10

IP Fragmentation and Reassembly

10.1 Introduction

In this chapter we describe the IP fragmentation and reassembly processing that we postponed in Chapter 8.

IP has an important capability of being able to fragment a packet when it is too large to be transmitted by the selected hardware interface. The oversized packet is split into two or more IP fragments, each of which is small enough to be transmitted on the selected network. Fragments may be further split by routers farther along the path to the final destination. Thus, at the destination host, an IP datagram can be contained in a single IP packet or, if it was fragmented in transit, it can arrive in multiple IP packets. Because individual fragments may take different paths to the destination host, only the destination host has a chance to see all the fragments. Thus only the destination host can reassemble the fragments into a complete datagram to be delivered to the appropriate transport protocol.

Figure 8.5 shows that 0.3% (72,786/27,881,978) of the packets received were fragments and 0.12% (260,484/(29,447,726 − 796,084)) of the datagrams sent were fragmented. On `world.std.com`, 9.5% of the packets received were fragments. `world` has more NFS activity, which is a common source of IP fragmentation.

Three fields in the IP header implement fragmentation and reassembly: the identification field (`ip_id`), the flags field (the 3 high-order bits of `ip_off`), and the offset field (the 13 low-order bits of `ip_off`). The flags field is composed of three 1-bit flags. Bit 0 is reserved and must be 0, bit 1 is the "don't fragment" (DF) flag, and bit 2 is the "more fragments" (MF) flag. In Net/3, the flag and offset fields are combined and accessed by `ip_off`, as shown in Figure 10.1.

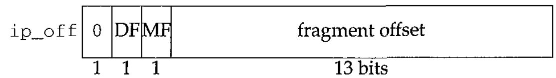

Figure 10.1 `ip_off` controls fragmentation of an IP packet.

Net/3 accesses the DF and MF bits by masking `ip_off` with `IP_DF` and `IP_MF` respectively. An IP implementation must allow an application to request that the DF bit be set in an outgoing datagram.

> Net/3 does not provide *application-level* control over the DF bit when using UDP or TCP.
>
> A process may construct and send its own IP headers with the raw IP interface (Chapter 32). The DF bit may be set by the transport layers directly such as when TCP performs *path MTU discovery*.

The remaining 13 bits of `ip_off` specify the fragment's position within the original datagram, measured in 8-byte units. Accordingly, every fragment except the last must contain a multiple of 8 bytes of data so that the following fragment starts on an 8-byte boundary. Figure 10.2 illustrates the relationship between the byte offset within the original datagram and the fragment offset (low-order 13 bits of `ip_off`) in the fragment's IP header.

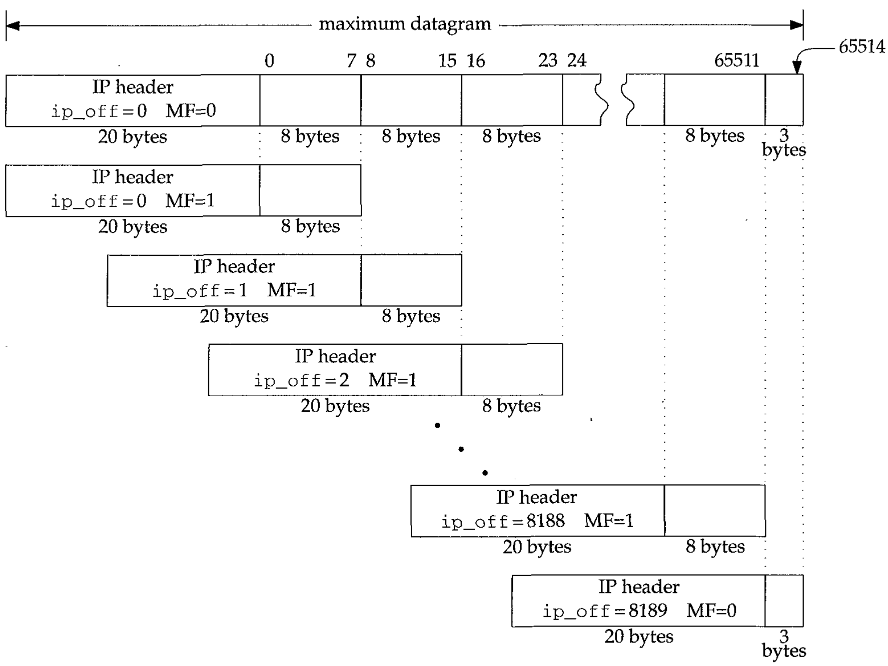

Figure 10.2 Fragmentation of a 65535-byte datagram.

Figure 10.2 shows a maximally sized IP datagram divided into 8190 fragments. Each fragment contains 8 bytes except the last, which contains only 3 bytes. We also show the MF bit set in all the fragments except the last. This is an unrealistic example, but it illustrates several implementation issues.

The numbers above the original datagram are the byte offsets for the *data* portion of the datagram. The fragment offset (ip_off) is computed from the start of the data portion of the datagram. It is impossible for a fragment to include a byte beyond offset 65514 since the reassembled datagram would be larger than 65535 bytes—the maximum value of the ip_len field. This restricts the maximum value of ip_off to 8189 ($8189 \times 8 = 65512$), which leaves room for 3 bytes in the last fragment. If IP options are present, the offset must be smaller still.

Because an IP internet is connectionless, fragments from one datagram may be interleaved with those from another at the destination. ip_id uniquely identifies the fragments of a particular datagram. The source system sets ip_id in each datagram to a unique value for all datagrams using the same source (ip_src), destination (ip_dst), and protocol (ip_p) values for the lifetime of the datagram on the internet.

To summarize, ip_id identifies the fragments of a particular datagram, ip_off positions the fragment within the original datagram, and the MF bit marks every fragment except the last.

10.2 Code Introduction

The reassembly data structures appear in a single header. Reassembly and fragmentation processing is found in two C files. The three files are listed in Figure 10.3.

File	Description
netinet/ip_var.h	reassembly data structures
netinet/ip_output.c	fragmentation code
netinet/ip_input.c	reassembly code

Figure 10.3 Files discussed in this chapter.

Global Variables

Only one global variable, ipq, is described in this chapter.

Variable	Type	Description
ipq	struct ipq *	reassembly list

Figure 10.4 Global variable introduced in this chapter.

Statistics

The statistics modified by the fragmentation and reassembly code are shown in Figure 10.5. They are a subset of the statistics included in the `ipstat` structure described by Figure 8.4.

`ipstat` member	Description
`ips_cantfrag`	#datagrams not sent because fragmentation was required but was prohibited by the DF bit
`ips_odropped`	#output packets dropped because of a memory shortage
`ips_ofragments`	#fragments transmitted
`ips_fragmented`	#packets fragmented for output

Figure 10.5 Statistics collected in this chapter.

10.3 Fragmentation

We now return to `ip_output` and describe the fragmentation code. Recall from Figure 8.25 that if a packet fits within the MTU of the selected outgoing interface, it is transmitted in a single link-level frame. Otherwise the packet must be fragmented and transmitted in multiple frames. A packet may be a complete datagram or it may itself be a fragment that was created by a previous system. We describe the fragmentation code in three parts:

- determine fragment size (Figure 10.6),
- construct fragment list (Figure 10.7), and
- construct initial fragment and send fragments (Figure 10.8).

ip_output.c

```
253     /*
254      * Too large for interface; fragment if possible.
255      * Must be able to put at least 8 bytes per fragment.
256      */
257     if (ip->ip_off & IP_DF) {
258         error = EMSGSIZE;
259         ipstat.ips_cantfrag++;
260         goto bad;
261     }
262     len = (ifp->if_mtu - hlen) & ~7;
263     if (len < 8) {
264         error = EMSGSIZE;
265         goto bad;
266     }
```

ip_output.c

Figure 10.6 `ip_output` function: determine fragment size.

253–261 The fragmentation algorithm is straightforward, but the implementation is complicated by the manipulation of the mbuf structures and chains. If fragmentation is prohibited by the DF bit, ip_output discards the packet and returns EMSGSIZE. If the datagram was generated on this host, a transport protocol passes the error back to the process, but if the datagram is being forwarded, ip_forward generates an ICMP destination unreachable error with an indication that the packet could not be forwarded without fragmentation (Figure 8.21).

Net/3 does not implement the path MTU discovery algorithms used to probe the path to a destination and discover the largest transmission unit supported by all the intervening networks. Sections 11.8 and 24.2 of Volume 1 describe path MTU discovery for UDP and TCP.

262–266 len, the number of data bytes in each fragment, is computed as the MTU of the interface less the size of the packet's header and then rounded down to an 8-byte boundary by clearing the low-order 3 bits (& ~7). If the MTU is so small that each fragment contains less than 8 bytes, ip_output returns EMSGSIZE.

Each new fragment contains an IP header, some of the options from the original packet, and at most len data bytes.

The code in Figure 10.7, which is the start of a C compound statement, constructs the list of fragments starting with the second fragment. The original packet is converted into the initial fragment after the list is created (Figure 10.8).

267–269 The extra block allows mhlen, firstlen, and mnext to be declared closer to their use in the function. These variables are in scope until the end of the block and hide any similarly named variables outside the block.

270–276 Since the original mbuf chain becomes the first fragment, the for loop starts with the offset of the second fragment: hlen + len. For each fragment ip_output takes the following actions:

277–284 • Allocate a new packet mbuf and adjust its m_data pointer to leave room for a 16-byte link-layer header (max_linkhdr). If ip_output didn't do this, the network interface driver would have to allocate an additional mbuf to hold the link header or move the data. Both are time-consuming tasks that are easily avoided here.

285–290 • Copy the IP header and IP options from the original packet into the new packet. The former is copied with a structure assignment. ip_optcopy copies only those options that get copied into each fragment (Section 10.4).

291–297 • Set the offset field (ip_off) for the fragment including the MF bit. If MF is set in the original packet, then MF is set in all the fragments. If MF is not set in the original packet, then MF is set for every fragment except the last.

298 • Set the length of this fragment accounting for a shorter header (ip_optcopy may not have copied all the options) and a shorter data area for the last fragment. The length is stored in network byte order.

299–305 • Copy the data from the original packet into this fragment. m_copy allocates additional mbufs if necessary. If m_copy fails, ENOBUFS is posted. Any mbufs already allocated are discarded at sendorfree.

ip_output.c

```
{
        int     mhlen, firstlen = len;
        struct mbuf **mnext = &m->m_nextpkt;

        /*
         * Loop through length of segment after first fragment,
         * make new header and copy data of each part and link onto chain.
         */
        m0 = m;
        mhlen = sizeof(struct ip);
        for (off = hlen + len; off < (u_short) ip->ip_len; off += len) {
            MGETHDR(m, M_DONTWAIT, MT_HEADER);
            if (m == 0) {
                error = ENOBUFS;
                ipstat.ips_odropped++;
                goto sendorfree;
            }
            m->m_data += max_linkhdr;
            mhip = mtod(m, struct ip *);
            *mhip = *ip;
            if (hlen > sizeof(struct ip)) {
                mhlen = ip_optcopy(ip, mhip) + sizeof(struct ip);
                mhip->ip_hl = mhlen >> 2;
            }
            m->m_len = mhlen;
            mhip->ip_off = ((off - hlen) >> 3) + (ip->ip_off & ~IP_MF);
            if (ip->ip_off & IP_MF)
                mhip->ip_off |= IP_MF;
            if (off + len >= (u_short) ip->ip_len)
                len = (u_short) ip->ip_len - off;
            else
                mhip->ip_off |= IP_MF;
            mhip->ip_len = htons((u_short) (len + mhlen));
            m->m_next = m_copy(m0, off, len);
            if (m->m_next == 0) {
                (void) m_free(m);
                error = ENOBUFS;    /* ??? */
                ipstat.ips_odropped++;
                goto sendorfree;
            }
            m->m_pkthdr.len = mhlen + len;
            m->m_pkthdr.rcvif = (struct ifnet *) 0;
            mhip->ip_off = htons((u_short) mhip->ip_off);
            mhip->ip_sum = 0;
            mhip->ip_sum = in_cksum(m, mhlen);
            *mnext = m;
            mnext = &m->m_nextpkt;
            ipstat.ips_ofragments++;
        }
```

ip_output.c

Figure 10.7 ip_output function: construct fragment list.

306–314 • Adjust the mbuf packet header of the newly created fragment to have the correct total length, clear the new fragment's interface pointer, convert ip_off to network byte order, compute the checksum for the new fragment, and link the fragment to the previous fragment through m_nextpkt.

In Figure 10.8, ip_output constructs the initial fragment and then passes each fragment to the interface layer.

ip_output.c

```
315         /*
316          * Update first fragment by trimming what's been copied out
317          * and updating header, then send each fragment (in order).
318          */
319         m = m0;
320         m_adj(m, hlen + firstlen - (u_short) ip->ip_len);
321         m->m_pkthdr.len = hlen + firstlen;
322         ip->ip_len = htons((u_short) m->m_pkthdr.len);
323         ip->ip_off = htons((u_short) (ip->ip_off | IP_MF));
324         ip->ip_sum = 0;
325         ip->ip_sum = in_cksum(m, hlen);
326       sendorfree:
327         for (m = m0; m; m = m0) {
328             m0 = m->m_nextpkt;
329             m->m_nextpkt = 0;
330             if (error == 0)
331                 error = (*ifp->if_output) (ifp, m,
332                                            (struct sockaddr *) dst, ro->ro_rt);
333             else
334                 m_freem(m);
335         }

336         if (error == 0)
337             ipstat.ips_fragmented++;
338     }
```

ip_output.c

Figure 10.8 ip_output function: send fragments.

315–325 The original packet is converted into the first fragment by trimming the extra data from its end, setting the MF bit, converting ip_len and ip_off to network byte order, and computing the new checksum. All the IP options are retained in this fragment. At the destination host, only the IP options from the first fragment of a datagram are retained when the datagram is reassembled (Figure 10.28). Some options, such as source routing, must be copied into each fragment even though the option is discarded during reassembly.

326–338 At this point, ip_output has either a complete list of fragments or an error has occurred and the partial list of fragments must be discarded. The for loop traverses the list either sending or discarding fragments according to error. Any error encountered while sending fragments causes the remaining fragments to be discarded.

10.4 ip_optcopy Function

During fragmentation, ip_optcopy (Figure 10.9) copies the options from the incoming packet (if the packet is being forwarded) or from the original datagram (if the datagram is locally generated) into the outgoing fragments.

ip_output.c

```
395 int
396 ip_optcopy(ip, jp)
397 struct ip *ip, *jp;
398 {
399     u_char *cp, *dp;
400     int     opt, optlen, cnt;

401     cp = (u_char *) (ip + 1);
402     dp = (u_char *) (jp + 1);
403     cnt = (ip->ip_hl << 2) - sizeof(struct ip);
404     for (; cnt > 0; cnt -= optlen, cp += optlen) {
405         opt = cp[0];
406         if (opt == IPOPT_EOL)
407             break;
408         if (opt == IPOPT_NOP) {
409             /* Preserve for IP mcast tunnel's LSRR alignment. */
410             *dp++ = IPOPT_NOP;
411             optlen = 1;
412             continue;
413         } else
414             optlen = cp[IPOPT_OLEN];
415         /* bogus lengths should have been caught by ip_dooptions */
416         if (optlen > cnt)
417             optlen = cnt;
418         if (IPOPT_COPIED(opt)) {
419             bcopy((caddr_t) cp, (caddr_t) dp, (unsigned) optlen);
420             dp += optlen;
421         }
422     }
423     for (optlen = dp - (u_char *) (jp + 1); optlen & 0x3; optlen++)
424         *dp++ = IPOPT_EOL;
425     return (optlen);
426 }
```

ip_output.c

Figure 10.9 ip_optcopy function.

395–422 The arguments to ip_optcopy are: ip, a pointer to the IP header of the outgoing packet; and jp, a pointer to the IP header of the newly created fragment. ip_optcopy initializes cp and dp to point to the first option byte in each packet and advances cp and dp as it processes each option. The first for loop copies a single option during each iteration stopping when it encounters an EOL option or when it has examined all the options. NOP options are copied to preserve any alignment constraints in the subsequent options.

The Net/2 release discarded NOP options.

If IPOPT_COPIED indicates that the *copied* bit is on, ip_optcopy copies the option to the new fragment. Figure 9.5 shows which options have the *copied* bit set. If an option length is too large, it is truncated; ip_dooptions should have already discovered this type of error.

423–426 The second for loop pads the option list out to a 4-byte boundary. This is required, since the packet's header length (ip_hlen) is measured in 4-byte units. It also ensures that the transport header that follows is aligned on a 4-byte boundary. This improves performance since many transport protocols are designed so that 32-bit header fields are aligned on 32-bit boundaries if the transport header starts on a 32-bit boundary. This arrangement increases performance on CPUs that have difficulty accessing unaligned 32-bit words.

Figure 10.10 illustrates the operation of ip_optcopy.

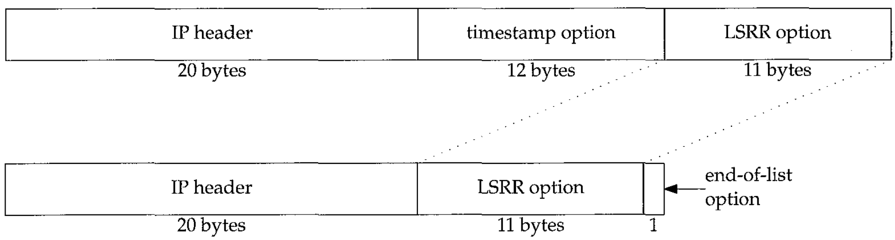

Figure 10.10 Not all options are copied during fragmentation.

In Figure 10.10 we see that ip_optcopy does not copy the timestamp option (its *copied* bit is 0) but does copy the LSRR option (its *copied* bit is 1). ip_optcopy has also added a single EOL option to pad the new options to a 4-byte boundary.

10.5 Reassembly

Now that we have described the fragmentation of a datagram (or of a fragment), we return to ipintr and the reassembly process. In Figure 8.15 we omitted the reassembly code from ipintr and postponed its discussion. ipintr can pass only entire datagrams up to the transport layer for processing. Fragments that are received by ipintr are passed to ip_reass, which attempts to reassemble fragments into complete datagrams. The code from ipintr is shown in Figure 10.11.

271–279 Recall that ip_off contains the DF bit, the MF bit, and the fragment offset. The DF bit is masked out and if either the MF bit or fragment offset is nonzero, the packet is a fragment that must be reassembled. If both are zero, the packet is a complete datagram, the reassembly code is skipped and the else clause at the end of Figure 10.11 is executed, which excludes the header length from the total datagram length.

280–286 m_pullup moves data in an external cluster into the data area of the mbuf. Recall that the SLIP interface (Section 5.3) may return an entire IP packet in an external cluster if it does not fit in a single mbuf. Also m_devget can return the entire packet in a cluster (Section 2.6). Before the mtod macros will work (Section 2.6), m_pullup must move the IP header from the cluster into the data area of an mbuf.

ip_input.c

```
ours:
  /*
   * If offset or IP_MF are set, must reassemble.
   * Otherwise, nothing need be done.
   * (We could look in the reassembly queue to see
   * if the packet was previously fragmented,
   * but it's not worth the time; just let them time out.)
   */
  if (ip->ip_off & ~IP_DF) {
      if (m->m_flags & M_EXT) {    /* XXX */
          if ((m = m_pullup(m, sizeof(struct ip))) == 0) {
              ipstat.ips_toosmall++;
              goto next;
          }
          ip = mtod(m, struct ip *);
      }
      /*
       * Look for queue of fragments
       * of this datagram.
       */
      for (fp = ipq.next; fp != &ipq; fp = fp->next)
          if (ip->ip_id == fp->ipq_id &&
              ip->ip_src.s_addr == fp->ipq_src.s_addr &&
              ip->ip_dst.s_addr == fp->ipq_dst.s_addr &&
              ip->ip_p == fp->ipq_p)
              goto found;
      fp = 0;
    found:

      /*
       * Adjust ip_len to not reflect header,
       * set ip_mff if more fragments are expected,
       * convert offset of this to bytes.
       */
      ip->ip_len -= hlen;
      ((struct ipasfrag *) ip)->ipf_mff &= ~1;
      if (ip->ip_off & IP_MF)
          ((struct ipasfrag *) ip)->ipf_mff |= 1;
      ip->ip_off <<= 3;

      /*
       * If datagram marked as having more fragments
       * or if this is not the first fragment,
       * attempt reassembly; if it succeeds, proceed.
       */
      if (((struct ipasfrag *) ip)->ipf_mff & 1 || ip->ip_off) {
          ipstat.ips_fragments++;
          ip = ip_reass((struct ipasfrag *) ip, fp);
          if (ip == 0)
              goto next;
          ipstat.ips_reassembled++;
          m = dtom(ip);
      } else if (fp)
          ip_freef(fp);
```

```
323     } else
324         ip->ip_len -= hlen;
```
ip_input.c

Figure 10.11 `ipintr` function: fragment processing.

287–297 Net/3 keeps incomplete datagrams on the global doubly linked list, `ipq`. The name is somewhat confusing since the data structure isn't a queue. That is, insertions and deletions can occur anywhere in the list, not just at the ends. We'll use the term *list* to emphasize this fact.

`ipintr` performs a linear search of the list to locate the appropriate datagram for the current fragment. Remember that fragments are uniquely identified by the 4-tuple: {`ip_id`, `ip_src`, `ip_dst`, `ip_p`}. Each entry in `ipq` is a list of fragments and `fp` points to the appropriate list if `ipintr` finds a match.

> Net/3 uses linear searches to access many of its data structures. While simple, this method can become a bottleneck in hosts supporting large numbers of network connections.

298–303 At `found`, the packet is modified by `ipintr` to facilitate reassembly:

304 • `ipintr` changes `ip_len` to exclude the standard IP header and any options. We must keep this in mind to avoid confusion with the standard interpretation of `ip_len`, which includes the standard header, options, and data. `ip_len` is also changed if the reassembly code is skipped because this is not a fragment.

305–307 • `ipintr` copies the MF flag into the low-order bit of `ipf_mff`, which overlays `ip_tos` (`&= ~1` clears the low-order bit only). Notice that `ip` must be cast to a pointer to an `ipasfrag` structure before `ipf_mff` is a valid member. Section 10.6 and Figure 10.14 describe the `ipasfrag` structure.

> Although RFC 1122 requires the IP layer to provide a mechanism that enables the transport layer to set `ip_tos` for every outgoing datagram, it only recommends that the IP layer pass `ip_tos` values to the transport layer at the destination host. Since the low-order bit of the TOS field must always be 0, it is available to hold the MF bit while `ip_off` (where the MF bit is normally found) is used by the reassembly algorithm.

`ip_off` can now be accessed as a 16-bit offset instead of 3 flag bits and a 13-bit offset.

308 • `ip_off` is multiplied by 8 to convert from 8-byte to 1-byte units.

`ipf_mff` and `ip_off` determine if `ipintr` should attempt reassembly. Figure 10.12 describes the different cases and the corresponding actions. Remember that `fp` points to the list of fragments the system has previously received for the datagram. Most of the work is done by `ip_reass`.

309–322 If `ip_reass` is able to assemble a complete datagram by combining the current fragment with previously received fragments, it returns a pointer to the reassembled datagram. If reassembly is not possible, `ip_reass` saves the fragment and `ipintr` jumps to `next` to process the next packet (Figure 8.12).

323–324 This `else` branch is taken when a complete datagram arrives and `ip_hlen` is modified as described earlier. This is the normal flow, since most received datagrams are not fragments.

ip_off	ipf_mff	fp	Description	Action
0	false	null	complete datagram	no assembly required
0	false	nonnull	complete datagram	discard the previous fragments
any	true	null	fragment of new datagram	initialize new fragment list with this fragment
any	true	nonnull	fragment of incomplete datagram	insert into existing fragment list, attempt reassembly
nonzero	false	null	tail fragment of new datagram	initialize new fragment list
nonzero	false	nonnull	tail fragment of incomplete datagram	insert into existing fragment list, attempt reassembly

Figure 10.12 IP fragment processing in `ipintr` and `ip_reass`.

If a complete datagram is available after reassembly processing, it is passed up to the appropriate transport protocol by `ipintr` (Figure 8.15):

```
(*inetsw[ip_protox[ip->ip_p]].pr_input)(m, hlen);
```

10.6 `ip_reass` Function

`ipintr` passes `ip_reass` a fragment to be processed, and a pointer to the matching reassembly header from `ipq`. `ip_reass` attempts to assemble and return a complete datagram or links the fragment into the datagram's reassembly list for reassembly when the remaining fragments arrive. The head of each reassembly list is an `ipq` structure, show in Figure 10.13.

ip_var.h

```
52 struct ipq {
53     struct ipq *next, *prev;     /* to other reass headers */
54     u_char  ipq_ttl;             /* time for reass q to live */
55     u_char  ipq_p;               /* protocol of this fragment */
56     u_short ipq_id;              /* sequence id for reassembly */
57     struct ipasfrag *ipq_next, *ipq_prev;
58     /* to ip headers of fragments */
59     struct in_addr ipq_src, ipq_dst;
60 };
```

ip_var.h

Figure 10.13 `ipq` structure.

52–60 The four fields required to identify a datagram's fragments, `ip_id`, `ip_p`, `ip_src`, and `ip_dst`, are kept in the `ipq` structure at the head of each reassembly list. Net/3 constructs the list of datagrams with `next` and `prev` and the list of fragments with `ipq_next` and `ipq_prev`.

The IP header of incoming IP packets is converted to an `ipasfrag` structure (Figure 10.14) before it is placed on a reassembly list.

ip_var.h

```
66 struct  ipasfrag {
67 #if BYTE_ORDER == LITTLE_ENDIAN
68     u_char  ip_hl:4,
69         ip_v:4;
70 #endif
71 #if BYTE_ORDER == BIG_ENDIAN
72     u_char  ip_v:4,
73         ip_hl:4;
74 #endif
75     u_char  ipf_mff;        /* XXX overlays ip_tos: use low bit
76                              * to avoid destroying tos;
77                              * copied from (ip_off&IP_MF) */
78     short   ip_len;
79     u_short ip_id;
80     short   ip_off;
81     u_char  ip_ttl;
82     u_char  ip_p;
83     u_short ip_sum;
84     struct  ipasfrag *ipf_next; /* next fragment */
85     struct  ipasfrag *ipf_prev; /* previous fragment */
86 };
```

ip_var.h

Figure 10.14 ipasfrag structure.

66–86 ip_reass collects fragments for a particular datagram on a circular doubly linked list joined by the ipf_next and ipf_prev members. These pointers overlay the source and destination addresses in the IP header. The ipf_mff member overlays ip_tos from the ip structure. The other members are the same.

Figure 10.15 illustrates the relationship between the fragment header list (ipq) and the fragments (ipasfrag).

Down the left side of Figure 10.15 is the list of reassembly headers. The first node in the list is the global ipq structure, ipq. It never has a fragment list associated with it. The ipq list is a doubly linked list used to support fast insertions and deletions. The next and prev pointers reference the next or previous ipq structure, which we have shown by terminating the arrows at the corners of the structures.

Each ipq structure is the head node of a circular doubly linked list of ipasfrag structures. Incoming fragments are placed on these fragment lists ordered by their fragment offset. We've highlighted the pointers for these lists in Figure 10.15.

Figure 10.15 still does not show all the complexity of the reassembly structures. The reassembly code is difficult to follow because it relies so heavily on casting pointers to three different structures on the underlying mbuf. We've seen this technique already, for example, when an ip structure overlays the data portion of an mbuf.

Figure 10.16 illustrates the relationship between an mbuf, an ipq structure, an ipasfrag structure, and an ip structure.

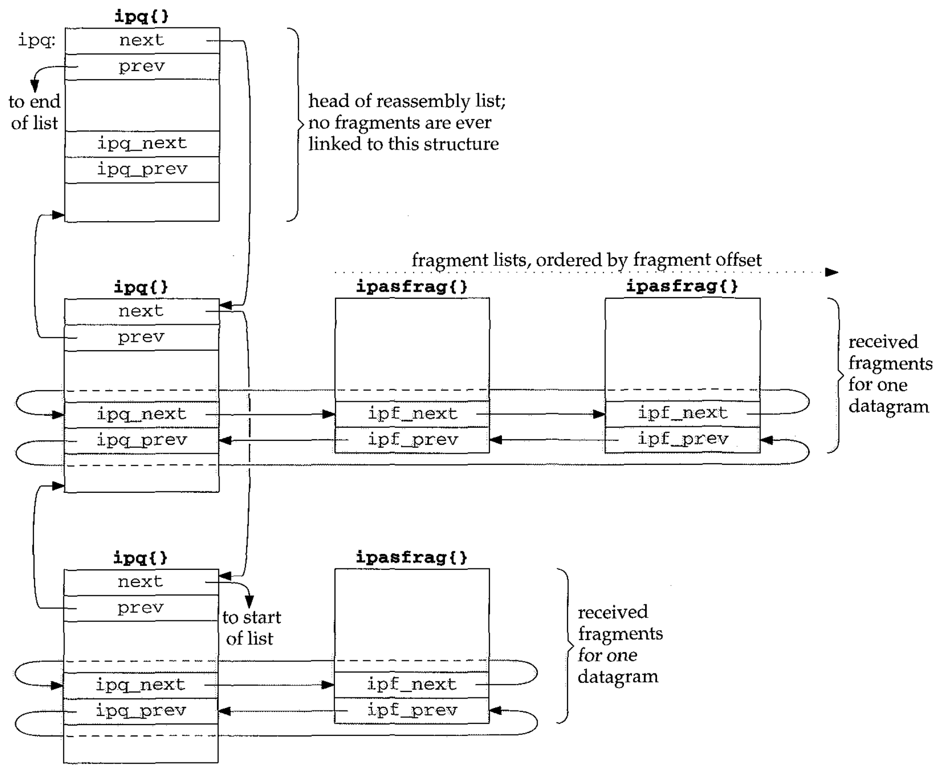

Figure 10.15 The fragment header list, ipq, and fragments.

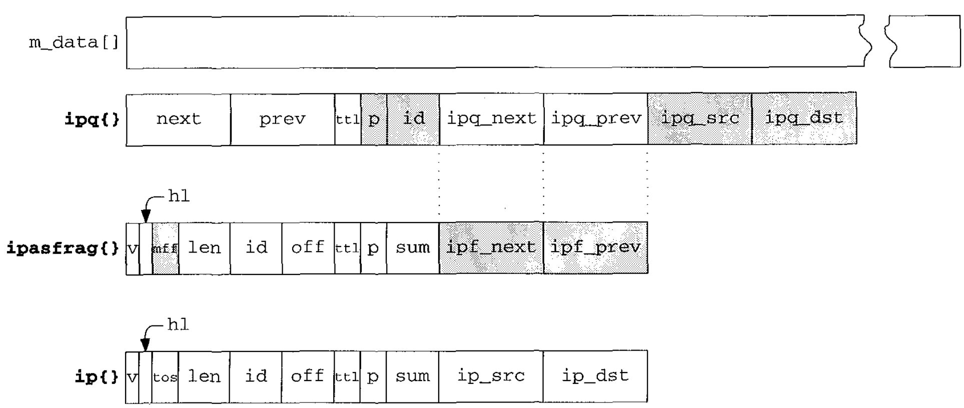

Figure 10.16 An area of memory can be accessed through multiple structures.

A lot of information is contained within Figure 10.16:

- All the structures are located within the data area of an mbuf.
- The ipq list consists of ipq structures joined by next and prev. Within the structure, the four fields that uniquely identify an IP datagram are saved (shaded in Figure 10.16).
- Each ipq structure is treated as an ipasfrag structure when accessed as the head of a linked list of fragments. The fragments are joined by ipf_next and ipf_prev, which overlay the ipq structures' ipq_next and ipq_prev members.
- Each ipasfrag structure overlays the ip structure from the incoming fragment. The data that arrived with the fragment follows the structure in the mbuf. The members that have a different meaning in the ipasfrag structure than they do in the ip structure are shaded.

Figure 10.15 showed the physical connections between the reassembly structures and Figure 10.16 illustrated the overlay technique used by ip_reass. In Figure 10.17 we show the reassembly structures from a logical point of view: this figure shows the reassembly of three datagrams and the relationship between the ipq list and the ipasfrag structures.

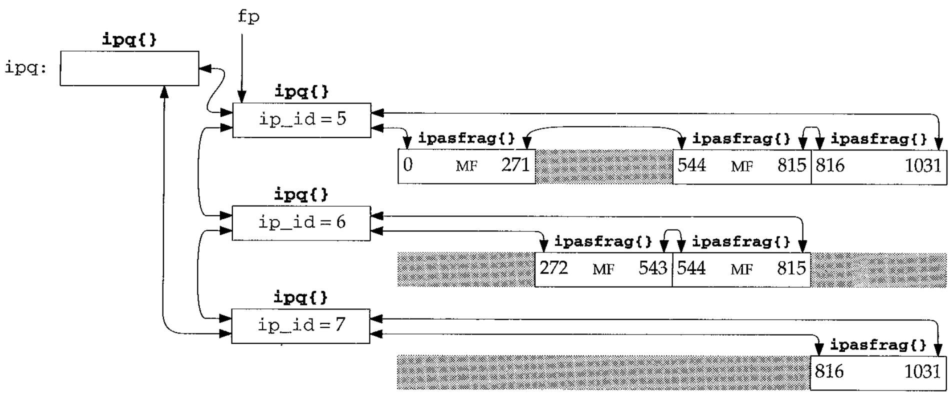

Figure 10.17 Reassembly of three IP datagrams.

The head of each reassembly list contains the id, protocol, source, and destination address of the original datagram. Only the ip_id field is shown in the figure. Each fragment list is ordered by the offset field, the fragment is labeled with MF if the MF bit is set, and missing fragments appear as shaded boxes. The numbers within each fragment show the starting and ending byte offset for the fragment relative to the *data portion* of the original datagram, not to the IP header of the original datagram.

The example is constructed to show three UDP datagrams with no IP options and 1024 bytes of data each. The total length of each datagram is 1052 (20 + 8 + 1024) bytes, which is well within the 1500-byte MTU of an Ethernet. The datagrams encounter a SLIP link on the way to the destination, and the router at that link fragments the datagrams to fit within a typical 296-byte SLIP MTU. Each datagram arrives as four fragments. The first fragment contain a standard 20-byte IP header, the 8-byte UDP header, and 264 bytes of data. The second and third fragments contain a 20-byte IP header and 272 bytes of data. The last fragment has a 20-byte header and 216 bytes of data ($1032 = 272 \times 3 + 216$).

In Figure 10.17, datagram 5 is missing a single fragment containing bytes 272 through 543. Datagram 6 is missing the first fragment, bytes 0 through 271, and the end of the datagram starting at offset 816. Datagram 7 is missing the first three fragments, bytes 0 through 815.

Figure 10.18 lists `ip_reass`. Remember that `ipintr` calls `ip_reass` when an IP fragment has arrived for this host, and after any options have been processed.

ip_input.c

```
337 /*
338  * Take incoming datagram fragment and try to
339  * reassemble it into whole datagram.  If a chain for
340  * reassembly of this datagram already exists, then it
341  * is given as fp; otherwise have to make a chain.
342  */
343 struct ip *
344 ip_reass(ip, fp)
345 struct ipasfrag *ip;
346 struct ipq *fp;
347 {
348     struct mbuf *m = dtom(ip);
349     struct ipasfrag *q;
350     struct mbuf *t;
351     int     hlen = ip->ip_hl << 2;
352     int     i, next;

353     /*
354      * Presence of header sizes in mbufs
355      * would confuse code below.
356      */
357     m->m_data += hlen;
358     m->m_len -= hlen;

                                /* reassembly code */

465   dropfrag:
466     ipstat.ips_fragdropped++;
467     m_freem(m);
468     return (0);
469 }
```

ip_input.c

Figure 10.18 `ip_reass` function: datagram reassembly.

343–358 When `ip_reass` is called, `ip` points to the fragment and `fp` either points to the matching `ipq` structure or is null.

Since reassembly involves only the data portion of each fragment, ip_reass adjusts m_data and m_len from the mbuf containing the fragment to exclude the IP header in each fragment.

465–469 When an error occurs during reassembly, the function jumps to dropfrag, which increments ips_fragdropped, discards the fragment, and returns a null pointer.

Dropping fragments usually incurs a serious performance penalty at the transport layer since the entire datagram must be retransmitted. TCP is careful to avoid fragmentation, but a UDP application must take steps to avoid fragmentation on its own. [Kent and Mogul 1987] explain why fragmentation should be avoided.

All IP implementations must to be able to reassemble a datagram of up to 576 bytes. There is no general way to determine the size of the largest datagram that can be reassembled by a remote host. We'll see in Section 27.5 that TCP has a mechanism to determine the size of the maximum datagram that can be processed by the remote host. UDP has no such mechanism, so many UDP-based protocols (e.g., RIP, TFTP, BOOTP, SNMP, and DNS) are designed around the 576-byte limit.

We'll show the reassembly code in seven parts, starting with Figure 10.19.

ip_input.c

```
359     /*
360      * If first fragment to arrive, create a reassembly queue.
361      */
362     if (fp == 0) {
363         if ((t = m_get(M_DONTWAIT, MT_FTABLE)) == NULL)
364             goto dropfrag;
365         fp = mtod(t, struct ipq *);
366         insque(fp, &ipq);
367         fp->ipq_ttl = IPFRAGTTL;
368         fp->ipq_p = ip->ip_p;
369         fp->ipq_id = ip->ip_id;
370         fp->ipq_next = fp->ipq_prev = (struct ipasfrag *) fp;
371         fp->ipq_src = ((struct ip *) ip)->ip_src;
372         fp->ipq_dst = ((struct ip *) ip)->ip_dst;
373         q = (struct ipasfrag *) fp;
374         goto insert;
375     }
```

ip_input.c

Figure 10.19 ip_reass function: create reassembly list.

Create reassembly list

359–366 When fp is null, ip_reass creates a reassembly list with the first fragment of the new datagram. It allocates an mbuf to hold the head of the new list (an ipq structure), and calls insque to insert the structure in the list of reassembly lists.

Figure 10.20 lists the functions that manipulate the datagram and fragment lists.

> The functions insque and remque are defined in machdep.c for the 386 version of Net/3. Each machine has its own machdep.c file in which customized versions of kernel functions are defined, typically to improve performance. This file also contains architecture-dependent functions such as the interrupt handler support, cpu and device configuration, and memory management functions.

Function	Description
insque	Insert *node* just after *prev*. void **insque**(void **node*, void **prev*);
remque	Remove *node* from list. void **remque**(void **node*);
ip_enq	Insert fragment *p* just after fragment *prev*. void **ip_enq**(struct ipasfrag **p*, struct ipasfrag **prev*);
ip_deq	Remove fragment *p*. void **ip_deq**(struct ipasfrag **p*);

Figure 10.20 Queueing functions used by ip_reass.

insque and remque exist primarily to maintain the kernel's run queue. Net/3 can use them for the datagram reassembly list because both lists have next and previous pointers as the first two members of their respective node structures. These functions work for any similarly structured list, although the compiler may issue some warnings. This is yet another example of accessing memory through two different structures.

In all the kernel structures the next pointer always precedes the previous pointer (Figure 10.14, for example). This is because the insque and remque functions were first implemented on the VAX using the insque and remque hardware instructions, which require this ordering of the forward and backward pointers.

The fragment lists are not joined with the first two members of the ipasfrag structures (Figure 10.14) so Net/3 calls ip_deq and ip_enq instead of insque and remque.

Reassembly timeout

367 The time-to-live field (ipq_ttl) is required by RFC 1122 and limits the time Net/3 waits for fragments to complete a datagram. It is different from the TTL field in the IP header, which limits the amount of time a packet circulates in the internet. The IP header TTL field is reused as the reassembly timeout since the header TTL is not needed once the fragment arrives at its final destination.

In Net/3, the initial value of the reassembly timeout is 60 (IPFRAGTTL). Since ipq_ttl is decremented every time the kernel calls ip_slowtimo and the kernel calls ip_slowtimo every 500 ms, the system discards an IP reassembly list if it hasn't assembled a complete IP datagram within 30 seconds of receiving any one of the datagram's fragments. The reassembly timer starts ticking on the first call to ip_slowtimo after the list is created.

RFC 1122 recommends that the reassembly time be between 60 and 120 seconds and that an ICMP time exceeded error be sent to the source host if the timer expires and the first fragment of the datagram has been received. The header and options of the other fragments are always discarded after reassembly and an ICMP error must contain the first 64 bits of the erroneous datagram (or less if the datagram was shorter than 8 bytes). So, if the kernel hasn't received fragment 0, it can't send an ICMP message.

Net/3's timer is a bit too short and Net/3 neglects to send the ICMP message when a fragment is discarded. The requirement to return the first 64 bits of the datagram ensures that the first portion of the transport header is included, which allows the error message to be returned to the application that generated it. Note that TCP and UDP purposely put their port numbers in the first 8 bytes of their headers for this reason.

