```
61 #define rn_mask       rn_u.rn_leaf.rn_Mask
62 #define rn_off        rn_u.rn_node.rn_Off
63 #define rn_l          rn_u.rn_node.rn_L
64 #define rn_r          rn_u.rn_node.rn_R
```

radix.h

Figure 18.18 `radix_node` structure: the nodes of the routing tree.

41–45 The first five members are common to both internal nodes and leaves, followed by a `union` defining three members if the node is a leaf, or a different three members if the node is internal. As is common throughout the Net/3 code, a set of `#define` statements provide shorthand names for the members in the `union`.

41–42 `rn_mklist` is the head of a linked list of masks for this node. We describe this field in Section 18.9. `rn_p` points to the parent node.

43 If `rn_b` is greater than or equal to 0, the node is an internal node, else the node is a leaf. For the internal nodes, `rn_b` is the bit number to test: for example, its value is 32 in the top node of the tree in Figure 18.4. For leaves, `rn_b` is negative and its value is –1 minus the *index of the network mask.* This index is the first bit number where a 0 occurs. Figure 18.19 shows the indexes of the masks from Figure 18.4.

	32-bit IP mask (bits 32–63)								index	`rn_b`
	3333 2345	3333 6789	4444 0123	4444 4567	4455 8901	5555 2345	5555 6789	6666 0123		
`00000000:`	0000	0000	0000	0000	0000	0000	0000	0000	0	–1
`ff000000:`	1111	1111	0000	0000	0000	0000	0000	0000	40	–41
`ffffffe0:`	1111	1111	1111	1111	1111	1111	1110	0000	59	–60

Figure 18.19 Example of mask indexes.

As we can see, the index of the all-zero mask is handled specially: its index is 0, not 32.

44 `rn_bmask` is a 1-byte mask used with the internal nodes to test whether the corresponding bit is on or off. Its value is 0 in leaves. We'll see how this member is used with the `rn_off` member shortly.

45 Figure 18.20 shows the three values for the `rn_flags` member.

Constant	Description
RNF_ACTIVE	this node is alive (for `rtfree`)
RNF_NORMAL	leaf contains normal route (not currently used)
RNF_ROOT	leaf is a root leaf for the tree

Figure 18.20 `rn_flags` values.

The `RNF_ROOT` flag is set only for the three radix nodes in the `radix_node_head` structure: the top of the tree and the left and right end nodes. These three nodes can never be deleted from the routing tree.

48–49 For a leaf, `rn_key` points to the socket address structure and `rn_mask` points to a socket address structure containing the mask. If `rn_mask` is null, the implied mask is all one bits (i.e., this route is to a host, not to a network).

Figure 18.21 shows an example corresponding to the leaf for 140.252.13.32 in Figure 18.4.

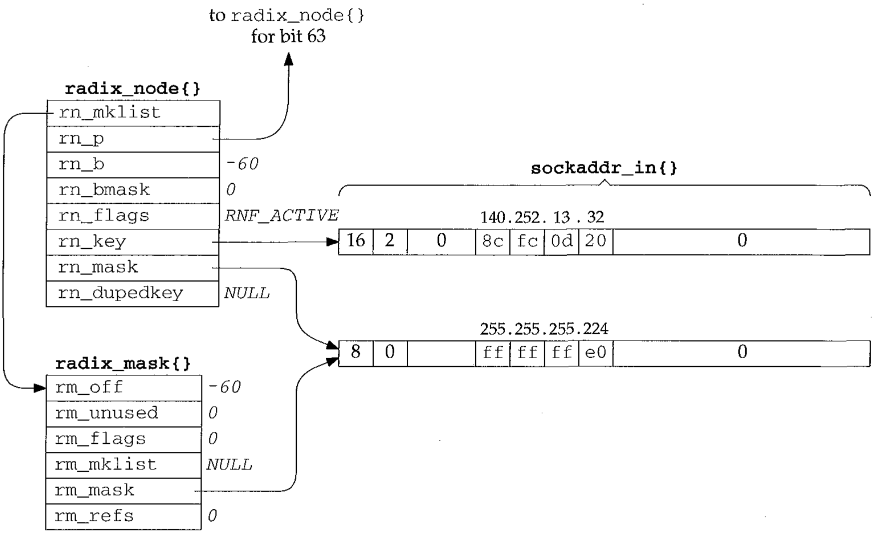

Figure 18.21 `radix_node` structure corresponding to leaf for 140.252.13.32 in Figure 18.4.

This example also shows a `radix_mask` structure, which we describe in Figure 18.22. We draw this latter structure with a smaller width, to help distinguish it as a different structure from the `radix_node`; we'll encounter both structures in many of the figures that follow. We describe the reason for the `radix_mask` structure in Section 18.9.

The `rn_b` of –60 corresponds to an index of 59. `rn_key` points to a `sockaddr_in`, with a length of 16 and an address family of 2 (`AF_INET`). The mask structure pointed to by `rn_mask` and `rm_mask` has a length of 8 and a family of 0 (this family is `AF_UNSPEC`, but it is never even looked at).

50–51 The `rn_dupedkey` pointer is used when there are multiple leaves with the same key. We describe these in Section 18.9.

52–58 We describe `rn_off` in Section 18.8. `rn_l` and `rn_r` are the left and right pointers for the internal node.

Figure 18.22 shows the `radix_mask` structure.

radix.h

```
76 extern struct radix_mask {
77     short   rm_b;                /* bit offset; -1-index(netmask) */
78     char    rm_unused;           /* cf. rn_bmask */
79     u_char  rm_flags;            /* cf. rn_flags */
80     struct radix_mask *rm_mklist;   /* more masks to try */
81     caddr_t rm_mask;             /* the mask */
82     int     rm_refs;             /* # of references to this struct */
83 }      *rn_mkfreelist;
```

radix.h

Figure 18.22 `radix_mask` structure.

76–83 Each of these structures contains a pointer to a mask: `rm_mask`, which is really a pointer to a socket address structure containing the mask. Each `radix_node` structure points to a linked list of `radix_mask` structures, allowing multiple masks per node: `rn_mklist` points to the first, and then each `rm_mklist` points to the next. This structure definition also declares the global `rn_mkfreelist`, which is the head of a linked list of available structures.

18.6 Routing Structures

The focal points of access to the kernel's routing information are

1. the `rtalloc` function, which searches for a route to a destination,
2. the `route` structure that is filled in by this function, and
3. the `rtentry` structure that is pointed to by the `route` structure.

Figure 18.8 showed that the protocol control blocks (PCBs) used by UDP and TCP (Chapter 22) contain a `route` structure, which we show in Figure 18.23.

route.h

```
46 struct route {
47     struct rtentry *ro_rt;      /* pointer to struct with information */
48     struct sockaddr ro_dst;     /* destination of this route */
49 };
```

route.h

Figure 18.23 `route` structure.

`ro_dst` is declared as a generic socket address structure, but for the Internet protocols it is a `sockaddr_in`. Notice that unlike most references to this type of structure, `ro_dst` is the structure itself, not a pointer to one.

At this point it is worth reviewing Figure 8.24, which shows the use of these routes every time an IP datagram is output.

- If the caller passes a pointer to a `route` structure, that structure is used. Otherwise a local `route` structure is used and it is set to 0, setting `ro_rt` to a null pointer. UDP and TCP pass a pointer to the `route` structure in their PCB to `ip_output`.

- If the `route` structure points to an `rtentry` structure (the `ro_rt` pointer is nonnull), and if the referenced interface is still up, and if the destination address in the `route` structure equals the destination address of the IP datagram, that route is used. Otherwise the socket address structure `so_dst` is filled in with the destination IP address and `rtalloc` is called to locate a route to that destination. For a TCP connection the destination address of the datagram never changes from the destination address of the route, but a UDP application can send a datagram to a different destination with each `sendto`.
- If `rtalloc` returns a null pointer in `ro_rt`, a route was not found and `ip_output` returns an error.
- If the `RTF_GATEWAY` flag is set in the `rtentry` structure, the route is indirect (the `G` flag in Figure 18.2). The destination address (`dst`) for the interface output function becomes the IP address of the gateway, the `rt_gateway` member, not the destination address of the IP datagram.

Figure 18.24 shows the `rtentry` structure.

route.h

```
83 struct rtentry {
84     struct radix_node rt_nodes[2];  /* a leaf and an internal node */

85     struct sockaddr *rt_gateway;     /* value associated with rn_key */
86     short   rt_flags;            /* Figure 18.25 */
87     short   rt_refcnt;           /* #held references */
88     u_long  rt_use;              /* raw #packets sent */
89     struct ifnet *rt_ifp;        /* interface to use */
90     struct ifaddr *rt_ifa;       /* interface address to use */
91     struct sockaddr *rt_genmask;     /* for generation of cloned routes */
92     caddr_t rt_llinfo;           /* pointer to link level info cache */
93     struct rt_metrics rt_rmx;    /* metrics: Figure 18.26 */
94     struct rtentry *rt_gwroute;  /* implied entry for gatewayed routes */
95 };

96 #define rt_key(r)    ((struct sockaddr *)((r)->rt_nodes->rn_key))
97 #define rt_mask(r)   ((struct sockaddr *)((r)->rt_nodes->rn_mask))
```

route.h

Figure 18.24 `rtentry` structure.

83–84 Two `radix_node` structures are contained within this structure. As we noted in the example with Figure 18.7, each time a new leaf is added to the routing tree a new internal node is also added. `rt_nodes[0]` contains the leaf entry and `rt_nodes[1]` contains the internal node. The two `#define` statements at the end of Figure 18.24 provide a shorthand access to the key and mask of this leaf node.

86 Figure 18.25 shows the various constants stored in `rt_flags` and the corresponding character output by `netstat` in the "Flags" column (Figure 18.2).

The `RTF_BLACKHOLE` flag is not output by `netstat` and the two with lowercase flag characters, `RTF_DONE` and `RTF_MASK`, are used in routing messages and not normally stored in the routing table entry.

85 If the `RTF_GATEWAY` flag is set, `rt_gateway` contains a pointer to a socket address structure containing the address (e.g., the IP address) of that gateway. Also,

Constant	netstat flag	Description
RTF_BLACKHOLE		discard packets without error (loopback driver: Figure 5.27)
RTF_CLONING	C	generate new routes on use (used by ARP)
RTF_DONE	d	kernel confirmation that message from process was completed
RTF_DYNAMIC	D	created dynamically (by redirect)
RTF_GATEWAY	G	destination is a gateway (indirect route)
RTF_HOST	H	host entry (else network entry)
RTF_LLINFO	L	set by ARP when `rt_llinfo` pointer valid
RTF_MASK	m	subnet mask present (not used)
RTF_MODIFIED	M	modified dynamically (by redirect)
RTF_PROTO1	1	protocol-specific routing flag
RTF_PROTO2	2	protocol-specific routing flag (ARP uses)
RTF_REJECT	R	discard packets with error (loopback driver: Figure 5.27)
RTF_STATIC	S	manually added entry (`route` program)
RTF_UP	U	route usable
RTF_XRESOLVE	X	external daemon resolves name (used with X.25)

Figure 18.25 `rt_flags` values.

`rt_gwroute` points to the `rtentry` for that gateway. This latter pointer was used in `ether_output` (Figure 4.15).

87 `rt_refcnt` counts the "held" references to this structure. We describe this counter at the end of Section 19.3. This counter is output as the "Refs" column in Figure 18.2.

88 `rt_use` is initialized to 0 when the structure is allocated; we saw it incremented in Figure 8.24 each time an IP datagram was output using the route. This counter is also the value printed in the "Use" column in Figure 18.2.

89–90 `rt_ifp` and `rt_ifa` point to the interface structure and the interface address structure, respectively. Recall from Figure 6.5 that a given interface can have multiple addresses, so minimally the `rt_ifa` is required.

92 The `rt_llinfo` pointer allows link-layer protocols to store pointers to their protocol-specific structures in the routing table entry. This pointer is normally used with the `RTF_LLINFO` flag. Figure 21.1 shows how ARP uses this pointer.

route.h

```
struct rt_metrics {
    u_long  rmx_locks;          /* bitmask for values kernel leaves alone */
    u_long  rmx_mtu;            /* MTU for this path */
    u_long  rmx_hopcount;       /* max hops expected */
    u_long  rmx_expire;         /* lifetime for route, e.g. redirect */
    u_long  rmx_recvpipe;       /* inbound delay-bandwith product */
    u_long  rmx_sendpipe;       /* outbound delay-bandwith product */
    u_long  rmx_ssthresh;       /* outbound gateway buffer limit */
    u_long  rmx_rtt;            /* estimated round trip time */
    u_long  rmx_rttvar;         /* estimated RTT variance */
    u_long  rmx_pksent;         /* #packets sent using this route */
};
```

route.h

Figure 18.26 `rt_metrics` structure.

93 Figure 18.26 shows the `rt_metrics` structure, which is contained within the `rtentry` structure. Figure 27.3 shows that TCP uses six members in this structure.

54–65 `rmx_locks` is a bitmask telling the kernel which of the eight metrics that follow must not be modified. The values for this bitmask are shown in Figure 20.13.

`rmx_expire` is used by ARP (Chapter 21) as a timer for each ARP entry. Contrary to the comment with `rmx_expire`, it is not used for redirects.

Figure 18.28 summarizes the structures that we've described, their relationships, and the various types of socket address structures they reference. The `rtentry` that we show is for the route to 128.32.33.5 in Figure 18.2. The other `radix_node` contained in the `rtentry` is for the bit 36 test right above this node in Figure 18.4. The two `sockaddr_dl` structures pointed to by the first `ifaddr` were shown in Figure 3.38. Also note from Figure 6.5 that the `ifnet` structure is contained within an `le_softc` structure, and the second `ifaddr` structure is contained within an `in_ifaddr` structure.

18.7 Initialization: `route_init` and `rtable_init` Functions

The initialization of the routing tables is somewhat obscure and takes us back to the `domain` structures in Chapter 7. Before outlining the function calls, Figure 18.27 shows the relevant fields from the `domain` structure (Figure 7.5) for various protocol families.

Member	OSI value	Internet value	Routing value	Unix value	XNS value	Comment
`dom_family`	*AF_ISO*	*AF_INET*	*PF_ROUTE*	*AF_UNIX*	*AF_NS*	
`dom_init`	*0*	*0*	*route_init*	*0*	*0*	
`dom_rtattach`	*rn_inithead*	*rn_inithead*	*0*	*0*	*rn_inithead*	
`dom_rtoffset`	*48*	*32*	*0*	*0*	*16*	in bits
`dom_maxrtkey`	*32*	*16*	*0*	*0*	*16*	in bytes

Figure 18.27 Members of `domain` structure relevant to routing.

The `PF_ROUTE` domain is the only one with an initialization function. Also, only the domains that require a routing table have a `dom_rtattach` function, and it is always `rn_inithead`. The routing domain and the Unix domain protocols do not require a routing table.

The `dom_rtoffset` member is the offset, in bits, (from the beginning of the domain's socket address structure) of the first bit to be examined for routing. The size of this structure in bytes is given by `dom_maxrtkey`. We saw earlier in this chapter that the offset of the IP address in the `sockaddr_in` structure is 32 bits. The `dom_maxrtkey` member is the size in bytes of the protocol's socket address structure: 16 for `sockaddr_in`.

Figure 18.29 outlines the steps involved in initializing the routing tables.

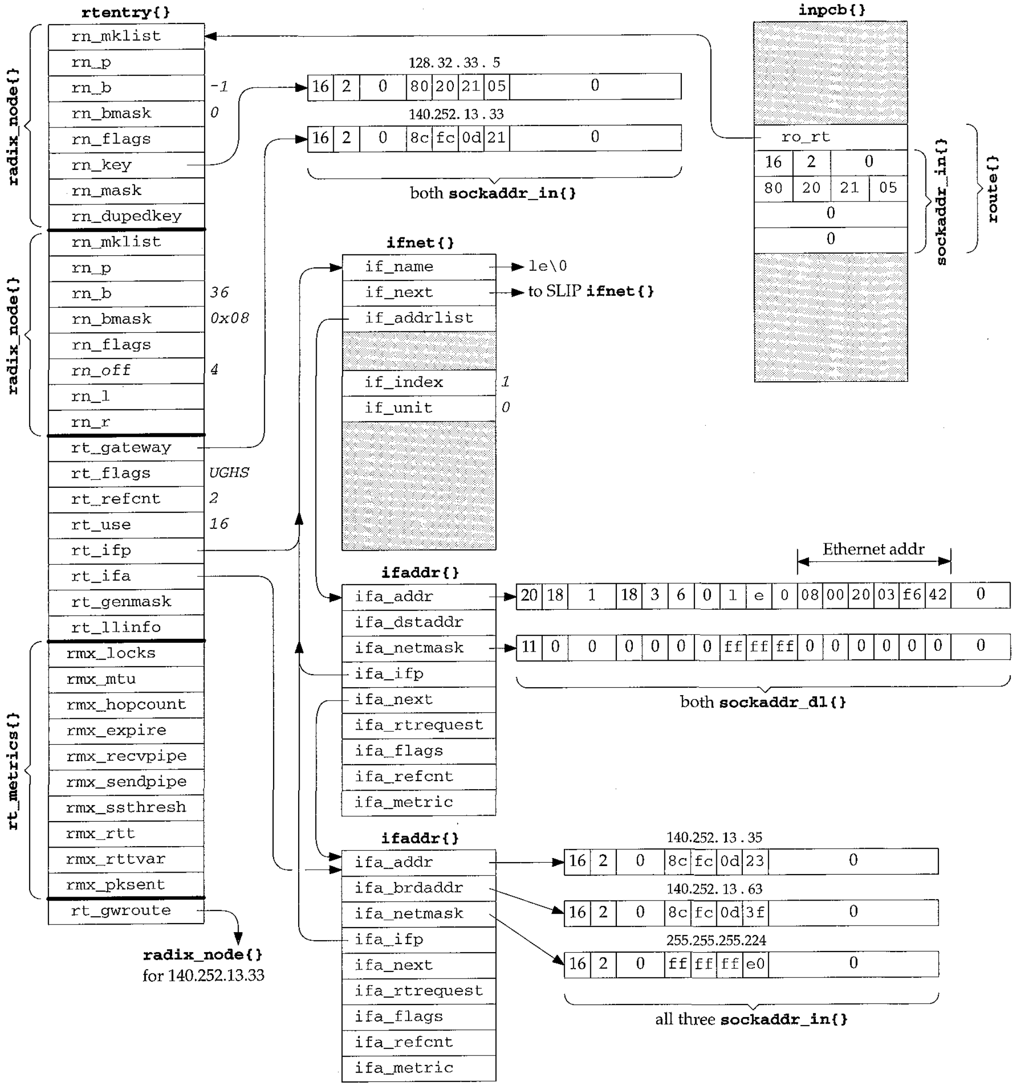

Figure 18.28 Summary of routing structures.

```
main()            /* kernel initialization */
{
    ...
    ifinit();
    domaininit();
    ...
}

domaininit()      /* Figure 7.15 */
{
    ...
    ADDDOMAIN(unix);
    ADDDOMAIN(route);
    ADDDOMAIN(inet);
    ADDDOMAIN(osi);
    ...
    for ( dp = all domains ) {
        (*dp->dom_init)();
        for ( pr = all protocols for this domain )
            (*pr->pr_init)();
    }
}

raw_init()        /* pr_init() function for SOCK_RAW/PF_ROUTE protocol */
{
    initialize head of routing protocol control blocks;
}

route_init()      /* dom_init() function for PF_ROUTE domain */
{
    rn_init();
    rtable_init();
}

rn_init()
{
    for ( dp = all domains )
        if (dp->dom_maxrtkey > max_keylen)
            max_keylen = dp->dom_maxrtkey;
    allocate and initialize rn_zeros, rn_ones, masked_key;
    rn_inithead(&mask_rnhead);   /* allocate and init tree for masks */
}

rtable_init()
{
    for ( dp = all domains )
        (*dp->dom_rtattach)(&rt_tables[dp->dom_family]);
}

rn_inithead()     /* dom_attach() function for all protocol families */
{
    allocate and initialize one radix_node_head structure;
}
```

Figure 18.29 Steps involved in initialization of routing tables.

`domaininit` is called once by the kernel's `main` function when the system is initialized. The linked list of `domain` structures is built by the `ADDDOMAIN` macro and the linked list is traversed, calling each domain's `dom_init` function, if defined. As we saw in Figure 18.27, the only `dom_init` function is `route_init`, which is shown in Figure 18.30.

route.c

```
void
route_init()
{
    rn_init();  /* initialize all zeros, all ones, mask table */

    rtable_init((void **) rt_tables);
}
```

route.c

Figure 18.30 `route_init` function.

The function `rn_init`, shown in Figure 18.32, is called only once.

The function `rtable_init`, shown in Figure 18.31, is also called only once. It in turn calls all the `dom_rtattach` functions, which initialize a routing table tree for that domain.

route.c

```
void
rtable_init(table)
void  **table;
{
    struct domain *dom;
    for (dom = domains; dom; dom = dom->dom_next)
        if (dom->dom_rtattach)
            dom->dom_rtattach(&table[dom->dom_family],
                              dom->dom_rtoffset);
}
```

route.c

Figure 18.31 `rtable_init` function: call each domain's `dom_rtattach` function.

We saw in Figure 18.27 that the only `dom_rtattach` function is `rn_inithead`, which we describe in the next section.

18.8 Initialization: `rn_init` and `rn_inithead` Functions

The function `rn_init`, shown in Figure 18.32, is called once by `route_init` to initialize some of the globals used by the radix functions.

radix.c

```
void
rn_init()
{
    char   *cp, *cplim;
    struct domain *dom;
```

```
755     for (dom = domains; dom; dom = dom->dom_next)
756         if (dom->dom_maxrtkey > max_keylen)
757             max_keylen = dom->dom_maxrtkey;
758     if (max_keylen == 0) {
759         printf("rn_init: radix functions require max_keylen be set\n");
760         return;
761     }
762     R_Malloc(rn_zeros, char *, 3 * max_keylen);
763     if (rn_zeros == NULL)
764         panic("rn_init");
765     Bzero(rn_zeros, 3 * max_keylen);
766     rn_ones = cp = rn_zeros + max_keylen;
767     maskedKey = cplim = rn_ones + max_keylen;
768     while (cp < cplim)
769         *cp++ = -1;

770     if (rn_inithead((void **) &mask_rnhead, 0) == 0)
771         panic("rn_init 2");
772 }
```

radix.c

Figure 18.32 `rn_init` function.

Determine `max_keylen`

750–761 All the `domain` structures are examined and the global `max_keylen` is set to the largest value of `dom_maxrtkey`. In Figure 18.27 the largest value is 32 for `AF_ISO`, but in a typical system that excludes the OSI and XNS protocols, `max_keylen` is 16, the size of a `sockaddr_in` structure.

Allocate and initialize `rn_zeros`, `rn_ones`, and `maskedKey`

762–769 A buffer three times the size of `max_keylen` is allocated and the pointer stored in the global `rn_zeros`. `R_Malloc` is a macro that calls the kernel's `malloc` function, specifying a type of `M_RTABLE` and `M_DONTWAIT`. We'll also encounter the macros `Bcmp`, `Bcopy`, `Bzero`, and `Free`, which call kernel functions of similar names, with the arguments appropriately type cast.

This buffer is divided into three pieces, and each piece is initialized as shown in Figure 18.33.

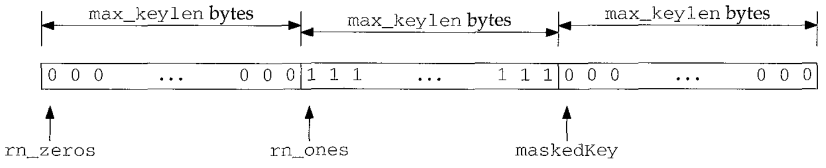

Figure 18.33 `rn_zeros`, `rn_ones`, and `maskedKey` arrays.

`rn_zeros` is an array of all zero bits, `rn_ones` is an array of all one bits, and `maskedKey` is an array used to hold a temporary copy of a search key that has been masked.

Initialize tree of masks

770–772 The function `rn_inithead` is called to initialize the head of the routing tree for the address masks; the `radix_node_head` structure pointed to by the global `mask_rnhead` in Figure 18.8.

From Figure 18.27 we see that `rn_inithead` is also the `dom_attach` function for all the protocols that require a routing table. Instead of showing the source code for this function, Figure 18.34 shows the `radix_node_head` structure that it builds for the Internet protocols.

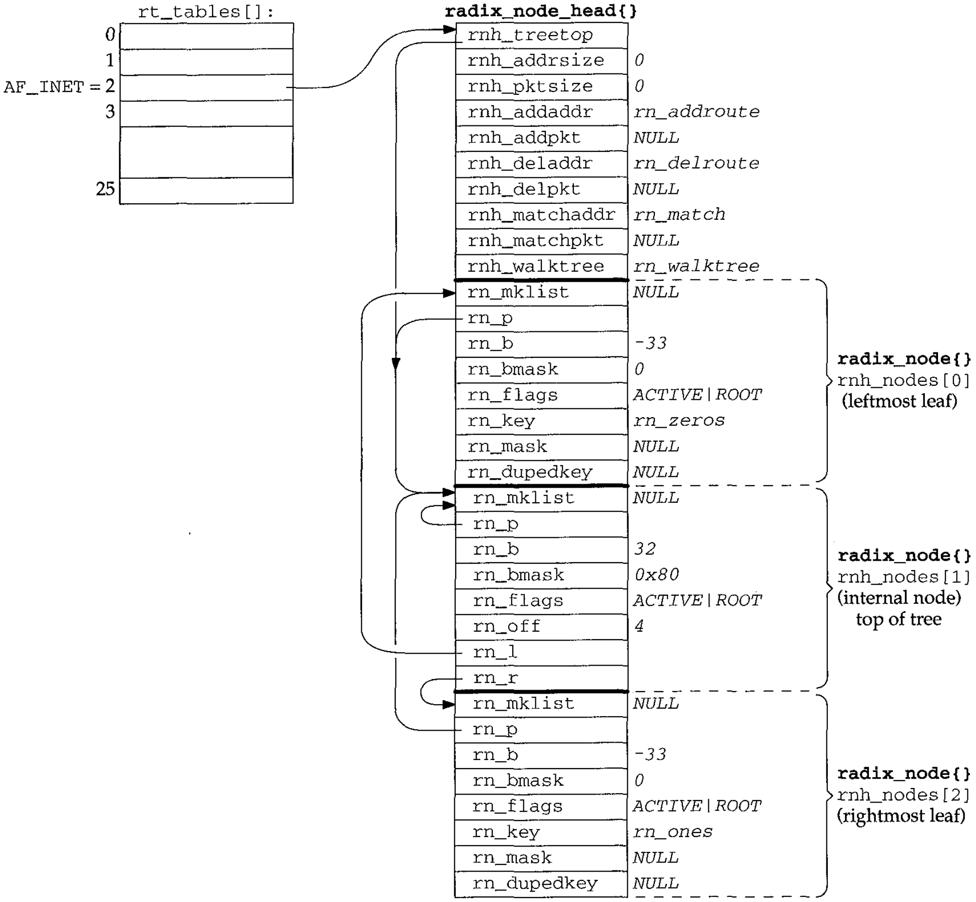

Figure 18.34 `radix_node_head` structure built by `rn_inithead` for Internet protocols.

The three `radix_node` structures form a tree: the middle of the three is the top (it is pointed to by `rnh_treetop`), the first of the three is the leftmost leaf of the tree, and the last of the three is the rightmost leaf of the tree. The parent pointer of all three nodes (`rn_p`) points to the middle node.

The value 32 for `rnh_nodes[1].rn_b` is the bit position to test. It is from the `dom_rtoffset` member of the Internet `domain` structure (Figure 18.27). Instead of performing shifts and masks during forwarding, the byte offset and corresponding byte mask are precomputed. The byte offset from the start of a socket address structure is in the `rn_off` member of the `radix_node` structure (4 in this case) and the byte mask is in the `rn_bmask` member (`0x80` in this case). These values are computed whenever a `radix_node` structure is added to the tree, to speed up the comparisons during forwarding. As additional examples, the offset and byte mask for the two nodes that test bit 33 in Figure 18.4 would be 4 and `0x40`, respectively. The offset and byte mask for the two nodes that test bit 63 would be 7 and `0x01`.

The value of –33 for the `rn_b` member of both leaves is negative one minus the index of the leaf.

The key of the leftmost node is all zero bits (`rn_zeros`) and the key of the rightmost node is all one bits (`rn_ones`).

All three nodes have the `RNF_ROOT` flag set. (We have omitted the `RNF_` prefix.) This indicates that the node is one of the three original nodes used to build the tree. These are the only nodes with this flag.

> One detail we have not mentioned is that the Network File System (NFS) also uses the routing table functions. For each mount point on the local host a `radix_node_head` structure is allocated, along with an array of pointers to these structures (indexed by the protocol family), similar to the `rt_tables` array. Each time this mount point is exported, the protocol address of the host that can mount this filesystem is added to the appropriate tree for the mount point.

18.9 Duplicate Keys and Mask Lists

Before looking at the source code that looks up entries in a routing table we need to understand two fields in the `radix_node` structure: `rn_dupedkey`, which forms a linked list of additional `radix_node` structures containing duplicate keys, and `rn_mklist`, which starts a linked list of `radix_mask` structures containing network masks.

We first return to Figure 18.4 and the two boxes on the far left of the tree labeled "end" and "default." These are duplicate keys. The leftmost node with the `RNF_ROOT` flag set (`rnh_nodes[0]` in Figure 18.34) has a key of all zero bits, but this is the same key as the default route. We would have the same problem with the rightmost end node in the tree, which has a key of all one bits, if an entry were created for 255.255.255.255, but this is the limited broadcast address, which doesn't appear in the routing table. In general, the radix node functions in Net/3 allow any key to be duplicated, if each occurrence has a unique mask.

Figure 18.35 shows the two nodes with a duplicate key of all zero bits. In this figure we have removed the `RNF_` prefix for the `rn_flags` and omit nonnull parent, left, and right pointers, which add nothing to the discussion.

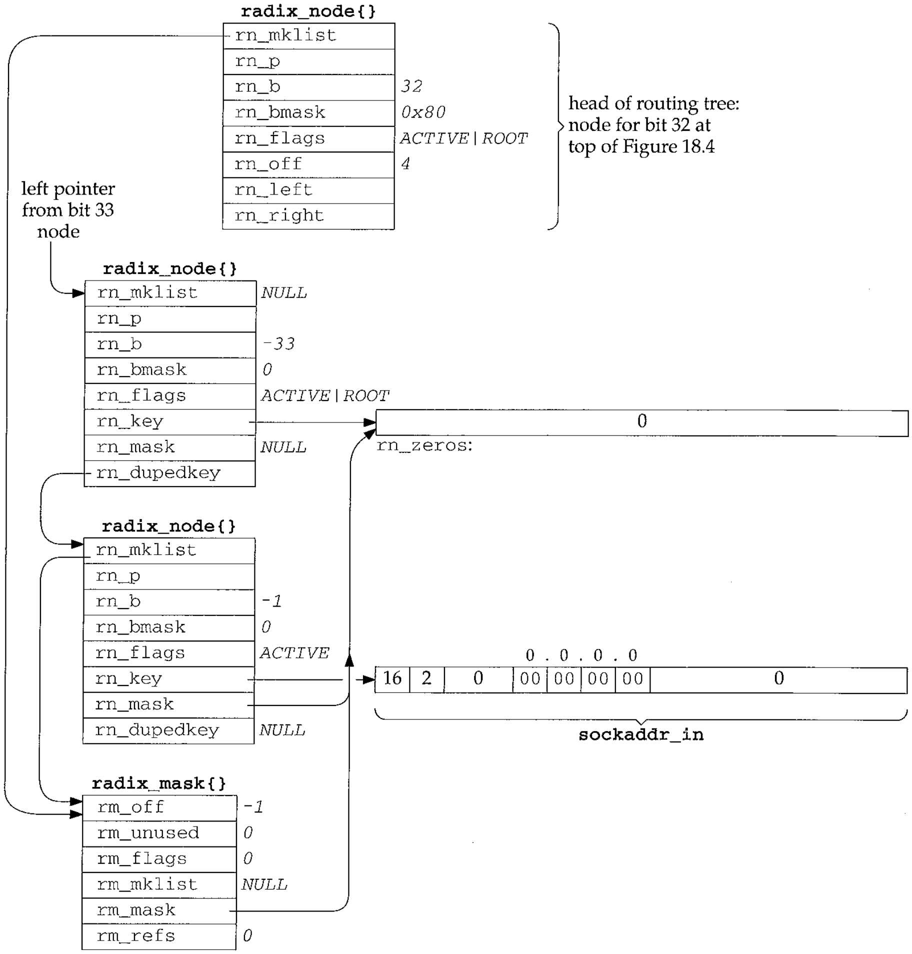

Figure 18.35 Duplicated nodes with a key of all zero bits.

The top node is the top of the routing tree—the node for bit 32 at the top of Figure 18.4. The next two nodes are leaves (their `rn_b` values are negative) with the `rn_dupedkey` member of the first pointing to the second. The first of these two leaves is the `rnh_nodes[0]` structure from Figure 18.34, which is the left end marker of the tree—its `RNF_ROOT` flag is set. Its key was explicitly set by `rn_inithead` to `rn_zeros`.

The second of these leaves is the entry for the default route. Its `rn_key` points to a `sockaddr_in` with the value 0.0.0.0, and it has a mask of all zero bits. Its `rn_mask` points to `rn_zeros`, since equivalent masks in the mask table are shared.

Normally keys are not shared, let alone shared with masks. The `rn_key` pointers of the two end markers (those with the `RNF_ROOT` flag) are special since they are built by `rn_inithead` (Figure 18.34). The key of the left end marker points to `rn_zeros` and the key of the right end marker points to `rn_ones`.

The final structure is a `radix_mask` structure and is pointed to by both the top node of the tree and the leaf for the default route. The list from the top node of the tree is used with the backtracking algorithm when the search is looking for a network mask. The list of `radix_mask` structures with an internal node specifies the masks that apply to subtrees starting at that node. In the case of duplicate keys, a mask list also appears with the leaves, as we'll see in the following example.

We now show a duplicate key that is added to the routing tree intentionally and the resulting mask list. In Figure 18.4 we have a host route for 127.0.0.1 and a network route for 127.0.0.0. The default mask for the class A network route is `0xff000000`, as we show in the figure. If we divide the 24 bits following the class A network ID into a 16-bit subnet ID and an 8-bit host ID, we can add a route for the subnet 127.0.0 with a mask of `0xffffff00`:

```
bsdi $ route add 127.0.0.0 -netmask 0xffffff00 140.252.13.33
```

Although it makes little practical sense to use network 127 in this fashion, our interest is in the resulting routing table structure. Although duplicate keys are not common with the Internet protocols (other than the previous example with the default route), duplicate keys are required to provide routes to subnet 0 of any network.

There is an implied priority in these three entries with a network ID of 127. If the search key is 127.0.0.1 it matches all three entries, but the host route is selected because it is the *most specific*: its mask (`0xffffffff`) has the most one bits. If the search key is 127.0.0.2 it matches both network routes, but the route for subnet 0, with a mask of `0xffffff00`, is more specific than the route with a mask of `0xff000000`. The search key 127.1.2.3 matches only the entry with a mask of `0xff000000`.

Figure 18.36 shows the resulting tree structure, starting at the internal node for bit 33 from Figure 18.4. We show two boxes for the entry with the key of 127.0.0.0 since there are two leaves with this duplicate key.

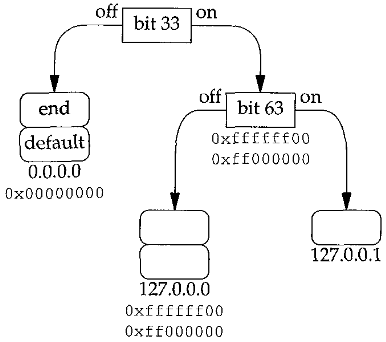

Figure 18.36 Routing tree showing duplicate keys for 127.0.0.0.

Figure 18.37 shows the resulting `radix_node` and `radix_mask` structures.

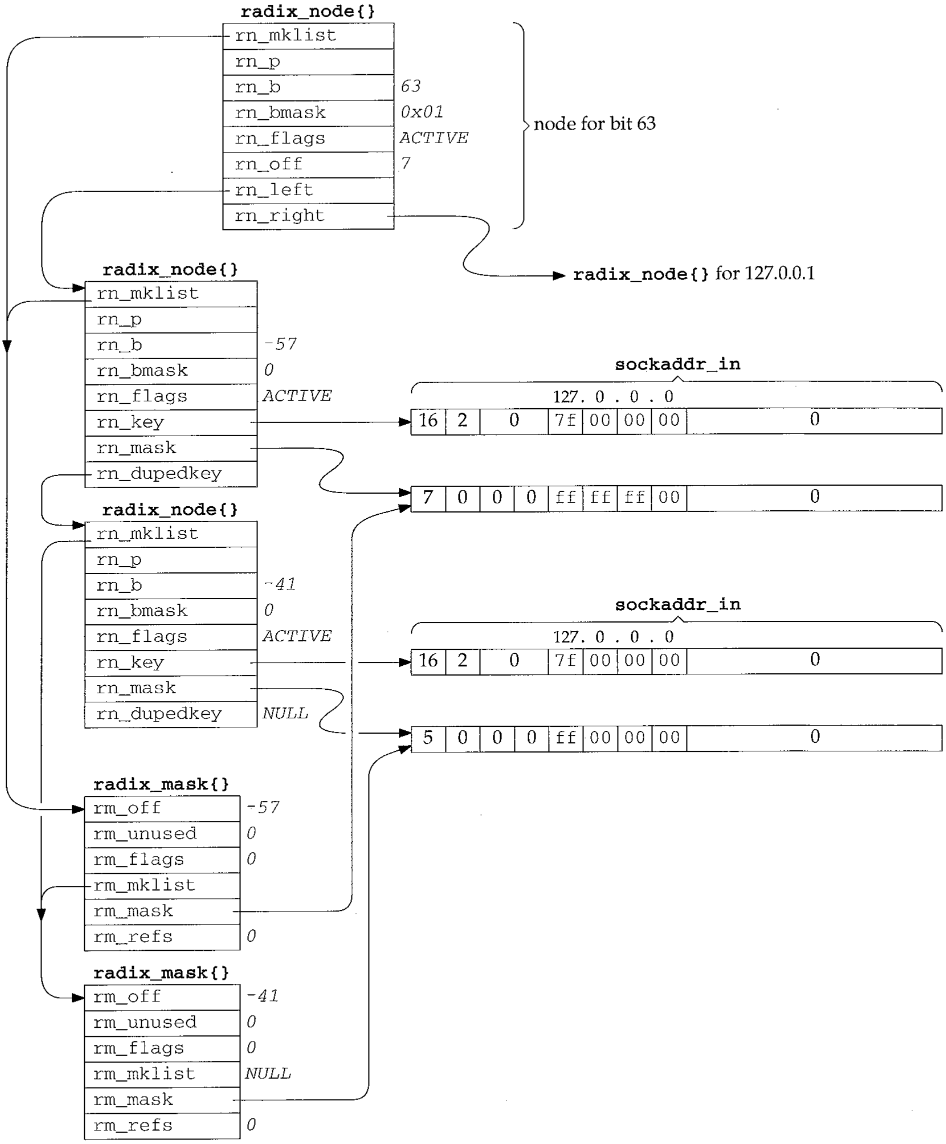

Figure 18.37 Example routing table structures for the duplicate keys for network 127.0.0.0.

First look at the linked list of `radix_mask` structures for each `radix_node`. The mask list for the top node (bit 63) consists of the entry for `0xffffff00` followed by `0xff000000`. The more-specific mask comes first in the list so that it is tried first. The mask list for the second `radix_node` (the one with the `rn_b` of –57) is the same as that of the first. But the list for the third `radix_node` consists of only the entry with a mask of `0xff000000`.

Notice that masks with the same value are shared but keys with the same value are not. This is because the masks are maintained in their own routing tree, explicitly to be shared, because equal masks are so common (e.g., every class C network route has the same mask of `0xffffff00`), while equal keys are infrequent.

18.10 `rn_match` Function

We now show the `rn_match` function, which is called as the `rnh_matchaddr` function for the Internet protocols. We'll see that it is called by the `rtalloc1` function, which is called by the `rtalloc` function. The algorithm is as follows:

1. Start at the top of the tree and go to the leaf corresponding to the bits in the search key. Check the leaf for an exact match (Figure 18.38).
2. Check the leaf for a network match (Figure 18.40).
3. Backtrack (Figure 18.43).

Figure 18.38 shows the first part of `rn_match`.

radix.c

```
struct radix_node *
rn_match(v_arg, head)
void   *v_arg;
struct radix_node_head *head;
{
    caddr_t v = v_arg;
    struct radix_node *t = head->rnh_treetop, *x;
    caddr_t cp = v, cp2, cp3;
    caddr_t cplim, mstart;
    struct radix_node *saved_t, *top = t;
    int     off = t->rn_off, vlen = *(u_char *) cp, matched_off;

    /*
     * Open code rn_search(v, top) to avoid overhead of extra
     * subroutine call.
     */
    for (; t->rn_b >= 0;) {
        if (t->rn_bmask & cp[t->rn_off])
            t = t->rn_r;        /* right if bit on */
        else
            t = t->rn_l;        /* left if bit off */
    }
```

```
156      /*
157       * See if we match exactly as a host destination
158       */
159      cp += off;
160      cp2 = t->rn_key + off;
161      cplim = v + vlen;
162      for (; cp < cplim; cp++, cp2++)
163          if (*cp != *cp2)
164              goto onl;
165      /*
166       * This extra grot is in case we are explicitly asked
167       * to look up the default.  Ugh!
168       */
169      if ((t->rn_flags & RNF_ROOT) && t->rn_dupedkey)
170          t = t->rn_dupedkey;
171      return t;
172    onl:
```
radix.c

Figure 18.38 `rn_match` function: go down tree, check for exact host match.

135–145 The first argument `v_arg` is a pointer to a socket address structure, and the second argument `head` is a pointer to the `radix_node_head` structure for the protocol. All protocols call this function (Figure 18.17) but each calls it with a different `head` argument.

In the assignment statements, `off` is the `rn_off` member of the top node of the tree (4 for Internet addresses, from Figure 18.34), and `vlen` is the length field from the socket address structure of the search key (16 for Internet addresses).

Go down the tree to the corresponding leaf

146–155 This loop starts at the top of the tree and moves down the left and right branches until a leaf is encountered (`rn_b` is less than 0). Each test of the appropriate bit is made using the precomputed byte mask in `rn_bmask` and the corresponding precomputed offset in `rn_off`. For Internet addresses, `rn_off` will be 4, 5, 6, or 7.

Check for exact match

156–164 When the leaf is encountered, a check is first made for an exact match. *All* bytes of the socket address structure, starting at the `rn_off` value for the protocol family, are compared. This is shown in Figure 18.39 for an Internet socket address structure.

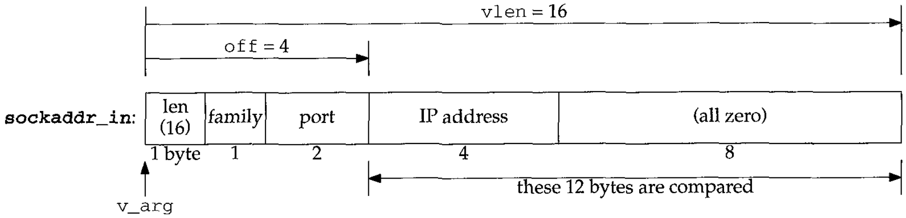

Figure 18.39 Variables during comparison of `sockaddr_in` structures.

As soon as a mismatch is found, a jump is made to `onl`.

> Normally the final 8 bytes of the `sockaddr_in` are 0 but proxy ARP (Section 21.12) sets one of these bytes nonzero. This allows two routing table entries for a given IP address: one for the normal IP address (with the final 8 bytes of 0) and a proxy ARP entry for the same IP address (with one of the final 8 bytes nonzero).

The length byte in Figure 18.39 was assigned to `vlen` at the beginning of the function, and we'll see that `rtalloc1` uses the family member to select the routing table to search. The port is never used by the routing functions.

Explicit check for default

165–172 Figure 18.35 showed that the default route is stored as a duplicate leaf with a key of 0. The first of the duplicate leaves has the `RNF_ROOT` flag set. Hence if the `RNF_ROOT` flag is set in the matching node and the leaf contains a duplicate key, the value of the pointer `rn_dupedkey` is returned (i.e., the pointer to the node containing the default route in Figure 18.35). If a default route has not been entered and the search matches the left end marker (a key of all zero bits), or if the search encounters the right end marker (a key of all one bits), the returned pointer `t` points to a node with the `RNF_ROOT` flag set. We'll see that `rtalloc1` explicitly checks whether the matching node has this flag set, and considers such a match an error.

At this point in `rn_match` a leaf has been reached but it is not an exact match with the search key. The next part of the function, shown in Figure 18.40, checks whether the leaf is a network match.

radix.c

```
173     matched_off = cp - v;
174     saved_t = t;
175     do {
176         if (t->rn_mask) {
177             /*
178              * Even if we don't match exactly as a host;
179              * we may match if the leaf we wound up at is
180              * a route to a net.
181              */
182             cp3 = matched_off + t->rn_mask;
183             cp2 = matched_off + t->rn_key;
184             for (; cp < cplim; cp++)
185                 if ((*cp2++ ^ *cp) & *cp3++)
186                     break;
187             if (cp == cplim)
188                 return t;
189             cp = matched_off + v;
190         }
191     } while (t = t->rn_dupedkey);
192     t = saved_t;
```

radix.c

Figure 18.40 `rn_match` function: check for network match.

173–174 `cp` points to the unequal byte in the search key. `matched_off` is set to the offset of this byte from the start of the socket address structure.

175–183 The `do while` loop iterates through all duplicate leaves and each one with a network mask is compared. Let's work through the code with an example. Assume we're looking up the IP address 140.252.13.60 in the routing table in Figure 18.4. The search will end up at the node labeled 140.252.13.32 (bits 62 and 63 are both off), which contains a network mask. Figure 18.41 shows the structures when the `for` loop in Figure 18.40 starts executing.

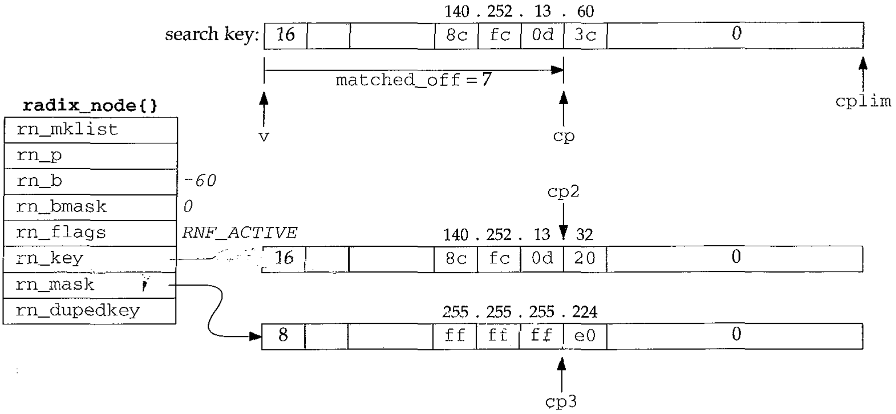

Figure 18.41 Example for network mask comparison.

The search key and the routing table key are both `sockaddr_in` structures, but the length of the mask is different. The mask length is the minimum number of bytes containing nonzero values. All the bytes past this point, up through `max_keylen`, are 0.

184–190 The search key is exclusive ORed with the routing table key, and the result logically ANDed with the network mask, one byte at a time. If the resulting byte is ever nonzero, the loop terminates because they don't match (Exercise 18.1). If the loop terminates normally, however, the search key ANDed with the network mask matches the routing table entry. The pointer to the routing table entry is returned.

Figure 18.42 shows how this example matches, and how the IP address 140.252.13.188 does not match, looking at just the fourth byte of the IP address. The search for both IP addresses ends up at this node since both addresses have bits 57, 62, and 63 off.

	search key = 140.252.13.60	search key = 140.252.13.188
search key byte (`*cp`):	`0011 1100 = 3c`	`1011 1100 = bc`
routing table key byte (`*cp2`):	`0010 0000 = 20`	`0010 0000 = 20`
exclusive OR:	`0001 1100`	`1001 1100`
network mask byte (`*cp3`):	`1110 0000 = e0`	`1110 0000 = e0`
logical AND:	`0000 0000`	`1000 0000`

Figure 18.42 Example of search key match using network mask.

