /* NET_RT_xxx */
43      int      w_arg;         /* RTF_xxx for FLAGS, if_index for IFLIST */
44      int      w_given;       /* size of process' buffer */
45      int      w_needed;      /* #bytes actually needed (at end) */
46      int      w_tmemsize;    /* size of buffer pointed to by w_tmem */
47      caddr_t  w_where;       /* ptr to process' buffer (maybe null) */
48      caddr_t  w_tmem;        /* ptr to our malloc'ed buffer */
49 };
```
—— *rtsock.c*

Figure 19.31   `walkarg` structure: used with the `sysctl` system call in the routing domain.

Figure 19.32 shows the first half of the `rt_msg2` function. This portion is similar to the first half of `rt_msg1`.

rtsock.c

```
446 static int
447 rt_msg2(type, rtinfo, cp, w)
448 int      type;
449 struct rt_addrinfo *rtinfo;
450 caddr_t cp;
451 struct walkarg *w;
452 {
453     int      i;
454     int      len, dlen, second_time = 0;
455     caddr_t cp0;

456     rtinfo->rti_addrs = 0;
457 again:
458     switch (type) {

459     case RTM_DELADDR:
460     case RTM_NEWADDR:
461         len = sizeof(struct ifa_msghdr);
462         break;

463     case RTM_IFINFO:
464         len = sizeof(struct if_msghdr);
465         break;

466     default:
467         len = sizeof(struct rt_msghdr);
468     }
469     if (cp0 = cp)
470         cp += len;
471     for (i = 0; i < RTAX_MAX; i++) {
472         struct sockaddr *sa;

473         if ((sa = rtinfo->rti_info[i]) == 0)
474             continue;
475         rtinfo->rti_addrs |= (1 << i);
476         dlen = ROUNDUP(sa->sa_len);
477         if (cp) {
478             bcopy((caddr_t) sa, cp, (unsigned) dlen);
479             cp += dlen;
480         }
481         len += dlen;
482     }
```

rtsock.c

Figure 19.32 rt_msg2 function: copy socket address structures.

446–455   Since this function stores the resulting message in a memory buffer, the caller specifies the start of that buffer in the cp argument. It is the caller's responsibility to ensure that the buffer is large enough for the message that is generated. To help the caller determine this size, if the cp argument is null, rt_msg2 doesn't store anything but processes the input and returns the total number of bytes required to hold the result. We'll see that route_output uses this feature and calls this function twice: first to determine the size and then to store the result, after allocating a buffer of the correct size. When rt_msg2 is called by route_output, the final argument is null. This final argument is nonnull when called as part of the sysctl system call processing.

Determine size of structure

458–470   The size of the fixed-length message structure is set based on the message type. If the `cp` pointer is nonnull, it is incremented by this size.

Copy socket address structures

471–482   The `for` loop goes through the `rti_info` array, and for each element that is a non-null pointer it sets the appropriate bit in the `rti_addrs` bitmask, copies the socket address structure (if `cp` is nonnull), and updates the length.

Figure 19.33 shows the second half of `rt_msg2`, most of which handles the optional `walkarg` structure.

```
                                                                    ─ rtsock.c
483        if (cp == 0 && w != NULL && !second_time) {
484            struct walkarg *rw = w;

485            rw->w_needed += len;
486            if (rw->w_needed <= 0 && rw->w_where) {
487                if (rw->w_tmemsize < len) {
488                    if (rw->w_tmem)
489                        free(rw->w_tmem, M_RTABLE);
490                    if (rw->w_tmem = (caddr_t)
491                        malloc(len, M_RTABLE, M_NOWAIT))
492                        rw->w_tmemsize = len;
493                }
494                if (rw->w_tmem) {
495                    cp = rw->w_tmem;
496                    second_time = 1;
497                    goto again;
498                } else
499                    rw->w_where = 0;
500            }
501        }
502        if (cp) {
503            struct rt_msghdr *rtm = (struct rt_msghdr *) cp0;

504            rtm->rtm_version = RTM_VERSION;
505            rtm->rtm_type = type;
506            rtm->rtm_msglen = len;
507        }
508        return (len);
509 }
                                                                    ─ rtsock.c
```

Figure 19.33   `rt_msg2` function: handle optional `walkarg` argument.

483–484   This `if` statement is true only when a pointer to a `walkarg` structure was passed and this is the first loop through the function. The variable `second_time` was initialized to 0 but can be set to 1 within this `if` statement, and a jump made back to the label `again` in Figure 19.32. The test for `cp` being a null pointer is superfluous since whenever the `w` pointer is nonnull, the `cp` pointer is null, and vice versa.

Check if data to be stored

485–486   `w_needed` is incremented by the size of the message. This variable is initialized to 0 minus the size of the user's buffer to the `sysctl` function. For example, if the buffer

size is 500 bytes, w_needed is initialized to –500. As long as it remains negative, there
is room in the buffer. w_where is a pointer to the buffer in the calling process. It is null
if the process doesn't want the result—the process just wants sysctl to return the size
of the result, so the process can allocate a buffer and call sysctl again. rt_msg2
doesn't copy the data back to the process—that is up to the caller—but if the w_where
pointer is null, there's no need for rt_msg2 to malloc a buffer to hold the result and
loop back through the function again, storing the result in this buffer. There are really
five different scenarios that this function handles, summarized in Figure 19.34.

called from	cp	w	w.w_where	second_time	Description
route_output	null	null			wants return length
	nonnull	null			wants result
sysctl_rtable	null	nonnull	null	0	process wants return length
	null	nonnull	nonnull	0	first time around to calculate length
	nonnull	nonnull	nonnull	1	second time around to store result

Figure 19.34 Summary of different scenarios for rt_msg2.

Allocate buffer first time or if message length increases

487–493   w_tmemsize is the size of the buffer pointed to by w_tmem. It is initialized to 0 by
sysctl_rtable, so the first time rt_msg2 is called for a given sysctl request, the
buffer must be allocated. Also, if the size of the result increases, the existing buffer must
be released and a new (larger) buffer allocated.

Go around again and store result

494–499   If w_tmem is nonnull, a buffer already exists or one was just allocated. cp is set to
point to this buffer, second_time is set to 1, and a jump is made to again. The if
statement at the beginning of this figure won't be true during this second pass, since
second_time is now 1. If w_tmem is null, the call to malloc failed, so the pointer to
the buffer in the process is set to null, preventing anything from being returned.

Store length, version, and type

502–509   If cp is nonnull, the first three elements of the message header are stored. The func-
tion returns the length of the message.

19.14 sysctl_rtable Function

This function handles the sysctl system call on a routing socket. It is called by
net_sysctl as shown in Figure 18.11.
   Before going through the source code, Figure 19.35 shows the typical use of this sys-
tem call with respect to the routing table. This example is from the arp program.
   The first three elements in the mib array cause the kernel to call sysctl_rtable
to process the remaining elements.