Datagram identifiers

368–375 ip_reass saves ip_p, ip_id, ip_src, and ip_dst in the ipq structure allocated for this datagram, points the ipq_next and ipq_prev pointers to the ipq structure (i.e., it constructs a circular list with one node), points q at this structure, and jumps to insert (Figure 10.25) where it inserts the first fragment, ip, into the new reassembly list.

The next part of ip_reass, shown in Figure 10.21, is executed when fp is not null and locates the correct position in the existing list for the new fragment.

ip_input.c

```
376     /*
377      * Find a fragment which begins after this one does.
378      */
379     for (q = fp->ipq_next; q != (struct ipasfrag *) fp; q = q->ipf_next)
380         if (q->ip_off > ip->ip_off)
381             break;
```

ip_input.c

Figure 10.21 ip_reass function: find position in reassembly list.

376–381 Since fp is not null, the for loop searches the datagram's fragment list to locate a fragment with an offset greater than ip_off.

The byte ranges contained within fragments may overlap at the destination. This can happen when a transport-layer protocol retransmits a datagram that gets sent along a route different from the one followed by the original datagram. The fragmentation pattern may also be different resulting in overlaps at the destination. The transport protocol must be able to force IP to use the original ID field in order for the datagram to be recognized as a retransmission at the destination.

Net/3 does not provide a mechanism for a transport protocol to ensure that IP ID fields are reused on a retransmitted datagram. ip_output always assigns a new value by incrementing the global integer ip_id when preparing a new datagram (Figure 8.22). Nevertheless, a Net/3 system could receive overlapping fragments from a system that lets the transport layer retransmit IP datagrams with the same ID field.

Figure 10.22 illustrates the different ways in which the fragment may overlap with existing fragments. The fragments are numbered according to the order in which they *arrive* at the destination host. The reassembled fragment is shown at the bottom of Figure 10.22 The shaded areas of the fragments are the duplicate bytes that are discarded.

In the following discussion, an *earlier* fragment is a fragment that previously arrived at the host.

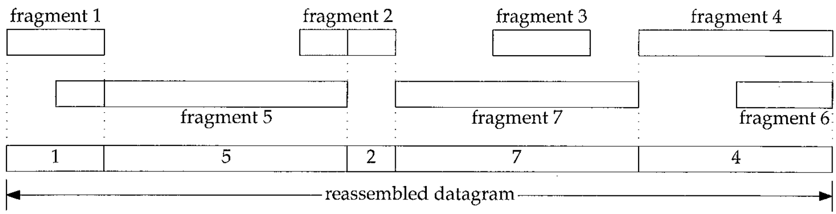

Figure 10.22 The byte range of fragments may overlap at the destination.

The code in Figure 10.23 trims or discards incoming fragments.

382–396 `ip_reass` discards bytes that overlap the end of an earlier fragment by trimming the new fragment (the front of fragment 5 in Figure 10.22) or discarding the new fragment (fragment 6) if all its bytes arrived in an earlier fragment (fragment 4).

The code in Figure 10.24 trims or discards existing fragments.

397–412 If the current fragment partially overlaps the front of an earlier fragment, the duplicate data is trimmed from the earlier fragment (the front of fragment 2 in Figure 10.22). Any earlier fragments that are completely overlapped by the arriving fragment are discarded (fragment 3).

In Figure 10.25, the incoming fragment is inserted into the reassembly list.

413–426 After trimming, `ip_enq` inserts the fragment into the list and the list is scanned to determine if all the fragments have arrived. If any fragment is missing, or the last fragment in the list has `ipf_mff` set, `ip_reass` returns 0 and waits for more fragments.

When the current fragment completes a datagram, the entire list is converted to an mbuf chain by the code shown in Figure 10.26.

427–440 If all the fragments for the datagram have been received, the `while` loop reconstructs the datagram from the fragments with `m_cat`.

Figure 10.27 shows the relationships between mbufs and the `ipq` structure for a datagram composed of three fragments.

The darkest areas in the figure mark the data portions of a packet and the lighter shaded areas mark the unused portions of the mbufs. We show three fragments each contained in a chain of two mbufs; a packet header, and a cluster. The `m_data` pointer in the first mbuf of each fragment points to the packet data, not the packet header. Therefore, the mbuf chain constructed by `m_cat` includes only the data portion of the fragments.

This is the typical scenario when a fragment contains more than 208 bytes of data (Section 2.6). The "frag" portion of the mbufs is the IP header from the fragment. The `m_data` pointer of the first mbuf in each chain points beyond "opts" because of the code in Figure 10.18.

Figure 10.28 shows the reassembled datagram using the mbufs from all the fragments. Notice that the IP header and options from fragments 2 and 3 are not included in the reassembled datagram.

ip_input.c

```
    /*
     * If there is a preceding fragment, it may provide some of
     * our data already.  If so, drop the data from the incoming
     * fragment.  If it provides all of our data, drop us.
     */
    if (q->ipf_prev != (struct ipasfrag *) fp) {
        i = q->ipf_prev->ip_off + q->ipf_prev->ip_len - ip->ip_off;
        if (i > 0) {
            if (i >= ip->ip_len)
                goto dropfrag;
            m_adj(dtom(ip), i);
            ip->ip_off += i;
            ip->ip_len -= i;
        }
    }
```

ip_input.c

Figure 10.23 ip_reass function: trim incoming packet.

ip_input.c

```
    /*
     * While we overlap succeeding fragments trim them or,
     * if they are completely covered, dequeue them.
     */
    while (q != (struct ipasfrag *) fp && ip->ip_off + ip->ip_len > q->ip_off) {
        i = (ip->ip_off + ip->ip_len) - q->ip_off;
        if (i < q->ip_len) {
            q->ip_len -= i;
            q->ip_off += i;
            m_adj(dtom(q), i);
            break;
        }
        q = q->ipf_next;
        m_freem(dtom(q->ipf_prev));
        ip_deq(q->ipf_prev);
    }
```

ip_input.c

Figure 10.24 ip_reass function: trim existing packets.

ip_input.c

```
  insert:
    /*
     * Stick new fragment in its place;
     * check for complete reassembly.
     */
    ip_enq(ip, q->ipf_prev);
    next = 0;
    for (q = fp->ipq_next; q != (struct ipasfrag *) fp; q = q->ipf_next) {
        if (q->ip_off != next)
            return (0);
        next += q->ip_len;
    }
    if (q->ipf_prev->ipf_mff & 1)
        return (0);
```

ip_input.c

Figure 10.25 ip_reass function: insert packet.

ip_input.c

```
    /*
     * Reassembly is complete; concatenate fragments.
     */
    q = fp->ipq_next;
    m = dtom(q);
    t = m->m_next;
    m->m_next = 0;
    m_cat(m, t);
    q = q->ipf_next;
    while (q != (struct ipasfrag *) fp) {
        t = dtom(q);
        q = q->ipf_next;
        m_cat(m, t);
    }
```

ip_input.c

Figure 10.26 `ip_reass` function: reassemble datagram.

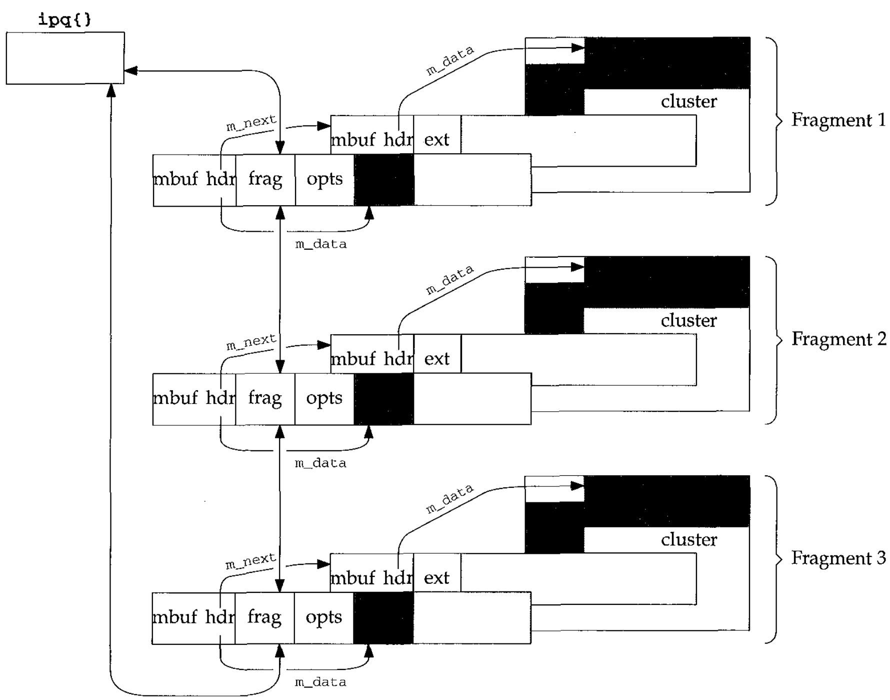

Figure 10.27 `m_cat` reassembles the fragments within mbufs.

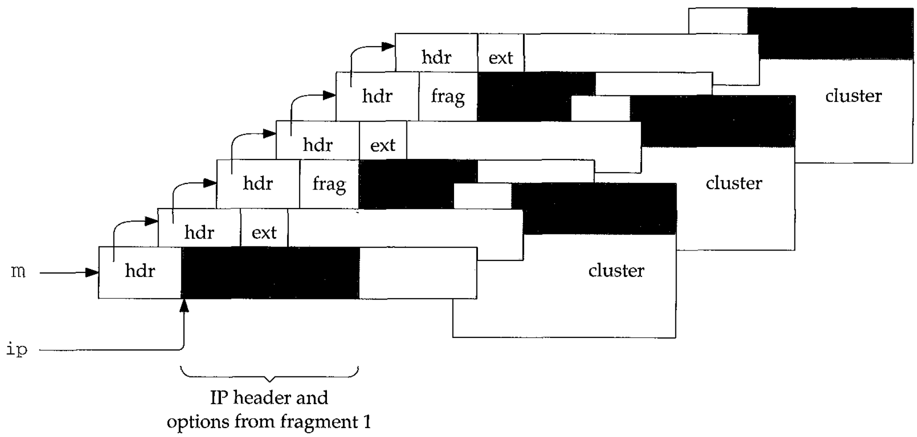

Figure 10.28 The reassembled datagram.

The header of the first fragment is still being used as an ipasfrag structure. It is restored to a valid IP datagram header by the code shown in Figure 10.29.

ip_input.c

```
	/*
	 * Create header for new ip packet by
	 * modifying header of first packet;
	 * dequeue and discard fragment reassembly header.
	 * Make header visible.
	 */
	ip = fp->ipq_next;
	ip->ip_len = next;
	ip->ipf_mff &= ~1;
	((struct ip *) ip)->ip_src = fp->ipq_src;
	((struct ip *) ip)->ip_dst = fp->ipq_dst;
	remque(fp);
	(void) m_free(dtom(fp));
	m = dtom(ip);
	m->m_len += (ip->ip_hl << 2);
	m->m_data -= (ip->ip_hl << 2);
	/* some debugging cruft by sklower, below, will go away soon */
	if (m->m_flags & M_PKTHDR) {     /* XXX this should be done elsewhere */
	    int     plen = 0;
	    for (t = m; m; m = m->m_next)
	        plen += m->m_len;
	    t->m_pkthdr.len = plen;
	}
	return ((struct ip *) ip);
```

ip_input.c

Figure 10.29 ip_reass function: datagram reassembly.

Reconstruct datagram header

441–456 ip_reass points ip to the first fragment in the list and changes the ipasfrag structure back to an ip structure by restoring the length of the datagram to ip_len, the source address to ip_src, the destination address to ip_dst; and by clearing the low-order bit in ipf_mff. (Recall from Figure 10.14 that ipf_mff in the ipasfrag structure overlays ipf_tos in the ip structure.)

ip_reass removes the entire packet from the reassembly list with remque, discards the ipq structure that was the head of the list, and adjusts m_len and m_data in the first mbuf to include the previously hidden IP header and options from the first fragment.

Compute packet length

457–464 The code here is always executed, since the first mbuf for the datagram is always a packet header. The for loop computes the number of data bytes in the mbuf chain and saves the value in m_pkthdr.len.

The purpose of the *copied* bit in the option type field should be clear now. Since the only options retained at the destination are those that appear in the first fragment, only options that control processing of the packet as it travels toward its destination are copied. Options that collect information while in transit are not copied, since the information collected is discarded at the destination when the packet is reassembled.

10.7 ip_slowtimo Function

As shown in Section 7.4, each protocol in Net/3 may specify a function to be called every 500 ms. For IP, that function is ip_slowtimo, shown in Figure 10.30, which times out the fragments on the reassembly list.

515–534 ip_slowtimo traverses the list of partial datagrams and decrements the reassembly TTL field. ip_freef is called if the field drops to 0 to discard the fragments associated with the datagram. ip_slowtimo runs at splnet to prevent the lists from being modified by incoming packets.

ip_freef is shown in Figure 10.31.

470–486 ip_freef removes and releases every fragment on the list pointed to by fp and then releases the list itself.

ip_drain Function

In Figure 7.14 we showed that IP defines ip_drain as the function to be called when the kernel needs additional memory. This usually occurs during mbuf allocation, which we described with (Figure 2.13). ip_drain is shown in Figure 10.32.

538–545 The simplest way for IP to release memory is to discard all the IP fragments on the reassembly list. For IP fragments that belong to a TCP segment, TCP eventually retransmits the data. IP fragments that belong to a UDP datagram are lost and UDP-based protocols must handle this at the application layer.

ip_input.c

```
void
ip_slowtimo(void)
{
    struct ipq *fp;
    int     s = splnet();

    fp = ipq.next;
    if (fp == 0) {
        splx(s);
        return;
    }
    while (fp != &ipq) {
        --fp->ipq_ttl;
        fp = fp->next;
        if (fp->prev->ipq_ttl == 0) {
            ipstat.ips_fragtimeout++;
            ip_freef(fp->prev);
        }
    }
    splx(s);
}
```

ip_input.c

Figure 10.30 ip_slowtimo function.

ip_input.c

```
void
ip_freef(fp)
struct ipq *fp;
{
    struct ipasfrag *q, *p;

    for (q = fp->ipq_next; q != (struct ipasfrag *) fp; q = p) {
        p = q->ipf_next;
        ip_deq(q);
        m_freem(dtom(q));
    }
    remque(fp);
    (void) m_free(dtom(fp));
}
```

ip_input.c

Figure 10.31 ip_freef function.

ip_input.c

```
void
ip_drain()
{

    while (ipq.next != &ipq) {
        ipstat.ips_fragdropped++;
        ip_freef(ipq.next);
    }
}
```

ip_input.c

Figure 10.32 ip_drain function.

10.8 Summary

In this chapter we showed how ip_output splits an outgoing datagram into fragments if it is too large to be transmitted on the selected network. Since fragments may themselves be fragmented as they travel toward their final destination and may take multiple paths, only the destination host can reassemble the original datagram.

ip_reass accepts incoming fragments and attempts to reassemble datagrams. If it is successful, the datagram is passed back to ipintr and then to the appropriate transport protocol. Every IP implementation must reassemble datagrams of up to 576 bytes. The only limit for Net/3 is the number of mbufs that are available. ip_slowtimo discards incomplete datagrams when all their fragments haven't been received within a reasonable amount of time.

Exercises

10.1 Modify ip_slowtimo to send an ICMP time exceeded message when it discards an incomplete datagram (Figure 11.1).

10.2 The recorded route in a fragmented datagram may be different in each fragment. When a datagram is reassembled at the destination host, which return route is available to the transport protocols?

10.3 Draw a picture showing the mbufs involved in the ipq structure and its associated fragment list for the fragment with an ID of 7 in Figure 10.17.

10.4 [Auerbach 1994] suggests that after fragmenting a datagram, the last fragment should be sent first. If the receiving system gets that last fragment first, it can use the offset to allocate an appropriately sized reassembly buffer for the datagram. Modify ip_output to send the last fragment first.

> [Auerbach 1994] notes that some commercial TCP/IP implementations have been known to crash if they receive the last fragment first.

10.5 Use the statistics in Figure 8.5 to answer the following questions. What is the average number of fragments per reassembled datagram? What is the average number of fragments created when an outgoing datagram is fragmented?

10.6 What happens to a packet when the reserved bit in ip_off is set?

11

ICMP: Internet Control Message Protocol

11.1 Introduction

ICMP communicates error and administrative messages between IP systems and is an integral and required part of any IP implementation. The specification for ICMP appears in RFC 792 [Postel 1981b]. RFC 950 [Mogul and Postel 1985] and RFC 1256 [Deering 1991a] define additional ICMP message types. RFC 1122 [Braden 1989a] also provides important details on ICMP.

ICMP has its own transport protocol number (1) allowing ICMP messages to be carried within an IP datagram. Application programs can send and receive ICMP messages directly through the raw IP interface discussed in Chapter 32.

We can divide the ICMP messages into two classes: errors and queries. Query messages are defined in pairs: a request and its reply. ICMP error messages always include the IP header (and options) along with at least the first 8 bytes of the data from the initial fragment of the IP datagram that caused the error. The standard assumes that the 8 bytes includes any demultiplexing information from the transport protocol header of the original packet, which allows a transport protocol to deliver an ICMP error to the correct process.

TCP and UDP port numbers appear within the first 8 bytes of their respective headers.

Figure 11.1 shows all the currently defined ICMP messages. The messages above the double line are ICMP requests and replies; those below the double line are ICMP errors.

type and code	Description	PRC_
ICMP_ECHO	echo request	
ICMP_ECHOREPLY	echo reply	
ICMP_TSTAMP	timestamp request	
ICMP_TSTAMPREPLY	timestamp reply	
ICMP_MASKREQ	address mask request	
ICMP_MASKREPLY	address mask reply	
ICMP_IREQ	information request (obsolete)	
ICMP_IREQREPLY	information reply (obsolete)	
ICMP_ROUTERADVERT	router advertisement	
ICMP_ROUTERSOLICIT	router solicitation	
ICMP_REDIRECT	better route available	
ICMP_REDIRECT_NET	better route available for network	PRC_REDIRECT_HOST
ICMP_REDIRECT_HOST	better route available for host	PRC_REDIRECT_HOST
ICMP_REDIRECT_TOSNET	better route available for TOS and network	PRC_REDIRECT_HOST
ICMP_REDIRECT_TOSHOST	better route available for TOS and host	PRC_REDIRECT_HOST
other	unrecognized code	
ICMP_UNREACH	destination unreachable	
ICMP_UNREACH_NET	network unreachable	PRC_UNREACH_NET
ICMP_UNREACH_HOST	host unreachable	PRC_UNREACH_HOST
ICMP_UNREACH_PROTOCOL	protocol unavailable at destination	PRC_UNREACH_PROTOCOL
ICMP_UNREACH_PORT	port inactive at destination	PRC_UNREACH_PORT
ICMP_UNREACH_SRCFAIL	source route failed	PRC_UNREACH_SRCFAIL
ICMP_UNREACH_NEEDFRAG	fragmentation needed and DF bit set	PRC_MSGSIZE
ICMP_UNREACH_NET_UNKNOWN	destination network unknown	PRC_UNREACH_NET
ICMP_UNREACH_HOST_UNKNOWN	destination host unknown	PRC_UNREACH_HOST
ICMP_UNREACH_ISOLATED	source host isolated	PRC_UNREACH_HOST
ICMP_UNREACH_NET_PROHIB	communication with destination network administratively prohibited	PRC_UNREACH_NET
ICMP_UNREACH_HOST_PROHIB	communication with destination host administratively prohibited	PRC_UNREACH_HOST
ICMP_UNREACH_TOSNET	network unreachable for type of service	PRC_UNREACH_NET
ICMP_UNREACH_TOSHOST	host unreachable for type of service	PRC_UNREACH_HOST
13	communication administratively prohibited by filtering	
14	host precedence violation	
15	precedence cutoff in effect	
other	unrecognized code	
ICMP_TIMXCEED	time exceeded	
ICMP_TIMXCEED_INTRANS	IP time-to-live expired in transit	PRC_TIMXCEED_INTRANS
ICMP_TIMXCEED_REASS	reassembly time-to-live expired	PRC_TIMXCEED_REASS
other	unrecognized code	
ICMP_PARAMPROB	problem with IP header	
0	unspecified header error	PRC_PARAMPROB
ICMP_PARAMPROB_OPTABSENT	required option missing	PRC_PARAMPROB
other	byte offset of invalid byte	
ICMP_SOURCEQUENCH	request to slow transmission	PRC_QUENCH
other	unrecognized type	

Figure 11.1 ICMP message types and codes.

type and code	icmp_input	UDP	TCP	errno
ICMP_ECHO	icmp_reflect			
ICMP_ECHOREPLY	rip_input			
ICMP_TSTAMP	icmp_reflect			
ICMP_TSTAMPREPLY	rip_input			
ICMP_MASKREQ	icmp_reflect			
ICMP_MASKREPLY	rip_input			
ICMP_IREQ	rip_input			
ICMP_IREQREPLY	rip_input			
ICMP_ROUTERADVERT	rip_input			
ICMP_ROUTERSOLICIT	rip_input			
ICMP_REDIRECT				
ICMP_REDIRECT_NET	pfctlinput	in_rtchange	in_rtchange	
ICMP_REDIRECT_HOST	pfctlinput	in_rtchange	in_rtchange	
ICMP_REDIRECT_TOSNET	pfctlinput	in_rtchange	in_rtchange	
ICMP_REDIRECT_TOSHOST	pfctlinput	in_rtchange	in_rtchange	
other	rip_input			
ICMP_UNREACH				
ICMP_UNREACH_NET	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
ICMP_UNREACH_HOST	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
ICMP_UNREACH_PROTOCOL	pr_ctlinput	udp_notify	tcp_notify	ECONNREFUSED
ICMP_UNREACH_PORT	pr_ctlinput	udp_notify	tcp_notify	ECONNREFUSED
ICMP_UNREACH_SRCFAIL	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
ICMP_UNREACH_NEEDFRAG	pr_ctlinput	udp_notify	tcp_notify	EMSGSIZE
ICMP_UNREACH_NET_UNKNOWN	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
ICMP_UNREACH_HOST_UNKNOWN	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
ICMP_UNREACH_ISOLATED	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
ICMP_UNREACH_NET_PROHIB	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
ICMP_UNREACH_HOST_PROHIB	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
ICMP_UNREACH_TOSNET	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
ICMP_UNREACH_TOSHOST	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
13	rip_input			
14	rip_input			
15	rip_input			
other	rip_input			
ICMP_TIMXCEED				
ICMP_TIMXCEED_INTRANS	pr_ctlinput	udp_notify	tcp_notify	
ICMP_TIMXCEED_REASS	pr_ctlinput	udp_notify	tcp_notify	
other	rip_input			
ICMP_PARAMPROB				
0	pr_ctlinput	udp_notify	tcp_notify	ENOPROTOOPT
ICMP_PARAMPROB_OPTABSENT	pr_ctlinput	udp_notify	tcp_notify	ENOPROTOOPT
other	rip_input			
ICMP_SOURCEQUENCH	pr_ctlinput	udp_notify	tcp_quench	
other	rip_input			

Figure 11.2 ICMP message types and codes (continued).

Figures 11.1 and 11.2 contain a lot of information:

- The PRC_ column shows the mapping between the ICMP messages and the protocol-independent error codes processed by Net/3 (Section 11.6). This column is blank for requests and replies, since no error is generated in that case. If this column is blank for an ICMP error, the code is not recognized by Net/3 and the error message is silently discarded.

- Figure 11.3 shows where we discuss each of the functions listed in Figure 11.2.

Function	Description	Reference
icmp_reflect	generate reply to ICMP request	Section 11.12
in_rtchange	update IP routing tables	Figure 22.34
pfctlinput	report error to all protocols	Section 7.7
pr_ctlinput	report error to the protocol associated with the socket	Section 7.4
rip_input	process unrecognized ICMP messages	Section 32.5
tcp_notify	ignore or report error to process	Figure 27.12
tcp_quench	slow down the output	Figure 27.13
udp_notify	report error to process	Figure 23.31

Figure 11.3 Functions called during ICMP input processing.

- The icmp_input column shows the function called by icmp_input for each ICMP message.
- The UDP column shows the functions that process ICMP messages for UDP sockets.
- The TCP column shows the functions that process ICMP messages for TCP sockets. Note that ICMP source quench errors are handled by tcp_quench, not tcp_notify.
- If the errno column is blank, the kernel does not report the ICMP message to the process.
- The last line in the tables shows that unrecognized ICMP messages are delivered to the raw IP protocol where they may be received by processes that have arranged to receive ICMP messages.

In Net/3, ICMP is implemented as a transport-layer protocol above IP and does not generate errors or requests; it formats and sends these messages on behalf of the other protocols. ICMP passes incoming errors and replies to the appropriate transport protocol or to processes that are waiting for ICMP messages. On the other hand, ICMP responds to most incoming ICMP requests with an appropriate ICMP reply. Figure 11.4 summarizes this information.

ICMP message type	Incoming	Outgoing
request	kernel responds with reply	generated by a process
reply	passed to raw IP	generated by kernel
error	passed to transport protocols and raw IP	generated by IP or transport protocols
unknown	passed to raw IP	generated by a process

Figure 11.4 ICMP message processing.

11.2 Code Introduction

The two files listed in Figure 11.5 contain the ICMP data structures, statistics, and processing code described in this chapter.

File	Description
`netinet/ip_icmp.h`	ICMP structure definitions
`netinet/ip_icmp.c`	ICMP processing

Figure 11.5 Files discussed in this chapter.

Global Variables

The global variables shown in Figure 11.6 are introduced in this chapter.

Variable	Type	Description
`icmpmaskrepl`	`int`	enables the return of ICMP address mask replies
`icmpstat`	`struct icmpstat`	ICMP statistics (Figure 11.7)

Figure 11.6 Global variables introduced in this chapter.

Statistics

Statistics are collected by the members of the `icmpstat` structure shown in Figure 11.7.

`icmpstat` member	Description	Used by SNMP
`icps_oldicmp`	#errors discarded because datagram was an ICMP message	•
`icps_oldshort`	#errors discarded because IP datagram was too short	•
`icps_badcode`	#ICMP messages discarded because of an invalid code	•
`icps_badlen`	#ICMP messages discarded because of an invalid ICMP body	•
`icps_checksum`	#ICMP messages discarded because of a bad ICMP checksum	•
`icps_tooshort`	#ICMP messages discarded because of a short ICMP header	•
`icps_outhist[]`	array of output counters; one for each ICMP type	•
`icps_inhist[]`	array of input counters; one for each ICMP type	•
`icps_error`	#of calls to `icmp_error` (excluding redirects)	
`icps_reflect`	#ICMP messages reflected by the kernel	

Figure 11.7 Statistics collected in this chapter.

We'll see where these counters are incremented as we proceed through the code.

Figure 11.8 shows some sample output of these statistics, from the `netstat -s` command.

`netstat -s` output	`icmpstat` member
`84124 calls to icmp_error`	`icps_error`
`0 errors not generated 'cuz old message was icmp`	`icps_oldicmp`
`Output histogram:`	`icps_outhist[]`
`        echo reply: 11770`	`ICMP_ECHOREPLY`
`        destination unreachable: 84118`	`ICMP_UNREACH`
`        time exceeded: 6`	`ICMP_TIMXCEED`
`6 messages with bad code fields`	`icps_badcode`
`0 messages < minimum length`	`icps_badlen`
`0 bad checksums`	`icps_checksum`
`143 messages with bad length`	`icps_tooshort`
`Input histogram:`	`icps_inhist[]`
`        echo reply: 793`	`ICMP_ECHOREPLY`
`        destination unreachable: 305869`	`ICMP_UNREACH`
`        source quench: 621`	`ICMP_SOURCEQUENCH`
`        routing redirect: 103`	`ICMP_REDIRECT`
`        echo: 11770`	`ICMP_ECHO`
`        time exceeded: 25296`	`ICMP_TIMXCEED`
`11770 message responses generated`	`icps_reflect`

Figure 11.8 Sample ICMP statistics.

SNMP Variables

Figure 11.9 shows the relationship between the variables in the SNMP ICMP group and the statistics collected by Net/3.

SNMP variable	icmpstat member	Description
icmpInMsgs	see text	#ICMP messages received
icmpInErrors	icps_badcode + icps_badlen + icps_checksum + icps_tooshort	#ICMP messages discarded because of an error
icmpInDestUnreachs icmpInTimeExcds icmpInParmProbs icmpInSrcQuenchs icmpInRedirects icmpInEchos icmpInEchoReps icmpInTimestamps icmpInTimestampReps icmpInAddrMasks icmpInAddrMaskReps	icps_inhist[] counter	#ICMP messages received for each type
icmpOutMsgs icmpOutErrors	see text icps_oldicmp + icps_oldshort	#ICMP messages sent #ICMP errors not sent because of an error
icmpOutDestUnreachs icmpOutTimeExcds icmpOutParmProbs icmpOutSrcQuenchs icmpOutRedirects icmpOutEchos icmpOutEchoReps icmpOutTimestamps icmpOutTimestampReps icmpOutAddrMasks icmpOutAddrMaskReps	icps_outhist[] counter	#ICMP messages sent for each type

Figure 11.9 *Simple SNMP variables in ICMP group.*

icmpInMsgs is the sum of the counts in the icps_inhist array and icmpInErrors, and icmpOutMsgs is the sum of the counts in the icps_outhist array and icmpOutErrors.

11.3 icmp Structure

Net/3 accesses an ICMP message through the icmp structure shown in Figure 11.10.

ip_icmp.h

```
struct icmp {
    u_char  icmp_type;          /* type of message, see below */
    u_char  icmp_code;          /* type sub code */
    u_short icmp_cksum;         /* ones complement cksum of struct */
    union {
        u_char  ih_pptr;        /* ICMP_PARAMPROB */
        struct in_addr ih_gwaddr;   /* ICMP_REDIRECT */
        struct ih_idseq {
            n_short icd_id;
            n_short icd_seq;
        } ih_idseq;
        int     ih_void;

        /* ICMP_UNREACH_NEEDFRAG -- Path MTU Discovery (RFC1191) */
        struct ih_pmtu {
            n_short ipm_void;
            n_short ipm_nextmtu;
        } ih_pmtu;
    } icmp_hun;
#define icmp_pptr   icmp_hun.ih_pptr
#define icmp_gwaddr icmp_hun.ih_gwaddr
#define icmp_id     icmp_hun.ih_idseq.icd_id
#define icmp_seq    icmp_hun.ih_idseq.icd_seq
#define icmp_void   icmp_hun.ih_void
#define icmp_pmvoid icmp_hun.ih_pmtu.ipm_void
#define icmp_nextmtu    icmp_hun.ih_pmtu.ipm_nextmtu
    union {
        struct id_ts {
            n_time  its_otime;
            n_time  its_rtime;
            n_time  its_ttime;
        } id_ts;
        struct id_ip {
            struct ip idi_ip;
            /* options and then 64 bits of data */
        } id_ip;
        u_long  id_mask;
        char    id_data[1];
    } icmp_dun;
#define icmp_otime  icmp_dun.id_ts.its_otime
#define icmp_rtime  icmp_dun.id_ts.its_rtime
#define icmp_ttime  icmp_dun.id_ts.its_ttime
#define icmp_ip     icmp_dun.id_ip.idi_ip
#define icmp_mask   icmp_dun.id_mask
#define icmp_data   icmp_dun.id_data
};
```

ip_icmp.h

Figure 11.10 icmp structure.

42–45 icmp_type identifies the particular message, and icmp_code further specifies the message (the first column of Figure 11.1). icmp_cksum is computed with the same algorithm as the IP header checksum and protects the entire ICMP message (not just the header as with IP).

46–79 The unions icmp_hun (header union) and icmp_dun (data union) access the various ICMP messages according to icmp_type and icmp_code. Every ICMP message uses icmp_hun; only some utilize icmp_dun. Unused fields must be set to 0.

80–86 As we have seen with other nested structures (e.g., mbuf, le_softc, and ether_arp) the #define macros simplify access to structure members.

Figure 11.11 shows the overall structure of an ICMP message and reiterates that an ICMP message is encapsulated within an IP datagram. We show the specific structure of each message when we encounter it in the code.

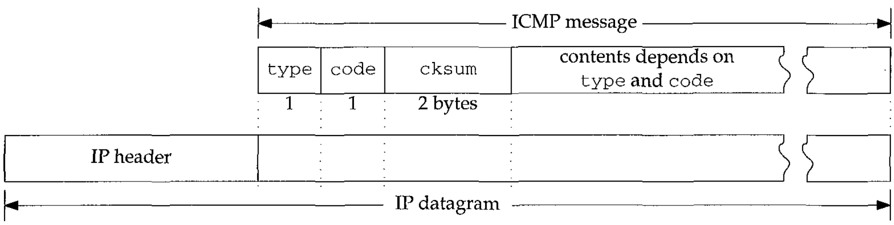

Figure 11.11 An ICMP message (icmp_ omitted).

11.4 ICMP protosw Structure

The protosw structure in inetsw[4] (Figure 7.13) describes ICMP and supports both kernel and process access to the protocol. We show this structure in Figure 11.12. Within the kernel, incoming ICMP messages are processed by icmp_input. Outgoing ICMP messages generated by processes are handled by rip_output. The three functions beginning with rip_ are described in Chapter 32.

Member	inetsw[4]	Description
pr_type	*SOCK_RAW*	ICMP provides raw packet services
pr_domain	*&inetdomain*	ICMP is part of the Internet domain
pr_protocol	*IPPROTO_ICMP (1)*	appears in the ip_p field of the IP header
pr_flags	*PR_ATOMIC|PR_ADDR*	socket layer flags, not used by ICMP
pr_input	*icmp_input*	receives ICMP messages from the IP layer
pr_output	*rip_output*	sends ICMP messages to the IP layer
pr_ctlinput	*0*	not used by ICMP
pr_ctloutput	*rip_ctloutput*	respond to administrative requests from a process
pr_usrreq	*rip_usrreq*	respond to communication requests from a process
pr_init	*0*	not used by ICMP
pr_fasttimo	*0*	not used by ICMP
pr_slowtimo	*0*	not used by ICMP
pr_drain	*0*	not used by ICMP
pr_sysctl	*0*	not used by ICMP

Figure 11.12 ICMP inetsw entry.

11.5 Input Processing: icmp_input Function

Recall that ipintr demultiplexes datagrams based on the transport protocol number, ip_p, in the IP header. For ICMP messages, ip_p is 1, and through ip_protox, it selects inetsw[4].

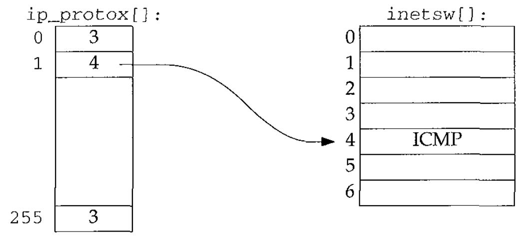

Figure 11.13 An ip_p value of 1 selects inetsw[4].

The IP layer calls icmp_input indirectly through the pr_input function of inetsw[4] when an ICMP message arrives (Figure 10.11).

We'll see in icmp_input that each ICMP message may be processed up to three times: by icmp_input, by the transport protocol associated with the IP packet within an ICMP error message, and by a process that registers interest in receiving ICMP messages. Figure 11.14 shows the overall organization of ICMP input processing.

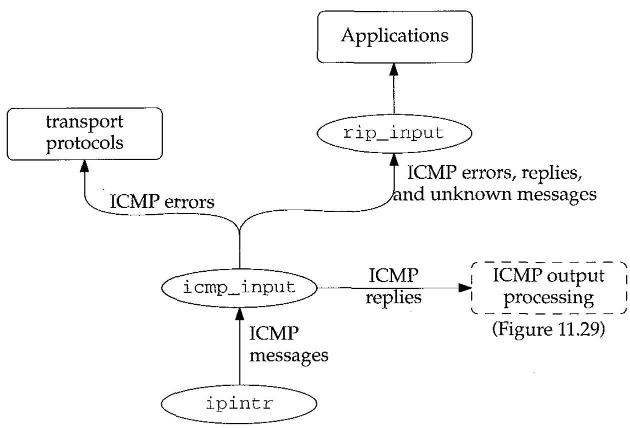

Figure 11.14 ICMP input processing.

We discuss icmp_input in five sections: (1) verification of the received message, (2) ICMP error messages, (3) ICMP requests messages, (4) ICMP redirect messages, (5) ICMP reply messages. Figure 11.15 shows the first portion of the icmp_input function.

ip_icmp.c

```
static struct sockaddr_in icmpsrc = { sizeof (struct sockaddr_in), AF_INET };
static struct sockaddr_in icmpdst = { sizeof (struct sockaddr_in), AF_INET };
static struct sockaddr_in icmpgw = { sizeof (struct sockaddr_in), AF_INET };
struct sockaddr_in icmpmask = { 8, 0 };

void
icmp_input(m, hlen)
struct mbuf *m;
int     hlen;
{
    struct icmp *icp;
    struct ip *ip = mtod(m, struct ip *);
    int     icmplen = ip->ip_len;
    int     i;
    struct in_ifaddr *ia;
    void    (*ctlfunc) (int, struct sockaddr *, struct ip *);
    int     code;
    extern u_char ip_protox[];

    /*
     * Locate icmp structure in mbuf, and check
     * that not corrupted and of at least minimum length.
     */
    if (icmplen < ICMP_MINLEN) {
        icmpstat.icps_tooshort++;
        goto freeit;
    }
    i = hlen + min(icmplen, ICMP_ADVLENMIN);
    if (m->m_len < i && (m = m_pullup(m, i)) == 0) {
        icmpstat.icps_tooshort++;
        return;
    }
    ip = mtod(m, struct ip *);
    m->m_len -= hlen;
    m->m_data += hlen;
    icp = mtod(m, struct icmp *);
    if (in_cksum(m, icmplen)) {
        icmpstat.icps_checksum++;
        goto freeit;
    }
    m->m_len += hlen;
    m->m_data -= hlen;

    if (icp->icmp_type > ICMP_MAXTYPE)
        goto raw;
    icmpstat.icps_inhist[icp->icmp_type]++;
    code = icp->icmp_code;
    switch (icp->icmp_type) {

                        /* ICMP message processing */
```

```
317         default:
318             break;
319         }
320     raw:
321         rip_input(m);
322         return;

323     freeit:
324         m_freem(m);
325 }
```

ip_icmp.c

Figure 11.15 icmp_input function.

Static structures

131–134 These four structures are statically allocated to avoid the delays of dynamic allocation every time icmp_input is called and to minimize the size of the stack since icmp_input is called at interrupt time when the stack size is limited. icmp_input uses these structures as temporary variables.

> The naming of icmpsrc is misleading since icmp_input uses it as a temporary sockaddr_in variable and it never contains a source address. In the Net/2 version of icmp_input, the source address of the message was copied to icmpsrc at the end of the function before the message was delivered to the raw IP mechanism by the raw_input function. Net/3 calls rip_input, which expects only a pointer to the packet instead of raw_input. Despite this change, icmpsrc retains its name from Net/2.

Validate message

135–139 icmp_input expects a pointer to the datagram containing the received ICMP message (m) and the length of the datagram's IP header in bytes (hlen). Figure 11.16 lists several constants that simplify the detection of invalid ICMP messages in icmp_input.

Constant/Macro	Value	Description
ICMP_MINLEN	8	minimum size of an ICMP message
ICMP_TSLEN	20	size of ICMP timestamp messages
ICMP_MASKLEN	12	size of ICMP address mask messages
ICMP_ADVLENMIN	36	minimum size of an ICMP error (advise) message ($IP + ICMP + BADIP = 20 + 8 + 8 = 36$)
ICMP_ADVLEN(p)	36 + *optsize*	size of an ICMP error message including *optsize* bytes of IP options from the invalid packet p.

Figure 11.16 Constants referenced by ICMP to validate messages.

140–160 icmp_input pulls the size of the ICMP message from ip_len and stores it in icmplen. Remember from Chapter 8 that ipintr excludes the length of the header from ip_len. If the message is too short to be a valid ICMP message, icps_tooshort is incremented and the message discarded. If the ICMP header and the IP header are not contiguous in the first mbuf, m_pullup ensures that the ICMP header and the IP header of any enclosed IP packet are in a single mbuf.

Verify checksum

161–170 `icmp_input` hides the IP header in the mbuf and verifies the ICMP checksum with `in_cksum`. If the message is damaged, `icps_checksum` is incremented and the message discarded.

Verify type

171–175 If the message type (`icmp_type`) is out of the recognized range, `icmp_input` jumps around the `switch` to `raw` (Section 11.9). If it is in the recognized range, `icmp_input` duplicates `icmp_code` and the `switch` processes the message according to `icmp_type`.

After the processing within the ICMP `switch` statement, `icmp_input` sends ICMP messages to `rip_input` where they are distributed to processes that are prepared to receive ICMP messages. The only messages that are not passed to `rip_input` are damaged messages (length or checksum errors) and ICMP request messages, which are handled exclusively by the kernel. In both cases, `icmp_input` returns immediately, skipping the code at `raw`.

Raw ICMP input

317–325 `icmp_input` passes the incoming message to `rip_input`, which distributes it to listening processes based on the protocol and the source and destination addresses within the message (Chapter 32).