The first example (140.252.13.60) matches since the result of the logical AND is 0 (and all the remaining bytes in the address, the key, and the mask are all 0). The other example does not match since the result of the logical AND is nonzero.

191 If the routing table entry has duplicate keys, the loop is repeated for each key.

The final portion of rn_match, shown in Figure 18.43, backtracks up the tree, looking for a network match or a match with the default.

radix.c

```
193     /* start searching up the tree */
194     do {
195         struct radix_mask *m;
196         t = t->rn_p;
197         if (m = t->rn_mklist) {
198             /*
199              * After doing measurements here, it may
200              * turn out to be faster to open code
201              * rn_search_m here instead of always
202              * copying and masking.
203              */
204             off = min(t->rn_off, matched_off);
205             mstart = maskedKey + off;
206             do {
207                 cp2 = mstart;
208                 cp3 = m->rm_mask + off;
209                 for (cp = v + off; cp < cplim;)
210                     *cp2++ = *cp++ & *cp3++;
211                 x = rn_search(maskedKey, t);
212                 while (x && x->rn_mask != m->rm_mask)
213                     x = x->rn_dupedkey;
214                 if (x &&
215                     (Bcmp(mstart, x->rn_key + off,
216                           vlen - off) == 0))
217                     return x;
218             } while (m = m->rm_mklist);
219         }
220     } while (t != top);
221     return 0;
222 };
```

radix.c

Figure 18.43 rn_match function: backtrack up the tree.

193–195 The do while loop continues up the tree, checking each level, until the top has been checked.

196 The pointer t is replaced with the pointer to the parent node, moving up one level. Having the parent pointer in each node simplifies backtracking.

197–210 Each level is checked only if the internal node has a nonnull list of masks. rn_mklist is a pointer to a linked list of radix_node structures, each containing a mask that applies to the subtree starting at that node. The inner do while loop iterates through each radix_mask structure on the list.

Using the previous example, 140.252.13.188, Figure 18.44 shows the various data structures when the innermost for loop starts. This loop logically ANDs each byte of the search key with each byte of the mask, storing the result in the global maskedKey. The mask value is 0xffffffe0 and the search would have backtracked from the leaf for 140.252.13.32 in Figure 18.4 two levels to the node that tests bit 62.

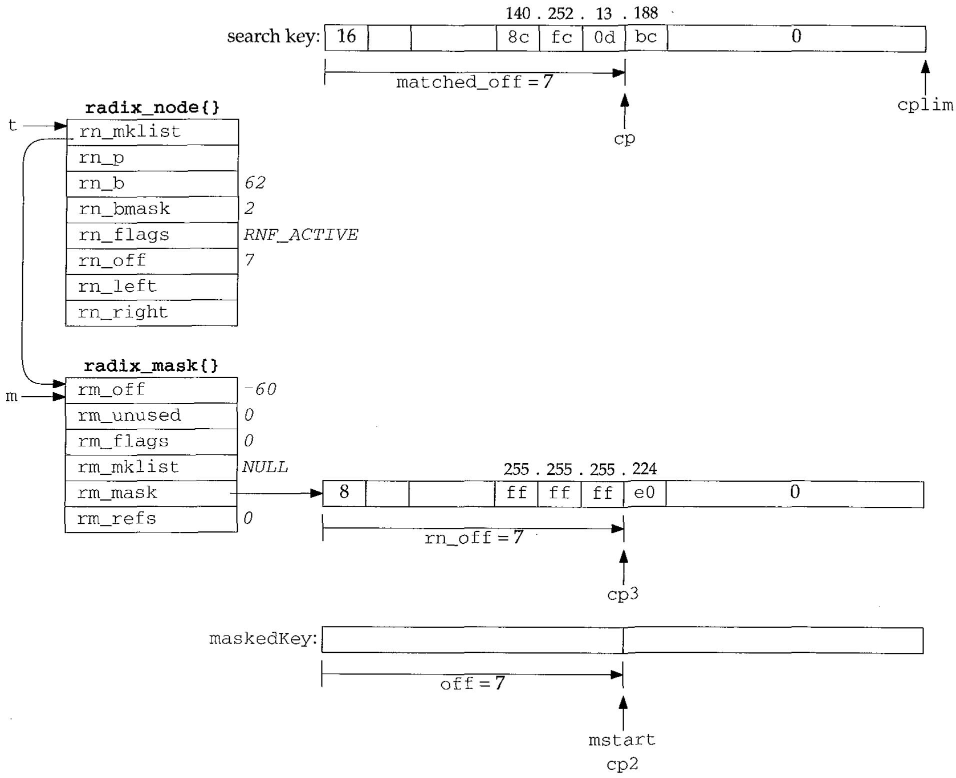

Figure 18.44 Preparation to search again using masked search key.

Once the `for` loop completes, the masking is complete, and `rn_search` (shown in Figure 18.48) is called with `maskedKey` as the search key and the pointer `t` as the top of the subtree to search. Figure 18.45 shows the value of `maskedKey` for our example.

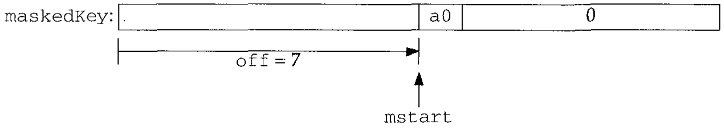

Figure 18.45 `maskedKey` when `rn_search` is called.

The byte `0xa0` is the logical AND of `0xbc` (188, the search key) and `0xe0` (the mask).

211 `rn_search` proceeds down the tree from its starting point, branching right or left depending on the key, until a leaf is reached. In this example the search key is the 9 bytes shown in Figure 18.45 and the leaf that's reached is the one labeled 140.252.13.32 in Figure 18.4, since bits 62 and 63 are off in the byte `0xa0`. Figure 18.46 shows the data structures when `Bcmp` is called to check if a match has been found.

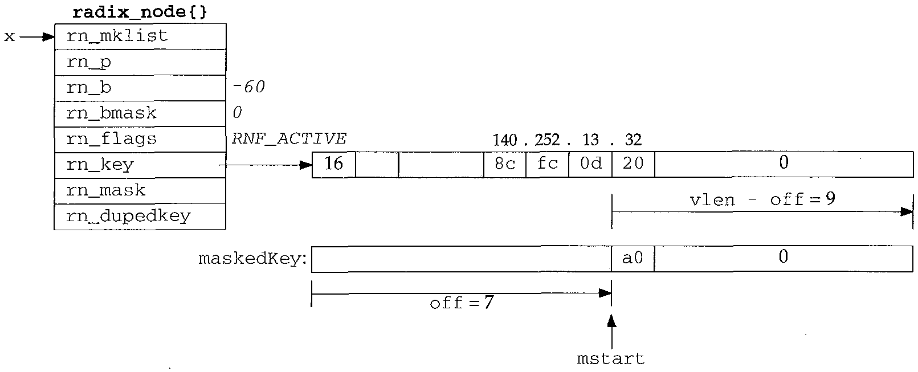

Figure 18.46 Comparison of `maskedKey` and new leaf.

Since the 9-byte strings are not the same, the comparison fails.

212–221 This `while` loop handles duplicate keys, each with a different mask. The only key of the duplicates that is compared is the one whose `rn_mask` pointer equals `m->rm_mask`. As an example, recall Figures 18.36 and 18.37. If the search starts at the node for bit 63, the first time through the inner `do while` loop `m` points to the `radix_mask` structure for `0xffffff00`. When `rn_search` returns the pointer to the first of the duplicate leaves for 127.0.0.0, the `rm_mask` of this leaf equals `m->rm_mask`, so `Bcmp` is called. If the comparison fails, `m` is replaced with the pointer to the next `radix_mask` structure on the list (the one with a mask of `0xff000000`) and the `do while` loop iterates around again with the new mask. `rn_search` again returns the pointer to the first of the duplicate leaves for 127.0.0.0, but its `rn_mask` does not equal `m->rm_mask`. The `while` steps to the next of the duplicate leaves and its `rn_mask` is the right one.

Returning to our example with the search key of 140.252.13.188, since the search from the node that tests bit 62 failed, the backtracking continues up the tree until the top is reached, which is the next node up the tree with a nonnull `rn_mklist`.

Figure 18.47 shows the data structures when the top node of the tree is reached. At this point `maskedKey` is computed (it is all zero bits) and `rn_search` starts at this node (the top of the tree) and continues down the two left branches to the leaf labeled "default" in Figure 18.4.

When `rn_search` returns, `x` points to the `radix_node` with an `rn_b` of –33, which is the first leaf encountered after the two left branches from the top of the tree. But `x->rn_mask` (which is null) does not equal `m->rm_mask`, so `x` is replaced with `x->rn_dupedkey`. The test of the `while` loop occurs again, but now `x->rn_mask` equals `m->rm_mask`, so the `while` loop terminates. `Bcmp` compares the 12 bytes of 0 starting at `mstart` with the 12 bytes of 0 stating at `x->rn_key` plus 4, and since they're equal, the function returns the pointer `x`, which points to the entry for the default route.

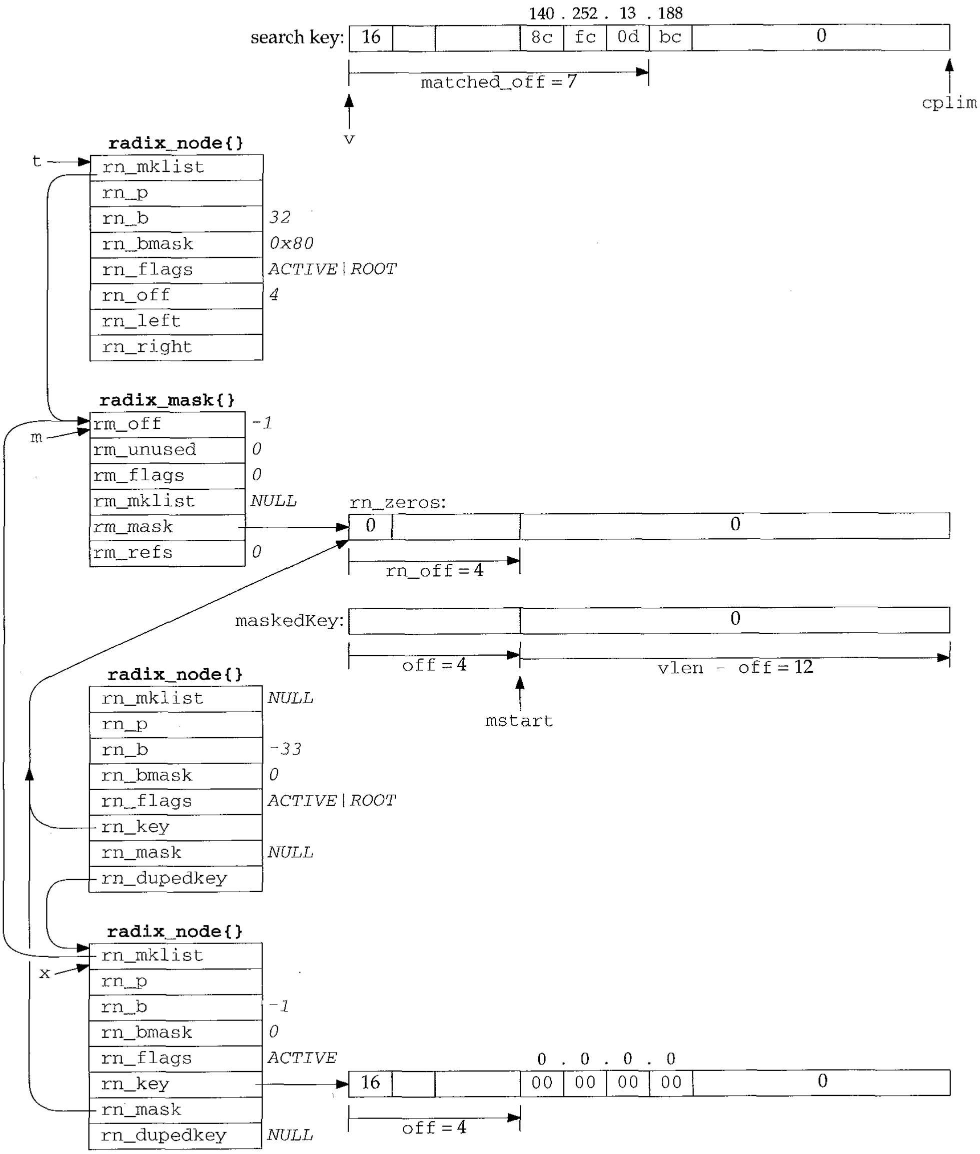

Figure 18.47 Backtrack to top of tree and `rn_search` that locates default leaf.

18.11 `rn_search` Function

`rn_search` was called in the previous section from `rn_match` to search a subtree of the routing table.

radix.c

```
struct radix_node *
rn_search(v_arg, head)
void    *v_arg;
struct radix_node *head;
{
    struct radix_node *x;
    caddr_t v;

    for (x = head, v = v_arg; x->rn_b >= 0;) {
        if (x->rn_bmask & v[x->rn_off])
            x = x->rn_r;        /* right if bit on */
        else
            x = x->rn_l;        /* left if bit off */
    }
    return (x);
};
```

radix.c

Figure 18.48 `rn_search` function.

This loop is similar to the one in Figure 18.38. It compares one bit in the search key at each node, branching left if the bit is off or right if the bit is on, terminating when a leaf is encountered. The pointer to that leaf is returned.

18.12 Summary

Each routing table entry is identified by a key: the destination IP address in the case of the Internet protocols, which is either a host address or a network address with an associated network mask. Once the entry is located by searching for the key, additional information in the entry specifies the IP address of a router to which datagrams should be sent for the destination, a pointer to the interface to use, metrics, and so on.

The information maintained by the Internet protocols is the `route` structure, composed of just two elements: a pointer to a routing table entry and the destination address. We'll encounter one of these `route` structures in each of the Internet protocol control blocks used by UDP, TCP, and raw IP.

The Patricia tree data structure is well suited to routing tables. Routing table lookups occur much more frequently than adding or deleting routes, so from a performance standpoint using Patricia trees for the routing table makes sense. Patricia trees provide fast lookups at the expense of additional work in adding and deleting. Measurements in [Sklower 1991] comparing the radix tree approach to the Net/1 hash table show that the radix tree method is about two times faster in building a test tree and four times faster in searching.

Exercises

18.1 We said with Figure 18.3 that the general condition for matching a routing table entry is that the search key logically ANDed with the routing table mask equal the routing table key. But in Figure 18.40 a different test is used. Build a logic truth table showing that the two tests are the same.

18.2 Assume a Net/3 system needs a routing table with 20,000 entries (IP addresses). Approximately how much memory is required for this, ignoring the space required for the masks?

18.3 What is the limit imposed on the length of a routing table key by the `radix_node` structure?

19

Routing Requests and Routing Messages

19.1 Introduction

The various protocols within the kernel don't access the routing trees directly, using the functions from the previous chapter, but instead call a few functions that we describe in this chapter: `rtalloc` and `rtalloc1` are two that perform routing table lookups, `rtrequest` adds and deletes routing table entries, and `rtinit` is called by most interfaces when the interface goes up or down.

Routing messages communicate information in two directions. A process such as the `route` command or one of the routing daemons (`routed` or `gated`) writes routing messages to a routing socket, causing the kernel to add a new route, delete an existing route, or modify an existing route. The kernel also generates routing messages that can be read by any routing socket when events occur in which the processes might be interested: an interface has gone down, a redirect has been received, and so on. In this chapter we cover the formats of these routing messages and the information contained therein, and we save our discussion of routing sockets until the next chapter.

Another interface provided by the kernel to the routing tables is through the `sysctl` system call, which we describe at the end of this chapter. This system call allows a process to read the entire routing table or a list of all the configured interfaces and interface addresses.

19.2 `rtalloc` and `rtalloc1` Functions

`rtalloc` and `rtalloc1` are the functions normally called to look up an entry in the routing table. Figure 19.1 shows `rtalloc`.

route.c

```
58 void
59 rtalloc(ro)
60 struct route *ro;
61 {
62     if (ro->ro_rt && ro->ro_rt->rt_ifp && (ro->ro_rt->rt_flags & RTF_UP))
63         return;                 /* XXX */
64     ro->ro_rt = rtalloc1(&ro->ro_dst, 1);
65 }
```

route.c

Figure 19.1 `rtalloc` function.

58–65 The argument `ro` is often the pointer to a `route` structure contained in an Internet PCB (Chapter 22) which is used by UDP and TCP. If `ro` already points to an `rtentry` structure (`ro_rt` is nonnull), and that structure points to an interface structure, and the route is up, the function returns. Otherwise `rtalloc1` is called with a second argument of 1. We'll see the purpose of this argument shortly.

`rtalloc1`, shown in Figure 19.2, calls the `rnh_matchaddr` function, which is always `rn_match` (Figure 18.17) for Internet addresses.

66–76 The first argument is a pointer to a socket address structure containing the address to search for. The `sa_family` member selects the routing table to search.

Call `rn_match`

77–78 If the following three conditions are met, the search is successful.

1. A routing table exists for the protocol family,
2. `rn_match` returns a nonnull pointer, and
3. the matching `radix_node` does not have the `RNF_ROOT` flag set.

Remember that the two leaves that mark the end of the tree both have the `RNF_ROOT` flag set.

Search fails

94–101 If the search fails because any one of the three conditions is not met, the statistic `rts_unreach` is incremented and if the second argument to `rtalloc1` (`report`) is nonzero, a routing message is generated that can be read by any interested processes on a routing socket. The routing message has the type `RTM_MISS`, and the function returns a null pointer.

79 If all three of the conditions are met, the lookup succeeded and the pointer to the matching `radix_node` is stored in `rt` and `newrt`. Notice that in the definition of the `rtentry` structure (Figure 18.24) the two `radix_node` structures are at the beginning, and, as shown in Figure 18.8, the first of these two structures contains the leaf node. Therefore the pointer to a `radix_node` structure returned by `rn_match` is really a pointer to an `rtentry` structure, which is the matching leaf node.

route.c

```
 66 struct rtentry *
 67 rtalloc1(dst, report)
 68 struct sockaddr *dst;
 69 int     report;
 70 {
 71     struct radix_node_head *rnh = rt_tables[dst->sa_family];
 72     struct rtentry *rt;
 73     struct radix_node *rn;
 74     struct rtentry *newrt = 0;
 75     struct rt_addrinfo info;
 76     int     s = splnet(), err = 0, msgtype = RTM_MISS;

 77     if (rnh && (rn = rnh->rnh_matchaddr((caddr_t) dst, rnh)) &&
 78         ((rn->rn_flags & RNF_ROOT) == 0)) {
 79         newrt = rt = (struct rtentry *) rn;
 80         if (report && (rt->rt_flags & RTF_CLONING)) {
 81             err = rtrequest(RTM_RESOLVE, dst, SA(0),
 82                             SA(0), 0, &newrt);
 83             if (err) {
 84                 newrt = rt;
 85                 rt->rt_refcnt++;
 86                 goto miss;
 87             }
 88             if ((rt = newrt) && (rt->rt_flags & RTF_XRESOLVE)) {
 89                 msgtype = RTM_RESOLVE;
 90                 goto miss;
 91             }
 92         } else
 93             rt->rt_refcnt++;
 94     } else {
 95         rtstat.rts_unreach++;
 96       miss:if (report) {
 97             bzero((caddr_t) & info, sizeof(info));
 98             info.rti_info[RTAX_DST] = dst;
 99             rt_missmsg(msgtype, &info, 0, err);
100         }
101     }
102     splx(s);
103     return (newrt);
104 }
```

route.c

Figure 19.2 `rtalloc1` function.

Create clone entries

80–82 If the caller specified a nonzero second argument, and if the `RTF_CLONING` flag is set, `rtrequest` is called with a command of `RTM_RESOLVE` to create a new `rtentry` structure that is a clone of the one that was located. This feature is used by ARP and for multicast addresses.

Clone creation fails

83–87 If `rtrequest` returns an error, `newrt` is set back to the entry returned by `rn_match` and its reference count is incremented. A jump is made to `miss` where an `RTM_MISS` message is generated.

Check for external resolution

88–91 If `rtrequest` succeeds but the newly cloned entry has the `RTF_XRESOLVE` flag set, a jump is made to `miss`, this time to generate an `RTM_RESOLVE` message. The intent of this message is to notify a user process when the route is created, and it could be used with the conversion of IP addresses to X.121 addresses.

Increment reference count for normal successful search

92–93 When the search succeeds but the `RTF_CLONING` flag is not set, this statement increments the entry's reference count. This is the normal flow through the function, which then returns the nonnull pointer.

For a small function, `rtalloc1` has many options in how it operates. There are seven different flows through the function, summarized in Figure 19.3.

	`report` argument	`RTF_CLONING` flag	`RTM_RESOLVE` return	`RTF_XRESOLVE` flag	routing message generated	`rt_refcnt`	return value
entry not found	0						null
	1				`RTM_MISS`		null
entry found		0				++	ptr
	0					++	ptr
	1	1	OK	0		++	ptr
	1	1	OK	1	`RTM_RESOLVE`	++	ptr
	1	1	error		`RTM_MISS`	++	ptr

Figure 19.3 Summary of operation of `rtalloc1`.

We note that the first two rows (entry not found) are impossible if a default route exists. Also we show `rt_refcnt` being incremented in the fifth and sixth rows when the call to `rtrequest` with a command of `RTM_RESOLVE` is OK. The increment is done by `rtrequest`.

19.3 `RTFREE` Macro and `rtfree` Function

The `RTFREE` macro, shown in Figure 19.4, calls the `rtfree` function only if the reference count is less than or equal to 1, otherwise it just decrements the reference count.

209–213 The `rtfree` function, shown in Figure 19.5, releases an `rtentry` structure when there are no more references to it. We'll see in Figure 22.7, for example, that when a process control block is released, if it points to a routing entry, `rtfree` is called.

route.h

```
209 #define RTFREE(rt) \
210     if ((rt)->rt_refcnt <= 1) \
211         rtfree(rt); \
212     else \
213         (rt)->rt_refcnt--;       /* no need for function call */
```

route.h

Figure 19.4 RTFREE macro.

route.c

```
105 void
106 rtfree(rt)
107 struct rtentry *rt;
108 {
109     struct ifaddr *ifa;

110     if (rt == 0)
111         panic("rtfree");
112     rt->rt_refcnt--;
113     if (rt->rt_refcnt <= 0 && (rt->rt_flags & RTF_UP) == 0) {
114         if (rt->rt_nodes->rn_flags & (RNF_ACTIVE | RNF_ROOT))
115             panic("rtfree 2");
116         rttrash--;
117         if (rt->rt_refcnt < 0) {
118             printf("rtfree: %x not freed (neg refs)\n", rt);
119             return;
120         }
121         ifa = rt->rt_ifa;
122         IFAFREE(ifa);
123         Free(rt_key(rt));
124         Free(rt);
125     }
126 }
```

route.c

Figure 19.5 `rtfree` function: release an `rtentry` structure.

105–115 The entry's reference count is decremented and if it is less than or equal to 0 and the route is not usable, the entry can be released. If either of the flags `RNF_ACTIVE` or `RNF_ROOT` are set, this is an internal error. If `RNF_ACTIVE` is set, this structure is still part of the routing table tree. If `RNF_ROOT` is set, this structure is one of the end markers built by `rn_inithead`.

116 `rttrash` is a debugging counter of the number of routing entries not in the routing tree, but not released. It is incremented by `rtrequest` when it begins deleting a route, and then decremented here. Its value should normally be 0.

Release interface reference

117–122 A check is made that the reference count is not negative, and then `IFAFREE` decrements the reference count for the `ifaddr` structure and releases it by calling `ifafree` when it reaches 0.

Release routing memory

123–124 The memory occupied by the routing entry key and its gateway is released. We'll see in `rt_setgate` that the memory for both is allocated in one contiguous chunk, allowing both to be released with a single call to `Free`. Finally the `rtentry` structure itself is released.

Routing Table Reference Counts

The handling of the routing table reference count, `rt_refcnt`, differs from most other reference counts. We see in Figure 18.2 that most routes have a reference count of 0, yet the routing table entries without any references are not deleted. We just saw the reason in `rtfree`: an entry with a reference count of 0 is not deleted unless the entry's `RTF_UP` flag is not set. The only time this flag is cleared is by `rtrequest` when a route is deleted from the routing tree.

Most routes are used in the following fashion.

- If the route is created automatically as a route to an interface when the interface is configured (which is typical for Ethernet interfaces, for example), then `rtinit` calls `rtrequest` with a command of `RTM_ADD`, creating the new entry and setting the reference count to 1. `rtinit` then decrements the reference count to 0 before returning.

  A point-to-point interface follows a similar procedure, so the route starts with a reference count of 0.

  If the route is created manually by the `route` command or by a routing daemon, a similar procedure occurs, with `route_output` calling `rtrequest` with a command of `RTM_ADD`, setting the reference count to 1. This is then decremented by `route_output` to 0 before it returns.

  Therefore all newly created routes start with a reference count of 0.

- When an IP datagram is sent on a socket, be it TCP or UDP, we saw that `ip_output` calls `rtalloc`, which calls `rtalloc1`. In Figure 19.3 we saw that the reference count is incremented by `rtalloc1` if the route is found.

  The located route is called a *held route*, since a pointer to the routing table entry is being held by the protocol, normally in a `route` structure contained within a protocol control block. An `rtentry` structure that is being held by someone else cannot be deleted, which is why `rtfree` doesn't release the structure until its reference count reaches 0.

- A protocol releases a held route by calling `RTFREE` or `rtfree`. We saw this in Figure 8.24 when `ip_output` detects a change in the destination address. We'll encounter it in Chapter 22 when a protocol control block that holds a route is released.

Part of the confusion we'll encounter in the code that follows is that `rtalloc1` is often called to look up a route in order to verify that a route to the destination exists, but when the caller doesn't want to hold the route. Since `rtalloc1` increments the counter, the caller immediately decrements it.

Consider a route being deleted by `rtrequest`. The `RTF_UP` flag is cleared, and if no one is holding the route (its reference count is 0), `rtfree` should be called. But `rtfree` considers it an error for the reference count to go below 0, so `rtrequest` checks whether its reference count is less than or equal to 0, and, if so, increments it and calls `rtfree`. Normally this sets the reference count to 1 and `rtfree` decrements it to 0 and deletes the route.

19.4 `rtrequest` Function

The `rtrequest` function is the focal point for adding and deleting routing table entries. Figure 19.6 shows some of the other functions that call it.

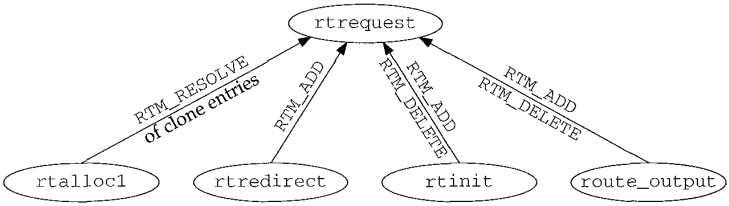

Figure 19.6 Summary of functions that call `rtrequest`.

`rtrequest` is a `switch` statement with one `case` per command: `RTM_ADD`, `RTM_DELETE`, and `RTM_RESOLVE`. Figure 19.7 shows the start of the function and the `RTM_DELETE` command.

route.c

```
int
rtrequest(req, dst, gateway, netmask, flags, ret_nrt)
int     req, flags;
struct sockaddr *dst, *gateway, *netmask;
struct rtentry **ret_nrt;
{
    int     s = splnet();
    int     error = 0;
    struct rtentry *rt;
    struct radix_node *rn;
    struct radix_node_head *rnh;
    struct ifaddr *ifa;
    struct sockaddr *ndst;
#define senderr(x) { error = x ; goto bad; }

    if ((rnh = rt_tables[dst->sa_family]) == 0)
        senderr(ESRCH);
    if (flags & RTF_HOST)
        netmask = 0;
```

```
308     switch (req) {
309     case RTM_DELETE:
310         if ((rn = rnh->rnh_deladdr(dst, netmask, rnh)) == 0)
311             senderr(ESRCH);
312         if (rn->rn_flags & (RNF_ACTIVE | RNF_ROOT))
313             panic("rtrequest delete");
314         rt = (struct rtentry *) rn;
315         rt->rt_flags &= ~RTF_UP;
316         if (rt->rt_gwroute) {
317             rt = rt->rt_gwroute;
318             RTFREE(rt);
319             (rt = (struct rtentry *) rn)->rt_gwroute = 0;
320         }
321         if ((ifa = rt->rt_ifa) && ifa->ifa_rtrequest)
322             ifa->ifa_rtrequest(RTM_DELETE, rt, SA(0));
323         rttrash++;
324         if (ret_nrt)
325             *ret_nrt = rt;
326         else if (rt->rt_refcnt <= 0) {
327             rt->rt_refcnt++;
328             rtfree(rt);
329         }
330         break;
```

route.c

Figure 19.7 `rtrequest` function: `RTM_DELETE` command.

290–307 The second argument, `dst`, is a socket address structure specifying the key to be added or deleted from the routing table. The `sa_family` from this key selects the routing table. If the `flags` argument indicates a host route (instead of a route to a network), the `netmask` pointer is set to null, ignoring any value the caller may have passed.

Delete from routing tree

309–315 The `rnh_deladdr` function (`rn_delete` from Figure 18.17) deletes the entry from the routing table tree and returns a pointer to the corresponding `rtentry` structure. The `RTF_UP` flag is cleared.

Remove reference to gateway routing table entry

316–320 If the entry is an indirect route through a gateway, `RTFREE` decrements the `rt_refcnt` member of the gateway's entry and deletes it if the count reaches 0. The `rt_gwroute` pointer is set to null and `rt` is set back to point to the entry that was deleted.

Call interface request function

321–322 If an `ifa_rtrequest` function is defined for this entry, that function is called. This function is used by ARP, for example, in Chapter 21 to delete the corresponding ARP entry.

Return pointer or release reference

323–330 The `rttrash` global is incremented because the entry may not be released in the code that follows. If the caller wants the pointer to the `rtentry` structure that was deleted from the routing tree (if `ret_nrt` is nonnull), then that pointer is returned, but the entry cannot be released: it is the caller's responsibility to call `rtfree` when it is finished with the entry. If `ret_nrt` is null, the entry can be released: if the reference count is less than or equal to 0, it is incremented, and `rtfree` is called. The `break` causes the function to return.

Figure 19.8 shows the next part of the function, which handles the `RTM_RESOLVE` command. This function is called with this command only from `rtalloc1`, when a new entry is to be created from an entry with the `RTF_CLONING` flag set.

route.c

```
331     case RTM_RESOLVE:
332         if (ret_nrt == 0 || (rt = *ret_nrt) == 0)
333             senderr(EINVAL);
334         ifa = rt->rt_ifa;
335         flags = rt->rt_flags & ~RTF_CLONING;
336         gateway = rt->rt_gateway;
337         if ((netmask = rt->rt_genmask) == 0)
338             flags |= RTF_HOST;
339         goto makeroute;
```

route.c

Figure 19.8 `rtrequest` function: `RTM_RESOLVE` command.

331–339 The final argument, `ret_nrt`, is used differently for this command: it contains the pointer to the entry with the `RTF_CLONING` flag set (Figure 19.2). The new entry will have the same `rt_ifa` pointer, the same flags (with the `RTF_CLONING` flag cleared), and the same `rt_gateway`. If the entry being cloned has a null `rt_genmask` pointer, the new entry has its `RTF_HOST` flag set, because it is a host route; otherwise the new entry is a network route and the network mask of the new entry is copied from the `rt_genmask` value. We give an example of cloned routes with a network mask at the end of this section. This `case` continues at the label `makeroute`, which is in the next figure.

Figure 19.9 shows the `RTM_ADD` command.

Locate corresponding interface

340–342 The function `ifa_ifwithroute` finds the appropriate local interface for the destination (`dst`), returning a pointer to its `ifaddr` structure.

Allocate memory for routing table entry

343–348 An `rtentry` structure is allocated. Recall that this structure contains both the two `radix_node` structures for the routing tree and the other routing information. The structure is zeroed and the `rt_flags` are set from the caller's flags, including the `RTF_UP` flag.

Allocate and copy gateway address

349–352 The `rt_setgate` function (Figure 19.11) allocates memory for both the routing table key (`dst`) and its `gateway`. It then copies `gateway` into the new memory and sets the pointers `rt_key`, `rt_gateway`, and `rt_gwroute`.

route.c

```
340     case RTM_ADD:
341         if ((ifa = ifa_ifwithroute(flags, dst, gateway)) == 0)
342             senderr(ENETUNREACH);

343       makeroute:
344         R_Malloc(rt, struct rtentry *, sizeof(*rt));
345         if (rt == 0)
346             senderr(ENOBUFS);
347         Bzero(rt, sizeof(*rt));
348         rt->rt_flags = RTF_UP | flags;
349         if (rt_setgate(rt, dst, gateway)) {
350             Free(rt);
351             senderr(ENOBUFS);
352         }
353         ndst = rt_key(rt);
354         if (netmask) {
355             rt_maskedcopy(dst, ndst, netmask);
356         } else
357             Bcopy(dst, ndst, dst->sa_len);

358         rn = rnh->rnh_addaddr((caddr_t) ndst, (caddr_t) netmask,
359                               rnh, rt->rt_nodes);
360         if (rn == 0) {
361             if (rt->rt_gwroute)
362                 rtfree(rt->rt_gwroute);
363             Free(rt_key(rt));
364             Free(rt);
365             senderr(EEXIST);
366         }
367         ifa->ifa_refcnt++;
368         rt->rt_ifa = ifa;
369         rt->rt_ifp = ifa->ifa_ifp;
370         if (req == RTM_RESOLVE)
371             rt->rt_rmx = (*ret_nrt)->rt_rmx;     /* copy metrics */
372         if (ifa->ifa_rtrequest)
373             ifa->ifa_rtrequest(req, rt, SA(ret_nrt ? *ret_nrt : 0));
374         if (ret_nrt) {
375             *ret_nrt = rt;
376             rt->rt_refcnt++;
377         }
378         break;
379     }
380   bad:
381     splx(s);
382     return (error);
383 }
```

route.c

Figure 19.9 `rtrequest` function: RTM_ADD command.

Copy destination address

353–357 The destination address (the routing table key `dst`) must now be copied into the memory pointed to by `rn_key`. If a network mask is supplied, `rt_maskedcopy` logically ANDs `dst` and `netmask`, forming the new key. Otherwise `dst` is copied into the new key. The reason for logically ANDing `dst` and `netmask` is to guarantee that the key in the table has already been ANDed with its mask, so when a search key is compared against the key in the table only the search key needs to be ANDed. For example, the following command adds another IP address (an alias) to the Ethernet interface `le0`, with subnet 12 instead of 13:

```
bsdi $ ifconfig le0 inet 140.252.12.63 netmask 0xffffffe0 alias
```

The problem is that we've incorrectly specified all one bits for the host ID. Nevertheless, when the key is stored in the routing table we can verify with `netstat` that the address is first logically ANDed with the mask:

```
Destination     Gateway            Flags     Refs      Use  Interface
140.252.12.32   link#1             U C         0         0  le0
```

Add entry to routing tree

358–366 The `rnh_addaddr` function (`rn_addroute` from Figure 18.17) adds this `rtentry` structure, with its destination and mask, to the routing table tree. If an error occurs, the structures are released and `EEXIST` returned (i.e., the entry is already in the routing table).

Store interface pointers

367–369 The `ifaddr` structure's reference count is incremented and the pointers to its `ifaddr` and `ifnet` structures are stored.

Copy metrics for newly cloned route

370–371 If the command was `RTM_RESOLVE` (not `RTM_ADD`), the entire metrics structure is copied from the cloned entry into the new entry. If the command was `RTM_ADD`, the caller can set the metrics after this function returns.

Call interface request function

372–373 If an `ifa_rtrequest` function is defined for this entry, that function is called. ARP uses this to perform additional processing for both the `RTM_ADD` and `RTM_RESOLVE` commands (Section 21.13).

Return pointer and increment reference count

374–378 If the caller wants a copy of the pointer to the new structure, it is returned through `ret_nrt` and the `rt_refcnt` reference count is incremented from 0 to 1.

Example: Cloned Routes with Network Masks

The only use of the `rt_genmask` value is with cloned routes created by the `RTM_RESOLVE` command in `rtrequest`. If an `rt_genmask` pointer is nonnull, then the socket address structure pointed to by this pointer becomes the network mask of the newly created route. In our routing table, Figure 18.2, the cloned routes are for the local Ethernet and for multicast addresses. The following example from [Sklower 1991] provides a different use of cloned routes. Another example is in Exercise 19.2.

Consider a class B network, say 128.1, that is behind a point-to-point link. The subnet mask is `0xffffff00`, the typical value that uses 8 bits for the subnet ID and 8 bits for the host ID. We need a routing table entry for all possible 254 subnets, with a gateway value of a router that is directly connected to our host and that knows how to reach the link to which the 128.1 network is connected.

The easiest solution, assuming the gateway router isn't our default router, is a single entry with a destination of 128.1.0.0 and a mask of `0xffff0000`. Assume, however, that the topology of the 128.1 network is such that each of the possible 254 subnets can have different operational characteristics: RTTs, MTUs, delays, and so on. If a separate routing table entry were used for each subnet, we would see that whenever a connection is closed, TCP would update the routing table entry with statistics about that route—its RTT, RTT variance, and so on (Figure 27.3). While we could create up to 254 entries by hand using the `route` command, one per subnet, a better solution is to use the *cloning* feature.

One entry is created by the system administrator with a destination of 128.1.0.0 and a network mask of `0xffff0000`. Additionally, the `RTF_CLONING` flag is set and the genmask is set to `0xffffff00`, which differs from the network mask. If the routing table is searched for 128.1.2.3, and an entry does not exist for the 128.1.2 subnet, the entry for 128.1 with the mask of `0xffff0000` is the best match. A new entry is created (since the `RTF_CLONING` flag is set) with a destination of 128.1.2 and a network mask of `0xffffff00` (the genmask value). The next time any host on this subnet is referenced, say 128.1.2.88, it will match this newly created entry.

19.5 `rt_setgate` Function

Each leaf in the routing tree has a key (`rt_key`, which is just the `rn_key` member of the `radix_node` structure contained at the beginning of the `rtentry` structure), and an associated gateway (`rt_gateway`). Both are socket address structures specified when the routing table entry is created. Memory is allocated for both structures by `rt_setgate`, as shown in Figure 19.10.

This example shows two of the entries from Figure 18.2, the ones with keys of 127.0.0.1 and 140.252.13.33. The former's gateway member points to an Internet socket address structure, while the latter's points to a data-link socket address structure that contains an Ethernet address. The former was entered into the routing table by the `route` system when the system was initialized, and the latter was created by ARP.

We purposely show the two structures pointed to by `rt_key` one right after the other, since they are allocated together by `rt_setgate`, which we show in Figure 19.11.

Set lengths from socket address structures

384–391 `dlen` is the length of the destination socket address structure, and `glen` is the length of the gateway socket address structure. The `ROUNDUP` macro rounds the value up to the next multiple of 4 bytes, but the size of most socket address structures is already a multiple of 4.

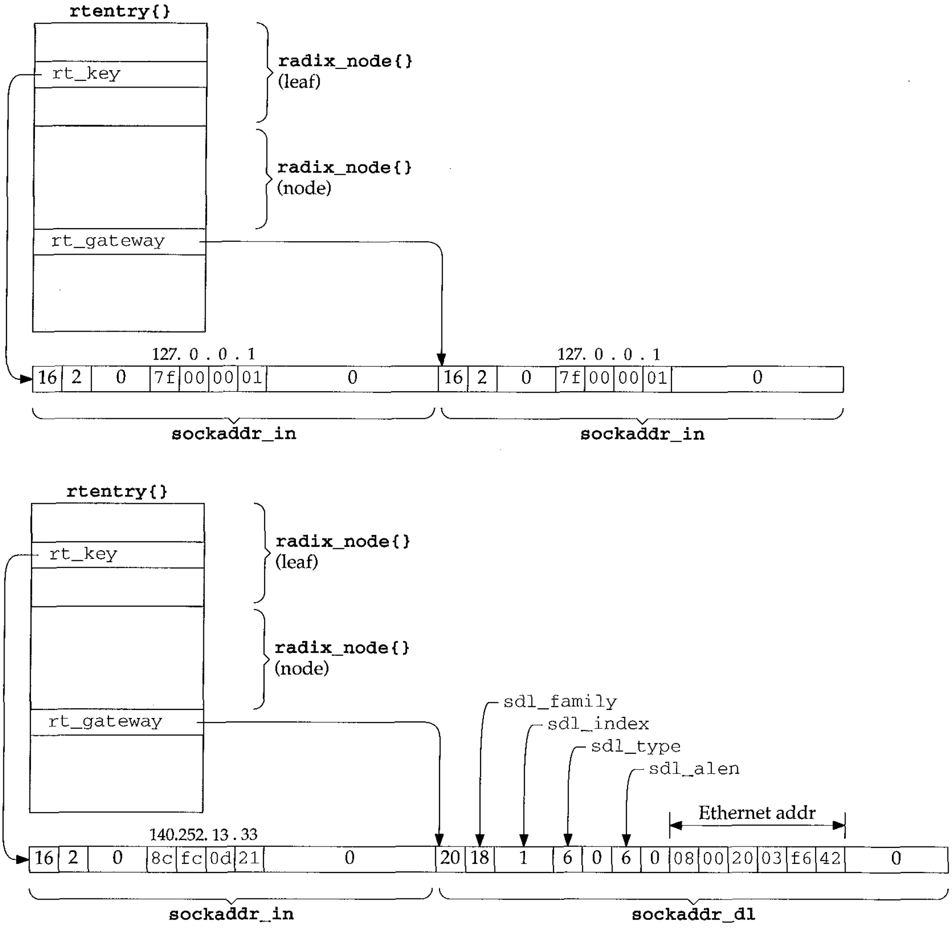

Figure 19.10 Example of routing table keys and associated gateways.

Allocate memory

392–397 If memory has not been allocated for this routing table key and gateway yet, or if `glen` is greater than the current size of the structure pointed to by `rt_gateway`, a new piece of memory is allocated and `rn_key` is set to point to the new memory.

Use memory already allocated for key and gateway

398–401 An adequately sized piece of memory is already allocated for the key and gateway, so `new` is set to point to this existing memory.

route.c

```
384 int
385 rt_setgate(rt0, dst, gate)
386 struct rtentry *rt0;
387 struct sockaddr *dst, *gate;
388 {
389     caddr_t new, old;
390     int     dlen = ROUNDUP(dst->sa_len), glen = ROUNDUP(gate->sa_len);
391     struct rtentry *rt = rt0;

392     if (rt->rt_gateway == 0 || glen > ROUNDUP(rt->rt_gateway->sa_len)) {
393         old = (caddr_t) rt_key(rt);
394         R_Malloc(new, caddr_t, dlen + glen);
395         if (new == 0)
396             return 1;
397         rt->rt_nodes->rn_key = new;
398     } else {
399         new = rt->rt_nodes->rn_key;
400         old = 0;
401     }
402     Bcopy(gate, (rt->rt_gateway = (struct sockaddr *) (new + dlen)), glen);
403     if (old) {
404         Bcopy(dst, new, dlen);
405         Free(old);
406     }
407     if (rt->rt_gwroute) {
408         rt = rt->rt_gwroute;
409         RTFREE(rt);
410         rt = rt0;
411         rt->rt_gwroute = 0;
412     }
413     if (rt->rt_flags & RTF_GATEWAY) {
414         rt->rt_gwroute = rtalloc1(gate, 1);
415     }
416     return 0;
417 }
```

route.c

Figure 19.11 `rt_setgate` function.

Copy new gateway

402 The new gateway structure is copied and `rt_gateway` is set to point to the socket address structure.

Copy key from old memory to new memory

403–406 If a new piece of memory was allocated, the routing table key (`dst`) is copied right before the gateway field that was just copied. The old piece of memory is released.

Release gateway routing pointer

407–412 If the routing table entry contains a nonnull `rt_gwroute` pointer, that structure is released by `RTFREE` and the `rt_gwroute` pointer is set to null.

Locate and store new gateway routing pointer

413–415 If the routing table entry is an indirect route, `rtalloc1` locates the entry for the new gateway, which is stored in `rt_gwroute`. If an invalid gateway is specified for an indirect route, an error is not returned by `rt_setgate`, but the `rt_gwroute` pointer will be null.

19.6 `rtinit` Function

There are four calls to `rtinit` from the Internet protocols to add or delete routes associated with interfaces.

- `in_control` calls `rtinit` twice when the destination address of a point-to-point interface is set (Figure 6.21). The first call specifies `RTM_DELETE` to delete any existing route to the destination; the second call specifies `RTM_ADD` to add the new route.
- `in_ifinit` calls `rtinit` to add a network route for a broadcast network or a host route for a point-to-point link (Figure 6.19). If the route is for an Ethernet interface, the `RTF_CLONING` flag is automatically set by `in_ifinit`.
- `in_ifscrub` calls `rtinit` to delete an existing route for an interface.

Figure 19.12 shows the first part of the `rtinit` function. The `cmd` argument is always `RTM_ADD` or `RTM_DELETE`.

Get destination address for route

452 If the route is to a host, the destination address is the other end of the point-to-point link. Otherwise we're dealing with a network route and the destination address is the unicast address of the interface (masked with `ifa_netmask`).

Mask network address with network mask

453–459 If a route is being deleted, the destination must be looked up in the routing table to locate its routing table entry. If the route being deleted is a network route and the interface has an associated network mask, an mbuf is allocated and the destination address is copied into the mbuf by `rt_maskedcopy`, logically ANDing the caller's address with the mask. `dst` is set to point to the masked copy in the mbuf, and that is the destination looked up in the next step.

Search for routing table entry

460–469 `rtalloc1` searches the routing table for the destination address. If the entry is found, its reference count is decremented (since `rtalloc1` incremented the reference count). If the pointer to the interface's `ifaddr` in the routing table does not equal the caller's argument, an error is returned.

Process request

470–473 `rtrequest` executes the command, either `RTM_ADD` or `RTM_DELETE`. When it returns, if an mbuf was allocated earlier, it is released.

route.c

```
int
rtinit(ifa, cmd, flags)
struct ifaddr *ifa;
int     cmd, flags;
{
    struct rtentry *rt;
    struct sockaddr *dst;
    struct sockaddr *deldst;
    struct mbuf *m = 0;
    struct rtentry *nrt = 0;
    int     error;

    dst = flags & RTF_HOST ? ifa->ifa_dstaddr : ifa->ifa_addr;
    if (cmd == RTM_DELETE) {
        if ((flags & RTF_HOST) == 0 && ifa->ifa_netmask) {
            m = m_get(M_WAIT, MT_SONAME);
            deldst = mtod(m, struct sockaddr *);
            rt_maskedcopy(dst, deldst, ifa->ifa_netmask);
            dst = deldst;
        }
        if (rt = rtalloc1(dst, 0)) {
            rt->rt_refcnt--;
            if (rt->rt_ifa != ifa) {
                if (m)
                    (void) m_free(m);
                return (flags & RTF_HOST ? EHOSTUNREACH
                        : ENETUNREACH);
            }
        }
    }
    error = rtrequest(cmd, dst, ifa->ifa_addr, ifa->ifa_netmask,
                      flags | ifa->ifa_flags, &nrt);
    if (m)
        (void) m_free(m);
```

route.c

Figure 19.12 `rtinit` function: call `rtrequest` to handle command.

Figure 19.13 shows the second half of `rtinit`.

Generate routing message on successful delete

474–480 If a route was deleted, and `rtrequest` returned 0 along with a pointer to the `rtentry` structure that was deleted (in `nrt`), a routing socket message is generated by `rt_newaddrmsg`. If the reference count is less than or equal to 0, it is incremented and the route is released by `rtfree`.