```
int     mib[6];
size_t  needed;
char    *buf, *lim, *next;
struct rt_msghdr  *rtm;

mib[0] = CTL_NET;
mib[1] = PF_ROUTE;
mib[2] = 0;
mib[3] = AF_INET;         /* address family; can be 0 */
mib[4] = NET_RT_FLAGS;    /* operation */
mib[5] = RTF_LLINFO;      /* flags; can be 0 */

if (sysctl(mib, 6, NULL, &needed, NULL, 0) < 0)
    quit("sysctl error, estimate");

if ( (buf = malloc(needed)) == NULL)
    quit("malloc");

if (sysctl(mib, 6, buf, &needed, NULL, 0) < 0)
    quit("sysctl error, retrieval");

lim = buf + needed;
for (next = buf; next < lim; next += rtm->rtm_msglen) {
    rtm = (struct rt_msghdr *)next;
    ...   /* do whatever */
}
```

Figure 19.35   Example of sysctl with routing table.

mib[4] specifies the operation. Three operations are supported.

1. NET_RT_DUMP: return the routing table corresponding to the address family specified by mib[3]. If the address family is 0, all routing tables are returned.

   An RTM_GET routing message is returned for each routing table entry containing two, three, or four socket address structures per message: those addresses pointed to by rt_key, rt_gateway, rt_netmask, and rt_genmask. The final two pointers might be null.

2. NET_RT_FLAGS: the same as the previous command except mib[5] specifies an RTF_xxx flag (Figure 18.25), and only entries with this flag set are returned.

3. NET_RT_IFLIST: return information on all the configured interfaces. If the mib[5] value is nonzero it specifies an interface index and only the interface with the corresponding if_index is returned. Otherwise all interfaces on the ifnet linked list are returned.

   For each interface one RTM_IFINFO message is returned, with information about the interface itself, followed by one RTM_NEWADDR message for each ifaddr structure on the interface's if_addrlist linked list. If the mib[3] value is nonzero, RTM_NEWADDR messages are returned for only the addresses

with an address family that matches the mib[3] value. Otherwise mib[3] is 0 and information on all addresses is returned.

This operation is intended to replace the SIOCGIFCONF ioctl (Figure 4.26).

One problem with this system call is that the amount of information returned can vary, depending on the number of routing table entries or the number of interfaces. Therefore the first call to sysctl typically specifies a null pointer as the third argument, which means: don't return any data, just return the number of bytes of return information. As we see in Figure 19.35, the process then calls malloc, followed by sysctl to fetch the information. This second call to sysctl again returns the number of bytes through the fourth argument (which might have changed since the previous call), and this value provides the pointer lim that points just beyond the final byte of data that was returned. The process then steps through the routing messages in the buffer, using the rtm_msglen member to step to the next message.

Figure 19.36 shows the values for these six mib variables that various Net/3 programs specify to access the routing table and interface list.

mib[]	arp	route	netstat	routed	gated	rwhod
0	CTL_NET	CTL_NET	CTL_NET	CTL_NET	CTL_NET	CTL_NET
1	PF_ROUTE	PF_ROUTE	PF_ROUTE	PF_ROUTE	PF_ROUTE	PF_ROUTE
2	0	0	0	0	0	0
3	AF_INET	0	0	AF_INET	0	AF_INET
4	NET_RT_FLAGS	NET_RT_DUMP	NET_RT_DUMP	NET_RT_IFLIST	NET_RT_IFLIST	NET_RT_IFLIST
5	RTF_LLINFO	0	0	0	0	0

Figure 19.36 Examples of programs that call sysctl to obtain routing table and interface list.

The first three programs fetch entries from the routing table and the last three fetch the interface list. The routed program supports only the Internet routing protocols, so it specifies a mib[3] value of AF_INET, while gated supports other protocols, so its value for mib[3] is 0.

Figure 19.37 shows the organization of the three sysctl_xxx functions that we cover in the following sections.

Figure 19.38 shows the sysctl_rtable function.

Validate arguments

705–719   The new argument is used when the process is calling sysctl to set the value of a variable, which isn't supported with the routing tables. Therefore this argument must be a null pointer.

720–721   namelen must be 3 because at this point in the processing of the system call, three elements in the name array remain: name[0], the address family (what the process specifies as mib[3]); name[1], the operation (mib[4]); and name[2], the flags (mib[5]).

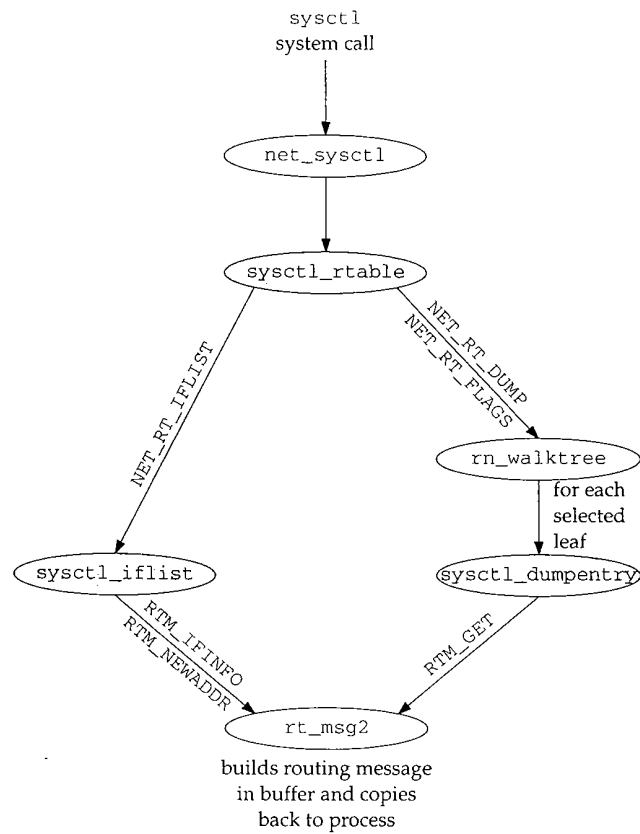**Figure 19.37**   Functions that support the sysctl system call for routing sockets.

```
                                                                                ─── rtsock.c
705 int
706 sysctl_rtable(name, namelen, where, given, new, newlen)
707 int     *name;
708 int      namelen;
709 caddr_t where;
710 size_t *given;
711 caddr_t *new;
712 size_t  newlen;
713 {
714     struct radix_node_head *rnh;
715     int     i, s, error = EINVAL;
716     u_char  af;
717     struct walkarg w;

718     if (new)
719         return (EPERM);
```

```
720    if (namelen != 3)
721        return (EINVAL);
722    af = name[0];
723    Bzero(&w, sizeof(w));
724    w.w_where = where;
725    w.w_given = *given;
726    w.w_needed = 0 - w.w_given;
727    w.w_op = name[1];
728    w.w_arg = name[2];

729    s = splnet();
730    switch (w.w_op) {

731    case NET_RT_DUMP:
732    case NET_RT_FLAGS:
733        for (i = 1; i <= AF_MAX; i++)
734            if ((rnh = rt_tables[i]) && (af == 0 || af == i) &&
735                (error = rnh->rnh_walktree(rnh,
736                                           sysctl_dumpentry, &w)))
737                break;
738        break;

739    case NET_RT_IFLIST:
740        error = sysctl_iflist(af, &w);
741    }
742    splx(s);
743    if (w.w_tmem)
744        free(w.w_tmem, M_RTABLE);
745    w.w_needed += w.w_given;
746    if (where) {
747        *given = w.w_where - where;
748        if (*given < w.w_needed)
749            return (ENOMEM);
750    } else {
751        *given = (11 * w.w_needed) / 10;
752    }
753    return (error);
754 }
```
—— *rtsock.c*

— *rtsock.c*

Figure 19.38 syscl_rtable function: process syscl system call requests.

Initialize walkarg structure

723–728   A walkarg structure (Figure 19.31) is set to 0 and the following members are initialized: w_where is the address in the calling process of the buffer for the results (this can be a null pointer, as we mentioned); w_given is the size of the buffer in bytes (this is meaningless on input if w_where is a null pointer, but it must be set on return to the amount of data that would have been returned); w_needed is set to the negative of the buffer size; w_op is the operation (the NET_RT_*xxx* value); and w_arg is the flags value.

Dump routing table

731–738   The NET_RT_DUMP and NET_RT_FLAGS operations are handled the same way: a loop is made through all the routing tables (the rt_tables array), and if the routing

table is in use and either the address family argument was 0 or the address family argument matches the family of this routing table, the rnh_walktree function is called to process the entire routing table. In Figure 18.17 we show that this function is normally rn_walktree. The second argument to this function is the address of another function that is called for each leaf of the routing tree (sysctl_dumpentry). The third argument is just a pointer to anything that rn_walktree passes to the sysctl_dumpentry function. This argument is a pointer to the walkarg structure that contains all the information about this sysctl call.

Return interface list

739–740   The NET_RT_IFLIST operation calls the function sysctl_iflist, which goes through all the ifnet structures.

Release buffer

743–744   If a buffer was allocated by rt_msg2 to contain a routing message, it is now released.

Update w_needed

745   The size of each message was added to w_needed by rt_msg2. Since this variable was initialized to the negative of w_given, its value can now be expressed as

$$w\_needed = 0 - w\_given + totalbytes$$

where totalbytes is the sum of all the message lengths added by rt_msg2. By adding the value of w_given back into w_needed, we get

$$\begin{aligned} w\_needed &= 0 - w\_given + totalbytes + w\_given \\ &= totalbytes \end{aligned}$$

the total number of bytes. Since the two values of w_given in this equation end up canceling each other, when the process specifies w_where as a null pointer it need not initialize the value of w_given. Indeed, we see in Figure 19.35 that the variable needed was not initialized.

Return actual size of message

746–749   If where is nonnull, the number of bytes stored in the buffer is returned through the given pointer. If this value is less than the size of the buffer specified by the process, an error is returned because the return information has been truncated.

Return estimated size of message

750–752   When the where pointer is null, the process just wants the total number of bytes returned. A 10% fudge factor is added to the size, in case the size of the desired tables increases between this call to sysctl and the next.

19.15 sysctl_dumpentry Function

In the previous section we described how this function is called by rn_walktree, which in turn is called by sysctl_rtable. Figure 19.39 shows the function.