The raw IP mechanism allows a process to send and to receive ICMP messages directly, which is desirable for several reasons:

- New ICMP messages can be handled by a process without having to modify the kernel (e.g., router advertisement, Figure 11.28).
- Utilities for sending ICMP requests and processing the replies can be implemented as a process instead of as a kernel module (`ping` and `traceroute`).
- A process can augment the kernel processing of a message. This is common with the ICMP redirect messages that are passed to a routing daemon after the kernel has updated its routing tables.

11.6 Error Processing

We first consider the ICMP error messages. A host receives these messages when a datagram that it sent cannot successfully be delivered to its destination. The intended destination host or an intermediate router generates the error message and returns it to the originating system. Figure 11.17 illustrates the format of the various ICMP error messages.

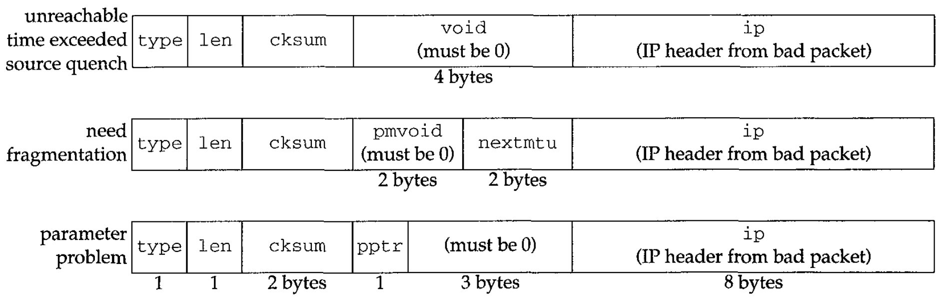

Figure 11.17 ICMP error messages (`icmp_` omitted).

The code in Figure 11.18 is from the `switch` shown in Figure 11.15.

ip_icmp.c

```
    case ICMP_UNREACH:
        switch (code) {
        case ICMP_UNREACH_NET:
        case ICMP_UNREACH_HOST:
        case ICMP_UNREACH_PROTOCOL:
        case ICMP_UNREACH_PORT:
        case ICMP_UNREACH_SRCFAIL:
            code += PRC_UNREACH_NET;
            break;

        case ICMP_UNREACH_NEEDFRAG:
            code = PRC_MSGSIZE;
            break;

        case ICMP_UNREACH_NET_UNKNOWN:
        case ICMP_UNREACH_NET_PROHIB:
        case ICMP_UNREACH_TOSNET:
            code = PRC_UNREACH_NET;
            break;

        case ICMP_UNREACH_HOST_UNKNOWN:
        case ICMP_UNREACH_ISOLATED:
        case ICMP_UNREACH_HOST_PROHIB:
        case ICMP_UNREACH_TOSHOST:
            code = PRC_UNREACH_HOST;
            break;

        default:
            goto badcode;
        }
        goto deliver;

    case ICMP_TIMXCEED:
        if (code > 1)
            goto badcode;
        code += PRC_TIMXCEED_INTRANS;
        goto deliver;
```

```
208     case ICMP_PARAMPROB:
209         if (code > 1)
210             goto badcode;
211         code = PRC_PARAMPROB;
212         goto deliver;

213     case ICMP_SOURCEQUENCH:
214         if (code)
215             goto badcode;
216         code = PRC_QUENCH;

217       deliver:
218         /*
219          * Problem with datagram; advise higher level routines.
220          */
221         if (icmplen < ICMP_ADVLENMIN || icmplen < ICMP_ADVLEN(icp) ||
222             icp->icmp_ip.ip_hl < (sizeof(struct ip) >> 2)) {
223             icmpstat.icps_badlen++;
224             goto freeit;
225         }
226         NTOHS(icp->icmp_ip.ip_len);
227         icmpsrc.sin_addr = icp->icmp_ip.ip_dst;
228         if (ctlfunc = inetsw[ip_protox[icp->icmp_ip.ip_p]].pr_ctlinput)
229             (*ctlfunc) (code, (struct sockaddr *) &icmpsrc,
230                         &icp->icmp_ip);
231         break;

232       badcode:
233         icmpstat.icps_badcode++;
234         break;
```
ip_icmp.c

Figure 11.18 icmp_input function: error messages.

176–216 The processing of ICMP errors is minimal since responsibility for responding to ICMP errors lies primarily with the transport protocols. icmp_input maps icmp_type and icmp_code to a set of protocol-independent error codes represented by the PRC_ constants. There is an implied ordering of the PRC_ constants that matches the ICMP code values. This explains why code is incremented by a PRC_ constant.

If the type and code are recognized, icmp_input jumps to deliver. If the type and code are not recognized, icmp_input jumps to badcode.

217–225 If the message length is incorrect for the error being reported, icps_badlen is incremented and the message discarded. Net/3 always discards invalid ICMP messages, without generating an ICMP error about the invalid message. This prevent an infinite sequence of error messages from forming between two faulty implementations.

226–231 icmp_input calls the pr_ctlinput function of the transport protocol that created the *original* IP datagram by demultiplexing the incoming packets to the correct transport protocol based on ip_p from the original datagram. pr_ctlinput (if it is defined for the protocol) is passed the error code (code), the destination of the original IP datagram (icmpsrc), and a pointer to the invalid datagram (icmp_ip). We discuss these errors with Figures 23.31 and 27.12.

232–234 icps_badcode is incremented and control breaks out of the switch statement.

Constant	Description
PRC_HOSTDEAD	host appears to be down
PRC_IFDOWN	network interface shut down
PRC_MSGSIZE	invalid message size
PRC_PARAMPROB	header incorrect
PRC_QUENCH	someone said to slow down
PRC_QUENCH2	congestion bit says slow down
PRC_REDIRECT_HOST	host routing redirect
PRC_REDIRECT_NET	network routing redirect
PRC_REDIRECT_TOSHOST	redirect for TOS and host
PRC_REDIRECT_TOSNET	redirect for TOS and network
PRC_ROUTEDEAD	select new route if possible
PRC_TIMXCEED_INTRANS	packet lifetime expired in transit
PRC_TIMXCEED_REASS	fragment lifetime expired during reassembly
PRC_UNREACH_HOST	no route available to host
PRC_UNREACH_NET	no route available to network
PRC_UNREACH_PORT	destination says port is not active
PRC_UNREACH_PROTOCOL	destination says protocol is not available
PRC_UNREACH_SRCFAIL	source route failed

Figure 11.19 The protocol-independent error codes.

While the PRC_ constants are ostensibly protocol independent, they are primarily based on the Internet protocols. This results in some loss of specificity when a protocol outside the Internet domain maps its errors to the PRC_ constants.

11.7 Request Processing

Net/3 responds to properly formatted ICMP request messages but passes invalid ICMP request messages to `rip_input`. We show in Chapter 32 how ICMP request messages may be generated by an application process.

Most ICMP request messages received by Net/3 generate a reply message, except the router advertisement message. To avoid allocation of a new mbuf for the reply, `icmp_input` converts the mbuf containing the incoming request to the reply and returns it to the sender. We discuss each request separately.

Echo Query: `ICMP_ECHO` and `ICMP_ECHOREPLY`

For all its simplicity, an ICMP echo request and reply is arguably the single most powerful diagnostic tool available to a network administrator. Sending an ICMP echo request is called *pinging* a host, a reference to the `ping` program that most systems provide for manually sending ICMP echo requests. Chapter 7 of Volume 1 discusses `ping` in detail.

The program `ping` is named after sonar pings used to locate objects by listening for the echo generated as the ping is reflected by the other objects. Volume 1 incorrectly described the name as standing for Packet InterNet Groper.

Figure 11.20 shows the structure of an ICMP echo and reply message.

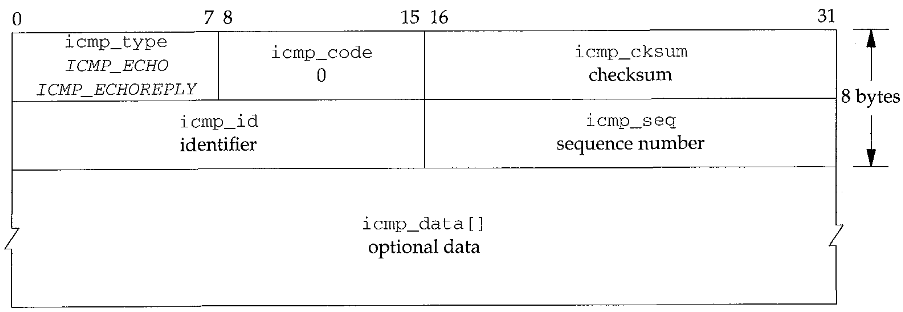

Figure 11.20 ICMP echo request and reply.

`icmp_code` is always 0. `icmp_id` and `icmp_seq` are set by the sender of the request and returned without modification in the reply. The source system can match requests and replies with these fields. Any data that arrives in `icmp_data` is also reflected. Figure 11.21 shows the ICMP echo processing and also the common code in `icmp_input` that implements the reflection of ICMP requests.

ip_icmp.c

```
235     case ICMP_ECHO:
236         icp->icmp_type = ICMP_ECHOREPLY;
237         goto reflect;

                       /* other ICMP request processing */

277       reflect:
278         ip->ip_len += hlen;      /* since ip_input deducts this */
279         icmpstat.icps_reflect++;
280         icmpstat.icps_outhist[icp->icmp_type]++;
281         icmp_reflect(m);
282         return;
```

ip_icmp.c

Figure 11.21 `icmp_input` function: echo request and reply.

235–237 `icmp_input` converts an echo request into an echo reply by changing `icmp_type` to `ICMP_ECHOREPLY` and jumping to `reflect` to send the reply.

277–282 After constructing the reply for each ICMP request, `icmp_input` executes the code at `reflect`. The correct datagram length is restored, the number of requests and the type of ICMP messages are counted in `icps_reflect` and `icps_outhist[]`, and `icmp_reflect` (Section 11.12) sends the reply back to the requestor.

Timestamp Query: ICMP_TSTAMP and ICMP_TSTAMPREPLY

The ICMP timestamp message is illustrated in Figure 11.22.

0 ... 7	8 ... 15	16 ... 31	
icmp_type *ICMP_TSTAMP* *ICMP_TSTAMPREPLY*	icmp_code 0	icmp_cksum checksum	20 bytes
icmp_id identifier		icmp_seq sequence number	
icmp_otime 32-bit originate timestamp			
icmp_rtime 32-bit receive timestamp			
icmp_ttime 32-bit transmit timestamp			

Figure 11.22 ICMP timestamp request and reply.

icmp_code is always 0. icmp_id and icmp_seq serve the same purpose as those in the ICMP echo messages. The sender of the request sets icmp_otime (the time the request originated); icmp_rtime (the time the request was received) and icmp_ttime (the time the reply was transmitted) are set by the sender of the reply. All times are in milliseconds since midnight UTC; the high-order bit is set if the time value is recorded in nonstandard units, as with the IP timestamp option.

Figure 11.23 shows the code that implements the timestamp messages.

ip_icmp.c

```
238     case ICMP_TSTAMP:
239         if (icmplen < ICMP_TSLEN) {
240             icmpstat.icps_badlen++;
241             break;
242         }
243         icp->icmp_type = ICMP_TSTAMPREPLY;
244         icp->icmp_rtime = iptime();
245         icp->icmp_ttime = icp->icmp_rtime;   /* bogus, do later! */
246         goto reflect;
```

ip_icmp.c

Figure 11.23 icmp_input function: timestamp request and reply.

238–246 icmp_input responds to an ICMP timestamp request by changing icmp_type to ICMP_TSTAMPREPLY, recording the current time in icmp_rtime and icmp_ttime, and jumping to reflect to send the reply.

It is difficult to set icmp_rtime and icmp_ttime accurately. When the system executes this code, the message may have already waited on the IP input queue to be processed and icmp_rtime is set too late. Likewise, the datagram still requires processing and may be delayed in the transmit queue of the network interface so icmp_ttime is set too early here. To set the timestamps closer to the true receive and transmit times would require modifying the interface drivers for every network to understand ICMP messages (Exercise 11.8).

Address Mask Query: ICMP_MASKREQ and ICMP_MASKREPLY

The ICMP address mask request and reply are illustrated in Figure 11.24.

0 7	8 15	16 31	
icmp_type *ICMP_MASKREQ* *ICMP_MASKREQREPLY*	icmp_code 0	icmp_cksum checksum	12 bytes
icmp_id identifier		icmp_seq sequence number	
icmp_mask 32-bit subnet mask			

Figure 11.24 ICMP address request and reply.

RFC 950 [Mogul and Postel 1985] added the address mask messages to the original ICMP specification. They enable a system to discover the subnet mask in use on a network.

RFC 1122 forbids sending mask replies unless a system has been explicitly configured as an authoritative agent for address masks. This prevents a system from sharing an incorrect address mask with every system that sends a request. Without administrative authority to respond, a system should ignore address mask requests.

If the global integer icmpmaskrepl is nonzero, Net/3 responds to address mask requests. The default value is 0 and can be changed by icmp_sysctl through the sysctl(8) program (Section 11.14).

In Net/2 systems there was no mechanism to control the reply to address mask requests. As a result, it is very important to configure Net/2 interfaces with the correct address mask; the information is shared with any system on the network that sends an address mask request.

The address mask message processing is shown in Figure 11.25.

247–256 If the system is not configured to respond to mask requests, or if the request is too short, this code breaks out of the switch and passes the message to rip_input (Figure 11.15).

Net/3 fails to increment icps_badlen here. It does increment icps_badlen for all other ICMP length errors.

Select subnet mask

257–267 If the request was sent to 0.0.0.0 or 255.255.255.255, the source address is saved in icmpdst where it is used by ifaof_ifpforaddr to locate the in_ifaddr structure

ip_icmp.c

```
247     case ICMP_MASKREQ:
248 #define satosin(sa) ((struct sockaddr_in *)(sa))
249         if (icmpmaskrepl == 0)
250             break;
251         /*
252          * We are not able to respond with all ones broadcast
253          * unless we receive it over a point-to-point interface.
254          */
255         if (icmplen < ICMP_MASKLEN)
256             break;
257         switch (ip->ip_dst.s_addr) {

258         case INADDR_BROADCAST:
259         case INADDR_ANY:
260             icmpdst.sin_addr = ip->ip_src;
261             break;

262         default:
263             icmpdst.sin_addr = ip->ip_dst;
264         }
265         ia = (struct in_ifaddr *) ifaof_ifpforaddr(
266                         (struct sockaddr *) &icmpdst, m->m_pkthdr.rcvif);
267         if (ia == 0)
268             break;
269         icp->icmp_type = ICMP_MASKREPLY;
270         icp->icmp_mask = ia->ia_sockmask.sin_addr.s_addr;
271         if (ip->ip_src.s_addr == 0) {
272             if (ia->ia_ifp->if_flags & IFF_BROADCAST)
273                 ip->ip_src = satosin(&ia->ia_broadaddr)->sin_addr;
274             else if (ia->ia_ifp->if_flags & IFF_POINTOPOINT)
275                 ip->ip_src = satosin(&ia->ia_dstaddr)->sin_addr;
276         }
```

ip_icmp.c

Figure 11.25 `icmp_input` function: address mask request and reply.

on the same network as the source address. If the source address is 0.0.0.0 or 255.255.255.255, `ifaof_ifpforaddr` returns a pointer to the first IP address associated with the receiving interface.

The `default` case (for unicast or directed broadcasts) saves the destination address for `ifaof_ifpforaddr`.

Convert to reply

269–270 The request is converted into a reply by changing `icmp_type` and by copying the selected subnet mask, `ia_sockmask`, into `icmp_mask`.

Select destination address

271–276 If the source address of the request is all 0s ("this host on this net," which can be used only as a source address during bootstrap, RFC 1122), then the source does not know its own address and Net/3 must broadcast the reply so the source system can receive the message. In this case, the destination for the reply is `ia_broadaddr` or `ia_dstaddr` if the receiving interface is on a broadcast or point-to-point network, respectively. icmp_input puts the destination address for the reply in ip_src since the code at reflect (Figure 11.21) reverses the source and destination addresses. The addresses of a unicast request remain unchanged.

Information Query: ICMP_IREQ and ICMP_IREQREPLY

The ICMP information messages are obsolete. They were intended to allow a host to discover the number of an attached IP network by broadcasting a request with 0s in the network portion of the source and destination address fields. A host responding to the request would return a message with the appropriate network numbers filled in. Some other method was required for a host to discover the host portion of the address.

RFC 1122 recommends that a host not implement the ICMP information messages because RARP (RFC 903 [Finlayson et al. 1984]), and BOOTP (RFC 951 [Croft and Gilmore 1985]) are better suited for discovering addresses. A new protocol, the Dynamic Host Configuration Protocol (DHCP), described in RFC 1541 [Droms 1993], will probably replace and augment the capabilities of BOOTP. It is currently a proposed standard.

Net/2 did respond to ICMP information request messages, but Net/3 passes them on to rip_input.

Router Discovery: ICMP_ROUTERADVERT and ICMP_ROUTERSOLICIT

RFC 1256 defines the ICMP router discovery messages. The Net/3 kernel does not process these messages directly but instead passes them, by rip_input, to a user-level daemon, which sends and responds to the messages.

Section 9.6 of Volume 1 discusses the design and operation of these messages.

11.8 Redirect Processing

Figure 11.26 shows the format of ICMP redirect messages.

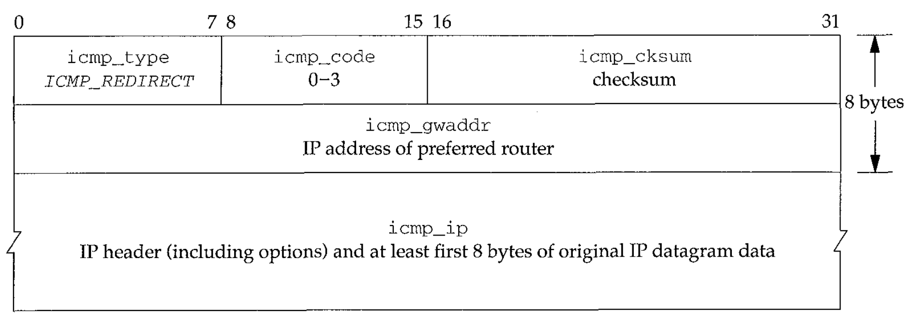

Figure 11.26 ICMP redirect message.

The last `case` to discuss in `icmp_input` is `ICMP_REDIRECT`. As discussed in Section 8.5, a redirect message arrives when a packet is sent to the wrong router. The router forwards the packet to the correct router and sends back a ICMP redirect message, which the system incorporates into its routing tables.

Figure 11.27 shows the code executed by `icmp_input` to process redirect messages.

ip_icmp.c

```
283     case ICMP_REDIRECT:
284         if (code > 3)
285             goto badcode;
286         if (icmplen < ICMP_ADVLENMIN || icmplen < ICMP_ADVLEN(icp) ||
287             icp->icmp_ip.ip_hl < (sizeof(struct ip) >> 2)) {
288             icmpstat.icps_badlen++;
289             break;
290         }
291         /*
292          * Short circuit routing redirects to force
293          * immediate change in the kernel's routing
294          * tables.  The message is also handed to anyone
295          * listening on a raw socket (e.g. the routing
296          * daemon for use in updating its tables).
297          */
298         icmpgw.sin_addr = ip->ip_src;
299         icmpdst.sin_addr = icp->icmp_gwaddr;
300         icmpsrc.sin_addr = icp->icmp_ip.ip_dst;
301         rtredirect((struct sockaddr *) &icmpsrc,
302                    (struct sockaddr *) &icmpdst,
303                    (struct sockaddr *) 0, RTF_GATEWAY | RTF_HOST,
304                    (struct sockaddr *) &icmpgw, (struct rtentry **) 0);
305         pfctlinput(PRC_REDIRECT_HOST, (struct sockaddr *) &icmpsrc);
306         break;
```

ip_icmp.c

Figure 11.27 `icmp_input` function: redirect messages.

Validate

283–290 `icmp_input` jumps to `badcode` (Figure 11.18, line 232) if the redirect message includes an unrecognized ICMP code, and drops out of the switch if the message has an invalid length or if the enclosed IP packet has an invalid header length. Figure 11.16 showed that 36 (`ICMP_ADVLENMIN`) is the minimum size of an ICMP error message, and `ICMP_ADVLEN(icp)` is the minimum size of an ICMP error message including any IP options that may be in the packet pointed to by `icp`.

291–300 `icmp_input` assigns to the static structures `icmpgw`, `icmpdst`, and `icmpsrc`, the source address of the redirect message (the gateway that sent the message), the recommended router for the original packet (the first-hop destination), and the final destination of the original packet.

Here, `icmpsrc` does not contain a source address—it is a convenient location for holding the destination address instead of declaring another `sockaddr` structure.

Update routes

301–306 Net/3 follows RFC 1122 recommendations and treats a network redirect and a host redirect identically. The redirect information is passed to `rtredirect`, which updates the routing tables. The redirected destination (saved in `icmpsrc`) is passed to `pfctlinput`, which informs all the protocol domains about the redirect (Section 7.7). This gives the protocols an opportunity to invalidate any route caches to the destination.

> According to RFC 1122, network redirects should be treated as host redirects since they may provide incorrect routing information when the destination network is subnetted. In fact, RFC 1009 requires routers *not* to send network redirects when the network is subnetted. Unfortunately, many routers violate this requirement. Net/3 never sends network redirects.

ICMP redirect messages are a fundamental part of the IP routing architecture. While classified as an error message, redirect messages appear during normal operations on any network with more than a single router. Chapter 18 covers IP routing issues in more detail.

11.9 Reply Processing

The kernel does not process any of the ICMP reply messages. ICMP requests are generated by processes, never by the kernel, so the kernel passes any replies that it receives to processes waiting for ICMP messages. In addition, the ICMP router discovery messages are passed to `rip_input`.

ip_icmp.c

```
307         /*
308          * No kernel processing for the following;
309          * just fall through to send to raw listener.
310          */
311     case ICMP_ECHOREPLY:
312     case ICMP_ROUTERADVERT:
313     case ICMP_ROUTERSOLICIT:
314     case ICMP_TSTAMPREPLY:
315     case ICMP_IREQREPLY:
316     case ICMP_MASKREPLY:
317     default:
318         break;
319     }
320   raw:
321     rip_input(m);
322     return;
```

ip_icmp.c

Figure 11.28 `icmp_input` function: reply messages.

307–322 No actions are required by the kernel for ICMP reply messages, so execution continues after the `switch` statement at `raw` (Figure 11.15). Note that the `default` case for the `switch` statement (unrecognized ICMP messages) also passes control to the code at `raw`.

11.10 Output Processing

Outgoing ICMP messages are generated in several ways. We saw in Chapter 8 that IP calls icmp_error to generate and send ICMP error messages. ICMP reply messages are sent by icmp_reflect, and it is possible for a process to generate ICMP messages through the raw ICMP protocol. Figure 11.29 shows how these functions relate to ICMP output processing.

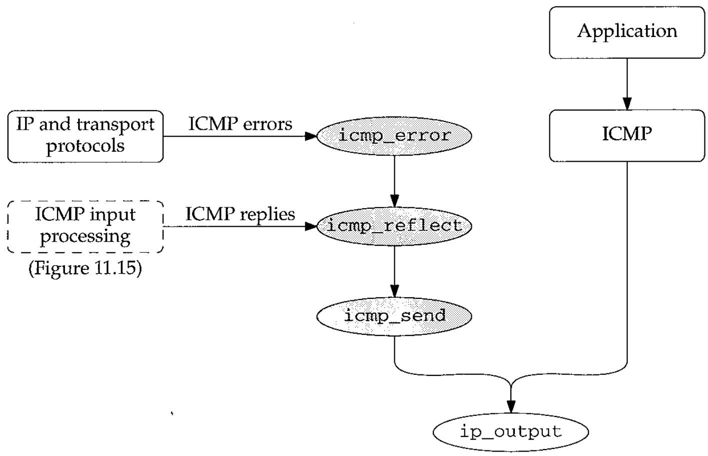

Figure 11.29 ICMP output processing.

11.11 icmp_error Function

The icmp_error function constructs an ICMP error message at the request of IP or the transport protocols and passes it to icmp_reflect, where it is returned to the source of the invalid datagram. The function is shown in three parts:

- validate the message (Figure 11.30),
- construct the header (Figure 11.32), and
- include the original datagram (Figure 11.33).

46–57 The arguments are: n, a pointer to an mbuf chain containing the invalid datagram; type and code, the ICMP error type and code values; dest, the next-hop router address included in ICMP redirect messages; and destifp, a pointer to the outgoing interface for the original IP packet. mtod converts the mbuf pointer n to oip, a pointer to the ip structure in the mbuf. The length in bytes of the original IP packet is kept in oiplen.

58–75 All ICMP errors except redirect messages are counted in icps_error. Net/3 does not consider redirect messages as errors and icps_error is not an SNMP variable.

ip_icmp.c

```
void
icmp_error(n, type, code, dest, destifp)
struct mbuf *n;
int     type, code;
n_long  dest;
struct ifnet *destifp;
{
    struct ip *oip = mtod(n, struct ip *), *nip;
    unsigned oiplen = oip->ip_hl << 2;
    struct icmp *icp;
    struct mbuf *m;
    unsigned icmplen;

    if (type != ICMP_REDIRECT)
        icmpstat.icps_error++;
    /*
     * Don't send error if not the first fragment of message.
     * Don't error if the old packet protocol was ICMP
     * error message, only known informational types.
     */
    if (oip->ip_off & ~(IP_MF | IP_DF))
        goto freeit;
    if (oip->ip_p == IPPROTO_ICMP && type != ICMP_REDIRECT &&
        n->m_len >= oiplen + ICMP_MINLEN &&
        !ICMP_INFOTYPE(((struct icmp *) ((caddr_t) oip + oiplen))->icmp_type)) {
        icmpstat.icps_oldicmp++;
        goto freeit;
    }
    /* Don't send error in response to a multicast or broadcast packet */
    if (n->m_flags & (M_BCAST | M_MCAST))
        goto freeit;
```

ip_icmp.c

Figure 11.30 icmp_error function: validation.

icmp_error discards the invalid datagram, oip, and does not send an error message if:

- some bits of ip_off, except those represented by IP_MF and IP_DF, are nonzero (Exercise 11.10). This indicates that oip is not the first fragment of a datagram and that ICMP must not generate error messages for trailing fragments of a datagram.
- the invalid datagram is itself an ICMP error message. ICMP_INFOTYPE returns true if icmp_type is an ICMP request or response type and false if it is an error type. This rule avoids creating an infinite sequence of errors about errors.

  Net/3 does not consider ICMP redirect messages errors, although RFC 1122 does.

- the datagram arrived as a link-layer broadcast or multicast (indicated by the M_BCAST and M_MCAST flags).

ICMP error messages must not be sent in two other circumstances:

- The datagram was sent to an IP broadcast or IP multicast address.
- The datagram's source address is not a unicast IP address (i.e., the source address is a 0 address, a loopback address, a broadcast address, a multicast address, or a class E address)

Net/3 fails to check for the first case. The second case is addressed by the `icmp_reflect` function (Section 11.12).

> Interestingly, the Deering multicast extensions to Net/2 do discard datagrams of the first type. Since the Net/3 multicast code was derived from the Deering multicast extensions, it appears the test was removed.

These restrictions attempt to prevent a single broadcast datagram with an error from triggering ICMP error messages from every host on the network. These *broadcast storms* can disrupt communication on a network for an extended period of time as all the hosts attempt to send an error message simultaneously.

These rules apply to ICMP error messages but not to ICMP replies. As RFCs 1122 and 1127 discuss, responding to broadcast requests is allowed but neither recommended nor discouraged. Net/3 responds only to broadcast requests with a unicast source address, since `ip_output` will drop ICMP messages returned to a broadcast address (Figure 11.39).

Figure 11.31 illustrates the construction of an ICMP error message.

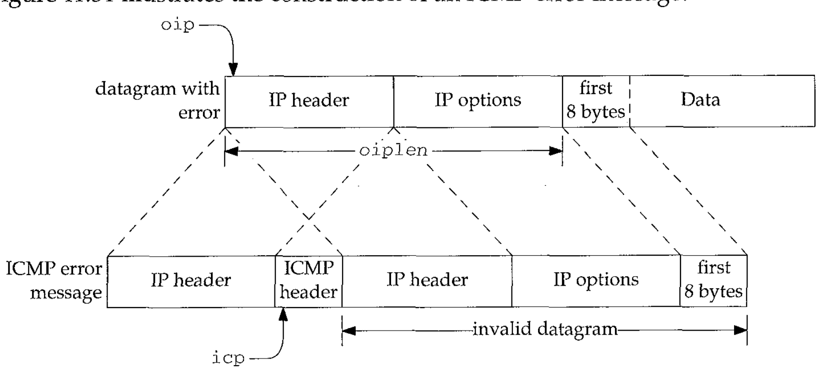

Figure 11.31 The construction of an ICMP error message.

The code in Figure 11.32 builds the error message.

76–106 `icmp_error` constructs the ICMP message header in the following way:

- `m_gethdr` allocates a new packet header mbuf. `MH_ALIGN` positions the mbuf's data pointer so that the ICMP header, the IP header (and options) of the invalid datagram, and up to 8 bytes of the invalid datagram's data are located at the end of the mbuf.

ip_icmp.c

```
    /*
     * First, formulate icmp message
     */
    m = m_gethdr(M_DONTWAIT, MT_HEADER);
    if (m == NULL)
        goto freeit;
    icmplen = oiplen + min(8, oip->ip_len);
    m->m_len = icmplen + ICMP_MINLEN;
    MH_ALIGN(m, m->m_len);
    icp = mtod(m, struct icmp *);
    if ((u_int) type > ICMP_MAXTYPE)
        panic("icmp_error");
    icmpstat.icps_outhist[type]++;
    icp->icmp_type = type;
    if (type == ICMP_REDIRECT)
        icp->icmp_gwaddr.s_addr = dest;
    else {
        icp->icmp_void = 0;
        /*
         * The following assignments assume an overlay with the
         * zeroed icmp_void field.
         */
        if (type == ICMP_PARAMPROB) {
            icp->icmp_pptr = code;
            code = 0;
        } else if (type == ICMP_UNREACH &&
                   code == ICMP_UNREACH_NEEDFRAG && destifp) {
            icp->icmp_nextmtu = htons(destifp->if_mtu);
        }
    }
    icp->icmp_code = code;
```

ip_icmp.c

Figure 11.32 `icmp_error` function: message header construction.

- `icmp_type`, `icmp_code`, `icmp_gwaddr` (for redirects), `icmp_pptr` (for parameter problems), and `icmp_nextmtu` (for the fragmentation required message) are initialized. The `icmp_nextmtu` field implements the extension to the fragmentation required message described in RFC 1191. Section 24.2 of Volume 1 describes the *path MTU discovery* algorithm, which relies on this message.

Once the ICMP header has been constructed, a portion of the original datagram must be attached to the header, as shown in Figure 11.33.

107–125 The IP header, options, and data (a total of `icmplen` bytes) are copied from the invalid datagram into the ICMP error message. Also, the header length is added back into the invalid datagram's `ip_len`.

> In `udp_usrreq`, UDP also adds the header length back into the invalid datagram's `ip_len`. The result is an ICMP message with an incorrect datagram length in the IP header of the invalid packet. The authors found that many systems based on Net/2 code have this bug. Net/1 systems do not have this problem.

ip_icmp.c

```
107         bcopy((caddr_t) oip, (caddr_t) & icp->icmp_ip, icmplen);
108         nip = &icp->icmp_ip;
109         nip->ip_len = htons((u_short) (nip->ip_len + oiplen));

110         /*
111          * Now, copy old ip header (without options)
112          * in front of icmp message.
113          */
114         if (m->m_data - sizeof(struct ip) < m->m_pktdat)
115                 panic("icmp len");
116         m->m_data -= sizeof(struct ip);
117         m->m_len += sizeof(struct ip);
118         m->m_pkthdr.len = m->m_len;
119         m->m_pkthdr.rcvif = n->m_pkthdr.rcvif;
120         nip = mtod(m, struct ip *);
121         bcopy((caddr_t) oip, (caddr_t) nip, sizeof(struct ip));
122         nip->ip_len = m->m_len;
123         nip->ip_hl = sizeof(struct ip) >> 2;
124         nip->ip_p = IPPROTO_ICMP;
125         nip->ip_tos = 0;
126         icmp_reflect(m);

127       freeit:
128         m_freem(n);
129 }
```

ip_icmp.c

Figure 11.33 `icmp_error` function: including the original datagram.

Since `MH_ALIGN` located the ICMP message at the end of the mbuf, there should be enough room to prepend an IP header at the front. The IP header (excluding options) is copied from the invalid datagram to the front of the ICMP message.

> The Net/2 release included a bug in this portion of the code: the last `bcopy` in the function moved `oiplen` bytes, which includes the options from the invalid datagram. Only the standard header without options should be copied.

The IP header is completed by restoring the correct datagram length (`ip_len`), header length (`ip_hl`), and protocol (`ip_p`), and clearing the TOS field (`ip_tos`).

> RFCs 792 and 1122 recommend that the TOS field be set to 0 for ICMP messages.

126–129 The completed message is passed to `icmp_reflect`, where it is sent back to the source host. The invalid datagram is discarded.

11.12 `icmp_reflect` Function

`icmp_reflect` sends ICMP replies and errors back to the source of the request or back to the source of the invalid datagram. It is important to remember that `icmp_reflect` reverses the source and destination addresses in the datagram before sending it. The rules regarding source and destination addresses of ICMP messages are complex. Figure 11.34 summarizes the actions of several functions in this area.

Function	Summary
icmp_input	Replace an all-0s source address in address mask requests with the broadcast or destination address of the receiving interface.
icmp_error	Discard error messages caused by datagrams sent as link-level broadcasts or multicasts. Should discard (but does not) messages caused by datagrams sent to IP broadcast or multicast addresses.
icmp_reflect	Discard messages instead of returning them to a multicast or experimental address. Convert nonunicast destinations to the address of the receiving interface, which makes the destination address a valid source address for the return message. Swap the source and destination addresses.
ip_output	Discards outgoing broadcasts at the request of ICMP (i.e., discards errors generated by packets sent to a broadcast address)

Figure 11.34 ICMP discard and address summary.

We describe the icmp_reflect function in three parts: source and destination address selection, option construction, and assembly and transmission. Figure 11.35 shows the first part of the function.

Set destination address

329–345 icmp_reflect starts by making a copy of ip_dst and moving ip_src, the source of the request or error datagram, to ip_dst. icmp_error and icmp_reflect ensure that ip_src is a valid destination address for the error message. ip_output discards any packets sent to a broadcast address.

Select source address

346–371 icmp_reflect selects a source address for the message by searching in_ifaddr for the interface with a unicast or broadcast address matching the destination address of the original datagram. On a multihomed host, the matching interface may not be the interface on which the datagram was received. If there is no match, the in_ifaddr structure of the receiving interface is selected or, failing that (the interface may not be configured for IP), the first address in in_ifaddr. The function sets ip_src to the selected address and changes ip_ttl to 255 (MAXTTL) because the error is a new datagram.

RFC 1700 recommends that the TTL field of all IP packets be set to 64. Many systems, however, set the TTL of ICMP messages to 255 nowadays.

There is a tradeoff associated with TTL values. A small TTL prevents a packet from circulating in a routing loop but may not allow a packet to reach a site far (many hops) away. A large TTL allows packets to reach distant hosts but lets packets circulate in routing loops for a longer period of time.

ip_icmp.c

```
void
icmp_reflect(m)
struct mbuf *m;
{
    struct ip *ip = mtod(m, struct ip *);
    struct in_ifaddr *ia;
    struct in_addr t;
    struct mbuf *opts = 0, *ip_srcroute();
    int     optlen = (ip->ip_hl << 2) - sizeof(struct ip);

    if (!in_canforward(ip->ip_src) &&
        ((ntohl(ip->ip_src.s_addr) & IN_CLASSA_NET) !=
         (IN_LOOPBACKNET << IN_CLASSA_NSHIFT))) {
        m_freem(m);                 /* Bad return address */
        goto done;                  /* Ip_output() will check for broadcast */
    }
    t = ip->ip_dst;
    ip->ip_dst = ip->ip_src;
    /*
     * If the incoming packet was addressed directly to us,
     * use dst as the src for the reply.  Otherwise (broadcast
     * or anonymous), use the address which corresponds
     * to the incoming interface.
     */
    for (ia = in_ifaddr; ia; ia = ia->ia_next) {
        if (t.s_addr == IA_SIN(ia)->sin_addr.s_addr)
            break;
        if ((ia->ia_ifp->if_flags & IFF_BROADCAST) &&
            t.s_addr == satosin(&ia->ia_broadaddr)->sin_addr.s_addr)
            break;
    }
    icmpdst.sin_addr = t;
    if (ia == (struct in_ifaddr *) 0)
        ia = (struct in_ifaddr *) ifaof_ifpforaddr(
                            (struct sockaddr *) &icmpdst, m->m_pkthdr.rcvif);
    /*
     * The following happens if the packet was not addressed to us,
     * and was received on an interface with no IP address.
     */
    if (ia == (struct in_ifaddr *) 0)
        ia = in_ifaddr;
    t = IA_SIN(ia)->sin_addr;
    ip->ip_src = t;
    ip->ip_ttl = MAXTTL;
```

ip_icmp.c

Figure 11.35 `icmp_reflect` function: address selection.

RFC 1122 *requires* that source route options, and *recommends* that record route and timestamp options, from an incoming echo request or timestamp request, be attached to a reply. The source route must be reversed in the process. RFC 1122 is silent on how these options should be handled on other types of ICMP replies. Net/3 applies these rules to the address mask request, since it calls icmp_reflect (Figure 11.21) after constructing the address mask reply.

The next section of code (Figure 11.36) constructs the options for the ICMP message.

ip_icmp.c

```
    if (optlen > 0) {
        u_char *cp;
        int     opt, cnt;
        u_int   len;

        /*
         * Retrieve any source routing from the incoming packet;
         * add on any record-route or timestamp options.
         */
        cp = (u_char *) (ip + 1);
        if ((opts = ip_srcroute()) == 0 &&
            (opts = m_gethdr(M_DONTWAIT, MT_HEADER))) {
            opts->m_len = sizeof(struct in_addr);
            mtod(opts, struct in_addr *)->s_addr = 0;
        }
        if (opts) {
            for (cnt = optlen; cnt > 0; cnt -= len, cp += len) {
                opt = cp[IPOPT_OPTVAL];
                if (opt == IPOPT_EOL)
                    break;
                if (opt == IPOPT_NOP)
                    len = 1;
                else {
                    len = cp[IPOPT_OLEN];
                    if (len <= 0 || len > cnt)
                        break;
                }
                /*
                 * Should check for overflow, but it "can't happen"
                 */
                if (opt == IPOPT_RR || opt == IPOPT_TS ||
                    opt == IPOPT_SECURITY) {
                    bcopy((caddr_t) cp,
                          mtod(opts, caddr_t) + opts->m_len, len);
                    opts->m_len += len;
                }
            }
            /* Terminate & pad, if necessary */
            if (cnt = opts->m_len % 4) {
                for (; cnt < 4; cnt++) {
                    *(mtod(opts, caddr_t) + opts->m_len) =
                        IPOPT_EOL;
                    opts->m_len++;
                }
            }
    }
```

ip_icmp.c

Figure 11.36 icmp_reflect function: option construction.

Get reversed source route

372–385 If the incoming datagram did not contain options, control passes to line 430 (Figure 11.37). The error messages that `icmp_error` sends to `icmp_reflect` never have IP options, and so the following code applies only to ICMP requests that are converted to replies and passed directly to `icmp_reflect`.

`cp` points to the start of the options for the *reply*. `ip_srcroute` reverses and returns any source route option saved when `ipintr` processed the datagram. If `ip_srcroute` returns 0, the request did not contain a source route option so `icmp_reflect` allocates and initializes an mbuf to serve as an empty `ipoption` structure.

Add record route and timestamp options

386–416 If `opts` points to an mbuf, the `for` loop searches the options from the *original* IP header and appends the record route and timestamp options to the source route returned by `ip_srcroute`.

The options in the original header must be removed before the ICMP message can be sent. This is done by the code shown in Figure 11.37.

ip_icmp.c

```
417             /*
418              * Now strip out original options by copying rest of first
419              * mbuf's data back, and adjust the IP length.
420              */
421             ip->ip_len -= optlen;
422             ip->ip_hl = sizeof(struct ip) >> 2;
423             m->m_len -= optlen;
424             if (m->m_flags & M_PKTHDR)
425                 m->m_pkthdr.len -= optlen;
426             optlen += sizeof(struct ip);
427             bcopy((caddr_t) ip + optlen, (caddr_t) (ip + 1),
428                   (unsigned) (m->m_len - sizeof(struct ip)));
429         }
430         m->m_flags &= ~(M_BCAST | M_MCAST);
431         icmp_send(m, opts);
432   done:
433     if (opts)
434         (void) m_free(opts);
435 }
```

ip_icmp.c

Figure 11.37 `icmp_reflect` function: final assembly.