Successful add

481–482 If a route was added, and `rtrequest` returned 0 along with a pointer to the `rtentry` structure that was added (in `nrt`), the reference count is decremented (since `rtrequest` incremented it).

route.c

```
474     if (cmd == RTM_DELETE && error == 0 && (rt = nrt)) {
475         rt_newaddrmsg(cmd, ifa, error, nrt);
476         if (rt->rt_refcnt <= 0) {
477             rt->rt_refcnt++;
478             rtfree(rt);
479         }
480     }
481     if (cmd == RTM_ADD && error == 0 && (rt = nrt)) {
482         rt->rt_refcnt--;
483         if (rt->rt_ifa != ifa) {
484             printf("rtinit: wrong ifa (%x) was (%x)\n", ifa,
485                    rt->rt_ifa);
486             if (rt->rt_ifa->ifa_rtrequest)
487                 rt->rt_ifa->ifa_rtrequest(RTM_DELETE, rt, SA(0));
488             IFAFREE(rt->rt_ifa);
489             rt->rt_ifa = ifa;
490             rt->rt_ifp = ifa->ifa_ifp;
491             ifa->ifa_refcnt++;
492             if (ifa->ifa_rtrequest)
493                 ifa->ifa_rtrequest(RTM_ADD, rt, SA(0));
494         }
495         rt_newaddrmsg(cmd, ifa, error, nrt);
496     }
497     return (error);
498 }
```

route.c

Figure 19.13 `rtinit` function: second half.

Incorrect interface

483–494 If the pointer to the interface's `ifaddr` in the new routing table entry does not equal the caller's argument, an error occurred. Recall that `rtrequest` determines the `ifa` pointer that is stored in the new entry by calling `ifa_ifwithroute` (Figure 19.9). When this error occurs the following steps take place: an error message is output to the console, the `ifa_rtrequest` function is called (if defined) with a command of `RTM_DELETE`, the `ifaddr` structure is released, the `rt_ifa` pointer is set to the value specified by the caller, the interface reference count is incremented, and the new interface's `ifa_rtrequest` function (if defined) is called with a command of `RTM_ADD`.

Generate routing message

495 A routing socket message is generated by `rt_newaddrmsg` for the `RTM_ADD` command.

19.7 `rtredirect` Function

When an ICMP redirect is received, `icmp_input` calls `rtredirect` and then calls `pfctlinput` (Figure 11.27). This latter function calls `udp_ctlinput` and `tcp_ctlinput`, which go through all the UDP and TCP protocol control blocks. If the

PCB is connected to the foreign address that has been redirected, and if the PCB holds a route to that foreign address, the route is released by `rtfree`. The next time any of these control blocks is used to send an IP datagram to that foreign address, `rtalloc` will be called and the destination will be looked up in the routing table, possibly finding a new (redirected) route.

The purpose of `rtredirect`, the first half of which is shown in Figure 19.14, is to validate the information in the redirect, update the routing table immediately, and then generate a routing socket message.

route.c

```
int
rtredirect(dst, gateway, netmask, flags, src, rtp)
struct sockaddr *dst, *gateway, *netmask, *src;
int     flags;
struct rtentry **rtp;
{
    struct rtentry *rt;
    int     error = 0;
    short   *stat = 0;
    struct rt_addrinfo info;
    struct ifaddr *ifa;

    /* verify the gateway is directly reachable */
    if ((ifa = ifa_ifwithnet(gateway)) == 0) {
        error = ENETUNREACH;
        goto out;
    }
    rt = rtalloc1(dst, 0);
    /*
     * If the redirect isn't from our current router for this dst,
     * it's either old or wrong.  If it redirects us to ourselves,
     * we have a routing loop, perhaps as a result of an interface
     * going down recently.
     */
#define equal(a1, a2) (bcmp((caddr_t)(a1), (caddr_t)(a2), (a1)->sa_len) == 0)
    if (!(flags & RTF_DONE) && rt &&
        (!equal(src, rt->rt_gateway) || rt->rt_ifa != ifa))
        error = EINVAL;
    else if (ifa_ifwithaddr(gateway))
        error = EHOSTUNREACH;
    if (error)
        goto done;
    /*
     * Create a new entry if we just got back a wildcard entry
     * or if the lookup failed.  This is necessary for hosts
     * which use routing redirects generated by smart gateways
     * to dynamically build the routing tables.
     */
    if ((rt == 0) || (rt_mask(rt) && rt_mask(rt)->sa_len < 2))
        goto create;
```

route.c

Figure 19.14 `rtredirect` function: validate received redirect.

147–157 The arguments are `dst`, the destination IP address of the datagram that caused the redirect (HD in Figure 8.18); `gateway`, the IP address of the router to use as the new gateway field for the destination (R2 in Figure 8.18); `netmask`, which is a null pointer; `flags`, which is `RTF_GATEWAY` and `RTF_HOST`; `src`, the IP address of the router that sent the redirect (R1 in Figure 8.18); and `rtp`, which is a null pointer. We indicate that `netmask` and `rtp` are both null pointers when called by `icmp_input`, but these arguments might be nonnull when called from other protocols.

New gateway must be directly connected

158–162 The new gateway must be directly connected or the redirect is invalid.

Locate routing table entry for destination and validate redirect

163–177 `rtalloc1` searches the routing table for a route to the destination. The following conditions must all be true, or the redirect is invalid and an error is returned. Notice that `icmp_input` ignores any error return from `rtredirect`. ICMP does not generate an error in response to an invalid redirect—it just ignores it.

- the `RTF_DONE` flag must not be set;
- `rtalloc` must have located a routing table entry for `dst`;
- the address of the router that sent the redirect (`src`) must equal the current `rt_gateway` for the destination;
- the interface for the new gateway (the `ifa` returned by `ifa_ifwithnet`) must equal the current interface for the destination (`rt_ifa`), that is, the new gateway must be on the same network as the current gateway; and
- the new gateway cannot redirect this host to itself, that is, there cannot exist an attached interface with a unicast address or a broadcast address equal to `gateway`.

Must create a new route

178–185 If a route to the destination was not found, or if the routing table entry that was located is the default route, a new entry is created for the destination. As the comment indicates, a host with access to multiple routers can use this feature to learn of the correct router when the default is not correct. The test for finding the default route is whether the routing table entry has an associated mask and if the length field of the mask is less than 2, since the mask for the default route is `rn_zeros` (Figure 18.35).

Figure 19.15 shows the second half of this function.

Create new host route

186–195 If the current route to the destination is a network route and the redirect is a host redirect and not a network redirect, a new host route is created for the destination and the existing network route is left alone. We mentioned that the `flags` argument always specifies `RTF_HOST` since the Net/3 ICMP considers all received redirects as host redirects.

route.c

```
    /*
     * Don't listen to the redirect if it's
     * for a route to an interface.
     */
    if (rt->rt_flags & RTF_GATEWAY) {
        if (((rt->rt_flags & RTF_HOST) == 0) && (flags & RTF_HOST)) {
            /*
             * Changing from route to net => route to host.
             * Create new route, rather than smashing route to net.
             */
          create:
            flags |= RTF_GATEWAY | RTF_DYNAMIC;
            error = rtrequest((int) RTM_ADD, dst, gateway,
                              netmask, flags,
                              (struct rtentry **) 0);
            stat = &rtstat.rts_dynamic;
        } else {
            /*
             * Smash the current notion of the gateway to
             * this destination.  Should check about netmask!!!
             */
            rt->rt_flags |= RTF_MODIFIED;
            flags |= RTF_MODIFIED;
            stat = &rtstat.rts_newgateway;
            rt_setgate(rt, rt_key(rt), gateway);
        }
    } else
        error = EHOSTUNREACH;
  done:
    if (rt) {
        if (rtp && !error)
            *rtp = rt;
        else
            rtfree(rt);
    }
  out:
    if (error)
        rtstat.rts_badredirect++;
    else if (stat != NULL)
        (*stat)++;

    bzero((caddr_t) & info, sizeof(info));
    info.rti_info[RTAX_DST] = dst;
    info.rti_info[RTAX_GATEWAY] = gateway;
    info.rti_info[RTAX_NETMASK] = netmask;
    info.rti_info[RTAX_AUTHOR] = src;
    rt_missmsg(RTM_REDIRECT, &info, flags, error);
}
```

route.c

Figure 19.15 `rtredirect` function: second half.

Create route

196–201 `rtrequest` creates the new route, setting the `RTF_GATEWAY` and `RTF_DYNAMIC` flags. The `netmask` argument is a null pointer, since the new route is a host route with an implied mask of all one bits. `stat` points to a counter that is incremented later.

Modify existing host route

202–211 This code is executed when the current route to the destination is already a host route. A new entry is not created, but the existing entry is modified. The `RTF_MODIFIED` flag is set and `rt_setgate` changes the `rt_gateway` field of the routing table entry to the new gateway address.

Ignore if destination is directly connected

212–213 If the current route to the destination is a direct route (the `RTF_GATEWAY` flag is not set), it is a redirect for a destination that is already directly connected. `EHOSTUNREACH` is returned.

Return pointer and increment statistic

214–225 If a routing table entry was located, it is either returned (if `rtp` is nonnull and there were no errors) or released by `rtfree`. The appropriate statistic is incremented.

Generate routing message

226–232 An `rt_addrinfo` structure is cleared and a routing socket message is generated by `rt_missmsg`. This message is sent by `raw_input` to any processes interested in the redirect.

19.8 Routing Message Structures

Routing messages consist of a fixed-length header followed by up to eight socket address structures. The fixed-length header is one of the following three structures:

- `rt_msghdr`
- `if_msghdr`
- `ifa_msghdr`

Figure 18.11 provided an overview of which functions generated the different messages and Figure 18.9 showed which structure is used by each message type. The first three members of the three structures have the same data type and meaning: the message length, version, and type. This allows the receiver of the message to decode the message. Also, each structure has a member that encodes which of the eight potential socket address structures follow the structure (a bitmask): the `rtm_addrs`, `ifm_addrs`, and `ifam_addrs` members.

Figure 19.16 shows the most common of the structures, `rt_msghdr`. The `RTM_IFINFO` message uses an `if_msghdr` structure, shown in Figure 19.17. The `RTM_NEWADDR` and `RTM_DELADDR` messages use an `ifa_msghdr` structure, shown in Figure 19.18.

route.h

```
struct rt_msghdr {
    u_short rtm_msglen;          /* to skip over non-understood messages */
    u_char  rtm_version;         /* future binary compatibility */
    u_char  rtm_type;            /* message type */

    u_short rtm_index;           /* index for associated ifp */
    int     rtm_flags;           /* flags, incl. kern & message, e.g. DONE */
    int     rtm_addrs;           /* bitmask identifying sockaddrs in msg */
    pid_t   rtm_pid;             /* identify sender */
    int     rtm_seq;             /* for sender to identify action */
    int     rtm_errno;           /* why failed */
    int     rtm_use;             /* from rtentry */
    u_long  rtm_inits;           /* which metrics we are initializing */
    struct rt_metrics rtm_rmx;   /* metrics themselves */
};
```

route.h

Figure 19.16 `rt_msghdr` structure.

if.h

```
struct if_msghdr {
    u_short ifm_msglen;          /* to skip over non-understood messages */
    u_char  ifm_version;         /* future binary compatability */
    u_char  ifm_type;            /* message type */

    int     ifm_addrs;           /* like rtm_addrs */
    int     ifm_flags;           /* value of if_flags */
    u_short ifm_index;           /* index for associated ifp */
    struct if_data ifm_data;     /* statistics and other data about if */
};
```

if.h

Figure 19.17 `if_msghdr` structure.

if.h

```
struct ifa_msghdr {
    u_short ifam_msglen;         /* to skip over non-understood messages */
    u_char  ifam_version;        /* future binary compatability */
    u_char  ifam_type;           /* message type */

    int     ifam_addrs;          /* like rtm_addrs */
    int     ifam_flags;          /* value of ifa_flags */
    u_short ifam_index;          /* index for associated ifp */
    int     ifam_metric;         /* value of ifa_metric */
};
```

if.h

Figure 19.18 `ifa_msghdr` structure.

Note that the first three members across the three different structures have the same data types and meanings.

The three variables `rtm_addrs`, `ifm_addrs`, and `ifam_addrs` are bitmasks defining which socket address structures follow the header. Figure 19.19 shows the constants used with these bitmasks.

Bitmask		Array index		Name in `rtsock.c`	Description
Constant	Value	Constant	Value		
RTA_DST	`0x01`	*RTAX_DST*	0	`dst`	destination socket address structure
RTA_GATEWAY	`0x02`	*RTAX_GATEWAY*	1	`gate`	gateway socket address structure
RTA_NETMASK	`0x04`	*RTAX_NETMASK*	2	`netmask`	netmask socket address structure
RTA_GENMASK	`0x08`	*RTAX_GENMASK*	3	`genmask`	cloning mask socket address structure
RTA_IFP	`0x10`	*RTAX_IFP*	4	`ifpaddr`	interface name socket address structure
RTA_IFA	`0x20`	*RTAX_IFA*	5	`ifaaddr`	interface address socket address structure
RTA_AUTHOR	`0x40`	*RTAX_AUTHOR*	6		socket address structure for author of redirect
RTA_BRD	`0x80`	*RTAX_BRD*	7	`brdaddr`	broadcast or point-to-point destination address
		RTAX_MAX	8		#elements in an `rti_info[]` array

Figure 19.19 Constants used to refer to members of `rti_info` array.

The bitmask value is always the constant 1 left shifted by the number of bits specified by the array index. For example, `0x20` (`RTA_IFA`) is 1 left shifted by five bits (`RTAX_IFA`). We'll see this fact used in the code.

The socket address structures that are present always occur in order of increasing array index, one right after the other. For example, if the bitmask is `0x87`, the first socket address structure contains the destination, followed by the gateway, followed by the network mask, followed by the broadcast address.

The array indexes in Figure 19.19 are used within the kernel to refer to its `rt_addrinfo` structure, shown in Figure 19.20. This structure holds the same bitmask that we described, indicating which addresses are present, and pointers to those socket address structures.

route.h

```
struct rt_addrinfo {
    int     rti_addrs;          /* bitmask, same as rtm_addrs */
    struct sockaddr *rti_info[RTAX_MAX];
};
```

route.h

Figure 19.20 `rt_addrinfo` structure: encode which addresses are present and pointers to them.

For example, if the `RTA_GATEWAY` bit is set in the `rti_addrs` member, then the member `rti_info[RTAX_GATEWAY]` is a pointer to a socket address structure containing the gateway's address. In the case of the Internet protocols, the socket address structure is a `sockaddr_in` containing the gateway's IP address.

The fifth column in Figure 19.19 shows the names used for the corresponding members of an `rti_info` array throughout the file `rtsock.c`. These definitions look like

```
#define dst   info.rti_info[RTAX_DST]
```

We'll encounter these names in many of the source files later in this chapter. The `RTAX_AUTHOR` element is not assigned a name because it is never passed from a process to the kernel.

We've already encountered this `rt_addrinfo` structure twice: in `rtalloc1` (Figure 19.2) and `rtredirect` (Figure 19.14). Figure 19.21 shows the format of this structure when built by `rtalloc1`, after a routing table lookup fails, when `rt_missmsg` is called.

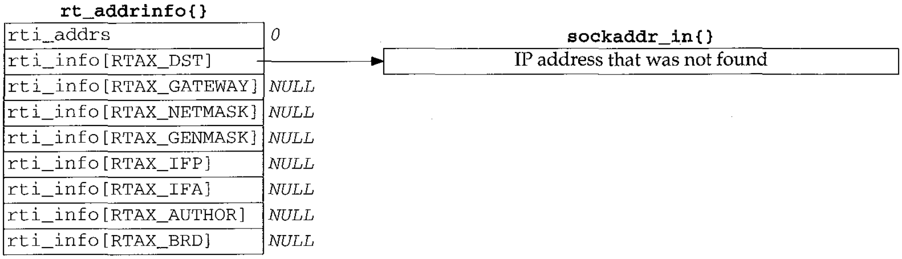

Figure 19.21 `rt_addrinfo` structure passed by `rtalloc1` to `rt_missmsg`.

All the unused pointers are null because the structure is set to 0 before it is used. Also note that the `rti_addrs` member is not initialized with the appropriate bitmask because when this structure is used within the kernel, a null pointer in the `rti_info` array indicates a nonexistent socket address structure. The bitmask is needed only for messages between a process and the kernel.

Figure 19.22 shows the format of the structure built by `rtredirect` when it calls `rt_missmsg`.

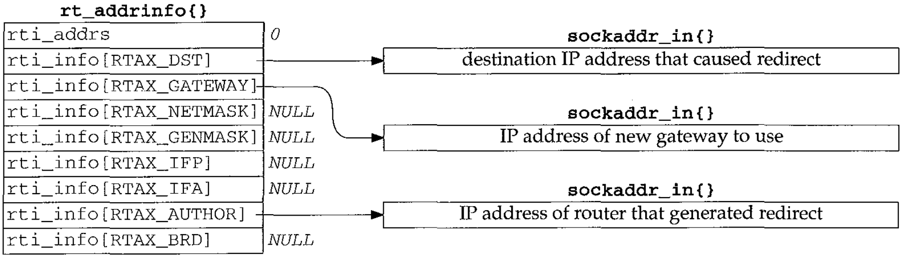

Figure 19.22 `rt_addrinfo` structure passed by `rtredirect` to `rt_missmsg`.

The following sections show how these structures are placed into the messages sent to a process.

Figure 19.23 shows the `route_cb` structure, which we'll encounter in the following sections. It contains four counters; one each for the IP, XNS, and OSI protocols, and an "any" counter. Each counter is the number of routing sockets currently in existence for that domain.

203–208 By keeping track of the number of routing socket listeners, the kernel avoids building a routing message and calling `raw_input` to send the message when there aren't any processes waiting for a message.

route.h

```
203 struct route_cb {
204     int     ip_count;           /* IP */
205     int     ns_count;           /* XNS */
206     int     iso_count;          /* ISO */
207     int     any_count;          /* sum of above three counters */
208 };
```

route.h

Figure 19.23 `route_cb` structure: counters of routing socket listeners.

19.9 `rt_missmsg` Function

The function `rt_missmsg`, shown in Figure 19.24, takes the structures shown in Figures 19.21 and 19.22, calls `rt_msg1` to build a corresponding variable-length message for a process in an mbuf chain, and then calls `raw_input` to pass the mbuf chain to all appropriate routing sockets.

rtsock.c

```
516 void
517 rt_missmsg(type, rtinfo, flags, error)
518 int     type, flags, error;
519 struct rt_addrinfo *rtinfo;
520 {
521     struct rt_msghdr *rtm;
522     struct mbuf *m;
523     struct sockaddr *sa = rtinfo->rti_info[RTAX_DST];

524     if (route_cb.any_count == 0)
525         return;

526     m = rt_msg1(type, rtinfo);
527     if (m == 0)
528         return;

529     rtm = mtod(m, struct rt_msghdr *);
530     rtm->rtm_flags = RTF_DONE | flags;
531     rtm->rtm_errno = error;
532     rtm->rtm_addrs = rtinfo->rti_addrs;

533     route_proto.sp_protocol = sa ? sa->sa_family : 0;
534     raw_input(m, &route_proto, &route_src, &route_dst);
535 }
```

rtsock.c

Figure 19.24 `rt_missmsg` function.

516–525 If there aren't any routing socket listeners, the function returns immediately.

Build message in mbuf chain

526–528 `rt_msg1` (Section 19.12) builds the appropriate message in an mbuf chain, and returns the pointer to the chain. Figure 19.25 shows an example of the resulting mbuf chain, using the `rt_addrinfo` structure from Figure 19.22. The information needs to be in an mbuf chain because `raw_input` calls `sbappendaddr` to append the mbuf chain to a socket's receive buffer.

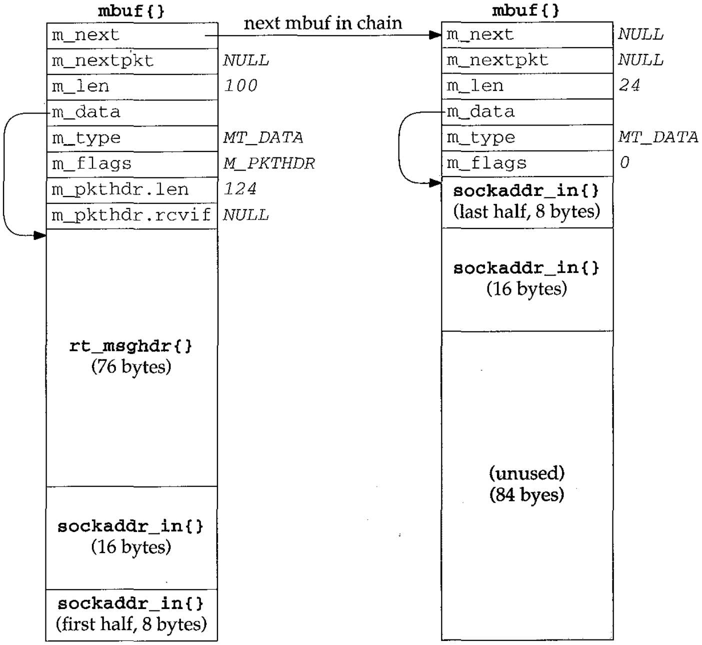

Figure 19.25 Mbuf chain built by `rt_msg1` corresponding to Figure 19.22.

Finish building message

529–532 The two members `rtm_flags` and `rtm_errno` are set to the values passed by the caller. The `rtm_addrs` member is copied from the `rti_addrs` value. We showed this value as 0 in Figures 19.21 and 19.22, but `rt_msg1` calculates and stores the appropriate bitmask, based on which pointers in the `rti_info` array are nonnull.

Set protocol of message, call `raw_input`

533–534 The final three arguments to `raw_input` specify the protocol, source, and destination of the routing message. These three structures are initialized as

```
struct  sockaddr  route_dst = { 2, PF_ROUTE, };
struct  sockaddr  route_src = { 2, PF_ROUTE, };
struct  sockproto route_proto = { PF_ROUTE, };
```

The first two structures are never modified by the kernel. The `sockproto` structure, shown in Figure 19.26, is one we haven't seen before.

socket.h

```
128 struct sockproto {
129     u_short sp_family;          /* address family */
130     u_short sp_protocol;        /* protocol */
131 };
```

socket.h

Figure 19.26 `sockproto` structure.

The family is never changed from its initial value of `PF_ROUTE`, but the protocol is set each time `raw_input` is called. When a process creates a routing socket by calling `socket`, the third argument (the protocol) specifies the protocol in which the process is interested. The caller of `raw_input` sets the `sp_protocol` member of the `route_proto` structure to the protocol of the routing message. In the case of `rt_missmsg`, it is set to the `sa_family` of the destination socket address structure (if specified by the caller), which in Figures 19.21 and 19.22 would be `AF_INET`.

19.10 `rt_ifmsg` Function

In Figure 4.30 we saw that `if_up` and `if_down` both call `rt_ifmsg`, shown in Figure 19.27, to generate a routing socket message when an interface goes up or down.

rtsock.c

```
540 void
541 rt_ifmsg(ifp)
542 struct ifnet *ifp;
543 {
544     struct if_msghdr *ifm;
545     struct mbuf *m;
546     struct rt_addrinfo info;

547     if (route_cb.any_count == 0)
548         return;

549     bzero((caddr_t) & info, sizeof(info));
550     m = rt_msg1(RTM_IFINFO, &info);
551     if (m == 0)
552         return;

553     ifm = mtod(m, struct if_msghdr *);
554     ifm->ifm_index = ifp->if_index;
555     ifm->ifm_flags = ifp->if_flags;
556     ifm->ifm_data = ifp->if_data;   /* structure assignment */
557     ifm->ifm_addrs = 0;

558     route_proto.sp_protocol = 0;
559     raw_input(m, &route_proto, &route_src, &route_dst);
560 }
```

rtsock.c

Figure 19.27 `rt_ifmsg` function.

547–548 If there aren't any routing socket listeners, the function returns immediately.

Build message in mbuf chain

549–552 An `rt_addrinfo` structure is set to 0 and `rt_msg1` builds an appropriate message in an mbuf chain. Notice that all socket address pointers in the `rt_addrinfo` structure are null, so only the fixed-length `if_msghdr` structure becomes the routing message; there are no addresses.

Finish building message

553–557 The interface's index, flags, and `if_data` structure are copied into the message in the mbuf and the `ifm_addrs` bitmask is set to 0.

Set protocol of message, call `raw_input`

558–559 The protocol of the routing message is set to 0 because this message can apply to all protocol suites. It is a message about an interface, not about some specific destination. `raw_input` delivers the message to the appropriate listeners.

19.11 `rt_newaddrmsg` Function

In Figure 19.13 we saw that `rtinit` calls `rt_newaddrmsg` with a command of `RTM_ADD` or `RTM_DELETE` when an interface has an address added or deleted. Figure 19.28 shows the first half of the function.

rtsock.c

```
void
rt_newaddrmsg(cmd, ifa, error, rt)
int     cmd, error;
struct ifaddr *ifa;
struct rtentry *rt;
{
    struct rt_addrinfo info;
    struct sockaddr *sa;
    int     pass;
    struct mbuf *m;
    struct ifnet *ifp = ifa->ifa_ifp;

    if (route_cb.any_count == 0)
        return;

    for (pass = 1; pass < 3; pass++) {
        bzero((caddr_t) & info, sizeof(info));
        if ((cmd == RTM_ADD && pass == 1) ||
            (cmd == RTM_DELETE && pass == 2)) {
            struct ifa_msghdr *ifam;
            int     ncmd = cmd == RTM_ADD ? RTM_NEWADDR : RTM_DELADDR;

            ifaaddr = sa = ifa->ifa_addr;
            ifpaddr = ifp->if_addrlist->ifa_addr;
            netmask = ifa->ifa_netmask;
            brdaddr = ifa->ifa_dstaddr;
            if ((m = rt_msg1(ncmd, &info)) == NULL)
                continue;
            ifam = mtod(m, struct ifa_msghdr *);
            ifam->ifam_index = ifp->if_index;
            ifam->ifam_metric = ifa->ifa_metric;
            ifam->ifam_flags = ifa->ifa_flags;
            ifam->ifam_addrs = info.rti_addrs;
        }
```

rtsock.c

Figure 19.28 `rt_newaddrmsg` function: first half: create `ifa_msghdr` message.

580–581 If there aren't any routing socket listeners, the function returns immediately.

Generate two routing messages

582 The `for` loop iterates twice because two messages are generated. If the command is `RTM_ADD`, the first message is of type `RTM_NEWADDR` and the second message is of type `RTM_ADD`. If the command is `RTM_DELETE`, the first message is of type `RTM_DELETE` and the second message is of type `RTM_DELADDR`. The `RTM_NEWADDR` and `RTM_DELADDR` messages are built from an `ifa_msghdr` structure, while the `RTM_ADD` and `RTM_DELETE` messages are built from an `rt_msghdr` structure. The function generates two messages because one message provides information about the interface and the other about the addresses.

583 An `rt_addrinfo` structure is set to 0.

Generate message with up to four addresses

588–591 Pointers to four socket address structures containing information about the interface address that has been added or deleted are stored in the `rti_info` array. Recall from Figure 19.19 that `ifaaddr`, `ifpaddr`, `netmask`, and `brdaddr` reference elements in the `rti_info` array named `info`. `rt_msg1` builds the appropriate message in an mbuf chain. Notice that `sa` is set to point to the `ifa_addr` structure, and we'll see at the end of the function that the family of this socket address structure becomes the protocol of the routing message.

Remaining members of the `ifa_msghdr` structure are filled in with the interface's index, metric, and flags, along with the bitmask set by `rt_msg1`.

Figure 19.29 shows the second half of `rt_newaddrmsg`, which creates an `rt_msghdr` message with information about the routing table entry that was added or deleted.

Build message

600–609 Pointers to three socket address structures are stored in the `rti_info` array: the `rt_mask`, `rt_key`, and `rt_gateway` structures. `sa` is set to point to the destination address, and its family becomes the protocol of the routing message. `rt_msg1` builds the appropriate message in an mbuf chain.

Additional fields in the `rt_msghdr` structure are filled in, including the bitmask set by `rt_msg1`.

Set protocol of message, call `raw_input`

616–619 The protocol of the routing message is set and `raw_input` passes the message to the appropriate listeners. The function returns after two iterations through the loop.

rtsock.c

```
        if ((cmd == RTM_ADD && pass == 2) ||
            (cmd == RTM_DELETE && pass == 1)) {
            struct rt_msghdr *rtm;

            if (rt == 0)
                continue;
            netmask = rt_mask(rt);
            dst = sa = rt_key(rt);
            gate = rt->rt_gateway;
            if ((m = rt_msg1(cmd, &info)) == NULL)
                continue;
            rtm = mtod(m, struct rt_msghdr *);
            rtm->rtm_index = ifp->if_index;
            rtm->rtm_flags |= rt->rt_flags;
            rtm->rtm_errno = error;
            rtm->rtm_addrs = info.rti_addrs;
        }
        route_proto.sp_protocol = sa ? sa->sa_family : 0;
        raw_input(m, &route_proto, &route_src, &route_dst);
    }
}
```

rtsock.c

Figure 19.29 `rt_newaddrmsg` function: second half, create `rt_msghdr` message.

19.12 `rt_msg1` Function

The functions described in the previous three sections each called `rt_msg1` to build the appropriate routing message. In Figure 19.25 we showed the mbuf chain that was built by `rt_msg1` from the `rt_msghdr` and `rt_addrinfo` structures in Figure 19.22. Figure 19.30 shows the function.

Get mbuf and determine fixed size of message

399–422 An mbuf with a packet header is obtained and the length of the fixed-size message is stored in `len`. Two of the message types in Figure 18.9 use an `ifa_msghdr` structure, one uses an `if_msghdr` structure, and the remaining nine use an `rt_msghdr` structure.

Verify structure fits in mbuf

423–424 The size of the fixed-length structure must fit entirely within the data portion of the packet header mbuf, because the mbuf pointer is cast to a structure pointer using `mtod` and the structure is then referenced through the pointer. The largest of the three structures is `if_msghdr`, which at 84 bytes is less than `MHLEN` (100).

Initialize mbuf packet header and zero structure

425–428 The two fields in the packet header are initialized and the structure in the mbuf is set to 0.

rtsock.c

```
static struct mbuf *
rt_msg1(type, rtinfo)
int     type;
struct rt_addrinfo *rtinfo;
{
    struct rt_msghdr *rtm;
    struct mbuf *m;
    int     i;
    struct sockaddr *sa;
    int     len, dlen;

    m = m_gethdr(M_DONTWAIT, MT_DATA);
    if (m == 0)
        return (m);
    switch (type) {

    case RTM_DELADDR:
    case RTM_NEWADDR:
        len = sizeof(struct ifa_msghdr);
        break;

    case RTM_IFINFO:
        len = sizeof(struct if_msghdr);
        break;

    default:
        len = sizeof(struct rt_msghdr);
    }
    if (len > MHLEN)
        panic("rt_msg1");
    m->m_pkthdr.len = m->m_len = len;
    m->m_pkthdr.rcvif = 0;
    rtm = mtod(m, struct rt_msghdr *);
    bzero((caddr_t) rtm, len);

    for (i = 0; i < RTAX_MAX; i++) {
        if ((sa = rtinfo->rti_info[i]) == NULL)
            continue;
        rtinfo->rti_addrs |= (1 << i);
        dlen = ROUNDUP(sa->sa_len);
        m_copyback(m, len, dlen, (caddr_t) sa);
        len += dlen;
    }
    if (m->m_pkthdr.len != len) {
        m_freem(m);
        return (NULL);
    }
    rtm->rtm_msglen = len;
    rtm->rtm_version = RTM_VERSION;
    rtm->rtm_type = type;
    return (m);
}
```

rtsock.c

Figure 19.30 `rt_msg1` function: obtain and initialize mbuf.

Copy socket address structures into mbuf chain

429–436 The caller passes a pointer to an `rt_addrinfo` structure. The socket address structures corresponding to all the nonnull pointers in the `rti_info` are copied into the mbuf by `m_copyback`. The value 1 is left shifted by the `RTAX_`*xxx* index to generate the corresponding `RTA_`*xxx* bitmask (Figure 19.19), and each individual bitmask is logically ORed into the `rti_addrs` member, which the caller can store on return into the corresponding member of the message structure. The `ROUNDUP` macro rounds the size of each socket address structure up to the next multiple of 4 bytes.

437–440 If, when the loop terminates, the length in the mbuf packet header does not equal `len`, the function `m_copyback` wasn't able to obtain a required mbuf.

Store length, version, and type

441–445 The length, version, and message type are stored in the first three members of the message structure. Again, all three *xxx*`_msghdr` structures start with the same three members, so this code works with all three structures even though the pointer `rtm` is a pointer to an `rt_msghdr` structure.

19.13 `rt_msg2` Function

`rt_msg1` constructs a routing message in an mbuf chain, and the three functions that called it then called `raw_input` to append the mbuf chain to one or more socket's receive buffer. `rt_msg2` is different—it builds a routing message in a memory buffer, not an mbuf chain, and has an argument to a `walkarg` structure that is used when `rt_msg2` is called by the two functions that handle the `sysctl` system call for the routing domain. `rt_msg2` is called in two different scenarios:

1. from `route_output` to process the `RTM_GET` command, and
2. from `sysctl_dumpentry` and `sysctl_iflist` to process a `sysctl` system call.

Before looking at `rt_msg2`, Figure 19.31 shows the `walkarg` structure that is used in scenario 2. We go through all these members as we encounter them.

rtsock.c

```
struct walkarg {
    int     w_op;               /* NET_RT_xxx */
    int     w_arg;              /* RTF_xxx for FLAGS, if_index for IFLIST */
    int     w_given;            /* size of process' buffer */
    int     w_needed;           /* #bytes actually needed (at end) */
    int     w_tmemsize;         /* size of buffer pointed to by w_tmem */
    caddr_t w_where;            /* ptr to process' buffer (maybe null) */
    caddr_t w_tmem;             /* ptr to our malloc'ed buffer */
};
```

rtsock.c

Figure 19.31 `walkarg` structure: used with the `sysctl` system call in the routing domain.

Figure 19.32 shows the first half of the `rt_msg2` function. This portion is similar to the first half of `rt_msg1`.

rtsock.c

```
446 static int
447 rt_msg2(type, rtinfo, cp, w)
448 int     type;
449 struct rt_addrinfo *rtinfo;
450 caddr_t cp;
451 struct walkarg *w;
452 {
453     int     i;
454     int     len, dlen, second_time = 0;
455     caddr_t cp0;

456     rtinfo->rti_addrs = 0;
457   again:
458     switch (type) {

459     case RTM_DELADDR:
460     case RTM_NEWADDR:
461         len = sizeof(struct ifa_msghdr);
462         break;

463     case RTM_IFINFO:
464         len = sizeof(struct if_msghdr);
465         break;

466     default:
467         len = sizeof(struct rt_msghdr);
468     }
469     if (cp0 = cp)
470         cp += len;
471     for (i = 0; i < RTAX_MAX; i++) {
472         struct sockaddr *sa;

473         if ((sa = rtinfo->rti_info[i]) == 0)
474             continue;
475         rtinfo->rti_addrs |= (1 << i);
476         dlen = ROUNDUP(sa->sa_len);
477         if (cp) {
478             bcopy((caddr_t) sa, cp, (unsigned) dlen);
479             cp += dlen;
480         }
481         len += dlen;
482     }
```

rtsock.c

Figure 19.32 `rt_msg2` function: copy socket address structures.

446–455 Since this function stores the resulting message in a memory buffer, the caller specifies the start of that buffer in the `cp` argument. It is the caller's responsibility to ensure that the buffer is large enough for the message that is generated. To help the caller determine this size, if the `cp` argument is null, `rt_msg2` doesn't store anything but processes the input and returns the total number of bytes required to hold the result. We'll see that `route_output` uses this feature and calls this function twice: first to determine the size and then to store the result, after allocating a buffer of the correct size. When `rt_msg2` is called by `route_output`, the final argument is null. This final argument is nonnull when called as part of the `sysctl` system call processing.

Determine size of structure

458–470 The size of the fixed-length message structure is set based on the message type. If the `cp` pointer is nonnull, it is incremented by this size.

Copy socket address structures

471–482 The `for` loop goes through the `rti_info` array, and for each element that is a nonnull pointer it sets the appropriate bit in the `rti_addrs` bitmask, copies the socket address structure (if `cp` is nonnull), and updates the length.

Figure 19.33 shows the second half of `rt_msg2`, most of which handles the optional `walkarg` structure.

rtsock.c

```
483     if (cp == 0 && w != NULL && !second_time) {
484         struct walkarg *rw = w;

485         rw->w_needed += len;
486         if (rw->w_needed <= 0 && rw->w_where) {
487             if (rw->w_tmemsize < len) {
488                 if (rw->w_tmem)
489                     free(rw->w_tmem, M_RTABLE);
490                 if (rw->w_tmem = (caddr_t)
491                     malloc(len, M_RTABLE, M_NOWAIT))
492                     rw->w_tmemsize = len;
493             }
494             if (rw->w_tmem) {
495                 cp = rw->w_tmem;
496                 second_time = 1;
497                 goto again;
498             } else
499                 rw->w_where = 0;
500         }
501     }
502     if (cp) {
503         struct rt_msghdr *rtm = (struct rt_msghdr *) cp0;

504         rtm->rtm_version = RTM_VERSION;
505         rtm->rtm_type = type;
506         rtm->rtm_msglen = len;
507     }
508     return (len);
509 }
```

rtsock.c

Figure 19.33 `rt_msg2` function: handle optional `walkarg` argument.

483–484 This `if` statement is true only when a pointer to a `walkarg` structure was passed and this is the first loop through the function. The variable `second_time` was initialized to 0 but can be set to 1 within this `if` statement, and a jump made back to the label `again` in Figure 19.32. The test for `cp` being a null pointer is superfluous since whenever the `w` pointer is nonnull, the `cp` pointer is null, and vice versa.

Check if data to be stored

485–486 `w_needed` is incremented by the size of the message. This variable is initialized to 0 minus the size of the user's buffer to the `sysctl` function. For example, if the buffer size is 500 bytes, w_needed is initialized to –500. As long as it remains negative, there is room in the buffer. w_where is a pointer to the buffer in the calling process. It is null if the process doesn't want the result—the process just wants sysctl to return the size of the result, so the process can allocate a buffer and call sysctl again. rt_msg2 doesn't copy the data back to the process—that is up to the caller—but if the w_where pointer is null, there's no need for rt_msg2 to malloc a buffer to hold the result and loop back through the function again, storing the result in this buffer. There are really five different scenarios that this function handles, summarized in Figure 19.34.

called from	cp	w	w.w_where	second_time	Description
route_output	null	null			wants return length
	nonnull	null			wants result
sysctl_rtable	null	nonnull	null	0	process wants return length
	null	nonnull	nonnull	0	first time around to calculate length
	nonnull	nonnull	nonnull	1	second time around to store result

Figure 19.34 Summary of different scenarios for rt_msg2.

Allocate buffer first time or if message length increases

487–493 w_tmemsize is the size of the buffer pointed to by w_tmem. It is initialized to 0 by sysctl_rtable, so the first time rt_msg2 is called for a given sysctl request, the buffer must be allocated. Also, if the size of the result increases, the existing buffer must be released and a new (larger) buffer allocated.

Go around again and store result

494–499 If w_tmem is nonnull, a buffer already exists or one was just allocated. cp is set to point to this buffer, second_time is set to 1, and a jump is made to again. The if statement at the beginning of this figure won't be true during this second pass, since second_time is now 1. If w_tmem is null, the call to malloc failed, so the pointer to the buffer in the process is set to null, preventing anything from being returned.

Store length, version, and type

502–509 If cp is nonnull, the first three elements of the message header are stored. The function returns the length of the message.

19.14 sysctl_rtable Function

This function handles the sysctl system call on a routing socket. It is called by net_sysctl as shown in Figure 18.11.

Before going through the source code, Figure 19.35 shows the typical use of this system call with respect to the routing table. This example is from the arp program.

The first three elements in the mib array cause the kernel to call sysctl_rtable to process the remaining elements.

```
int      mib[6];
size_t   needed;
char     *buf, *lim, *next;
struct rt_msghdr  *rtm;

mib[0] = CTL_NET;
mib[1] = PF_ROUTE;
mib[2] = 0;
mib[3] = AF_INET;          /* address family; can be 0 */
mib[4] = NET_RT_FLAGS;     /* operation */
mib[5] = RTF_LLINFO;       /* flags; can be 0 */

if (sysctl(mib, 6, NULL, &needed, NULL, 0) < 0)
    quit("sysctl error, estimate");

if ( (buf = malloc(needed)) == NULL)
    quit("malloc");

if (sysctl(mib, 6, buf, &needed, NULL, 0) < 0)
    quit("sysctl error, retrieval");

lim = buf + needed;
for (next = buf; next < lim; next += rtm->rtm_msglen) {
    rtm = (struct rt_msghdr *)next;
    ...  /* do whatever */
}
```

Figure 19.35 Example of `sysctl` with routing table.

`mib[4]` specifies the operation. Three operations are supported.

1. `NET_RT_DUMP`: return the routing table corresponding to the address family specified by `mib[3]`. If the address family is 0, all routing tables are returned.

   An `RTM_GET` routing message is returned for each routing table entry containing two, three, or four socket address structures per message: those addresses pointed to by `rt_key`, `rt_gateway`, `rt_netmask`, and `rt_genmask`. The final two pointers might be null.

2. `NET_RT_FLAGS`: the same as the previous command except `mib[5]` specifies an `RTF_`*xxx* flag (Figure 18.25), and only entries with this flag set are returned.

3. `NET_RT_IFLIST`: return information on all the configured interfaces. If the `mib[5]` value is nonzero it specifies an interface index and only the interface with the corresponding `if_index` is returned. Otherwise all interfaces on the `ifnet` linked list are returned.

   For each interface one `RTM_IFINFO` message is returned, with information about the interface itself, followed by one `RTM_NEWADDR` message for each `ifaddr` structure on the interface's `if_addrlist` linked list. If the `mib[3]` value is nonzero, `RTM_NEWADDR` messages are returned for only the addresses with an address family that matches the mib[3] value. Otherwise mib[3] is 0 and information on all addresses is returned.

This operation is intended to replace the SIOCGIFCONF ioctl (Figure 4.26).

One problem with this system call is that the amount of information returned can vary, depending on the number of routing table entries or the number of interfaces. Therefore the first call to sysctl typically specifies a null pointer as the third argument, which means: don't return any data, just return the number of bytes of return information. As we see in Figure 19.35, the process then calls malloc, followed by sysctl to fetch the information. This second call to sysctl again returns the number of bytes through the fourth argument (which might have changed since the previous call), and this value provides the pointer lim that points just beyond the final byte of data that was returned. The process then steps through the routing messages in the buffer, using the rtm_msglen member to step to the next message.

Figure 19.36 shows the values for these six mib variables that various Net/3 programs specify to access the routing table and interface list.

mib[]	arp	route	netstat	routed	gated	rwhod
0	CTL_NET	CTL_NET	CTL_NET	CTL_NET	CTL_NET	CTL_NET
1	PF_ROUTE	PF_ROUTE	PF_ROUTE	PF_ROUTE	PF_ROUTE	PF_ROUTE
2	0	0	0	0	0	0
3	AF_INET	0	0	AF_INET	0	AF_INET
4	NET_RT_FLAGS	NET_RT_DUMP	NET_RT_DUMP	NET_RT_IFLIST	NET_RT_IFLIST	NET_RT_IFLIST
5	RTF_LLINFO	0	0	0	0	0

Figure 19.36 Examples of programs that call sysctl to obtain routing table and interface list.

The first three programs fetch entries from the routing table and the last three fetch the interface list. The routed program supports only the Internet routing protocols, so it specifies a mib[3] value of AF_INET, while gated supports other protocols, so its value for mib[3] is 0.

Figure 19.37 shows the organization of the three sysctl_*xxx* functions that we cover in the following sections.

Figure 19.38 shows the sysctl_rtable function.

Validate arguments

705–719 The new argument is used when the process is calling sysctl to set the value of a variable, which isn't supported with the routing tables. Therefore this argument must be a null pointer.

720–721 namelen must be 3 because at this point in the processing of the system call, three elements in the name array remain: name[0], the address family (what the process specifies as mib[3]); name[1], the operation (mib[4]); and name[2], the flags (mib[5]).

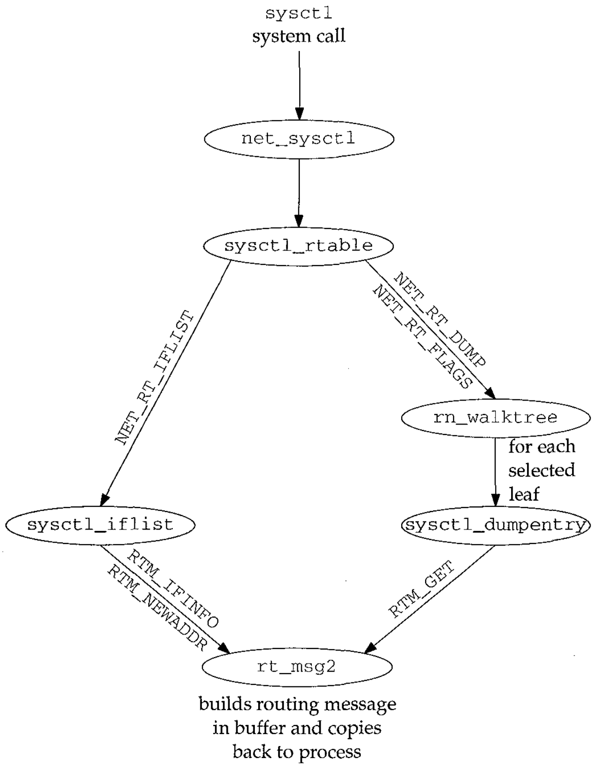

Figure 19.37 Functions that support the `sysctl` system call for routing sockets.

rtsock.c

```
int
sysctl_rtable(name, namelen, where, given, new, newlen)
int     *name;
int      namelen;
caddr_t where;
size_t *given;
caddr_t *new;
size_t  newlen;
{
    struct radix_node_head *rnh;
    int     i, s, error = EINVAL;
    u_char  af;
    struct walkarg w;

    if (new)
        return (EPERM);
```

```
720     if (namelen != 3)
721         return (EINVAL);
722     af = name[0];
723     Bzero(&w, sizeof(w));
724     w.w_where = where;
725     w.w_given = *given;
726     w.w_needed = 0 - w.w_given;
727     w.w_op = name[1];
728     w.w_arg = name[2];

729     s = splnet();
730     switch (w.w_op) {

731     case NET_RT_DUMP:
732     case NET_RT_FLAGS:
733         for (i = 1; i <= AF_MAX; i++)
734             if ((rnh = rt_tables[i]) && (af == 0 || af == i) &&
735                 (error = rnh->rnh_walktree(rnh,
736                                            sysctl_dumpentry, &w)))
737                 break;
738         break;

739     case NET_RT_IFLIST:
740         error = sysctl_iflist(af, &w);
741     }
742     splx(s);
743     if (w.w_tmem)
744         free(w.w_tmem, M_RTABLE);
745     w.w_needed += w.w_given;
746     if (where) {
747         *given = w.w_where - where;
748         if (*given < w.w_needed)
749             return (ENOMEM);
750     } else {
751         *given = (11 * w.w_needed) / 10;
752     }
753     return (error);
754 }
```

rtsock.c

Figure 19.38 `sysctl_rtable` function: process `sysctl` system call requests.

Initialize `walkarg` structure

723–728 A `walkarg` structure (Figure 19.31) is set to 0 and the following members are initialized: `w_where` is the address in the calling process of the buffer for the results (this can be a null pointer, as we mentioned); `w_given` is the size of the buffer in bytes (this is meaningless on input if `w_where` is a null pointer, but it must be set on return to the amount of data that would have been returned); `w_needed` is set to the negative of the buffer size; `w_op` is the operation (the `NET_RT_`*xxx* value); and `w_arg` is the flags value.

Dump routing table

731–738 The `NET_RT_DUMP` and `NET_RT_FLAGS` operations are handled the same way: a loop is made through all the routing tables (the `rt_tables` array), and if the routing table is in use and either the address family argument was 0 or the address family argument matches the family of this routing table, the `rnh_walktree` function is called to process the entire routing table. In Figure 18.17 we show that this function is normally `rn_walktree`. The second argument to this function is the address of another function that is called for each leaf of the routing tree (`sysctl_dumpentry`). The third pointer is just a pointer to anything that `rn_walktree` passes to the `sysctl_dumpentry` function. This argument is a pointer to the `walkarg` structure that contains all the information about this `sysctl` call.

Return interface list

739–740 The `NET_RT_IFLIST` operation calls the function `sysctl_iflist`, which goes through all the `ifnet` structures.

Release buffer

743–744 If a buffer was allocated by `rt_msg2` to contain a routing message, it is now released.