```
                                                                    ──────── rtsock.c
623  int
624  sysctl_dumpentry(rn, w)
625  struct radix_node *rn;
626  struct walkarg *w;
627  {
628      struct rtentry *rt = (struct rtentry *) rn;
629      int      error = 0, size;
630      struct rt_addrinfo info;

631      if (w->w_op == NET_RT_FLAGS && !(rt->rt_flags & w->w_arg))
632          return 0;
633      bzero((caddr_t) & info, sizeof(info));
634      dst = rt_key(rt);
635      gate = rt->rt_gateway;
636      netmask = rt_mask(rt);
637      genmask = rt->rt_genmask;
638      size = rt_msg2(RTM_GET, &info, 0, w);
639      if (w->w_where && w->w_tmem) {
640          struct rt_msghdr *rtm = (struct rt_msghdr *) w->w_tmem;

641          rtm->rtm_flags = rt->rt_flags;
642          rtm->rtm_use = rt->rt_use;
643          rtm->rtm_rmx = rt->rt_rmx;
644          rtm->rtm_index = rt->rt_ifp->if_index;
645          rtm->rtm_errno = rtm->rtm_pid = rtm->rtm_seq = 0;
646          rtm->rtm_addrs = info.rti_addrs;
647          if (error = copyout((caddr_t) rtm, w->w_where, size))
648              w->w_where = NULL;
649          else
650              w->w_where += size;
651      }
652      return (error);
653  }
                                                                    ──────── rtsock.c
```

Figure 19.39 sysctl_dumpentry function: process one routing table entry.

623–630   Each time this function is called, its first argument points to a radix_node struc-
ture, which is also a pointer to a rtentry structure. The second argument points to the
walkarg structure that was initialized by sysctl_rtable.

Check flags of routing table entry

631–632   If the process specified a flag value (mib[5]), this entry is skipped if the rt_flags
member doesn't have the desired flag set. We see in Figure 19.36 that the arp program
uses this to select only those entries with the RTF_LLINFO flag set, since these are the
entries of interest to ARP.

Form routing message

633–638   The following four pointers in the rti_info array are copied from the routing
table entry: dst, gate, netmask, and genmask. The first two are always nonnull, but
the other two can be null. rt_msg2 forms an RTM_GET message.

Copy message back to process

639–651   If the process wants the message returned and a buffer was allocated by `rt_msg2`, the remainder of the routing message is formed in the buffer pointed to by `w_tmem` and `copyout` copies the message back to the process. If the copy was successful, `w_where` is incremented by the number of bytes copied.

19.16 `sysctl_iflist` Function

This function, shown in Figure 19.40, is called directly by `sysctl_rtable` to return the interface list to the process.

—————————————————————————————————— rtsock.c

```
654 int
655 sysctl_iflist(af, w)
656 int        af;
657 struct walkarg *w;
658 {
659     struct ifnet *ifp;
660     struct ifaddr *ifa;
661     struct rt_addrinfo info;
662     int    len, error = 0;

663     bzero((caddr_t) & info, sizeof(info));
664     for (ifp = ifnet; ifp; ifp = ifp->if_next) {
665         if (w->w_arg && w->w_arg != ifp->if_index)
666             continue;
667         ifa = ifp->if_addrlist;
668         ifpaddr = ifa->ifa_addr;
669         len = rt_msg2(RTM_IFINFO, &info, (caddr_t) 0, w);
670         ifpaddr = 0;
671         if (w->w_where && w->w_tmem) {
672             struct if_msghdr *ifm;

673             ifm = (struct if_msghdr *) w->w_tmem;
674             ifm->ifm_index = ifp->if_index;
675             ifm->ifm_flags = ifp->if_flags;
676             ifm->ifm_data = ifp->if_data;
677             ifm->ifm_addrs = info.rti_addrs;
678             if (error = copyout((caddr_t) ifm, w->w_where, len))
679                 return (error);
680             w->w_where += len;
681         }
682         while (ifa = ifa->ifa_next) {
683             if (af && af != ifa->ifa_addr->sa_family)
684                 continue;
685             ifaaddr = ifa->ifa_addr;
686             netmask = ifa->ifa_netmask;
687             brdaddr = ifa->ifa_dstaddr;
688             len = rt_msg2(RTM_NEWADDR, &info, 0, w);
689             if (w->w_where && w->w_tmem) {
690                 struct ifa_msghdr *ifam;
```

```
691                        ifam = (struct ifa_msghdr *) w->w_tmem;
692                        ifam->ifam_index = ifa->ifa_ifp->if_index;
693                        ifam->ifam_flags = ifa->ifa_flags;
694                        ifam->ifam_metric = ifa->ifa_metric;
695                        ifam->ifam_addrs = info.rti_addrs;
696                        if (error = copyout(w->w_tmem, w->w_where, len))
697                            return (error);
698                        w->w_where += len;
699                    }
700                }
701            ifaaddr = netmask = brdaddr = 0;
702        }
703    return (0);
704 }
```
                                                                                              —— rtsock.c

Figure 19.40 sysctl_iflist function: return list of interfaces and their addresses.

This function is a for loop that iterates through each interface starting with the one pointed to by ifnet. Then a while loop proceeds through the linked list of ifaddr structures for each interface. An RTM_IFINFO routing message is generated for each interface and an RTM_NEWADDR message for each address.

Check interface index

654–666   The process can specify a nonzero flags argument (mib[5] in Figure 19.36) to select only the interface with a matching if_index value.

Build routing message

667–670   The only socket address structure returned with the RTM_IFINFO message is ifpaddr. The message is built by rt_msg2. The pointer ifpaddr in the info structure is then set to 0, since the same info structure is used for generating the subsequent RTM_NEWADDR messages.

Copy message back to process

671–681   If the process wants the message returned, the remainder of the if_msghdr structure is filled in, copyout copies the buffer to the process, and w_where is incremented.

Iterate through address structures, check address family

682–684   Each ifaddr structure for the interface is processed and the process can specify a nonzero address family (mib[3] in Figure 19.36) to select only the interface addresses of the given family.

Build routing message

685–688   Up to three socket address structures are returned in each RTM_NEWADDR message: ifaaddr, netmask, and brdaddr. The message is built by rt_msg2.

Copy message back to process

689–699   If the process wants the message returned, the remainder of the ifa_msghdr structure is filled in, copyout copies the buffer to the process, and w_where is incremented.

701   These three pointers in the info array are set to 0, since the same array is used for the next interface message.

19.17 Summary

Routing messages all have the same format—a fixed-length structure followed by a variable number of socket address structures. There are three different types of messages, each corresponding to a different fixed-length structure, and the first three elements of each structure identify the length, version, and type of message. A bitmask in each structure identifies which socket address structures follow the fixed-length structure.

These messages are passed between a process and the kernel in two different ways. Messages can be passed in either direction, one message per read or write, across a routing socket. This allows a superuser process complete read and write access to the kernel's routing tables. This is how routing daemons such as `routed` and `gated` implement their desired routing policy.

Alternatively any process can read the contents of the kernel's routing tables using the `sysctl` system call. This does not involve a routing socket and does not require special privileges. The entire result, normally consisting of many routing messages, is returned as part of the system call. Since the process does not know the size of the result, a method is provided for the system call to return this size without returning the actual result.

Exercises

19.1 What is the difference in the `RTF_DYNAMIC` and `RTF_MODIFIED` flags? Can both be set for a given routing table entry?

19.2 What happens when the default route is entered with a command of the form