Remove original options

417–429 `icmp_reflect` removes the options from the original request by moving the ICMP message up to the end of the IP header. This is shown in Figure 11.38). The new options, which are in the mbuf pointed to by `opts`, are reinserted by `ip_output`.

Send message and cleanup

430–435 The broadcast and multicast flags are explicitly cleared before passing the message and options to `icmp_send`, after which the mbuf containing the options is released.

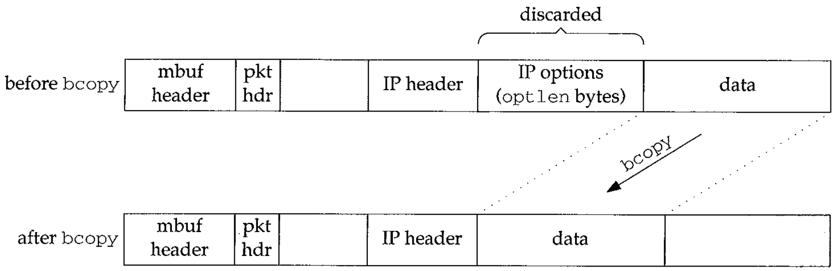

Figure 11.38 icmp_reflect: removal of options.

11.13 icmp_send Function

icmp_send (Figure 11.39) processes all outgoing ICMP messages and computes the ICMP checksum before passing them to the IP layer.

ip_icmp.c

```
440 void
441 icmp_send(m, opts)
442 struct mbuf *m;
443 struct mbuf *opts;
444 {
445     struct ip *ip = mtod(m, struct ip *);
446     int     hlen;
447     struct icmp *icp;

448     hlen = ip->ip_hl << 2;
449     m->m_data += hlen;
450     m->m_len -= hlen;
451     icp = mtod(m, struct icmp *);
452     icp->icmp_cksum = 0;
453     icp->icmp_cksum = in_cksum(m, ip->ip_len - hlen);
454     m->m_data -= hlen;
455     m->m_len += hlen;
456     (void) ip_output(m, opts, NULL, 0, NULL);
457 }
```

ip_icmp.c

Figure 11.39 icmp_send function.

440–457 As it does when checking the ICMP checksum in icmp_input, Net/3 adjusts the mbuf data pointer and length to hide the IP header and lets in_cksum look only at the ICMP message. The computed checksum is placed in the header at icmp_cksum and the datagram and any options are passed to ip_output. The ICMP layer does not maintain a route cache, so icmp_send passes a null pointer to ip_output instead of a route entry as the third argument. icmp_send also does not pass any control flags to ip_output (the fourth argument). In particular, IP_ALLOWBROADCAST isn't passed, so ip_output discards any ICMP messages with a broadcast destination address (i.e., the original datagram arrived with an invalid source address).

11.14 icmp_sysctl Function

The icmp_sysctl function for IP supports the single option listed in Figure 11.40. The system administrator can modify the option through the sysctl(8) program.

sysctl constant	Net/3 variable	Description
ICMPCTL_MASKREPL	icmpmaskrepl	Should system respond to ICMP address mask requests?

Figure 11.40 icmp_sysctl parameters.

Figure 11.41 shows the icmp_sysctl function.

ip_icmp.c

```
467 int
468 icmp_sysctl(name, namelen, oldp, oldlenp, newp, newlen)
469 int     *name;
470 u_int    namelen;
471 void    *oldp;
472 size_t *oldlenp;
473 void    *newp;
474 size_t   newlen;
475 {

476     /* All sysctl names at this level are terminal. */
477     if (namelen != 1)
478         return (ENOTDIR);

479     switch (name[0]) {
480     case ICMPCTL_MASKREPL:
481         return (sysctl_int(oldp, oldlenp, newp, newlen, &icmpmaskrepl));
482     default:
483         return (ENOPROTOOPT);
484     }
485     /* NOTREACHED */
486 }
```

ip_icmp.c

Figure 11.41 icmp_sysctl function.

467–478 ENOTDIR is returned if the required ICMP sysctl name is missing.

479–486 There are no options below the ICMP level, so this function calls sysctl_int to modify icmpmaskrepl or returns ENOPROTOOPT if the option is not recognized.

11.15 Summary

The ICMP protocol is implemented as a transport layer above IP, but it is tightly integrated with the IP layer. We've seen that the kernel responds directly to ICMP request messages but passes errors and replies to the appropriate transport protocol or application program for processing. The kernel makes immediate changes to the routing tables when an ICMP redirect message arrives but also passes redirects to any waiting processes, typically a routing daemon.

In Sections 23.9 and 27.6 we'll see how the UDP and TCP protocols respond to ICMP error messages, and in Chapter 32 we'll see how a process can generate ICMP requests.

Exercises

11.1 What is the source address of an ICMP address mask reply message generated by a request with a destination address of 0.0.0.0?

11.2 Describe how a link-level broadcast of a packet with a forged unicast source address can interfere with the operation of another host on the network.

11.3 RFC 1122 suggests that a host should discard an ICMP redirect message if the new first-hop router is on a different subnet from the old first-hop router or if the message came from a router other than the current first-hop router for the final destination included in the message. Why should this advice be followed?

11.4 If the ICMP information request is obsolete, why does `icmp_input` pass it to `rip_input` instead of discarding it?

11.5 We pointed out that Net/3 does not convert the offset and length field of an IP packet to network byte order before including the packet in an ICMP error message. Why is this inconsequential in the case of the IP offset field?

11.6 Describe a situation in which `ifaof_ifpforaddr` from Figure 11.25 returns a null pointer.

11.7 What happens to data included after the timestamps in a timestamp query?

11.8 Implement the following changes to improve the ICMP timestamp code:

Add a timestamp field to the mbuf packet header. Have the device drivers record the exact time a packet is received in this field and have the ICMP timestamp code copy the value into the `icmp_rtime` field.

On output, have the ICMP timestamp code store the byte offset of where in the packet to store the current time in the timestamp field. Modify a device driver to insert the timestamp right before sending the packet.

11.9 Modify `icmp_error` to return up to 64 bytes (as does Solaris 2.x) of the original datagram in ICMP error messages.

11.10 In Figure 11.30, what happens to a packet that has the high-order bit of `ip_off` set?

11.11 Why is the return value from `ip_output` discarded in Figure 11.39?

12

IP Multicasting

12.1 Introduction

Recall from Chapter 8 that class D IP addresses (224.0.0.0 to 239.255.255.255) do not identify individual interfaces in an internet but instead identify groups of interfaces. For this reason, class D addresses are called *multicast groups*. A datagram with a class D destination address is delivered to every interface in an internet that has *joined* the corresponding multicast group.

Experimental applications on the Internet that take advantage of multicasting include audio and video conferencing applications, resource discovery tools, and shared whiteboards.

Group membership is determined dynamically as interfaces join and leave groups based on requests from processes running on each system. Since group membership is relative to an interface, it is possible for a multihomed host have different group membership lists for each interface. We'll refer to group membership on a particular interface as an {interface, group} pair.

Group membership on a single network is communicated between systems by the IGMP protocol (Chapter 13). Multicast routers propagate group membership information using multicast routing protocols (Chapter 14), such as DVMRP (Distance Vector Multicast Routing Protocol). A standard IP router may support multicast routing, or multicast routing may be handled by a router dedicated to that purpose.

Networks such as Ethernet, token ring, and FDDI directly support hardware multicasting. In Net/3, if an interface supports multicasting, the `IFF_MULTICAST` bit is on in `if_flags` in the interface's `ifnet` structure (Figure 3.7). We'll use Ethernet to illustrate hardware-supported IP multicasting, since Ethernet is in widespread use and Net/3 includes sample Ethernet drivers. Multicast services are trivially implemented on point-to-point networks such as SLIP and the loopback interface.

IP multicasting services may not be available on a particular interface if the local network does not support hardware-level multicast. RFC 1122 does not prevent the interface layer from providing a software-level multicast service as long as it is transparent to IP.

RFC 1112 [Deering 1989] describes the host requirements for IP multicasting. There are three levels of conformance:

Level 0 The host cannot send or receive IP multicasts.

Such a host should silently discard any packets it receives with a class D destination address.

Level 1 The host can send but cannot receive IP multicasts.

A host is not required to join an IP multicast group before sending a datagram to the group. A multicast datagram is sent in the same way as a unicast datagram except the destination address is the IP multicast group. The network drivers must recognize this and multicast the datagram on the local network.

Level 2 The host can send and receive IP multicasts.

To receive IP multicasts, the host must be able to join and leave multicast groups and must support IGMP for exchanging group membership information on at least one interface. A multihomed host may support multicasting on a subset of its interfaces.

Net/3 meets the level 2 host requirements and can additionally act as a multicast router. As with unicast IP routing, we assume that the system we are describing is a multicast router and we include the Net/3 multicast routing code in our presentation.

Well-Known IP Multicast Groups

As with UDP and TCP port numbers, the *Internet Assigned Numbers Authority* (IANA) maintains a list of registered IP multicast groups. The current list can be found in RFC 1700. For more information about the IANA, see RFC 1700. Figure 12.1 shows only some of the well-known groups.

Group	Description	Net/3 constant
224.0.0.0	reserved	*INADDR_UNSPEC_GROUP*
224.0.0.1	all systems on this subnet	*INADDR_ALLHOSTS_GROUP*
224.0.0.2	all routers on this subnet	
224.0.0.3	unassigned	
224.0.0.4	DVMRP routers	
224.0.0.255	unassigned	*INADDR_MAX_LOCAL_GROUP*
224.0.1.1	NTP Network Time Protocol	
224.0.1.2	SGI-Dogfight	

Figure 12.1 Some registered IP multicast groups.

The first 256 groups (224.0.0.0 to 224.0.0.255) are reserved for protocols that implement IP unicast and multicast routing mechanisms. Datagrams sent to any of these groups are not forwarded beyond the local network by multicast routers, regardless of the TTL value in the IP header.

RFC 1075 places this requirement only on the 224.0.0.0 and 224.0.0.1 groups but `mrouted`, the most common multicast routing implementation, restricts the remaining groups as described here. Group 224.0.0.0 (`INADDR_UNSPEC_GROUP`) is reserved and group 224.0.0.255 (`INADDR_MAX_LOCAL_GROUP`) marks the last local multicast group.

Every level-2 conforming system is required to join the 224.0.0.1 (`INADDR_ALLHOSTS_GROUP`) group on all multicast interfaces at system initialization time (Figure 6.17) and remain a member of the group until the system is shut down. There is no multicast group that corresponds to every interface on an internet.

Imagine if your voice-mail system had the option of sending a message to every voice mailbox in your company. Maybe you have such an option. Do you find it useful? Does it scale to larger companies? Can anyone send to the "all-mailbox" group, or is it restricted?

Unicast and multicast routers may join group 224.0.0.2 to communicate with each other. The ICMP router solicitation message and router advertisement messages may be sent to 224.0.0.2 (the all-routers group) and 224.0.0.1 (the all-hosts group), respectively, instead of to the limited broadcast address (255.255.255.255).

The 224.0.0.4 group supports communication between multicast routers that implement DVMRP. Other groups within the local multicast group range are similarly assigned for other routing protocols.

Beyond the first 256 groups, the remaining groups (224.0.1.0–239.255.255.255) are assigned to various multicast application protocols or remain unassigned. Figure 12.1 lists two examples, the Network Time Protocol (224.0.1.1), and SGI-Dogfight (224.0.1.2).

Throughout this chapter, we note that multicast packets are sent and received by the transport layer on a host. While the multicasting code is not aware of the specific transport protocol that sends and receives multicast datagrams, the only Internet transport protocol that supports multicasting is UDP.

12.2 Code Introduction

The basic multicasting code discussed in this chapter is contained within the same files as the standard IP code. Figure 12.2 lists the files that we examine.

File	Description
netinet/if_ether.h	Ethernet multicasting structure and macro definitions
netinet/in.h	more Internet multicast structures
netinet/in_var.h	Internet multicast structure and macro definitions
netinet/ip_var.h	IP multicast structures
net/if_ethersubr.c	Ethernet multicast functions
netinet/in.c	group membership functions
netinet/ip_input.c	input multicast processing
netinet/ip_output.c	output multicast processing

Figure 12.2 Files discussed in this chapter.

Global Variables

Three new global variables are introduced in this chapter:

Variable	Datatype	Description
ether_ipmulticast_min	u_char []	minimum Ethernet multicast address reserved for IP
ether_ipmulticast_max	u_char []	maximum Ethernet multicast address reserved for IP
ip_mrouter	struct socket *	pointer to socket created by multicast routing daemon

Figure 12.3 Global variables introduced in this chapter.

Statistics

The code in this chapter updates a few of the counters maintained in the global `ipstat` structure.

`ipstat` member	Description
ips_forward	#packets forwarded by this system
ips_cantforward	#packets that cannot be forwarded—system is not a router
ips_noroute	#packets that cannot be forwarded because a route is not available

Figure 12.4 Multicast processing statistics.

Link-level multicast statistics are collected in the `ifnet` structure (Figure 4.5) and may include multicasting of protocols other than IP.

12.3 Ethernet Multicast Addresses

An efficient implementation of IP multicasting requires IP to take advantage of hardware-level multicasting, without which each IP datagram would have to be broadcast to the network and every host would have to examine each datagram and discard those not intended for the host. The hardware filters unwanted datagrams before they reach the IP layer.

For the hardware filter to work, the network interface must convert the IP multicast group destination to a link-layer multicast address recognized by the network hardware. On point-to-point networks, such as SLIP and the loopback interface, the mapping is implicit since there is only one possible destination. On other networks, such as Ethernet, an explicit mapping function is required. The standard mapping for Ethernet applies to any network that employs 802.3 addressing.

Figure 4.12 illustrated the difference between a Ethernet unicast and multicast address: if the low-order bit of the high-order byte of the Ethernet address is a 1, it is a multicast address; otherwise it is a unicast address. Unicast Ethernet addresses are assigned by the interface's manufacturer, but multicast addresses are assigned dynamically by network protocols.

IP to Ethernet Multicast Address Mapping

Because Ethernet supports multiple protocols, a method to allocate the multicast addresses and prevent conflicts is needed. Ethernet addresses allocation is administered by the IEEE. A block of Ethernet multicast addresses is assigned to the IANA by the IEEE to support IP multicasting. The addresses in the block all start with `01:00:5e`.

The block of Ethernet unicast addresses starting with `00:00:5e` is also assigned to the IANA but remains reserved for future use.

Figure 12.5 illustrates the construction of an Ethernet multicast address from a class D IP address.

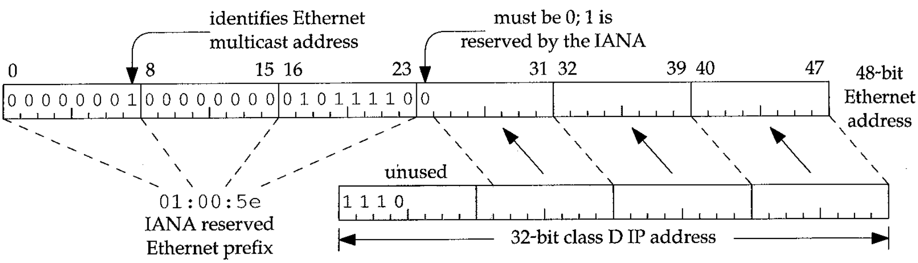

Figure 12.5 Mapping between IP and Ethernet addresses.

The mapping illustrated by Figure 12.5 is a many-to-one mapping. The high-order 9 bits of the class D IP address are not used when constructing the Ethernet address. 32 IP multicast groups map to a single Ethernet multicast address (Exercise 12.3). In

Section 12.14 we'll see how this affects input processing. Figure 12.6 shows the macro that implements this mapping in Net/3.

if_ether.h
```
61 #define ETHER_MAP_IP_MULTICAST(ipaddr, enaddr) \
62     /* struct in_addr *ipaddr; */ \
63     /* u_char enaddr[6];       */ \
64 { \
65     (enaddr)[0] = 0x01; \
66     (enaddr)[1] = 0x00; \
67     (enaddr)[2] = 0x5e; \
68     (enaddr)[3] = ((u_char *)ipaddr)[1] & 0x7f; \
69     (enaddr)[4] = ((u_char *)ipaddr)[2]; \
70     (enaddr)[5] = ((u_char *)ipaddr)[3]; \
71 }
```
if_ether.h

Figure 12.6 ETHER_MAP_IP_MULTICAST macro.

IP to Ethernet multicast mapping

61–71 ETHER_MAP_IP_MULTICAST implements the mapping shown in Figure 12.5. ipaddr points to the class D multicast address, and the matching Ethernet address is constructed in enaddr, an array of 6 bytes. The first 3 bytes of the Ethernet multicast address are 0x01, 0x00, and 0x5e followed by a 0 bit and then the low-order 23 bits of the class D IP address.

12.4 ether_multi Structure

For each Ethernet interface, Net/3 maintains a list of Ethernet multicast address ranges to be received by the hardware. This list defines the multicast filtering to be implemented by the device. Because most Ethernet devices are limited in the number of addresses they can selectively receive, the IP layer must be prepared to discard datagrams that pass through the hardware filter. Each address range is stored in an ether_multi structure:

if_ether.h
```
147 struct ether_multi {
148     u_char  enm_addrlo[6];        /* low  or only address of range */
149     u_char  enm_addrhi[6];        /* high or only address of range */
150     struct arpcom *enm_ac;        /* back pointer to arpcom */
151     u_int   enm_refcount;         /* no. claims to this addr/range */
152     struct ether_multi *enm_next;    /* ptr to next ether_multi */
153 };
```
if_ether.h

Figure 12.7 ether_multi structure.

Ethernet multicast addresses

147–153 enm_addrlo and enm_addrhi specify a range of Ethernet multicast addresses that should be received. A single Ethernet address is specified when enm_addrlo and enm_addrhi are the same. The entire list of ether_multi structures is attached to the arpcom structure of each Ethernet interface (Figure 3.26). Ethernet multicasting is independent of ARP—using the arpcom structure is a matter of convenience, since the structure is already included in every Ethernet interface structure.

We'll see that the start and end of the ranges are always the same since there is no way in Net/3 for a process to specify an address range.

enm_ac points back to the arpcom structure of the associated interface and enm_refcount tracks the usage of the ether_multi structure. When the reference count drops to 0, the structure is released. enm_next joins the ether_multi structures for a single interface into a linked list. Figure 12.8 shows a list of three ether_multi structures attached to le_softc[0], the ifnet structure for our sample Ethernet interface.

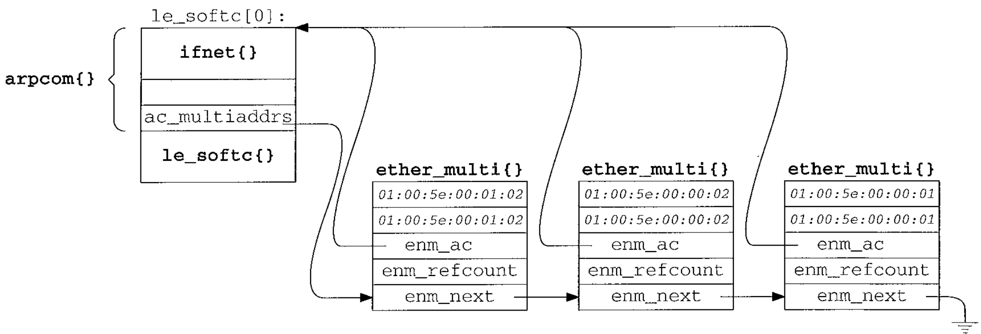

Figure 12.8 The LANCE interface with three ether_multi structures.

In Figure 12.8 we see that:

- The interface has joined three groups. Most likely they are: 224.0.0.1 (all-hosts), 224.0.0.2 (all-routers), and 224.0.1.2 (SGI-dogfight). Because the Ethernet to IP mapping is a one-to-many mapping, we cannot determine the exact IP multicast groups by examining the resulting Ethernet multicast addresses. The interface may have joined 225.0.0.1, 225.0.0.2, and 226.0.1.2, for example.
- The most recently joined group appears at the front of the list.
- The enm_ac back-pointer makes it easy to find the beginning of the list and to release an ether_multi structure, without having to implement a doubly linked list.
- The ether_multi structures apply to Ethernet devices only. Other multicast devices may have a different multicast implementation.

The ETHER_LOOKUP_MULTI macro, shown in Figure 12.9, searches an ether_multi list for a range of addresses.

if_ether.h
```
166 #define ETHER_LOOKUP_MULTI(addrlo, addrhi, ac, enm) \
167     /* u_char addrlo[6]; */ \
168     /* u_char addrhi[6]; */ \
169     /* struct arpcom *ac; */ \
170     /* struct ether_multi *enm; */ \
171 { \
172     for ((enm) = (ac)->ac_multiaddrs; \
173         (enm) != NULL && \
174         (bcmp((enm)->enm_addrlo, (addrlo), 6) != 0 || \
175          bcmp((enm)->enm_addrhi, (addrhi), 6) != 0); \
176         (enm) = (enm)->enm_next); \
177 }
```
if_ether.h

Figure 12.9 ETHER_LOOKUP_MULTI macro.

Ethernet multicast lookups

166–177 addrlo and addrhi specify the search range and ac points to the arpcom structure containing the list to search. The for loop performs a linear search, stopping at the end of the list or when enm_addrlo and enm_addrhi both match the supplied addrlo and addrhi addresses. When the loop terminates, enm is null or points to a matching ether_multi structure.

12.5 Ethernet Multicast Reception

After this section, this chapter discusses only IP multicasting, but it is possible in Net/3 to configure the system to receive any Ethernet multicast packet. Although not useful with the IP protocols, other protocol families within the kernel might be prepared to receive these multicasts. Explicit multicast configuration is done by issuing the ioctl commands shown in Figure 12.10.

Command	Argument	Function	Description
SIOCADDMULTI	struct ifreq *	ifioctl	add multicast address to reception list
SIOCDELMULTI	struct ifreq *	ifioctl	delete multicast address from reception list

Figure 12.10 Multicast ioctl commands.

These two commands are passed by ifioctl (Figure 12.11) directly to the device driver for the interface specified in the ifreq structure (Figure 6.12).

440–446 If the process does not have superuser privileges, or if the interface does not have an if_ioctl function, ifioctl returns an error; otherwise the request is passed directly to the device driver.

if.c

```
    case SIOCADDMULTI:
    case SIOCDELMULTI:
        if (error = suser(p->p_ucred, &p->p_acflag))
            return (error);
        if (ifp->if_ioctl == NULL)
            return (EOPNOTSUPP);
        return ((*ifp->if_ioctl) (ifp, cmd, data));
```

if.c

Figure 12.11 ifioctl function: multicast commands.

12.6 in_multi Structure

The Ethernet multicast data structures described in Section 12.4 are not specific to IP; they must support multicast activity by any of the protocol families supported by the kernel. At the network level, IP maintains a list of IP multicast groups associated with each interface.

As a matter of implementation convenience, the IP multicast list is attached to the in_ifaddr structure associated with the interface. Recall from Section 6.5 that this structure contains the unicast address for the interface. There is no relationship between the unicast address and the attached multicast group list other than that they both are associated with the same interface.

This is an artifact of the Net/3 implementation. It is possible for an implementation to support IP multicast groups on an interface that does not accept IP unicast packets.

Each IP multicast {interface, group} pair is described by an in_multi structure shown in Figure 12.12.

in_var.h

```
111 struct in_multi {
112     struct in_addr inm_addr;        /* IP multicast address */
113     struct ifnet *inm_ifp;          /* back pointer to ifnet */
114     struct in_ifaddr *inm_ia;       /* back pointer to in_ifaddr */
115     u_int   inm_refcount;           /* no. membership claims by sockets */
116     u_int   inm_timer;              /* IGMP membership report timer */
117     struct in_multi *inm_next;      /* ptr to next multicast address */
118 };
```

in_var.h

Figure 12.12 in_multi structure.

IP multicast addresses

111–118 inm_addr is a class D multicast address (e.g., 224.0.0.1, the all-hosts group). inm_ifp points back to the ifnet structure of the associated interface and inm_ia points back to the interface's in_ifaddr structure.

An in_multi structure exists only if at least one process on the system has notified the kernel that it wants to receive multicast datagrams for a particular {interface, group} pair. Since multiple processes may elect to receive datagrams sent to a particular pair, inm_refcount keeps track of the number of references to the pair. When no more processes are interested in the pair, inm_refcount drops to 0 and the structure is released. This action may cause an associated ether_multi structure to be released if its reference count also drops to 0.

inm_timer is part of the IGMP protocol implementation described in Chapter 13. Finally, inm_next points to the next in_multi structure in the list.

Figure 12.13 illustrates the relationship between an interface, its IP unicast address, and its IP multicast group list using the le_softc[0] sample interface.

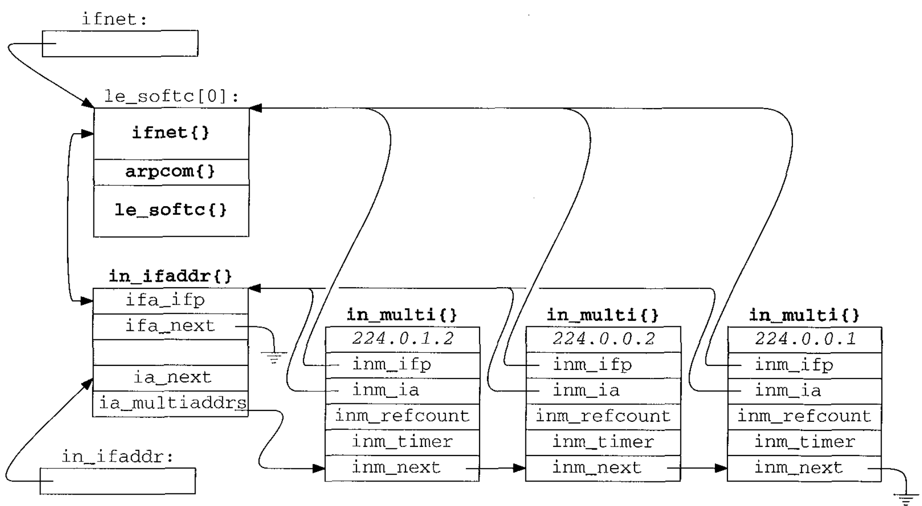

Figure 12.13 An IP multicast group list for the le interface.

We've omitted the corresponding ether_multi structures for clarity (but see Figure 12.34). If the system had two Ethernet cards, the second card would be managed through le_softc[1] and would have its own multicast group list attached to its arpcom structure. The macro IN_LOOKUP_MULTI (Figure 12.14) searches the IP multicast list for a particular multicast group.

IP multicast lookups

131–146 IN_LOOKUP_MULTI looks for the multicast group addr in the multicast group list associated with interface ifp. IFP_TO_IA searches the Internet address list, in_ifaddr, for the in_ifaddr structure associated with the interface identified by ifp. If IFP_TO_IA finds an interface, the for loop searches its IP multicast list. After the loop, inm is null or points to the matching in_multi structure.

in_var.h

```
#define IN_LOOKUP_MULTI(addr, ifp, inm) \
    /* struct in_addr addr; */ \
    /* struct ifnet *ifp; */ \
    /* struct in_multi *inm; */ \
{ \
     struct in_ifaddr *ia; \
\
    IFP_TO_IA((ifp), ia); \
    if (ia == NULL) \
        (inm) = NULL; \
    else \
        for ((inm) = ia->ia_multiaddrs; \
            (inm) != NULL && (inm)->inm_addr.s_addr != (addr).s_addr; \
             (inm) = inm->inm_next) \
             continue; \
}
```

in_var.h

Figure 12.14 IN_LOOKUP_MULTI macro.

12.7 ip_moptions Structure

The ip_moptions structure contains the multicast options through which the transport layer controls multicast output processing. For example, the UDP call to ip_output is:

```
error = ip_output(m, inp->inp_options, &inp->inp_route,
                  inp->inp_socket->so_options & (SO_DONTROUTE|SO_BROADCAST),
                  inp->inp_moptions);
```

In Chapter 22 we'll see that inp points to an Internet protocol control block (PCB) and that UDP associates a PCB with each socket created by a process. Within the PCB, inp_moptions is a pointer to an ip_moptions structure. From this we see that a different ip_moptions structure may be passed to ip_output for each outgoing datagram. Figure 12.15 shows the definition of the ip_moptions structure.

ip_var.h

```
100 struct ip_moptions {
101     struct  ifnet *imo_multicast_ifp; /* ifp for outgoing multicasts */
102     u_char  imo_multicast_ttl;        /* TTL for outgoing multicasts */
103     u_char  imo_multicast_loop;       /* 1 => hear sends if a member */
104     u_short imo_num_memberships;      /* no. memberships this socket */
105     struct  in_multi *imo_membership[IP_MAX_MEMBERSHIPS];
106 };
```

ip_var.h

Figure 12.15 ip_moptions structure.

Multicast options

100–106 ip_output routes outgoing multicast datagrams through the interface pointed to by imo_multicast_ifp or, if imo_multicast_ifp is null, through the default interface for the destination multicast group (Chapter 14).

imo_multicast_ttl specifies the initial IP TTL value for outgoing multicasts. The default is 1, which causes multicast datagrams to remain on the local network.

If imo_multicast_loop is 0, the multicast datagram is not looped back and delivered to the transmitting interface even if the interface is a member of the multicast group. If imo_multicast_loop is 1, the multicast datagram is looped back to the transmitting interface if the interface is a member of the multicast group.

Finally, the integer imo_num_memberships and the array imo_membership maintain the list of {interface, group} pairs associated with the structure. Changes to the list are communicated to IP, which announces membership changes on the locally attached network. Each entry in the imo_membership array is a pointer to an in_multi structure attached to the in_ifaddr structure of the appropriate interface.

12.8 Multicast Socket Options

Several IP-level socket options, shown in Figure 12.10, provide process-level access to ip_moptions structures.

Command	Argument	Function	Description
IP_MULTICAST_IF	struct in_addr	ip_ctloutput	select default interface for outgoing multicasts
IP_MULTICAST_TTL	u_char	ip_ctloutput	select default TTL for outgoing multicasts
IP_MULTICAST_LOOP	u_char	ip_ctloutput	enable or disable loopback of outgoing multicasts
IP_ADD_MEMBERSHIP	struct ip_mreq	ip_ctloutput	join a multicast group
IP_DROP_MEMBERSHIP	struct ip_mreq	ip_ctloutput	leave a multicast group

Figure 12.16 Multicast socket options.

In Figure 8.31 we looked at the overall structure of the ip_ctloutput function. Figure 12.17 shows the cases relevant to changing and retrieving multicast options.

486–491 All the multicast options are handled through the ip_setmoptions and
539–549 ip_getmoptions functions. The ip_moptions structure passed by reference to ip_getmoptions or to ip_setmoptions is the one associated with the socket on which the ioctl command was issued.

The error code returned when an option is not recognized is different for the get and set cases. ENOPROTOOPT is the more reasonable choice.

12.9 Multicast TTL Values

Multicast TTL values are difficult to understand because they have two purposes. The primary purpose of the TTL value, as with all IP packets, is to limit the lifetime of the packet within an internet and prevent it from circulating indefinitely. The second purpose is to contain packets within a region of the internet specified by administrative

ip_output.c

```
        case PRCO_SETOPT:
            switch (optname) {

                        /* other set cases */

            case IP_MULTICAST_IF:
            case IP_MULTICAST_TTL:
            case IP_MULTICAST_LOOP:
            case IP_ADD_MEMBERSHIP:
            case IP_DROP_MEMBERSHIP:
                error = ip_setmoptions(optname, &inp->inp_moptions, m);
                break;
              freeit:
            default:
                error = EINVAL;
                break;
            }
            if (m)
                (void) m_free(m);
            break;

        case PRCO_GETOPT:
            switch (optname) {

                        /* other get cases */

            case IP_MULTICAST_IF:
            case IP_MULTICAST_TTL:
            case IP_MULTICAST_LOOP:
            case IP_ADD_MEMBERSHIP:
            case IP_DROP_MEMBERSHIP:
                error = ip_getmoptions(optname, inp->inp_moptions, mp);
                break;

            default:
                error = ENOPROTOOPT;
                break;
            }
```

ip_output.c

Figure 12.17 `ip_ctloutput` function: multicast options.

boundaries. This administrative region is specified in subjective terms such as "this site," "this company," or "this state," and is relative to the starting point of the packet. The region associated with a multicast packet is called its *scope*.

The standard implementation of RFC 1112 multicasting merges the two concepts of lifetime and scope into the single TTL value in the IP header. In addition to discarding packets when the IP TTL drops to 0, *multicast* routers associate with each interface a TTL threshold that limits multicast transmission on that interface. A packet must have a

TTL greater than or equal to the interface's threshold value for it to be transmitted on the interface. Because of this, a multicast packet may be dropped even before its TTL value reaches 0.

Threshold values are assigned by an administrator when configuring a multicast router. These values define the scope of multicast packets. The significance of an initial TTL value for multicast datagrams is defined by the threshold policy used by the administrator and the distance between the source of the datagram and the multicast interfaces.

Figure 12.18 shows the recommended TTL values for various applications as well as recommended threshold values.

ip_ttl	Application	Scope
0		same interface
1		same subnet
31	local event video	
32		same site
63	local event audio	
64		same region
95	IETF channel 2 video	
127	IETF channel 1 video	
128		same continent
159	IETF channel 2 audio	
191	IETF channel 1 audio	
223	IETF channel 2 low-rate audio	
255	IETF channel 1 low-rate audio unrestricted in scope	

Figure 12.18 TTL values for IP multicast datagrams.

The first column lists the starting value of `ip_ttl` in the IP header. The second column illustrates an application specific use of threshold values ([Casner 1993]). The third column lists the recommended scopes to associate with the TTL values.

For example, an interface that communicates to a network outside the local site would be configured with a multicast threshold of 32. The TTL field of any datagram that starts with a TTL of 32 (or less) is less than 32 when it reaches this interface (there is at least one hop between the source and the router) and is discarded before the router forwards it to the external network—even if the TTL is still greater than 0.

A multicast datagram that start with a TTL of 128 would pass through site interfaces with a threshold of 32 (as long as it reached the interface within $128 - 32 = 96$ hops) but would be discarded by intercontinental interfaces with a threshold of 128.

The MBONE

A subset of routers on the Internet supports IP multicast routing. This multicast backbone is called the *MBONE*, which is described in [Casner 1993]. It exists to support experimentation with IP multicasting—in particular with audio and video data streams. In the MBONE, threshold values limit how far various data streams propagate. In Figure 12.18, we see that local event video packets always start with a TTL of

31. An interface with a threshold of 32 always blocks local event video. At the other end of the scale, IETF channel 1 low-rate audio is restricted only by the inherent IP TTL maximum of 255 hops. It propagates through the entire MBONE. An administrator of a multicast router within the MBONE can select a threshold value to accept or discard MBONE data streams selectively.

Expanding-Ring Search

Another use of the multicast TTL is to probe the internet for a resource by varying the initial TTL value of the probe datagram. This technique is called an *expanding-ring search* ([Boggs 1982]). A datagram with an initial TTL of 0 reaches only a resource on the local system associated with the outgoing interface. A TTL of 1 reaches the resource if it exists on the local subnet. A TTL of 2 reaches resources within two hops of the source. An application increases the TTL exponentially to probe a large internet quickly.

RFC 1546 [Partridge, Mendez, and Milliken 1993] describes a related service called *anycasting*. As proposed, anycasting relies on a distinguished set of IP addresses to represent groups of hosts much like multicasting. Unlike multicast addresses, the network is expected to propagate an anycast packet until it is received by at least one host. This simplifies the implementation of an application, which no longer needs to perform expanding-ring searches.

12.10 ip_setmoptions Function

The bulk of the ip_setmoptions function consists of a switch statement to handle each option. Figure 12.19 shows the beginning and end of ip_setmoptions. The body of the switch is discussed in the following sections.

650–664 The first argument, optname, indicates which multicast option is being changed. The second argument, imop, references a pointer to an ip_moptions structure. If *imop is nonnull, ip_setmoptions modifies the structure it points to. Otherwise, ip_setmoptions allocates a new ip_moptions structure and saves its address in *imop. If no memory is available, ip_setmoptions returns ENOBUFS immediately. Any subsequent errors that occur are posted in error, which is returned to the caller at the end of the function. The third argument, m, points to an mbuf that contains the data for the option to be changed (second column of Figure 12.16).

Construct the defaults

665–679 When a new ip_moptions structure is allocated, ip_setmoptions initializes the default multicast interface pointer to null, initializes the default TTL to 1 (IP_DEFAULT_MULTICAST_TTL), enables the loopback of multicast datagrams, and clears the group membership list. With these defaults, ip_output selects an outgoing interface by consulting the routing tables, multicasts are kept on the local network, and the system receives its own multicast transmissions if the outgoing interface is a member of the destination group.

Process options

680–860 The body of ip_setmoptions consists of a switch statement with a case for each option. The default case (for unknown options) sets error to EOPNOTSUPP.

ip_output.c

```
int
ip_setmoptions(optname, imop, m)
int     optname;
struct ip_moptions **imop;
struct mbuf *m;
{
    int     error = 0;
    u_char  loop;
    int     i;
    struct in_addr addr;
    struct ip_mreq *mreq;
    struct ifnet *ifp;
    struct ip_moptions *imo = *imop;
    struct route ro;
    struct sockaddr_in *dst;

    if (imo == NULL) {
        /*
         * No multicast option buffer attached to the pcb;
         * allocate one and initialize to default values.
         */
        imo = (struct ip_moptions *) malloc(sizeof(*imo), M_IPMOPTS,
                                            M_WAITOK);

        if (imo == NULL)
            return (ENOBUFS);
        *imop = imo;
        imo->imo_multicast_ifp = NULL;
        imo->imo_multicast_ttl = IP_DEFAULT_MULTICAST_TTL;
        imo->imo_multicast_loop = IP_DEFAULT_MULTICAST_LOOP;
        imo->imo_num_memberships = 0;
    }
    switch (optname) {

                        /* switch cases */

    default:
        error = EOPNOTSUPP;
        break;
    }

    /*
     * If all options have default values, no need to keep the structure.
     */
    if (imo->imo_multicast_ifp == NULL &&
        imo->imo_multicast_ttl == IP_DEFAULT_MULTICAST_TTL &&
        imo->imo_multicast_loop == IP_DEFAULT_MULTICAST_LOOP &&
        imo->imo_num_memberships == 0) {
        free(*imop, M_IPMOPTS);
        *imop = NULL;
    }
    return (error);
}
```

ip_output.c

Figure 12.19 `ip_setmoptions` function.

Discard structure if defaults are OK

861–872 After the switch statement, ip_setmoptions examines the ip_moptions structure. If all the multicast options match their respective default values, the structure is unnecessary and is released. ip_setmoptions returns 0 or the posted error code.

Selecting an Explicit Multicast Interface: IP_MULTICAST_IF

When optname is IP_MULTICAST_IF, the mbuf passed to ip_setmoptions contains the unicast address of a multicast interface, which specifies the particular interface for multicasts sent on this socket. Figure 12.20 shows the code for this option.

ip_output.c

```
681     case IP_MULTICAST_IF:
682         /*
683          * Select the interface for outgoing multicast packets.
684          */
685         if (m == NULL || m->m_len != sizeof(struct in_addr)) {
686             error = EINVAL;
687             break;
688         }
689         addr = *(mtod(m, struct in_addr *));
690         /*
691          * INADDR_ANY is used to remove a previous selection.
692          * When no interface is selected, a default one is
693          * chosen every time a multicast packet is sent.
694          */
695         if (addr.s_addr == INADDR_ANY) {
696             imo->imo_multicast_ifp = NULL;
697             break;
698         }
699         /*
700          * The selected interface is identified by its local
701          * IP address.  Find the interface and confirm that
702          * it supports multicasting.
703          */
704         INADDR_TO_IFP(addr, ifp);
705         if (ifp == NULL || (ifp->if_flags & IFF_MULTICAST) == 0) {
706             error = EADDRNOTAVAIL;
707             break;
708         }
709         imo->imo_multicast_ifp = ifp;
710         break;
```

ip_output.c

Figure 12.20 ip_setmoptions function: selecting a multicast output interface.

Validation

681–698 If no mbuf has been provided or the data within the mbuf is not the size of an in_addr structure, ip_setmoptions posts an EINVAL error; otherwise the data is copied into addr. If the interface address is INADDR_ANY, any previously selected interface is discarded. Subsequent multicasts with this ip_moptions structure are routed according to their destination group instead of through an explicitly named interface (Figure 12.40).

Select the default interface

699–710 If addr contains an address, INADDR_TO_IFP locates the matching interface. If a match can't be found or the interface does not support multicasting, EADDRNOTAVAIL is posted. Otherwise, ifp, the matching interface, becomes the multicast interface for output requests associated with this ip_moptions structure.