Update `w_needed`

745 The size of each message was added to `w_needed` by `rt_msg2`. Since this variable was initialized to the negative of `w_given`, its value can now be expressed as

```
w_needed = 0 - w_given + totalbytes
```

where `totalbytes` is the sum of all the message lengths added by `rt_msg2`. By adding the value of `w_given` back into `w_needed`, we get

```
w_needed = 0 - w_given + totalbytes + w_given
         = totalbytes
```

the total number of bytes. Since the two values of `w_given` in this equation end up canceling each other, when the process specifies `w_where` as a null pointer it need not initialize the value of `w_given`. Indeed, we see in Figure 19.35 that the variable `needed` was not initialized.

Return actual size of message

746–749 If `where` is nonnull, the number of bytes stored in the buffer is returned through the `given` pointer. If this value is less than the size of the buffer specified by the process, an error is returned because the return information has been truncated.

Return estimated size of message

750–752 When the `where` pointer is null, the process just wants the total number of bytes returned. A 10% fudge factor is added to the size, in case the size of the desired tables increases between this call to `sysctl` and the next.

19.15 `sysctl_dumpentry` Function

In the previous section we described how this function is called by `rn_walktree`, which in turn is called by `sysctl_rtable`. Figure 19.39 shows the function.

rtsock.c

```
623 int
624 sysctl_dumpentry(rn, w)
625 struct radix_node *rn;
626 struct walkarg *w;
627 {
628     struct rtentry *rt = (struct rtentry *) rn;
629     int     error = 0, size;
630     struct rt_addrinfo info;

631     if (w->w_op == NET_RT_FLAGS && !(rt->rt_flags & w->w_arg))
632         return 0;
633     bzero((caddr_t) & info, sizeof(info));
634     dst = rt_key(rt);
635     gate = rt->rt_gateway;
636     netmask = rt_mask(rt);
637     genmask = rt->rt_genmask;
638     size = rt_msg2(RTM_GET, &info, 0, w);
639     if (w->w_where && w->w_tmem) {
640         struct rt_msghdr *rtm = (struct rt_msghdr *) w->w_tmem;

641         rtm->rtm_flags = rt->rt_flags;
642         rtm->rtm_use = rt->rt_use;
643         rtm->rtm_rmx = rt->rt_rmx;
644         rtm->rtm_index = rt->rt_ifp->if_index;
645         rtm->rtm_errno = rtm->rtm_pid = rtm->rtm_seq = 0;
646         rtm->rtm_addrs = info.rti_addrs;
647         if (error = copyout((caddr_t) rtm, w->w_where, size))
648             w->w_where = NULL;
649         else
650             w->w_where += size;
651     }
652     return (error);
653 }
```

rtsock.c

Figure 19.39 sysctl_dumpentry function: process one routing table entry.

623–630 Each time this function is called, its first argument points to a `radix_node` structure, which is also a pointer to a `rtentry` structure. The second argument points to the `walkarg` structure that was initialized by `sysctl_rtable`.

Check flags of routing table entry

631–632 If the process specified a flag value (`mib[5]`), this entry is skipped if the `rt_flags` member doesn't have the desired flag set. We see in Figure 19.36 that the `arp` program uses this to select only those entries with the `RTF_LLINFO` flag set, since these are the entries of interest to ARP.

Form routing message

633–638 The following four pointers in the `rti_info` array are copied from the routing table entry: `dst`, `gate`, `netmask`, and `genmask`. The first two are always nonnull, but the other two can be null. `rt_msg2` forms an `RTM_GET` message.

Copy message back to process

639–651 If the process wants the message returned and a buffer was allocated by `rt_msg2`, the remainder of the routing message is formed in the buffer pointed to by `w_tmem` and `copyout` copies the message back to the process. If the copy was successful, `w_where` is incremented by the number of bytes copied.

19.16 `sysctl_iflist` Function

This function, shown in Figure 19.40, is called directly by `sysctl_rtable` to return the interface list to the process.

rtsock.c

```
int
sysctl_iflist(af, w)
int     af;
struct walkarg *w;
{
    struct ifnet *ifp;
    struct ifaddr *ifa;
    struct rt_addrinfo info;
    int     len, error = 0;

    bzero((caddr_t) & info, sizeof(info));
    for (ifp = ifnet; ifp; ifp = ifp->if_next) {
        if (w->w_arg && w->w_arg != ifp->if_index)
            continue;
        ifa = ifp->if_addrlist;
        ifpaddr = ifa->ifa_addr;
        len = rt_msg2(RTM_IFINFO, &info, (caddr_t) 0, w);
        ifpaddr = 0;
        if (w->w_where && w->w_tmem) {
            struct if_msghdr *ifm;

            ifm = (struct if_msghdr *) w->w_tmem;
            ifm->ifm_index = ifp->if_index;
            ifm->ifm_flags = ifp->if_flags;
            ifm->ifm_data = ifp->if_data;
            ifm->ifm_addrs = info.rti_addrs;
            if (error = copyout((caddr_t) ifm, w->w_where, len))
                return (error);
            w->w_where += len;
        }
        while (ifa = ifa->ifa_next) {
            if (af && af != ifa->ifa_addr->sa_family)
                continue;
            ifaaddr = ifa->ifa_addr;
            netmask = ifa->ifa_netmask;
            brdaddr = ifa->ifa_dstaddr;
            len = rt_msg2(RTM_NEWADDR, &info, 0, w);
            if (w->w_where && w->w_tmem) {
                struct ifa_msghdr *ifam;
```

```
691                 ifam = (struct ifa_msghdr *) w->w_tmem;
692                 ifam->ifam_index = ifa->ifa_ifp->if_index;
693                 ifam->ifam_flags = ifa->ifa_flags;
694                 ifam->ifam_metric = ifa->ifa_metric;
695                 ifam->ifam_addrs = info.rti_addrs;
696                 if (error = copyout(w->w_tmem, w->w_where, len))
697                     return (error);
698                 w->w_where += len;
699             }
700         }
701         ifaddr = netmask = brdaddr = 0;
702     }
703     return (0);
704 }
```

rtsock.c

Figure 19.40 `sysctl_iflist` function: return list of interfaces and their addresses.

This function is a `for` loop that iterates through each interface starting with the one pointed to by `ifnet`. Then a `while` loop proceeds through the linked list of `ifaddr` structures for each interface. An `RTM_IFINFO` routing message is generated for each interface and an `RTM_NEWADDR` message for each address.

Check interface index

654–666 The process can specify a nonzero flags argument (`mib[5]` in Figure 19.36) to select only the interface with a matching `if_index` value.

Build routing message

667–670 The only socket address structure returned with the `RTM_IFINFO` message is `ifpaddr`. The message is built by `rt_msg2`. The pointer `ifpaddr` in the `info` structure is then set to 0, since the same `info` structure is used for generating the subsequent `RTM_NEWADDR` messages.

Copy message back to process

671–681 If the process wants the message returned, the remainder of the `if_msghdr` structure is filled in, `copyout` copies the buffer to the process, and `w_where` is incremented.

Iterate through address structures, check address family

682–684 Each `ifaddr` structure for the interface is processed and the process can specify a nonzero address family (`mib[3]` in Figure 19.36) to select only the interface addresses of the given family.

Build routing message

685–688 Up to three socket address structures are returned in each `RTM_NEWADDR` message: `ifaaddr`, `netmask`, and `brdaddr`. The message is built by `rt_msg2`.

Copy message back to process

689–699 If the process wants the message returned, the remainder of the `ifa_msghdr` structure is filled in, `copyout` copies the buffer to the process, and `w_where` is incremented.

701 These three pointers in the `info` array are set to 0, since the same array is used for the next interface message.

19.17 Summary

Routing messages all have the same format—a fixed-length structure followed by a variable number of socket address structures. There are three different types of messages, each corresponding to a different fixed-length structure, and the first three elements of each structure identify the length, version, and type of message. A bitmask in each structure identifies which socket address structures follow the fixed-length structure.

These messages are passed between a process and the kernel in two different ways. Messages can be passed in either direction, one message per read or write, across a routing socket. This allows a superuser process complete read and write access to the kernel's routing tables. This is how routing daemons such as `routed` and `gated` implement their desired routing policy.

Alternatively any process can read the contents of the kernel's routing tables using the `sysctl` system call. This does not involve a routing socket and does not require special privileges. The entire result, normally consisting of many routing messages, is returned as part of the system call. Since the process does not know the size of the result, a method is provided for the system call to return this size without returning the actual result.

Exercises

19.1 What is the difference in the `RTF_DYNAMIC` and `RTF_MODIFIED` flags? Can both be set for a given routing table entry?

19.2 What happens when the default route is entered with the command of the form

```
bsdi $ route add default -cloning -genmask 255.255.255.255 sun
```

19.3 Estimate the space required by `sysctl` to dump a routing table that contains 15 ARP entries and 20 routes.

20

Routing Sockets

20.1 Introduction

A process sends and receives the routing messages described in the previous chapter by using a socket in the *routing domain.* The `socket` system call is issued specifying a family of `PF_ROUTE` and a socket type of `SOCK_RAW`.

The process can then send five routing messages to the kernel:

1. `RTM_ADD`: add a new route.
2. `RTM_DELETE`: delete an existing route.
3. `RTM_GET`: fetch all the information about a route.
4. `RTM_CHANGE`: change the gateway, interface, or metrics of an existing route.
5. `RTM_LOCK`: specify which metrics the kernel should not modify.

Additionally, the process can receive any of the other seven types of routing messages that are generated by the kernel when some event, such as interface down, redirect received, etc., occurs.

This chapter looks at the routing domain, the routing control blocks that are created for each routing socket, the function that handles messages from a process (`route_output`), the function that sends routing messages to one or more processes (`raw_input`), and the various functions that support all the socket operations on a routing socket.

20.2 `routedomain` and `protosw` Structures

Before describing the routing socket functions, we need to discuss additional details about the routing domain; the `SOCK_RAW` protocol supported in the routing domain; and routing control blocks, one of which is associated with each routing socket.

Figure 20.1 lists the `domain` structure for the `PF_ROUTE` domain, named `routedomain`.

Member	`Value`	Description
`dom_family`	*PF_ROUTE*	protocol family for domain
`dom_name`	*route*	name
`dom_init`	*route_init*	domain initialization, Figure 18.30
`dom_externalize`	*0*	not used in routing domain
`dom_dispose`	*0*	not used in routing domain
`dom_protosw`	*routesw*	protocol switch structure, Figure 20.2
`dom_protoswNPROTOSW`		pointer past end of protocol switch structure
`dom_next`		filled in by `domaininit`, Figure 7.15
`dom_rtattach`	*0*	not used in routing domain
`dom_rtoffset`	*0*	not used in routing domain
`dom_maxrtkey`	*0*	not used in routing domain

Figure 20.1 `routedomain` structure.

Unlike the Internet domain, which supports multiple protocols (TCP, UDP, ICMP, etc.), only one protocol (of type `SOCK_RAW`) is supported in the routing domain. Figure 20.2 lists the protocol switch entry for the `PF_ROUTE` domain.

Member	`routesw[0]`	Description
`pr_type`	*SOCK_RAW*	raw socket
`pr_domain`	*&routedomain*	part of the routing domain
`pr_protocol`	*0*	
`pr_flags`	*PR_ATOMIC | PR_ADDR*	socket layer flags, not used by protocol processing
`pr_input`	*raw_input*	this entry not used; `raw_input` called directly
`pr_output`	*route_output*	called for `PRU_SEND` requests
`pr_ctlinput`	*raw_ctlinput*	control input function
`pr_ctloutput`	*0*	not used
`pr_usrreq`	*route_usrreq*	respond to communication requests from a process
`pr_init`	*raw_init*	initialization
`pr_fasttimo`	*0*	not used
`pr_slowtimo`	*0*	not used
`pr_drain`	*0*	not used
`pr_sysctl`	*sysctl_rtable*	for `sysctl`(8) system call

Figure 20.2 The routing protocol `protosw` structure.

20.3 Routing Control Blocks

Each time a routing socket is created with a call of the form

```
socket(PF_ROUTE, SOCK_RAW, protocol);
```

the corresponding `PRU_ATTACH` request to the protocol's user-request function (`route_usrreq`) allocates a routing control block and links it to the socket structure. The *protocol* can restrict the messages sent to the process on this socket to one particular family. If a *protocol* of `AF_INET` is specified, for example, only routing messages containing Internet addresses will be sent to the process. A *protocol* of 0 causes all routing messages from the kernel to be sent on the socket.

> Recall that we call these structures *routing control blocks*, not *raw control blocks*, to avoid confusion with the raw IP control blocks in Chapter 32.

Figure 20.3 shows the definition of the `rawcb` structure.

raw_cb.h

```
39 struct rawcb {
40     struct rawcb *rcb_next;      /* doubly linked list */
41     struct rawcb *rcb_prev;
42     struct socket *rcb_socket;   /* back pointer to socket */
43     struct sockaddr *rcb_faddr;  /* destination address */
44     struct sockaddr *rcb_laddr;  /* socket's address */
45     struct sockproto rcb_proto;  /* protocol family, protocol */
46 };

47 #define sotorawcb(so)       ((struct rawcb *)(so)->so_pcb)
```

raw_cb.h

Figure 20.3 `rawcb` structure.

Additionally, a global of the same name, `rawcb`, is allocated as the head of the doubly linked list. Figure 20.4 shows the arrangement.

39–47 We showed the `sockproto` structure in Figure 19.26. Its `sp_family` member is set to `PF_ROUTE` and its `sp_protocol` member is set to the third argument to the `socket` system call. The `rcb_faddr` member is permanently set to point to `route_src`, which we described with Figure 19.26. `rcb_laddr` is always a null pointer.

20.4 `raw_init` Function

The `raw_init` function, shown in Figure 20.5, is the protocol initialization function in the `protosw` structure in Figure 20.2. We described the entire initialization of the routing domain with Figure 18.29.

38–42 The function initializes the doubly linked list of routing control blocks by setting the next and previous pointers of the head structure to point to itself.

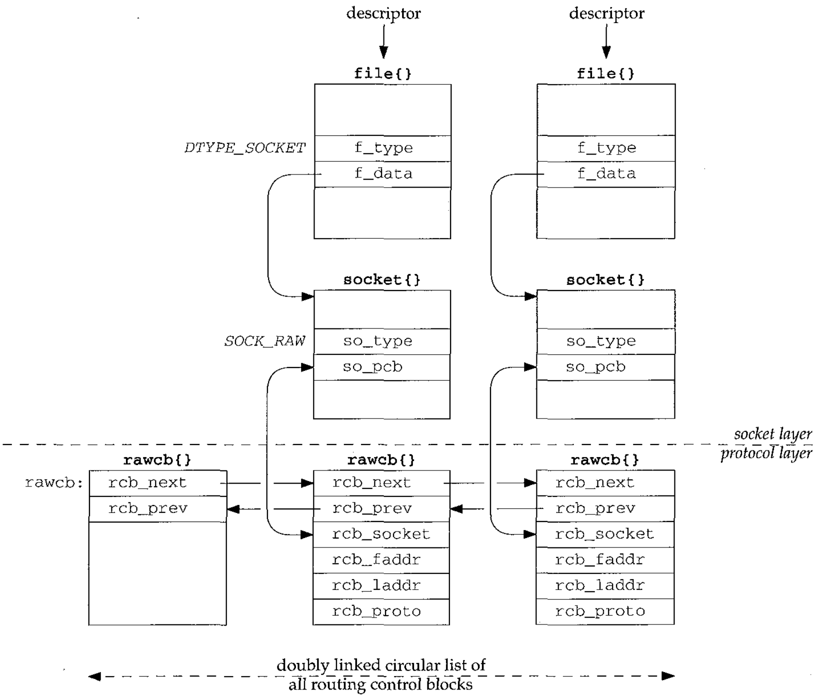

Figure 20.4 Relationship of raw protocol control blocks to other data structures.

raw_usrreq.c

```
void
raw_init()
{

    rawcb.rcb_next = rawcb.rcb_prev = &rawcb;
}
```

raw_usrreq.c

Figure 20.5 `raw_init` function: initialize doubly linked list of routing control blocks.

20.5 `route_output` Function

As we showed in Figure 18.11, `route_output` is called when the `PRU_SEND` request is issued to the protocol's user-request function, which is the result of a write operation by a process to a routing socket. In Figure 18.9 we indicated that five different types of routing messages are accepted by the kernel from a process.

Since this function is invoked as a result of a write by a process, the data from the process (the routing message to process) is in an mbuf chain from `sosend`. Figure 20.6

shows an overview of the processing steps, assuming the process sends an RTM_ADD command, specifying three addresses: the destination, its gateway, and a network mask (hence this is a network route, not a host route).

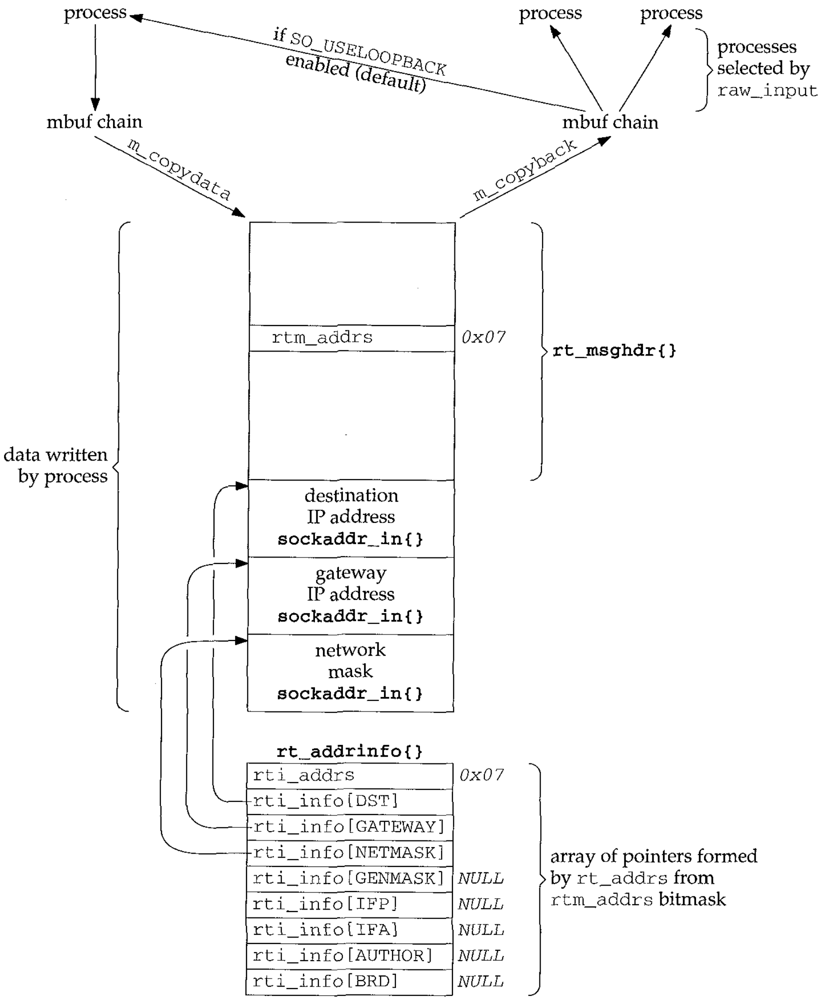

Figure 20.6 Example processing of an RTM_ADD command from a process.

There are numerous points to note in this figure, most of which we'll cover as we proceed through the source code for `route_output`. Also note that, to save space, we omit the RTAX_ prefix for each array index in the `rt_addrinfo` structure.

- The process specifies which socket address structures follow the fixed-length `rt_msghdr` structure by setting the bitmask `rtm_addrs`. We show a bitmask of `0x07`, which corresponds to a destination address, a gateway address, and a network mask (Figure 19.19). The `RTM_ADD` command requires the first two; the third is optional. Another optional address, the `genmask` specifies the mask to be used for generating cloned routes.
- The `write` system call (the `sosend` function) copies the buffer from the process into an mbuf chain in the kernel.
- `m_copydata` copies the mbuf chain into a buffer that `route_output` obtains using `malloc`. It is easier to access all the information in the structure and the socket address structures that follow when stored in a single contiguous buffer than it is when stored in an mbuf chain.
- The function `rt_xaddrs` is called by `route_output` to take the bitmask and build the `rt_addrinfo` structure that points into the buffer. The code in `route_output` references these structures using the names shown in the fifth column in Figure 19.19. The bitmask is also copied into the `rti_addrs` member.
- `route_output` normally modifies the `rt_msghdr` structure. If an error occurs, the corresponding `errno` value is returned in `rtm_errno` (for example, `EEXIST` if the route already exists); otherwise the flag `RTF_DONE` is logically ORed into the `rtm_flags` supplied by the process.
- The `rt_msghdr` structure and the addresses that follow become input to 0 or more processes that are reading from a routing socket. The buffer is first converted back into an mbuf chain by `m_copyback`. `raw_input` goes through all the routing PCBs and passes a copy to the appropriate processes. We also show that a process with a routing socket receives a copy of each message it writes to that socket unless it disables the `SO_USELOOPBACK` socket option.

> To avoid receiving a copy of their own routing messages, some programs, such as `route`, call `shutdown` with a second argument of 0 to prevent any data from being received on the routing socket.

We examine the source code for `route_output` in seven parts. Figure 20.7 shows an overview of the function.

```
int
route_output()
{
    R_Malloc() to allocate buffer;
    m_copydata() to copy from mbuf chain into buffer;
    rt_xaddrs() to build rt_addrinfo{};

    switch (message type) {
    case RTM_ADD:
        rtrequest(RTM_ADD);
        rt_setmetrics();
        break;
```

```
        case RTM_DELETE:
            rtrequest(RTM_DELETE);
            break;

        case RTM_GET:
        case RTM_CHANGE:
        case RTM_LOCK:
            rtalloc1();

            switch (message type) {
            case RTM_GET:
                rt_msg2(RTM_GET);
                break;

            case RTM_CHANGE:
                change appropriate fields;
                /* fall through */

            case RTM_LOCK:
                set rmx_locks;
                break;
            }
            break;
        }

        set rtm_error if error, else set RTF_DONE flag;

        m_copyback() to copy from buffer into mbuf chain;

        raw_input();     /* mbuf chain to appropriate processes */
}
```

Figure 20.7 Summary of `route_output` processing steps.

The first part of `route_output` is shown in Figure 20.8.

Check mbuf for validity

113–136 The mbuf chain is checked for validity: its length must be at least the size of an `rt_msghdr` structure. The first longword is fetched from the data portion of the mbuf, which contains the `rtm_msglen` value.

Allocate buffer

137–142 A buffer is allocated to hold the entire message and `m_copydata` copies the message from the mbuf chain into the buffer.

Check version number

143–146 The version of the message is checked. In the future, should a new version of the routing messages be introduced, this member could be used to provide support for older versions.

147–149 The process ID is copied into `rtm_pid` and the bitmask supplied by the process is copied into `info.rti_addrs`, a structure local to this function. The function `rt_xaddrs` (shown in the next section) fills in the eight socket address pointers in the `info` structure to point into the buffer now containing the message.

rtsock.c

```
int
route_output(m, so)
struct mbuf *m;
struct socket *so;
{
    struct rt_msghdr *rtm = 0;
    struct rtentry *rt = 0;
    struct rtentry *saved_nrt = 0;
    struct rt_addrinfo info;
    int     len, error = 0;
    struct ifnet *ifp = 0;
    struct ifaddr *ifa = 0;

#define senderr(e) { error = e; goto flush;}
    if (m == 0 || ((m->m_len < sizeof(long)) &&
                           (m = m_pullup(m, sizeof(long))) == 0))
                return (ENOBUFS);
    if ((m->m_flags & M_PKTHDR) == 0)
        panic("route_output");
    len = m->m_pkthdr.len;
    if (len < sizeof(*rtm) ||
        len != mtod(m, struct rt_msghdr *)->rtm_msglen) {
        dst = 0;
        senderr(EINVAL);
    }
    R_Malloc(rtm, struct rt_msghdr *, len);
    if (rtm == 0) {
        dst = 0;
        senderr(ENOBUFS);
    }
    m_copydata(m, 0, len, (caddr_t) rtm);
    if (rtm->rtm_version != RTM_VERSION) {
        dst = 0;
        senderr(EPROTONOSUPPORT);
    }
    rtm->rtm_pid = curproc->p_pid;

    info.rti_addrs = rtm->rtm_addrs;
    rt_xaddrs((caddr_t) (rtm + 1), len + (caddr_t) rtm, &info);

    if (dst == 0)
        senderr(EINVAL);

    if (genmask) {
        struct radix_node *t;
        t = rn_addmask((caddr_t) genmask, 1, 2);
        if (t && Bcmp(genmask, t->rn_key, *(u_char *) genmask) == 0)
            genmask = (struct sockaddr *) (t->rn_key);
        else
            senderr(ENOBUFS);
    }
```

rtsock.c

Figure 20.8 `route_output` function: initial processing, copy message from mbuf chain.

Destination address required

150–151 A destination address is a required address for all commands. If the `info.rti_info[RTAX_DST]` element is a null pointer, `EINVAL` is returned. Remember that `dst` refers to this array element (Figure 19.19).

Handle optional genmask

152–159 A `genmask` is optional and is used as the network mask for routes created when the `RTF_CLONING` flag is set (Figure 19.8). `rn_addmask` adds the mask to the tree of masks, first searching for an existing entry for the mask and then referencing that entry if found. If the mask is found or added to the mask tree, an additional check is made that the entry in the mask tree really equals the `genmask` value, and, if so, the `genmask` pointer is replaced with a pointer to the mask in the mask tree.

Figure 20.9 shows the next part of `route_output`, which handles the `RTM_ADD` and `RTM_DELETE` commands.

rtsock.c

```
160     switch (rtm->rtm_type) {

161     case RTM_ADD:
162         if (gate == 0)
163             senderr(EINVAL);
164         error = rtrequest(RTM_ADD, dst, gate, netmask,
165                           rtm->rtm_flags, &saved_nrt);
166         if (error == 0 && saved_nrt) {
167             rt_setmetrics(rtm->rtm_inits,
168                           &rtm->rtm_rmx, &saved_nrt->rt_rmx);
169             saved_nrt->rt_refcnt--;
170             saved_nrt->rt_genmask = genmask;
171         }
172         break;

173     case RTM_DELETE:
174         error = rtrequest(RTM_DELETE, dst, gate, netmask,
175                           rtm->rtm_flags, (struct rtentry **) 0);
176         break;
```

rtsock.c

Figure 20.9 `route_output` function: process `RTM_ADD` and `RTM_DELETE` commands.

162–163 An `RTM_ADD` command requires the process to specify a gateway.

164–165 `rtrequest` processes the request. The `netmask` pointer can be null if the route being entered is a host route. If all is OK, the pointer to the new routing table entry is returned through `saved_nrt`.

166–172 The `rt_metrics` structure is copied from the caller's buffer into the routing table entry. The reference count is decremented and the `genmask` pointer is stored (possibly a null pointer).

173–176 Processing the `RTM_DELETE` command is simple because all the work is done by `rtrequest`. Since the final argument is a null pointer, `rtrequest` calls `rtfree` if the reference count is 0, deleting the entry from the routing table (Figure 19.7).

The next part of the processing is shown in Figure 20.10, which handles the common code for the RTM_GET, RTM_CHANGE, and RTM_LOCK commands.

rtsock.c

```
177     case RTM_GET:
178     case RTM_CHANGE:
179     case RTM_LOCK:
180         rt = rtalloc1(dst, 0);
181         if (rt == 0)
182             senderr(ESRCH);
183         if (rtm->rtm_type != RTM_GET) {     /* XXX: too grotty */
184             struct radix_node *rn;
185             extern struct radix_node_head *mask_rnhead;

186             if (Bcmp(dst, rt_key(rt), dst->sa_len) != 0)
187                 senderr(ESRCH);
188             if (netmask && (rn = rn_search(netmask,
189                                            mask_rnhead->rnh_treetop)))
190                 netmask = (struct sockaddr *) rn->rn_key;
191             for (rn = rt->rt_nodes; rn; rn = rn->rn_dupedkey)
192                 if (netmask == (struct sockaddr *) rn->rn_mask)
193                     break;
194             if (rn == 0)
195                 senderr(ETOOMANYREFS);
196             rt = (struct rtentry *) rn;
197         }
```

rtsock.c

Figure 20.10 `route_output` function: common processing for RTM_GET, RTM_CHANGE, and RTM_LOCK.

Locate existing entry

177–182 Since all three commands reference an existing entry, `rtalloc1` locates the entry. If the entry isn't found, `ESRCH` is returned.

Do not allow network match

183–187 For the `RTM_CHANGE` and `RTM_LOCK` commands, a network match is inadequate: an exact match with the routing table key is required. Therefore, if the `dst` argument doesn't equal the routing table key, the match was a network match and `ESRCH` is returned.

Use network mask to find correct entry

188–193 Even with an exact match, if there are duplicate keys, each with a different network mask, the correct entry must still be located. If a `netmask` argument was supplied, it is looked up in the mask table (`mask_rnhead`). If found, the `netmask` pointer is replaced with the pointer to the mask in the mask tree. Each leaf node in the duplicate key list is examined, looking for an entry with an `rn_mask` pointer that equals `netmask`. This test compares the pointers, not the structures that they point to. This works because all masks appear in the mask tree, and only one copy of each unique mask is stored in this tree. In the common case, keys are not duplicated, so the `for` loop iterates once. If a host entry is being modified, a mask must not be specified and then both `netmask` and `rn_mask` are null pointers (which are equal). But if an entry that has an associated mask is being modified, that mask must be specified as the `netmask` argument.

194–195 If the `for` loop terminates without finding a matching network mask, `ETOOMANYREFS` is returned.

> The comment `XXX` is because this function must go to all this work to find the desired entry. All these details should be hidden in another function similar to `rtalloc1` that detects a network match and handles a mask argument.

The next part of this function, shown in Figure 20.11, continues processing the `RTM_GET` command. This command is unique among the commands supported by `route_output` in that it can return more data than it was passed. For example, only a single socket address structure is required as input, the destination, but at least two are returned: the destination and its gateway. With regard to Figure 20.6, this means the buffer allocated for `m_copydata` to copy into might need to be increased in size.

rtsock.c

```
    switch (rtm->rtm_type) {

    case RTM_GET:
        dst = rt_key(rt);
        gate = rt->rt_gateway;
        netmask = rt_mask(rt);
        genmask = rt->rt_genmask;
        if (rtm->rtm_addrs & (RTA_IFP | RTA_IFA)) {
            if (ifp = rt->rt_ifp) {
                ifpaddr = ifp->if_addrlist->ifa_addr;
                ifaaddr = rt->rt_ifa->ifa_addr;
                rtm->rtm_index = ifp->if_index;
            } else {
                ifpaddr = 0;
                ifaaddr = 0;
            }
        }
        len = rt_msg2(RTM_GET, &info, (caddr_t) 0,
                      (struct walkarg *) 0);
        if (len > rtm->rtm_msglen) {
            struct rt_msghdr *new_rtm;
            R_Malloc(new_rtm, struct rt_msghdr *, len);
            if (new_rtm == 0)
                senderr(ENOBUFS);
            Bcopy(rtm, new_rtm, rtm->rtm_msglen);
            Free(rtm);
            rtm = new_rtm;
        }
        (void) rt_msg2(RTM_GET, &info, (caddr_t) rtm,
                       (struct walkarg *) 0);
        rtm->rtm_flags = rt->rt_flags;
        rtm->rtm_rmx = rt->rt_rmx;
        rtm->rtm_addrs = info.rti_addrs;
        break;
```

rtsock.c

Figure 20.11 `route_output` function: `RTM_GET` processing.

Return destination, gateway, and masks

198–203 Four pointers are stored in the `rti_info` array: `dst`, `gate`, `netmask`, and `genmask`. The latter two might be null pointers. These pointers in the `info` structure point to the socket address structures that will be returned to the process.

Return interface information

204–213 The process can set the masks `RTA_IFP` and `RTA_IFA` in the `rtm_flags` bitmask. If either or both are set, the process wants to receive the contents of both the `ifaddr` structures pointed to by this routing table entry: the link-level address of the interface (pointed to by `rt_ifp->if_addrlist`) and the protocol address for this entry (pointed to by `rt_ifa->ifa_addr`). The interface index is also returned.

Construct reply

214–224 `rt_msg2` is called with a null third pointer to calculate the length of the routing message corresponding to `RTM_GET` and the addresses pointed to by the `info` structure. If the length of the result message exceeds the length of the input message, then a new buffer is allocated, the input message is copied into the new buffer, the old buffer is released, and `rtm` is set to point to the new buffer.

225–230 `rt_msg2` is called again, this time with a nonnull third pointer, which builds the result message in the buffer. The final three members in the `rt_msghdr` structure are then filled in.

Figure 20.12 shows the processing of the `RTM_CHANGE` and `RTM_LOCK` commands.

Change gateway

231–233 If a `gate` address was passed by the process, `rt_setgate` is called to change the gateway for the entry.

Locate new interface

234–244 The new gateway (if changed) can also require new `rt_ifp` and `rt_ifa` pointers. The process can specify these new values by passing either an `ifpaddr` socket address structure or an `ifaaddr` socket address structure. The former is tried first, and then the latter. If neither is passed by the process, the `rt_ifp` and `rt_ifa` pointers are left alone.

Check if interface changed

245–256 If an interface was located (`ifa` is nonnull), then the existing `rt_ifa` pointer for the route is compared to the new value. If it has changed, new values for `rt_ifp` and `rt_ifa` are stored in the routing table entry. Before doing this the interface request function (if defined) is called with a command of `RTM_DELETE`. The delete is required because the link-layer information from one type of network to another can be quite different, say changing a route from an X.25 network to an Ethernet, and the output routines must be notified.

Update metrics

257–258 The metrics in the routing table entry are updated by `rt_setmetrics`.

rtsock.c

```
231         case RTM_CHANGE:
232             if (gate && rt_setgate(rt, rt_key(rt), gate))
233                 senderr(EDQUOT);
234             /* new gateway could require new ifaddr, ifp; flags may also be
235                different; ifp may be specified by ll sockaddr when protocol
236                address is ambiguous */
237             if (ifpaddr && (ifa = ifa_ifwithnet(ifpaddr)) &&
238                 (ifp = ifa->ifa_ifp))
239                 ifa = ifaof_ifpforaddr(ifaaddr ? ifaaddr : gate,
240                                       ifp);
241             else if ((ifaaddr && (ifa = ifa_ifwithaddr(ifaaddr))) ||
242                      (ifa = ifa_ifwithroute(rt->rt_flags,
243                                             rt_key(rt), gate)))
244                 ifp = ifa->ifa_ifp;
245             if (ifa) {
246                 struct ifaddr *oifa = rt->rt_ifa;
247                 if (oifa != ifa) {
248                     if (oifa && oifa->ifa_rtrequest)
249                         oifa->ifa_rtrequest(RTM_DELETE,
250                                             rt, gate);
251                     IFAFREE(rt->rt_ifa);
252                     rt->rt_ifa = ifa;
253                     ifa->ifa_refcnt++;
254                     rt->rt_ifp = ifp;
255                 }
256             }
257             rt_setmetrics(rtm->rtm_inits, &rtm->rtm_rmx,
258                           &rt->rt_rmx);
259             if (rt->rt_ifa && rt->rt_ifa->ifa_rtrequest)
260                 rt->rt_ifa->ifa_rtrequest(RTM_ADD, rt, gate);
261             if (genmask)
262                 rt->rt_genmask = genmask;
263             /*
264              * Fall into
265              */
266         case RTM_LOCK:
267             rt->rt_rmx.rmx_locks &= ~(rtm->rtm_inits);
268             rt->rt_rmx.rmx_locks |=
269                 (rtm->rtm_inits & rtm->rtm_rmx.rmx_locks);
270             break;
271         }
272         break;

273     default:
274         senderr(EOPNOTSUPP);
275     }
```

rtsock.c

Figure 20.12 route_output function: RTM_CHANGE and RTM_LOCK processing.

Call interface request function

259–260 If an interface request function is defined, it is called with a command of RTM_ADD.

Store clone generation mask

261–262 If the process specifies the `genmask` argument, the pointer to the mask that was obtained in Figure 20.8 is saved in `rt_genmask`.

Update bitmask of locked metrics

266–270 The `RTM_LOCK` command updates the bitmask stored in `rt_rmx.rmx_locks`. Figure 20.13 shows the values of the different bits in this bitmask, one value per metric.

Constant	Value	Description
RTV_MTU	`0x01`	initialize or lock `rmx_mtu`
RTV_HOPCOUNT	`0x02`	initialize or lock `rmx_hopcount`
RTV_EXPIRE	`0x04`	initialize or lock `rmx_expire`
RTV_RPIPE	`0x08`	initialize or lock `rmx_recvpipe`
RTV_SPIPE	`0x10`	initialize or lock `rmx_sendpipe`
RTV_SSTHRESH	`0x20`	initialize or lock `rmx_ssthresh`
RTV_RTT	`0x40`	initialize or lock `rmx_rtt`
RTV_RTTVAR	`0x80`	initialize or lock `rmx_rttvar`

Figure 20.13 Constants to initialize or lock metrics.

The `rmx_locks` member of the `rt_metrics` structure in the routing table entry is the bitmask telling the kernel which metrics to leave alone. That is, those metrics specified by `rmx_locks` won't be updated by the kernel. The only use of these metrics by the kernel is with TCP, as noted with Figure 27.3. The `rmx_pksent` metric cannot be locked or initialized, but it turns out this member is never even referenced or updated by the kernel.

The `rtm_inits` value in the message from the process specifies the bitmask of which metrics were just initialized by `rt_setmetrics`. The `rtm_rmx.rmx_locks` value in the message specifies the bitmask of which metrics should now be locked. The value of `rt_rmx.rmx_locks` is the bitmask in the routing table of which metrics are currently locked. First, any bits to be initialized (`rtm_inits`) are unlocked. Any bits that are both initialized (`rtm_inits`) and locked (`rtm_rmx.rmx_locks`) are locked.

273–275 This `default` is for the `switch` at the beginning of Figure 20.9 and catches any of the routing commands other than the five that are supported in messages from a process.

The final part of `route_output`, shown in Figure 20.14, sends the reply to `raw_input`.

rtsock.c

```
276   flush:
277     if (rtm) {
278         if (error)
279             rtm->rtm_errno = error;
280         else
281             rtm->rtm_flags |= RTF_DONE;
282     }
283     if (rt)
284         rtfree(rt);
285     {
286         struct rawcb *rp = 0;
287         /*
288          * Check to see if we don't want our own messages.
289          */
290         if ((so->so_options & SO_USELOOPBACK) == 0) {
291             if (route_cb.any_count <= 1) {
292                 if (rtm)
293                     Free(rtm);
294                 m_freem(m);
295                 return (error);
296             }
297             /* There is another listener, so construct message */
298             rp = sotorawcb(so);
299         }
300         if (rtm) {
301             m_copyback(m, 0, rtm->rtm_msglen, (caddr_t) rtm);
302             Free(rtm);
303         }
304         if (rp)
305             rp->rcb_proto.sp_family = 0;    /* Avoid us */
306         if (dst)
307             route_proto.sp_protocol = dst->sa_family;
308         raw_input(m, &route_proto, &route_src, &route_dst);
309         if (rp)
310             rp->rcb_proto.sp_family = PF_ROUTE;
311     }
312     return (error);
313 }
```

rtsock.c

Figure 20.14 route_output function: pass results to raw_input.

Return error or OK

276–282 flush is the label jumped to by the senderr macro defined at the beginning of the function. If an error occurred it is returned in the rtm_errno member; otherwise the RTF_DONE flag is set.

Release held route

283–284 If a route is being held, it is released. The call to rtalloc1 at the beginning of Figure 20.10 holds the route, if found.

No process to receive message

285–296 The `SO_USELOOPBACK` socket option is true by default and specifies that the sending process is to receive a copy of each routing message that it writes to a routing socket. (If the sender doesn't receive a copy, it can't receive any of the information returned by `RTM_GET`.) If that option is not set, and the total count of routing sockets is less than or equal to 1, there are no other processes to receive the message and the sender doesn't want a copy. The buffer and mbuf chain are both released and the function returns.

Other listeners but no loopback copy

297–299 There is at least one other listener but the sending process does not want a copy. The pointer `rp`, which defaults to null, is set to point to the routing control block for the sender and is also used as a flag that the sender doesn't want a copy.

Convert buffer into mbuf chain

300–303 The buffer is converted back into an mbuf chain (Figure 20.6) and the buffer released.

Avoid loopback copy

304–305 If `rp` is set, some other process might want the message but the sender does not want a copy. The `sp_family` member of the sender's routing control block is temporarily set to 0, but the `sp_family` of the message (the `route_proto` structure, shown with Figure 19.26) has a family of `PF_ROUTE`. This trick prevents `raw_input` from passing a copy of the result to the sending process because `raw_input` does not pass a copy to any socket with an `sp_family` of 0.

Set address family of routing message

306–308 If `dst` is a nonnull pointer, the address family of that socket address structure becomes the protocol of the routing message. With the Internet protocols this value would be `PF_INET`. A copy is passed to the appropriate listeners by `raw_input`.

309–313 If the `sp_family` member in the calling process was temporarily set to 0, it is reset to `PF_ROUTE`, its normal value.

20.6 `rt_xaddrs` Function

The `rt_xaddrs` function is called only once from `route_output` (Figure 20.8) after the routing message from the process has been copied from the mbuf chain into a buffer and after the bitmask from the process (`rtm_addrs`) has been copied into the `rti_info` member of an `rt_addrinfo` structure. The purpose of `rt_xaddrs` is to take this bitmask and set the pointers in the `rti_info` array to point to the corresponding address in the buffer. Figure 20.15 shows the function.

rtsock.c

```
#define ROUNDUP(a) \
    ((a) > 0 ? (1 + (((a) - 1) | (sizeof(long) - 1))) : sizeof(long))
#define ADVANCE(x, n) (x += ROUNDUP((n)->sa_len))
```

```
333 static void
334 rt_xaddrs(cp, cplim, rtinfo)
335 caddr_t cp, cplim;
336 struct rt_addrinfo *rtinfo;
337 {
338     struct sockaddr *sa;
339     int     i;

340     bzero(rtinfo->rti_info, sizeof(rtinfo->rti_info));
341     for (i = 0; (i < RTAX_MAX) && (cp < cplim); i++) {
342         if ((rtinfo->rti_addrs & (1 << i)) == 0)
343             continue;
344         rtinfo->rti_info[i] = sa = (struct sockaddr *) cp;
345         ADVANCE(cp, sa);
346     }
347 }
```
rtsock.c

Figure 20.15 `rt_xaddrs` function: fill `rti_into` array with pointers.

330–340 The array of pointers is set to 0 so all the pointers to address structures not appearing in the bitmask will be null.

341–347 Each of the 8 (`RTAX_MAX`) possible bits in the bitmask is tested and, if set, a pointer is stored in the `rti_info` array to the corresponding socket address structure. The `ADVANCE` macro takes the `sa_len` field of the socket address structure, rounds it up to the next multiple of 4 bytes, and increments the pointer `cp` accordingly.

20.7 `rt_setmetrics` Function

This function was called twice from `route_output`: when a new route was added and when an existing route was changed. The `rtm_inits` member in the routing message from the process specifies which of the metrics the process wants to initialize from the `rtm_rmx` array. The bit values in the bitmask are shown in Figure 20.13.

Notice that both `rtm_addrs` and `rtm_inits` are bitmasks in the message from the process, the former specifying the socket address structures that follow, and the latter specifying which metrics are to be initialized. Socket address structures whose bits don't appear in `rtm_addrs` don't even appear in the routing message, to save space. But the entire `rt_metrics` array always appears in the fixed-length `rt_msghdr` structure—elements in the array whose bits are not set in `rtm_inits` are ignored.

Figure 20.16 shows the `rt_setmetrics` function.

314–318 The `which` argument is always the `rtm_inits` member of the routing message from the process. `in` points to the `rt_metrics` structure from the process, and `out` points to the `rt_metrics` structure in the routing table entry that is being created or modified.

319–329 Each of the 8 bits in the bitmask is tested and if set, the corresponding metric is copied. Notice that when a new routing table entry is being created with the `RTM_ADD` command, `route_output` calls `rtrequest`, which sets the entire routing table entry to 0 (Figure 19.9). Hence, any metrics not specified by the process in the routing message default to 0.

rtsock.c

```
void
rt_setmetrics(which, in, out)
u_long  which;
struct rt_metrics *in, *out;
{
#define metric(f, e) if (which & (f)) out->e = in->e;
    metric(RTV_RPIPE, rmx_recvpipe);
    metric(RTV_SPIPE, rmx_sendpipe);
    metric(RTV_SSTHRESH, rmx_ssthresh);
    metric(RTV_RTT, rmx_rtt);
    metric(RTV_RTTVAR, rmx_rttvar);
    metric(RTV_HOPCOUNT, rmx_hopcount);
    metric(RTV_MTU, rmx_mtu);
    metric(RTV_EXPIRE, rmx_expire);
#undef metric
}
```

rtsock.c

Figure 20.16 `rt_setmetrics` function: set elements of the `rt_metrics` structure.

20.8 `raw_input` Function

All routing messages destined for a process—those that originate from within the kernel and those that originate from a process—are given to `raw_input`, which selects the processes to receive the message. Figure 18.11 summarizes the four functions that call `raw_input`.

When a routing socket is created, the family is always `PF_ROUTE` and the protocol, the third argument to `socket`, can be 0, which means the process wants to receive all routing messages, or a value such as `AF_INET`, which restricts the socket to messages containing addresses of that specific protocol family. A routing control block is created for each routing socket (Section 20.3) and these two values are stored in the `sp_family` and `sp_protocol` members of the `rcb_proto` structure.

Figure 20.17 shows the `raw_input` function.

raw_usrreq.c

```
void
raw_input(m0, proto, src, dst)
struct mbuf *m0;
struct sockproto *proto;
struct sockaddr *src, *dst;
{
    struct rawcb *rp;
    struct mbuf *m = m0;
    int     sockets = 0;
    struct socket *last;
```

```
    last = 0;
    for (rp = rawcb.rcb_next; rp != &rawcb; rp = rp->rcb_next) {
        if (rp->rcb_proto.sp_family != proto->sp_family)
            continue;
        if (rp->rcb_proto.sp_protocol &&
            rp->rcb_proto.sp_protocol != proto->sp_protocol)
            continue;
        /*
         * We assume the lower level routines have
         * placed the address in a canonical format
         * suitable for a structure comparison.
         *
         * Note that if the lengths are not the same
         * the comparison will fail at the first byte.
         */
#define equal(a1, a2) \
  (bcmp((caddr_t)(a1), (caddr_t)(a2), a1->sa_len) == 0)
        if (rp->rcb_laddr && !equal(rp->rcb_laddr, dst))
            continue;
        if (rp->rcb_faddr && !equal(rp->rcb_faddr, src))
            continue;
        if (last) {
            struct mbuf *n;
            if (n = m_copy(m, 0, (int) M_COPYALL)) {
                if (sbappendaddr(&last->so_rcv, src,
                                  n, (struct mbuf *) 0) == 0)
                    /* should notify about lost packet */
                    m_freem(n);
                else {
                    sorwakeup(last);
                    sockets++;
                }
            }
        }
        last = rp->rcb_socket;
    }
    if (last) {
        if (sbappendaddr(&last->so_rcv, src,
                         m, (struct mbuf *) 0) == 0)
            m_freem(m);
        else {
            sorwakeup(last);
            sockets++;
        }
    } else
        m_freem(m);
}
```

raw_usrreq.c

Figure 20.17 `raw_input` function: pass routing messages to 0 or more processes.

51–61 In all four calls to `raw_input` that we've seen, the `proto`, `src`, and `dst` arguments are pointers to the three globals `route_proto`, `route_src`, and `route_dst`, which are declared and initialized as shown with Figure 19.26.

Compare address family and protocol

62–67 The `for` loop goes through every routing control block checking for a match. The family in the control block (normally `PF_ROUTE`) must match the family in the `sockproto` structure or the control block is skipped. Next, if the protocol in the control block (the third argument to `socket`) is nonzero, it must match the family in the `sockproto` structure, or the message is skipped. Hence a process that creates a routing socket with a protocol of 0 receives all routing messages.

Compare local and foreign addresses

68–81 These two tests compare the local address in the control block and the foreign address in the control block, if specified. Currently the process is unable to set the `rcb_laddr` or `rcb_faddr` members of the control block. Normally a process would set the former with `bind` and the latter with `connect`, but that is not possible with routing sockets in Net/3. Instead, we'll see that `route_usrreq` permanently connects the socket to the `route_src` socket address structure, which is OK since that is always the `src` argument to this function.