```
bsdi $ route add default -cloning -genmask 255.255.255.255 sun
```

19.3 Estimate the space required by `sysctl` to dump a routing table that contains 15 ARP entries and 20 routes.

19
—

a
:s-
ie-
in
ic-

ys.
it-
er-
ed

ng
ire
, is
:he
:he

for

\RP

20

Routing Sockets

20.1 Introduction

A process sends and receives the routing messages described in the previous chapter by using a socket in the *routing domain*. The socket system call is issued specifying a family of PF_ROUTE and a socket type of SOCK_RAW.

The process can then send five routing messages to the kernel:

1. RTM_ADD: add a new route.
2. RTM_DELETE: delete an existing route.
3. RTM_GET: fetch all the information about a route.
4. RTM_CHANGE: change the gateway, interface, or metrics of an existing route.
5. RTM_LOCK: specify which metrics the kernel should not modify.

Additionally, the process can receive any of the other seven types of routing messages that are generated by the kernel when some event, such as interface down, redirect received, etc., occurs.

This chapter looks at the routing domain, the routing control blocks that are created for each routing socket, the function that handles messages from a process (route_output), the function that sends routing messages to one or more processes (raw_input), and the various functions that support all the socket operations on a routing socket.

645

20.2 `routedomain` and `protosw` Structures

Before describing the routing socket functions, we need to discuss additional details about the routing domain; the `SOCK_RAW` protocol supported in the routing domain; and routing control blocks, one of which is associated with each routing socket.

Figure 20.1 lists the `domain` structure for the `PF_ROUTE` domain, named `routedomain`.

Member	Value	Description
dom_family	PF_ROUTE	protocol family for domain
dom_name	route	name
dom_init	route_init	domain initialization, Figure 18.30
dom_externalize	0	not used in routing domain
dom_dispose	0	not used in routing domain
dom_protosw	routesw	protocol switch structure, Figure 20.2
dom_protoswNPROTOSW		pointer past end of protocol switch structure
dom_next		filled in by domaininit, Figure 7.15
dom_rtattach	0	not used in routing domain
dom_rtoffset	0	not used in routing domain
dom_maxrtkey	0	not used in routing domain

Figure 20.1   `routedomain` structure.

Unlike the Internet domain, which supports multiple protocols (TCP, UDP, ICMP, etc.), only one protocol (of type `SOCK_RAW`) is supported in the routing domain. Figure 20.2 lists the protocol switch entry for the `PF_ROUTE` domain.

Member	routesw[0]	Description
pr_type	SOCK_RAW	raw socket
pr_domain	&routedomain	part of the routing domain
pr_protocol	0	
pr_flags	PR_ATOMIC | PR_ADDR	socket layer flags, not used by protocol processing
pr_input	raw_input	this entry not used; raw_input called directly
pr_output	route_output	called for PRU_SEND requests
pr_ctlinput	raw_ctlinput	control input function
pr_ctloutput	0	not used
pr_usrreq	route_usrreq	respond to communication requests from a process
pr_init	raw_init	initialization
pr_fasttimo	0	not used
pr_slowtimo	0	not used
pr_drain	0	not used
pr_sysctl	sysctl_rtable	for sysctl(8) system call

Figure 20.2   The routing protocol `protosw` structure.

20.3 Routing Control Blocks

Each time a routing socket is created with a call of the form

    socket(PF_ROUTE, SOCK_RAW, protocol);

the corresponding PRU_ATTACH request to the protocol's user-request function (route_usrreq) allocates a routing control block and links it to the socket structure. The *protocol* can restrict the messages sent to the process on this socket to one particular family. If a *protocol* of AF_INET is specified, for example, only routing messages containing Internet addresses will be sent to the process. A *protocol* of 0 causes all routing messages from the kernel to be sent on the socket.

> Recall that we call these structures *routing control blocks*, not *raw control blocks*, to avoid confusion with the raw IP control blocks in Chapter 32.

Figure 20.3 shows the definition of the rawcb structure.

raw_cb.h
```
39 struct rawcb {
40     struct rawcb *rcb_next;      /* doubly linked list */
41     struct rawcb *rcb_prev;
42     struct socket *rcb_socket;   /* back pointer to socket */
43     struct sockaddr *rcb_faddr;  /* destination address */
44     struct sockaddr *rcb_laddr;  /* socket's address */
45     struct sockproto rcb_proto;  /* protocol family, protocol */
46 };

47 #define sotorawcb(so)        ((struct rawcb *)(so)->so_pcb)
```
raw_cb.h

Figure 20.3 rawcb structure.

Additionally, a global of the same name, rawcb, is allocated as the head of the doubly linked list. Figure 20.4 shows the arrangement.

39–47 We showed the sockproto structure in Figure 19.26. Its sp_family member is set to PF_ROUTE and its sp_protocol member is set to the third argument to the socket system call. The rcb_faddr member is permanently set to point to route_src, which we described with Figure 19.26. rcb_laddr is always a null pointer.

20.4 raw_init Function

The raw_init function, shown in Figure 20.5, is the protocol initialization function in the protosw structure in Figure 20.2. We described the entire initialization of the routing domain with Figure 18.29.

38–42 The function initializes the doubly linked list of routing control blocks by setting the next and previous pointers of the head structure to point to itself.
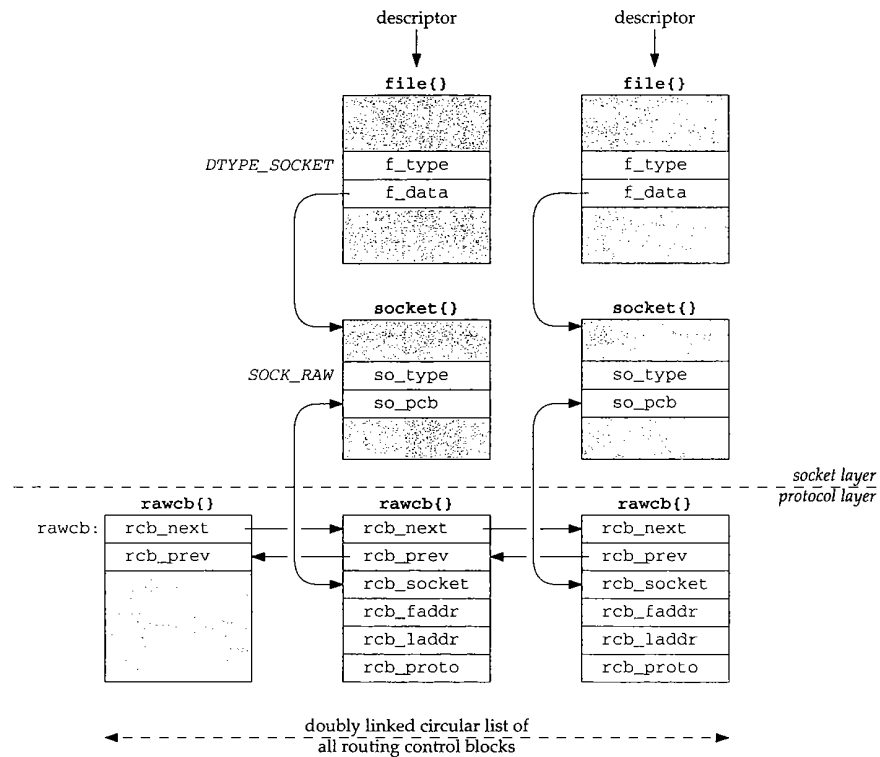
Figure 20.4   Relationship of raw protocol control blocks to other data structures.