Selecting an Explicit Multicast TTL: IP_MULTICAST_TTL

When optname is IP_MULTICAST_TTL, the mbuf is expected to contain a single byte specifying the IP TTL for outgoing multicasts. This TTL is inserted by ip_output into every multicast datagram sent on the associated socket. Figure 12.21 shows the code for this option.

ip_output.c

```
711     case IP_MULTICAST_TTL:
712         /*
713          * Set the IP time-to-live for outgoing multicast packets.
714          */
715         if (m == NULL || m->m_len != 1) {
716             error = EINVAL;
717             break;
718         }
719         imo->imo_multicast_ttl = *(mtod(m, u_char *));
720         break;
```

ip_output.c

Figure 12.21 ip_setmoptions function: selecting an explicit multicast TTL.

Validate and select the default TTL

711–720 If the mbuf contains a single byte of data, it is copied into imo_multicast_ttl. Otherwise, EINVAL is posted.

Selecting Multicast Loopbacks: IP_MULTICAST_LOOP

In general, multicast applications come in two forms:

- An application with one sender per system and multiple remote receivers. In this configuration only one local process is sending datagrams to the group so there is no need to loopback outgoing multicasts. Examples include a multicast routing daemon and conferencing systems.
- An application with multiple senders and receivers on a system. Datagrams must be looped back so that each process receives the transmissions of the other senders on the system.

The IP_MULTICAST_LOOP option (Figure 12.22) selects the loopback policy associated with an ip_moptions structure.

ip_output.c
```
721       case IP_MULTICAST_LOOP:
722           /*
723            * Set the loopback flag for outgoing multicast packets.
724            * Must be zero or one.
725            */
726           if (m == NULL || m->m_len != 1 ||
727               (loop = *(mtod(m, u_char *))) > 1) {
728               error = EINVAL;
729               break;
730           }
731           imo->imo_multicast_loop = loop;
732           break;
```
ip_output.c

Figure 12.22 `ip_setmoptions` function: selecting multicast loopbacks.

Validate and select the loopback policy

721–732 If `m` is null, does not contain 1 byte of data, or the byte is not 0 or 1, `EINVAL` is posted. Otherwise, the byte is copied into `imo_multicast_loop`. A 0 indicates that datagrams should not be looped back, and a 1 enables the loopback mechanism.

Figure 12.23 shows the relationship between, the maximum scope of a multicast datagram, `imo_multicast_ttl`, and `imo_multicast_loop`.

`imo_multicast-`		Recipients			
`_loop`	`_ttl`	Outgoing Interface?	Local Network?	Remote Networks?	Other Interfaces?
1	0	•			
1	1	•	•		
1	>1	•	•	•	see text

Figure 12.23 Loopback and TTL effects on multicast scope.

Figure 12.23 shows that the set of interfaces that may receive a multicast packet depends on what the loopback policy is for the transmission and what TTL value is specified in the packet. A packet may be received on an interface if the hardware receives its own transmissions, regardless of the loopback policy. A datagram may be routed through the network and arrive on another interface attached to the system (Exercise 12.6). If the sending system is itself a multicast router, outgoing packets may be forwarded to the other interfaces, but they will only be accepted for input processing on one interface (Chapter 14).

12.11 Joining an IP Multicast Group

Other than the IP all-hosts group, which the kernel automatically joins (Figure 6.17), membership in a group is driven by explicit requests from processes on the system. The process of joining (or leaving) a multicast group is more involved than the other multicast options. The `in_multi` list for an interface must be modified as well as any link-layer multicast structures such as the `ether_multi` list we described for Ethernet.

The data passed in the mbuf when `optname` is `IP_ADD_MEMBERSHIP` is an `ip_mreq` structure shown in Figure 12.24.

in.h

```
148 struct ip_mreq {
149     struct in_addr imr_multiaddr;    /* IP multicast address of group */
150     struct in_addr imr_interface;    /* local IP address of interface */
151 };
```

in.h

Figure 12.24 `ip_mreq` structure.

148–151 `imr_multiaddr` specifies the multicast group and `imr_interface` identifies the interface by its associated unicast IP address. The `ip_mreq` structure specifies the {interface, group} pair for membership changes.

Figure 12.25 illustrates the functions involved with joining and leaving a multicast group associated with our example Ethernet interface.

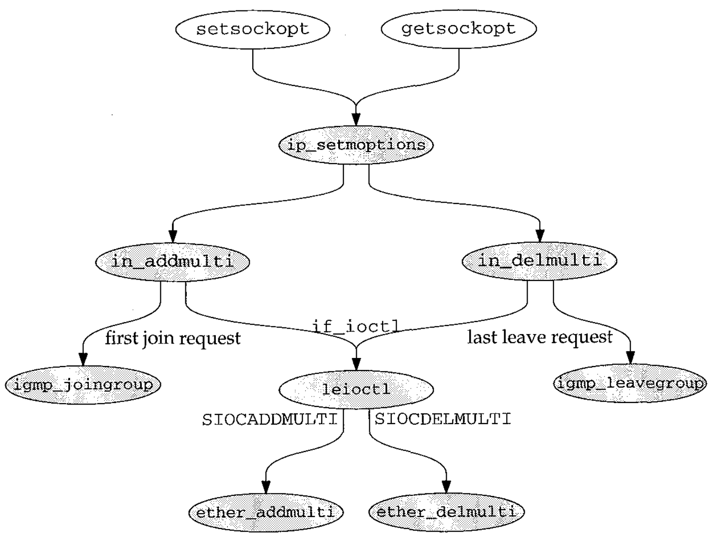

Figure 12.25 Joining and leaving a multicast group.

We start by describing the changes to the `ip_moptions` structure in the `IP_ADD_MEMBERSHIP` case in `ip_setmoptions` (Figure 12.26). Then we follow the request down through the IP layer, the Ethernet driver, and to the physical device—in our case, the LANCE Ethernet card.

ip_output.c

```
    case IP_ADD_MEMBERSHIP:
        /*
         * Add a multicast group membership.
         * Group must be a valid IP multicast address.
         */
        if (m == NULL || m->m_len != sizeof(struct ip_mreq)) {
            error = EINVAL;
            break;
        }
        mreq = mtod(m, struct ip_mreq *);
        if (!IN_MULTICAST(ntohl(mreq->imr_multiaddr.s_addr))) {
            error = EINVAL;
            break;
        }
        /*
         * If no interface address was provided, use the interface of
         * the route to the given multicast address.
         */
        if (mreq->imr_interface.s_addr == INADDR_ANY) {
            ro.ro_rt = NULL;
            dst = (struct sockaddr_in *) &ro.ro_dst;
            dst->sin_len = sizeof(*dst);
            dst->sin_family = AF_INET;
            dst->sin_addr = mreq->imr_multiaddr;
            rtalloc(&ro);
            if (ro.ro_rt == NULL) {
                error = EADDRNOTAVAIL;
                break;
            }
            ifp = ro.ro_rt->rt_ifp;
            rtfree(ro.ro_rt);
        } else {
            INADDR_TO_IFP(mreq->imr_interface, ifp);
        }
        /*
         * See if we found an interface, and confirm that it
         * supports multicast.
         */
        if (ifp == NULL || (ifp->if_flags & IFF_MULTICAST) == 0) {
            error = EADDRNOTAVAIL;
            break;
        }
```

```
775         /*
776          * See if the membership already exists or if all the
777          * membership slots are full.
778          */
779         for (i = 0; i < imo->imo_num_memberships; ++i) {
780             if (imo->imo_membership[i]->inm_ifp == ifp &&
781                 imo->imo_membership[i]->inm_addr.s_addr
782                 == mreq->imr_multiaddr.s_addr)
783                 break;
784         }
785         if (i < imo->imo_num_memberships) {
786             error = EADDRINUSE;
787             break;
788         }
789         if (i == IP_MAX_MEMBERSHIPS) {
790             error = ETOOMANYREFS;
791             break;
792         }
793         /*
794          * Everything looks good; add a new record to the multicast
795          * address list for the given interface.
796          */
797         if ((imo->imo_membership[i] =
798              in_addmulti(&mreq->imr_multiaddr, ifp)) == NULL) {
799             error = ENOBUFS;
800             break;
801         }
802         ++imo->imo_num_memberships;
803         break;
```

ip_output.c

Figure 12.26 ip_setmoptions function: joining a multicast group.

Validation

733–746 ip_setmoptions starts by validating the request. If no mbuf was passed, if it is not the correct size, or if the address (imr_multiaddr) within the structure is not a multicast group, then ip_setmoptions posts EINVAL. mreq points to the valid ip_mreq structure.

Locate the interface

747–774 If the unicast address of the interface (imr_interface) is INADDR_ANY, ip_setmoptions must locate the default interface for the specified group. A route structure is constructed with the group as the desired destination and passed to rtalloc, which locates a route for the group. If no route is available, the add request fails with the error EADDRNOTAVAIL. If a route is located, a pointer to the outgoing interface for the route is saved in ifp and the route entry, which is no longer needed, is released.

If imr_interface is not INADDR_ANY, an explicit interface has been requested. The macro INADDR_TO_IFP searches for the interface with the requested unicast address. If an interface isn't found or if it does not support multicasting, the request fails with the error EADDRNOTAVAIL.

We described the `route` structure in Section 8.5. The function `rtalloc` is described in Section 19.2, and the use of the routing tables for selecting multicast interfaces is described in Chapter 14.

Already a member?

775–792 The last check performed on the request is to examine the `imo_membership` array to see if the selected interface is already a member of the requested group. If the `for` loop finds a match, or if the membership array is full, `EADDRINUSE` or `ETOOMANYREFS` is posted and processing of this option stops.

Join the group

793–803 At this point the request looks reasonable. `in_addmulti` arranges for IP to begin receiving multicast datagrams for the group. The pointer returned by `in_addmulti` points to a new or existing `in_multi` structure (Figure 12.12) in the interface's multicast group list. It is saved in the membership array and the size of the array is incremented.

`in_addmulti` Function

`in_addmulti` and its companion `in_delmulti` (Figures 12.27 and 12.36) maintain the list of multicast groups that an interface has joined. Join requests either add a new `in_multi` structure to the interface list or increase the reference count of an existing structure.

in.c

```
469 struct in_multi *
470 in_addmulti(ap, ifp)
471 struct in_addr *ap;
472 struct ifnet *ifp;
473 {
474     struct in_multi *inm;
475     struct ifreq ifr;
476     struct in_ifaddr *ia;
477     int     s = splnet();

478     /*
479      * See if address already in list.
480      */
481     IN_LOOKUP_MULTI(*ap, ifp, inm);
482     if (inm != NULL) {
483         /*
484          * Found it; just increment the reference count.
485          */
486         ++inm->inm_refcount;
487     } else {
```

in.c

Figure 12.27 `in_addmulti` function: first half.

Already a member

469–487 `ip_setmoptions` has already verified that `ap` points to a class D multicast address and that `ifp` points to a multicast-capable interface. `IN_LOOKUP_MULTI` (Figure 12.14) determines if the interface is already a member of the group. If it is a member, `in_addmulti` updates the reference count and returns.

If the interface is not yet a member of the group, the code in Figure 12.28 is executed.

in.c

```
    } else {
        /*
         * New address; allocate a new multicast record
         * and link it into the interface's multicast list.
         */
        inm = (struct in_multi *) malloc(sizeof(*inm),
                                         M_IPMADDR, M_NOWAIT);
        if (inm == NULL) {
            splx(s);
            return (NULL);
        }
        inm->inm_addr = *ap;
        inm->inm_ifp = ifp;
        inm->inm_refcount = 1;
        IFP_TO_IA(ifp, ia);
        if (ia == NULL) {
            free(inm, M_IPMADDR);
            splx(s);
            return (NULL);
        }
        inm->inm_ia = ia;
        inm->inm_next = ia->ia_multiaddrs;
        ia->ia_multiaddrs = inm;
        /*
         * Ask the network driver to update its multicast reception
         * filter appropriately for the new address.
         */
        ((struct sockaddr_in *) &ifr.ifr_addr)->sin_family = AF_INET;
        ((struct sockaddr_in *) &ifr.ifr_addr)->sin_addr = *ap;
        if ((ifp->if_ioctl == NULL) ||
            (*ifp->if_ioctl) (ifp, SIOCADDMULTI, (caddr_t) & ifr) != 0) {
            ia->ia_multiaddrs = inm->inm_next;
            free(inm, M_IPMADDR);
            splx(s);
            return (NULL);
        }
        /*
         * Let IGMP know that we have joined a new IP multicast group.
         */
        igmp_joingroup(inm);
    }
    splx(s);
    return (inm);
}
```

in.c

Figure 12.28 `in_addmulti` function: second half.

Update the `in_multi` list

487–509 If the interface isn't a member yet, `in_addmulti` allocates, initializes, and inserts the new `in_multi` structure at the front of the `ia_multiaddrs` list in the interface's `in_ifaddr` structure (Figure 12.13).

Update the interface and announce the change

510–530 If the interface driver has defined an `if_ioctl` function, `in_addmulti` constructs an `ifreq` structure (Figure 4.23) containing the group address and passes the `SIOCADDMULTI` request to the interface. If the interface rejects the request, the `in_multi` structure is unlinked from the interface and released. Finally, `in_addmulti` calls `igmp_joingroup` to propagate the membership change to other hosts and routers.

`in_addmulti` returns a pointer to the `in_multi` structure or null if an error occurred.

`slioctl` and `loioctl` Functions: `SIOCADDMULTI` and `SIOCDELMULTI`

Multicast group processing for the SLIP and loopback interfaces is trivial: there is nothing to do other than error checking. Figure 12.29 shows the SLIP processing.

if_sl.c

```
673         case SIOCADDMULTI:
674         case SIOCDELMULTI:
675             ifr = (struct ifreq *) data;
676             if (ifr == 0) {
677                 error = EAFNOSUPPORT;    /* XXX */
678                 break;
679             }
680             switch (ifr->ifr_addr.sa_family) {

681             case AF_INET:
682                 break;

683             default:
684                 error = EAFNOSUPPORT;
685                 break;
686             }
687             break;
```

if_sl.c

Figure 12.29 `slioctl` function: multicast processing.

673–687 `EAFNOSUPPORT` is returned whether the request is empty or not for the `AF_INET` protocol family.

Figure 12.30 shows the loopback processing.

152–166 The processing for the loopback interface is identical to the SLIP code in Figure 12.29. `EAFNOSUPPORT` is returned whether the request is empty or not for the `AF_INET` protocol family.

if_loop.c

```
    case SIOCADDMULTI:
    case SIOCDELMULTI:
        ifr = (struct ifreq *) data;
        if (ifr == 0) {
            error = EAFNOSUPPORT;    /* XXX */
            break;
        }
        switch (ifr->ifr_addr.sa_family) {

        case AF_INET:
            break;

        default:
            error = EAFNOSUPPORT;
            break;
        }
        break;
```

if_loop.c

Figure 12.30 `loioctl` function: multicast processing.

`leioctl` Function: `SIOCADDMULTI` and `SIOCDELMULTI`

Recall from Figure 4.2 that `leioctl` is the `if_ioctl` function for the LANCE Ethernet driver. Figure 12.31 shows the code for the `SIOCADDMULTI` and `SIOCDELMULTI` options.

if_le.c

```
657    case SIOCADDMULTI:
658    case SIOCDELMULTI:
659        /* Update our multicast list  */
660        error = (cmd == SIOCADDMULTI) ?
661            ether_addmulti((struct ifreq *) data, &le->sc_ac) :
662            ether_delmulti((struct ifreq *) data, &le->sc_ac);

663        if (error == ENETRESET) {
664            /*
665             * Multicast list has changed; set the hardware
666             * filter accordingly.
667             */
668            lereset(ifp->if_unit);
669            error = 0;
670        }
671        break;
```

if_le.c

Figure 12.31 `leioctl` function: multicast processing.

657–671 `leioctl` passes add and delete requests directly to the `ether_addmulti` or `ether_delmulti` functions. Both functions return `ENETRESET` if the request changes the set of IP multicast addresses that must be received by the physical hardware. If this occurs, `leioctl` calls `lereset` to reinitialize the hardware with the new multicast reception list.

We don't show lereset, as it is specific to the LANCE Ethernet hardware. For multicasting, lereset arranges for the hardware to receive frames addressed to any of the Ethernet multicast addresses contained in the ether_multi list associated with the interface. The LANCE driver uses a hashing mechanism if each entry on the multicast list is a single address. The hash code allows the hardware to receive multicast packets selectively. If the driver finds an entry that describes a range of addresses, it abandons the hash strategy and configures the hardware to receive *all* multicast packets. If the driver must fall back to receiving all Ethernet multicast addresses, the IFF_ALLMULTI flag is on when lereset returns.

ether_addmulti Function

Every Ethernet driver calls ether_addmulti to process the SIOCADDMULTI request. This function maps the IP class D address to the appropriate Ethernet multicast address (Figure 12.5) and updates the ether_multi list. Figure 12.32 shows the first half of the ether_addmulti function.

Initialize address range

366–399 First, ether_addmulti initializes a range of multicast addresses in addrlo and addrhi (both are arrays of six unsigned characters). If the requested address is from the AF_UNSPEC family, ether_addmulti assumes the address is an explicit Ethernet multicast address and copies it into addrlo and addrhi. If the address is in the AF_INET family and is INADDR_ANY (0.0.0.0), ether_addmulti initializes addrlo to ether_ipmulticast_min and addrhi to ether_ipmulticast_max. These two constant Ethernet addresses are defined as:

```
u_char  ether_ipmulticast_min[6] = { 0x01, 0x00, 0x5e, 0x00, 0x00, 0x00 };
u_char  ether_ipmulticast_max[6] = { 0x01, 0x00, 0x5e, 0x7f, 0xff, 0xff };
```

As with etherbroadcastaddr (Section 4.3), this is a convenient way to define a 48-bit constant.

IP multicast routers must listen for all IP multicasts. Specifying the group as INADDR_ANY is considered a request to join *every* IP multicast group. The Ethernet address range selected in this case spans the entire block of IP multicast addresses allocated to the IANA.

The mrouted(8) daemon issues a SIOCADDMULTI request with INADDR_ANY when it begins routing packets for a multicast interface.

ETHER_MAP_IP_MULTICAST maps any other specific IP multicast group to the appropriate Ethernet multicast address. Requests for other address families are rejected with an EAFNOSUPPORT error.

While the Ethernet multicast list supports address ranges, there is no way for a process or the kernel to request a specific range, other than to enumerate the addresses, since addrlo and addrhi are always set to the same address.

The second half of ether_addmulti, shown in Figure 12.33, verifies the address range and adds it to the list if it is new.

if_ethersubr.c

```
int
ether_addmulti(ifr, ac)
struct ifreq *ifr;
struct arpcom *ac;
{
    struct ether_multi *enm;
    struct sockaddr_in *sin;
    u_char  addrlo[6];
    u_char  addrhi[6];
    int     s = splimp();

    switch (ifr->ifr_addr.sa_family) {

    case AF_UNSPEC:
        bcopy(ifr->ifr_addr.sa_data, addrlo, 6);
        bcopy(addrlo, addrhi, 6);
        break;

    case AF_INET:
        sin = (struct sockaddr_in *) &(ifr->ifr_addr);
        if (sin->sin_addr.s_addr == INADDR_ANY) {
            /*
             * An IP address of INADDR_ANY means listen to all
             * of the Ethernet multicast addresses used for IP.
             * (This is for the sake of IP multicast routers.)
             */
            bcopy(ether_ipmulticast_min, addrlo, 6);
            bcopy(ether_ipmulticast_max, addrhi, 6);
        } else {
            ETHER_MAP_IP_MULTICAST(&sin->sin_addr, addrlo);
            bcopy(addrlo, addrhi, 6);
        }
        break;

    default:
        splx(s);
        return (EAFNOSUPPORT);
    }
```

if_ethersubr.c

Figure 12.32 `ether_addmulti` function: first half.

Already receiving

400–418 `ether_addmulti` checks the multicast bit (Figure 4.12) of the high and low addresses to ensure that they are indeed Ethernet multicast addresses. `ETHER_LOOKUP_MULTI` (Figure 12.9) determines if the hardware is already listening for the specified multicast addresses. If so, the reference count (`enm_refcount`) in the matching `ether_multi` structure is incremented and `ether_addmulti` returns 0.

Update `ether_multi` list

419–441 If this is a new address range, a new `ether_multi` structure is allocated, initialized, and linked to the `ac_multiaddrs` list in the interfaces `arpcom` structure (Figure 12.8). If `ENETRESET` is returned by `ether_addmulti`, the device driver that called

if_ethersubr.c

```
    /*
     * Verify that we have valid Ethernet multicast addresses.
     */
    if ((addrlo[0] & 0x01) != 1 || (addrhi[0] & 0x01) != 1) {
        splx(s);
        return (EINVAL);
    }
    /*
     * See if the address range is already in the list.
     */
    ETHER_LOOKUP_MULTI(addrlo, addrhi, ac, enm);
    if (enm != NULL) {
        /*
         * Found it; just increment the reference count.
         */
        ++enm->enm_refcount;
        splx(s);
        return (0);
    }
    /*
     * New address or range; malloc a new multicast record
     * and link it into the interface's multicast list.
     */
    enm = (struct ether_multi *) malloc(sizeof(*enm), M_IFMADDR, M_NOWAIT);
    if (enm == NULL) {
        splx(s);
        return (ENOBUFS);
    }
    bcopy(addrlo, enm->enm_addrlo, 6);
    bcopy(addrhi, enm->enm_addrhi, 6);
    enm->enm_ac = ac;
    enm->enm_refcount = 1;
    enm->enm_next = ac->ac_multiaddrs;
    ac->ac_multiaddrs = enm;
    ac->ac_multicnt++;
    splx(s);
    /*
     * Return ENETRESET to inform the driver that the list has changed
     * and its reception filter should be adjusted accordingly.
     */
    return (ENETRESET);
}
```

if_ethersubr.c

Figure 12.33 `ether_addmulti` function: second half.

the function knows that the multicast list has changed and the hardware reception filter must be updated.

Figure 12.34 shows the relationships between the `ip_moptions`, `in_multi`, and `ether_multi` structures after the LANCE Ethernet interface has joined the all-hosts group.

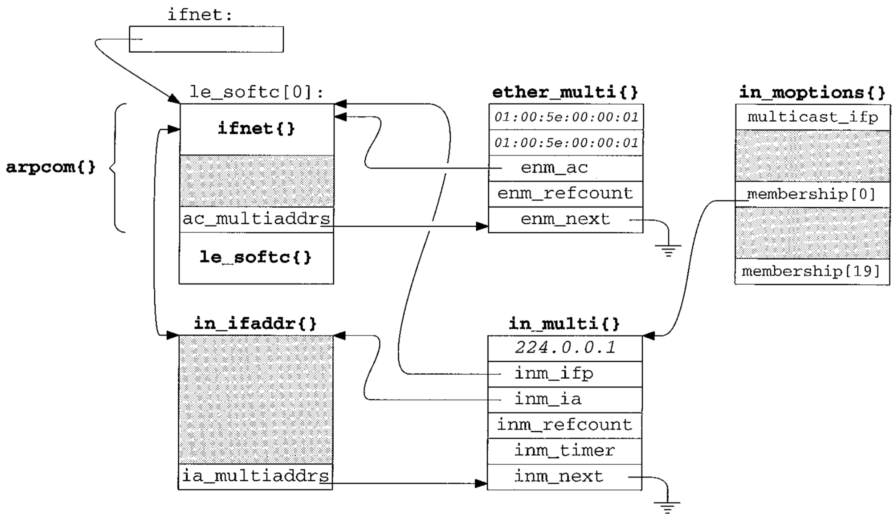

Figure 12.34 Overview of multicast data structures.

12.12 Leaving an IP Multicast Group

In general, the steps required to leave a group are the reverse of those required to join a group. The membership list in the ip_moptions structure is updated, the in_multi list for the IP interface is updated, and the ether_multi list for the device is updated. First, we return to ip_setmoptions and the IP_DROP_MEMBERSHIP case, which we show in Figure 12.35.

Validation

804–830 The mbuf must contain an ip_mreq structure, within the structure imr_multiaddr must be a multicast group, and there must be an interface associated with the unicast address imr_interface. If these conditions aren't met, EINVAL or EADDRNOTAVAIL is posted and processing continues at the end of the switch.

Delete membership references

831–856 The for loop searches the group membership list for an in_multi structure with the requested {interface, group} pair. If a match isn't found, EADDRNOTAVAIL is posted. Otherwise, in_delmulti updates the in_multi list and the second for loop removes the unused entry in the membership array by shifting subsequent entries to fill the gap. The size of the array is updated accordingly.

ip_output.c

```
    case IP_DROP_MEMBERSHIP:
        /*
         * Drop a multicast group membership.
         * Group must be a valid IP multicast address.
         */
        if (m == NULL || m->m_len != sizeof(struct ip_mreq)) {
            error = EINVAL;
            break;
        }
        mreq = mtod(m, struct ip_mreq *);
        if (!IN_MULTICAST(ntohl(mreq->imr_multiaddr.s_addr))) {
            error = EINVAL;
            break;
        }
        /*
         * If an interface address was specified, get a pointer
         * to its ifnet structure.
         */
        if (mreq->imr_interface.s_addr == INADDR_ANY)
            ifp = NULL;
        else {
            INADDR_TO_IFP(mreq->imr_interface, ifp);
            if (ifp == NULL) {
                error = EADDRNOTAVAIL;
                break;
            }
        }
        /*
         * Find the membership in the membership array.
         */
        for (i = 0; i < imo->imo_num_memberships; ++i) {
            if ((ifp == NULL ||
                 imo->imo_membership[i]->inm_ifp == ifp) &&
                imo->imo_membership[i]->inm_addr.s_addr ==
                mreq->imr_multiaddr.s_addr)
                break;
        }
        if (i == imo->imo_num_memberships) {
            error = EADDRNOTAVAIL;
            break;
        }
        /*
         * Give up the multicast address record to which the
         * membership points.
         */
        in_delmulti(imo->imo_membership[i]);
        /*
         * Remove the gap in the membership array.
         */
        for (++i; i < imo->imo_num_memberships; ++i)
            imo->imo_membership[i - 1] = imo->imo_membership[i];
        --imo->imo_num_memberships;
        break;
```

ip_output.c

Figure 12.35 ip_setmoptions function: leaving a multicast group.

`in_delmulti` Function

Since many processes may be receiving multicast datagrams, calling `in_delmulti` (Figure 12.36) results only in leaving the specified group when there are no more references to the `in_multi` structure.

in.c

```
534 int
535 in_delmulti(inm)
536 struct in_multi *inm;
537 {
538     struct in_multi **p;
539     struct ifreq ifr;
540     int     s = splnet();

541     if (--inm->inm_refcount == 0) {
542         /*
543          * No remaining claims to this record; let IGMP know that
544          * we are leaving the multicast group.
545          */
546         igmp_leavegroup(inm);
547         /*
548          * Unlink from list.
549          */
550         for (p = &inm->inm_ia->ia_multiaddrs;
551              *p != inm;
552              p = &(*p)->inm_next)
553             continue;
554         *p = (*p)->inm_next;
555         /*
556          * Notify the network driver to update its multicast reception
557          * filter.
558          */
559         ((struct sockaddr_in *) &(ifr.ifr_addr))->sin_family = AF_INET;
560         ((struct sockaddr_in *) &(ifr.ifr_addr))->sin_addr =
561             inm->inm_addr;
562         (*inm->inm_ifp->if_ioctl) (inm->inm_ifp, SIOCDELMULTI,
563                                    (caddr_t) & ifr);
564         free(inm, M_IPMADDR);
565     }
566     splx(s);
567 }
```

in.c

Figure 12.36 `in_delmulti` function.

Update `in_multi` structure

534–567 `in_delmulti` starts by decrementing the reference count of the `in_multi` structure and returning if the reference count is nonzero. If the reference count drops to 0, there are no longer any processes waiting for the multicast datagrams on the specified {interface, group} pair. `igmp_leavegroup` is called, but as we'll see in Section 13.8, the function does nothing.

The `for` loop traverses the linked list of `in_multi` structures until it locates the matching structure.

> The body of this `for` loop consists of the single `continue` statement. All the work is done by the expressions at the top of the loop. The `continue` is not required but stands out more clearly than a bare semicolon.
>
> The `ETHER_LOOKUP_MULTI` macro in Figure 12.9 does not use the `continue` and the bare semicolon is almost undetectable.

After the loop, the matching `in_multi` structure is unlinked and `in_delmulti` issues the `SIOCDELMULTI` request to the interface so that any device-specific data structures can be updated. For Ethernet interfaces, this means the `ether_multi` list is updated. Finally, the `in_multi` structure is released.

> The `SIOCDELMULTI` case for the LANCE driver was included in Figure 12.31 where we also discussed the `SIOCADDMULTI` case.

`ether_delmulti` Function

When IP releases an `in_multi` structure associated with an Ethernet device, the device may be able to release the matching `ether_multi` structure. We say *may* because IP may be unaware of other software listening for IP multicasts. When the reference count for the `ether_multi` structure drops to 0, it can be released. Figure 12.37 shows the `ether_delmulti` function.

445–479 `ether_delmulti` initializes the `addrlo` and `addrhi` arrays in the same way as `ether_addmulti` does.

Locate `ether_multi` structure

480–494 `ETHER_LOOKUP_MULTI` locates a matching `ether_multi` structure. If it isn't found, `ENXIO` is returned. If the matching structure is found, the reference count is decremented and if the result is nonzero, `ether_delmulti` returns immediately. In this case, the structure may not be released because another protocol has elected to receive the same multicast packets.

Delete `ether_multi` structure

495–511 The `for` loop searches the `ether_multi` list for the matching address range. The matching structure is unlinked from the list and released. Finally, the size of the list is updated and `ENETRESET` is returned so that the device driver can update its hardware reception filter.

if_ethersubr.c

```
int
ether_delmulti(ifr, ac)
struct ifreq *ifr;
struct arpcom *ac;
{
    struct ether_multi *enm;
    struct ether_multi **p;
    struct sockaddr_in *sin;
    u_char  addrlo[6];
    u_char  addrhi[6];
    int     s = splimp();

    switch (ifr->ifr_addr.sa_family) {

    case AF_UNSPEC:
        bcopy(ifr->ifr_addr.sa_data, addrlo, 6);
        bcopy(addrlo, addrhi, 6);
        break;

    case AF_INET:
        sin = (struct sockaddr_in *) &(ifr->ifr_addr);
        if (sin->sin_addr.s_addr == INADDR_ANY) {
            /*
             * An IP address of INADDR_ANY means stop listening
             * to the range of Ethernet multicast addresses used
             * for IP.
             */
            bcopy(ether_ipmulticast_min, addrlo, 6);
            bcopy(ether_ipmulticast_max, addrhi, 6);
        } else {
            ETHER_MAP_IP_MULTICAST(&sin->sin_addr, addrlo);
            bcopy(addrlo, addrhi, 6);
        }
        break;

    default:
        splx(s);
        return (EAFNOSUPPORT);
    }

    /*
     * Look up the address in our list.
     */
    ETHER_LOOKUP_MULTI(addrlo, addrhi, ac, enm);
    if (enm == NULL) {
        splx(s);
        return (ENXIO);
    }
    if (--enm->enm_refcount != 0) {
        /*
         * Still some claims to this record.
         */
        splx(s);
        return (0);
    }
```

```
    /*
     * No remaining claims to this record; unlink and free it.
     */
    for (p = &enm->enm_ac->ac_multiaddrs;
         *p != enm;
         p = &(*p)->enm_next)
        continue;
    *p = (*p)->enm_next;
    free(enm, M_IFMADDR);
    ac->ac_multicnt--;
    splx(s);
    /*
     * Return ENETRESET to inform the driver that the list has changed
     * and its reception filter should be adjusted accordingly.
     */
    return (ENETRESET);
}
```

if_ethersubr.c

Figure 12.37 ether_delmulti function.

12.13 ip_getmoptions Function

Fetching the current option settings is considerably easier than setting them. All the work is done by ip_getmoptions, shown in Figure 12.38.

Copy the option data and return

876–914 The three arguments to ip_getmoptions are: optname, the option to fetch; imo, the ip_moptions structure; and mp, which points to a pointer to an mbuf. m_get allocates an mbuf to hold the option data. For each of the three options, a pointer (addr, ttl, and loop, respectively) is initialized to the data area of the mbuf and the length of the mbuf is set to the length of the option data.

For IP_MULTICAST_IF, the unicast address found by IFP_TO_IA is returned or INADDR_ANY is returned if no explicit multicast interface has been selected.

For IP_MULTICAST_TTL, imo_multicast_ttl is returned or if an explicit multicast TTL has not been selected, 1 (IP_DEFAULT_MULTICAST_TTL) is returned.

For IP_MULTICAST_LOOP, imo_multicast_loop is returned or if an explicit multicast loopback policy has not been selected, 1 (IP_DEFAULT_MULTICAST_LOOP) is returned.

Finally, EOPNOTSUPP is returned if the option isn't recognized.

ip_output.c

```
int
ip_getmoptions(optname, imo, mp)
int     optname;
struct ip_moptions *imo;
struct mbuf **mp;
{
    u_char *ttl;
    u_char *loop;
    struct in_addr *addr;
    struct in_ifaddr *ia;

    *mp = m_get(M_WAIT, MT_SOOPTS);

    switch (optname) {

    case IP_MULTICAST_IF:
        addr = mtod(*mp, struct in_addr *);
        (*mp)->m_len = sizeof(struct in_addr);
        if (imo == NULL || imo->imo_multicast_ifp == NULL)
            addr->s_addr = INADDR_ANY;
        else {
            IFP_TO_IA(imo->imo_multicast_ifp, ia);
            addr->s_addr = (ia == NULL) ? INADDR_ANY
                : IA_SIN(ia)->sin_addr.s_addr;
        }
        return (0);

    case IP_MULTICAST_TTL:
        ttl = mtod(*mp, u_char *);
        (*mp)->m_len = 1;
        *ttl = (imo == NULL) ? IP_DEFAULT_MULTICAST_TTL
            : imo->imo_multicast_ttl;
        return (0);

    case IP_MULTICAST_LOOP:
        loop = mtod(*mp, u_char *);
        (*mp)->m_len = 1;
        *loop = (imo == NULL) ? IP_DEFAULT_MULTICAST_LOOP
            : imo->imo_multicast_loop;
        return (0);

    default:
        return (EOPNOTSUPP);
    }
}
```

ip_output.c

Figure 12.38 `ip_getmoptions` function.

12.14 Multicast Input Processing: `ipintr` Function

Now that we have described multicast addressing, group memberships, and the various data structures associated with IP and Ethernet multicasting, we can move on to multicast datagram processing.

In Figure 4.13 we saw that an incoming Ethernet multicast packet is detected by `ether_input`, which sets the `M_MCAST` flag in the mbuf header before placing an IP packet on the IP input queue (`ipintrq`). The `ipintr` function processes each packet in turn. The multicast processing code we omitted from the discussion of `ipintr` appears in Figure 12.39.

The code is from the section of `ipintr` that determines if a packet is addressed to the local system or if it should be forwarded. At this point, the packet has been checked for errors and any options have been processed. `ip` points to the IP header within the packet.

Forward packets if configured as multicast router

214–245 This entire section of code is skipped if the destination address is not an IP multicast group. If the address is a multicast group and the system is configured as an IP multicast router (`ip_mrouter`), `ip_id` is converted to network byte order (the form that `ip_mforward` expects), and the packet is passed to `ip_mforward`. If `ip_mforward` returns a nonzero value, an error was detected or the packet arrived through a *multicast tunnel*. The packet is discarded and `ips_cantforward` incremented.

> We describe multicast tunnels in Chapter 14. They transport multicast packets between multicast routers separated by standard IP routers. Packets that arrive through a tunnel must be processed by `ip_mforward` and not `ipintr`.

If `ip_mforward` returns 0, `ip_id` is converted back to host byte order and `ipintr` may continue processing the packet.

If `ip` points to an IGMP packet, it is accepted and execution continues at `ours` (`ipintr`, Figure 10.11). A multicast router must accept all IGMP packets irrespective of their individual destination groups or of the group memberships of the incoming interface. The IGMP packets contain announcements of membership changes.

246–257 The remaining code in Figure 12.39 is executed whether or not the system is configured as a multicast router. `IN_LOOKUP_MULTI` searches the list of multicast groups that the interface has joined. If a match is not found, the packet is discarded. This occurs when the hardware filter accepts unwanted packets or when a group associated with the interface and the destination group of the packet map to the same Ethernet multicast address.

If the packet is accepted, execution continues at the label `ours` in `ipintr` (Figure 10.11).

ip_input.c

```
    if (IN_MULTICAST(ntohl(ip->ip_dst.s_addr))) {
        struct in_multi *inm;
        extern struct socket *ip_mrouter;

        if (ip_mrouter) {
            /*
             * If we are acting as a multicast router, all
             * incoming multicast packets are passed to the
             * kernel-level multicast forwarding function.
             * The packet is returned (relatively) intact; if
             * ip_mforward() returns a non-zero value, the packet
             * must be discarded, else it may be accepted below.
             *
             * (The IP ident field is put in the same byte order
             * as expected when ip_mforward() is called from
             * ip_output().)
             */
            ip->ip_id = htons(ip->ip_id);
            if (ip_mforward(m, m->m_pkthdr.rcvif) != 0) {
                ipstat.ips_cantforward++;
                m_freem(m);
                goto next;
            }
            ip->ip_id = ntohs(ip->ip_id);

            /*
             * The process-level routing demon needs to receive
             * all multicast IGMP packets, whether or not this
             * host belongs to their destination groups.
             */
            if (ip->ip_p == IPPROTO_IGMP)
                goto ours;
            ipstat.ips_forward++;
        }
        /*
         * See if we belong to the destination multicast group on the
         * arrival interface.
         */
        IN_LOOKUP_MULTI(ip->ip_dst, m->m_pkthdr.rcvif, inm);
        if (inm == NULL) {
            ipstat.ips_cantforward++;
            m_freem(m);
            goto next;
        }
        goto ours;
    }
```

ip_input.c

Figure 12.39 `ipintr` function: multicast input processing.

12.15 Multicast Output Processing: ip_output Function

When we discussed ip_output in Chapter 8, we postponed discussion of the mp argument to ip_output and the multicast processing code. In ip_output, if mp points to an ip_moptions structure, it overrides the default multicast output processing. The omitted code from ip_output appears in Figures 12.40 and 12.41. ip points to the outgoing packet, m points to the mbuf holding the packet, and ifp points to the interface selected by the routing tables for the destination group.

ip_output.c

```
    if (IN_MULTICAST(ntohl(ip->ip_dst.s_addr))) {
        struct in_multi *inm;
        extern struct ifnet loif;

        m->m_flags |= M_MCAST;
        /*
         * IP destination address is multicast.  Make sure "dst"
         * still points to the address in "ro".  (It may have been
         * changed to point to a gateway address, above.)
         */
        dst = (struct sockaddr_in *) &ro->ro_dst;
        /*
         * See if the caller provided any multicast options
         */
        if (imo != NULL) {
            ip->ip_ttl = imo->imo_multicast_ttl;
            if (imo->imo_multicast_ifp != NULL)
                ifp = imo->imo_multicast_ifp;
        } else
            ip->ip_ttl = IP_DEFAULT_MULTICAST_TTL;
        /*
         * Confirm that the outgoing interface supports multicast.
         */
        if ((ifp->if_flags & IFF_MULTICAST) == 0) {
            ipstat.ips_noroute++;
            error = ENETUNREACH;
            goto bad;
        }
        /*
         * If source address not specified yet, use address
         * of outgoing interface.
         */
        if (ip->ip_src.s_addr == INADDR_ANY) {
            struct in_ifaddr *ia;

            for (ia = in_ifaddr; ia; ia = ia->ia_next)
                if (ia->ia_ifp == ifp) {
                    ip->ip_src = IA_SIN(ia)->sin_addr;
                    break;
                }
        }
```

ip_output.c

Figure 12.40 ip_output function: defaults and source address.

Establish defaults

129–155 The code in Figure 12.40 is executed only if the packet is destined for a multicast group. If so, ip_output sets M_MCAST in the mbuf and dst is reset to the final destination as it may have been set to the next-hop router earlier in ip_output (Figure 8.24).

If an ip_moptions structure was passed, ip_ttl and ifp are changed accordingly. Otherwise, ip_ttl is set to 1 (IP_DEFAULT_MULTICAST_TTL), which prevents the multicast from escaping to a remote network. The interface selected by consulting the routing tables or the interface specified within the ip_moptions structure must support multicasting. If they do not, ip_output discards the packet and returns ENETUNREACH.

Select source address

156–167 If the source address is unspecified, the for loop finds the Internet unicast address associated with the outgoing interface and fills in ip_src in the IP header.

Unlike a unicast packet, an outgoing multicast packet may be transmitted on more than one interface if the system is configured as a multicast router. Even if the system is not a multicast router, the outgoing interface may be a member of the destination group and may need to receive the packet. Finally, we need to consider the multicast loopback policy and the loopback interface itself. Taking all this into account, there are three questions to consider:

- Should the packet be received on the outgoing interface?
- Should the packet be forwarded to other interfaces?
- Should the packet be transmitted on the outgoing interface?

Figure 12.41 shows the code from ip_output that answers these questions.

Loopback or not?