Append message to socket receive buffer

82–107 If `last` is nonnull, it points to the most recently seen `socket` structure that should receive this message. If this variable is nonnull, a copy of the message is appended to that socket's receive buffer by `m_copy` and `sbappendaddr`, and any processes waiting on this receive buffer are awakened. Then `last` is set to point to this socket that just matched the previous tests. The use of `last` is to avoid calling `m_copy` (an expensive operation) if only one process is to receive the message.

If *N* processes are to receive the message, the first *N* – 1 receive a copy and the final one receives the message itself.

The variable `sockets` that is incremented within this function is not used. Since it is incremented only when a message is passed to a process, if it is 0 at the end of the function it indicates that no process received the message (but the value isn't stored anywhere).

20.9 `route_usrreq` Function

`route_usrreq` is the routing protocol's user-request function. It is called for a variety of operations. Figure 20.18 shows the function.

rtsock.c

```
int
route_usrreq(so, req, m, nam, control)
struct socket *so;
int     req;
struct mbuf *m, *nam, *control;
{
```

```
 70     int     error = 0;
 71     struct rawcb *rp = sotorawcb(so);
 72     int     s;

 73     if (req == PRU_ATTACH) {
 74         MALLOC(rp, struct rawcb *, sizeof(*rp), M_PCB, M_WAITOK);
 75         if (so->so_pcb = (caddr_t) rp)
 76             bzero(so->so_pcb, sizeof(*rp));
 77     }
 78     if (req == PRU_DETACH && rp) {
 79         int     af = rp->rcb_proto.sp_protocol;
 80         if (af == AF_INET)
 81             route_cb.ip_count--;
 82         else if (af == AF_NS)
 83             route_cb.ns_count--;
 84         else if (af == AF_ISO)
 85             route_cb.iso_count--;
 86         route_cb.any_count--;
 87     }
 88     s = splnet();
 89     error = raw_usrreq(so, req, m, nam, control);
 90     rp = sotorawcb(so);
 91     if (req == PRU_ATTACH && rp) {
 92         int     af = rp->rcb_proto.sp_protocol;
 93         if (error) {
 94             free((caddr_t) rp, M_PCB);
 95             splx(s);
 96             return (error);
 97         }
 98         if (af == AF_INET)
 99             route_cb.ip_count++;
100         else if (af == AF_NS)
101             route_cb.ns_count++;
102         else if (af == AF_ISO)
103             route_cb.iso_count++;
104         route_cb.any_count++;

105         rp->rcb_faddr = &route_src;
106         soisconnected(so);
107         so->so_options |= SO_USELOOPBACK;
108     }
109     splx(s);
110     return (error);
111 }
```

rtsock.c

Figure 20.18 route_usrreq function: process PRU_*xxx* requests.

PRU_ATTACH: allocate control block

64–77 The PRU_ATTACH request is issued when the process calls socket. Memory is allocated for a routing control block. The pointer returned by MALLOC is stored in the so_pcb member of the socket structure, and if the memory was allocated, the rawcb structure is set to 0.

`PRU_DETACH`: decrement counters

78–87 The `close` system call issues the `PRU_DETACH` request. If the `socket` structure points to a protocol control block, two of the counters in the `route_cb` structure are decremented: one is the `any_count` and one is based on the protocol.

Process request

88–90 The function `raw_usrreq` is called to process the `PRU_`*xxx* request further.

Increment counters

91–104 If the request is `PRU_ATTACH` and the socket points to a routing control block, a check is made for an error from `raw_usrreq`. Two of the counters in the `route_cb` structure are then incremented: one is the `any_count` and one is based on the protocol.

Connect socket

105–106 The foreign address in the routing control block is set to `route_src`. This permanently connects the new socket to receive routing messages from the `PF_ROUTE` family.

Enable `SO_USELOOPBACK` by default

107–111 The `SO_USELOOPBACK` socket option is enabled. This is a socket option that defaults to being enabled—all others default to being disabled.

20.10 `raw_usrreq` Function

`raw_usrreq` performs most of the processing for the user request in the routing domain. It was called by `route_usrreq` in the previous section. The reason the user-request processing is divided between these two functions is that other protocols (e.g., the OSI CLNP) call `raw_usrreq` but not `route_usrreq`. `raw_usrreq` is not intended to be the `pr_usrreq` function for a protocol. Instead it is a common subroutine called by the various `pr_usrreq` functions.

Figure 20.19 shows the beginning and end of the `raw_usrreq` function. The body of the `switch` is discussed in separate figures following this figure.

`PRU_CONTROL` requests invalid

119–129 The `PRU_CONTROL` request is from the `ioctl` system call and is not supported in the routing domain.

Control information invalid

130–133 If control information was passed by the process (using the `sendmsg` system call) an error is returned, since the routing domain doesn't use this optional information.

Socket must have a control block

134–137 If the `socket` structure doesn't point to a routing control block, an error is returned. If a new socket is being created, it is the caller's responsibility (i.e., `route_usrreq`) to allocate this control block and store the pointer in the `so_pcb` member before calling this function.

262–269 The `default` for this `switch` catches two requests that are not handled by `case` statements: `PRU_BIND` and `PRU_CONNECT`. The code for these two requests is present but commented out in Net/3. Therefore issuing the `bind` or `connect` system calls on a *raw_usrreq.c*

```
119 int
120 raw_usrreq(so, req, m, nam, control)
121 struct socket *so;
122 int     req;
123 struct mbuf *m, *nam, *control;
124 {
125     struct rawcb *rp = sotorawcb(so);
126     int     error = 0;
127     int     len;

128     if (req == PRU_CONTROL)
129         return (EOPNOTSUPP);
130     if (control && control->m_len) {
131         error = EOPNOTSUPP;
132         goto release;
133     }
134     if (rp == 0) {
135         error = EINVAL;
136         goto release;
137     }
138     switch (req) {

                                        /* switch cases */

262     default:
263         panic("raw_usrreq");
264     }
265   release:
266     if (m != NULL)
267         m_freem(m);
268     return (error);
269 }
```

raw_usrreq.c

Figure 20.19 Body of `raw_usrreq` function.

routing socket causes a kernel panic. This is a bug. Fortunately it requires a superuser process to create this type of socket.

We now discuss the individual `case` statements. Figure 20.20 shows the processing for the `PRU_ATTACH` and `PRU_DETACH` requests.

139–148 The `PRU_ATTACH` request is a result of the `socket` system call. A routing socket must be created by a superuser process.

149–150 The function `raw_attach` (Figure 20.24) links the control block into the doubly linked list. The `nam` argument is the third argument to `socket` and gets stored in the control block.

151–159 The `PRU_DETACH` is issued by the `close` system call. The test of a null `rp` pointer is superfluous, since the test was already done before the `switch` statement.

160–161 `raw_detach` (Figure 20.25) removes the control block from the doubly linked list.

raw_usrreq.c

```
        /*
         * Allocate a raw control block and fill in the
         * necessary info to allow packets to be routed to
         * the appropriate raw interface routine.
         */
    case PRU_ATTACH:
        if ((so->so_state & SS_PRIV) == 0) {
            error = EACCES;
            break;
        }
        error = raw_attach(so, (int) nam);
        break;

        /*
         * Destroy state just before socket deallocation.
         * Flush data or not depending on the options.
         */
    case PRU_DETACH:
        if (rp == 0) {
            error = ENOTCONN;
            break;
        }
        raw_detach(rp);
        break;
```

raw_usrreq.c

Figure 20.20 `raw_usrreq` function: `PRU_ATTACH` and `PRU_DETACH` requests.

Figure 20.21 shows the processing of the `PRU_CONNECT2`, `PRU_DISCONNECT`, and `PRU_SHUTDOWN` requests.

raw_usrreq.c

```
    case PRU_CONNECT2:
        error = EOPNOTSUPP;
        goto release;

    case PRU_DISCONNECT:
        if (rp->rcb_faddr == 0) {
            error = ENOTCONN;
            break;
        }
        raw_disconnect(rp);
        soisdisconnected(so);
        break;

        /*
         * Mark the connection as being incapable of further input.
         */
    case PRU_SHUTDOWN:
        socantsendmore(so);
        break;
```

raw_usrreq.c

Figure 20.21 `raw_usrreq` function: `PRU_CONNECT2`, `PRU_DISCONNECT`, and `PRU_SHUTDOWN` requests.

186–188 The `PRU_CONNECT2` request is from the `socketpair` system call and is not supported in the routing domain.

189–196 Since a routing socket is always connected (Figure 20.18), the `PRU_DISCONNECT` request is issued by `close` before the `PRU_DETACH` request. The socket must already be connected to a foreign address, which is always true for a routing socket. `raw_disconnect` and `soisdisconnected` complete the processing.

197–202 The `PRU_SHUTDOWN` request is from the `shutdown` system call when the argument specifies that no more writes will be performed on the socket. `socantsendmore` disables further writes.

The most common request for a routing socket, `PRU_SEND`, and the `PRU_ABORT` and `PRU_SENSE` requests are shown in Figure 20.22.

raw_usrreq.c

```
203         /*
204          * Ship a packet out.  The appropriate raw output
205          * routine handles any massaging necessary.
206          */
207     case PRU_SEND:
208         if (nam) {
209             if (rp->rcb_faddr) {
210                 error = EISCONN;
211                 break;
212             }
213             rp->rcb_faddr = mtod(nam, struct sockaddr *);
214         } else if (rp->rcb_faddr == 0) {
215             error = ENOTCONN;
216             break;
217         }
218         error = (*so->so_proto->pr_output) (m, so);
219         m = NULL;
220         if (nam)
221             rp->rcb_faddr = 0;
222         break;

223     case PRU_ABORT:
224         raw_disconnect(rp);
225         sofree(so);
226         soisdisconnected(so);
227         break;

228     case PRU_SENSE:
229         /*
230          * stat: don't bother with a blocksize.
231          */
232         return (0);
```

raw_usrreq.c

Figure 20.22 `raw_usrreq` function: `PRU_SEND`, `PRU_ABORT`, and `PRU_SENSE` requests.

203–217 The `PRU_SEND` request is issued by `sosend` when the process writes to the socket. If a `nam` argument is specified, that is, the process specified a destination address using either `sendto` or `sendmsg`, an error is returned because `route_usrreq` always sets `rcb_faddr` for a routing socket.

218–222 The message in the mbuf chain pointed to by `m` is passed to the protocol's `pr_output` function, which is `route_output`.

223–227 If a `PRU_ABORT` request is issued, the control block is disconnected, the socket is released, and the socket is disconnected.

228–232 The `PRU_SENSE` request is issued by the `fstat` system call. The function returns OK.

Figure 20.23 shows the remaining `PRU_`*xxx* requests.

raw_usrreq.c

```
233         /*
234          * Not supported.
235          */
236     case PRU_RCVOOB:
237     case PRU_RCVD:
238         return (EOPNOTSUPP);

239     case PRU_LISTEN:
240     case PRU_ACCEPT:
241     case PRU_SENDOOB:
242         error = EOPNOTSUPP;
243         break;

244     case PRU_SOCKADDR:
245         if (rp->rcb_laddr == 0) {
246             error = EINVAL;
247             break;
248         }
249         len = rp->rcb_laddr->sa_len;
250         bcopy((caddr_t) rp->rcb_laddr, mtod(nam, caddr_t), (unsigned) len);
251         nam->m_len = len;
252         break;

253     case PRU_PEERADDR:
254         if (rp->rcb_faddr == 0) {
255             error = ENOTCONN;
256             break;
257         }
258         len = rp->rcb_faddr->sa_len;
259         bcopy((caddr_t) rp->rcb_faddr, mtod(nam, caddr_t), (unsigned) len);
260         nam->m_len = len;
261         break;
```

raw_usrreq.c

Figure 20.23 `raw_usrreq` function: final part.

233–243 These five requests are not supported.

244–261 The `PRU_SOCKADDR` and `PRU_PEERADDR` requests are from the `getsockname` and `getpeername` system calls respectively. The former always returns an error, since the `bind` system call, which sets the local address, is not supported in the routing domain. The latter always returns the contents of the socket address structure `route_src`, which was set by `route_usrreq` as the foreign address.

20.11 `raw_attach`, `raw_detach`, and `raw_disconnect` Functions

The `raw_attach` function, shown in Figure 20.24, was called by `raw_input` to finish processing the `PRU_ATTACH` request.

raw_cb.c

```
49 int
50 raw_attach(so, proto)
51 struct socket *so;
52 int     proto;
53 {
54     struct rawcb *rp = sotorawcb(so);
55     int     error;

56     /*
57      * It is assumed that raw_attach is called
58      * after space has been allocated for the
59      * rawcb.
60      */
61     if (rp == 0)
62         return (ENOBUFS);
63     if (error = soreserve(so, raw_sendspace, raw_recvspace))
64         return (error);
65     rp->rcb_socket = so;
66     rp->rcb_proto.sp_family = so->so_proto->pr_domain->dom_family;
67     rp->rcb_proto.sp_protocol = proto;
68     insque(rp, &rawcb);
69     return (0);
70 }
```

raw_cb.c

Figure 20.24 `raw_attach` function.

49–64 The caller must have already allocated the raw protocol control block. `soreserve` sets the high-water marks for the send and receive buffers to 8192. This should be more than adequate for the routing messages.

65–67 A pointer to the `socket` structure is stored in the protocol control block along with the `dom_family` (which is `PF_ROUTE` from Figure 20.1 for the routing domain) and the `proto` argument (which is the third argument to `socket`).

68–70 `insque` adds the control block to the front of the doubly linked list headed by the global `rawcb`.

The `raw_detach` function, shown in Figure 20.25, was called by `raw_input` to finish processing the `PRU_DETACH` request.

75–84 The `so_pcb` pointer in the `socket` structure is set to null and the socket is released. The control block is removed from the doubly linked list by `remque` and the memory used for the control block is released by `free`.

The `raw_disconnect` function, shown in Figure 20.26, was called by `raw_input` to process the `PRU_DISCONNECT` and `PRU_ABORT` requests.

88–94 If the socket does not reference a descriptor, `raw_detach` releases the socket and control block.

raw_cb.c

```
void
raw_detach(rp)
struct rawcb *rp;
{
    struct socket *so = rp->rcb_socket;

    so->so_pcb = 0;
    sofree(so);
    remque(rp);
    free((caddr_t) (rp), M_PCB);
}
```

raw_cb.c

Figure 20.25 `raw_detach` function.

raw_cb.c

```
void
raw_disconnect(rp)
struct rawcb *rp;
{
    if (rp->rcb_socket->so_state & SS_NOFDREF)
        raw_detach(rp);
}
```

raw_cb.c

Figure 20.26 `raw_disconnect` function.

20.12 Summary

A routing socket is a raw socket in the `PF_ROUTE` domain. Routing sockets can be created only by a superuser process. If a nonprivileged process wants to read the routing information contained in the kernel, the `sysctl` system call supported by the routing domain can be used (we described this in the previous chapter).

This chapter was our first encounter with the protocol control blocks (PCBs) that are normally associated with each socket. In the routing domain a special `rawcb` contains information about the routing socket: the local and foreign addresses, the address family, and the protocol. We'll see in Chapter 22 that the larger Internet protocol control block (`inpcb`) is used with UDP, TCP, and raw IP sockets. The concepts are the same, however: the `socket` structure is used by the socket layer, and the PCB, a `rawcb` or an `inpcb`, is used by the protocol layer. The `socket` structure points to the PCB and vice versa.

The `route_output` function handles the five routing requests that can be issued by a process. `raw_input` delivers a routing message to one or more routing sockets, depending on the protocol and address family. The various `PRU_`*xxx* requests for a routing socket are handled by `raw_usrreq` and `route_usrreq`. In later chapters we'll encounter additional *xxx*`_usrreq` functions, one per protocol (UDP, TCP, and raw IP), each consisting of a `switch` statement to handle each request.

Exercises

20.1 List two ways a process can receive the return value from `route_output` when the process writes a message to a routing socket. Which method is more reliable?

20.2 What happens when a process specifies a nonzero *protocol* argument to the `socket` system call, since the `pr_protocol` member of the `routesw` structure is 0?

20.3 Routes in the routing table (other than ARP entries) never time out. Implement a timeout on routes.

21

ARP: Address Resolution Protocol

21.1 Introduction

ARP, the Address Resolution Protocol, handles the translation of 32-bit IP addresses into the corresponding hardware address. For an Ethernet, the hardware addresses are 48-bit Ethernet addresses. In this chapter we only consider mapping IP addresses into 48-bit Ethernet addresses, although ARP is more general and can work with other types of data links. ARP is specified in RFC 826 [Plummer 1982].

When a host has an IP datagram to send to another host on a locally attached Ethernet, the local host first looks up the destination host in the *ARP cache*, a table that maps a 32-bit IP address into its corresponding 48-bit Ethernet address. If the entry is found for the destination, the corresponding Ethernet address is copied into the Ethernet header and the datagram is added to the appropriate interface's output queue. If the entry is not found, the ARP functions hold onto the IP datagram, broadcast an ARP request asking the destination host for its Ethernet address, and, when a reply is received, send the datagram to its destination.

This simple overview handles the common case, but there are many details that we describe in this chapter as we examine the Net/3 implementation of ARP. Chapter 4 of Volume 1 contains additional ARP examples.

21.2 ARP and the Routing Table

The Net/3 implementation of ARP is tied to the routing table, which is why we postponed discussing ARP until we had described the structure of the Net/3 routing tables. Figure 21.1 shows an example that we use in this chapter when describing ARP.

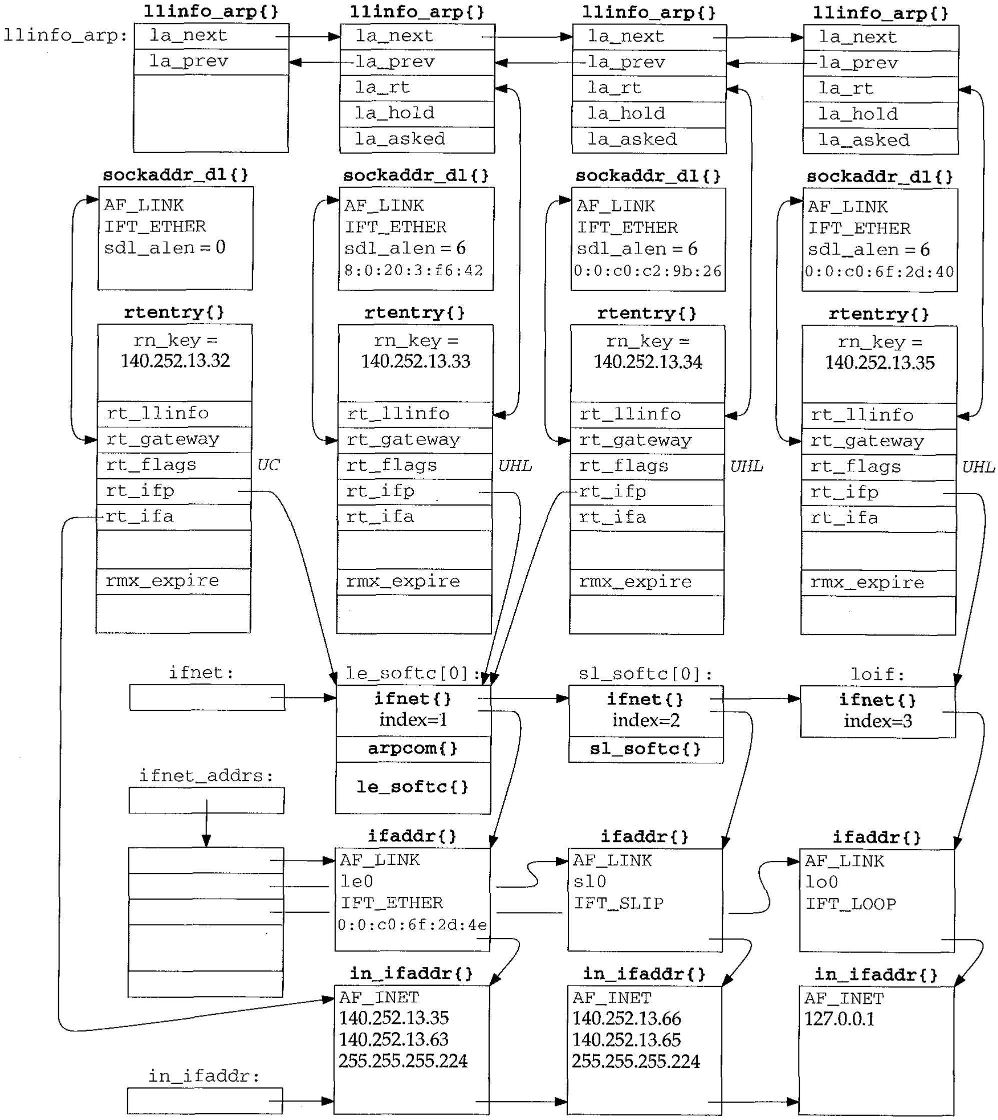

Figure 21.1 Relationship of ARP to routing table and interface structures.

The entire figure corresponds to the example network used throughout the text (Figure 1.17). It shows the ARP entries on the system `bsdi`. The `ifnet`, `ifaddr`, and `in_ifaddr` structures are simplified from Figures 3.32 and 6.5. We have removed some of the details from these three structures, which were covered in Chapters 3 and 6.

For example, we don't show the two `sockaddr_dl` structures that appear after each `ifaddr` structure—instead we summarize the information contained in these two structures. Similarly, we summarize the information contained in the three `in_ifaddr` structures.

We briefly summarize some relevant points from this figure, the details of which we cover as we proceed through the chapter.

1. A doubly linked list of `llinfo_arp` structures contains a minimal amount of information for each hardware address known by ARP. The global `llinfo_arp` is the head of this list. Not shown in this figure is that the `la_prev` pointer of the first entry points to the last entry, and the `la_next` pointer of the last entry points to the first entry. This linked list is processed by the ARP timer function every 5 minutes.
2. For each IP address with a known hardware address, a routing table entry exists (an `rtentry` structure). The `llinfo_arp` structure points to the corresponding `rtentry` structure, and vice versa, using the `la_rt` and `rt_llinfo` pointers. The three routing table entries in this figure with an associated `llinfo_arp` structure are for the hosts `sun` (140.252.13.33), `svr4` (140.252.13.34), and `bsdi` itself (140.252.13.35). These three are also shown in Figure 18.2.
3. We show a fourth routing table entry on the left, without an `llinfo_arp` structure, which is the entry for the interface route to the local Ethernet (140.252.13.32). We show its `rt_flags` with the `C` bit on, since this entry is cloned to form the other three routing table entries. This entry is created by the call to `rtinit` when the IP address is assigned to the interface by `in_ifinit` (Figure 6.19). The other three entries are host entries (the `H` flag) and are generated by ARP (the `L` flag) when a datagram is sent to that IP address.
4. The `rt_gateway` member of the `rtentry` structure points to a `sockaddr_dl` structure. This data-link socket address structure contains the hardware address if the `sdl_alen` member equals 6.
5. The `rt_ifp` member of the routing table entry points to the `ifnet` structure of the outgoing interface. Notice that the two routing table entries in the middle, for other hosts on the local Ethernet, both point to `le_softc[0]`, but the routing table entry on the right, for the host `bsdi` itself, points to the loopback structure. Since `rt_ifp.if_output` (Figure 8.25) points to the output routine, packets sent to the local IP address are routed to the loopback interface.
6. Each routing table entry also points to the corresponding `in_ifaddr` structure. (Actually the `rt_ifa` member points to an `ifaddr` structure, but recall from Figure 6.8 that the first member of an `in_ifaddr` structure is an `ifaddr` structure.) We show only one of these pointers in the figure, although all four point to the same structure. Remember that a single interface, say `le0`, can have multiple IP addresses, each with its own `in_ifaddr` structure, which is why the `rt_ifa` pointer is required in addition to the `rt_ifp` pointer.

7. The `la_hold` member is a pointer to an mbuf chain. An ARP request is broadcast because a datagram is sent to that IP address. While the kernel awaits the ARP reply it holds onto the mbuf chain for the datagram by storing its address in `la_hold`. When the ARP reply is received, the mbuf chain pointed to by `la_hold` is sent.
8. Finally, we show the variable `rmx_expire`, which is in the `rt_metrics` structure within the routing table entry. This value is the timer associated with each ARP entry. Some time after an ARP entry has been created (normally 20 minutes) the ARP entry is deleted.

Even though major routing table changes took place with 4.3BSD Reno, the ARP cache was left alone with 4.3BSD Reno and Net/2. 4.4BSD, however, removed the stand-alone ARP cache and moved the ARP information into the routing table.

The ARP table in Net/2 was an array of structures composed of the following members: an IP address, an Ethernet address, a timer, flags, and a pointer to an mbuf (similar to the `la_hold` member in Figure 21.1). We see with Net/3 that the same information is now spread throughout multiple structures, all of which are linked.

21.3 Code Introduction

There are nine ARP functions in a single C file and definitions in two headers, as shown in Figure 21.2.

File	Description
`net/if_arp.h`	`arphdr` structure definition
`netinet/if_ether.h`	various structure and constant definitions
`netinet/if_ether.c`	ARP functions

Figure 21.2 Files discussed in this chapter.

Figure 21.3 shows the relationship of the ARP functions to other kernel functions. In this figure we also show the relationship between the ARP functions and some of the routing functions from Chapter 19. We describe all these relationships as we proceed through the chapter.

Global Variables

Ten global variables are introduced in this chapter, which are shown in Figure 21.4.

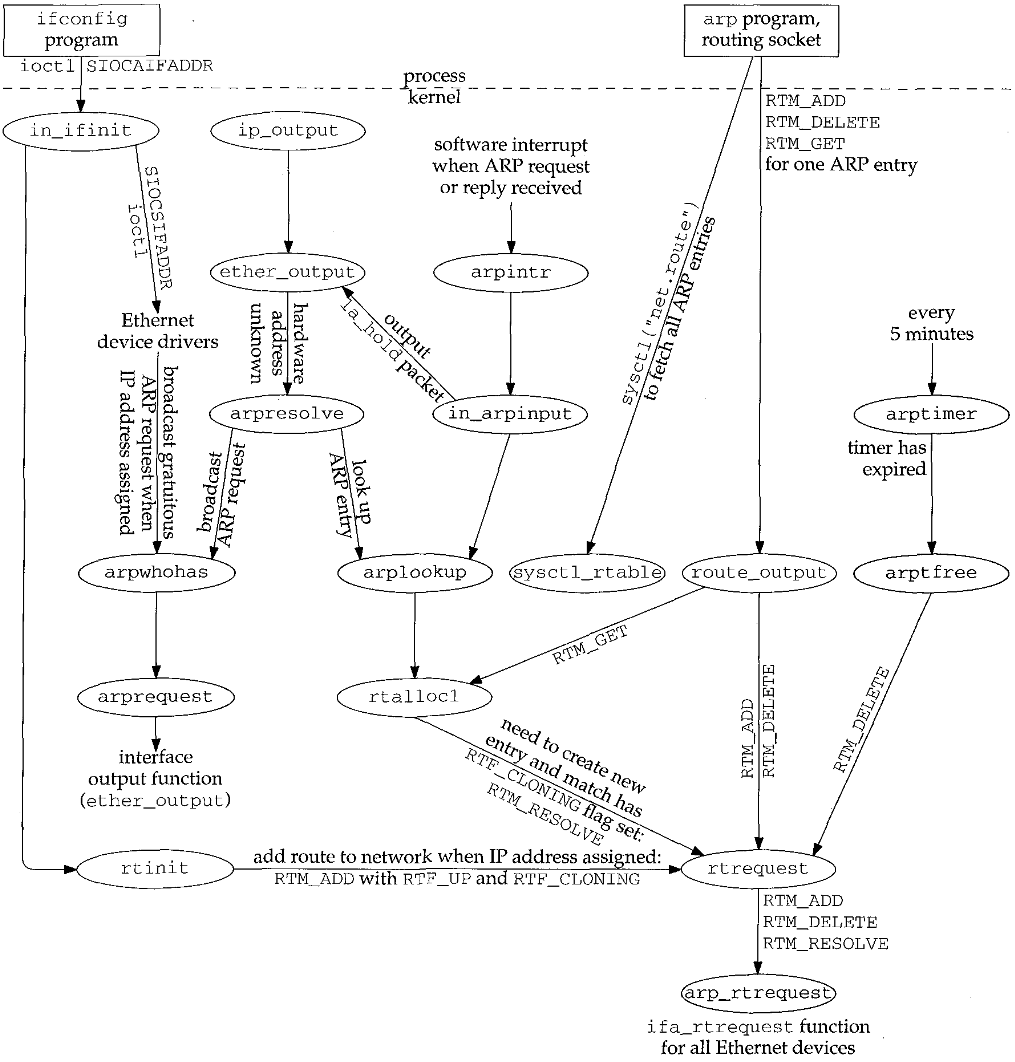

Figure 21.3 Relationship of ARP functions to rest of kernel.

Variable	Datatype	Description
`llinfo_arp`	`struct llinfo_arp`	head of `llinfo_arp` doubly linked list (Figure 21.1)
`arpintrq`	`struct ifqueue`	ARP input queue from Ethernet device drivers (Figure 4.9)
`arpt_prune`	`int`	#minutes between checking ARP list (5)
`arpt_keep`	`int`	#minutes ARP entry valid once resolved (20)
`arpt_down`	`int`	#seconds between ARP flooding algorithm (20)
`arp_inuse`	`int`	#ARP entries currently in use
`arp_allocated`	`int`	#ARP entries ever allocated
`arp_maxtries`	`int`	max #tries for an IP address before pausing (5)
`arpinit_done`	`int`	initialization-performed flag
`useloopback`	`int`	use loopback for local host (default true)

Figure 21.4 Global variables introduced in this chapter.

Statistics

The only statistics maintained by ARP are the two globals `arp_inuse` and `arp_allocated`, from Figure 21.4. The former counts the number of ARP entries currently in use and the latter counts the total number of ARP entries allocated since the system was initialized. Neither counter is output by the `netstat` program, but they can be examined with a debugger.

The entire ARP cache can be listed using the `arp -a` command, which uses the `sysctl` system call with the arguments shown in Figure 19.36. Figure 21.5 shows the output from this command, for the entries shown in Figure 18.2.

```
bsdi $ arp -a
sun.tuc.noao.edu (140.252.13.33) at 8:0:20:3:f6:42
svr4.tuc.noao.edu (140.252.13.34) at 0:0:c0:c2:9b:26
bsdi.tuc.noao.edu (140.252.13.35) at 0:0:c0:6f:2d:40 permanent
ALL-SYSTEMS.MCAST.NET (224.0.0.1) at (incomplete)
```

Figure 21.5 `arp -a` output corresponding to Figure 18.2.

Since the multicast group 224.0.0.1 has the `L` flag set in Figure 18.2, and since the `arp` program looks for entries with the `RTF_LLINFO` flag set, the multicast groups are output by the program. Later in this chapter we'll see why this entry is marked as "incomplete" and why the entry above it is "permanent."

SNMP Variables

As described in Section 25.8 of Volume 1, the original SNMP MIB defined an address translation group that was the system's ARP cache. MIB-II deprecated this group and instead each network protocol group (i.e., IP) contains its own address translation tables. Notice that the change in Net/2 to Net/3 from a stand-alone ARP table to an integration of the ARP information within the IP routing table parallels this SNMP change.

Figure 21.6 shows the IP address translation table from MIB-II, named `ipNetToMediaTable`. The values returned by SNMP for this table are taken from the routing table entry and its corresponding `ifnet` structure.

IP address translation table, index = <*ipNetToMediaIfIndex*>.<*ipNetToMediaNetAddress*>		
Name	Member	Description
`ipNetToMediaIfIndex`	`if_index`	corresponding interface: `ifIndex`
`ipNetToMediaPhysAddress`	`rt_gateway`	physical address
`ipNetToMediaNetAddress`	`rt_key`	IP address
`ipNetToMediaType`	`rt_flags`	type of mapping: 1 = other, 2 = invalidated, 3 = dynamic, 4 = static (see text)

Figure 21.6 IP address translation table: `ipNetToMediaTable`.

If the routing table entry has an expiration time of 0 it is considered permanent and hence "static." Otherwise the entry is considered "dynamic."

21.4 ARP Structures

Figure 21.7 shows the format of an ARP packet when transmitted on an Ethernet.

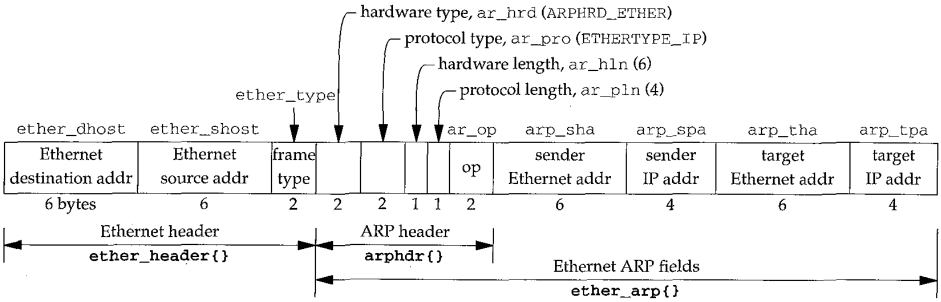

Figure 21.7 Format of an ARP request or reply when used on an Ethernet.

The `ether_header` structure (Figure 4.10) defines the 14-byte Ethernet header; the `arphdr` structure defines the next five fields, which are common to ARP requests and ARP replies on any type of media; and the `ether_arp` structure combines the `arphdr` structure with the sender and target addresses when ARP is used on an Ethernet.

Figure 21.8 shows the definition of the `arphdr` structure. Figure 21.7 shows the values of the first four fields in this structure when ARP is mapping IP addresses to Ethernet addresses.

Figure 21.9 shows the combination of the `arphdr` structure with the fields used with IP addresses and Ethernet addresses, forming the `ether_arp` structure. Notice that ARP uses the terms *hardware* to describe the 48-bit Ethernet address, and *protocol* to describe the 32-bit IP address.

if_arp.h

```
struct arphdr {
    u_short ar_hrd;              /* format of hardware address */
    u_short ar_pro;              /* format of protocol address */
    u_char  ar_hln;              /* length of hardware address */
    u_char  ar_pln;              /* length of protocol address */
    u_short ar_op;               /* ARP/RARP operation, Figure 21.15 */
};
```

if_arp.h

Figure 21.8 `arphdr` structure: common ARP request/reply header.

if_ether.h

```
struct ether_arp {
    struct arphdr ea_hdr;        /* fixed-size header */
    u_char  arp_sha[6];          /* sender hardware address */
    u_char  arp_spa[4];          /* sender protocol address */
    u_char  arp_tha[6];          /* target hardware address */
    u_char  arp_tpa[4];          /* target protocol address */
};

#define arp_hrd ea_hdr.ar_hrd
#define arp_pro ea_hdr.ar_pro
#define arp_hln ea_hdr.ar_hln
#define arp_pln ea_hdr.ar_pln
#define arp_op  ea_hdr.ar_op
```

if_ether.h

Figure 21.9 `ether_arp` structure.

One `llinfo_arp` structure, shown in Figure 21.10, exists for each ARP entry. Additionally, one of these structures is allocated as a global of the same name and used as the head of the linked list of all these structures. We often refer to this list as the *ARP cache,* since it is the only data structure in Figure 21.1 that has a one-to-one correspondence with the ARP entries.

if_ether.h

```
struct llinfo_arp {
    struct llinfo_arp *la_next;
    struct llinfo_arp *la_prev;
    struct rtentry *la_rt;
    struct mbuf *la_hold;        /* last packet until resolved/timeout */
    long    la_asked;            /* #times we've queried for this addr */
};

#define la_timer la_rt->rt_rmx.rmx_expire   /* deletion time in seconds */
```

if_ether.h

Figure 21.10 `llinfo_arp` structure.

With Net/2 and earlier systems it was easy to identify the structure called the *ARP cache,* since a single structure contained everything for each ARP entry. Since Net/3 stores the ARP information among multiple structures, no single structure can be called the *ARP cache.* Nevertheless, having the concept of an ARP cache, which is the collection of information describing a single ARP entry, simplifies the discussion.

104–106 The first two entries form the doubly linked list, which is updated by the `insque` and `remque` functions. `la_rt` points to the associated routing table entry, and the `rt_llinfo` member of the routing table entry points to this structure.

107 When ARP receives an IP datagram to send to another host but the destination's hardware address is not in the ARP cache, an ARP request must be sent and the ARP reply received before the datagram can be sent. While waiting for the reply the mbuf pointer to the datagram is saved in `la_hold`. When the ARP reply is received, the packet pointed to by `la_hold` (if any) is sent.

108–109 `la_asked` counts how many consecutive times an ARP request has been sent to this IP address without receiving a reply. We'll see in Figure 21.24 that when this counter reaches a limit, that host is considered down and another ARP request won't be sent for a while.

110 This definition uses the `rmx_expire` member of the `rt_metrics` structure in the routing table entry as the ARP timer. When the value is 0, the ARP entry is considered permanent. When nonzero, the value is the number of seconds since the Unix Epoch when the entry expires.

21.5 `arpwhohas` Function

The `arpwhohas` function is normally called by `arpresolve` to broadcast an ARP request. It is also called by each Ethernet device driver to issue a *gratuitous ARP* request when the IP address is assigned to the interface (the `SIOCSIFADDR` `ioctl` in Figure 6.28). Section 4.7 of Volume 1 describes gratuitous ARP—it detects if another host on the Ethernet is using the same IP address and also allows other hosts with ARP entries for this host to update their ARP entry if this host has changed its Ethernet address. `arpwhohas` simply calls `arprequest`, shown in the next section, with the correct arguments.

if_ether.c

```
196 void
197 arpwhohas(ac, addr)
198 struct arpcom *ac;
199 struct in_addr *addr;
200 {
201     arprequest(ac, &ac->ac_ipaddr.s_addr, &addr->s_addr, ac->ac_enaddr);
202 }
```

if_ether.c

Figure 21.11 `arpwhohas` function: broadcast an ARP request.

196–202 The `arpcom` structure (Figure 3.26) is common to all Ethernet devices and is part of the `le_softc` structure, for example (Figure 3.20). The `ac_ipaddr` member is a copy of the interface's IP address, which is set by the driver when the `SIOCSIFADDR` `ioctl` is executed (Figure 6.28). `ac_enaddr` is the Ethernet address of the device.

The second argument to this function, `addr`, is the IP address for which the ARP request is being issued: the target IP address. In the case of a gratuitous ARP request, `addr` equals `ac_ipaddr`, so the second and third arguments to `arprequest` are the same, which means the sender IP address will equal the target IP address in the gratuitous ARP request.

21.6 `arprequest` Function

The `arprequest` function is called by `arpwhohas` to broadcast an ARP request. It builds an ARP request packet and passes it to the interface's output function.

Before looking at the source code, let's examine the data structures built by the function. To send the ARP request the interface output function for the Ethernet device (`ether_output`) is called. One argument to `ether_output` is an mbuf containing the data to send: everything that follows the Ethernet type field in Figure 21.7. Another argument is a socket address structure containing the destination address. Normally this destination address is an IP address (e.g., when `ip_output` calls `ether_output` in Figure 21.3). For the special case of an ARP request, the `sa_family` member of the socket address structure is set to `AF_UNSPEC`, which tells `ether_output` that it contains a filled-in Ethernet header, including the destination Ethernet address. This prevents `ether_output` from calling `arpresolve`, which would cause an infinite loop. We don't show this loop in Figure 21.3, but the "interface output function" below `arprequest` is `ether_output`. If `ether_output` were to call `arpresolve` again, the infinite loop would occur.

Figure 21.12 shows the mbuf and the socket address structure built by this function. We also show the two pointers `eh` and `ea`, which are used in the function.

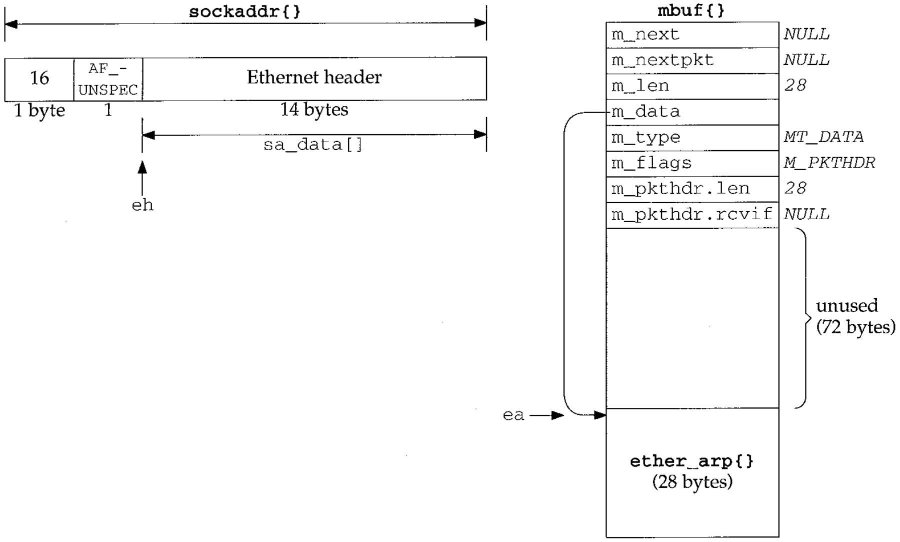

Figure 21.12 `sockaddr` and mbuf built by `arprequest`.

Figure 21.13 shows the `arprequest` function.

if_ether.c

```
209 static void
210 arprequest(ac, sip, tip, enaddr)
211 struct arpcom *ac;
212 u_long *sip, *tip;
213 u_char *enaddr;
214 {
215     struct mbuf *m;
216     struct ether_header *eh;
217     struct ether_arp *ea;
218     struct sockaddr sa;

219     if ((m = m_gethdr(M_DONTWAIT, MT_DATA)) == NULL)
220         return;
221     m->m_len = sizeof(*ea);
222     m->m_pkthdr.len = sizeof(*ea);
223     MH_ALIGN(m, sizeof(*ea));

224     ea = mtod(m, struct ether_arp *);
225     eh = (struct ether_header *) sa.sa_data;
226     bzero((caddr_t) ea, sizeof(*ea));

227     bcopy((caddr_t) etherbroadcastaddr, (caddr_t) eh->ether_dhost,
228           sizeof(eh->ether_dhost));
229     eh->ether_type = ETHERTYPE_ARP;     /* if_output() will swap */

230     ea->arp_hrd = htons(ARPHRD_ETHER);
231     ea->arp_pro = htons(ETHERTYPE_IP);
232     ea->arp_hln = sizeof(ea->arp_sha);  /* hardware address length */
233     ea->arp_pln = sizeof(ea->arp_spa);  /* protocol address length */
234     ea->arp_op = htons(ARPOP_REQUEST);
235     bcopy((caddr_t) enaddr, (caddr_t) ea->arp_sha, sizeof(ea->arp_sha));
236     bcopy((caddr_t) sip, (caddr_t) ea->arp_spa, sizeof(ea->arp_spa));
237     bcopy((caddr_t) tip, (caddr_t) ea->arp_tpa, sizeof(ea->arp_tpa));

238     sa.sa_family = AF_UNSPEC;
239     sa.sa_len = sizeof(sa);

240     (*ac->ac_if.if_output) (&ac->ac_if, m, &sa, (struct rtentry *) 0);
241 }
```

if_ether.c

Figure 21.13 `arprequest` function: build an ARP request packet and send it.

Allocate and initialize mbuf

209–223 A packet header mbuf is allocated and the two length fields are set. `MH_ALIGN` allows room for a 28-byte `ether_arp` structure at the end of the mbuf, and sets the `m_data` pointer accordingly. The reason for moving this structure to the end of the mbuf is to allow `ether_output` to prepend the 14-byte Ethernet header in the same mbuf.

Initialize pointers

224–226 The two pointers `ea` and `eh` are set and the `ether_arp` structure is set to 0. The only purpose of the call to `bzero` is to set the target hardware address to 0, because the other eight fields in this structure are explicitly set to their respective value.

Fill in Ethernet header

227–229 The destination Ethernet address is set to the Ethernet broadcast address and the Ethernet type field is set to `ETHERTYPE_ARP`. Note the comment that this 2-byte field will be converted from host byte order to network byte order by the interface output function. This function also fills in the Ethernet source address field. Figure 21.14 shows the different values for the Ethernet type field.

Constant	Value	Description
ETHERTYPE_IP	0x0800	IP frames
ETHERTYPE_ARP	0x0806	ARP frames
ETHERTYPE_REVARP	0x8035	reverse ARP (RARP) frames
ETHERTYPE_IPTRAILERS	0x1000	trailer encapsulation (deprecated)

Figure 21.14 Ethernet type fields.

RARP maps an Ethernet address to an IP address and is used when a diskless system bootstraps. RARP is normally not part of the kernel's implementation of TCP/IP, so it is not covered in this text. Chapter 5 of Volume 1 describes RARP.

Fill in ARP fields

230–237 All fields in the `ether_arp` structure are filled in, except the target hardware address, which is what the ARP request is looking for. The constant `ARPHRD_ETHER`, which has a value of 1, specifies the format of the hardware addresses as 6-byte Ethernet addresses. To identify the protocol addresses as 4-byte IP addresses, `arp_pro` is set to the Ethernet type field for IP from Figure 21.14. Figure 21.15 shows the various ARP operation codes. We encounter the first two in this chapter. The last two are used with RARP.

Constant	Value	Description
ARPOP_REQUEST	1	ARP request to resolve protocol address
ARPOP_REPLY	2	reply to ARP request
ARPOP_REVREQUEST	3	RARP request to resolve hardware address
ARPOP_REVREPLY	4	reply to RARP request

Figure 21.15 ARP operation codes.

Fill in `sockaddr` and call interface output function

238–241 The `sa_family` member of the socket address structure is set to `AF_UNSPEC` and the `sa_len` member is set to 16. The interface output function is called, which we said is `ether_output`.

21.7 `arpintr` Function

In Figure 4.13 we saw that when `ether_input` receives an Ethernet frame with a type field of `ETHERTYPE_ARP`, it schedules a software interrupt of priority `NETISR_ARP` and appends the frame to ARP's input queue: `arpintrq`. When the kernel processes the software interrupt, the function `arpintr`, shown in Figure 21.16, is called.

if_ether.c

```
319 void
320 arpintr()
321 {
322     struct mbuf *m;
323     struct arphdr *ar;
324     int     s;

325     while (arpintrq.ifq_head) {
326         s = splimp();
327         IF_DEQUEUE(&arpintrq, m);
328         splx(s);
329         if (m == 0 || (m->m_flags & M_PKTHDR) == 0)
330             panic("arpintr");

331         if (m->m_len >= sizeof(struct arphdr) &&
332             (ar = mtod(m, struct arphdr *)) &&
333             ntohs(ar->ar_hrd) == ARPHRD_ETHER &&
334             m->m_len >= sizeof(struct arphdr) + 2*ar->ar_hln + 2*ar->ar_pln)

335                 switch (ntohs(ar->ar_pro)) {
336                 case ETHERTYPE_IP:
337                 case ETHERTYPE_IPTRAILERS:
338                     in_arpinput(m);
339                     continue;
340                 }

341         m_freem(m);
342     }
343 }
```

if_ether.c

Figure 21.16 `arpintr` function: process Ethernet frames containing ARP requests or replies.

319–343 The `while` loop processes one frame at a time, as long as there are frames on the queue. The frame is processed if the hardware type specifies Ethernet addresses, and if the size of the frame is greater than or equal to the size of an `arphdr` structure plus the sizes of two hardware addresses and two protocol addresses. If the type of protocol addresses is either `ETHERTYPE_IP` or `ETHERTYPE_IPTRAILERS`, the `in_arpinput` function, shown in the next section, is called. Otherwise the frame is discarded.

Notice the order of the tests within the `if` statement. The length is checked twice. First, if the length is at least the size of an `arphdr` structure, then the fields in that structure can be examined. The length is checked again, using the two length fields in the `arphdr` structure.

21.8 `in_arpinput` Function

This function is called by `arpintr` to process each received ARP request or ARP reply. While ARP is conceptually simple, numerous rules add complexity to the implementation. The following two scenarios are typical:

1. If a request is received for one of the host's IP addresses, a reply is sent. This is the normal case of some other host on the Ethernet wanting to send this host a packet. Also, since we're about to receive a packet from that other host, and we'll probably send a reply, an ARP entry is created for that host (if one doesn't already exist) because we have its IP address and hardware address. This optimization avoids another ARP exchange when the packet is received from the other host.
2. If a reply is received in response to a request sent by this host, the corresponding ARP entry is now complete (the hardware address is known). The other host's hardware address is stored in the `sockaddr_dl` structure and any queued packet for that host can now be sent. Again, this is the normal case.