```
                                                                     raw_usrreq.c
38 void
39 raw_init()
40 {

41      rawcb.rcb_next = rawcb.rcb_prev = &rawcb;
42 }
                                                                     raw_usrreq.c
```

Figure 20.5   `raw_init` function: initialize doubly linked list of routing control blocks.

20.5   `route_output` Function

As we showed in Figure 18.11, `route_output` is called when the PRU_SEND request is issued to the protocol's user-request function, which is the result of a write operation by a process to a routing socket. In Figure 18.9 we indicated that five different types of routing messages are accepted by the kernel from a process.

Since this function is invoked as a result of a write by a process, the data from the process (the routing message to process) is in an mbuf chain from `sosend`. Figure 20.6

shows an overview of the processing steps, assuming the process sends an RTM_ADD command, specifying three addresses: the destination, its gateway, and a network mask (hence this is a network route, not a host route).
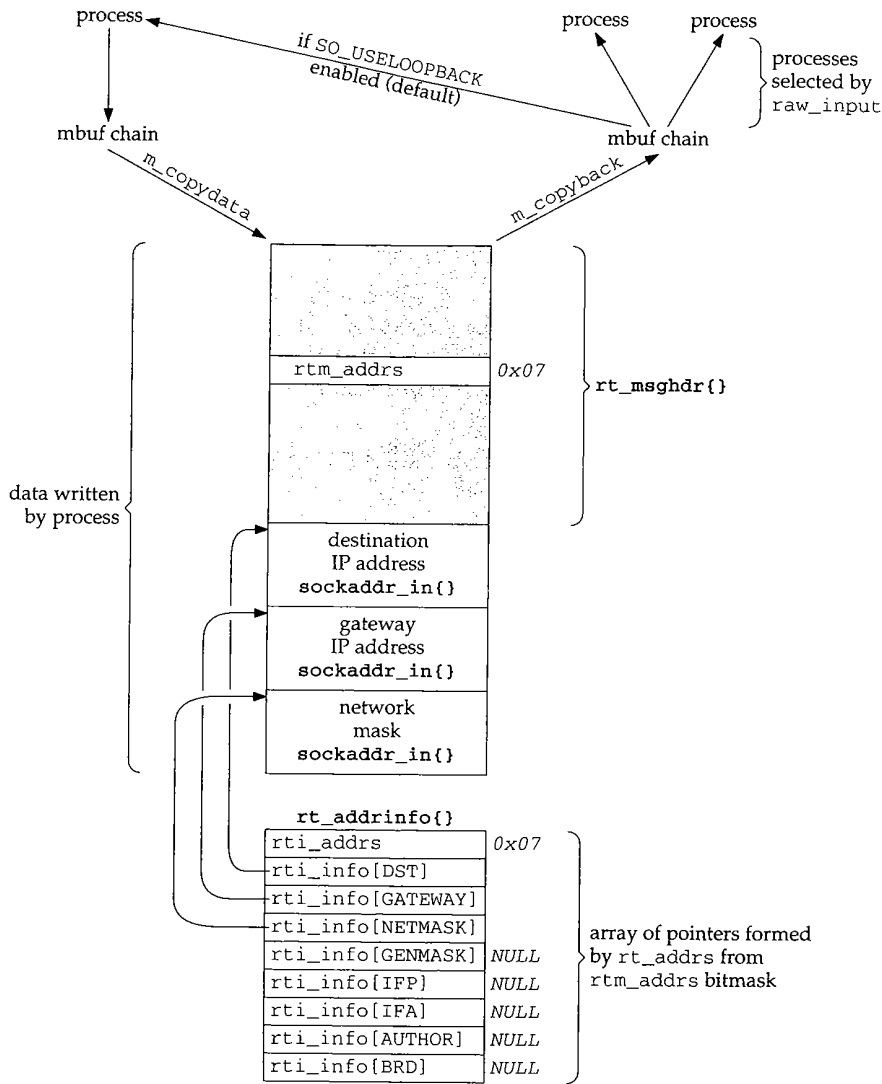

Figure 20.6   Example processing of an RTM_ADD command from a process.

There are numerous points to note in this figure, most of which we'll cover as we proceed through the source code for route_output. Also note that, to save space, we omit the RTAX_ prefix for each array index in the rt_addrinfo structure.

- The process specifies which socket address structures follow the fixed-length rt_msghdr structure by setting the bitmask rtm_addrs. We show a bitmask of 0x07, which corresponds to a destination address, a gateway address, and a network mask (Figure 19.19). The RTM_ADD command requires the first two; the third is optional. Another optional address, the genmask specifies the mask to be used for generating cloned routes.

- The write system call (the sosend function) copies the buffer from the process into an mbuf chain in the kernel.

- m_copydata copies the mbuf chain into a buffer that route_output obtains using malloc. It is easier to access all the information in the structure and the socket address structures that follow when stored in a single contiguous buffer than it is when stored in an mbuf chain.

- The function rt_xaddrs is called by route_output to take the bitmask and build the rt_addrinfo structure that points into the buffer. The code in route_output references these structures using the names shown in the fifth column in Figure 19.19. The bitmask is also copied into the rti_addrs member.

- route_output normally modifies the rt_msghdr structure. If an error occurs, the corresponding errno value is returned in rtm_errno (for example, EEXIST if the route already exists); otherwise the flag RTF_DONE is logically ORed into the rtm_flags supplied by the process.

- The rt_msghdr structure and the addresses that follow become input to 0 or more processes that are reading from a routing socket. The buffer is first converted back into an mbuf chain by m_copyback. raw_input goes through all the routing PCBs and passes a copy to the appropriate processes. We also show that a process with a routing socket receives a copy of each message it writes to that socket unless it disables the SO_USELOOPBACK socket option.

> To avoid receiving a copy of their own routing messages, some programs, such as route, call shutdown with a second argument of 0 to prevent any data from being received on the routing socket.

We examine the source code for route_output in seven parts. Figure 20.7 shows an overview of the function.

```
int
route_output()
{
    R_Malloc() to allocate buffer;
    m_copydata() to copy from mbuf chain into buffer;
    rt_xaddrs() to build rt_addrinfo();

    switch (message type) {
    case RTM_ADD:
        rtrequest(RTM_ADD);
        rt_setmetrics();
        break;
```

Left margin fragments:
:d-length
bitmask
ss, and a
: two; the
· mask to

e process

: obtains
: and the
us buffer

nask and
code in
. the fifth
rs mem-

or occurs,
example,
logically

it to 0 or
first con-
rough all
also show
writes to

n as route,
received on

shows an

```
        case RTM_DELETE:
            rtrequest(RTM_DELETE);
            break;

    case RTM_GET:
    case RTM_CHANGE:
    case RTM_LOCK:
        rtalloc1();

        switch (message type) {
        case RTM_GET:
            rt_msg2(RTM_GET);
            break;

        case RTM_CHANGE:
            change appropriate fields;
            /* fall through */

        case RTM_LOCK:
            set rmx_locks;
            break;
        }
        break;
    }

    set rtm_error if error, else set RTF_DONE flag;

    m_copyback() to copy from buffer into mbuf chain;

    raw_input();    /* mbuf chain to appropriate processes */
}
```

Figure 20.7   Summary of route_output processing steps.

The first part of route_output is shown in Figure 20.8.

Check mbuf for validity

113–136   The mbuf chain is checked for validity: its length must be at least the size of an rt_msghdr structure. The first longword is fetched from the data portion of the mbuf, which contains the rtm_msglen value.

Allocate buffer

137–142   A buffer is allocated to hold the entire message and m_copydata copies the message from the mbuf chain into the buffer.

Check version number

143–146   The version of the message is checked. In the future, should a new version of the routing messages be introduced, this member could be used to provide support for older versions.

147–149   The process ID is copied into rtm_pid and the bitmask supplied by the process is copied into info.rti_addrs, a structure local to this function. The function rt_xaddrs (shown in the next section) fills in the eight socket address pointers in the info structure to point into the buffer now containing the message.

rtsock.c