168–176 If IN_LOOKUP_MULTI determines that the outgoing interface is a member of the destination group and imo_multicast_loop is nonzero, the packet is queued for *input* on the output interface by ip_mloopback. In this case, the original packet is *not* considered for forwarding, since the copy is forwarded during input processing if necessary.

Forward or not?

178–197 If the packet is *not* looped back, but the system is configured as a multicast router and the packet is eligible for forwarding, ip_mforward distributes copies to other multicast interfaces. If ip_mforward does not return 0, ip_output discards the packet and does not attempt to transmit it. This indicates an error with the packet.

To prevent infinite recursion between ip_mforward and ip_output, ip_mforward always turns on IP_FORWARDING before calling ip_output. A datagram originating on the system is eligible for forwarding because the transport protocols do not turn on IP_FORWARDING.

ip_output.c

```
168         IN_LOOKUP_MULTI(ip->ip_dst, ifp, inm);
169         if (inm != NULL &&
170             (imo == NULL || imo->imo_multicast_loop)) {
171             /*
172              * If we belong to the destination multicast group
173              * on the outgoing interface, and the caller did not
174              * forbid loopback, loop back a copy.
175              */
176             ip_mloopback(ifp, m, dst);
177         } else {
178             /*
179              * If we are acting as a multicast router, perform
180              * multicast forwarding as if the packet had just
181              * arrived on the interface to which we are about
182              * to send.  The multicast forwarding function
183              * recursively calls this function, using the
184              * IP_FORWARDING flag to prevent infinite recursion.
185              *
186              * Multicasts that are looped back by ip_mloopback(),
187              * above, will be forwarded by the ip_input() routine,
188              * if necessary.
189              */
190             extern struct socket *ip_mrouter;
191             if (ip_mrouter && (flags & IP_FORWARDING) == 0) {
192                 if (ip_mforward(m, ifp) != 0) {
193                     m_freem(m);
194                     goto done;
195                 }
196             }
197         }
198         /*
199          * Multicasts with a time-to-live of zero may be looped-
200          * back, above, but must not be transmitted on a network.
201          * Also, multicasts addressed to the loopback interface
202          * are not sent -- the above call to ip_mloopback() will
203          * loop back a copy if this host actually belongs to the
204          * destination group on the loopback interface.
205          */
206         if (ip->ip_ttl == 0 || ifp == &loif) {
207             m_freem(m);
208             goto done;
209         }
210         goto sendit;
211     }
```

ip_output.c

Figure 12.41 ip_output function: loopback, forward, and send.

Transmit or not?

198–209 Packets with a TTL of 0 may be looped back, but they are never forwarded (ip_mforward discards them) and are never transmitted. If the TTL is 0 or if the output interface is the loopback interface, ip_output discards the packet since the TTL has expired or the packet has already been looped back by ip_mloopback.

Send packet

210–211 If the packet has made it this far, it is ready to be physically transmitted on the output interface. The code at `sendit` (`ip_output`, Figure 8.25) may fragment the datagram before passing it (or the resulting fragments) to the interface's `if_output` function. We'll see in Section 21.10 that the Ethernet output function, `ether_output`, calls `arpresolve`, which calls `ETHER_MAP_IP_MULTICAST` to construct an Ethernet multicast destination address based on the IP multicast destination address.

`ip_mloopback` Function

`ip_mloopback` relies on `looutput` (Figure 5.27) to do its job. Instead of passing a pointer to the loopback interface to `looutput`, `ip_mloopback` passes a pointer to the output multicast interface. The `ip_mloopback` function is shown in Figure 12.42.

ip_output.c

```
935 static void
936 ip_mloopback(ifp, m, dst)
937 struct ifnet *ifp;
938 struct mbuf *m;
939 struct sockaddr_in *dst;
940 {
941     struct ip *ip;
942     struct mbuf *copym;

943     copym = m_copy(m, 0, M_COPYALL);
944     if (copym != NULL) {
945         /*
946          * We don't bother to fragment if the IP length is greater
947          * than the interface's MTU.  Can this possibly matter?
948          */
949         ip = mtod(copym, struct ip *);
950         ip->ip_len = htons((u_short) ip->ip_len);
951         ip->ip_off = htons((u_short) ip->ip_off);
952         ip->ip_sum = 0;
953         ip->ip_sum = in_cksum(copym, ip->ip_hl << 2);
954         (void) looutput(ifp, copym, (struct sockaddr *) dst, NULL);
955     }
956 }
```

ip_output.c

Figure 12.42 `ip_mloopback` function.

Duplicate and queue packet

929–956 Copying the packet isn't enough; the packet must look as though it was received on the output interface, so `ip_mloopback` converts `ip_len` and `ip_off` to network byte order and computes the checksum for the packet. `looutput` takes care of putting the packet on the IP input queue.

12.16 Performance Considerations

The multicast implementation in Net/3 has several potential performance bottlenecks. Since many Ethernet cards do not support perfect filtering of multicast addresses, the operating system must be prepared to discard multicast packets that pass through the hardware filter. In the worst case, an Ethernet card may fall back to receiving all multicast packets, most of which must be discarded by `ipintr` when they are found not to contain a valid IP multicast group address.

IP uses a simple linear list and linear search to filter incoming IP datagrams. If the list grows to any appreciable length, a caching mechanism such as moving the most recently received address to the front of the list would help performance.

12.17 Summary

In this chapter we described how a single host processes IP multicast datagrams. We looked at the format of an IP class D address and an Ethernet multicast address and the mapping between the two.

We discussed the `in_multi` and `ether_multi` structures, and we saw that each IP multicast interface maintains its own group membership list and that each Ethernet interface maintains a list of Ethernet multicast addresses.

During input processing, IP multicasts are accepted only if they arrive on an interface that is a member of their destination group, although they may be forwarded to other interfaces if the system is configured as a multicast router.

Systems configured as multicast routers must accept all multicast packets on every interface. This can be done quickly by issuing the `SIOCADDMULTI` command for the `INADDR_ANY` address.

The `ip_moptions` structure is the cornerstone of multicast output processing. It controls the selection of an output interface, the TTL field of the multicast datagram, and the loopback policy. It also holds references to the `in_multi` structures, which determine when an interface joins or leaves an IP multicast group.

We also discussed the two concepts implemented by the multicast TTL value: packet lifetime and packet scope.

Exercises

12.1 What is the difference between sending an IP broadcast packet to 255.255.255.255 and sending an IP multicast to the all-hosts group 224.0.0.1?

12.2 Why are interfaces identified by their IP unicast addresses in the multicasting code? What must be changed so that an interface could send and receive multicast datagrams but not have a unicast IP address?

12.3 In Section 12.3 we said that 32 IP groups are mapped to a single Ethernet address. Since 9 bits of a 32-bit address are not included in the mapping, why didn't we say that 512 (2^9) IP groups mapped to a single Ethernet address?

12.4 Why do you think IP_MAX_MEMBERSHIPS is set to 20? Could it be set to a larger value? Hint: Consider the size of the ip_moptions structure (Figure 12.15).

12.5 What happens when a multicast datagram is looped back by IP and is also received by the hardware interface on which it is transmitted (i.e., a nonsimplex interface)?

12.6 Draw a picture of a network with a multihomed host so that a multicast packet sent on one interface may be received on the other interface even if the host is not acting as a multicast router.

12.7 Trace the membership add request through the SLIP and loopback interfaces instead of the Ethernet interface.

12.8 How could a process request that the kernel join more than IP_MAX_MEMBERSHIPS?

12.9 Computing the checksum on a looped back packet is superfluous. Design a method to avoid the checksum computation for loopback packets.

12.10 How many IP multicast groups could an interface join without reusing an Ethernet multicast address?

12.11 The careful reader might have noticed that in_delmulti assumes that the interface has defined an ioctl function when it issues the SIOCDELMULTI request. Why is this OK?

12.12 What happens to the mbuf allocated in ip_getmoptions if an unrecognized option is requested?

12.13 Why is the group membership mechanism separate from the binding mechanism used to receive unicast and broadcast datagrams?

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.

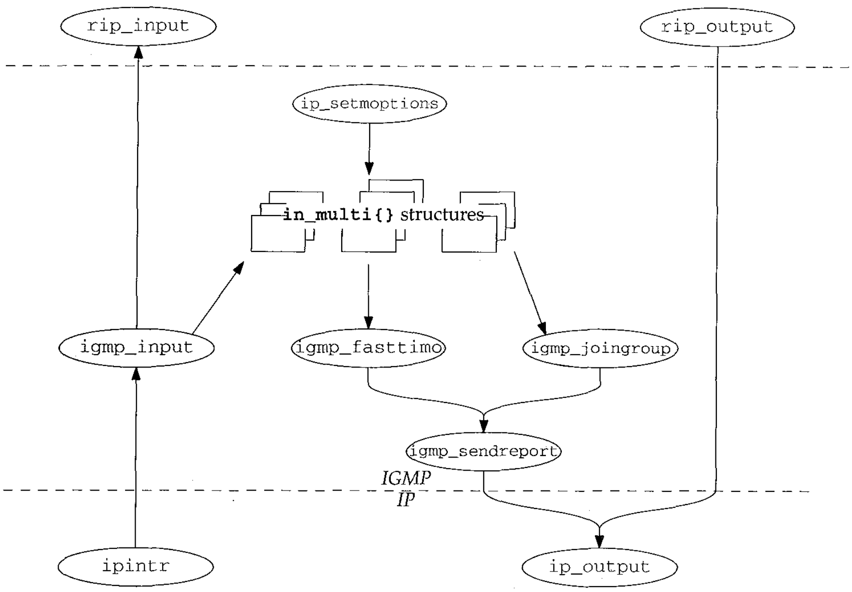

Figure 13.1 Summary of IGMP processing.

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.

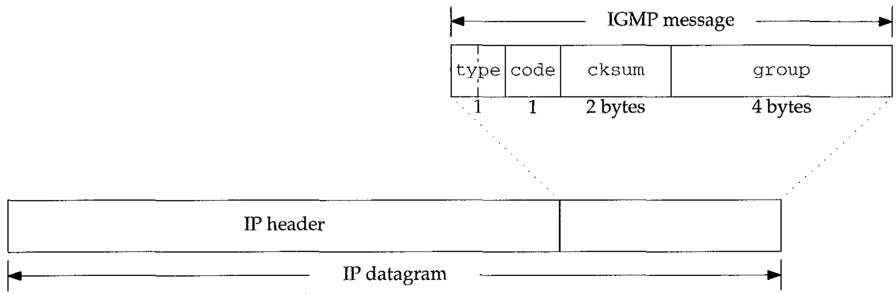

Figure 13.8 An IGMP message (igmp_ omitted).

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

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

and

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

igmp_var.h

```
/*
 * Macro to compute a random timer value between 1 and (IGMP_MAX_REPORTING_
 * DELAY * countdown frequency).  We generate a "random" number by adding
 * the total number of IP packets received, our primary IP address, and the
 * multicast address being timed-out.  The 4.3 random() routine really
 * ought to be available in the kernel!
 */
#define IGMP_RANDOM_DELAY(multiaddr) \
    /* struct in_addr multiaddr; */ \
    ( (ipstat.ips_total + \
       ntohl(IA_SIN(in_ifaddr)->sin_addr.s_addr) + \
       ntohl((multiaddr).s_addr) \
      ) \
      % (IGMP_MAX_HOST_REPORT_DELAY * PR_FASTHZ) + 1 \
    )
```

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

```
/*
 * Macro to step through all of the in_multi records, one at a time.
 * The current position is remembered in "step", which the caller must
 * provide.  IN_FIRST_MULTI(), below, must be called to initialize "step"
 * and get the first record.  Both macros return a NULL "inm" when there
 * are no remaining records.
 */
#define IN_NEXT_MULTI(step, inm) \
    /* struct in_multistep  step; */ \
    /* struct in_multi *inm; */ \
{ \
    if (((inm) = (step).i_inm) != NULL) \
        (step).i_inm = (inm)->inm_next; \
    else \
        while ((step).i_ia != NULL) { \
            (inm) = (step).i_ia->ia_multiaddrs; \
            (step).i_ia = (step).i_ia->ia_next; \
            if ((inm) != NULL) { \
                (step).i_inm = (inm)->inm_next; \
                break; \
            } \
        } \
}

#define IN_FIRST_MULTI(step, inm) \
    /* struct in_multistep step; */ \
    /* struct in_multi *inm; */ \
{ \
    (step).i_ia = in_ifaddr; \
    (step).i_inm = NULL; \
    IN_NEXT_MULTI((step), (inm)); \
}
```

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.

igmp.c

```
void
igmp_fasttimo()
{
    struct in_multi *inm;
    int     s;
    struct in_multistep step;

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

    s = splnet();
    igmp_timers_are_running = 0;
    IN_FIRST_MULTI(step, inm);
    while (inm != NULL) {
        if (inm->inm_timer == 0) {
            /* do nothing */
        } else if (--inm->inm_timer == 0) {
            igmp_sendreport(inm);
        } else {
            igmp_timers_are_running = 1;
        }
        IN_NEXT_MULTI(step, inm);
    }
    splx(s);
}
```

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

```
static void
igmp_sendreport(inm)
struct in_multi *inm;
{
    struct mbuf *m;
    struct igmp *igmp;
    struct ip *ip;
    struct ip_moptions *imo;
    struct ip_moptions simo;

    MGETHDR(m, M_DONTWAIT, MT_HEADER);
    if (m == NULL)
        return;
    /*
     * Assume max_linkhdr + sizeof(struct ip) + IGMP_MINLEN
     * is smaller than mbuf size returned by MGETHDR.
     */
    m->m_data += max_linkhdr;
    m->m_len = sizeof(struct ip) + IGMP_MINLEN;
    m->m_pkthdr.len = sizeof(struct ip) + IGMP_MINLEN;

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

    igmp = (struct igmp *) (ip + 1);
    igmp->igmp_type = IGMP_HOST_MEMBERSHIP_REPORT;
    igmp->igmp_code = 0;
    igmp->igmp_group = inm->inm_addr;
    igmp->igmp_cksum = 0;
    igmp->igmp_cksum = in_cksum(m, IGMP_MINLEN);

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

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

    ++igmpstat.igps_snd_reports;
}
```

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

```
void
igmp_input(m, iphlen)
struct mbuf *m;
int     iphlen;
{
    struct igmp *igmp;
    struct ip *ip;
    int     igmplen;
    struct ifnet *ifp = m->m_pkthdr.rcvif;
    int     minlen;
    struct in_multi *inm;
    struct in_ifaddr *ia;
    struct in_multistep step;

    ++igmpstat.igps_rcv_total;

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

    /*
     * Validate lengths
     */
    if (igmplen < IGMP_MINLEN) {
        ++igmpstat.igps_rcv_tooshort;
        m_freem(m);
        return;
    }
    minlen = iphlen + IGMP_MINLEN;
    if ((m->m_flags & M_EXT || m->m_len < minlen) &&
        (m = m_pullup(m, minlen)) == 0) {
        ++igmpstat.igps_rcv_tooshort;
        return;
    }
    /*
     * Validate checksum
     */
    m->m_data += iphlen;
    m->m_len -= iphlen;
    igmp = mtod(m, struct igmp *);
    if (in_cksum(m, igmplen)) {
        ++igmpstat.igps_rcv_badsum;
        m_freem(m);
        return;
    }
    m->m_data -= iphlen;
    m->m_len += iphlen;
    ip = mtod(m, struct ip *);

    switch (igmp->igmp_type) {

                                /* switch cases */

    }
```

```
    /*
     * Pass all valid IGMP packets up to any process(es) listening
     * on a raw IGMP socket.
     */
    rip_input(m);
}
```
igmp.c

Figure 13.16 igmp_input function.

igmp.c
```
97      case IGMP_HOST_MEMBERSHIP_QUERY:
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

```
    case IGMP_HOST_MEMBERSHIP_REPORT:
        ++igmpstat.igps_rcv_reports;

        if (ifp == &loif)
            break;

        if (!IN_MULTICAST(ntohl(igmp->igmp_group.s_addr)) ||
            igmp->igmp_group.s_addr != ip->ip_dst.s_addr) {
            ++igmpstat.igps_rcv_badreports;
            m_freem(m);
            return;
        }
        /*
         * KLUDGE: if the IP source address of the report has an
         * unspecified (i.e., zero) subnet number, as is allowed for
         * a booting host, replace it with the correct subnet number
         * so that a process-level multicast routing demon can
         * determine which subnet it arrived from.  This is necessary
         * to compensate for the lack of any way for a process to
         * determine the arrival interface of an incoming packet.
         */
        if ((ntohl(ip->ip_src.s_addr) & IN_CLASSA_NET) == 0) {
            IFP_TO_IA(ifp, ia);
            if (ia)
                ip->ip_src.s_addr = htonl(ia->ia_subnet);
        }
        /*
         * If we belong to the group being reported, stop
         * our timer for that group.
         */
        IN_LOOKUP_MULTI(igmp->igmp_group, ifp, inm);
        if (inm != NULL) {
            inm->inm_timer = 0;
            ++igmpstat.igps_rcv_ourreports;
        }
        break;
```

igmp.c

Figure 13.18 Input processing of the IGMP report message.

123–146 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?

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.

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.

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

Figure 14.5 Sample IP multicast routing statistics.

These statistics are from a system with two physical interfaces and one tunnel interface. 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.

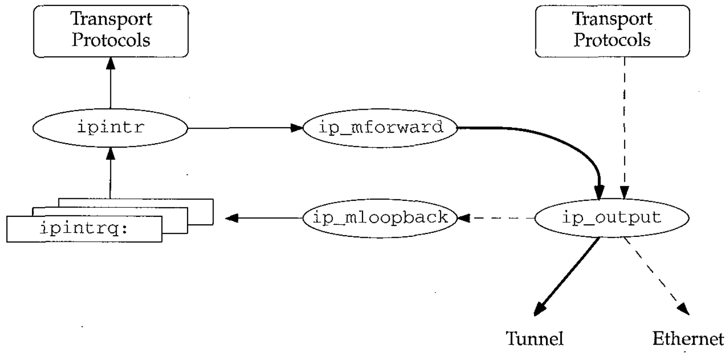

Figure 14.6 Multicast output processing with loopback.

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.

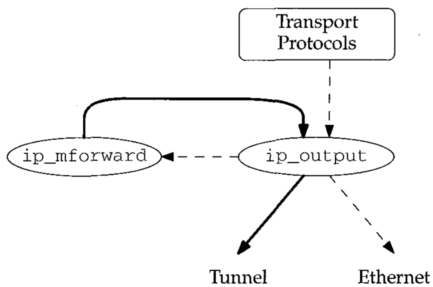

Figure 14.7 Multicast output processing with no loopback.

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 daemon. 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 documented 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

```
    case DVMRP_INIT:
    case DVMRP_DONE:
    case DVMRP_ADD_VIF:
    case DVMRP_DEL_VIF:
    case DVMRP_ADD_LGRP:
    case DVMRP_DEL_LGRP:
    case DVMRP_ADD_MRT:
    case DVMRP_DEL_MRT:
        if (op == PRCO_SETOPT) {
            error = ip_mrouter_cmd(optname, so, *m);
            if (*m)
                (void) m_free(*m);
        } else
            error = EINVAL;
        return (error);
```

raw_ip.c

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

ip_mroute.c

```
int
ip_mrouter_cmd(cmd, so, m)
int     cmd;
struct socket *so;
struct mbuf *m;
{
    int     error = 0;

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

        case DVMRP_INIT:
            error = ip_mrouter_init(so);
            break;

        case DVMRP_DONE:
            error = ip_mrouter_done();
            break;

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

        case DVMRP_DEL_VIF:
            if (m == NULL || m->m_len < sizeof(short))
                        error = EINVAL;
            else
                error = del_vif(mtod(m, vifi_t *));
            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

```
static int
ip_mrouter_init(so)
struct socket *so;
{
    if (so->so_type != SOCK_RAW ||
        so->so_proto->pr_protocol != IPPROTO_IGMP)
        return (EOPNOTSUPP);

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

    ip_mrouter = so;

    return (0);
}
```

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.

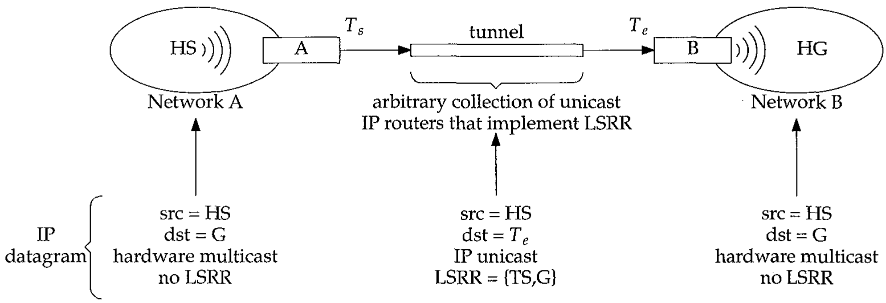

Figure 14.12 A multicast tunnel.

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 14.16 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.

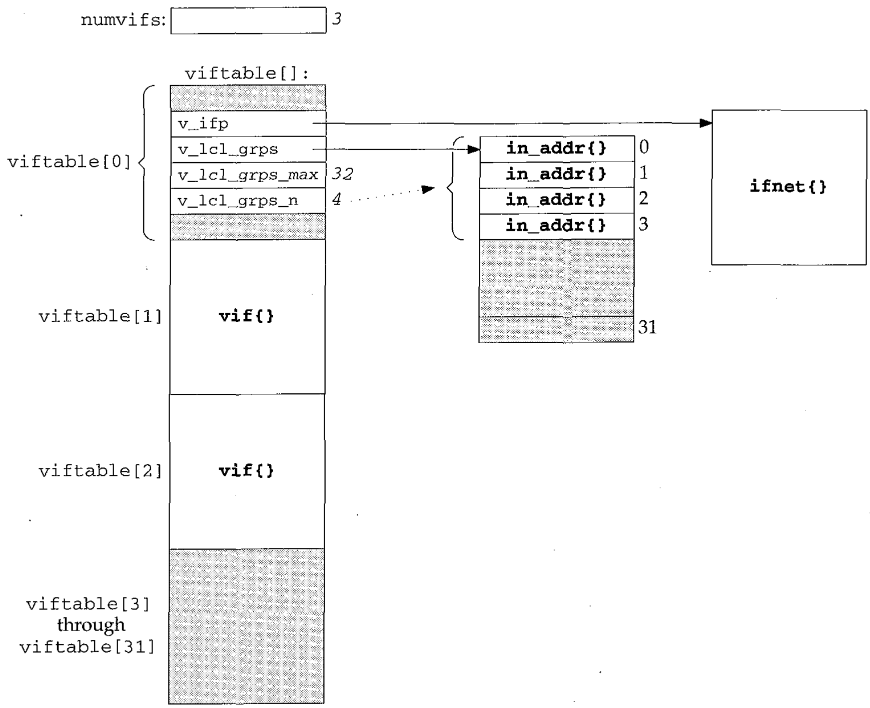

Figure 14.15 `viftable` array.

`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

```
struct vifctl {
    vifi_t  vifc_vifi;          /* the index of the vif to be added */
    u_char  vifc_flags;         /* VIFF_ flags (Figure 14.14) */
    u_char  vifc_threshold;     /* min ttl required to forward on vif */
    struct in_addr vifc_lcl_addr;   /* local interface address */
    struct in_addr vifc_rmt_addr;   /* remote address (tunnels only) */
};
```

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

```
static int
add_vif(vifcp)
struct vifctl *vifcp;
{
    struct vif *vifp = viftable + vifcp->vifc_vifi;
    struct ifaddr *ifa;
    struct ifnet *ifp;
    struct ifreq ifr;
    int     error, s;
    static struct sockaddr_in sin =
    {sizeof(sin), AF_INET};

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

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

    s = splnet();

    if (vifcp->vifc_flags & VIFF_TUNNEL)
        vifp->v_rmt_addr = vifcp->vifc_rmt_addr;
    else {
        /* Make sure the interface supports multicast */
        ifp = ifa->ifa_ifp;
        if ((ifp->if_flags & IFF_MULTICAST) == 0) {
            splx(s);
            return (EOPNOTSUPP);
        }
        /*
         * Enable promiscuous reception of all IP multicasts
         * from the interface.
         */
        satosin(&ifr.ifr_addr)->sin_family = AF_INET;
        satosin(&ifr.ifr_addr)->sin_addr.s_addr = INADDR_ANY;
        error = (*ifp->if_ioctl) (ifp, SIOCADDMULTI, (caddr_t) & ifr);
        if (error) {
            splx(s);
            return (error);
        }
    }
```

```
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.

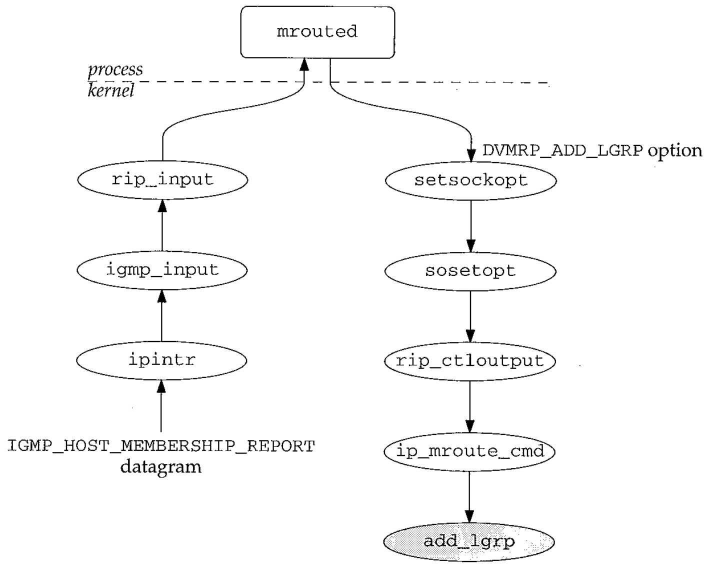

Figure 14.20 IGMP report processing.

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

```
static int
add_lgrp(gcp)
struct lgrplctl *gcp;
{
    struct vif *vifp;
    int     s;

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

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

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

        num = vifp->v_lcl_grps_max;
        if (num <= 0)
            num = 32;           /* initial number */
        else
            num += num;         /* double last number */
        ip = (struct in_addr *) malloc(num * sizeof(*ip),
                                       M_MRTABLE, M_NOWAIT);
        if (ip == NULL) {
            splx(s);
            return (ENOBUFS);
        }
        bzero((caddr_t) ip, num * sizeof(*ip));     /* XXX paranoid */
        bcopy((caddr_t) vifp->v_lcl_grps, (caddr_t) ip,
              vifp->v_lcl_grps_n * sizeof(*ip));

        vifp->v_lcl_grps_max = num;
        if (vifp->v_lcl_grps)
            free(vifp->v_lcl_grps, M_MRTABLE);
        vifp->v_lcl_grps = ip;

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

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

    splx(s);
    return (0);
}
```

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.

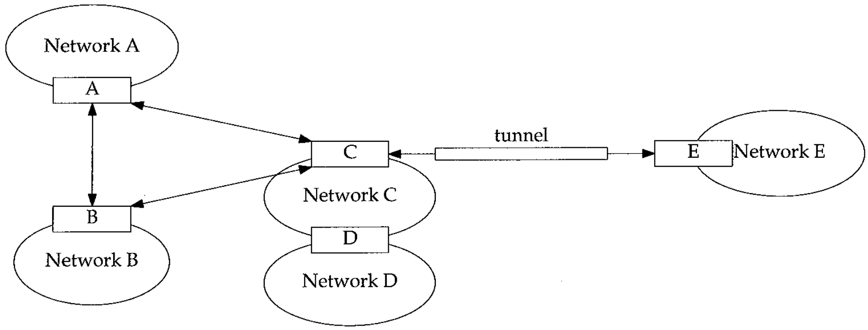

Figure 14.24 Sample multicast network.

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].

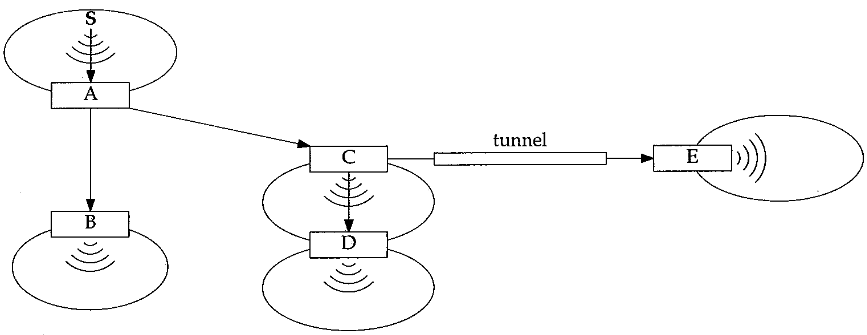

Figure 14.25 Spanning tree for network A.

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.

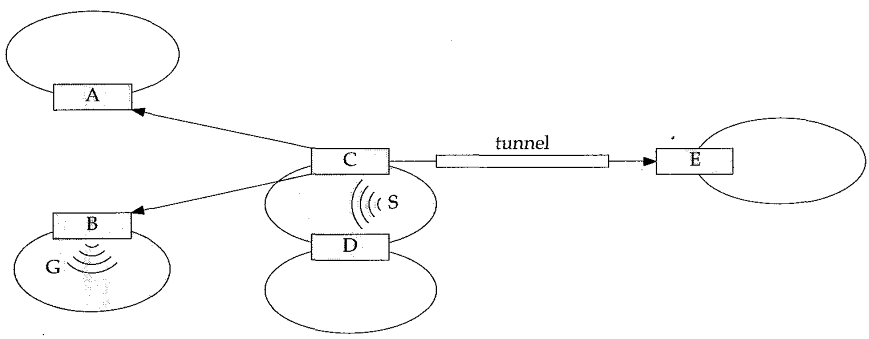

Figure 14.26 TRPB routing for network C.

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.

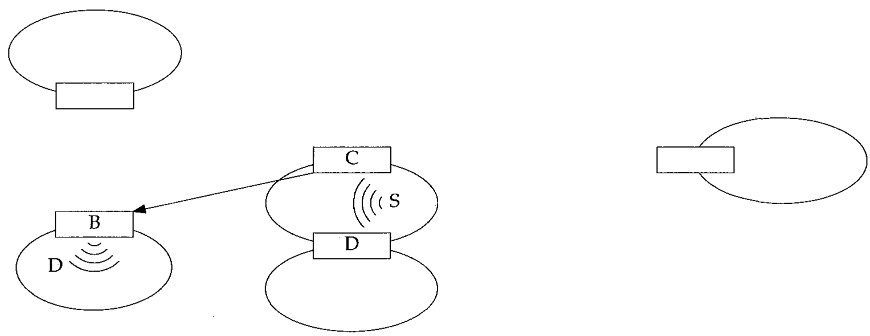

Figure 14.27 RPM routing for network C.

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.

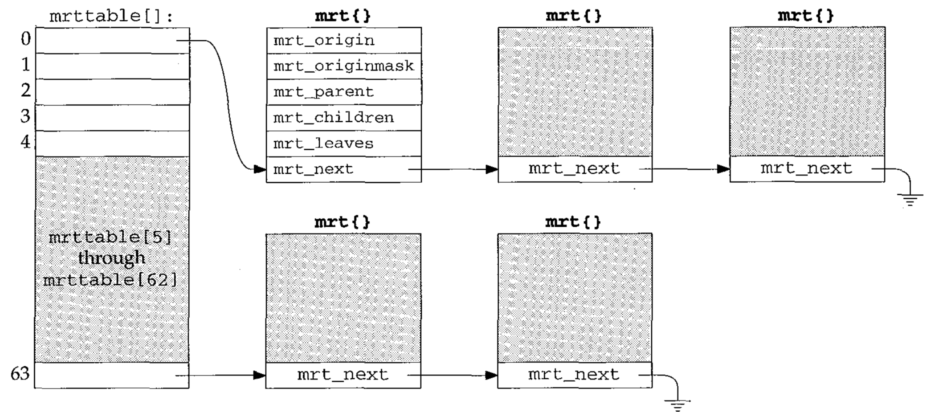

Figure 14.29 Multicast routing table.

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

```
static  u_long
nethash(in)
struct in_addr in;
{
    u_long  n;

    n = in_netof(in);
    while ((n & 0xff) == 0)
        n >>= 8;
    return (MRTHASHMOD(n));
}
```

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

```
static int
add_mrt(mrtcp)
struct mrtctl *mrtcp;
{
    struct mrt *rt;
    u_long  hash;
    int     s;

    if (rt = mrtfind(mrtcp->mrtc_origin)) {
        /* Just update the route */
        s = splnet();
        rt->mrt_parent = mrtcp->mrtc_parent;
        VIFM_COPY(mrtcp->mrtc_children, rt->mrt_children);
        VIFM_COPY(mrtcp->mrtc_leaves, rt->mrt_leaves);
        splx(s);
        return (0);
    }
    s = splnet();

    rt = (struct mrt *) malloc(sizeof(*rt), M_MRTABLE, M_NOWAIT);
    if (rt == NULL) {
        splx(s);
        return (ENOBUFS);
    }
    /*
     * insert new entry at head of hash chain
     */
    rt->mrt_origin = mrtcp->mrtc_origin;
    rt->mrt_originmask = mrtcp->mrtc_originmask;
    rt->mrt_parent = mrtcp->mrtc_parent;
    VIFM_COPY(mrtcp->mrtc_children, rt->mrt_children);
    VIFM_COPY(mrtcp->mrtc_leaves, rt->mrt_leaves);
    /* link into table */
    hash = nethash(mrtcp->mrtc_origin);
    rt->mrt_next = mrttable[hash];
    mrttable[hash] = rt;

    splx(s);
    return (0);
}
```

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.

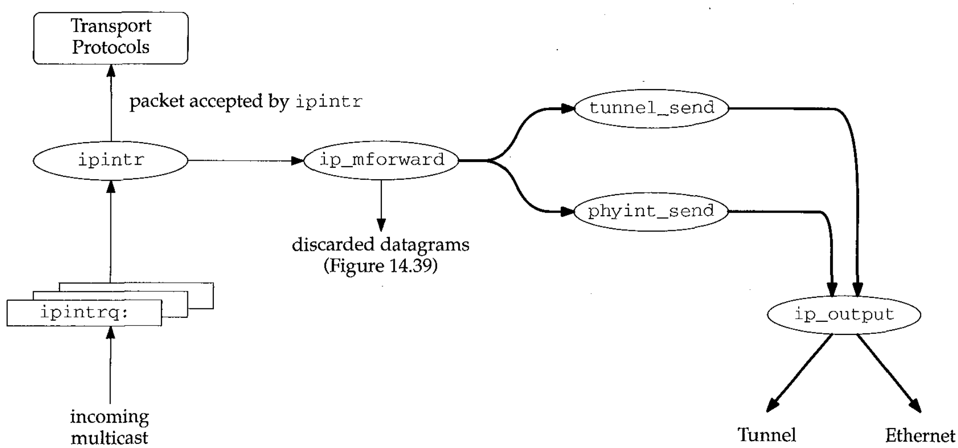

Figure 14.35 Multicast datagram arriving on physical interface.

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.

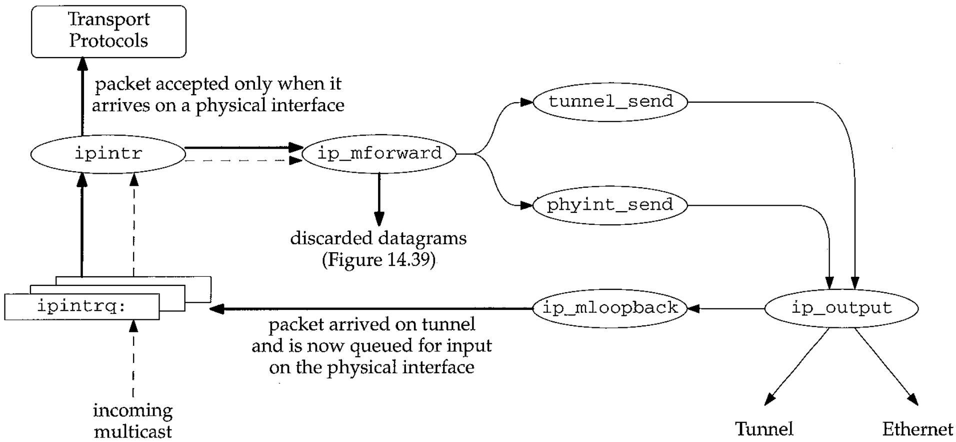

Figure 14.36 Multicast datagram arriving on a multicast tunnel.

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

```
int
ip_mforward(m, ifp)
struct mbuf *m;
struct ifnet *ifp;
{
    struct ip *ip = mtod(m, struct ip *);
    struct mrt *rt;
    struct vif *vifp;
    int     vifi;
    u_char *ipoptions;
    u_long  tunnel_src;

    if (ip->ip_hl < (IP_HDR_LEN + TUNNEL_LEN) >> 2 ||
        (ipoptions = (u_char *) (ip + 1))[1] != IPOPT_LSRR) {
        /* Packet arrived via a physical interface. */
        tunnel_src = 0;
    } else {
        /*
         * Packet arrived through a tunnel.
         * A tunneled packet has a single NOP option and a
         * two-element loose-source-and-record-route (LSRR)
         * option immediately following the fixed-size part of
         * the IP header.  At this point in processing, the IP
         * header should contain the following IP addresses:
         *
         * original source          - in the source address field
         * destination group        - in the destination address field
         * remote tunnel end-point  - in the first  element of LSRR
         * one of this host's addrs - in the second element of LSRR
         *
         * NOTE: RFC-1075 would have the original source and
         * remote tunnel end-point addresses swapped.  However,
         * that could cause delivery of ICMP error messages to
         * innocent applications on intermediate routing
         * hosts!  Therefore, we hereby change the spec.
         */
        /* Verify that the tunnel options are well-formed.  */
        if (ipoptions[0] != IPOPT_NOP ||
            ipoptions[2] != 11 ||    /* LSRR option length   */
            ipoptions[3] != 12 ||    /* LSRR address pointer */
            (tunnel_src = *(u_long *) (&ipoptions[4])) == 0) {
            mrtstat.mrts_bad_tunnel++;
            return (1);
        }
        /* Delete the tunnel options from the packet. */
        ovbcopy((caddr_t) (ipoptions + TUNNEL_LEN), (caddr_t) ipoptions,
                (unsigned) (m->m_len - (IP_HDR_LEN + TUNNEL_LEN)));
        m->m_len -= TUNNEL_LEN;
        ip->ip_len -= TUNNEL_LEN;
        ip->ip_hl -= TUNNEL_LEN >> 2;
    }
```

ip_mroute.c

Figure 14.37 ip_mforward function: tunnel arrival.

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.

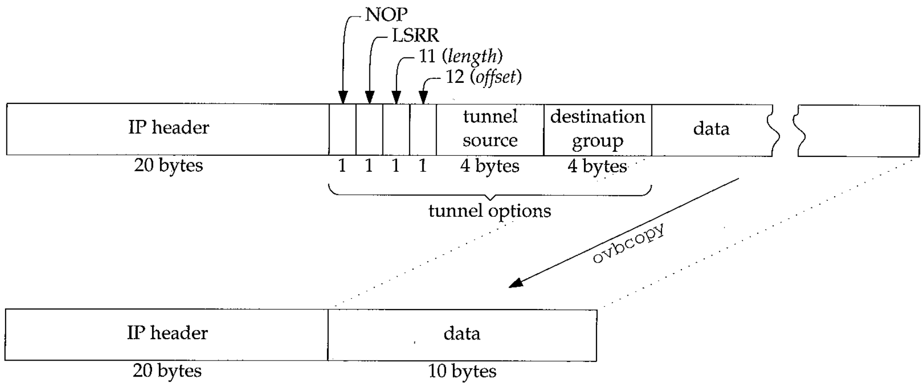

Figure 14.38 Multicast tunnel options.

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.

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.

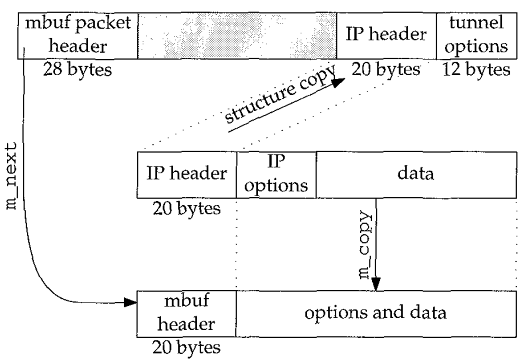

Figure 14.42 Inserting tunnel options.

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