ARP requests are normally broadcast so each host sees *all* ARP requests on the Ethernet, even those requests for which it is not the target. Recall from `arprequest` that when a request is sent, it contains the *sender's* IP address and hardware address. This allows the following tests also to occur.

3. If some other host sends a request or reply with a sender IP address that equals this host's IP address, one of the two hosts is misconfigured. Net/3 detects this error and logs a message for the administrator. (We say "request or reply" here because `in_arpinput` doesn't examine the operation type. But ARP replies are normally unicast, in which case only the target host of the reply receives the reply.)
4. If this host receives a request or reply from some other host for which an ARP entry already exists, and if the other host's hardware address has changed, the hardware address in the ARP entry is updated accordingly. This can happen if the other host is shut down and then rebooted with a different Ethernet interface (hence a different hardware address) before its ARP entry times out. The use of this technique, along with the other host sending a gratuitous ARP request when it reboots, prevents this host from being unable to communicate with the other host after the reboot because of an ARP entry that is no longer valid.
5. This host can be configured as a *proxy ARP server.* This means it responds to ARP requests for some other host, supplying the other host's hardware address in the reply. The host whose hardware address is supplied in the proxy ARP reply must be one that is able to forward IP datagrams to the host that is the target of the ARP request. Section 4.6 of Volume 1 discusses proxy ARP.

   A Net/3 system can be configured as a proxy ARP server. These ARP entries are added with the `arp` command, specifying the IP address, hardware address, and the keyword pub. We'll see the support for this in Figure 21.20 and we describe it in Section 21.12.

We examine `in_arpinput` in four parts. Figure 21.17 shows the first part.

if_ether.c

```
358 static void
359 in_arpinput(m)
360 struct mbuf *m;
361 {
362     struct ether_arp *ea;
363     struct arpcom *ac = (struct arpcom *) m->m_pkthdr.rcvif;
364     struct ether_header *eh;
365     struct llinfo_arp *la = 0;
366     struct rtentry *rt;
367     struct in_ifaddr *ia, *maybe_ia = 0;
368     struct sockaddr_dl *sdl;
369     struct sockaddr sa;
370     struct in_addr isaddr, itaddr, myaddr;
371     int     op;

372     ea = mtod(m, struct ether_arp *);
373     op = ntohs(ea->arp_op);
374     bcopy((caddr_t) ea->arp_spa, (caddr_t) & isaddr, sizeof(isaddr));
375     bcopy((caddr_t) ea->arp_tpa, (caddr_t) & itaddr, sizeof(itaddr));

376     for (ia = in_ifaddr; ia; ia = ia->ia_next)
377         if (ia->ia_ifp == &ac->ac_if) {
378             maybe_ia = ia;
379             if ((itaddr.s_addr == ia->ia_addr.sin_addr.s_addr) ||
380                 (isaddr.s_addr == ia->ia_addr.sin_addr.s_addr))
381                 break;
382         }
383     if (maybe_ia == 0)
384         goto out;
385     myaddr = ia ? ia->ia_addr.sin_addr : maybe_ia->ia_addr.sin_addr;
```

if_ether.c

Figure 21.17 `in_arpinput` function: look for matching interface.

358–375 The length of the `ether_arp` structure was verified by the caller, so `ea` is set to point to the received packet. The ARP operation (request or reply) is copied into `op` but it isn't examined until later in the function. The sender's IP address and target IP address are copied into `isaddr` and `itaddr`.

Look for matching interface and IP address

376–382 The linked list of Internet addresses for the host is scanned (the list of `in_ifaddr` structures, Figure 6.5). Remember that a given interface can have multiple IP addresses. Since the received packet contains a pointer (in the mbuf packet header) to the receiving interface's `ifnet` structure, the only IP addresses considered in the `for` loop are those associated with the receiving interface. If either the target IP address or the sender's IP address matches one of the IP addresses for the receiving interface, the `break` terminates the loop.

383–384 If the loop terminates with the variable `maybe_ia` equal to 0, the entire list of configured IP addresses was searched and not one was associated with the received interface. The function jumps to `out` (Figure 21.19), where the mbuf is discarded and the function returns. This should only happen if an ARP request is received on an interface that has been initialized but has not been assigned an IP address.

385 If the `for` loop terminates having located a receiving interface (`maybe_ia` is nonnull) but none of its IP addresses matched the sender or target IP address, `myaddr` is set to the final IP address assigned to the interface. Otherwise (the normal case) `myaddr` contains the local IP address that matched either the sender or target IP address.

Figure 21.18 shows the next part of the `in_arpinput` function, which performs some validation of the packet.

if_ether.c

```
386     if (!bcmp((caddr_t) ea->arp_sha, (caddr_t) ac->ac_enaddr,
387               sizeof(ea->arp_sha)))
388         goto out;                /* it's from me, ignore it. */
389     if (!bcmp((caddr_t) ea->arp_sha, (caddr_t) etherbroadcastaddr,
390               sizeof(ea->arp_sha))) {
391         log(LOG_ERR,
392             "arp: ether address is broadcast for IP address %x!\n",
393             ntohl(isaddr.s_addr));
394         goto out;
395     }
396     if (isaddr.s_addr == myaddr.s_addr) {
397         log(LOG_ERR,
398             "duplicate IP address %x!! sent from ethernet address: %s\n",
399             ntohl(isaddr.s_addr), ether_sprintf(ea->arp_sha));
400         itaddr = myaddr;
401         goto reply;
402     }
```

if_ether.c

Figure 21.18 `in_arpinput` function: validate received packet.

Validate sender's hardware address

386–388 If the sender's hardware address equals the hardware address of the interface, the host received a copy of its own request, which is ignored.

389–395 If the sender's hardware address is the Ethernet broadcast address, this is an error. The error is logged and the packet is discarded.

Check sender's IP address

396–402 If the sender's IP address equals `myaddr`, then the sender is using the same IP address as this host. This is also an error—probably a configuration error by the system administrator on either this host or the sending host. The error is logged and the function jumps to `reply` (Figure 21.19), after setting the target IP address to `myaddr` (the duplicate address). Notice that this ARP packet could have been destined for some other host on the Ethernet—it need not have been sent to this host. Nevertheless, if this form of IP address spoofing is detected, the error is logged and a reply generated.

Figure 21.19 shows the next part of `in_arpinput`.

if_ether.c

```
403     la = arplookup(isaddr.s_addr, itaddr.s_addr == myaddr.s_addr, 0);
404     if (la && (rt = la->la_rt) && (sdl = SDL(rt->rt_gateway))) {
405         if (sdl->sdl_alen &&
406             bcmp((caddr_t) ea->arp_sha, LLADDR(sdl), sdl->sdl_alen))
407             log(LOG_INFO, "arp info overwritten for %x by %s\n",
408                 isaddr.s_addr, ether_sprintf(ea->arp_sha));
409         bcopy((caddr_t) ea->arp_sha, LLADDR(sdl),
410               sdl->sdl_alen = sizeof(ea->arp_sha));
411         if (rt->rt_expire)
412             rt->rt_expire = time.tv_sec + arpt_keep;
413         rt->rt_flags &= ~RTF_REJECT;
414         la->la_asked = 0;
415         if (la->la_hold) {
416             (*ac->ac_if.if_output) (&ac->ac_if, la->la_hold,
417                                     rt_key(rt), rt);
418             la->la_hold = 0;
419         }
420     }

421   reply:
422     if (op != ARPOP_REQUEST) {
423       out:
424         m_freem(m);
425         return;
426     }
```

if_ether.c

Figure 21.19 `in_arpinput` function: create a new ARP entry or update existing entry.

Search routing table for match with sender's IP address

403 `arplookup` searches the ARP cache for the sender's IP address (`isaddr`). The second argument is 1 if the target IP address equals `myaddr` (meaning create a new entry if an entry doesn't exist), or 0 otherwise (do not create a new entry). An entry is always created for the sender if this host is the target; otherwise the host is processing a broadcast intended for some other target, so it just looks for an existing entry for the sender. As mentioned earlier, this means that if a host receives an ARP request for itself from another host, an ARP entry is created for that other host on the assumption that, since that host is about to send us a packet, we'll probably send a reply.

The third argument is 0, which means do not look for a proxy ARP entry (described later). The return value is a pointer to an `llinfo_arp` structure, or a null pointer if an entry is not found or created.

Update existing entry or fill in new entry

404 The code associated with the `if` statement is executed only if the following three conditions are all true:

1. an ARP entry was found or a new ARP entry was successfully created (`la` is nonnull),
2. the ARP entry points to a routing table entry (`rt`), and

3. the `rt_gateway` field of the routing table entry points to a `sockaddr_dl` structure.

The first condition is false for every broadcast ARP request not directed to this host, from some other host whose IP address is not currently in the routing table.

Check if sender's hardware addresses changed

405–408 If the link-level address length (`sdl_alen`) is nonzero (meaning that an existing entry is being referenced and not a new entry that was just created), the link-level address is compared to the sender's hardware address. If they are different, the sender's Ethernet address has changed. This can happen if the sending host is shut down, its Ethernet interface card replaced, and it reboots before the ARP entry times out. While not common, this is a possibility that must be handled. An informational message is logged and the code continues, which will update the hardware address with its new value.

> The sender's IP address in the log message should be converted to host byte order. This is a bug.

Record sender's hardware address

409–410 The sender's hardware address is copied into the `sockaddr_dl` structure pointed to by the `rt_gateway` member of the routing table entry. The link-level address length (`sdl_alen`) in the `sockaddr_dl` structure is also set to 6. This assignment of the length field is required if this is a newly created entry (Exercise 21.3).

Update newly resolved ARP entry

411–412 When the sender's hardware address is resolved, the following steps occur. If the expiration time is nonzero, it is reset to 20 minutes (`arpt_keep`) in the future. This test exists because the `arp` command can create permanent entries: entries that never time out. These entries are marked with an expiration time of 0. We'll also see in Figure 21.24 that when an ARP request is sent (i.e., for a nonpermanent ARP entry) the expiration time is set to the current time, which is nonzero.

413–414 The `RTF_REJECT` flag is cleared and the `la_asked` counter is set to 0. We'll see that these last two steps are used in `arpresolve` to avoid ARP flooding.

415–420 If ARP is holding onto an mbuf awaiting ARP resolution of that host's hardware address (the `la_hold` pointer), the mbuf is passed to the interface output function. (We show this in Figure 21.1.) Since this mbuf was being held by ARP, the destination address must be on a local Ethernet so the interface output function is `ether_output`. This function again calls `arpresolve`, but the hardware address was just filled in, allowing the mbuf to be queued on the actual device's output queue.

Finished with ARP reply packets

421–426 If the ARP operation is not a request, the received packet is discarded and the function returns.

The remainder of the function, shown in Figure 21.20, generates a reply to an ARP request. A reply is generated in only two instances:

1. this host is the target of a request for its hardware address, or
2. this host receives a request for another host's hardware address for which this host has been configured to act as an ARP proxy server.

At this point in the function, an ARP request has been received, but since ARP requests are normally broadcast, the request could be for any system on the Ethernet.

if_ether.c

```
427     if (itaddr.s_addr == myaddr.s_addr) {
428         /* I am the target */
429         bcopy((caddr_t) ea->arp_sha, (caddr_t) ea->arp_tha,
430               sizeof(ea->arp_sha));
431         bcopy((caddr_t) ac->ac_enaddr, (caddr_t) ea->arp_sha,
432               sizeof(ea->arp_sha));
433     } else {
434         la = arplookup(itaddr.s_addr, 0, SIN_PROXY);
435         if (la == NULL)
436             goto out;
437         rt = la->la_rt;
438         bcopy((caddr_t) ea->arp_sha, (caddr_t) ea->arp_tha,
439               sizeof(ea->arp_sha));
440         sdl = SDL(rt->rt_gateway);
441         bcopy(LLADDR(sdl), (caddr_t) ea->arp_sha, sizeof(ea->arp_sha));
442     }

443     bcopy((caddr_t) ea->arp_spa, (caddr_t) ea->arp_tpa, sizeof(ea->arp_spa));
444     bcopy((caddr_t) & itaddr, (caddr_t) ea->arp_spa, sizeof(ea->arp_spa));
445     ea->arp_op = htons(ARPOP_REPLY);
446     ea->arp_pro = htons(ETHERTYPE_IP);  /* let's be sure! */
447     eh = (struct ether_header *) sa.sa_data;
448     bcopy((caddr_t) ea->arp_tha, (caddr_t) eh->ether_dhost,
449           sizeof(eh->ether_dhost));
450     eh->ether_type = ETHERTYPE_ARP;
451     sa.sa_family = AF_UNSPEC;
452     sa.sa_len = sizeof(sa);
453     (*ac->ac_if.if_output) (&ac->ac_if, m, &sa, (struct rtentry *) 0);
454     return;
455 }
```

if_ether.c

Figure 21.20 `in_arpinput` function: form ARP reply and send it.

This host is the target

427–432 If the target IP address equals `myaddr`, this host is the target of the request. The source hardware address is copied into the target hardware address (i.e., whoever sent it becomes the target) and the Ethernet address of the interface is copied from the `arpcom` structure into the source hardware address. The remainder of the ARP reply is constructed after the `else` clause.

Check if this host is a proxy server for target

433–437 Even if this host is not the target, this host can be configured to be a proxy server for the specified target. `arplookup` is called again with the create flag set to 0 (the second argument) and the third argument set to `SIN_PROXY`. This finds an entry in the routing table only if that entry's `SIN_PROXY` flag is set. If an entry is not found (the typical case where this host receives a copy of some other ARP request on the Ethernet), the code at `out` discards the mbuf and returns.

Form proxy reply

437–442 To handle a proxy ARP request, the sender's hardware address becomes the target hardware address and the Ethernet address from the ARP entry is copied into the sender hardware address field. This value from the ARP entry can be the Ethernet address of any host on the Ethernet capable of sending IP datagrams to the target IP address. Normally the host providing the proxy ARP service supplies its own Ethernet address, but that's not required. Proxy entries are created by the system administrator using the `arp` command, with the keyword `pub`, specifying the target IP address (which becomes the key of the routing table entry) and an Ethernet address to return in the ARP reply.

Complete construction of ARP reply packet

443–444 The remainder of the function completes the construction of the ARP reply. The sender and target hardware addresses have been filled in. The sender and target IP addresses are now swapped. The target IP address is contained in `itaddr`, which might have been changed if another host was found using this host's IP address (Figure 21.18).

445–446 The ARP operation is set to `ARPOP_REPLY` and the type of protocol address is set to `ETHERTYPE_IP`. The comment "let's be sure!" is because `arpintr` also calls this function when the type of protocol address is `ETHERTYPE_IPTRAILERS`, but the use of trailer encapsulation is no longer supported.

Fill in `sockaddr` with Ethernet header

447–452 A `sockaddr` structure is filled in with the 14-byte Ethernet header, as shown in Figure 21.12. The target hardware address also becomes the Ethernet destination address.

453–455 The ARP reply is passed to the interface's output routine and the function returns.

21.9 ARP Timer Functions

ARP entries are normally dynamic—they are created when needed and time out automatically. It is also possible for the system administrator to create permanent entries (i.e., no timeout), and the proxy entries we discussed in the previous section are always permanent. Recall from Figure 21.1 and the `#define` at the end of Figure 21.10 that the `rmx_expire` member of the routing metrics structure is used by ARP as a timer.

`arptimer` Function

This function, shown in Figure 21.21, is called every 5 minutes. It goes through all the ARP entries to see if any have expired.

if_ether.c

```
74 static void
75 arptimer(ignored_arg)
76 void    *ignored_arg;
77 {
78     int      s = splnet();
79     struct llinfo_arp *la = llinfo_arp.la_next;

80     timeout(arptimer, (caddr_t) 0, arpt_prune * hz);
81     while (la != &llinfo_arp) {
82         struct rtentry *rt = la->la_rt;
83         la = la->la_next;
84         if (rt->rt_expire && rt->rt_expire <= time.tv_sec)
85             arptfree(la->la_prev);   /* timer has expired, clear */
86     }
87     splx(s);
88 }
```

if_ether.c

Figure 21.21 `arptimer` function: check all ARP timers every 5 minutes.

Set next timeout

80 We'll see that the `arp_rtrequest` function causes `arptimer` to be called the first time, and from that point `arptimer` causes itself to be called 5 minutes (`arpt_prune`) in the future.

Check all ARP entries

81–86 Each entry in the linked list is processed. If the timer is nonzero (it is not a permanent entry) and if the timer has expired, `arptfree` releases the entry. If `rt_expire` is nonzero, it contains a count of the number of seconds since the Unix Epoch when the entry expires.

`arptfree` Function

This function, shown in Figure 21.22, is called by `arptimer` to delete a single entry from the linked list of `llinfo_arp` entries.

Invalidate (don't delete) entries in use

467–473 If the routing table reference count is greater than 0 and the `rt_gateway` member points to a `sockaddr_dl` structure, `arptfree` takes the following steps:

1. the link-layer address length is set to 0,
2. the `la_asked` counter is reset to 0, and
3. the `RTF_REJECT` flag is cleared.

The function then returns. Since the reference count is nonzero, the routing table entry is not deleted. But setting `sdl_alen` to 0 invalidates the entry, so the next time the entry is used, an ARP request will be generated.

if_ether.c

```
459 static void
460 arptfree(la)
461 struct llinfo_arp *la;
462 {
463     struct rtentry *rt = la->la_rt;
464     struct sockaddr_dl *sdl;
465     if (rt == 0)
466         panic("arptfree");
467     if (rt->rt_refcnt > 0 && (sdl = SDL(rt->rt_gateway)) &&
468         sdl->sdl_family == AF_LINK) {
469         sdl->sdl_alen = 0;
470         la->la_asked = 0;
471         rt->rt_flags &= ~RTF_REJECT;
472         return;
473     }
474     rtrequest(RTM_DELETE, rt_key(rt), (struct sockaddr *) 0, rt_mask(rt),
475               0, (struct rtentry **) 0);
476 }
```

if_ether.c

Figure 21.22 `arptfree` function: delete or invalidate an ARP entry.

Delete unreferenced entries

474–475 `rtrequest` deletes the routing table entry, and we'll see in Section 21.13 that it calls `arp_rtrequest`. This latter function frees any mbuf chain held by the ARP entry (the `la_hold` pointer) and deletes the corresponding `llinfo_arp` entry.

21.10 `arpresolve` Function

We saw in Figure 4.16 that `ether_output` calls `arpresolve` to obtain the Ethernet address for an IP address. `arpresolve` returns 1 if the destination Ethernet address is known, allowing `ether_output` to queue the IP datagram on the interface's output queue. A return value of 0 means `arpresolve` does not know the Ethernet address. The datagram is "held" by `arpresolve` (using the `la_hold` member of the `llinfo_arp` structure) and an ARP request is sent. If and when an ARP reply is received, `in_arpinput` completes the ARP entry and sends the held datagram.

`arpresolve` must also avoid *ARP flooding*, that is, it must not repeatedly send ARP requests at a high rate when an ARP reply is not received. This can happen when several datagrams are sent to the same unresolved IP address before an ARP reply is received, or when a datagram destined for an unresolved address is fragmented, since each fragment is sent to `ether_output` as a separate packet. Section 11.9 of Volume 1 contains an example of ARP flooding caused by fragmentation, and discusses the associated problems. Figure 21.23 shows the first half of `arpresolve`.

252–261 `dst` is a pointer to a `sockaddr_in` containing the destination IP address and `desten` is an array of 6 bytes that is filled in with the corresponding Ethernet address, if known.

if_ether.c

```
252 int
253 arpresolve(ac, rt, m, dst, desten)
254 struct arpcom *ac;
255 struct rtentry *rt;
256 struct mbuf *m;
257 struct sockaddr *dst;
258 u_char *desten;
259 {
260     struct llinfo_arp *la;
261     struct sockaddr_dl *sdl;

262     if (m->m_flags & M_BCAST) { /* broadcast */
263         bcopy((caddr_t) etherbroadcastaddr, (caddr_t) desten,
264               sizeof(etherbroadcastaddr));
265         return (1);
266     }
267     if (m->m_flags & M_MCAST) { /* multicast */
268         ETHER_MAP_IP_MULTICAST(&SIN(dst)->sin_addr, desten);
269         return (1);
270     }
271     if (rt)
272         la = (struct llinfo_arp *) rt->rt_llinfo;
273     else {
274         if (la = arplookup(SIN(dst)->sin_addr.s_addr, 1, 0))
275             rt = la->la_rt;
276     }
277     if (la == 0 || rt == 0) {
278         log(LOG_DEBUG, "arpresolve: can't allocate llinfo");
279         m_freem(m);
280         return (0);
281     }
```

if_ether.c

Figure 21.23 `arpresolve` function: find ARP entry if required.

Handle broadcast and multicast destinations

262–270 If the `M_BCAST` flag of the mbuf is set, the destination is filled in with the Ethernet broadcast address and the function returns 1. If the `M_MCAST` flag is set, the `ETHER_MAP_IP_MULTICAST` macro (Figure 12.6) converts the class D address into the corresponding Ethernet address.

Get pointer to `llinfo_arp` structure

271–276 The destination address is a unicast address. If a pointer to a routing table entry is passed by the caller, `la` is set to the corresponding `llinfo_arp` structure. Otherwise `arplookup` searches the routing table for the specified IP address. The second argument is 1, telling `arplookup` to create the entry if it doesn't already exist; the third argument is 0, which means don't look for a proxy ARP entry.

277–281 If either `rt` or `la` are null pointers, one of the allocations failed, since `arplookup` should have created an entry if one didn't exist. An error message is logged, the packet released, and the function returns 0.

Figure 21.24 contains the last half of `arpresolve`. It checks whether the ARP entry is still valid, and, if not, sends an ARP request.

if_ether.c

```
282     sdl = SDL(rt->rt_gateway);
283     /*
284      * Check the address family and length is valid, the address
285      * is resolved; otherwise, try to resolve.
286      */
287     if ((rt->rt_expire == 0 || rt->rt_expire > time.tv_sec) &&
288         sdl->sdl_family == AF_LINK && sdl->sdl_alen != 0) {
289         bcopy(LLADDR(sdl), desten, sdl->sdl_alen);
290         return 1;
291     }
292     /*
293      * There is an arptab entry, but no ethernet address
294      * response yet.  Replace the held mbuf with this
295      * latest one.
296      */
297     if (la->la_hold)
298         m_freem(la->la_hold);
299     la->la_hold = m;

300     if (rt->rt_expire) {
301         rt->rt_flags &= ~RTF_REJECT;
302         if (la->la_asked == 0 || rt->rt_expire != time.tv_sec) {
303             rt->rt_expire = time.tv_sec;
304             if (la->la_asked++ < arp_maxtries)
305                 arpwhohas(ac, &(SIN(dst)->sin_addr));
306             else {
307                 rt->rt_flags |= RTF_REJECT;
308                 rt->rt_expire += arpt_down;
309                 la->la_asked = 0;
310             }
311         }
312     }
313     return (0);
314 }
```

if_ether.c

Figure 21.24 `arpresolve2` function: check if ARP entry valid, send ARP request if not.

Check ARP entry for validity

282–291 Even though an ARP entry is located, it must be checked for validity. The entry is valid if the following conditions are all true:

1. the entry is permanent (the expiration time is 0) or the expiration time is greater than the current time, and
2. the family of the socket address structure pointed to by `rt_gateway` is `AF_LINK`, and
3. the link-level address length (`sdl_alen`) is nonzero.

Recall that `arptfree` invalidated an ARP entry that was still referenced by setting `sdl_alen` to 0. If the entry is valid, the Ethernet address contained in the `sockaddr_dl` is copied into `desten` and the function returns 1.

Hold only most recent IP datagram

292–299 At this point an ARP entry exists but it does not contain a valid Ethernet address. An ARP request must be sent. First the pointer to the mbuf chain is saved in `la_hold`, after releasing any mbuf chain that was already pointed to by `la_hold`. This means that if multiple IP datagrams are sent quickly to a given destination, and an ARP entry does not already exist for the destination, during the time it takes to send an ARP request and receive a reply only the *last* datagram is held, and all prior ones are discarded. An example that generates this condition is NFS. If NFS sends an 8500-byte IP datagram that is fragmented into six IP fragments, and if all six fragments are sent by `ip_output` to `ether_output` in the time it takes to send an ARP request and receive a reply, the first five fragments are discarded and only the final fragment is sent when the reply is received. This in turn causes an NFS timeout, and a retransmission of all six fragments.

Send ARP request but avoid ARP flooding

300–314 RFC 1122 requires ARP to avoid sending ARP requests to a given destination at a high rate when a reply is not received. The technique used by Net/3 to avoid ARP flooding is as follows.

- Net/3 never sends more than one ARP request in any given second to a destination.
- If a reply is not received after five ARP requests (i.e., after about 5 seconds), the `RTF_REJECT` flag in the routing table is set and the expiration time is set for 20 seconds in the future. This causes `ether_output` to refuse to send IP datagrams to this destination for 20 seconds, returning `EHOSTDOWN` or `EHOSTUNREACH` instead (Figure 4.15).
- After the 20-second pause in ARP requests, `arpresolve` will send ARP requests to that destination again.

If the expiration time is nonzero (i.e., this is not a permanent entry) the `RTF_REJECT` flag is cleared, in case it had been set earlier to avoid flooding. The counter `la_asked` counts the number of consecutive times an ARP request has been sent to this destination. If the counter is 0 or if the expiration time does not equal the current time (looking only at the seconds portion of the current time), an ARP request might be sent. This comparison avoids sending more than one ARP request during any second. The expiration time is then set to the current time in seconds (i.e., the microseconds portion, `time.tv_usec` is ignored).

The counter is compared to the limit of 5 (`arp_maxtries`) and then incremented. If the value was less than 5, `arpwhohas` sends the request. If the request equals 5, however, ARP has reached its limit: the `RTF_REJECT` flag is set, the expiration time is set to 20 seconds in the future, and the counter `la_asked` is reset to 0.

Figure 21.25 shows an example to explain further the algorithm used by `arpresolve` and `ether_output` to avoid ARP flooding.

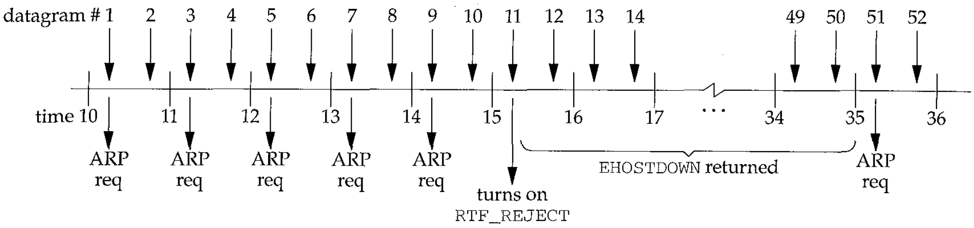

Figure 21.25 Algorithm used to avoid ARP flooding.

We show 26 seconds of time, labeled 10 through 36. We assume a process is sending an IP datagram every one-half second, causing two datagrams to be sent every second. The datagrams are numbered 1 through 52. We also assume that the destination host is down, so there are no replies to the ARP requests. The following actions take place:

- We assume `la_asked` is 0 when datagram 1 is written by the process. `la_hold` is set to point to datagram 1, `rt_expire` is set to the current time (10), `la_asked` becomes 1, and an ARP request is sent. The function returns 0.
- When datagram 2 is written by the process, datagram 1 is discarded and `la_hold` is set to point to datagram 2. Since `rt_expire` equals the current time (10), nothing else happens (an ARP request is not sent) and the function returns 0.
- When datagram 3 is written, datagram 2 is discarded and `la_hold` is set to point to datagram 3. The current time (11) does not equal `rt_expire` (10), so `rt_expire` is set to 11. `la_asked` is less than 5, so `la_asked` becomes 2 and an ARP request is sent.
- When datagram 4 is written, datagram 3 is discarded and `la_hold` is set to point to datagram 4. Since `rt_expire` equals the current time (11), nothing else happens and the function returns 0.
- Similar actions occur for datagrams 5 through 10. After datagram 9 causes an ARP request to be sent, `la_asked` is 5.
- When datagram 11 is written, datagram 10 is discarded and `la_hold` is set to point to datagram 11. The current time (15) does not equal `rt_expire` (14), so `rt_expire` is set to 15. `la_asked` is no longer less than 5, so the ARP flooding avoidance algorithm takes place: `RTF_REJECT` flag is set, `rt_expire` is set to 35 (20 seconds in the future), and `la_asked` is reset to 0. The function returns 0.
- When datagram 12 is written, `ether_output` notices that the `RTF_REJECT` flag is set and that the current time is less than `rt_expire` (35) causing `EHOSTDOWN` to be returned to the sender (normally `ip_output`).
- The `EHOSTDOWN` error is returned for datagrams 13 through 50.

- When datagram 51 is written, even though the `RTF_REJECT` flag is set `ether_output` does not return the error because the current time (35) is no longer less than `rt_expire` (35). `arpresolve` is called and the entire process starts over again: five ARP requests are sent in 5 seconds, followed by a 20-second pause. This continues until the sending process gives up or the destination host responds to an ARP request.

21.11 `arplookup` Function

`arplookup` calls the routing function `rtalloc1` to look up an ARP entry in the Internet routing table. We've seen three calls to `arplookup`:

1. from `in_arpinput` to look up and possibly create an entry corresponding to the source IP address of a received ARP packet,
2. from `in_arpinput` to see if a proxy ARP entry exists for the destination IP address of a received ARP request, and
3. from `arpresolve` to look up or create an entry corresponding to the destination IP address of a datagram that is about to be sent.

If `arplookup` succeeds, a pointer is returned to the corresponding `llinfo_arp` structure; otherwise a null pointer is returned.

`arplookup` has three arguments. The first is the IP address to search for, the second is a flag that is true if the entry is not found and a new entry should be created, and the third is a flag that is true if a proxy ARP entry should be searched for and possibly created.

Proxy ARP entries are handled by defining a different form of the Internet socket address structure, a `sockaddr_inarp` structure, shown in Figure 21.26 This structure is used only by ARP.

if_ether.h

```
111 struct sockaddr_inarp {
112     u_char  sin_len;             /* sizeof(struct sockaddr_inarp) = 16 */
113     u_char  sin_family;          /* AF_INET */
114     u_short sin_port;
115     struct in_addr sin_addr;     /* IP address */
116     struct in_addr sin_srcaddr;  /* not used */
117     u_short sin_tos;             /* not used */
118     u_short sin_other;           /* 0 or SIN_PROXY */
119 };
```

if_ether.h

Figure 21.26 `sockaddr_inarp` structure.

111–119 The first 8 bytes are the same as a `sockaddr_in` structure and the `sin_family` is also set to `AF_INET`. The final 8 bytes, however, are different: the `sin_srcaddr`, `sin_tos`, and `sin_other` members. Of these three, only the final one is used, being set to `SIN_PROXY` (1) if the entry is a proxy entry.

Figure 21.27 shows the `arplookup` function.

if_ether.c

```
480 static struct llinfo_arp *
481 arplookup(addr, create, proxy)
482 u_long  addr;
483 int     create, proxy;
484 {
485     struct rtentry *rt;
486     static struct sockaddr_inarp sin =
487     {sizeof(sin), AF_INET};

488     sin.sin_addr.s_addr = addr;
489     sin.sin_other = proxy ? SIN_PROXY : 0;
490     rt = rtalloc1((struct sockaddr *) &sin, create);
491     if (rt == 0)
492         return (0);
493     rt->rt_refcnt--;
494     if ((rt->rt_flags & RTF_GATEWAY) || (rt->rt_flags & RTF_LLINFO) == 0 ||
495         rt->rt_gateway->sa_family != AF_LINK) {
496         if (create)
497             log(LOG_DEBUG, "arptnew failed on %x\n", ntohl(addr));
498         return (0);
499     }
500     return ((struct llinfo_arp *) rt->rt_llinfo);
501 }
```

if_ether.c

Figure 21.27 `arplookup` function: look up an ARP entry in the routing table.

Initialize `sockaddr_inarp` to look up

480–489 The `sin_addr` member is set to the IP address that is being looked up. The `sin_other` member is set to `SIN_PROXY` if the `proxy` argument is nonzero, or 0 otherwise.

Look up entry in routing table

490–492 `rtalloc1` looks up the IP address in the Internet routing table, creating a new entry if the `create` argument is nonzero. If the entry is not found, the function returns 0 (a null pointer).

Decrement routing table reference count

493 If the entry is found, the reference count for the routing table entry is decremented. This is because ARP is not considered to "hold onto" a routing table entry like the transport layers, so the increment of `rt_refcnt` that was done by the routing table lookup is undone here by ARP.

494–499 If the `RTF_GATEWAY` flag is set, or the `RTF_LLINFO` flag is not set, or the address family of the socket address structure pointed to by `rt_gateway` is not `AF_LINK`, something is wrong and a null pointer is returned. If the entry was created this way, a log message is created.

> The comment in the log message with the function name `arptnew` refers to the older Net/2 function that created ARP entries.

If `rtalloc1` creates a new entry because the matching entry had the `RTF_CLONING` flag set, the function `arp_rtrequest` (which we describe in Section 21.13) is also called by `rtrequest`.

21.12 Proxy ARP

Net/3 supports proxy ARP, as we saw in the previous section. Two different types of proxy ARP entries can be added to the routing table. Both are added with the `arp` command, specifying the `pub` option. Adding a proxy ARP entry always causes a gratuitous ARP request to be issued by `arp_rtrequest` (Figure 21.28) because the `RTF_ANNOUNCE` flag is set when the entry is created.

The first type of proxy ARP entry allows an IP address for a host on an attached network to be entered into the ARP cache. Any Ethernet address can be assigned to the entry. These entries are added to the routing table with an explicit mask of `0xffffffff`. The purpose of this mask is to allow the call to `rtalloc1` in Figure 21.27 to match this entry, even if the `SIN_PROXY` flag is set in the socket address structure of the search key. This in turn allows the call to `arplookup` from Figure 21.20 to match this entry when a search is made for the target address with the `SIN_PROXY` flag set.

This type of entry can be used if a host H1 that doesn't implement ARP is on an attached network. The host with the proxy entry answers all ARP requests for H1's hardware address, supplying the Ethernet address that was specified when the proxy entry was created (i.e., the Ethernet address of H1). These entries are output with the notation "published" by the `arp -a` command.

The second type of proxy ARP entry is for a host for which a routing table entry already exists. The kernel creates another routing table entry for the destination, with this new entry containing the link-layer information (i.e., the Ethernet address). The `SIN_PROXY` flag is set in the `sin_other` member of the `sockaddr_inarp` structure (Figure 21.26) in the new routing table entry. Recall that routing table searches compare 12 bytes of the Internet socket address structure (Figure 18.39). This use of the `SIN_PROXY` flag is the only time the final 8 bytes of the structure are nonzero. When `arplookup` specifies the `SIN_PROXY` value in the `sin_other` member of the structure passed to `rtalloc1`, the only entries in the routing table that will match are ones that also have the `SIN_PROXY` flag set.

This type of entry normally specifies the Ethernet address of the host acting as the proxy server. If the proxy entry was created for a host HD, the sequence of steps is as follows.

1. The proxy server receives a broadcast ARP request for HD's hardware address from some other host HS. The host HS thinks HD is on the local network.
2. The proxy server responds, supplying its own Ethernet address.
3. HS sends the datagram with a destination IP address of HD to the proxy server's Ethernet address.

4. The proxy server receives the datagram for HD and forwards it, using the normal routing table entry for HD.

This type of entry was used on the router `netb` in the example in Section 4.6 of Volume 1. These entries are output by the `arp -a` command with the notation "published (proxy only)."

21.13 `arp_rtrequest` Function

Figure 21.3 provides an overview of the relationship between the ARP functions and the routing functions. We've encountered two calls to the routing table functions from the ARP functions.

1. `arplookup` calls `rtalloc1` to look up an ARP entry and possibly create a new entry if a match isn't found.

   If a matching entry is found in the routing table and the `RTF_CLONING` flag is not set (i.e., it is a matching entry for the destination host), the pointer to the matching entry is returned. But if the `RTF_CLONING` bit is set, `rtalloc1` calls `rtrequest` with a command of `RTM_RESOLVE`. This is how the entries for 140.252.13.33 and 140.252.13.34 in Figure 18.2 were created—they were cloned from the entry for 140.252.13.32.
2. `arptfree` calls `rtrequest` with a command of `RTM_DELETE` to delete an entry from the routing table that corresponds to an ARP entry.

Additionally, the `arp` command manipulates the ARP cache by sending and receiving routing messages on a routing socket. The `arp` command issues routing messages with commands of `RTM_ADD`, `RTM_DELETE`, and `RTM_GET`. The first two commands cause `rtrequest` to be called and the third causes `rtalloc1` to be called.

Finally, when an Ethernet device driver has an IP address assigned to the interface, `rtinit` adds a route to the network. This causes `rtrequest` to be called with a command of `RTM_ADD` and with the flags of `RTF_UP` and `RTF_CLONING`. This is how the entry for 140.252.13.32 in Figure 18.2 was created.

As described in Chapter 19, each `ifaddr` structure can contain a pointer to a function (the `ifa_rtrequest` member) that is automatically called when a routing table entry is added or deleted for that interface. We saw in Figure 6.17 that `in_ifinit` sets this pointer to the function `arp_rtrequest` for all Ethernet devices. Therefore, whenever the routing functions are called to add or delete a routing table entry for ARP, `arp_rtrequest` is also called. The purpose of this function is to do whatever type of initialization or cleanup is required above and beyond what the generic routing table functions perform. For example, this is where a new `llinfo_arp` structure is allocated and initialized whenever a new ARP entry is created. In a similar way, the `llinfo_arp` structure is deleted by this function after the generic routing routines have completed processing an `RTM_DELETE` command.

Figure 21.28 shows the first part of the `arp_rtrequest` function.

if_ether.c

```
void
arp_rtrequest(req, rt, sa)
int     req;
struct rtentry *rt;
struct sockaddr *sa;
{
    struct sockaddr *gate = rt->rt_gateway;
    struct llinfo_arp *la = (struct llinfo_arp *) rt->rt_llinfo;
    static struct sockaddr_dl null_sdl =
    {sizeof(null_sdl), AF_LINK};

    if (!arpinit_done) {
        arpinit_done = 1;
        timeout(arptimer, (caddr_t) 0, hz);
    }
    if (rt->rt_flags & RTF_GATEWAY)
        return;
    switch (req) {

    case RTM_ADD:
        /*
         * XXX: If this is a manually added route to interface
         * such as older version of routed or gated might provide,
         * restore cloning bit.
         */
        if ((rt->rt_flags & RTF_HOST) == 0 &&
            SIN(rt_mask(rt))->sin_addr.s_addr != 0xffffffff)
            rt->rt_flags |= RTF_CLONING;
        if (rt->rt_flags & RTF_CLONING) {
            /*
             * Case 1: This route should come from a route to iface.
             */
            rt_setgate(rt, rt_key(rt),
                       (struct sockaddr *) &null_sdl);
            gate = rt->rt_gateway;
            SDL(gate)->sdl_type = rt->rt_ifp->if_type;
            SDL(gate)->sdl_index = rt->rt_ifp->if_index;
            rt->rt_expire = time.tv_sec;
            break;
        }
        /* Announce a new entry if requested. */
        if (rt->rt_flags & RTF_ANNOUNCE)
            arprequest((struct arpcom *) rt->rt_ifp,
                       &SIN(rt_key(rt))->sin_addr.s_addr,
                       &SIN(rt_key(rt))->sin_addr.s_addr,
                       (u_char *) LLADDR(SDL(gate)));
        /* FALLTHROUGH */
```

if_ether.c

Figure 21.28 `arp_rtrequest` function: `RTM_ADD` command.

Initialize ARP timeout function

92–105 The first time `arp_rtrequest` is called (when the first Ethernet interface is assigned an IP address during system initialization), the `timeout` function schedules the function `arptimer` to be called in 1 clock tick. This starts the ARP timer code running every 5 minutes, since `arptimer` always calls `timeout`.

Ignore indirect routes

106–107 If the `RTF_GATEWAY` flag is set, the function returns. This flag indicates an indirect routing table entry and all ARP entries are direct routes.

108 The remainder of the function is a `switch` with three `cases`: `RTM_ADD`, `RTM_RESOLVE`, and `RTM_DELETE`. (The latter two are shown in figures that follow.)

`RTM_ADD` command

109 The first `case` for `RTM_ADD` is invoked by either the `arp` command manually creating an ARP entry or by an Ethernet interface being assigned an IP address by `rtinit` (Figure 21.3).

Backward compatibility

110–117 If the `RTF_HOST` flag is cleared, this routing table entry has an associated mask (i.e., it is a network route, not a host route). If that mask is not all one bits, then the entry is really a route to an interface, so the `RTF_CLONING` flag is set. As the comment indicates, this is for backward compatibility with older versions of some routing daemons. Also, the command

```
route add -net 224.0.0.0 -interface bsdi
```

that is in the file `/etc/netstart` creates the entry for this network shown in Figure 18.2 that has the `RTF_CLONING` flag set.

Initialize entry for network route to interface

118–126 If the `RTF_CLONING` flag is set (which `in_ifinit` sets for all Ethernet interfaces), this entry is probably being added by `rtinit`. `rt_setgate` allocates space for a `sockaddr_dl` structure, which is pointed to by the `rt_gateway` member. This data-link socket address structure is the one associated with the routing table entry for 140.252.13.32 in Figure 21.1. The `sdl_len` and `sdl_family` members are initialized from the `static` definition of `null_sdl` at the beginning of the function, and the `sdl_type` (probably `IFT_ETHER`) and `sdl_index` members are copied from the interface's `ifnet` structure. This structure never contains an Ethernet address and the `sdl_alen` member remains 0.

127–128 Finally, the expiration time is set to the current time, which is simply the time the entry was created, and the `break` causes the function to return. For entries created at system initialization, their `rmx_expire` value is the time at which the system was bootstrapped. Notice in Figure 21.1 that this routing table entry does not have an associated `llinfo_arp` structure, so it is never processed by `arptimer`. Nevertheless this `sockaddr_dl` structure is used: since it is the `rt_gateway` structure for the entry that is cloned for host-specific entries on this Ethernet, it is copied by `rtrequest` when the newly cloned entries are created with the `RTM_RESOLVE` command. Also, the `netstat` program prints the `sdl_index` value as `link#`*n*, as we see in Figure 18.2.

Send gratuitous ARP request

130–135 If the `RTF_ANNOUNCE` flag is set, this entry is being created by the `arp` command with the `pub` option. This option has two ramifications: (1) the `SIN_PROXY` flag will be set in the `sin_other` member of the `sockaddr_inarp` structure, and (2) the `RTF_ANNOUNCE` flag will be set. Since the `RTF_ANNOUNCE` flag is set, `arprequest` broadcasts a gratuitous ARP request. Notice that the second and third arguments are the same, which causes the sender IP address to equal the target IP address in the ARP request.

136 The code falls through to the `case` for the `RTM_RESOLVE` command.

Figure 21.29 shows the next part of the `arp_rtrequest` function, which handles the `RTM_RESOLVE` command. This command is issued when `rtalloc1` matches an entry with the `RTF_CLONING` flag set and its second argument is nonzero (the `create` argument to `arplookup`). A new `llinfo_arp` structure must be allocated and initialized.

Verify `sockaddr_dl` structure

137–144 The family and length of the `sockaddr_dl` structure pointed to by the `rt_gateway` pointer are verified. The interface type (probably `IFT_ETHER`) and index are then copied into the new `sockaddr_dl` structure.

Handle route changes

145–146 Normally the routing table entry is new and does not point to an `llinfo_arp` structure. If the `la` pointer is nonnull, however, `arp_rtrequest` was called when a route changed for an existing routing table entry. Since the `llinfo_arp` structure is already allocated, the `break` causes the function to return.

Initialize `llinfo_arp` structure

147–158 An `llinfo_arp` structure is allocated and its pointer is stored in the `rt_llinfo` pointer of the routing table entry. The two statistics `arp_inuse` and `arp_allocated` are incremented and the `llinfo_arp` structure is set to 0. This sets `la_hold` to a null pointer and `la_asked` to 0.

159–161 The `rt` pointer is stored in the `llinfo_arp` structure and the `RTF_LLINFO` flag is set. In Figure 18.2 we see that the three routing table entries created by ARP, 140.252.13.33, 140.252.13.34, and 140.252.13.35, all have the `L` flag enabled, as does the entry for 224.0.0.1. Recall that the `arp` program looks only for entries with this flag (Figure 19.36). Finally the new structure is added to the front of the linked list of `llinfo_arp` structures by `insque`.

The ARP entry has been created: `rtrequest` creates the routing table entry (often cloning a network-specific entry for the Ethernet) and `arp_rtrequest` allocates and initializes an `llinfo_arp` structure. All that remains is for an ARP request to be broadcast so that an ARP reply can fill in the host's Ethernet address. In the common sequence of events, `arp_rtrequest` is called because `arpresolve` called `arplookup` (the intermediate sequence of function calls can be followed in Figure 21.3). When control returns to `arpresolve`, it broadcasts the ARP request.

if_ether.c

```
    case RTM_RESOLVE:
        if (gate->sa_family != AF_LINK ||
            gate->sa_len < sizeof(null_sdl)) {
            log(LOG_DEBUG, "arp_rtrequest: bad gateway value");
            break;
        }
        SDL(gate)->sdl_type = rt->rt_ifp->if_type;
        SDL(gate)->sdl_index = rt->rt_ifp->if_index;
        if (la != 0)
            break;              /* This happens on a route change */
        /*
         * Case 2:  This route may come from cloning, or a manual route
         * add with a LL address.
         */
        R_Malloc(la, struct llinfo_arp *, sizeof(*la));
        rt->rt_llinfo = (caddr_t) la;
        if (la == 0) {
            log(LOG_DEBUG, "arp_rtrequest: malloc failed\n");
            break;
        }
        arp_inuse++, arp_allocated++;
        Bzero(la, sizeof(*la));

        la->la_rt = rt;
        rt->rt_flags |= RTF_LLINFO;
        insque(la, &llinfo_arp);

        if (SIN(rt_key(rt))->sin_addr.s_addr ==
            (IA_SIN(rt->rt_ifa))->sin_addr.s_addr) {
            /*
             * This test used to be
             *  if (loif.if_flags & IFF_UP)
             * It allowed local traffic to be forced
             * through the hardware by configuring the loopback down.
             * However, it causes problems during network configuration
             * for boards that can't receive packets they send.
             * It is now necessary to clear "useloopback" and remove
             * the route to force traffic out to the hardware.
             */
            rt->rt_expire = 0;
            Bcopy(((struct arpcom *) rt->rt_ifp)->ac_enaddr,
                  LLADDR(SDL(gate)), SDL(gate)->sdl_alen = 6);
            if (useloopback)
                rt->rt_ifp = &loif;

        }
        break;
```

if_ether.c

Figure 21.29 `arp_rtrequest` function: `RTM_RESOLVE` command.

Handle local host specially

162–173 This portion of code is a special test that is new with 4.4BSD (although the comment is left over from earlier releases). It creates the rightmost routing table entry in Figure 21.1 with a key consisting of the local host's IP address (140.252.13.35). The `if` test checks whether the routing table key equals the IP address of the interface. If so, the entry that was just created (probably as a clone of the interface entry) refers to the local host.

Make entry permanent and set Ethernet address

174–176 The expiration time is set to 0, making the entry permanent—it will never time out. The Ethernet address is copied from the `arpcom` structure of the interface into the `sockaddr_dl` structure pointed to by the `rt_gateway` member.

Set interface pointer to loopback interface

177–178 If the global `useloopback` is nonzero (it defaults to 1), the interface pointer in the routing table entry is changed to point to the loopback interface. This means that any datagrams sent to the host's own IP address are sent to the loopback interface instead. Prior to 4.4BSD, the route from the host's own IP address to the loopback interface was established using a command of the form

```
route add 140.252.13.35 127.0.0.1
```

in the `/etc/netstart` file. Although this still works with 4.4BSD, it is unnecessary because the code we just looked at creates an equivalent route automatically, the first time an IP datagram is sent to the host's own IP address. Also realize that this piece of code is executed only once per interface. Once the routing table entry and the permanent ARP entry are created, they don't expire, so another `RTM_RESOLVE` for this IP address won't occur.

The final part of `arp_rtrequest`, shown in Figure 21.30, handles the `RTM_DELETE` request. From Figure 21.3 we see that this command can be generated from the `arp` command, to delete an entry manually, and from the `arptfree` function, when an ARP entry times out.

if_ether.c

```
    case RTM_DELETE:
        if (la == 0)
            break;
        arp_inuse--;
        remque(la);
        rt->rt_llinfo = 0;
        rt->rt_flags &= ~RTF_LLINFO;
        if (la->la_hold)
            m_freem(la->la_hold);
        Free((caddr_t) la);
    }
}
```

if_ether.c

Figure 21.30 `arp_rtrequest` function: `RTM_DELETE` command.