```
113 int
114 route_output(m, so)
115 struct mbuf *m;
116 struct socket *so;
117 {
118     struct rt_msghdr *rtm = 0;
119     struct rtentry *rt = 0;
120     struct rtentry *saved_nrt = 0;
121     struct rt_addrinfo info;
122     int     len, error = 0;
123     struct ifnet *ifp = 0;
124     struct ifaddr *ifa = 0;

125 #define senderr(e) { error = e; goto flush;}
126     if (m == 0 || ((m->m_len < sizeof(long)) &&
127                          (m = m_pullup(m, sizeof(long))) == 0))
128             return (ENOBUFS);
129     if ((m->m_flags & M_PKTHDR) == 0)
130         panic("route_output");
131     len = m->m_pkthdr.len;
132     if (len < sizeof(*rtm) ||
133         len != mtod(m, struct rt_msghdr *)->rtm_msglen) {
134         dst = 0;
135         senderr(EINVAL);
136     }
137     R_Malloc(rtm, struct rt_msghdr *, len);
138     if (rtm == 0) {
139         dst = 0;
140         senderr(ENOBUFS);
141     }
142     m_copydata(m, 0, len, (caddr_t) rtm);
143     if (rtm->rtm_version != RTM_VERSION) {
144         dst = 0;
145         senderr(EPROTONOSUPPORT);
146     }
147     rtm->rtm_pid = curproc->p_pid;

148     info.rti_addrs = rtm->rtm_addrs;
149     rt_xaddrs((caddr_t) (rtm + 1), len + (caddr_t) rtm, &info);

150     if (dst == 0)
151         senderr(EINVAL);

152     if (genmask) {
153         struct radix_node *t;
154         t = rn_addmask((caddr_t) genmask, 1, 2);
155         if (t && Bcmp(genmask, t->rn_key, *(u_char *) genmask) == 0)
156             genmask = (struct sockaddr *) (t->rn_key);
157         else
158             senderr(ENOBUFS);
159     }
```

rtsock.c

Figure 20.8 route_output function: initial processing, copy message from mbuf chain.

- rtsock.c

Destination address required

150–151   A destination address is a required address for all commands. If the info.rti_info[RTAX_DST] element is a null pointer, EINVAL is returned. Remember that dst refers to this array element (Figure 19.19).

Handle optional genmask

152–159   A genmask is optional and is used as the network mask for routes created when the RTF_CLONING flag is set (Figure 19.8). rn_addmask adds the mask to the tree of masks, first searching for an existing entry for the mask and then referencing that entry if found. If the mask is found or added to the mask tree, an additional check is made that the entry in the mask tree really equals the genmask value, and, if so, the genmask pointer is replaced with a pointer to the mask in the mask tree.

Figure 20.9 shows the next part of route_output, which handles the RTM_ADD and RTM_DELETE commands.

```
                                                                  ──── rtsock.c
160     switch (rtm->rtm_type) {

161     case RTM_ADD:
162         if (gate == 0)
163             senderr(EINVAL);
164         error = rtrequest(RTM_ADD, dst, gate, netmask,
165                           rtm->rtm_flags, &saved_nrt);
166         if (error == 0 && saved_nrt) {
167             rt_setmetrics(rtm->rtm_inits,
168                           &rtm->rtm_rmx, &saved_nrt->rt_rmx);
169             saved_nrt->rt_refcnt--;
170             saved_nrt->rt_genmask = genmask;
171         }
172         break;

173     case RTM_DELETE:
174         error = rtrequest(RTM_DELETE, dst, gate, netmask,
175                           rtm->rtm_flags, (struct rtentry **) 0);
176         break;
                                                                  ──── rtsock.c
```

Figure 20.9   route_output function: process RTM_ADD and RTM_DELETE commands.

162–163   An RTM_ADD command requires the process to specify a gateway.
164–165   rtrequest processes the request. The netmask pointer can be null if the route being entered is a host route. If all is OK, the pointer to the new routing table entry is returned through saved_nrt.
166–172   The rt_metrics structure is copied from the caller's buffer into the routing table entry. The reference count is decremented and the genmask pointer is stored (possibly a null pointer).
173–176   Processing the RTM_DELETE command is simple because all the work is done by rtrequest. Since the final argument is a null pointer, rtrequest calls rtfree if the reference count is 0, deleting the entry from the routing table (Figure 19.7).

— rtsock.c

The next part of the processing is shown in Figure 20.10, which handles the common code for the RTM_GET, RTM_CHANGE, and RTM_LOCK commands.

```
                                                                    rtsock.c
177        case RTM_GET:
178        case RTM_CHANGE:
179        case RTM_LOCK:
180             rt = rtalloc1(dst, 0);
181             if (rt == 0)
182                 senderr(ESRCH);
183             if (rtm->rtm_type != RTM_GET) {        /* XXX: too grotty */
184                 struct radix_node *rn;
185                 extern struct radix_node_head *mask_rnhead;

186                 if (Bcmp(dst, rt_key(rt), dst->sa_len) != 0)
187                     senderr(ESRCH);
188                 if (netmask && (rn = rn_search(netmask,
189                                                 mask_rnhead->rnh_treetop)))
190                     netmask = (struct sockaddr *) rn->rn_key;
191                 for (rn = rt->rt_nodes; rn; rn = rn->rn_dupedkey)
192                     if (netmask == (struct sockaddr *) rn->rn_mask)
193                         break;
194                 if (rn == 0)
195                     senderr(ETOOMANYREFS);
196                 rt = (struct rtentry *) rn;
197             }
                                                                    rtsock.c
```

Figure 20.10   route_output function: common processing for RTM_GET, RTM_CHANGE, and RTM_LOCK.

Locate existing entry

177–182   Since all three commands reference an existing entry, rtalloc1 locates the entry. If the entry isn't found, ESRCH is returned.

Do not allow network match

183–187   For the RTM_CHANGE and RTM_LOCK commands, a network match is inadequate: an exact match with the routing table key is required. Therefore, if the dst argument doesn't equal the routing table key, the match was a network match and ESRCH is returned.

Use network mask to find correct entry

188–193   Even with an exact match, if there are duplicate keys, each with a different network mask, the correct entry must still be located. If a netmask argument was supplied, it is looked up in the mask table (mask_rnhead). If found, the netmask pointer is replaced with the pointer to the mask in the mask tree. Each leaf node in the duplicate key list is examined, looking for an entry with an rn_mask pointer that equals netmask. This test compares the pointers, not the structures that they point to. This works because all masks appear in the mask tree, and only one copy of each unique mask is stored in this tree. In the common case, keys are not duplicated, so the for loop iterates once. If a host entry is being modified, a mask must not be specified and then both netmask and rn_mask are null pointers (which are equal). But if an entry that has an associated mask is being modified, that mask must be specified as the netmask argument.

194–195   If the `for` loop terminates without finding a matching network mask, `ETOOMANYREFS` is returned.

> The comment XXX is because this function must go to all this work to find the desired entry. All these details should be hidden in another function similar to `rtalloc1` that detects a network match and handles a mask argument.

The next part of this function, shown in Figure 20.11, continues processing the `RTM_GET` command. This command is unique among the commands supported by `route_output` in that it can return more data than it was passed. For example, only a single socket address structure is required as input, the destination, but at least two are returned: the destination and its gateway. With regard to Figure 20.6, this means the buffer allocated for `m_copydata` to copy into might need to be increased in size.