```
int
ip_mrouter_done()
{
    vifi_t  vifi;
    int     i;
    struct ifnet *ifp;
    int     s;
    struct ifreq ifr;
    s = splnet();
    /*
     * For each phyint in use, free its local group list and
     * disable promiscuous reception of all IP multicasts.
     */
    for (vifi = 0; vifi < numvifs; vifi++) {
        if (viftable[vifi].v_lcl_addr.s_addr != 0 &&
            !(viftable[vifi].v_flags & VIFF_TUNNEL)) {
            if (viftable[vifi].v_lcl_grps)
                free(viftable[vifi].v_lcl_grps, M_MRTABLE);
            satosin(&ifr.ifr_addr)->sin_family = AF_INET;
            satosin(&ifr.ifr_addr)->sin_addr.s_addr = INADDR_ANY;
            ifp = viftable[vifi].v_ifp;
            (*ifp->if_ioctl) (ifp, SIOCDELMULTI, (caddr_t) & ifr);
        }
    }
    bzero((caddr_t) viftable, sizeof(viftable));
    numvifs = 0;
    /*
     * Free any multicast route entries.
     */
    for (i = 0; i < MRTHASHSIZ; i++)
        if (mrttable[i])
            free(mrttable[i], M_MRTABLE);
    bzero((caddr_t) mrttable, sizeof(mrttable));
    cached_mrt = NULL;
    ip_mrouter = NULL;
    splx(s);
    return (0);
}
```

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.

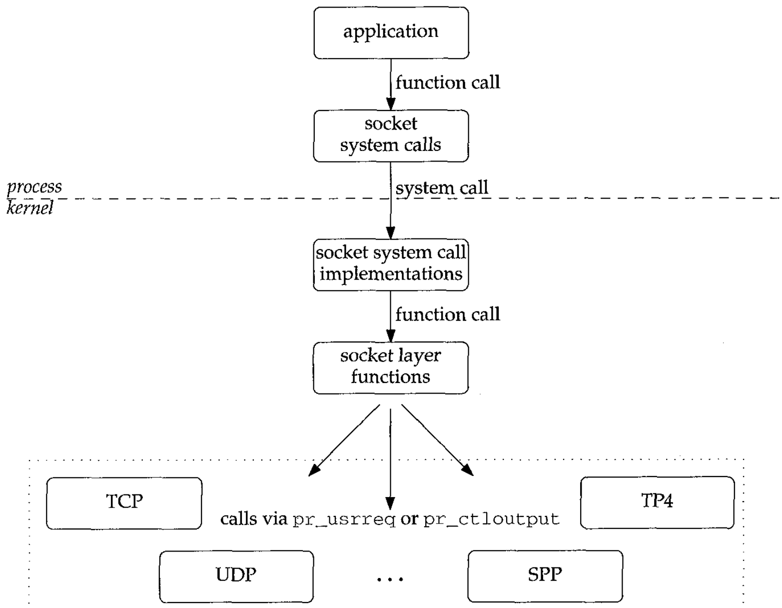

Figure 15.1 The socket layer converts generic requests to specific protocol operations.

`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	system call implementation
kern/uipc_socket.c	socket-layer functions

Figure 15.2 Files discussed in this chapter.

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

```
struct socket {
    short   so_type;            /* generic type, Figure 7.8 */
    short   so_options;         /* from socket call, Figure 15.4 */
    short   so_linger;          /* time to linger while closing */
    short   so_state;           /* internal state flags, Figure 15.6 */
    caddr_t so_pcb;             /* protocol control block */
    struct protosw *so_proto;   /* protocol handle */
/*
 * Variables for connection queueing.
 * Socket where accepts occur is so_head in all subsidiary sockets.
 * If so_head is 0, socket is not related to an accept.
 * For head socket so_q0 queues partially completed connections,
 * while so_q is a queue of connections ready to be accepted.
 * If a connection is aborted and it has so_head set, then
 * it has to be pulled out of either so_q0 or so_q.
 * We allow connections to queue up based on current queue lengths
 * and limit on number of queued connections for this socket.
 */
    struct socket *so_head;     /* back pointer to accept socket */
    struct socket *so_q0;       /* queue of partial connections */
    struct socket *so_q;        /* queue of incoming connections */
    short   so_q0len;           /* partials on so_q0 */
    short   so_qlen;            /* number of connections on so_q */
    short   so_qlimit;          /* max number queued connections */
    short   so_timeo;           /* connection timeout */
    u_short so_error;           /* error affecting connection */
    pid_t   so_pgid;            /* pgid for signals */
    u_long  so_oobmark;         /* chars to oob mark */
/*
 * Variables for socket buffering.
 */
    struct sockbuf {
        u_long  sb_cc;          /* actual chars in buffer */
        u_long  sb_hiwat;       /* max actual char count */
        u_long  sb_mbcnt;       /* chars of mbufs used */
        u_long  sb_mbmax;       /* max chars of mbufs to use */
        long    sb_lowat;       /* low water mark */
        struct mbuf *sb_mb;     /* the mbuf chain */
        struct selinfo sb_sel;  /* process selecting read/write */
        short   sb_flags;       /* Figure 16.5 */
        short   sb_timeo;       /* timeout for read/write */
    } so_rcv, so_snd;
    caddr_t so_tpcb;            /* Wisc. protocol control block XXX */
    void    (*so_upcall) (struct socket * so, caddr_t arg, int waitf);
    caddr_t so_upcallarg;       /* Arg for above */
};
```

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 closing 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 example, 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 network. 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 following reasons:

- a connection request has completed,
- a disconnect request has been initiated,
- a disconnect request has 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

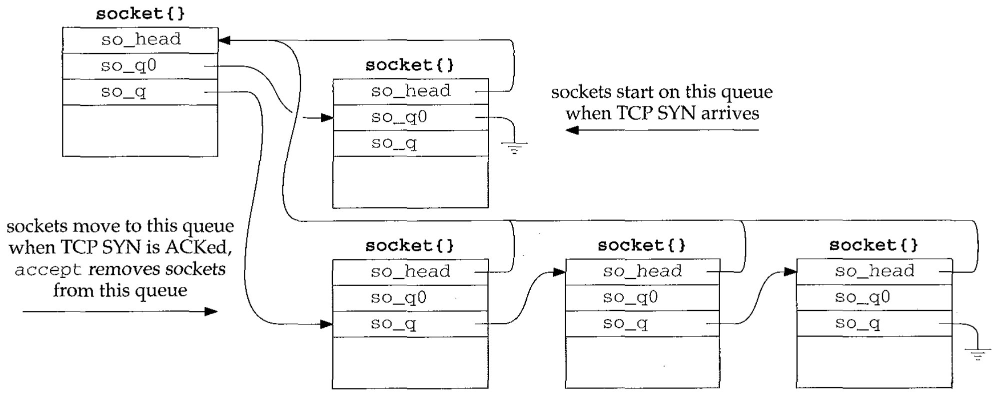

Figure 15.8 Socket connection queues.

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.

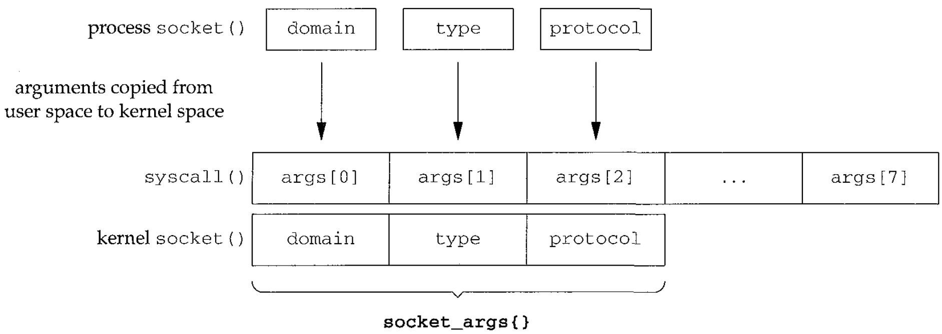

Figure 15.10 socket argument processing.

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].

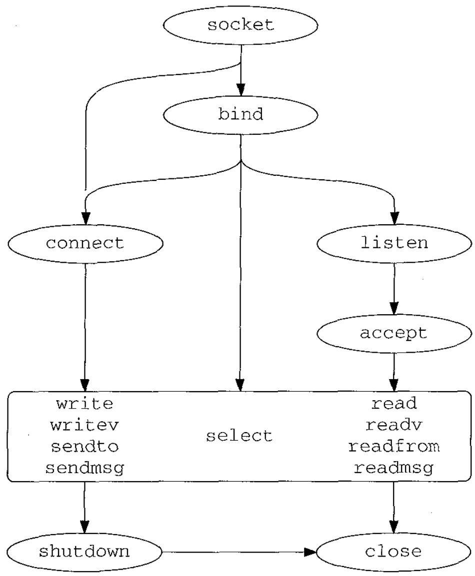

Figure 15.12 Network system call flowchart.

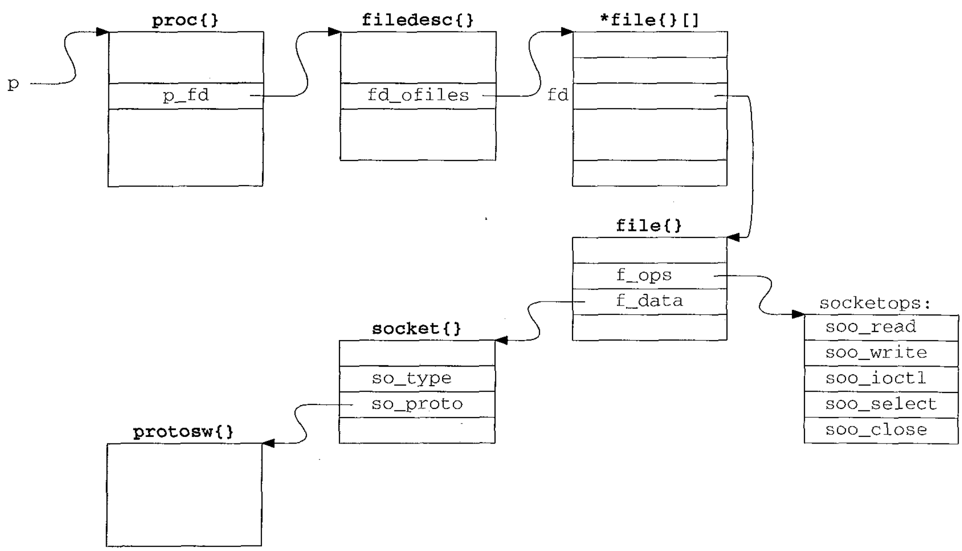

Figure 15.13 Process, file, and socket structures.

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

```
struct socket_args {
    int     domain;
    int     type;
    int     protocol;
};

socket(p, uap, retval)
struct proc *p;
struct socket_args *uap;
int    *retval;
{
    struct filedesc *fdp = p->p_fd;
    struct socket *so;
    struct file *fp;
    int     fd, error;

    if (error = falloc(p, &fp, &fd))
        return (error);
    fp->f_flag = FREAD | FWRITE;
    fp->f_type = DTYPE_SOCKET;
    fp->f_ops = &socketops;
    if (error = socreate(uap->domain, &so, uap->type, uap->protocol)) {
        fdp->fd_ofiles[fd] = 0;
        ffree(fp);
    } else {
        fp->f_data = (caddr_t) so;
        *retval = fd;
    }
    return (error);
}
```

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 nonsuperuser 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

```
getsock(fdp, fdes, fpp)
struct filedesc *fdp;
int     fdes;
struct file **fpp;
{
    struct file *fp;

    if ((unsigned) fdes >= fdp->fd_nfiles ||
        (fp = fdp->fd_ofiles[fdes]) == NULL)
        return (EBADF);
    if (fp->f_type != DTYPE_SOCKET)
        return (ENOTSOCK);
    *fpp = fp;
    return (0);
}
```

uipc_syscalls.c

Figure 15.19 getsock function.

uipc_syscalls.c

```
sockargs(mp, buf, buflen, type)
struct mbuf **mp;
caddr_t buf;
int     buflen, type;
{
    struct sockaddr *sa;
    struct mbuf *m;
    int     error;

    if ((u_int) buflen > MLEN) {
        return (EINVAL);
    }
    m = m_get(M_WAIT, type);
    if (m == NULL)
        return (ENOBUFS);
    m->m_len = buflen;
    error = copyin(buf, mtod(m, caddr_t), (u_int) buflen);
    if (error)
        (void) m_free(m);
    else {
        *mp = m;
        if (type == MT_SONAME) {
            sa = mtod(m, struct sockaddr *);
            sa->sa_len = buflen;
        }
    }
    return (error);
}
```

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

```
struct bind_args {
    int     s;
    caddr_t name;
    int     namelen;
};

bind(p, uap, retval)
struct proc *p;
struct bind_args *uap;
int     *retval;
{
    struct file *fp;
    struct mbuf *nam;
    int     error;

    if (error = getsock(p->p_fd, uap->s, &fp))
        return (error);
    if (error = sockargs(&nam, uap->name, uap->namelen, MT_SONAME))
        return (error);
    error = sobind((struct socket *) fp->f_data, nam);
    m_freem(nam);
    return (error);
}
```

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

```
sobind(so, nam)
struct socket *so;
struct mbuf *nam;
{
    int     s = splnet();
    int     error;

    error =
        (*so->so_proto->pr_usrreq) (so, PRU_BIND,
                                    (struct mbuf *) 0, nam, (struct mbuf *) 0);
    splx(s);
    return (error);
}
```

uipc_socket.c

Figure 15.22 `sobind` function.

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

```
solisten(so, backlog)
struct socket *so;
int     backlog;
{
    int     s = splnet(), error;

    error =
        (*so->so_proto->pr_usrreq) (so, PRU_LISTEN,
                   (struct mbuf *) 0, (struct mbuf *) 0, (struct mbuf *) 0);
    if (error) {
        splx(s);
        return (error);
    }
    if (so->so_q == 0)
        so->so_options |= SO_ACCEPTCONN;
    if (backlog < 0)
        backlog = 0;
    so->so_qlimit = min(backlog, SOMAXCONN);
    splx(s);
    return (0);
}
```

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

```
struct accept_args {
    int     s;
    caddr_t name;
    int     *anamelen;
};

accept(p, uap, retval)
struct proc *p;
struct accept_args *uap;
int     *retval;
{
    struct file *fp;
    struct mbuf *nam;
    int     namelen, error, s;
    struct socket *so;

    if (uap->name && (error = copyin((caddr_t) uap->anamelen,
                                     (caddr_t) & namelen, sizeof(namelen))))
        return (error);
    if (error = getsock(p->p_fd, uap->s, &fp))
        return (error);
    s = splnet();
    so = (struct socket *) fp->f_data;
    if ((so->so_options & SO_ACCEPTCONN) == 0) {
        splx(s);
        return (EINVAL);
    }
    if ((so->so_state & SS_NBIO) && so->so_qlen == 0) {
        splx(s);
        return (EWOULDBLOCK);
    }
    while (so->so_qlen == 0 && so->so_error == 0) {
        if (so->so_state & SS_CANTRCVMORE) {
            so->so_error = ECONNABORTED;
            break;
        }
        if (error = tsleep((caddr_t) & so->so_timeo, PSOCK | PCATCH,
                           netcon, 0)) {
            splx(s);
            return (error);
        }
    }
    if (so->so_error) {
        error = so->so_error;
        so->so_error = 0;
        splx(s);
        return (error);
    }
    if (error = falloc(p, &fp, retval)) {
        splx(s);
        return (error);
    }
```

```
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

```
soaccept(so, nam)
struct socket *so;
struct mbuf *nam;
{
    int     s = splnet();
    int     error;

    if ((so->so_state & SS_NOFDREF) == 0)
        panic("soaccept: !NOFDREF");
    so->so_state &= ~SS_NOFDREF;
    error = (*so->so_proto->pr_usrreq) (so, PRU_ACCEPT,
                                 (struct mbuf *) 0, nam, (struct mbuf *) 0);
    splx(s);
    return (error);
}
```

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.

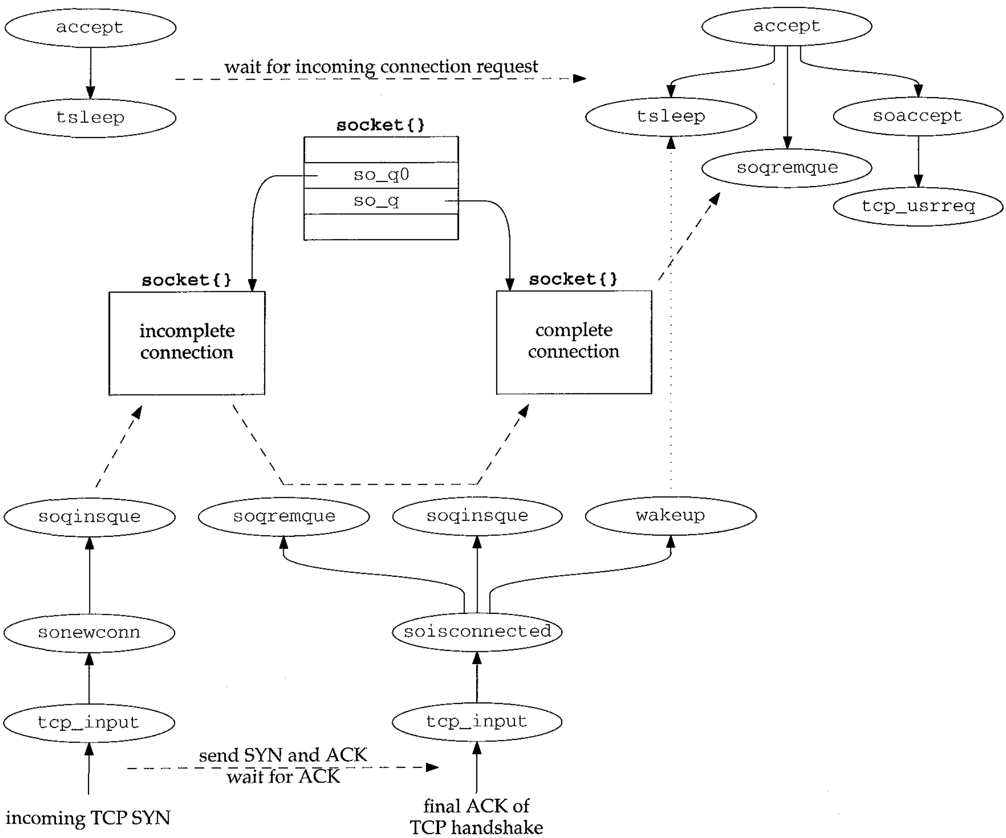

Figure 15.28 Incoming TCP connection processing.

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:

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

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 protocols, 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

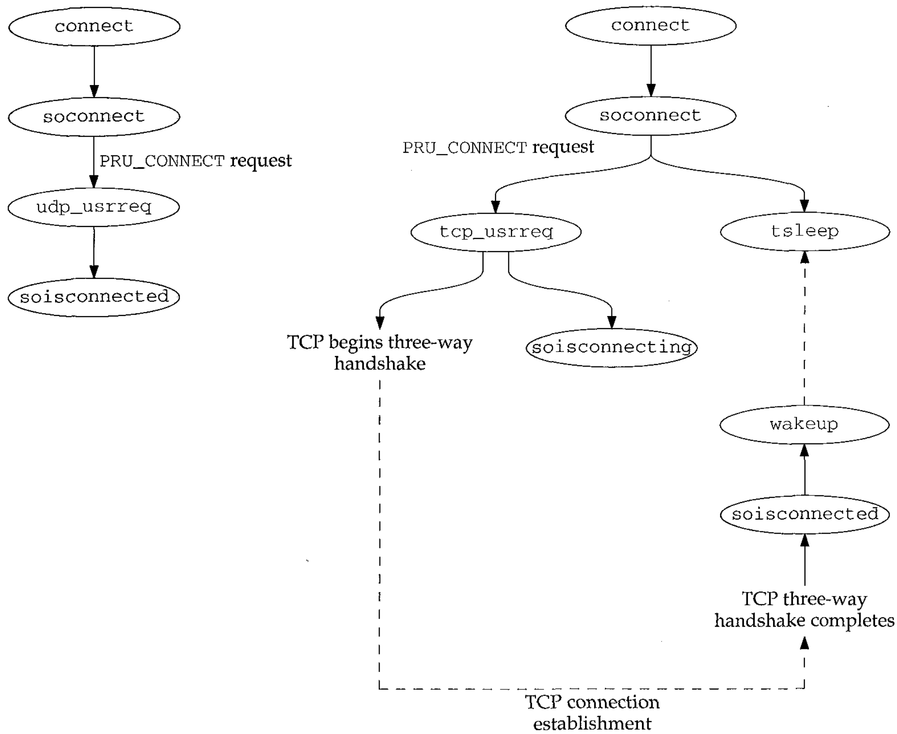

Figure 15.31 connect processing.

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

```
struct connect_args {
    int     s;
    caddr_t name;
    int     namelen;
};

connect(p, uap, retval)
struct proc *p;
struct connect_args *uap;
int    *retval;
{
    struct file *fp;
    struct socket *so;
    struct mbuf *nam;
    int     error, s;

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

    error = soconnect(so, nam);
    if (error)
        goto bad;
    if ((so->so_state & SS_NBIO) && (so->so_state & SS_ISCONNECTING)) {
        m_freem(nam);
        return (EINPROGRESS);
    }
    s = splnet();
    while ((so->so_state & SS_ISCONNECTING) && so->so_error == 0)
        if (error = tsleep((caddr_t) & so->so_timeo, PSOCK | PCATCH,
                           netcon, 0))
            break;
    if (error == 0) {
        error = so->so_error;
        so->so_error = 0;
    }
    splx(s);
  bad:
    so->so_state &= ~SS_ISCONNECTING;
    m_freem(nam);
    if (error == ERESTART)
        error = EINTR;
    return (error);
}
```

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

```
soconnect(so, nam)
struct socket *so;
struct mbuf *nam;
{
    int     s;
    int     error;

    if (so->so_options & SO_ACCEPTCONN)
        return (EOPNOTSUPP);
    s = splnet();
    /*
     * If protocol is connection-based, can only connect once.
     * Otherwise, if connected, try to disconnect first.
     * This allows user to disconnect by connecting to, e.g.,
     * a null address.
     */
    if (so->so_state & (SS_ISCONNECTED | SS_ISCONNECTING) &&
        ((so->so_proto->pr_flags & PR_CONNREQUIRED) ||
         (error = sodisconnect(so))))
        error = EISCONN;
    else
        error = (*so->so_proto->pr_usrreq) (so, PRU_CONNECT,
                                 (struct mbuf *) 0, nam, (struct mbuf *) 0);
    splx(s);
    return (error);
}
```

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

```
struct shutdown_args {
    int     s;
    int     how;
};

shutdown(p, uap, retval)
struct proc *p;
struct shutdown_args *uap;
int     *retval;
{
    struct file *fp;
    int     error;

    if (error = getsock(p->p_fd, uap->s, &fp))
        return (error);
    return (soshutdown((struct socket *) fp->f_data, uap->how));
}
```

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

```
soclose(so)
struct socket *so;
{
    int     s = splnet();       /* conservative */
    int     error = 0;

    if (so->so_options & SO_ACCEPTCONN) {
        while (so->so_q0)
            (void) soabort(so->so_q0);
        while (so->so_q)
            (void) soabort(so->so_q);
    }
    if (so->so_pcb == 0)
        goto discard;
    if (so->so_state & SS_ISCONNECTED) {
        if ((so->so_state & SS_ISDISCONNECTING) == 0) {
            error = sodisconnect(so);
            if (error)
                goto drop;
        }
        if (so->so_options & SO_LINGER) {
            if ((so->so_state & SS_ISDISCONNECTING) &&
                (so->so_state & SS_NBIO))
                goto drop;
            while (so->so_state & SS_ISCONNECTED)
                if (error = tsleep((caddr_t) & so->so_timeo,
                                   PSOCK | PCATCH, netcls, so->so_linger))
                    break;
        }
    }
  drop:
    if (so->so_pcb) {
        int     error2 =
        (*so->so_proto->pr_usrreq) (so, PRU_DETACH,
                  (struct mbuf *) 0, (struct mbuf *) 0, (struct mbuf *) 0);
        if (error == 0)
            error = error2;
    }
  discard:
    if (so->so_state & SS_NOFDREF)
        panic("soclose: NOFDREF");
    so->so_state |= SS_NOFDREF;
    sofree(so);
    splx(s);
    return (error);
}
```

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), the socket's queues should be empty. If they are not empty, there is a bug in the socket code and the kernel panics. If they are empty, 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.

File	Description
sys/socket.h	structures and macro for sockets API
sys/socketvar.h	socket structure and macros
sys/uio.h	uio structure definition
kern/uipc_syscalls.c	socket system calls
kern/uipc_socket.c	socket layer processing
kern/sys_generic.c	select system call
kern/sys_socket.c	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 flow control algorithms are described in Sections 16.4 and 16.8. 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 × sb_hiwat	40 × (1024 + 16)	1	2 × sb_hiwat
TCP	8 × 1024	2048	2 × sb_hiwat	8 × 1024	1	2 × sb_hiwat
raw IP ICMP IGMP	8 × 1024	2048 (ignored)	2 × sb_hiwat	8 × 1024	1	2 × 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	a process has locked the socket buffer
SB_WANT	a process is waiting to lock the buffer
SB_WAIT	a process is waiting for data (receive) or space (send) in this buffer
SB_SEL	one or more processes are selecting on this buffer
SB_ASYNC	generate asynchronous I/O signal for this buffer
SB_NOINTR	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 *sb* 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 *sb* 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*. `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. `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, intlen);`
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.

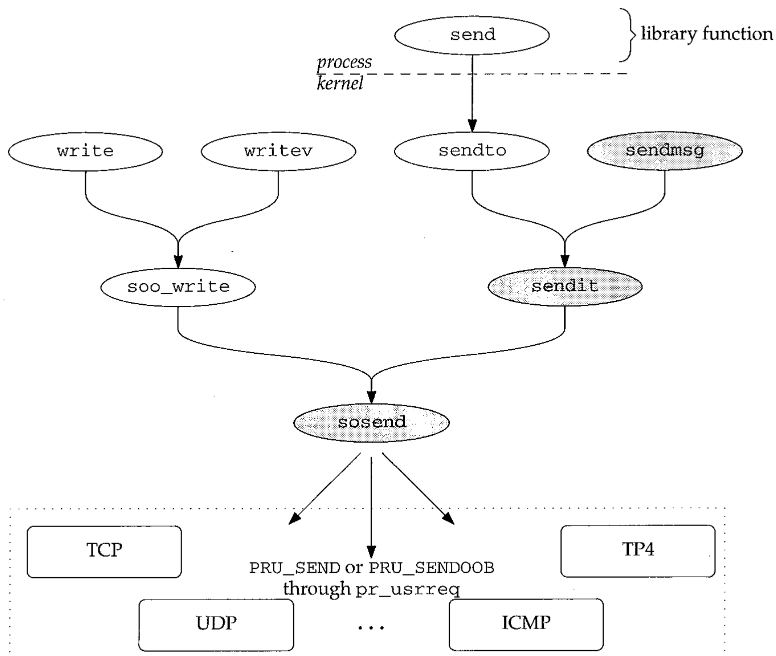

Figure 16.8 All socket output is handled by sosend.

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.

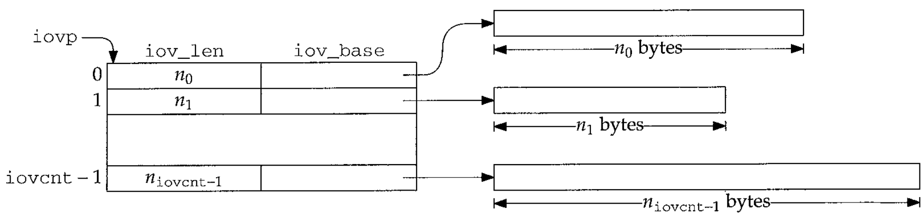

Figure 16.11 `iovec` arguments to `writev`.

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.

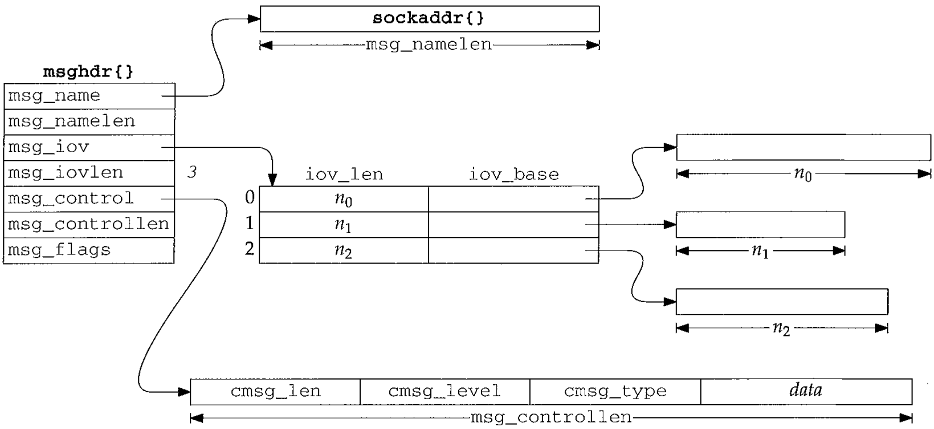

Figure 16.15 msghdr structure for sendmsg system call.

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.

307–319 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

320–334 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_USERISPACE*	*UIO_READ*	scatter *n* bytes from a kernel buffer *cp* to process buffers
UIO_USERSPACE *UIO_USERISPACE*	*UIO_WRITE*	gather *n* bytes from process buffers into the kernel buffer *cp*
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.

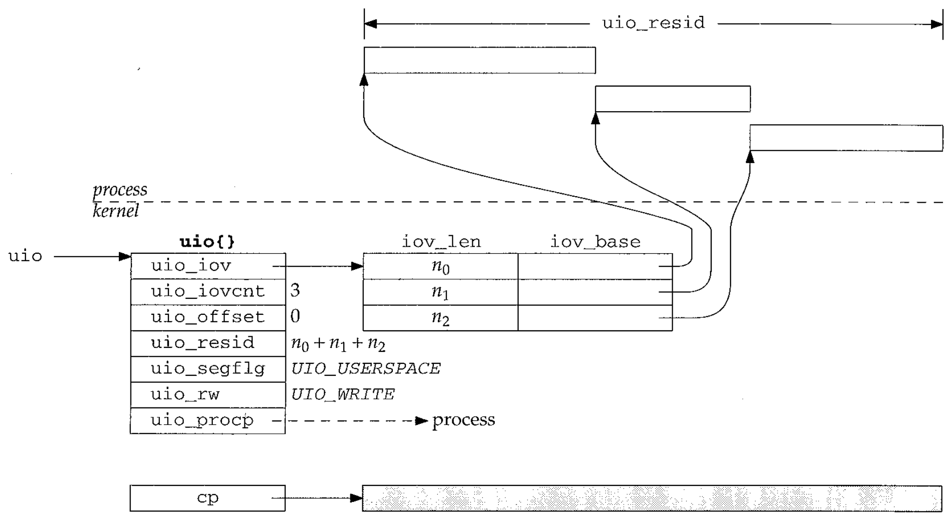

Figure 16.19 uiomove: before.

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 $< n_0 + n_1$).

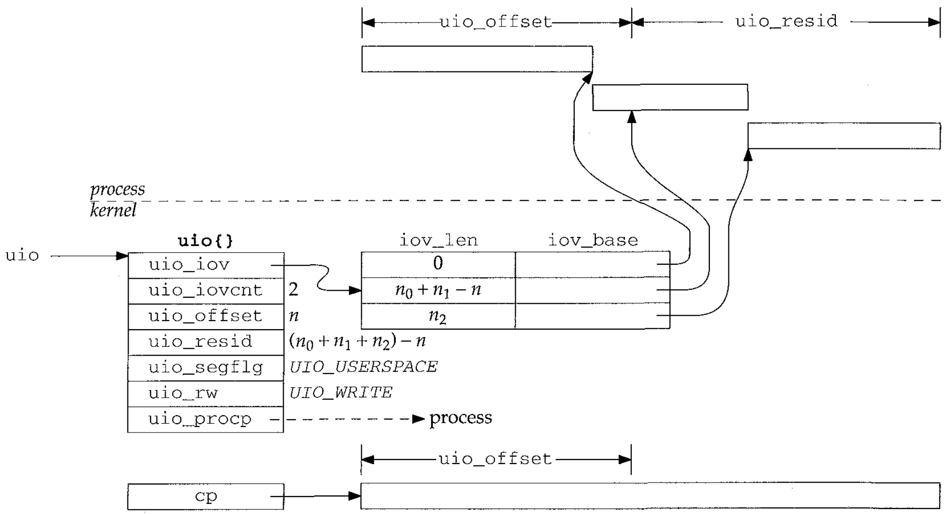

Figure 16.20 uiomove: after.

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.

uipc_syscalls.c

```
sendit(p, s, mp, flags, retsize)
struct proc *p;
int     s;
struct msghdr *mp;
int     flags, *retsize;
{
    struct file *fp;
    struct uio auio;
    struct iovec *iov;
    int     i;
    struct mbuf *to, *control;
    int     len, error;

    if (error = getsock(p->p_fd, s, &fp))
        return (error);
    auio.uio_iov = mp->msg_iov;
    auio.uio_iovcnt = mp->msg_iovlen;
    auio.uio_segflg = UIO_USERSPACE;
    auio.uio_rw = UIO_WRITE;
    auio.uio_procp = p;
    auio.uio_offset = 0;        /* XXX */
    auio.uio_resid = 0;
    iov = mp->msg_iov;
    for (i = 0; i < mp->msg_iovlen; i++, iov++) {
        if (iov->iov_len < 0)
            return (EINVAL);
        if ((auio.uio_resid += iov->iov_len) < 0)
            return (EINVAL);
    }
    if (mp->msg_name) {
        if (error = sockargs(&to, mp->msg_name, mp->msg_namelen,
                             MT_SONAME))
            return (error);
    } else
        to = 0;
    if (mp->msg_control) {
        if (mp->msg_controllen < sizeof(struct cmsghdr)
        ) {
            error = EINVAL;
            goto bad;
        }
        if (error = sockargs(&control, mp->msg_control,
                             mp->msg_controllen, MT_CONTROL))
            goto bad;
    } else
        control = 0;
    len = auio.uio_resid;
    if (error = sosend((struct socket *) fp->f_data, to, &auio,
                       (struct mbuf *) 0, control, flags)) {
        if (auio.uio_resid != len && (error == ERESTART ||
                                      error == EINTR || error == EWOULDBLOCK))
            error = 0;
        if (error == EPIPE)
            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 calculated if sosend does not accept all the data. The socket, destination address, uio structure, 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 condition, 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 calculated and saved in *retsize. Since sendit returns 0, syscall (Section 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 Figure 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 associated with the socket according to the semantics supported by the protocol and the buffer 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 ≤ sb_cc ≤ 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.

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

```
sosend(so, addr, uio, top, control, flags)
struct socket *so;
struct mbuf *addr;
struct uio *uio;
struct mbuf *top;
struct mbuf *control;
int     flags;
{

                   /* initialization (Figure 16.23) */

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

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

        do {
            if (uio == NULL) {
                /*
                 * Data is prepackaged in "top".
                 */
                resid = 0;
                if (flags & MSG_EOR)
                    top->m_flags |= M_EOR;
            } else
                do {

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

                } while (space > 0 && atomic);

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

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

  release:
    sbunlock(&so->so_snd);
  out:
    if (top)
        m_freem(top);
    if (control)
        m_freem(control);
    return (error);
}
```

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

395–414 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

```
    struct proc *p = curproc;    /* XXX */
    struct mbuf **mp;
    struct mbuf *m;
    long    space, len, resid;
    int     clen = 0, error, s, dontroute, mlen;
    int     atomic = sosendallatonce(so) || top;

    if (uio)
        resid = uio->uio_resid;
    else
        resid = top->m_pkthdr.len;
    /*
     * In theory resid should be unsigned.
     * However, space must be signed, as it might be less than 0
     * if we over-committed, and we must use a signed comparison
     * of space and resid.  On the other hand, a negative resid
     * causes us to loop sending 0-length segments to the protocol.
     */
    if (resid < 0)
        return (EINVAL);
    dontroute =
        (flags & MSG_DONTROUTE) && (so->so_options & SO_DONTROUTE) == 0 &&
        (so->so_proto->pr_flags & PR_ATOMIC);
    p->p_stats->p_ru.ru_msgsnd++;
    if (control)
        clen = control->m_len;
#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 received 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

```
		do {
			if (top == 0) {
				MGETHDR(m, M_WAIT, MT_DATA);
				mlen = MHLEN;
				m->m_pkthdr.len = 0;
				m->m_pkthdr.rcvif = (struct ifnet *) 0;
			} else {
				MGET(m, M_WAIT, MT_DATA);
				mlen = MLEN;
			}

			if (resid >= MINCLSIZE && space >= MCLBYTES) {
				MCLGET(m, M_WAIT);
				if ((m->m_flags & M_EXT) == 0)
					goto nopages;
				mlen = MCLBYTES;
				if (atomic && top == 0) {
					len = min(MCLBYTES - max_hdr, resid);
					m->m_data += max_hdr;
				} else
					len = min(MCLBYTES, resid);
				space -= MCLBYTES;
			} else {
			  nopages:
				len = min(min(mlen, resid), space);
				space -= len;
				/*
				 * For datagram protocols, leave room
				 * for protocol headers in first mbuf.
				 */
				if (atomic && top == 0 && len < mlen)
					MH_ALIGN(m, len);
			}

			error = uiomove(mtod(m, caddr_t), (int) len, uio);
			resid = uio->uio_resid;
			m->m_len = len;
			*mp = m;
			top->m_pkthdr.len += len;
			if (error)
				goto release;
			mp = &m->m_next;
			if (resid <= 0) {
				if (flags & MSG_EOR)
					top->m_flags |= M_EOR;
				break;
			}
		} 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 to 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 a 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.

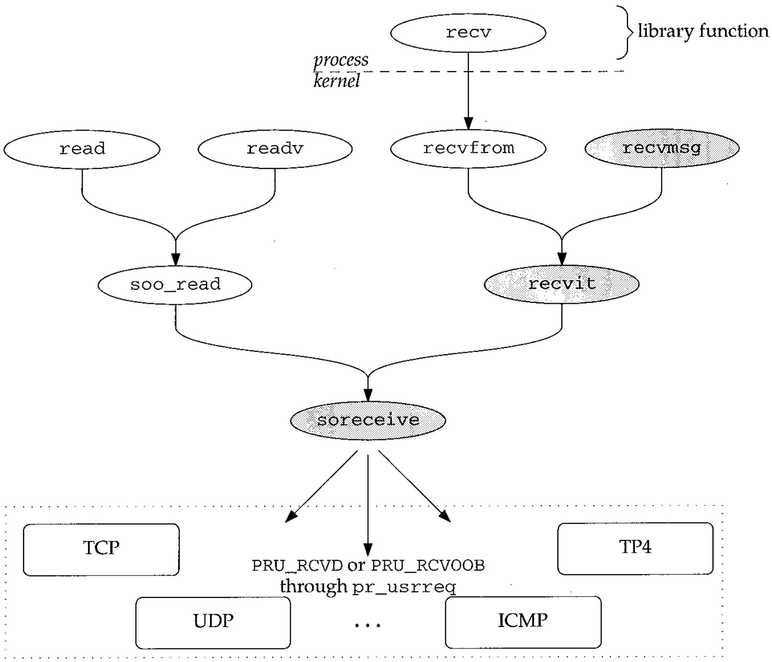

Figure 16.28 All socket input is processed by soreceive.

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 recv*xxx* 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