Verify `la` pointer

182–183 The `la` pointer should always be nonnull (that is, the routing table entry should always point to an `llinfo_arp` structure); otherwise the `break` causes the function to return.

Delete `llinfo_arp` structure

184–190 The `arp_inuse` statistic is decremented and the `llinfo_arp` structure is removed from the doubly linked list by `remque`. The `rt_llinfo` pointer is set to 0 and the `RTF_LLINFO` flag is cleared. If an mbuf is held by the ARP entry (i.e., an ARP request is outstanding), that mbuf is released. Finally the `llinfo_arp` structure is released.

Notice that the `switch` statement does not provide a `default case` and does not provide a `case` for the `RTM_GET` command. This is because the `RTM_GET` command issued by the `arp` program is handled entirely by the `route_output` function, and `rtrequest` is not called. Also, the call to `rtalloc1` that we show in Figure 21.3, which is caused by an `RTM_GET` command, specifies a second argument of 0; therefore `rtalloc1` does not call `rtrequest` in this case.

21.14 ARP and Multicasting

If an IP datagram is destined for a multicast group, `ip_output` checks whether the process has assigned a specific interface to the socket (Figure 12.40), and if so, the datagram is sent out that interface. Otherwise, `ip_output` selects the outgoing interface using the normal IP routing table (Figure 8.24). Therefore, on a system with more than one multicast-capable interface, the IP routing table specifies the default interface for each multicast group.

We saw in Figure 18.2 that an entry was created in our routing table for the 224.0.0.0 network and since that entry has its "clone" flag set, all multicast groups starting with 224 had the associated interface (`le0`) as its default. Additional routing table entries can be created for the other multicast groups (the ones beginning with 225–239), or specific entries can be created for particular multicast groups to assign an explicit default. For example, a routing table entry could be created for 224.0.1.1 (the network time protocol) with an interface that differs from the interface for 224.0.0.0. If an entry for a multicast group does not exist in the routing table, and the process doesn't specify an interface with the `IP_MULTICAST_IF` socket option, the default interface for the group becomes the interface associated with the "default" route in the table. In Figure 18.2 the entry for 224.0.0.0 isn't really needed, since both it and the default route use the interface `le0`.

Once the interface is selected, if the interface is an Ethernet, `arpresolve` is called to convert the multicast group address into its corresponding Ethernet address. In Figure 21.23 this was done by invoking the macro `ETHER_MAP_IP_MULTICAST`. Since this simple macro logically ORs the low-order 23 bits of the multicast group with a constant (Figure 12.6), an ARP request–reply is not required and the mapping does not need to go into the ARP cache. The macro is just invoked each time the conversion is required.

Multicast group addresses appear in the Net/3 ARP cache if the multicast group is cloned from another entry, as we saw in Figure 21.5. This is because these entries have the `RTF_LLINFO` flag set. These are not true ARP entries because they do not require an ARP request–reply, and they do not have an associated link-layer address, since the mapping is done when needed by the `ETHER_MAP_IP_MULTICAST` macro.

The timeout of the ARP entries for these multicast group addresses is different from normal ARP entries. When a routing table entry is created for a multicast group, such as the entry for 224.0.0.1 in Figure 18.2, `rtrequest` copies the `rt_metrics` structure from the entry being cloned (Figure 19.9). We mentioned with Figure 21.28 that the network entry has an `rmx_expire` value of the time the `RTM_ADD` command was executed, normally the time the system was initialized. The new entry for 224.0.0.1 has this same expiration time.

This means the ARP entry for a multicast group such as 224.0.0.1 expires the next time `arptimer` executes, because its expiration time is always in the past. The entry is created again the next time it is looked up in the routing table.

21.15 Summary

ARP provides the dynamic mapping between IP addresses and hardware addresses. This chapter has examined an implementation of ARP that maps IP addresses to Ethernet addresses.

The Net/3 implementation is a major change from previous BSD releases. The ARP information is now stored in various structures: the routing table, a data-link socket address structure, and an `llinfo_arp` structure. Figure 21.1 shows the relationships between all the structures.

Sending an ARP request is simple: the appropriate fields are filled in and the request is sent as a broadcast. Processing a received request is more complicated because each host receives *all* broadcast ARP requests. Besides responding to requests for one of the host's IP addresses, `in_arpinput` also checks that some other host isn't using the host's IP address. Since all ARP requests contain the sender's IP and hardware addresses, any host on the Ethernet can use this information to update an existing ARP entry for the sender.

ARP flooding can be a problem on a LAN and Net/3 is the first BSD release to handle this. A maximum of one ARP request per second is sent to any given destination, and after five consecutive requests without a reply, a 20-second pause occurs before another ARP request is sent to that destination.

Exercises

21.1 What assumption is made in the assignment of the local variable `ac` in Figure 21.17?

21.2 If we ping the broadcast address of the local Ethernet and then execute `arp -a`, we see that this causes the ARP cache to be filled with entries for almost every other host on the local Ethernet. Why?

21.3 Follow through the code and explain why the assignment of 6 to `sdl_alen` is required in Figure 21.19.

21.4 With the separate ARP table in Net/2, independent of the routing table, each time `arpresolve` was called, a search was made of the ARP table. Compare this to the Net/3 approach. Which is more efficient?

21.5 The ARP code in Net/2 explicitly set a timeout of 3 minutes for an incomplete entry in the ARP cache, that is, for an entry that is awaiting an ARP reply. We've never explicitly said how Net/3 handles this timeout. When does Net/3 time out an incomplete ARP entry?

21.6 What changes in the avoidance of ARP flooding when a Net/3 system is acting as a router and the packets that cause the flooding are from some other host?

21.7 What are the values of the four `rmx_expire` variables shown in Figure 21.1? Where in the code are the values set?

21.8 What change would be required to the code in this chapter to cause an ARP entry to be created for every host that broadcasts an ARP request?

21.9 To verify the example in Figure 21.25 the authors ran the `sock` program from Appendix C of Volume 1, writing a UDP datagram every 500 ms to a nonexistent host on the local Ethernet. (The `-p` option of the program was modified to allow millisecond waits.) But only 10 UDP datagrams were sent without an error, instead of the 11 shown in Figure 21.25, before the first `EHOSTDOWN` error was returned. Why?

21.10 Modify ARP to hold onto *all* packets for a destination, awaiting an ARP reply, instead of just the most recent one. What are the implications of this change? Should there be a limit, as there is for each interface's output queue? Are any changes required to the data structures?

22

Protocol Control Blocks

22.1 Introduction

Protocol control blocks (PCBs) are used at the protocol layer to hold the various pieces of information required for each UDP or TCP socket. The Internet protocols maintain *Internet protocol control blocks* and *TCP control blocks*. Since UDP is connectionless, everything it needs for an end point is found in the Internet PCB; there are no UDP control blocks.

The Internet PCB contains the information common to all UDP and TCP end points: foreign and local IP addresses, foreign and local port numbers, IP header prototype, IP options to use for this end point, and a pointer to the routing table entry for the destination of this end point. The TCP control block contains all of the state information that TCP maintains for each connection: sequence numbers in both directions, window sizes, retransmission timers, and the like.

In this chapter we describe the Internet PCBs used in Net/3, saving TCP's control blocks until we describe TCP in detail. We examine the numerous functions that operate on Internet PCBs, since we'll encounter them when we describe UDP and TCP. Most of the functions begin with the six characters `in_pcb`.

Figure 22.1 summarizes the protocol control blocks that we describe and their relationship to the `file` and `socket` structures. There are numerous points to consider in this figure.

- When a socket is created by either `socket` or `accept`, the socket layer creates a `file` structure and a `socket` structure. The file type is `DTYPE_SOCKET` and the socket type is `SOCK_DGRAM` for UDP end points or `SOCK_STREAM` for TCP end points.

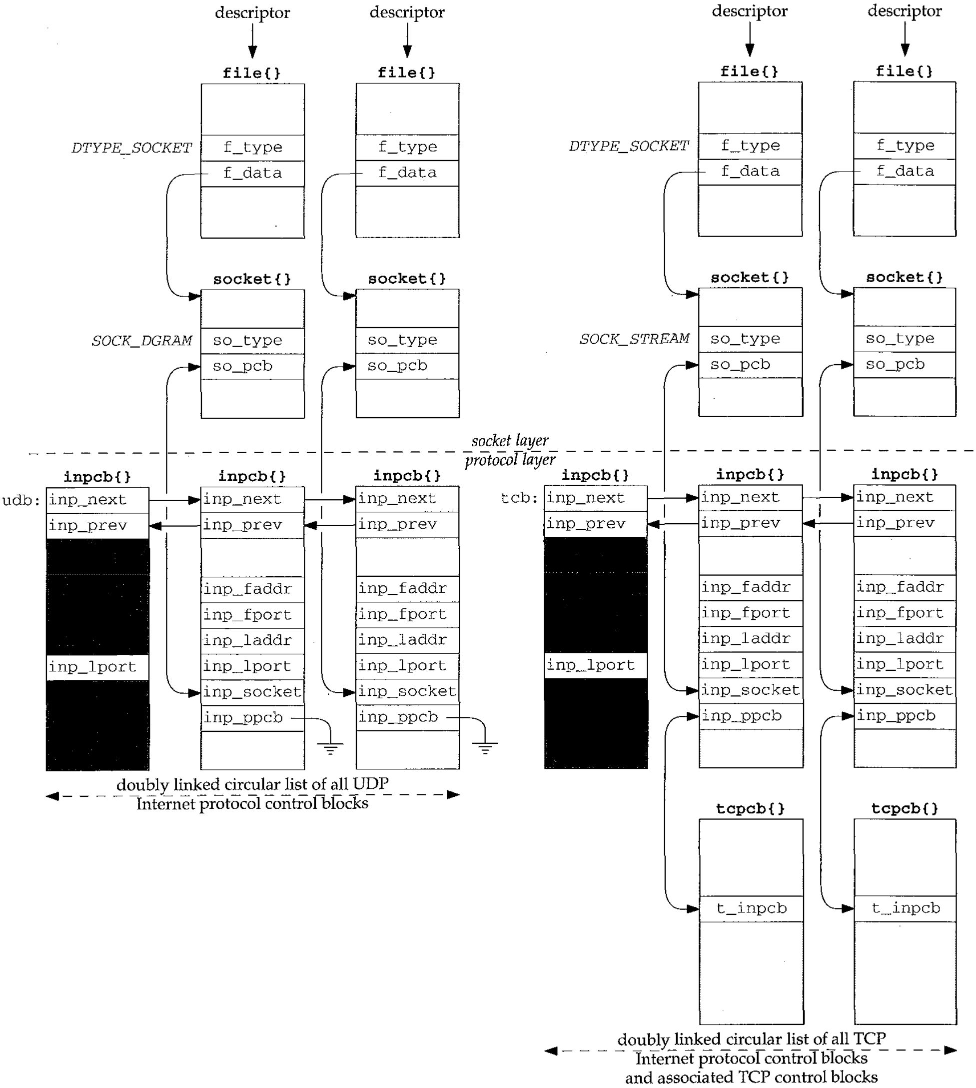

Figure 22.1 Internet protocol control blocks and their relationship to other structures.

- The protocol layer is then called. UDP creates an Internet PCB (an `inpcb` structure) and links it to the `socket` structure: the `so_pcb` member points to the `inpcb` structure and the `inp_socket` member points to the `socket` structure.
- TCP does the same and also creates its own control block (a `tcpcb` structure) and links it to the `inpcb` using the `inp_ppcb` and `t_inpcb` pointers. In the

two UDP `inpcb`s the `inp_ppcb` member is a null pointer, since UDP does not maintain its own control block.

- The four other members of the `inpcb` structure that we show, `inp_faddr` through `inp_lport`, form the socket pair for this end point: the foreign IP address and port number along with the local IP address and port number.
- Both UDP and TCP maintain a doubly linked list of all their Internet PCBs, using the `inp_next` and `inp_prev` pointers. They allocate a global `inpcb` structure as the head of their list (named `udb` and `tcb`) and only use three members in the structure: the next and previous pointers, and the local port number. This latter member contains the next ephemeral port number to use for this protocol.

The Internet PCB is a transport layer data structure. It is used by TCP, UDP, and raw IP, but not by IP, ICMP, or IGMP.

We haven't described raw IP yet, but it too uses Internet PCBs. Unlike TCP and UDP, raw IP does not use the port number members in the PCB, and raw IP uses only two of the functions that we describe in this chapter: `in_pcballoc` to allocate a PCB, and `in_pcbdetach` to release a PCB. We return to raw IP in Chapter 32.

22.2 Code Introduction

All the PCB functions are in a single C file and a single header contains the definitions, as shown in Figure 22.2.

File	Description
`netinet/in_pcb.h`	`inpcb` structure definition
`netinet/in_pcb.c`	PCB functions

Figure 22.2 Files discussed in this chapter.

Global Variables

One global variable is introduced in this chapter, which is shown in Figure 22.3.

Variable	Datatype	Description
`zeroin_addr`	`struct in_addr`	32-bit IP address of all zero bits

Figure 22.3 Global variable introduced in this chapter.

Statistics

Internet PCBs and TCP PCBs are both allocated by the kernel's `malloc` function with a type of `M_PCB`. This is just one of the approximately 60 different types of memory allocated by the kernel. Mbufs, for example, are allocated with a type of `M_BUF`, and `socket` structures are allocated with a type of `M_SOCKET`.

Since the kernel can keep counters of the different types of memory buffers that are allocated, various statistics on the number of PCBs can be maintained. The command `vmstat -m` shows the kernel's memory allocation statistics and the `netstat -m` command shows the mbuf allocation statistics.

22.3 `inpcb` Structure

Figure 22.4 shows the definition of the `inpcb` structure. It is not a big structure, and occupies only 84 bytes.

in_pcb.h

```
42 struct inpcb {
43     struct inpcb *inp_next, *inp_prev;  /* doubly linked list */
44     struct inpcb *inp_head;       /* pointer back to chain of inpcb's for
45                                      this protocol */
46     struct in_addr inp_faddr;     /* foreign IP address */
47     u_short inp_fport;            /* foreign port# */
48     struct in_addr inp_laddr;     /* local IP address */
49     u_short inp_lport;            /* local port# */
50     struct socket *inp_socket;    /* back pointer to socket */
51     caddr_t inp_ppcb;             /* pointer to per-protocol PCB */
52     struct route inp_route;       /* placeholder for routing entry */
53     int     inp_flags;            /* generic IP/datagram flags */
54     struct ip inp_ip;             /* header prototype; should have more */
55     struct mbuf *inp_options;     /* IP options */
56     struct ip_moptions *inp_moptions;   /* IP multicast options */
57 };
```

in_pcb.h

Figure 22.4 `inpcb` structure.

43–45 `inp_next` and `inp_prev` form the doubly linked list of all PCBs for UDP and TCP. Additionally, each PCB has a pointer to the head of the protocol's linked list (`inp_head`). For PCBs on the UDP list, `inp_head` always points to `udb` (Figure 22.1); for PCBs on the TCP list, this pointer always points to `tcb`.

46–49 The next four members, `inp_faddr`, `inp_fport`, `inp_laddr`, and `inp_lport`, contain the socket pair for this IP end point: the foreign IP address and port number and the local IP address and port number. These four values are maintained in the PCB in network byte order, not host byte order.

> The Internet PCB is used by both transport layers, TCP and UDP. While it makes sense to store the local and foreign IP addresses in this structure, the port numbers really don't belong here. The definition of a port number and its size are specified by each transport layer and could differ between different transport layers. This problem was identified in [Partridge 1987], where 8-bit port numbers were used in version 1 of RDP, which required reimplementing several standard kernel routines to use 8-bit port numbers. Version 2 of RDP [Partridge and Hinden 1990] uses 16-bit port numbers. The port numbers really belong in a transport-specific control block, such as TCP's `tcpcb`. A new UDP-specific PCB would then be required. While doable, this would complicate some of the routines we'll examine shortly.

50–51 `inp_socket` is a pointer to the `socket` structure for this PCB and `inp_ppcb` is a pointer to an optional transport-specific control block for this PCB. We saw in Figure 22.1 that the `inp_ppcb` pointer is used with TCP to point to the corresponding `tcpcb`, but is not used by UDP. The link between the `socket` and `inpcb` is two way because sometimes the kernel starts at the socket layer and needs to find the corresponding Internet PCB (e.g., user output), and sometimes the kernel starts at the PCB and needs to locate the corresponding `socket` structure (e.g., processing a received IP datagram).

52 If IP has a route to the foreign address, it is stored in the `inp_route` entry. We'll see that when an ICMP redirect message is received, all Internet PCBs are scanned and all those with a foreign IP address that matches the redirected IP address have their `inp_route` entry marked as invalid. This forces IP to find a new route to the foreign address the next time the PCB is used for output.

53 Various flags are stored in the `inp_flags` member. Figure 22.5 lists the individual flags.

`inp_flags`	Description
INP_HDRINCL	process supplies entire IP header (raw socket only)
INP_RECVOPTS	receive incoming IP options as control information (UDP only, not implemented)
INP_RECVRETOPTS	receive IP options for reply as control information (UDP only, not implemented)
INP_RECVDSTADDR	receive IP destination address as control information (UDP only)
INP_CONTROLOPTS	*INP_RECVOPTS | INP_RECVRETOPTS | INP_RECVDSTADDR*

Figure 22.5 `inp_flags` values.

54 A copy of an IP header is maintained in the PCB but only two members are used, the TOS and TTL. The TOS is initialized to 0 (normal service) and the TTL is initialized by the transport layer. We'll see that TCP and UDP both default the TTL to 64. A process can change these defaults using the `IP_TOS` or `IP_TTL` socket options, and the new value is recorded in the `inpcb.inp_ip` structure. This structure is then used by TCP and UDP as the prototype IP header when sending IP datagrams.

55–56 A process can set the IP options for outgoing datagrams with the `IP_OPTIONS` socket option. A copy of the caller's options are stored in an mbuf by the function `ip_pcbopts` and a pointer to that mbuf is stored in the `inp_options` member. Each time TCP or UDP calls the `ip_output` function, a pointer to these IP options is passed for IP to insert into the outgoing IP datagram. Similarly, a pointer to a copy of the user's IP multicast options is maintained in the `inp_moptions` member.

22.4 `in_pcballoc` and `in_pcbdetach` Functions

An Internet PCB is allocated by TCP, UDP, and raw IP when a socket is created. A `PRU_ATTACH` request is issued by the `socket` system call. In the case of UDP, we'll see in Figure 23.33 that the resulting call is

```
struct socket  *so;
int  error;

error = in_pcballoc(so, &udb);
```

Figure 22.6 shows the `in_pcballoc` function.

in_pcb.c

```
36 int
37 in_pcballoc(so, head)
38 struct socket *so;
39 struct inpcb *head;
40 {
41     struct inpcb *inp;

42     MALLOC(inp, struct inpcb *, sizeof(*inp), M_PCB, M_WAITOK);
43     if (inp == NULL)
44         return (ENOBUFS);
45     bzero((caddr_t) inp, sizeof(*inp));

46     inp->inp_head = head;
47     inp->inp_socket = so;
48     insque(inp, head);
49     so->so_pcb = (caddr_t) inp;
50     return (0);
51 }
```

in_pcb.c

Figure 22.6 `in_pcballoc` function: allocate an Internet PCB.

Allocate PCB and initialize to zero

36–45 `in_pcballoc` calls the kernel's memory allocator using the macro `MALLOC`. Since these PCBs are always allocated as the result of a system call, it is OK to wait for one.

> Net/2 and earlier Berkeley releases stored both Internet PCBs and TCP PCBs in mbufs. Their sizes were 80 and 108 bytes, respectively. With the Net/3 release, the sizes went to 84 and 140 bytes, so TCP control blocks no longer fit into an mbuf. Net/3 uses the kernel's memory allocator instead of mbufs for both types of control blocks.
>
> Careful readers may note that the example in Figure 2.6 shows 17 mbufs allocated for PCBs, yet we just said that Net/3 no longer uses mbufs for Internet PCBs or TCP PCBs. Net/3 does, however, use mbufs for Unix domain PCBs, and that is what this counter refers to. The mbuf statistics output by `netstat` are for all mbufs in the kernel across all protocol suites, not just the Internet protocols.

`bzero` sets the PCB to 0. This is important because the IP addresses and port numbers in the PCB must be initialized to 0.

Link structures together

46–49 The `inp_head` member points to the head of the protocol's PCB list (either `udb` or `tcb`), the `inp_socket` member points to the `socket` structure, the new PCB is added to the protocol's doubly linked list (`insque`), and the `socket` structure points to the PCB. The `insque` function puts the new PCB at the head of the protocol's list.

An Internet PCB is deallocated when a `PRU_DETACH` request is issued. This happens when the socket is closed. The function `in_pcbdetach`, shown in Figure 22.7, is eventually called.

in_pcb.c

```
252 int
253 in_pcbdetach(inp)
254 struct inpcb *inp;
255 {
256     struct socket *so = inp->inp_socket;

257     so->so_pcb = 0;
258     sofree(so);
259     if (inp->inp_options)
260         (void) m_free(inp->inp_options);
261     if (inp->inp_route.ro_rt)
262         rtfree(inp->inp_route.ro_rt);
263     ip_freemoptions(inp->inp_moptions);
264     remque(inp);
265     FREE(inp, M_PCB);
266 }
```

in_pcb.c

Figure 22.7 `in_pcbdetach` function: deallocate an Internet PCB.

252–263 The PCB pointer in the `socket` structure is set to 0 and that structure is released by `sofree`. If an mbuf with IP options was allocated for this PCB, it is released by `m_free`. If a route is held by this PCB, it is released by `rtfree`. Any multicast options are also released by `ip_freemoptions`.

264–265 The PCB is removed from the protocol's doubly linked list by `remque` and the memory used by the PCB is returned to the kernel.

22.5 Binding, Connecting, and Demultiplexing

Before examining the kernel functions that bind sockets, connect sockets, and demultiplex incoming datagrams, we describe the rules imposed by the kernel on these actions.

Binding of Local IP Address and Port Number

Figure 22.8 shows the six different combinations of a local IP address and local port number that a process can specify in a call to `bind`.

The first three lines are typical for servers—they bind a specific port, termed the server's *well-known port*, whose value is known by the client. The last three lines are typical for clients—they don't care what the local port, termed an *ephemeral port*, is, as long as it is unique on the client host.

Most servers and most clients specify the wildcard IP address in the call to `bind`. This is indicated in Figure 22.8 by the notation * on lines 3 and 6.

Local IP address	Local port	Description
unicast or broadcast	nonzero	one local interface, specific port
multicast	nonzero	one local multicast group, specific port
*	nonzero	any local interface or multicast group, specific port
unicast or broadcast	0	one local interface, kernel chooses port
multicast	0	one multicast group, kernel chooses port
*	0	any local interface, kernel chooses port

Figure 22.8 Combination of local IP address and local port number for `bind`.

If a server binds a specific IP address to a socket (i.e., not the wildcard address), then only IP datagrams arriving with that specific IP address as the destination IP address—be it unicast, broadcast, or multicast—are delivered to the process. Naturally, when the process binds a specific unicast or broadcast IP address to a socket, the kernel verifies that the IP address corresponds to a local interface.

It is rare, though possible, for a client to bind a specific IP address (lines 4 and 5 in Figure 22.8). Normally a client binds the wildcard IP address (the final line in Figure 22.8), which lets the kernel choose the outgoing interface based on the route chosen to reach the server.

What we don't show in Figure 22.8 is what happens if the client tries to bind a local port that is already in use with another socket. By default a process cannot bind a port number if that port is already in use. The error `EADDRINUSE` (address already in use) is returned if this occurs. The definition of *in use* is simply whether a PCB exists with that port as its local port. This notion of "in use" is relative to a given protocol: TCP or UDP, since TCP port numbers are independent of UDP port numbers.

Net/3 allows a process to change this default behavior by specifying one of following two socket options:

`SO_REUSEADDR` Allows the process to bind a port number that is already in use, but the IP address being bound (including the wildcard) must not already be bound to that same port.

For example, if an attached interface has the IP address 140.252.1.29 then one socket can be bound to 140.252.1.29, port 5555; another socket can be bound to 127.0.0.1, port 5555; and another socket can be bound to the wildcard IP address, port 5555. The call to `bind` for the second and third cases must be preceded by a call to `setsockopt`, setting the `SO_REUSEADDR` option.

`SO_REUSEPORT` Allows a process to reuse both the IP address and port number, but *each* binding of the IP address and port number, including the first, must specify this socket option. With `SO_REUSEADDR`, the first binding of the port number need not specify the socket option.

For example, if an attached interface has the IP address 140.252.1.29 and a socket is bound to 140.252.1.29, port 6666 specifying the

`SO_REUSEPORT` socket option, then another socket can also specify this same socket option and bind 140.252.1.29, port 6666.

Later in this section we describe what happens in this final example when an IP datagram arrives with a destination address of 140.252.1.29 and a destination port of 6666, since two sockets are bound to that end point.

The `SO_REUSEPORT` option is new with Net/3 and was introduced with the support for multicasting in 4.4BSD. Before this release it was never possible for two sockets to be bound to the same IP address and same port number.

Unfortunately the `SO_REUSEPORT` option was not part of the original Stanford multicast sources and is therefore not widely supported. Other systems that support multicasting, such as Solaris 2.x, let a process specify `SO_REUSEADDR` to specify that it is OK to bind multiple sockets to the same IP address and same port number.

Connecting a UDP Socket

We normally associate the `connect` system call with TCP clients, but it is also possible for a UDP client or a UDP server to call `connect` and specify the foreign IP address and foreign port number for the socket. This restricts the socket to exchanging UDP datagrams with that one particular peer.

There is a side effect when a UDP socket is connected: the local IP address, if not already specified by a call to `bind`, is automatically set by `connect`. It is set to the local interface address chosen by IP routing to reach the specified peer.

Figure 22.9 shows the three different states of a UDP socket along with the pseudocode of the function calls to end up in that state.

Local socket	Foreign socket	Description

localIP`.`*lport*	*foreignIP*`.`*fport*	restricted to one peer:
		`socket(),bind(*,` *lport*`),connect(`*foreignIP, fport*`)`
		`socket(),bind(`*localIP, lport*`),connect(`*foreignIP, fport*`)`

localIP`.`*lport*	`*.*`	restricted to datagrams arriving on one local interface: *localIP* `socket(),bind(`*localIP, lport*`)`
`*.`*lport*	`*.*`	receives all datagrams sent to *lport*: `socket(),bind(*,` *lport*`)`

Figure 22.9 Specification of local and foreign IP addresses and port numbers for UDP sockets.

The first of the three states is called a *connected UDP socket* and the next two states are called *unconnected UDP sockets*. The difference between the two unconnected sockets is that the first has a fully specified local address and the second has a wildcarded local IP address.

Demultiplexing of Received IP Datagrams by TCP

Figure 22.10 shows the state of three Telnet server sockets on the host `sun`. The first two sockets are in the LISTEN state, waiting for incoming connection requests, and the third is connected to a client at port 1500 on the host with an IP address of 140.252.1.11. The first listening socket will handle connection requests that arrive on the 140.252.1.29 interface and the second listening socket will handle all other interfaces (since its local IP address is the wildcard).

Local address	Local port	Foreign address	Foreign port	TCP state
140.252.1.29	23	*	*	LISTEN
*	23	*	*	LISTEN
140.252.1.29	23	140.252.1.11	1500	ESTABLISHED

Figure 22.10 Three TCP sockets with a local port of 23.

We show both of the listening sockets with unspecified foreign IP addresses and port numbers because the sockets API doesn't allow a TCP server to restrict either of these values. A TCP server must `accept` the client's connection and is then told of the client's IP address and port number after the connection establishment is complete (i.e., when TCP's three-way handshake is complete). Only then can the server close the connection if it doesn't like the client's IP address and port number. This isn't a required TCP feature, it is just the way the sockets API has always worked.

When TCP receives a segment with a destination port of 23 it searches through its list of Internet PCBs looking for a match by calling `in_pcblookup`. When we examine this function shortly we'll see that it has a preference for the smallest number of *wildcard matches*. To determine the number of wildcard matches we consider only the local and foreign IP addresses. We do not consider the foreign port number. The local port number must match, or we don't even consider the PCB. The number of wildcard matches can be 0, 1 (local IP address or foreign IP address), or 2 (both local and foreign IP addresses).

For example, assume the incoming segment is from 140.252.1.11, port 1500, destined for 140.252.1.29, port 23. Figure 22.11 shows the number of wildcard matches for the three sockets from Figure 22.10.

Local address	Local port	Foreign address	Foreign port	TCP state	#wildcard matches
140.252.1.29	23	*	*	LISTEN	1
*	23	*	*	LISTEN	2
140.252.1.29	23	140.252.1.11	1500	ESTABLISHED	0

Figure 22.11 Incoming segment from {140.252.1.11, 1500} to {140.252.1.29, 23}.

The first socket matches these four values, but with one wildcard match (the foreign IP address). The second socket also matches the incoming segment, but with two wildcard matches (the local and foreign IP addresses). The third socket is a complete match with no wildcards. Net/3 uses the third socket, the one with the smallest number of wildcard matches.

Continuing this example, assume the incoming segment is from 140.252.1.11, port 1501, destined for 140.252.1.29, port 23. Figure 22.12 shows the number of wildcard matches.

Local address	Local port	Foreign address	Foreign port	TCP state	#wildcard matches
140.252.1.29	23	*	*	LISTEN	1
*	23	*	*	LISTEN	2
140.252.1.29	23	140.252.1.11	1500	ESTABLISHED	

Figure 22.12 Incoming segment from {140.252.1.11, 1501} to {140.252.1.29, 23}.

The first socket matches with one wildcard match; the second socket matches with two wildcard matches; and the third socket doesn't match at all, since the foreign port numbers are unequal. (The foreign port numbers are compared only if the foreign IP address in the PCB is not a wildcard.) The first socket is chosen.

In these two examples we never said what type of TCP segment arrived: we assume that the segment in Figure 22.11 contains data or an acknowledgment for an established connection since it is delivered to an established socket. We also assume that the segment in Figure 22.12 is an incoming connection request (a SYN) since it is delivered to a listening socket. But the demultiplexing code in `in_pcblookup` doesn't care. If the TCP segment is the wrong type for the socket that it is delivered to, we'll see later how TCP handles this. For now the important fact is that the demultiplexing code only compares the source and destination socket pair from the IP datagram against the values in the PCB.

Demultiplexing of Received IP Datagrams by UDP

The delivery of UDP datagrams is more complicated than the TCP example we just examined, since UDP datagrams can be sent to a broadcast or multicast address. Since Net/3 (and most systems with multicast support) allow multiple sockets to have identical local IP addresses and ports, how are multiple recipients handled? The Net/3 rules are:

1. An incoming UDP datagram destined for either a broadcast IP address or a multicast IP address is delivered to *all* matching sockets. There is no concept of a "best" match here (i.e., the one with the smallest number of wildcard matches).
2. An incoming UDP datagram destined for a unicast IP address is delivered only to *one* matching socket, the one with the smallest number of wildcard matches. If there are multiple sockets with the same "smallest" number of wildcard matches, which socket receives the incoming datagram is implementation-dependent.

Figure 22.13 shows four UDP sockets that we'll use for some examples. Having four UDP sockets with the same local port number requires using either `SO_REUSEADDR` or `SO_REUSEPORT`. The first two sockets have been connected to a foreign IP address and port number, and the last two are unconnected.

Local address	Local port	Foreign address	Foreign port	Comment
140.252.1.29	577	140.252.1.11	1500	connected, local IP = unicast
140.252.13.63	577	140.252.13.35	1500	connected, local IP = broadcast
140.252.13.63	577	*	*	unconnected, local IP = broadcast
*	577	*	*	unconnected, local IP = wildcard

Figure 22.13 Four UDP sockets with a local port of 577.

Consider an incoming UDP datagram destined for 140.252.13.63 (the broadcast address on the 140.252.13 subnet), port 577, from 140.252.13.34, port 1500. Figure 22.14 shows that it is delivered to the third and fourth sockets.

Local address	Local port	Foreign address	Foreign port	Delivered?
140.252.1.29	577	140.252.1.11	1500	no, local and foreign IP mismatch
140.252.13.63	577	140.252.13.35	1500	no, foreign IP mismatch
140.252.13.63	577	*	*	yes
*	577	*	*	yes

Figure 22.14 Received datagram from {140.252.13.34, 1500} to {140.252.13.63, 577}.

The broadcast datagram is not delivered to the first socket because the local IP address doesn't match the destination IP address and the foreign IP address doesn't match the source IP address. It isn't delivered to the second socket because the foreign IP address doesn't match the source IP address.

As the next example, consider an incoming UDP datagram destined for 140.252.1.29 (a unicast address), port 577, from 140.252.1.11, port 1500. Figure 22.15 shows to which sockets the datagram is delivered.

Local address	Local port	Foreign address	Foreign port	Delivered?
140.252.1.29	577	140.252.1.11	1500	yes, 0 wildcard matches
140.252.13.63	577	140.252.13.35	1500	no, local and foreign IP mismatch
140.252.13.63	577	*	*	no, local IP mismatch
*	577	*	*	no, 2 wildcard matches

Figure 22.15 Received datagram from {140.252.1.11, 1500} to {140.252.1.29, 577}.

The datagram matches the first socket with no wildcard matches and also matches the fourth socket with two wildcard matches. It is delivered to the first socket, the best match.

22.6 `in_pcblookup` Function

The function `in_pcblookup` serves four different purposes.

1. When either TCP or UDP receives an IP datagram, `in_pcblookup` scans the protocol's list of Internet PCBs looking for a matching PCB to receive the

datagram. This is transport layer demultiplexing of a received datagram.

2. When a process executes the `bind` system call, to assign a local IP address and local port number to a socket, `in_pcbbind` is called by the protocol to verify that the requested local address pair is not already in use.
3. When a process executes the `bind` system call, requesting an ephemeral port be assigned to its socket, the kernel picks an ephemeral port and calls `in_pcbbind` to check if the port is in use. If it is in use, the next ephemeral port number is tried, and so on, until an unused port is located.
4. When a process executes the `connect` system call, either explicitly or implicitly, `in_pcbbind` verifies that the requested socket pair is unique. (An implicit call to `connect` happens when a UDP datagram is sent on an unconnected socket. We'll see this scenario in Chapter 23.)

In cases 2, 3, and 4 `in_pcbbind` calls `in_pcblookup`. Two options confuse the logic of the function. First, a process can specify either the `SO_REUSEADDR` or `SO_REUSEPORT` socket option to say that a duplicate local address is OK.

Second, sometimes a wildcard match is OK (e.g., an incoming UDP datagram can match a PCB that has a wildcard for its local IP address, meaning that the socket will accept UDP datagrams that arrive on any local interface), while other times a wildcard match is forbidden (e.g., when connecting to a foreign IP address and port number).

In the original Stanford IP multicast code appears the comment that "The logic of `in_pcblookup` is rather opaque and there is not a single comment, . . ." The adjective *opaque* is an understatement.

The publicly available IP multicast code available for BSD/386, which is derived from the port to 4.4BSD done by Craig Leres, fixed the overloaded semantics of this function by using `in_pcblookup` only for case 1 above. Cases 2 and 4 are handled by a new function named `in_pcbconflict`, and case 3 is handled by a new function named `in_uniqueport`. Dividing the original functionality into separate functions is much clearer, but in the Net/3 release, which we're describing in this text, the logic is still combined into the single function `in_pcblookup`.

Figure 22.16 shows the `in_pcblookup` function.

The function starts at the head of the protocol's PCB list and potentially goes through every PCB on the list. The variable `match` remembers the pointer to the entry with the best match so far, and `matchwild` remembers the number of wildcards in that match. The latter is initialized to 3, which is a value greater than the maximum number of wildcard matches that can be encountered. (Any value greater than 2 would work.) Each time around the loop, the variable `wildcard` starts at 0 and counts the number of wildcard matches for each PCB.

Compare local port number

416–417 The first comparison is the local port number. If the PCB's local port doesn't match the `lport` argument, the PCB is ignored.

in_pcb.c

```
struct inpcb *
in_pcblookup(head, faddr, fport_arg, laddr, lport_arg, flags)
struct inpcb *head;
struct in_addr faddr, laddr;
u_int   fport_arg, lport_arg;
int     flags;
{
    struct inpcb *inp, *match = 0;
    int     matchwild = 3, wildcard;
    u_short fport = fport_arg, lport = lport_arg;

    for (inp = head->inp_next; inp != head; inp = inp->inp_next) {
        if (inp->inp_lport != lport)
            continue;           /* ignore if local ports are unequal */

        wildcard = 0;

        if (inp->inp_laddr.s_addr != INADDR_ANY) {
            if (laddr.s_addr == INADDR_ANY)
                wildcard++;
            else if (inp->inp_laddr.s_addr != laddr.s_addr)
                continue;
        } else {
            if (laddr.s_addr != INADDR_ANY)
                wildcard++;
        }

        if (inp->inp_faddr.s_addr != INADDR_ANY) {
            if (faddr.s_addr == INADDR_ANY)
                wildcard++;
            else if (inp->inp_faddr.s_addr != faddr.s_addr ||
                     inp->inp_fport != fport)
                continue;
        } else {
            if (faddr.s_addr != INADDR_ANY)
                wildcard++;
        }

        if (wildcard && (flags & INPLOOKUP_WILDCARD) == 0)
            continue;           /* wildcard match not allowed */

        if (wildcard < matchwild) {
            match = inp;
            matchwild = wildcard;
            if (matchwild == 0)
                break;          /* exact match, all done */
        }
    }
    return (match);
}
```

in_pcb.c

Figure 22.16 in_pcblookup function: search all the PCBs for a match.

Compare local address

419–427 `in_pcblookup` compares the local address in the PCB with the `laddr` argument. If one is a wildcard and the other is not a wildcard, the `wildcard` counter is incremented. If both are not wildcards, then they must be the same, or this PCB is ignored. If both are wildcards, nothing changes: they can't be compared and the `wildcard` counter isn't incremented. Figure 22.17 summarizes the four different conditions.

PCB local IP	`laddr` argument	Description
not *	*	`wildcard++`
not *	not *	compare IP addresses, skip PCB if not equal
*	*	can't compare
*	not *	`wildcard++`

Figure 22.17 Four scenarios for the local IP address comparison done by `in_pcblookup`.

Compare foreign address and foreign port number

428–437 These lines perform the same test that we just described, but using the foreign addresses instead of the local addresses. Also, if both foreign addresses are not wildcards then not only must the two IP addresses be equal, but the two foreign ports must also be equal. Figure 22.18 summarizes the foreign IP comparisons.

PCB foreign IP	`faddr` argument	Description
not *	*	`wildcard++`
not *	not *	compare IP addresses and ports, skip PCB if not equal
*	*	can't compare
*	not *	`wildcard++`

Figure 22.18 Four scenarios for the foreign IP address comparison done by `in_pcblookup`.

The additional comparison of the foreign port numbers can be performed for the second line of Figure 22.18 because it is not possible to have a PCB with a nonwildcard foreign address and a foreign port number of 0. This restriction is enforced by `connect`, which we'll see shortly requires a nonwildcard foreign IP address and a nonzero foreign port. It is possible, however, and common, to have a wildcard local address with a nonzero local port. We saw this in Figures 22.10 and 22.13.

Check if wildcard match allowed

438–439 The `flags` argument can be set to `INPLOOKUP_WILDCARD`, which means a match containing wildcards is OK. If a match is found containing wildcards (`wildcard` is nonzero) and this flag was not specified by the caller, this PCB is ignored. When TCP and UDP call this function to demultiplex an incoming datagram, `INPLOOKUP_WILDCARD` is always set, since a wildcard match is OK. (Recall our examples using Figures 22.10 and 22.13.) But when this function is called as part of the `connect` system call, in order to verify that a socket pair is not already in use, the `flags` argument is set to 0.

Remember best match, return if exact match found

440–447 These statements remember the best match found so far. Again, the best match is considered the one with the fewest number of wildcard matches. If a match is found with one or two wildcards, that match is remembered and the loop continues. But if an exact match is found (`wildcard` is 0), the loop terminates, and a pointer to the PCB with that exact match is returned.

Example—Demultiplexing of Received TCP Segment

Figure 22.19 is from the TCP example we discussed with Figure 22.11. Assume `in_pcblookup` is demultiplexing a received datagram from 140.252.1.11, port 1500, destined for 140.252.1.29, port 23. Also assume that the order of the PCBs is the order of the rows in the figure. `laddr` is the destination IP address, `lport` is the destination TCP port, `faddr` is the source IP address, and `fport` is the source TCP port.

PCB values				`wildcard`
Local address	Local port	Foreign address	Foreign port	
140.252.1.29	23	*	*	1
*	23	*	*	2
140.252.1.29	23	140.252.1.11	1500	0

Figure 22.19 `laddr` = 140.252.1.29, `lport` = 23, `faddr` = 140.252.1.11, `fport` = 1500.

When the first row is compared to the incoming segment, `wildcard` is 1 (the foreign IP address), `flags` is set to `INPLOOKUP_WILDCARD`, so `match` is set to point to this PCB and `matchwild` is set to 1. The loop continues since an exact match has not been found yet. The next time around the loop, `wildcard` is 2 (the local and foreign IP addresses) and since this is greater than `matchwild`, the entry is not remembered, and the loop continues. The next time around the loop, `wildcard` is 0, which is less than `matchwild` (1), so this entry is remembered in `match`. The loop also terminates since an exact match has been found and the pointer to this PCB is returned to the caller.

If `in_pcblookup` were used by TCP and UDP only to demultiplex incoming datagrams, it could be simplified. First, there's no need to check whether the `faddr` or `laddr` arguments are wildcards, since these are the source and destination IP addresses from the received datagram. Also the `flags` argument could be removed, along with its corresponding test, since wildcard matches are always OK.

This section has covered the mechanics of the `in_pcblookup` function. We'll return to this function and discuss its meaning after seeing how it is called from the `in_pcbbind` and `in_pcbconnect` functions.

22.7 `in_pcbbind` Function

The next function, `in_pcbbind`, binds a local address and port number to a socket. It is called from five functions:

1. from `bind` for a TCP socket (normally to bind a server's well-known port);
2. from `bind` for a UDP socket (either to bind a server's well-known port or to bind an ephemeral port to a client's socket);
3. from `connect` for a TCP socket, if the socket has not yet been bound to a nonzero port (this is typical for TCP clients);
4. from `listen` for a TCP socket, if the socket has not yet been bound to a nonzero port (this is rare, since `listen` is called by a TCP server, which normally binds a well-known port, not an ephemeral port); and
5. from `in_pcbconnect` (Section 22.8), if the local IP address and local port number have not been set (typical for a call to `connect` for a UDP socket or for each call to `sendto` for an unconnected UDP socket).

In cases 3, 4, and 5, an ephemeral port number is bound to the socket and the local IP address is not changed (in case it is already set).

We call cases 1 and 2 *explicit binds* and cases 3, 4, and 5 *implicit binds*. We also note that although it is normal in case 2 for a server to bind a well-known port, servers invoked using remote procedure calls (RPC) often bind ephemeral ports and then register their ephemeral port with another program that maintains a mapping between the server's RPC program number and its ephemeral port (e.g., the Sun port mapper described in Section 29.4 of Volume 1).

We'll show the `in_pcbbind` function in three sections. Figure 22.20 is the first section.

in_pcb.c

```
int
in_pcbbind(inp, nam)
struct inpcb *inp;
struct mbuf *nam;
{
    struct socket *so = inp->inp_socket;
    struct inpcb *head = inp->inp_head;
    struct sockaddr_in *sin;
    struct proc *p = curproc;    /* XXX */
    u_short lport = 0;
    int     wild = 0, reuseport = (so->so_options & SO_REUSEPORT);
    int     error;

    if (in_ifaddr == 0)
        return (EADDRNOTAVAIL);
    if (inp->inp_lport || inp->inp_laddr.s_addr != INADDR_ANY)
        return (EINVAL);

    if ((so->so_options & (SO_REUSEADDR | SO_REUSEPORT)) == 0 &&
        ((so->so_proto->pr_flags & PR_CONNREQUIRED) == 0 ||
         (so->so_options & SO_ACCEPTCONN) == 0))
        wild = INPLOOKUP_WILDCARD;
```

in_pcb.c

Figure 22.20 `in_pcbbind` function: bind a local address and port number.

64–67 The first two tests verify that at least one interface has been assigned an IP address and that the socket is not already bound. You can't bind a socket twice.

68–71 This `if` statement is confusing. The net result sets the variable `wild` to `INPLOOKUP_WILDCARD` if neither `SO_REUSEADDR` or `SO_REUSEPORT` are set.

The second test is true for UDP sockets since `PR_CONNREQUIRED` is false for connectionless sockets and true for connection-oriented sockets.

The third test is where the confusion lies [Torek 1992]. The socket flag `SO_ACCEPTCONN` is set only by the `listen` system call (Section 15.9), which is valid only for a connection-oriented server. In the normal scenario, a TCP server calls `socket`, `bind`, and then `listen`. Therefore, when `in_pcbbind` is called by `bind`, this socket flag is cleared. Even if the process calls `socket` and then `listen`, without calling `bind`, TCP's `PRU_LISTEN` request calls `in_pcbbind` to assign an ephemeral port to the socket *before* the socket layer sets the `SO_ACCEPTCONN` flag. This means the third test in the `if` statement, testing whether `SO_ACCEPTCONN` is not set, is always true. The `if` statement is therefore equivalent to

```
if ((so->so_options & (SO_REUSEADDR|SO_REUSEPORT)) == 0 &&
    ((so->so_proto->pr_flags & PR_CONNREQUIRED) == 0 || 1))
        wild = INPLOOKUP_WILDCARD;
```

Since anything logically ORed with 1 is always true, this is equivalent to

```
if ((so->so_options & (SO_REUSEADDR|SO_REUSEPORT)) == 0)
        wild = INPLOOKUP_WILDCARD;
```

which is simpler to understand: if either of the `REUSE` socket options is set, `wild` is left as 0. If neither of the `REUSE` socket options are set, `wild` is set to `INPLOOKUP_WILDCARD`. In other words, when `in_pcblookup` is called later in the function, a wildcard match is allowed only if *neither* of the `REUSE` socket options are on.

The next section of the `in_pcbbind`, shown in Figure 22.22, function processes the optional `nam` argument.

72–75 The `nam` argument is a nonnull pointer only when the process calls `bind` explicitly. For an implicit bind (a side effect of `connect`, `listen`, or `in_pcbconnect`, cases 3, 4, and 5 from the beginning of this section), `nam` is a null pointer. When the argument is specified, it is an mbuf containing a `sockaddr_in` structure. Figure 22.21 shows the four cases for the nonnull `nam` argument.

nam argument:		PCB member gets set to:		Comment
localIP	*lport*	inp_laddr	inp_lport	
not *	0	*localIP*	ephemeral port	*localIP* must be local interface
not *	nonzero	*localIP*	*lport*	subject to in_pcblookup
*	0	*	ephemeral port	
*	nonzero	*	*lport*	subject to in_pcblookup

Figure 22.21 Four cases for `nam` argument to `in_pcbbind`.

76–83 The test for the correct address family is commented out, yet the identical test in the `in_pcbconnect` function (Figure 22.25) is performed. We expect either both to be in or both to be out.

in_pcb.c

```
 72     if (nam) {
 73         sin = mtod(nam, struct sockaddr_in *);
 74         if (nam->m_len != sizeof(*sin))
 75             return (EINVAL);
 76 #ifdef notdef
 77         /*
 78          * We should check the family, but old programs
 79          * incorrectly fail to initialize it.
 80          */
 81         if (sin->sin_family != AF_INET)
 82             return (EAFNOSUPPORT);
 83 #endif
 84         lport = sin->sin_port;  /* might be 0 */
 85         if (IN_MULTICAST(ntohl(sin->sin_addr.s_addr))) {
 86             /*
 87              * Treat SO_REUSEADDR as SO_REUSEPORT for multicast;
 88              * allow complete duplication of binding if
 89              * SO_REUSEPORT is set, or if SO_REUSEADDR is set
 90              * and a multicast address is bound on both
 91              * new and duplicated sockets.
 92              */
 93             if (so->so_options & SO_REUSEADDR)
 94                 reuseport = SO_REUSEADDR | SO_REUSEPORT;
 95         } else if (sin->sin_addr.s_addr != INADDR_ANY) {
 96             sin->sin_port = 0;  /* yech... */
 97             if (ifa_ifwithaddr((struct sockaddr *) sin) == 0)
 98                 return (EADDRNOTAVAIL);
 99         }
100         if (lport) {
101             struct inpcb *t;

102             /* GROSS */
103             if (ntohs(lport) < IPPORT_RESERVED &&
104                 (error = suser(p->p_ucred, &p->p_acflag)))
105                 return (error);
106             t = in_pcblookup(head, zeroin_addr, 0,
107                              sin->sin_addr, lport, wild);
108             if (t && (reuseport & t->inp_socket->so_options) == 0)
109                 return (EADDRINUSE);
110         }
111         inp->inp_laddr = sin->sin_addr;     /* might be wildcard */
112     }
```

in_pcb.c

Figure 22.22 in_pcbbind function: process optional nam argument.