```
                                                                  ── rtsock.c
198         switch (rtm->rtm_type) {

199         case RTM_GET:
200             dst = rt_key(rt);
201             gate = rt->rt_gateway;
202             netmask = rt_mask(rt);
203             genmask = rt->rt_genmask;
204             if (rtm->rtm_addrs & (RTA_IFP | RTA_IFA)) {
205                 if (ifp = rt->rt_ifp) {
206                     ifpaddr = ifp->if_addrlist->ifa_addr;
207                     ifaaddr = rt->rt_ifa->ifa_addr;
208                     rtm->rtm_index = ifp->if_index;
209                 } else {
210                     ifpaddr = 0;
211                     ifaaddr = 0;
212                 }
213             }
214             len = rt_msg2(RTM_GET, &info, (caddr_t) 0,
215                             (struct walkarg *) 0);
216             if (len > rtm->rtm_msglen) {
217                 struct rt_msghdr *new_rtm;
218                 R_Malloc(new_rtm, struct rt_msghdr *, len);
219                 if (new_rtm == 0)
220                     senderr(ENOBUFS);
221                 Bcopy(rtm, new_rtm, rtm->rtm_msglen);
222                 Free(rtm);
223                 rtm = new_rtm;
224             }
225             (void) rt_msg2(RTM_GET, &info, (caddr_t) rtm,
226                             (struct walkarg *) 0);
227             rtm->rtm_flags = rt->rt_flags;
228             rtm->rtm_rmx = rt->rt_rmx;
229             rtm->rtm_addrs = info.rti_addrs;
230             break;
                                                                  ── rtsock.c
```

Figure 20.11 `route_output` function: RTM_GET processing.

Return destination, gateway, and masks

198–203   Four pointers are stored in the `rti_info` array: `dst`, `gate`, `netmask`, and `genmask`. The latter two might be null pointers. These pointers in the `info` structure point to the socket address structures that will be returned to the process.

Return interface information

204–213   The process can set the masks `RTA_IFP` and `RTA_IFA` in the `rtm_flags` bitmask. If either or both are set, the process wants to receive the contents of both the `ifaddr` structures pointed to by this routing table entry: the link-level address of the interface (pointed to by `rt_ifp->if_addrlist`) and the protocol address for this entry (pointed to by `rt_ifa->ifa_addr`). The interface index is also returned.

Construct reply

214–224   `rt_msg2` is called with a null third pointer to calculate the length of the routing message corresponding to `RTM_GET` and the addresses pointed to by the `info` structure. If the length of the result message exceeds the length of the input message, then a new buffer is allocated, the input message is copied into the new buffer, the old buffer is released, and `rtm` is set to point to the new buffer.

225–230   `rt_msg2` is called again, this time with a nonnull third pointer, which builds the result message in the buffer. The final three members in the `rt_msghdr` structure are then filled in.

Figure 20.12 shows the processing of the `RTM_CHANGE` and `RTM_LOCK` commands.

Change gateway

231–233   If a `gate` address was passed by the process, `rt_setgate` is called to change the gateway for the entry.

Locate new interface

234–244   The new gateway (if changed) can also require new `rt_ifp` and `rt_ifa` pointers. The process can specify these new values by passing either an `ifpaddr` socket address structure or an `ifaaddr` socket address structure. The former is tried first, and then the latter. If neither is passed by the process, the `rt_ifp` and `rt_ifa` pointers are left alone.

Check if interface changed

245–256   If an interface was located (`ifa` is nonnull), then the existing `rt_ifa` pointer for the route is compared to the new value. If it has changed, new values for `rt_ifp` and `rt_ifa` are stored in the routing table entry. Before doing this the interface request function (if defined) is called with a command of `RTM_DELETE`. The delete is required because the link-layer information from one type of network to another can be quite different, say changing a route from an X.25 network to an Ethernet, and the output routines must be notified.

Update metrics

257–258   The metrics in the routing table entry are updated by `rt_setmetrics`.

rtsock.c
```
231        case RTM_CHANGE:
232            if (gate && rt_setgate(rt, rt_key(rt), gate))
233                senderr(EDQUOT);
234            /* new gateway could require new ifaddr, ifp; flags may also be
235               different; ifp may be specified by ll sockaddr when protocol
236               address is ambiguous */
237            if (ifpaddr && (ifa = ifa_ifwithnet(ifpaddr)) &&
238                (ifp = ifa->ifa_ifp))
239                ifa = ifaof_ifpforaddr(ifaaddr ? ifaaddr : gate,
240                                       ifp);
241            else if ((ifaaddr && (ifa = ifa_ifwithaddr(ifaaddr))) ||
242                     (ifa = ifa_ifwithroute(rt->rt_flags,
243                                            rt_key(rt), gate)))
244                ifp = ifa->ifa_ifp;
245            if (ifa) {
246                struct ifaddr *oifa = rt->rt_ifa;
247                if (oifa != ifa) {
248                    if (oifa && oifa->ifa_rtrequest)
249                        oifa->ifa_rtrequest(RTM_DELETE,
250                                            rt, gate);
251                    IFAFREE(rt->rt_ifa);
252                    rt->rt_ifa = ifa;
253                    ifa->ifa_refcnt++;
254                    rt->rt_ifp = ifp;
255                }
256            }
257            rt_setmetrics(rtm->rtm_inits, &rtm->rtm_rmx,
258                          &rt->rt_rmx);
259            if (rt->rt_ifa && rt->rt_ifa->ifa_rtrequest)
260                rt->rt_ifa->ifa_rtrequest(RTM_ADD, rt, gate);
261            if (genmask)
262                rt->rt_genmask = genmask;
263            /*
264             * Fall into
265             */
266        case RTM_LOCK:
267            rt->rt_rmx.rmx_locks &= ~(rtm->rtm_inits);
268            rt->rt_rmx.rmx_locks |=
269                (rtm->rtm_inits & rtm->rtm_rmx.rmx_locks);
270            break;
271        }
272        break;

273    default:
274        senderr(EOPNOTSUPP);
275    }
```
rtsock.c

Figure 20.12   route_output function: RTM_CHANGE and RTM_LOCK processing.

Call interface request function

259–260   If an interface request function is defined, it is called with a command of RTM_ADD.

Store clone generation mask

261–262   If the process specifies the genmask argument, the pointer to the mask that was obtained in Figure 20.8 is saved in rt_genmask.

Update bitmask of locked metrics

266–270   The RTM_LOCK command updates the bitmask stored in rt_rmx.rmx_locks. Figure 20.13 shows the values of the different bits in this bitmask, one value per metric.

Constant	Value	Description
RTV_MTU	0x01	initialize or lock rmx_mtu
RTV_HOPCOUNT	0x02	initialize or lock rmx_hopcount
RTV_EXPIRE	0x04	initialize or lock rmx_expire
RTV_RPIPE	0x08	initialize or lock rmx_recvpipe
RTV_SPIPE	0x10	initialize or lock rmx_sendpipe
RTV_SSTHRESH	0x20	initialize or lock rmx_ssthresh
RTV_RTT	0x40	initialize or lock rmx_rtt
RTV_RTTVAR	0x80	initialize or lock rmx_rttvar

Figure 20.13   Constants to initialize or lock metrics.

The rmx_locks member of the rt_metrics structure in the routing table entry is the bitmask telling the kernel which metrics to leave alone. That is, those metrics specified by rmx_locks won't be updated by the kernel. The only use of these metrics by the kernel is with TCP, as noted with Figure 27.3. The rmx_pksent metric cannot be locked or initialized, but it turns out this member is never even referenced or updated by the kernel.

The rtm_inits value in the message from the process specifies the bitmask of which metrics were just initialized by rt_setmetrics. The rtm_rmx.rmx_locks value in the message specifies the bitmask of which metrics should now be locked. The value of rt_rmx.rmx_locks is the bitmask in the routing table of which metrics are currently locked. First, any bits to be initialized (rtm_inits) are unlocked. Any bits that are both initialized (rtm_inits) and locked (rtm_rmx.rmx_locks) are locked.

273–275   This default is for the switch at the beginning of Figure 20.9 and catches any of the routing commands other than the five that are supported in messages from a process.

The final part of route_output, shown in Figure 20.14, sends the reply to raw_input.

—————————————————————————————————— rtsock.c