```
	if (error = soreceive((struct socket *) fp->f_data, &from, &auio,
	  (struct mbuf **) 0, mp->msg_control ? &control : (struct mbuf **) 0,
	                      &mp->msg_flags)) {
	    if (auio.uio_resid != len && (error == ERESTART ||
	                                error == EINTR || error == EWOULDBLOCK))
	        error = 0;
	}
	if (error)
	    goto out;
	*retsize = len - auio.uio_resid;
	if (mp->msg_name) {
	    len = mp->msg_namelen;
	    if (len <= 0 || from == 0)
	        len = 0;
	    else {
	        if (len > from->m_len)
	            len = from->m_len;
	        /* else if len < from->m_len ??? */
	        if (error = copyout(mtod(from, caddr_t),
	                            (caddr_t) mp->msg_name, (unsigned) len))
	            goto out;
	    }
	    mp->msg_namelen = len;
	    if (namelenp &&
	        (error = copyout((caddr_t) & len, namelenp, sizeof(int)))) {
	        goto out;
	    }
	}
	if (mp->msg_control) {
	    len = mp->msg_controllen;
	    if (len <= 0 || control == 0)
	        len = 0;
	    else {
	        if (len >= control->m_len)
	            len = control->m_len;
	        else
	            mp->msg_flags |= MSG_CTRUNC;
	        error = copyout((caddr_t) mtod(control, caddr_t),
	                        (caddr_t) mp->msg_control, (unsigned) len);
	    }
	    mp->msg_controllen = len;
	}
  out:
	if (from)
	    m_freem(from);
	if (control)
	    m_freem(control);
	return (error);
}
```

uipc_syscalls.c

Figure 16.31 `recvit` function: return results.

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 recv*xxx* 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 send*xxx* 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 recv*xxx* 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

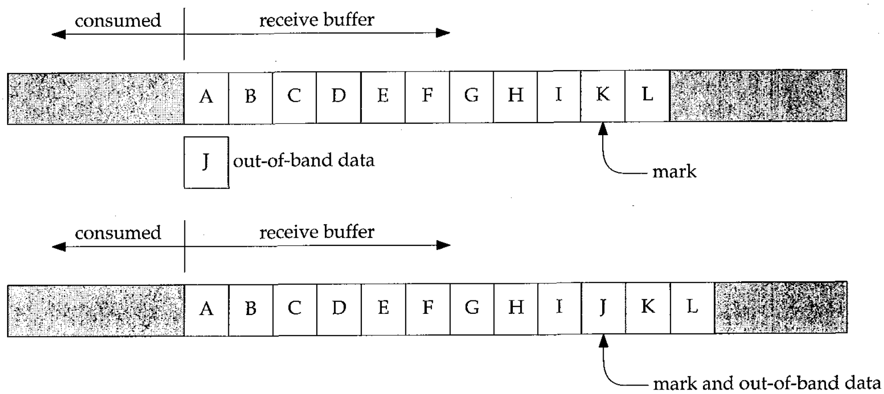

Figure 16.34 Receiving out-of-band data.

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.

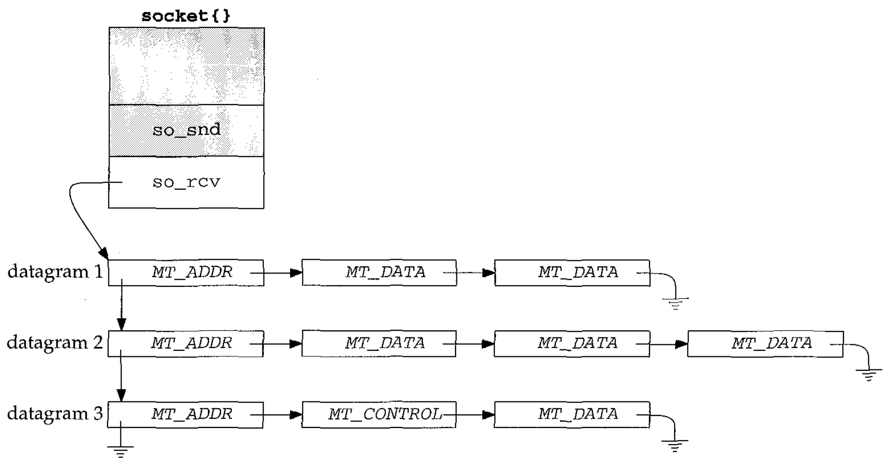

Figure 16.35 UDP receive buffer consisting of three datagrams.

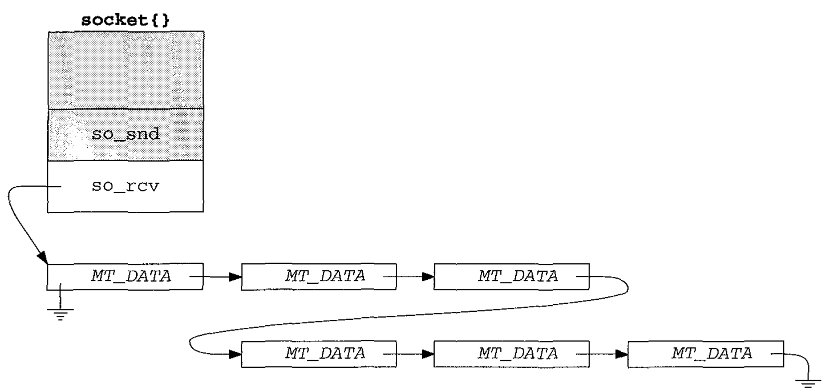

Figure 16.36 so_rcv buffer for TCP.

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.

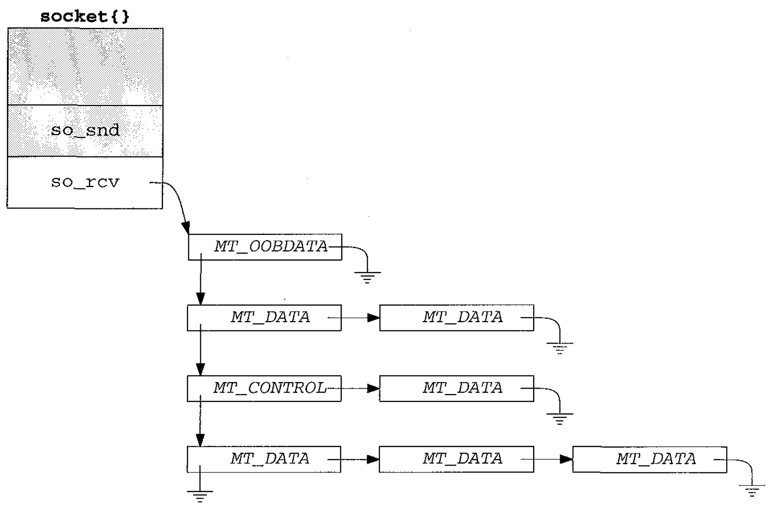

Figure 16.37 so_rcv buffer with control and OOB data.

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.

uipc_socket.c

```
soreceive(so, paddr, uio, mp0, controlp, flagsp)
struct socket *so;
struct mbuf **paddr;
struct uio *uio;
struct mbuf **mp0;
struct mbuf **controlp;
int     *flagsp;
{
    struct mbuf *m, **mp;
    int     flags, len, error, s, offset;
    struct protosw *pr = so->so_proto;
    struct mbuf *nextrecord;
    int     moff, type;
    int     orig_resid = uio->uio_resid;

    mp = mp0;
    if (paddr)
        *paddr = 0;
    if (controlp)
        *controlp = 0;
    if (flagsp)
        flags = *flagsp & ~MSG_EOR;
    else
        flags = 0;

                        /* MSG_OOB processing and */
                   /* implicit connection confirmation */

  restart:
    if (error = sblock(&so->so_rcv, SBLOCKWAIT(flags)))
        return (error);
    s = splnet();
    m = so->so_rcv.sb_mb;

                /* if necessary, wait for data to arrive */

  dontblock:
    if (uio->uio_procp)
        uio->uio_procp->p_stats->p_ru.ru_msgrcv++;
    nextrecord = m->m_nextpkt;

               /* process address and control information */

    if (m) {
        if ((flags & MSG_PEEK) == 0)
            m->m_nextpkt = nextrecord;
        type = m->m_type;
        if (type == MT_OOBDATA)
            flags |= MSG_OOB;
    }
```

```
                                /* process data */

    }                           /* while more data and more space to fill */

                                  /* cleanup */

  release:
    sbunlock(&so->so_rcv);
    splx(s);
    return (error);
}
```

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:

- 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

```
        if (so->so_error) {
            if (m)
                goto dontblock;
            error = so->so_error;
            if ((flags & MSG_PEEK) == 0)
                so->so_error = 0;
            goto release;
        }
        if (so->so_state & SS_CANTRCVMORE) {
            if (m)
                goto dontblock;
            else
                goto release;
        }
        for (; m; m = m->m_next)
            if (m->m_type == MT_OOBDATA || (m->m_flags & M_EOR)) {
                m = so->so_rcv.sb_mb;
                goto dontblock;
            }
        if ((so->so_state & (SS_ISCONNECTED | SS_ISCONNECTING)) == 0 &&
            (so->so_proto->pr_flags & PR_CONNREQUIRED)) {
            error = ENOTCONN;
            goto release;
        }
        if (uio->uio_resid == 0)
            goto release;
        if ((so->so_state & SS_NBIO) || (flags & MSG_DONTWAIT)) {
            error = EWOULDBLOCK;
            goto release;
        }
        sbunlock(&so->so_rcv);
        error = sbwait(&so->so_rcv);
        splx(s);
        if (error)
            return (error);
        goto restart;
    }
```

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 a 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

```
    while (m && m->m_type == MT_CONTROL && error == 0) {
        if (flags & MSG_PEEK) {
            if (controlp)
                *controlp = m_copy(m, 0, m->m_len);
            m = m->m_next;
        } else {
            sbfree(&so->so_rcv, m);
            if (controlp) {
                if (pr->pr_domain->dom_externalize &&
                    mtod(m, struct cmsghdr *)->cmsg_type ==
                    SCM_RIGHTS)
                            error = (*pr->pr_domain->dom_externalize) (m);
                *controlp = m;
                so->so_rcv.sb_mb = m->m_next;
                m->m_next = 0;
                m = so->so_rcv.sb_mb;
            } else {
                MFREE(m, so->so_rcv.sb_mb);
                m = so->so_rcv.sb_mb;
            }
        }
        if (controlp) {
            orig_resid = 0;
            controlp = &(*controlp)->m_next;
        }
    }
```

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

uipc_socket.c

```
    if (m) {
        if ((flags & MSG_PEEK) == 0)
            m->m_nextpkt = nextrecord;
        type = m->m_type;
        if (type == MT_OOBDATA)
            flags |= MSG_OOB;
    }
```

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

```
	moff = 0;
	offset = 0;
	while (m && uio->uio_resid > 0 && error == 0) {
		if (m->m_type == MT_OOBDATA) {
			if (type != MT_OOBDATA)
				break;
		} else if (type == MT_OOBDATA)
			break;
		so->so_state &= ~SS_RCVATMARK;
		len = uio->uio_resid;
		if (so->so_oobmark && len > so->so_oobmark - offset)
			len = so->so_oobmark - offset;
		if (len > m->m_len - moff)
			len = m->m_len - moff;
		/*
		 * If mp is set, just pass back the mbufs.
		 * Otherwise copy them out via the uio, then free.
		 * Sockbuf must be consistent here (points to current mbuf,
		 * it points to next record) when we drop priority;
		 * we must note any additions to the sockbuf when we
		 * block interrupts again.
		 */
		if (mp == 0) {
			splx(s);
			error = uiomove(mtod(m, caddr_t) + moff, (int) len, uio);
			s = splnet();
		} else
			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

```
        if (len == m->m_len - moff) {
            if (m->m_flags & M_EOR)
                flags |= MSG_EOR;
            if (flags & MSG_PEEK) {
                m = m->m_next;
                moff = 0;
            } else {
                nextrecord = m->m_nextpkt;
                sbfree(&so->so_rcv, m);
                if (mp) {
                    *mp = m;
                    mp = &m->m_next;
                    so->so_rcv.sb_mb = m = m->m_next;
                    *mp = (struct mbuf *) 0;
                } else {
                    MFREE(m, so->so_rcv.sb_mb);
                    m = so->so_rcv.sb_mb;
                }
                if (m)
                    m->m_nextpkt = nextrecord;
            }
        } else {
            if (flags & MSG_PEEK)
                moff += len;
            else {
                if (mp)
                    *mp = m_copym(m, 0, len, M_WAIT);
                m->m_data += len;
                m->m_len -= len;
                so->so_rcv.sb_cc -= len;
            }
        }
```

uipc_socket.c

Figure 16.48 soreceive function: update buffer.

uipc_socket.c

```
        if (so->so_oobmark) {
            if ((flags & MSG_PEEK) == 0) {
                so->so_oobmark -= len;
                if (so->so_oobmark == 0) {
                    so->so_state |= SS_RCVATMARK;
                    break;
                }
            } else {
                offset += len;
                if (offset == so->so_oobmark)
                    break;
            }
        }
        if (flags & MSG_EOR)
            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:

$$\texttt{ni} = \texttt{howmany}(65, 32) \times 4 = 3 \times 4 = 12$$

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

```
struct select_args {
    u_int   nd;
    fd_set *in, *ou, *ex;
    struct timeval *tv;
};

select(p, uap, retval)
struct proc *p;
struct select_args *uap;
int     *retval;
{
    fd_set  ibits[3], obits[3];
    struct timeval atv;
    int     s, ncoll, error = 0, timo;
    u_int   ni;

    bzero((caddr_t) ibits, sizeof(ibits));
    bzero((caddr_t) obits, sizeof(obits));
    if (uap->nd > FD_SETSIZE)
        return (EINVAL);
    if (uap->nd > p->p_fd->fd_nfiles)
        uap->nd = p->p_fd->fd_nfiles;   /* forgiving; slightly wrong */
    ni = howmany(uap->nd, NFDBITS) * sizeof(fd_mask);

#define getbits(name, x) \
    if (uap->name && \
        (error = copyin((caddr_t)uap->name, (caddr_t)&ibits[x], ni))) \
        goto done;
    getbits(in, 0);
    getbits(ou, 1);
    getbits(ex, 2);
#undef  getbits

    if (uap->tv) {
        error = copyin((caddr_t) uap->tv, (caddr_t) & atv,
                       sizeof(atv));
        if (error)
            goto done;
        if (itimerfix(&atv)) {
            error = EINVAL;
            goto done;
        }
        s = splclock();
        timevaladd(&atv, (struct timeval *) &time);
        timo = hzto(&atv);
        /*
         * Avoid inadvertently sleeping forever.
         */
        if (timo == 0)
            timo = 1;
        splx(s);
    } else
        timo = 0;
```

sys_generic.c

Figure 16.53 `select` function: initialization.

sys_generic.c

```
retry:
    ncoll = nselcoll;
    p->p_flag |= P_SELECT;
    error = selscan(p, ibits, obits, uap->nd, retval);
    if (error || *retval)
        goto done;
    s = splhigh();
    /* this should be timercmp(&time, &atv, >=) */
    if (uap->tv && (time.tv_sec > atv.tv_sec ||
              time.tv_sec == atv.tv_sec && time.tv_usec >= atv.tv_usec)) {
        splx(s);
        goto done;
    }
    if ((p->p_flag & P_SELECT) == 0 || nselcoll != ncoll) {
        splx(s);
        goto retry;
    }
    p->p_flag &= ~P_SELECT;
    error = tsleep((caddr_t) & selwait, PSOCK | PCATCH, "select", timo);
    splx(s);
    if (error == 0)
        goto retry;
  done:
    p->p_flag &= ~P_SELECT;
    /* select is not restarted after signals... */
    if (error == ERESTART)
        error = EINTR;
    if (error == EWOULDBLOCK)
        error = 0;
#define putbits(name, x) \
    if (uap->name && \
        (error2 = copyout((caddr_t)&obits[x], (caddr_t)uap->name, ni))) \
        error = error2;
    if (error == 0) {
        int     error2;

        putbits(in, 0);
        putbits(ou, 1);
        putbits(ex, 2);
#undef putbits
    }
    return (error);
}
```

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

```
selscan(p, ibits, obits, nfd, retval)
struct proc *p;
fd_set *ibits, *obits;
int     nfd, *retval;
{
    struct filedesc *fdp = p->p_fd;
    int     msk, i, j, fd;
    fd_mask bits;
    struct file *fp;
    int     n = 0;
    static int flag[3] =
    {FREAD, FWRITE, 0};

    for (msk = 0; msk < 3; msk++) {
        for (i = 0; i < nfd; i += NFDBITS) {
            bits = ibits[msk].fds_bits[i / NFDBITS];
            while ((j = ffs(bits)) && (fd = i + --j) < nfd) {
                bits &= ~(1 << j);
                fp = fdp->fd_ofiles[fd];
                if (fp == NULL)
                    return (EBADF);
                if ((*fp->f_ops->fo_select) (fp, flag[msk], p)) {
                    FD_SET(fd, &obits[msk]);
                    n++;
                }
            }
        }
    }
    *retval = n;
    return (0);
}
```

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, `sl_pid` is the process ID of the waiting process. When additional processes call select on the same buffer, `SI_COLL` is set in `sl_flags`. This is called a *collision*. This is the only flag currently defined for `sl_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.

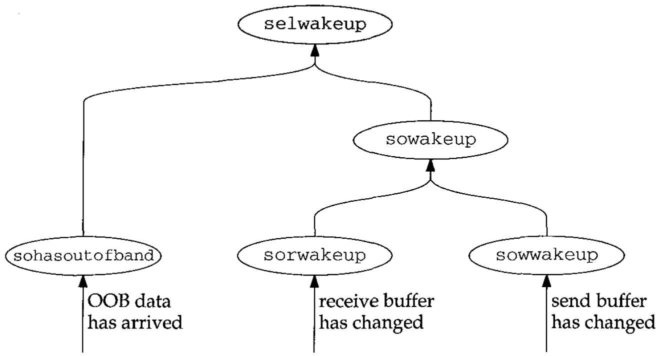

Figure 16.59 `selwakeup` processing.

sys_generic.c

```
void
selwakeup(sip)
struct selinfo *sip;
{
    struct proc *p;
    int     s;

    if (sip->si_pid == 0)
        return;
    if (sip->si_flags & SI_COLL) {
        nselcoll++;
        sip->si_flags &= ~SI_COLL;
        wakeup((caddr_t) & selwait);
    }
    p = pfind(sip->si_pid);
    sip->si_pid = 0;
    if (p != NULL) {
        s = splhigh();
        if (p->p_wchan == (caddr_t) & selwait) {
            if (p->p_stat == SSLEEP)
                setrunnable(p);
            else
                unsleep(p);
        } else if (p->p_flag & P_SELECT)
            p->p_flag &= ~P_SELECT;
        splx(s);
    }
}
```

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.

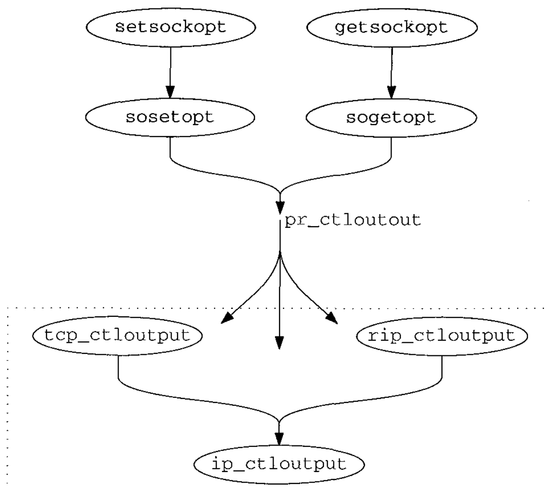

Figure 17.1 setsockopt and getsockopt system calls.

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 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

```
struct setsockopt_args {
    int     s;
    int     level;
    int     name;
    caddr_t val;
    int     valsize;
};

setsockopt(p, uap, retval)
struct proc *p;
struct setsockopt_args *uap;
int     *retval;
{
    struct file *fp;
    struct mbuf *m = NULL;
    int     error;

    if (error = getsock(p->p_fd, uap->s, &fp))
        return (error);
    if (uap->valsize > MLEN)
        return (EINVAL);
    if (uap->val) {
        m = m_get(M_WAIT, MT_SOOPTS);
        if (m == NULL)
            return (ENOBUFS);
        if (error = copyin(uap->val, mtod(m, caddr_t),
                           (u_int) uap->valsize)) {
            (void) m_free(m);
            return (error);
        }
        m->m_len = uap->valsize;
    }
    return (sosetopt((struct socket *) fp->f_data, uap->level,
                     uap->name, m));
}
```

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 TCP	ip_ctloutput tcp_ctloutput	Section 8.8 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

$$\texttt{tick} = \frac{1,000,000}{\texttt{hz}} \text{ and } \texttt{SHRT\_MAX} = 32767$$

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 16.11 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 third 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 `var`. 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.

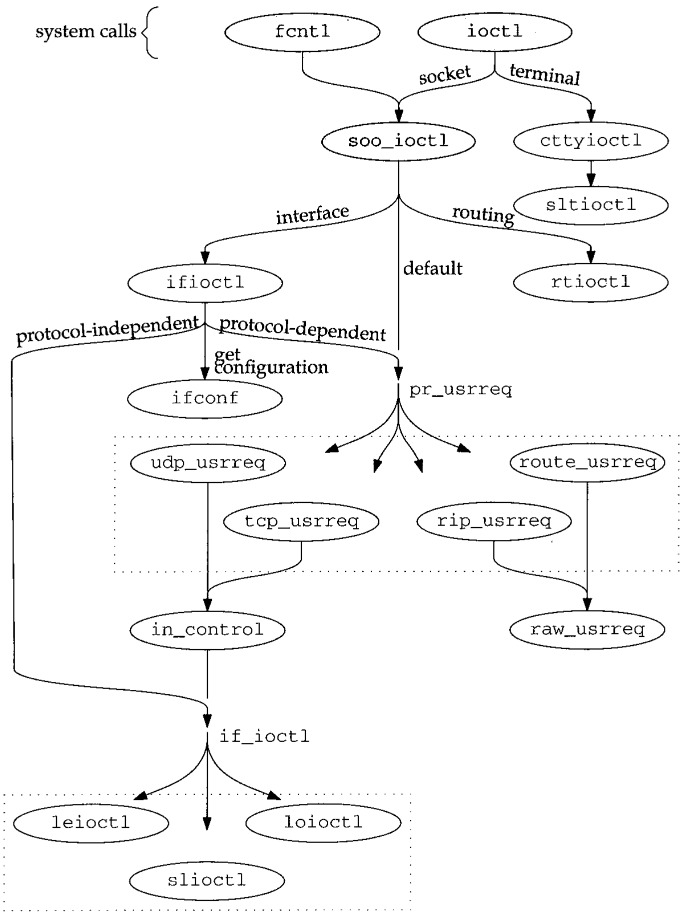

Figure 17.15 fcntl and ioctl functions.

The prototypes for ioctl and fcntlare:

```
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

```
struct fcntl_args {
    int     fd;
    int     cmd;
    int     arg;
};
/* ARGSUSED */
fcntl(p, uap, retval)
struct proc *p;
struct fcntl_args *uap;
int     *retval;
{
    struct filedesc *fdp = p->p_fd;
    struct file *fp;
    struct vnode *vp;
    int     i, tmp, error, flg = F_POSIX;
    struct flock fl;
    u_int   newmin;

    if ((unsigned) uap->fd >= fdp->fd_nfiles ||
        (fp = fdp->fd_ofiles[uap->fd]) == NULL)
        return (EBADF);
    switch (uap->cmd) {

                                    /* command processing */

    default:
        return (EINVAL);
    }
    /* NOTREACHED */
}
```

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

```
    case F_GETFL:
        *retval = OFLAGS(fp->f_flag);
        return (0);

    case F_SETFL:
        fp->f_flag &= ~FCNTLFLAGS;
        fp->f_flag |= FFLAGS(uap->arg) & FCNTLFLAGS;

        tmp = fp->f_flag & FNONBLOCK;
        error = (*fp->f_ops->fo_ioctl) (fp, FIONBIO, (caddr_t) & tmp, p);
        if (error)
            return (error);

        tmp = fp->f_flag & FASYNC;
        error = (*fp->f_ops->fo_ioctl) (fp, FIOASYNC, (caddr_t) & tmp, p);
        if (!error)
            return (0);

        fp->f_flag &= ~FNONBLOCK;
        tmp = 0;
        (void) (*fp->f_ops->fo_ioctl) (fp, FIONBIO, (caddr_t) & tmp, p);
        return (error);

    case F_GETOWN:
        if (fp->f_type == DTYPE_SOCKET) {
            *retval = ((struct socket *) fp->f_data)->so_pgid;
            return (0);
        }
        error = (*fp->f_ops->fo_ioctl)
            (fp, (int) TIOCGPGRP, (caddr_t) retval, p);
        *retval = -*retval;
        return (error);

    case F_SETOWN:
        if (fp->f_type == DTYPE_SOCKET) {
            ((struct socket *) fp->f_data)->so_pgid = uap->arg;
            return (0);
        }
        if (uap->arg <= 0) {
            uap->arg = -uap->arg;
        } else {
            struct proc *p1 = pfind(uap->arg);
            if (p1 == 0)
                return (ESRCH);
            uap->arg = p1->p_pgrp->pg_id;
        }
        return ((*fp->f_ops->fo_ioctl)
                (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.

186–194 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 non-blocking 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.

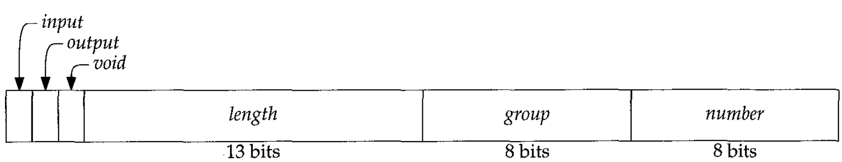

Figure 17.19 The structure of an ioctl command.

sys_socket.c

```
soo_ioctl(fp, cmd, data, p)
struct file *fp;
int     cmd;
caddr_t data;
struct proc *p;
{
    struct socket *so = (struct socket *) fp->f_data;

    switch (cmd) {
    case FIONBIO:
        if (*(int *) data)
            so->so_state |= SS_NBIO;
        else
            so->so_state &= ~SS_NBIO;
        return (0);

    case FIOASYNC:
        if (*(int *) data) {
            so->so_state |= SS_ASYNC;
            so->so_rcv.sb_flags |= SB_ASYNC;
            so->so_snd.sb_flags |= SB_ASYNC;
        } else {
            so->so_state &= ~SS_ASYNC;
            so->so_rcv.sb_flags &= ~SB_ASYNC;
            so->so_snd.sb_flags &= ~SB_ASYNC;
        }
        return (0);

    case FIONREAD:
        *(int *) data = so->so_rcv.sb_cc;
        return (0);

    case SIOCSPGRP:
        so->so_pgid = *(int *) data;
        return (0);

    case SIOCGPGRP:
        *(int *) data = so->so_pgid;
        return (0);

    case SIOCATMARK:
        *(int *) data = (so->so_state & SS_RCVATMARK) != 0;
        return (0);
    }
    /*
     * Interface/routing/protocol specific ioctls:
     * interface and routing ioctls should have a
     * different entry since a socket's unnecessary
     */
    if (IOCGROUP(cmd) == 'i')
        return (ifioctl(so, cmd, data, p));
    if (IOCGROUP(cmd) == 'r')
        return (rtioctl(cmd, data, p));
    return ((*so->so_proto->pr_usrreq) (so, PRU_CONTROL,
            (struct mbuf *) cmd, (struct mbuf *) data, (struct mbuf *) 0));
}
```

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 described 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

```
struct getsockname_args {
    int     fdes;
    caddr_t asa;
    int    *alen;
};

getsockname(p, uap, retval)
struct proc *p;
struct getsockname_args *uap;
int    *retval;
{
    struct file *fp;
    struct socket *so;
    struct mbuf *m;
    int     len, error;

    if (error = getsock(p->p_fd, uap->fdes, &fp))
        return (error);
    if (error = copyin((caddr_t) uap->alen, (caddr_t) & len, sizeof(len)))
        return (error);
    so = (struct socket *) fp->f_data;
    m = m_getclr(M_WAIT, MT_SONAME);
    if (m == NULL)
        return (ENOBUFS);
    if (error = (*so->so_proto->pr_usrreq) (so, PRU_SOCKADDR, 0, m, 0))
        goto bad;
    if (len > m->m_len)
        len = m->m_len;
    error = copyout(mtod(m, caddr_t), (caddr_t) uap->asa, (u_int) len);
    if (error == 0)
        error = copyout((caddr_t) & len, (caddr_t) uap->alen,
                        sizeof(len));
  bad:
    m_freem(m);
    return (error);
}
```

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, a 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.

18.2 Routing Table Structure

Before looking at the internal structure of the Net/3 routing table, we need to understand 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.

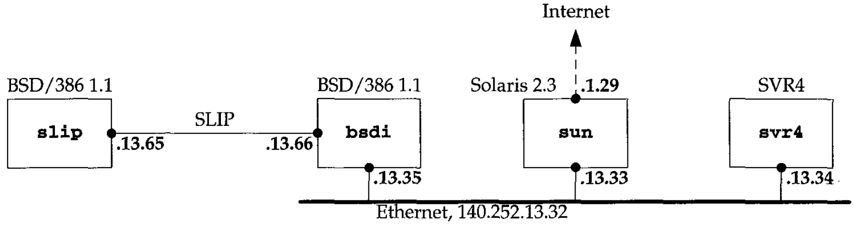

Figure 18.1 Subnet used for routing table example.

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 easier 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].

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.

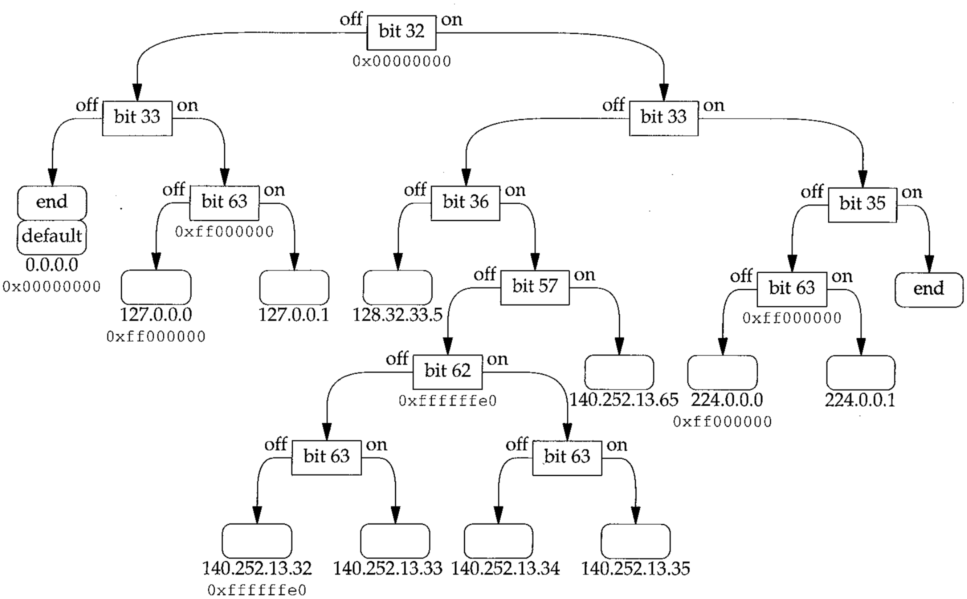

Figure 18.4 Net/3 routing table corresponding to Figure 18.2.

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.

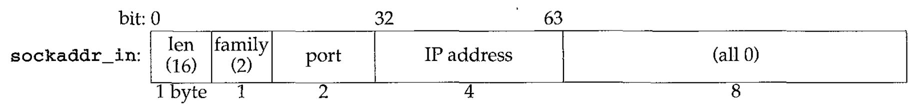

Figure 18.5 Bit offsets in Internet socket address structure.

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**	**3**333 **6**789	4444 0123	4444 4567	4455 8901	5555 2345	5**5**55 6**7**89	66**66** 01**23**	
	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 mask 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 started 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.

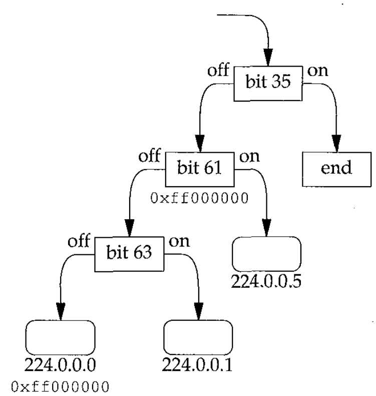

Figure 18.7 Modification of Figure 18.6 after inserting entry for 224.0.0.5.

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.

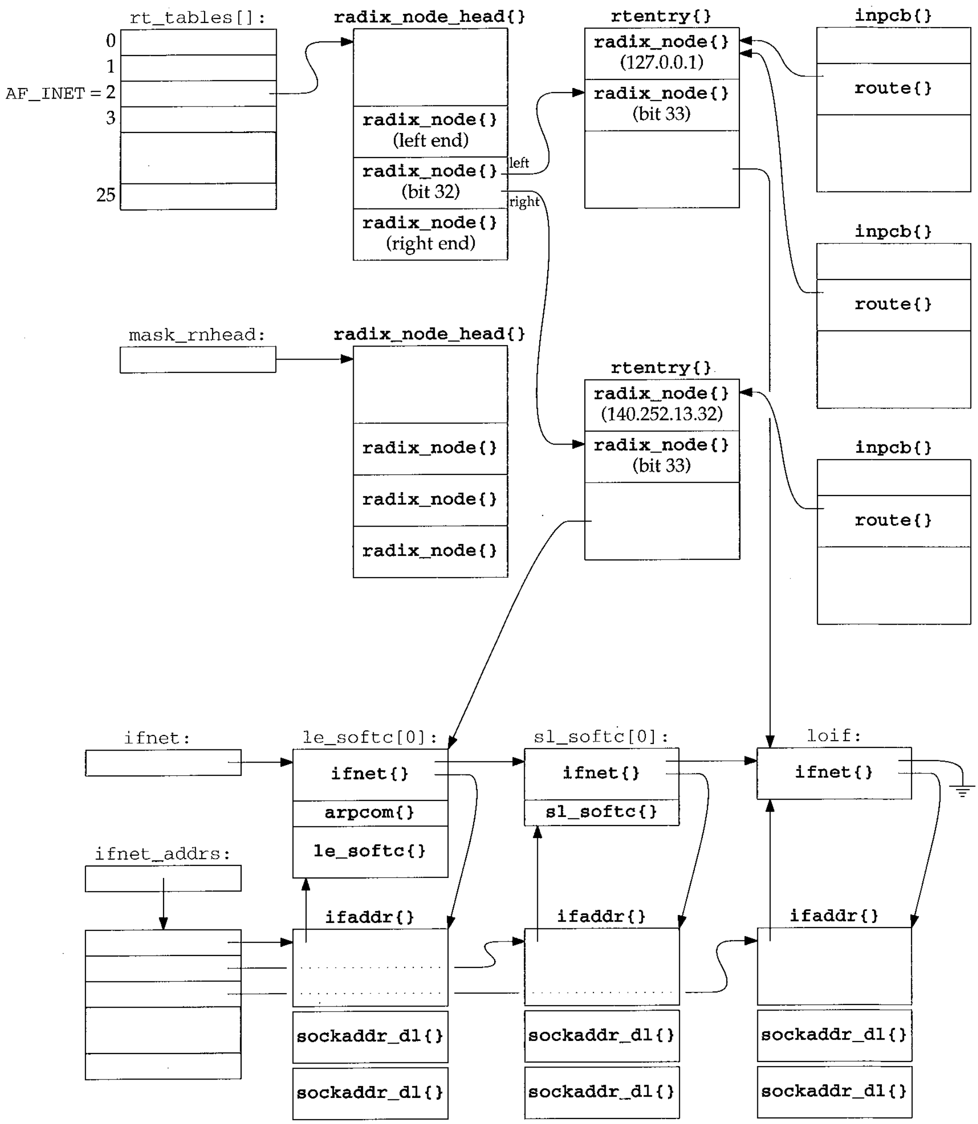

Figure 18.8 Data structures involved with routing tables.

- 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 ioctls 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.

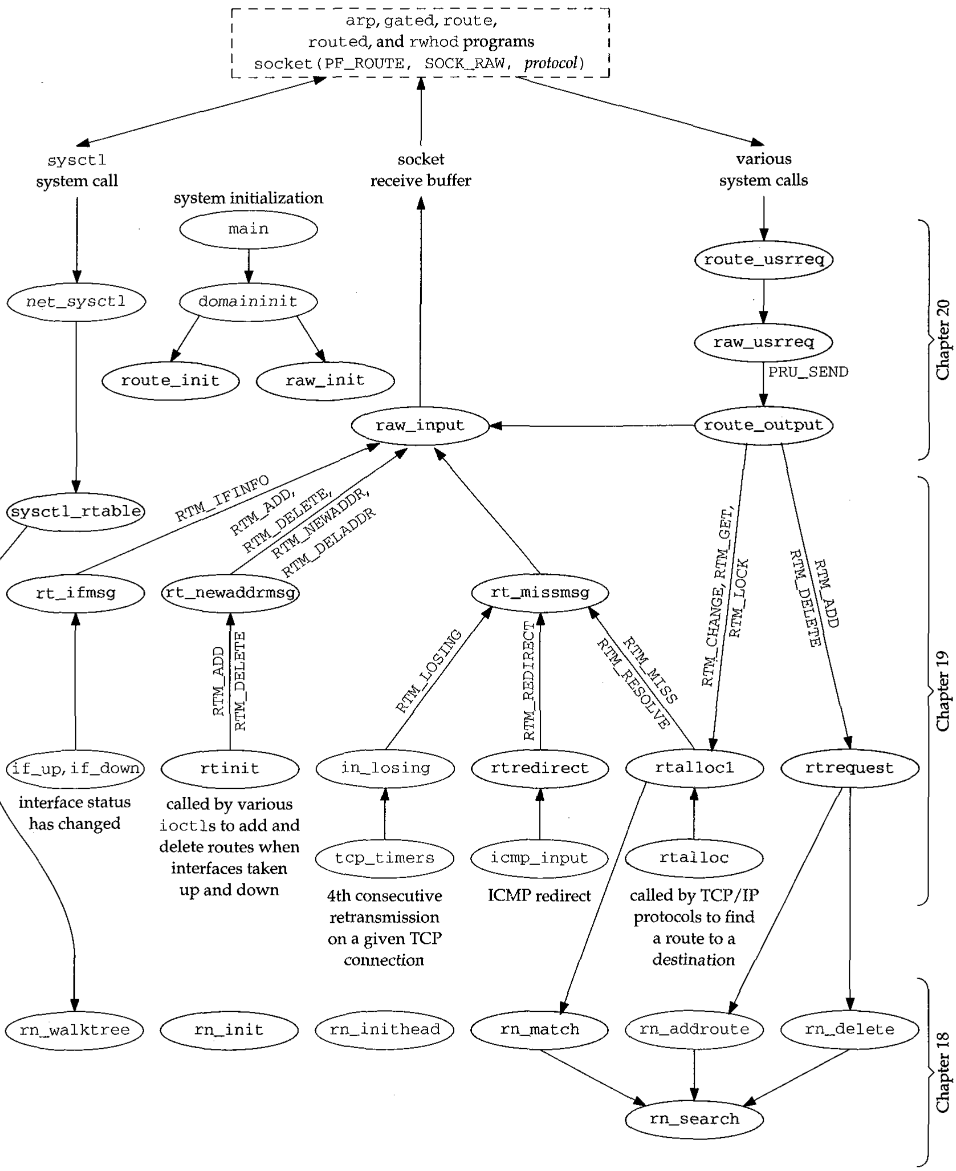

Figure 18.11 Relationships between the various routing functions.

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	struct radix_node_head * []	array of pointers to heads of routing tables
mask_rnhead	struct radix_node_head *	pointer to head of mask table
rn_mkfreelist	struct radix_mask *	head of linked list of available radix_mask structures
max_keylen	int	longest routing table key, in bytes
rn_zeros	char *	array of all zero bits, of length max_keylen
rn_ones	char *	array of all one bits, of length max_keylen
maskedKey	char *	array for masked search key, of length max_keylen
rtstat	struct rtstat	routing statistics (Figure 18.13)
rttrash	int	#routes not in table but not freed
rawcb	struct rawcb	head of doubly linked list of routing control blocks
raw_recvspace	u_long	default size of routing socket receive buffer, 8192 bytes
raw_sendspace	u_long	default size of routing socket send buffer, 8192 bytes
route_cb	struct route_cb	#routing socket listeners, per protocol, and total
route_dst	struct sockaddr	temporary for destination of routing message
route_src	struct sockaddr	temporary for source of routing message
route_proto	struct sockproto	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

```
struct radix_node_head {
    struct radix_node *rnh_treetop;
    int     rnh_addrsize;       /* (not currently used) */
    int     rnh_pktsize;        /* (not currently used) */
    struct radix_node *(*rnh_addaddr)   /* add based on sockaddr */
            (void *v, void *mask,
             struct radix_node_head * head, struct radix_node nodes[]);
    struct radix_node *(*rnh_addpkt)    /* add based on packet hdr */
            (void *v, void *mask,
             struct radix_node_head * head, struct radix_node nodes[]);
    struct radix_node *(*rnh_deladdr)   /* remove based on sockaddr */
            (void *v, void *mask, struct radix_node_head * head);
    struct radix_node *(*rnh_delpkt)    /* remove based on packet hdr */
            (void *v, void *mask, struct radix_node_head * head);
    struct radix_node *(*rnh_matchaddr)     /* locate based on sockaddr */
            (void *v, struct radix_node_head * head);
    struct radix_node *(*rnh_matchpkt)  /* locate based on packet hdr */
            (void *v, struct radix_node_head * head);
    int     (*rnh_walktree)     /* traverse tree */
            (struct radix_node_head * head, int (*f) (), void *w);

    struct radix_node rnh_nodes[3];     /* top and end nodes */
};
```

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 structures.

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

```
struct radix_node {
    struct radix_mask *rn_mklist;    /* list of masks contained in subtree */
    struct radix_node *rn_p;      /* parent pointer */
    short   rn_b;                 /* bit offset; -1-index(netmask) */
    char    rn_bmask;             /* node: mask for bit test */
    u_char  rn_flags;             /* Figure 18.20 */
    union {
        struct {                  /* leaf only data: rn_b < 0 */
            caddr_t rn_Key;       /* object of search */
            caddr_t rn_Mask;      /* netmask, if present */
            struct radix_node *rn_Dupedkey;
        } rn_leaf;
        struct {                  /* node only data: rn_b >= 0 */
            int     rn_Off;       /* where to start compare */
            struct radix_node *rn_L;    /* left pointer */
            struct radix_node *rn_R;    /* right pointer */
        } rn_node;
    } rn_u;
};

#define rn_dupedkey rn_u.rn_leaf.rn_Dupedkey
#define rn_key      rn_u.rn_leaf.rn_Key
```