```
276   flush:
277      if (rtm) {
278           if (error)
279                rtm->rtm_errno = error;
280           else
281                rtm->rtm_flags |= RTF_DONE;
282      }
283      if (rt)
284           rtfree(rt);
285      {
286           struct rawcb *rp = 0;
287           /*
288            * Check to see if we don't want our own messages.
289            */
290           if ((so->so_options & SO_USELOOPBACK) == 0) {
291                if (route_cb.any_count <= 1) {
292                     if (rtm)
293                          Free(rtm);
294                     m_freem(m);
295                     return (error);
296                }
297                /* There is another listener, so construct message */
298                rp = sotorawcb(so);
299           }
300           if (rtm) {
301                m_copyback(m, 0, rtm->rtm_msglen, (caddr_t) rtm);
302                Free(rtm);
303           }
304           if (rp)
305                rp->rcb_proto.sp_family = 0;       /* Avoid us */
306           if (dst)
307                route_proto.sp_protocol = dst->sa_family;
308           raw_input(m, &route_proto, &route_src, &route_dst);
309           if (rp)
310                rp->rcb_proto.sp_family = PF_ROUTE;
311      }
312      return (error);
313 }
```

—————————————————————————————————— rtsock.c

Figure 20.14 route_output function: pass results to raw_input.

Return error or OK

276–282   flush is the label jumped to by the senderr macro defined at the beginning of the function. If an error occurred it is returned in the rtm_errno member; otherwise the RTF_DONE flag is set.

Release held route

283–284   If a route is being held, it is released. The call to rtalloc1 at the beginning of Figure 20.10 holds the route, if found.

No process to receive message

285–296   The SO_USELOOPBACK socket option is true by default and specifies that the sending process is to receive a copy of each routing message that it writes to a routing socket. (If the sender doesn't receive a copy, it can't receive any of the information returned by RTM_GET.) If that option is not set, and the total count of routing sockets is less than or equal to 1, there are no other processes to receive the message and the sender doesn't want a copy. The buffer and mbuf chain are both released and the function returns.

Other listeners but no loopback copy

297–299   There is at least one other listener but the sending process does not want a copy. The pointer rp, which defaults to null, is set to point to the routing control block for the sender and is also used as a flag that the sender doesn't want a copy.

Convert buffer into mbuf chain

300–303   The buffer is converted back into an mbuf chain (Figure 20.6) and the buffer released.

Avoid loopback copy

304–305   If rp is set, some other process might want the message but the sender does not want a copy. The sp_family member of the sender's routing control block is temporarily set to 0, but the sp_family of the message (the route_proto structure, shown with Figure 19.26) has a family of PF_ROUTE. This trick prevents raw_input from passing a copy of the result to the sending process because raw_input does not pass a copy to any socket with an sp_family of 0.

Set address family of routing message

306–308   If dst is a nonnull pointer, the address family of that socket address structure becomes the protocol of the routing message. With the Internet protocols this value would be PF_INET. A copy is passed to the appropriate listeners by raw_input.

309–313   If the sp_family member in the calling process was temporarily set to 0, it is reset to PF_ROUTE, its normal value.

20.6   rt_xaddrs Function

The rt_xaddrs function is called only once from route_output (Figure 20.8) after the routing message from the process has been copied from the mbuf chain into a buffer and after the bitmask from the process (rtm_addrs) has been copied into the rti_info member of an rt_addrinfo structure. The purpose of rt_xaddrs is to take this bitmask and set the pointers in the rti_info array to point to the corresponding address in the buffer. Figure 20.15 shows the function.

——*rtsock.c*

```
330 #define ROUNDUP(a) \
331     ((a) > 0 ? (1 + (((a) - 1) | (sizeof(long) - 1))) : sizeof(long))
332 #define ADVANCE(x, n) (x += ROUNDUP((n)->sa_len))
```

```
333 static void
334 rt_xaddrs(cp, cplim, rtinfo)
335 caddr_t cp, cplim;
336 struct rt_addrinfo *rtinfo;
337 {
338     struct sockaddr *sa;
339     int     i;

340     bzero(rtinfo->rti_info, sizeof(rtinfo->rti_info));
341     for (i = 0; (i < RTAX_MAX) && (cp < cplim); i++) {
342         if ((rtinfo->rti_addrs & (1 << i)) == 0)
343             continue;
344         rtinfo->rti_info[i] = sa = (struct sockaddr *) cp;
345         ADVANCE(cp, sa);
346     }
347 }
```
———————————————————————————————— rtsock.c

Figure 20.15   rt_xaddrs function: fill rti_into array with pointers.

330–340   The array of pointers is set to 0 so all the pointers to address structures not appearing in the bitmask will be null.

341–347   Each of the 8 (RTAX_MAX) possible bits in the bitmask is tested and, if set, a pointer is stored in the rti_info array to the corresponding socket address structure. The ADVANCE macro takes the sa_len field of the socket address structure, rounds it up to the next multiple of 4 bytes, and increments the pointer cp accordingly.

20.7   rt_setmetrics Function

This function was called twice from route_output: when a new route was added and when an existing route was changed. The rtm_inits member in the routing message from the process specifies which of the metrics the process wants to initialize from the rtm_rmx array. The bit values in the bitmask are shown in Figure 20.13.

Notice that both rtm_addrs and rtm_inits are bitmasks in the message from the process, the former specifying the socket address structures that follow, and the latter specifying which metrics are to be initialized. Socket address structures whose bits don't appear in rtm_addrs don't even appear in the routing message, to save space. But the entire rt_metrics array always appears in the fixed-length rt_msghdr structure—elements in the array whose bits are not set in rtm_inits are ignored.

Figure 20.16 shows the rt_setmetrics function.

314–318   The which argument is always the rtm_inits member of the routing message from the process. in points to the rt_metrics structure from the process, and out points to the rt_metrics structure in the routing table entry that is being created or modified.

319–329   Each of the 8 bits in the bitmask is tested and if set, the corresponding metric is copied. Notice that when a new routing table entry is being created with the RTM_ADD command, route_output calls rtrequest, which sets the entire routing table entry to 0 (Figure 19.9). Hence, any metrics not specified by the process in the routing message default to 0.

―― *rtsock.c*
```
314 void
315 rt_setmetrics(which, in, out)
316 u_long  which;
317 struct rt_metrics *in, *out;
318 {
319 #define metric(f, e) if (which & (f)) out->e = in->e;
320     metric(RTV_RPIPE, rmx_recvpipe);
321     metric(RTV_SPIPE, rmx_sendpipe);
322     metric(RTV_SSTHRESH, rmx_ssthresh);
323     metric(RTV_RTT, rmx_rtt);
324     metric(RTV_RTTVAR, rmx_rttvar);
325     metric(RTV_HOPCOUNT, rmx_hopcount);
326     metric(RTV_MTU, rmx_mtu);
327     metric(RTV_EXPIRE, rmx_expire);
328 #undef metric
329 }
```
―― *rtsock.c*

Figure 20.16   rt_setmetrics function: set elements of the rt_metrics structure.

20.8   `raw_input` Function

All routing messages destined for a process—those that originate from within the kernel and those that originate from a process—are given to raw_input, which selects the processes to receive the message. Figure 18.11 summarizes the four functions that call raw_input.

When a routing socket is created, the family is always PF_ROUTE and the protocol, the third argument to socket, can be 0, which means the process wants to receive all routing messages, or a value such as AF_INET, which restricts the socket to messages containing addresses of that specific protocol family. A routing control block is created for each routing socket (Section 20.3) and these two values are stored in the sp_family and sp_protocol members of the rcb_proto structure.

Figure 20.17 shows the raw_input function.

―― *raw_usrreq.c*
```
51 void
52 raw_input(m0, proto, src, dst)
53 struct mbuf *m0;
54 struct sockproto *proto;
55 struct sockaddr *src, *dst;
56 {
57     struct rawcb *rp;
58     struct mbuf *m = m0;
59     int     sockets = 0;
60     struct socket *last;
```