85–94 Net/3 tests whether the IP address being bound is a multicast group. If so, the SO_REUSEADDR option is considered identical to SO_REUSEPORT.

95–99 Otherwise, if the local address being bound by the caller is not the wildcard, ifa_ifwithaddr verifies that the address corresponds to a local interface.

> The comment "yech" is probably because the port number in the socket address structure must be 0 because ifa_ifwithaddr does a binary comparison of the entire structure, not just a comparison of the IP addresses.

> This is one of the few instances where the process *must* zero the socket address structure before issuing the system call. If `bind` is called and the final 8 bytes of the socket address structure (`sin_zero[8]`) are nonzero, `ifa_ifwithaddr` will not find the requested interface, and `in_pcbbind` will return an error.

100–105 The next `if` statement is executed when the caller is binding a nonzero port, that is, the process wants to bind one particular port number (the second and fourth scenarios from Figure 22.21). If the requested port is less than 1024 (`IPPORT_RESERVED`) the process must have superuser privilege. This is not part of the Internet protocols, but a Berkeley convention. A port number less than 1024 is called a *reserved port* and is used, for example, by the `rcmd` function [Stevens 1990], which in turn is used by the `rlogin` and `rsh` client programs as part of their authentication with their servers.

106–109 The function `in_pcblookup` (Figure 22.16) is then called to check whether a PCB already exists with the same local IP address and local port number. The second argument is the wildcard IP address (the foreign IP address) and the third argument is a port number of 0 (the foreign port). The wildcard value for the second argument causes `in_pcblookup` to ignore the foreign IP address and foreign port in the PCB—only the local IP address and local port are compared to `sin->sin_addr` and `lport`, respectively. We mentioned earlier that `wild` is set to `INPLOOKUP_WILDCARD` only if neither of the `REUSE` socket options are set.

111 The caller's value for the local IP address is stored in the PCB. This can be the wildcard address, if that's the value specified by the caller. In this case the local IP address is chosen by the kernel, but not until the socket is connected at some later time. This is because the local IP address is determined by IP routing, based on foreign IP address.

The final section of `in_pcbbind` handles the assignment of an ephemeral port when the caller explicitly binds a port of 0, or when the `nam` argument is a null pointer (an implicit bind).

in_pcb.c

```
113     if (lport == 0)
114         do {
115             if (head->inp_lport++ < IPPORT_RESERVED ||
116                 head->inp_lport > IPPORT_USERRESERVED)
117                 head->inp_lport = IPPORT_RESERVED;
118             lport = htons(head->inp_lport);
119         } while (in_pcblookup(head,
120                               zeroin_addr, 0, inp->inp_laddr, lport, wild));
121     inp->inp_lport = lport;
122     return (0);
123 }
```

in_pcb.c

Figure 22.23 `in_pcbbind` function: choose an ephemeral port.

113–122 The next ephemeral port number to use for this protocol (TCP or UDP) is maintained in the `head` of the protocol's PCB list: `tcb` or `udb`. Other than the `inp_next` and `inp_back` pointers in the protocol's `head` PCB, the only other element of the `inpcb` structure that is used is the local port number. Confusingly, this local port number is maintained in host byte order in the `head` PCB, but in network byte order in all the other PCBs on the list! The ephemeral port numbers start at 1024

(IPPORT_RESERVED) and get incremented by 1 until port 5000 is used (IPPORT_USERRESERVED), then cycle back to 1024. The loop is executed until in_pcbbind does not find a match.

SO_REUSEADDR Examples

Let's look at some common examples to see the interaction of in_pcbbind with in_pcblookup and the two REUSE socket options.

1. A TCP or UDP server normally starts by calling socket and bind. Assume a TCP server that calls bind, specifying the wildcard IP address and its nonzero well-known port, say 23 (the Telnet server). Also assume that the server is not already running and that the process does not set the SO_REUSEADDR socket option.

   in_pcbbind calls in_pcblookup with INPLOOKUP_WILDCARD as the final argument. The loop in in_pcblookup won't find a matching PCB, assuming no other process is using the server's well-known TCP port, causing a null pointer to be returned. This is OK and in_pcbbind returns 0.

2. Assume the same scenario as above, but with the server already running when someone tries to start the server a second time.

   When in_pcblookup is called it finds the PCB with a local socket of {*, 23}. Since the wildcard counter is 0, in_pcblookup returns the pointer to this entry. Since reuseport is 0, in_pcbbind returns EADDRINUSE.

3. Assume the same scenario as the previous example, but when the attempt is made to start the server a second time, the SO_REUSEADDR socket option is specified.

   Since this socket option is specified, in_pcbbind calls in_pcblookup with a final argument of 0. But the PCB with a local socket of {*, 23} is still matched and returned because wildcard is 0, since in_pcblookup cannot compare the two wildcard addresses (Figure 22.17). in_pcbbind again returns EADDRINUSE, preventing us from starting two instances of the server with identical local sockets, regardless of whether we specify SO_REUSEADDR or not.

4. Assume that a Telnet server is already running with a local socket of {*, 23} and we try to start another with a local socket of {140.252.13.35, 23}.

   Assuming SO_REUSEADDR is not specified, in_pcblookup is called with a final argument of INPLOOKUP_WILDCARD. When it compares the PCB containing *.23, the counter wildcard is set to 1. Since a wildcard match is allowed, this match is remembered as the best match and a pointer to it is returned after all the TCP PCBs are scanned. in_pcbbind returns EADDRINUSE.

5. This example is the same as the previous one, but we specify the SO_REUSEADDR socket option for the second server that tries to bind the local socket {140.252.13.35, 23}.

   The final argument to in_pcblookup is now 0, since the socket option is specified. When the PCB with the local socket {*, 23} is compared, the wildcard counter is 1, but since the final `flags` argument is 0, this entry is skipped and is not remembered as a match. After comparing all the TCP PCBs, the function returns a null pointer and `in_pcbbind` returns 0.

6. Assume the first Telnet server is started with a local socket of {140.252.13.35, 23} when we try to start a second server with a local socket of {*, 23}. This is the same as the previous example, except we're starting the servers in reverse order this time.

   The first server is started without a problem, assuming no other socket has already bound port 23. When we start the second server, the final argument to `in_pcblookup` is `INPLOOKUP_WILDCARD`, assuming the `SO_REUSEADDR` socket option is not specified. When the PCB with the local socket of {140.252.13.35, 23} is compared, the `wildcard` counter is set to 1 and this entry is remembered. After all the TCP PCBs are compared, the pointer to this entry is returned, causing `in_pcbbind` to return `EADDRINUSE`.

7. What if we start two instances of a server, both with a nonwildcard local IP address? Assume we start the first Telnet server with a local socket of {140.252.13.35, 23} and then try to start a second with a local socket of {127.0.0.1, 23}, without specifying `SO_REUSEADDR`.

   When the second server calls `in_pcbbind`, it calls `in_pcblookup` with a final argument of `INPLOOKUP_WILDCARD`. When the PCB with the local socket of {140.252.13.35, 23} is compared, it is skipped because the local IP addresses are not equal. `in_pcblookup` returns a null pointer, and `in_pcbbind` returns 0.

   From this example we see that the `SO_REUSEADDR` socket option has no effect on nonwildcard IP addresses. Indeed the test on the flags value `INPLOOKUP_WILDCARD` in `in_pcblookup` is made only when `wildcard` is greater than 0, that is, when either the PCB entry has a wildcard IP address or the IP address being bound is the wildcard.

8. As a final example, assume we try to start two instances of the same server, both with the same nonwildcard local IP address, say 127.0.0.1.

   When the second server is started, `in_pcblookup` always returns a pointer to the matching PCB with the same local socket. This happens regardless of the `SO_REUSEADDR` socket option, because the `wildcard` counter is always 0 for this comparison. Since `in_pcblookup` returns a nonnull pointer, `in_pcbbind` returns `EADDRINUSE`.

From these examples we can state the rules about the binding of local IP addresses and the `SO_REUSEADDR` socket option. These rules are shown in Figure 22.24. We assume that *localIP1* and *localIP2* are two different unicast or broadcast IP addresses valid on the local host, and that *localmcastIP* is a multicast group. We also assume that the process is trying to bind the same nonzero port number that is already bound to the existing PCB.

We need to differentiate between a unicast or broadcast address and a multicast address, because we saw that `in_pcbbind` considers `SO_REUSEADDR` to be the same as `SO_REUSEPORT` for a multicast address.

Existing PCB	Try to bind	SO_REUSEADDR		Description
		off	on	
localIP1	*localIP1*	error	error	one server per IP address and port
localIP1	*localIP2*	OK	OK	one server for each local interface
localIP1	*	error	OK	one server for one interface, other server for remaining interfaces
*	*localIP1*	error	OK	one server for one interface, other server for remaining interfaces
*	*	error	error	can't duplicate local sockets (same as first example)
localmcastIP	*localmcastIP*	error	OK	multiple multicast recipients

Figure 22.24 Effect of `SO_REUSEADDR` socket option on binding of local IP address.

`SO_REUSEPORT` Socket Option

The handling of `SO_REUSEPORT` in Net/3 changes the logic of `in_pcbbind` to allow duplicate local sockets as long as both sockets specify `SO_REUSEPORT`. In other words, all the servers must agree to share the same local port.

22.8 `in_pcbconnect` Function

The function `in_pcbconnect` specifies the foreign IP address and foreign port number for a socket. It is called from four functions:

1. from `connect` for a TCP socket (required for a TCP client);
2. from `connect` for a UDP socket (optional for a UDP client, rare for a UDP server);
3. from `sendto` when a datagram is output on an unconnected UDP socket (common); and
4. from `tcp_input` when a connection request (a SYN segment) arrives on a TCP socket that is in the LISTEN state (standard for a TCP server).

In all four cases it is common, though not required, for the local IP address and local port be unspecified when `in_pcbconnect` is called. Therefore one function of `in_pcbconnect` is to assign the local values when they are unspecified.

We'll discuss the `in_pcbconnect` function in four sections. Figure 22.25 shows the first section.

in_pcb.c

```
130 int
131 in_pcbconnect(inp, nam)
132 struct inpcb *inp;
133 struct mbuf *nam;
134 {
135     struct in_ifaddr *ia;
136     struct sockaddr_in *ifaddr;
137     struct sockaddr_in *sin = mtod(nam, struct sockaddr_in *);
```

```
138     if (nam->m_len != sizeof(*sin))
139         return (EINVAL);
140     if (sin->sin_family != AF_INET)
141         return (EAFNOSUPPORT);
142     if (sin->sin_port == 0)
143         return (EADDRNOTAVAIL);
144     if (in_ifaddr) {
145         /*
146          * If the destination address is INADDR_ANY,
147          * use the primary local address.
148          * If the supplied address is INADDR_BROADCAST,
149          * and the primary interface supports broadcast,
150          * choose the broadcast address for that interface.
151          */
152 #define satosin(sa)     ((struct sockaddr_in *)(sa))
153 #define sintosa(sin)    ((struct sockaddr *)(sin))
154 #define ifatoia(ifa)    ((struct in_ifaddr *)(ifa))
155         if (sin->sin_addr.s_addr == INADDR_ANY)
156             sin->sin_addr = IA_SIN(in_ifaddr)->sin_addr;
157         else if (sin->sin_addr.s_addr == (u_long) INADDR_BROADCAST &&
158                  (in_ifaddr->ia_ifp->if_flags & IFF_BROADCAST))
159             sin->sin_addr = satosin(&in_ifaddr->ia_broadaddr)->sin_addr;
160     }
```

in_pcb.c

Figure 22.25 `in_pcbconnect` function: verify arguments, check foreign IP address.

Validate argument

130–143 The `nam` argument points to an mbuf containing a `sockaddr_in` structure with the foreign IP address and port number. These lines validate the argument and verify that the caller is not trying to connect to a port number of 0.

Handle connection to 0.0.0.0 and 255.255.255.255 specially

144–160 The test of the global `in_ifaddr` verifies that an IP interface has been configured. If the foreign IP address is 0.0.0.0 (`INADDR_ANY`), then 0.0.0.0 is replaced with the IP address of the primary IP interface. This means the calling process is connecting to a peer on this host. If the foreign IP address is 255.255.255.255 (`INADDR_BROADCAST`) and the primary interface supports broadcasting, then 255.255.255.255 is replaced with the broadcast address of the primary interface. This allows a UDP application to broadcast on the primary interface without having to figure out its IP address—it can simply send datagrams to 255.255.255.255, and the kernel converts this to the appropriate IP address for the interface.

The next section of code, Figure 22.26, handles the case of an unspecified local address. This is the common scenario for TCP and UDP clients, cases 1, 2, and 3 from the list at the beginning of this section.

in_pcb.c

```
    if (inp->inp_laddr.s_addr == INADDR_ANY) {
        struct route *ro;

        ia = (struct in_ifaddr *) 0;
        /*
         * If route is known or can be allocated now,
         * our src addr is taken from the i/f, else punt.
         */
        ro = &inp->inp_route;
        if (ro->ro_rt &&
            (satosin(&ro->ro_dst)->sin_addr.s_addr !=
             sin->sin_addr.s_addr ||
             inp->inp_socket->so_options & SO_DONTROUTE)) {
            RTFREE(ro->ro_rt);
            ro->ro_rt = (struct rtentry *) 0;
        }
        if ((inp->inp_socket->so_options & SO_DONTROUTE) == 0 &&    /* XXX */
            (ro->ro_rt == (struct rtentry *) 0 ||
             ro->ro_rt->rt_ifp == (struct ifnet *) 0)) {
            /* No route yet, so try to acquire one */
            ro->ro_dst.sa_family = AF_INET;
            ro->ro_dst.sa_len = sizeof(struct sockaddr_in);
            ((struct sockaddr_in *) &ro->ro_dst)->sin_addr =
                sin->sin_addr;
            rtalloc(ro);
        }
        /*
         * If we found a route, use the address
         * corresponding to the outgoing interface
         * unless it is the loopback (in case a route
         * to our address on another net goes to loopback).
         */
        if (ro->ro_rt && !(ro->ro_rt->rt_ifp->if_flags & IFF_LOOPBACK))
            ia = ifatoia(ro->ro_rt->rt_ifa);
        if (ia == 0) {
            u_short fport = sin->sin_port;

            sin->sin_port = 0;
            ia = ifatoia(ifa_ifwithdstaddr(sintosa(sin)));
            if (ia == 0)
                ia = ifatoia(ifa_ifwithnet(sintosa(sin)));
            sin->sin_port = fport;
            if (ia == 0)
                ia = in_ifaddr;
            if (ia == 0)
                return (EADDRNOTAVAIL);
        }
```

in_pcb.c

Figure 22.26 in_pcbconnect function: local IP address not yet specified.

Release route if no longer valid

164–175 If a route is held by the PCB but the destination of that route differs from the foreign address being connected to, or the `SO_DONTROUTE` socket option is set, that route is released.

To understand why a PCB may have an associated route, consider case 3 from the list at the beginning of this section: `in_pcbconnect` is called *every time* a UDP datagram is sent on an unconnected socket. Each time a process calls `sendto`, the UDP output function calls `in_pcbconnect`, `ip_output`, and `in_pcbdisconnect`. If all the datagrams sent on the socket go to the same destination IP address, then the first time through `in_pcbconnect` the route is allocated and it can be used from that point on. But since a UDP application can send datagrams to a different IP address with each call to `sendto`, the destination address must be compared to the saved route and the route released when the destination changes. This same test is done in `ip_output`, which seems to be redundant.

The `SO_DONTROUTE` socket option tells the kernel to bypass the normal routing decisions and send the IP datagram to the locally attached interface whose IP network address matches the network portion of the destination address.

Acquire route

176–185 If the `SO_DONTROUTE` socket option is not set, and a route to the destination is not held by the PCB, try to acquire one by calling `rtalloc`.

Determine outgoing interface

186–205 The goal in this section of code is to have `ia` point to an interface address structure (`in_ifaddr`, Section 6.5), which contains the IP address of the interface. If the PCB holds a route that is still valid, or if `rtalloc` found a route, and the route is not to the loopback interface, the corresponding interface is used. Otherwise `ifa_withdstaddr` and `ifa_withnet` are called to check if the foreign IP address is on the other end of a point-to-point link or on an attached network. Both of these functions require that the port number in the socket address structure be 0, so it is saved in `fport` across the calls. If this fails, the primary IP address is used (`in_ifaddr`), and if no interfaces are configured (`in_ifaddr` is zero), an error is returned.

Figure 22.27 shows the next section of `in_pcbconnect`, which handles a destination address that is a multicast address.

206–223 If the destination address is a multicast address and the process has specified the outgoing interface to use for multicast packets (using the `IP_MULTICAST_IF` socket option), then the IP address of that interface is used as the local address. A search is made of all IP interfaces for the one matching the interface that was specified with the socket option. An error is returned if that interface is no longer up.

224–225 The code that started at the beginning of Figure 22.26 to handle the case of a wildcard local address is complete. The pointer to the `sockaddr_in` structure for the local interface `ia` is saved in `ifaddr`.

The final section of `in_pcblookup` is shown in Figure 22.28.

in_pcb.c

```
206         /*
207          * If the destination address is multicast and an outgoing
208          * interface has been set as a multicast option, use the
209          * address of that interface as our source address.
210          */
211         if (IN_MULTICAST(ntohl(sin->sin_addr.s_addr)) &&
212             inp->inp_moptions != NULL) {
213             struct ip_moptions *imo;
214             struct ifnet *ifp;

215             imo = inp->inp_moptions;
216             if (imo->imo_multicast_ifp != NULL) {
217                 ifp = imo->imo_multicast_ifp;
218                 for (ia = in_ifaddr; ia; ia = ia->ia_next)
219                     if (ia->ia_ifp == ifp)
220                         break;
221                 if (ia == 0)
222                     return (EADDRNOTAVAIL);
223             }
224         }
225         ifaddr = (struct sockaddr_in *) &ia->ia_addr;
226     }
```

in_pcb.c

Figure 22.27 in_pcbconnect function: destination address is a multicast address.

in_pcb.c

```
227     if (in_pcblookup(inp->inp_head,
228                      sin->sin_addr,
229                      sin->sin_port,
230                 inp->inp_laddr.s_addr ? inp->inp_laddr : ifaddr->sin_addr,
231                      inp->inp_lport,
232                      0))
233         return (EADDRINUSE);

234     if (inp->inp_laddr.s_addr == INADDR_ANY) {
235         if (inp->inp_lport == 0)
236             (void) in_pcbbind(inp, (struct mbuf *) 0);
237         inp->inp_laddr = ifaddr->sin_addr;
238     }
239     inp->inp_faddr = sin->sin_addr;
240     inp->inp_fport = sin->sin_port;
241     return (0);
242 }
```

in_pcb.c

Figure 22.28 in_pcbconnect function: verify that socket pair is unique.

Verify that socket pair is unique

227–233 in_pcblookup verifies that the socket pair is unique. The foreign address and foreign port are the values specified as arguments to in_pcbconnect. The local address is either the value that was already bound to the socket or the value in ifaddr that was calculated in the code we just described. The local port can be 0, which is typical for a TCP client, and we'll see that later in this section of code an ephemeral port is chosen for the local port.

This test prevents two TCP connections to the same foreign address and foreign port from the same local address and local port. For example, if we establish a TCP connection with the echo server on the host `sun` and then try to establish another connection to the same server from the same local port (8888, specified with the `-b` option), the call to `in_pcblookup` returns a match, causing `connect` to return the error `EADDRINUSE`. (We use the `sock` program from Appendix C of Volume 1.)

```
bsdi $ sock -b 8888 sun echo &          start first one in the background
bsdi $ sock -A -b 8888 sun echo         then try again
connect() error: Address already in use
```

We specify the `-A` option to set the `SO_REUSEADDR` socket option, which lets the `bind` succeed, but the `connect` cannot succeed. This is a contrived example, as we explicitly bound the same local port (8888) to both sockets. In the normal scenario of two different clients from the host `bsdi` to the echo server on the host `sun`, the local port will be 0 when the second client calls `in_pcblookup` from Figure 22.28.

This test also prevents two UDP sockets from being connected to the same foreign address from the same local port. This test does not prevent two UDP sockets from alternately sending datagrams to the same foreign address from the same local port, as long as neither calls `connect`, since a UDP socket is only temporarily connected to a peer for the duration of a `sendto` system call.

Implicit bind and assignment of ephemeral port

234–238 If the local address is still wildcarded for the socket, it is set to the value saved in `ifaddr`. This is an implicit bind: cases 3, 4, and 5 from the beginning of Section 22.7. First a check is made as to whether the local port has been bound yet, and if not, `in_pcbbind` binds an ephemeral port to the socket. The order of the call to `in_pcbbind` and the assignment to `inp_laddr` is important, since `in_pcbbind` fails if the local address is not the wildcard address.

Store foreign address and foreign port in PCB

239–240 The final step of this function sets the foreign IP address and foreign port number in the PCB. We are guaranteed, on successful return from this function, that both socket pairs in the PCB—the local and foreign—are filled in with specific values.

IP Source Address Versus Outgoing Interface Address

There is a subtle difference between the source address in the IP datagram versus the IP address of the interface used to send the datagram.

The PCB member `inp_laddr` is used by TCP and UDP as the source address of the IP datagram. It can be set by the process to the IP address of *any* configured interface by `bind`. (The call to `ifa_ifwithaddr` in `in_pcbbind` verifies the local address desired by the application.) `in_pcbconnect` assigns the local address only if it is a wildcard, and when this happens the local address is based on the outgoing interface (since the destination address is known).

The outgoing interface, however, is also determined by `ip_output` based on the destination IP address. On a multihomed host it is possible for the source address to be a local interface that is not the outgoing interface, when the process explicitly binds a local address that differs from the outgoing interface. This is allowed because Net/3 chooses the weak end system model (Section 8.4).

22.9 in_pcbdisconnect Function

A UDP socket is disconnected by `in_pcbdisconnect`. This removes the foreign association by setting the foreign IP address to all 0s (`INADDR_ANY`) and foreign port number to 0.

This is done after a datagram has been sent on an unconnected UDP socket and when `connect` is called on a connected UDP socket. In the first case the sequence of steps when the process calls `sendto` is: UDP calls `in_pcbconnect` to connect the socket temporarily to the destination, `udp_output` sends the datagram, and then `in_pcbdisconnect` removes the temporary connection.

`in_pcbdisconnect` is not called when a socket is closed since `in_pcbdetach` handles the release of the PCB. A disconnect is required only when the PCB needs to be reused for a different foreign address or port number.

Figure 22.29 shows the function `in_pcbdisconnect`.

in_pcb.c

```
int
in_pcbdisconnect(inp)
struct inpcb *inp;
{

    inp->inp_faddr.s_addr = INADDR_ANY;
    inp->inp_fport = 0;
    if (inp->inp_socket->so_state & SS_NOFDREF)
        in_pcbdetach(inp);
}
```

in_pcb.c

Figure 22.29 `in_pcbdisconnect` function: disconnect from foreign address and port number.

If there is no longer a file table reference for this PCB (`SS_NOFDREF` is set) then `in_pcbdetach` (Figure 22.7) releases the PCB.

22.10 in_setsockaddr and in_setpeeraddr Functions

The `getsockname` system call returns the local protocol address of a socket (e.g., the IP address and port number for an Internet socket) and the `getpeername` system call returns the foreign protocol address. Both system calls end up issuing a `PRU_SOCKADDR` request or a `PRU_PEERADDR` request. The protocol then calls either `in_setsockaddr` or `in_setpeeraddr`. We show the first of these in Figure 22.30.

in_pcb.c

```
int
in_setsockaddr(inp, nam)
struct inpcb *inp;
struct mbuf *nam;
{
    struct sockaddr_in *sin;

    nam->m_len = sizeof(*sin);
    sin = mtod(nam, struct sockaddr_in *);
    bzero((caddr_t) sin, sizeof(*sin));
    sin->sin_family = AF_INET;
    sin->sin_len = sizeof(*sin);
    sin->sin_port = inp->inp_lport;
    sin->sin_addr = inp->inp_laddr;
}
```

in_pcb.c

Figure 22.30 `in_setsockaddr` function: return local address and port number.

The argument `nam` is a pointer to an mbuf that will hold the result: a `sockaddr_in` structure that the system call copies back to the process. The code fills in the socket address structure and copies the IP address and port number from the Internet PCB into the `sin_addr` and `sin_port` members.

Figure 22.31 shows the `in_setpeeraddr` function. It is nearly identical to Figure 22.30, but copies the foreign IP address and port number from the PCB.

in_pcb.c

```
int
in_setpeeraddr(inp, nam)
struct inpcb *inp;
struct mbuf *nam;
{
    struct sockaddr_in *sin;

    nam->m_len = sizeof(*sin);
    sin = mtod(nam, struct sockaddr_in *);
    bzero((caddr_t) sin, sizeof(*sin));
    sin->sin_family = AF_INET;
    sin->sin_len = sizeof(*sin);
    sin->sin_port = inp->inp_fport;
    sin->sin_addr = inp->inp_faddr;
}
```

in_pcb.c

Figure 22.31 `in_setpeeraddr` function: return foreign address and port number.

22.11 `in_pcbnotify`, `in_rtchange`, and `in_losing` Functions

The function `in_pcbnotify` is called when an ICMP error is received, in order to notify the appropriate process of the error. The "appropriate process" is found by searching all the PCBs for one of the protocols (TCP or UDP) and comparing the local and foreign IP addresses and port numbers with the values returned in the ICMP error. For example, when an ICMP source quench error is received in response to a TCP segment that some router discarded, TCP must locate the PCB for the connection that caused the error and slow down the transmission on that connection.

Before showing the function we must review how it is called. Figure 22.32 summarizes the functions called to process an ICMP error. The two shaded ellipses are the functions described in this section.

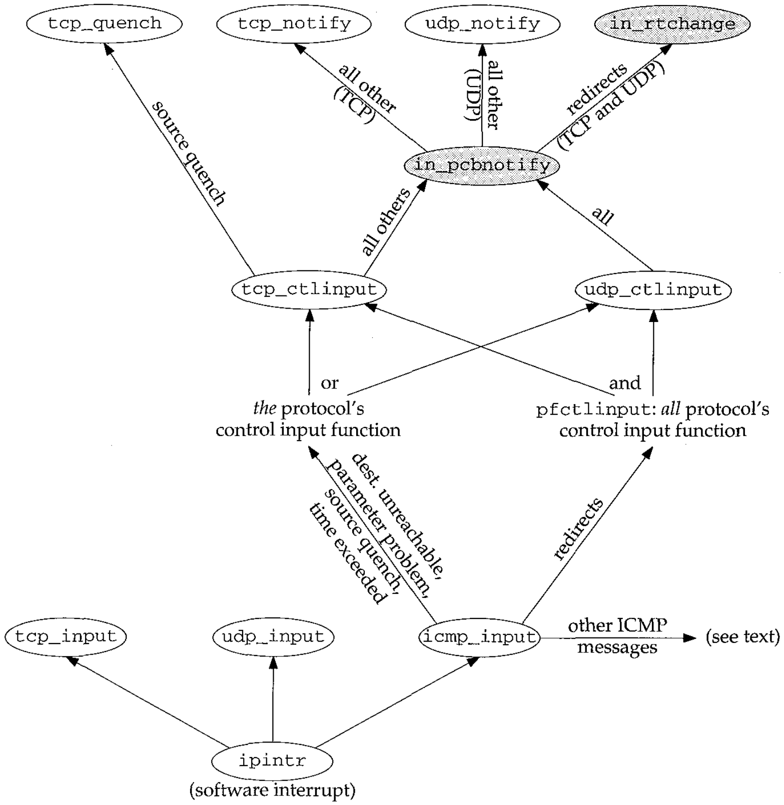

Figure 22.32 Summary of processing of ICMP errors.

When an ICMP message is received, `icmp_input` is called. Five of the ICMP messages are classified as errors (Figures 11.1 and 11.2):

- destination unreachable,
- parameter problem,
- redirect,
- source quench, and
- time exceeded.

Redirects are handled differently from the other four errors. All other ICMP messages (the queries) are handled as described in Chapter 11.

Each protocol defines its control input function, the `pr_ctlinput` entry in the `protosw` structure (Section 7.4). The ones for TCP and UDP are named `tcp_ctlinput` and `udp_ctlinput`, and we'll show their code in later chapters. Since the ICMP error that is received contains the IP header of the datagram that caused the error, the protocol that caused the error (TCP or UDP) is known. Four of the five ICMP errors cause that protocol's control input function to be called. Redirects are handled differently: the function `pfctlinput` is called, and it in turn calls the control input functions for *all* the protocols in the family (Internet). TCP and UDP are the only protocols in the Internet family with control input functions.

Redirects are handled specially because they affect *all* IP datagrams going to that destination, not just the one that caused the redirect. On the other hand, the other four errors need only be processed by the protocol that caused the error.

The final points we need to make about Figure 22.32 are that TCP handles source quenches differently from the other errors, and redirects are handled specially by `in_pcbnotify`: the function `in_rtchange` is called, regardless of the protocol that caused the error.

Figure 22.33 shows the `in_pcbnotify` function. When it is called by TCP, the first argument is the address of `tcb` and the final argument is the address of the function `tcp_notify`. For UDP, these two arguments are the address of `udb` and the address of the function `udp_notify`.

Verify arguments

306–324 The `cmd` argument and the address family of the destination are verified. The foreign address is checked to ensure it is not 0.0.0.0.

Handle redirects specially

325–338 If the error is a redirect it is handled specially. (The error `PRC_HOSTDEAD` is an old error that was generated by the IMPs. Current systems should never see this error—it is a historical artifact.) The foreign port, local port, and local address are all set to 0 so that the `for` loop that follows won't compare them. For a redirect we want that loop to select the PCBs to receive notification based only on the foreign IP address, because that is the IP address for which our host received a redirect. Also, the function that is called for a redirect is `in_rtchange` (Figure 22.34) instead of the `notify` argument specified by the caller.

339 The global array `inetctlerrmap` maps one of the protocol-independent error codes (the `PRC_`*xxx* values from Figure 11.19) into its corresponding Unix `errno` value (the final column in Figure 11.1).

in_pcb.c

```
int
in_pcbnotify(head, dst, fport_arg, laddr, lport_arg, cmd, notify)
struct inpcb *head;
struct sockaddr *dst;
u_int   fport_arg, lport_arg;
struct in_addr laddr;
int     cmd;
void    (*notify) (struct inpcb *, int);
{
    extern u_char inetctlerrmap[];
    struct inpcb *inp, *oinp;
    struct in_addr faddr;
    u_short fport = fport_arg, lport = lport_arg;
    int     errno;

    if ((unsigned) cmd > PRC_NCMDS || dst->sa_family != AF_INET)
        return;
    faddr = ((struct sockaddr_in *) dst)->sin_addr;
    if (faddr.s_addr == INADDR_ANY)
        return;

    /*
     * Redirects go to all references to the destination,
     * and use in_rtchange to invalidate the route cache.
     * Dead host indications: notify all references to the destination.
     * Otherwise, if we have knowledge of the local port and address,
     * deliver only to that socket.
     */
    if (PRC_IS_REDIRECT(cmd) || cmd == PRC_HOSTDEAD) {
        fport = 0;
        lport = 0;
        laddr.s_addr = 0;
        if (cmd != PRC_HOSTDEAD)
            notify = in_rtchange;
    }
    errno = inetctlerrmap[cmd];
    for (inp = head->inp_next; inp != head;) {
        if (inp->inp_faddr.s_addr != faddr.s_addr ||
            inp->inp_socket == 0 ||
            (lport && inp->inp_lport != lport) ||
            (laddr.s_addr && inp->inp_laddr.s_addr != laddr.s_addr) ||
            (fport && inp->inp_fport != fport)) {
            inp = inp->inp_next;
            continue;           /* skip this PCB */
        }
        oinp = inp;
        inp = inp->inp_next;
        if (notify)
            (*notify) (oinp, errno);
    }
}
```

in_pcb.c

Figure 22.33 in_pcbnotify function: pass error notification to processes.

Call notify function for selected PCBs

340–353 This loop selects the PCBs to be notified. Multiple PCBs can be notified—the loop keeps going even after a match is located. The first `if` statement combines five tests, and if any one of the five is true, the PCB is skipped: (1) if the foreign addresses are unequal, (2) if the PCB does not have a corresponding `socket` structure, (3) if the local ports are unequal, (4) if the local addresses are unequal, or (5) if the foreign ports are unequal. The foreign addresses *must* match, while the other three foreign and local elements are compared only if the corresponding argument is nonzero. When a match is found, the `notify` function is called.

`in_rtchange` Function

We saw that `in_pcbnotify` calls the function `in_rtchange` when the ICMP error is a redirect. This function is called for all PCBs with a foreign address that matches the IP address that has been redirected. Figure 22.34 shows the `in_rtchange` function.

in_pcb.c

```
void
in_rtchange(inp, errno)
struct inpcb *inp;
int     errno;
{
    if (inp->inp_route.ro_rt) {
        rtfree(inp->inp_route.ro_rt);
        inp->inp_route.ro_rt = 0;
        /*
         * A new route can be allocated the next time
         * output is attempted.
         */
    }
}
```

in_pcb.c

Figure 22.34 `in_rtchange` *function: invalidate route.*

If the PCB holds a route, that route is released by `rtfree`, and the PCB member is marked as empty. We don't try to update the route at this time, using the new router address returned in the redirect. The new route will be allocated by `ip_output` when this PCB is used next, based on the kernel's routing table, which is updated by the redirect, before `pfctlinput` is called.

Redirects and Raw Sockets

Let's examine the interaction of redirects, raw sockets, and the cached route in the PCB. If we run the Ping program, which uses a raw socket, and an ICMP redirect error is received for the IP address being pinged, Ping continues using the original route, not the redirected route. We can see this as follows.

We ping the host `svr4` on the 140.252.13 network from the host `gemini` on the 140.252.1 network. The default router for `gemini` is `gateway`, but the packets should be sent to the router `netb` instead. Figure 22.35 shows the arrangement.

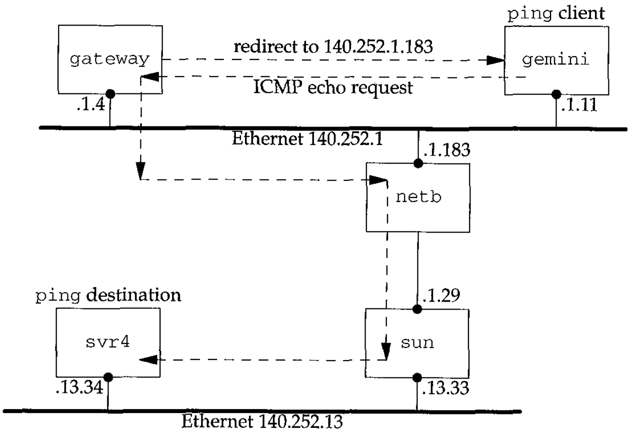

Figure 22.35 Example of ICMP redirect.

We expect `gateway` to send a redirect when it receives the first ICMP echo request.

```
gemini $ ping -sv svr4
PING 140.252.13.34: 56 data bytes
ICMP Host redirect from gateway 140.252.1.4
  to netb (140.252.1.183) for svr4 (140.252.13.34)
64 bytes from svr4 (140.252.13.34): icmp_seq=0. time=572. ms
ICMP Host redirect from gateway 140.252.1.4
  to netb (140.252.1.183) for svr4 (140.252.13.34)
64 bytes from svr4 (140.252.13.34): icmp_seq=1. time=392. ms
```

The `-s` option causes an ICMP echo request to be sent once a second, and the `-v` option prints every received ICMP message (instead of only the ICMP echo replies).

Every ICMP echo request elicits a redirect, but the raw socket used by ping never notices the redirect to change the route that it is using. The route that is first calculated and stored in the PCB, causing the IP datagrams to be sent to the router `gateway` (140.252.1.4), should be updated so that the datagrams are sent to the router `netb` (140.252.1.183) instead. We see that the ICMP redirects are received by the kernel on `gemini`, but they appear to be ignored.

If we terminate the program and start it again, we never see a redirect:

```
gemini $ ping -sv svr4
PING 140.252.13.34: 56 data bytes
64 bytes from svr4 (140.252.13.34): icmp_seq=0. time=388. ms
64 bytes from svr4 (140.252.13.34): icmp_seq=1. time=363. ms
```

The reason for this anomaly is that the raw IP socket code (Chapter 32) does not have a control input function. Only TCP and UDP have a control input function. When the redirect error is received, ICMP updates the kernel's routing table accordingly, and `pfctlinput` is called (Figure 22.32). But since there is no control input function for the raw IP protocol, the cached route in the PCB associated with Ping's raw socket is never released. When we start the Ping program a second time, however, the route that is allocated is based on the kernel's updated routing table, and we never see the redirects.

ICMP Errors and UDP Sockets

One confusing part of the sockets API is that ICMP errors received on a UDP socket are not passed to the application unless the application has issued a `connect` on the socket, restricting the foreign IP address and port number for the socket. We now see where this limitation is enforced by `in_pcbnotify`.

Consider an ICMP port unreachable, probably the most common ICMP error on a UDP socket. The foreign IP address and the foreign port number in the `dst` argument to `in_pcbnotify` are the IP address and port number that caused the ICMP error. But if the process has not issued a `connect` on the socket, the `inp_faddr` and `inp_fport` members of the PCB are both 0, preventing `in_pcbnotify` from ever calling the `notify` function for this socket. The `for` loop in Figure 22.33 will skip every UDP PCB.

This limitation arises for two reasons. First, if the sending process has an unconnected UDP socket, the only nonzero element in the socket pair is the local port. (This assumes the process did not call `bind`.) This is the only value available to `in_pcbnotify` to demultiplex the incoming ICMP error and pass it to the correct process. Although unlikely, there could be multiple processes bound to the same local port, making it ambiguous which process should receive the error. There's also the possibility that the process that sent the datagram that caused the ICMP error has terminated, with another process then starting and using the same local port. This is also unlikely since ephemeral ports are assigned in sequential order from 1024 to 5000 and reused only after cycling around (Figure 22.23).

The second reason for this limitation is because the error notification from the kernel to the process—an `errno` value—is inadequate. Consider a process that calls `sendto` on an unconnected UDP socket three times in a row, sending a UDP datagram to three different destinations, and then waits for the replies with `recvfrom`. If one of the datagrams generates an ICMP port unreachable error, and if the kernel were to return the corresponding error (ECONNREFUSED) to the `recvfrom` that the process issued, the `errno` value doesn't tell the process which of the three datagrams caused the error. The kernel has all the information required in the ICMP error, but the sockets API doesn't provide a way to return this to the process.

Therefore the design decision was made that if a process wants to be notified of these ICMP errors on a UDP socket, that socket must be connected to a single peer. If the error `ECONNREFUSED` is returned on that connected socket, there's no question which peer generated the error.

There is still a remote possibility of an ICMP error being delivered to the wrong process. One process sends the UDP datagram that elicits the ICMP error, but it terminates before the error is received. Another process then starts up before the error is received, binds the same local port, and connects to the same foreign address and foreign port, causing this new process to receive the error. There's no way to prevent this from occurring, given UDP's lack of memory. We'll see that TCP handles this with its TIME_WAIT state.

In our preceding example, one way for the application to get around this limitation is to use three connected UDP sockets instead of one unconnected socket, and call `select` to determine when any one of the three has a received datagram or an error to be read.

Here we have a scenario where the kernel has the information but the API (sockets) is inadequate. With most implementations of Unix System V and the other popular API (TLI), the reverse is true: the TLI function `t_rcvuderr` can return the peer's IP address, port number, and an error value, but most SVR4 streams implementations of TCP/IP don't provide a way for ICMP to pass the error to an unconnected UDP end point.

In an ideal world, `in_pcbnotify` delivers the ICMP error to all UDP sockets that match, even if the only nonwildcard match is the local port. The error returned to the process would include the destination IP address and destination UDP port that caused the error, allowing the process to determine if the error corresponds to a datagram sent by the process.

`in_losing` Function

The final function dealing with PCBs is `in_losing`, shown in Figure 22.36. It is called by TCP when its retransmission timer has expired four or more times in a row for a given connection (Figure 25.26).

in_pcb.c

```
int
in_losing(inp)
struct inpcb *inp;
{
    struct rtentry *rt;
    struct rt_addrinfo info;

    if ((rt = inp->inp_route.ro_rt)) {
        inp->inp_route.ro_rt = 0;
        bzero((caddr_t) & info, sizeof(info));
        info.rti_info[RTAX_DST] =
            (struct sockaddr *) &inp->inp_route.ro_dst;
        info.rti_info[RTAX_GATEWAY] = rt->rt_gateway;
        info.rti_info[RTAX_NETMASK] = rt_mask(rt);
        rt_missmsg(RTM_LOSING, &info, rt->rt_flags, 0);

        if (rt->rt_flags & RTF_DYNAMIC)
            (void) rtrequest(RTM_DELETE, rt_key(rt),
                             rt->rt_gateway, rt_mask(rt), rt->rt_flags,
                             (struct rtentry **) 0);
        else
            /*
             * A new route can be allocated
             * the next time output is attempted.
             */
            rtfree(rt);
    }
}
```

in_pcb.c

Figure 22.36 `in_losing` function: invalidate cached route information.

Generate routing message

361–374 If the PCB holds a route, that route is discarded. An `rt_addrinfo` structure is filled in with information about the cached route that appears to be failing. The function `rt_missmsg` is then called to generate a message from the routing socket of type `RTM_LOSING`, indicating a problem with the route.

Delete or release route

375–384 If the cached route was generated by a redirect (`RTF_DYNAMIC` is set), the route is deleted by calling `rtrequest` with a request of `RTM_DELETE`. Otherwise the cached route is released, causing the next output on the socket to allocate another route to the destination—hopefully a better route.

22.12 Implementation Refinements

Undoubtedly the most time-consuming algorithm we've encountered in this chapter is the linear searching of the PCBs done by `in_pcblookup`. At the beginning of Section 22.6 we noted four instances when this function is called. We can ignore the calls to `bind` and `connect`, as they occur much less frequently than the calls to `in_pcblookup` from TCP and UDP, to demultiplex *every* received IP datagram.

In later chapters we'll see that TCP and UDP both try to help this linear search by maintaining a pointer to the last PCB that the protocol referenced: a one-entry cache. If the local address, local port, foreign address, and foreign port in the cached PCB match the values in the received datagram, the protocol doesn't even call `in_pcblookup`. If the protocol's data fits the packet train model [Jain and Routhier 1986], this simple cache works well. But if the data does not fit this model and, for example, looks like data entry into an on-line transaction processing system, the one-entry cache performs poorly [McKenney and Dove 1992].

One proposal for a better PCB arrangement is to move a PCB to the front of the PCB list when the PCB is referenced. ([McKenney and Dove 1992] attribute this idea to Jon Crowcroft; [Partridge and Pink 1993] attribute it to Gary Delp.) This movement of the PCB is easy to do since it is a doubly linked list and a pointer to the head of the list is the first argument to `in_pcblookup`.

[McKenney and Dove 1992] compare the original Net/1 implementation (no cache), an enhanced one-entry send–receive cache, the move-to-the-front heuristic, and their own algorithm that uses hash chains. They show that maintaining a linear list of PCBs on hash chains provides an order of magnitude improvement over the other algorithms. The only cost for the hash chains is the memory required for the hash chain headers and the computation of the hash function. They also consider adding the move-to-the-front heuristic to their hash-chain algorithm and conclude that it is easier simply to add more hash chains.

Another comparison of the BSD linear search to a hash table search is in [Hutchinson and Peterson 1991]. They show that the time required to demultiplex an incoming UDP datagram is constant as the number of sockets increases for a hash table, but with a linear search the time increases as the number of sockets increases.

22.13 Summary

An Internet PCB is associated with every Internet socket: TCP, UDP, and raw IP. It contains information common to all Internet sockets: local and foreign IP addresses, pointer to a route structure, and so on. All the PCBs for a given protocol are placed on a doubly linked list maintained by that protocol.

In this chapter we've looked at numerous functions that manipulate the PCBs, and three in detail.

1. `in_pcblookup` is called by TCP and UDP to demultiplex every received datagram. It chooses which socket receives the datagram, taking into account wildcard matches.

   This function is also called by `in_pcbbind` to verify that the local address and local process are unique, and by `in_pcbconnect` to verify that the combination of a local address, local process, foreign address, and foreign process are unique.
2. `in_pcbbind` explicitly or implicitly binds a local address and local port to a socket. An explicit bind occurs when the process calls `bind`, and an implicit bind occurs when a TCP client calls `connect` without calling `bind`, or when a UDP process calls `sendto` or `connect` without calling `bind`.
3. `in_pcbconnect` sets the foreign address and foreign process. If the local address has not been set by the process, a route to the foreign address is calculated and the resulting local interface becomes the local address. If the local port has not been set by the process, `in_pcbbind` chooses an ephemeral port for the socket.

Figure 22.37 summarizes the common scenarios for various TCP and UDP applications and the values stored in the PCB for the local address and port and the foreign address and port. We have not yet covered all the actions shown in Figure 22.37 for TCP and UDP processes, but will examine the code in later chapters.

Application	local address: `inp_laddr`	local port: `inp_lport`	foreign address: `inp_faddr`	foreign port: `inp_fport`
TCP client: `connect` (*foreignIP*, *fport*)	`in_pcbconnect` calls `rtalloc` to allocate route to *foreignIP*. Local address is local interface.	`in_pcbconnect` calls `in_pcbbind` to choose ephemeral port.	*foreignIP*	*fport*
TCP client: `bind` (*localIP*, *lport*) `connect` (*foreignIP*, *fport*)	*localIP*	*lport*	*foreignIP*	*fport*
TCP client: `bind` (`*`, *lport*) `connect` (*foreignIP*, *fport*)	`in_pcbconnect` calls `rtalloc` to allocate route to *foreignIP*. Local address is local interface.	*lport*	*foreignIP*	*fport*
TCP client: `bind` (*localIP*, `0`) `connect` (*foreignIP*, *fport*)	*localIP*	`in_pcbbind` chooses ephemeral port.	*foreignIP*	*fport*

TCP server:
`bind` (*localIP*, *lport*)
`listen()`
`accept()`
	localIP	*lport*	Source address from IP header.	Source port from TCP header.

TCP server:
`bind` (`*`, *lport*)
`listen()`
`accept()`
	Destination address from IP header.	*lport*	Source address from IP header.	Source port from TCP header.

UDP client: `sendto` (*foreignIP*, *fport*)	`in_pcbconnect` calls `rtalloc` to allocate route to *foreignIP*. Local address is local interface. Reset to 0.0.0.0 after datagram sent.	`in_pcbconnect` calls `in_pcbbind` to choose ephemeral port. Not changed on subsequent calls to `sendto`.	*foreignIP*. Reset to 0.0.0.0 after datagram sent.	*fport*. Reset to 0 after datagram sent.
UDP client: `connect` (*foreignIP*, *fport*) `write()`	`in_pcbconnect` calls `rtalloc` to allocate route to *foreignIP*. Local address is local interface. Not changed on subsequent calls to `write`.	`in_pcbconnect` calls `in_pcbbind` to choose ephemeral port. Not changed on subsequent calls to `write`.	*foreignIP*	*fport*

Figure 22.37 Summary of `in_pcbbind` and `in_pcbconnect`.

Exercises

22.1 What happens in Figure 22.23 when the process asks for an ephemeral port and every ephemeral port is in use?

22.2 In Figure 22.10 we showed two Telnet servers with listening sockets: one with a specific local IP address and one with the wildcard for its local IP address. Does your system's Telnet daemon allow you to specify the local IP address, and if so, how?

22.3 Assume a socket is bound to the local socket {140.252.1.29, 8888}, and this is the only socket using local port 8888. (1) Go through the steps performed by `in_pcbbind` when another socket is bound to {140.252.13.33, 8888}, without any socket options. (2) Go through the steps performed when another socket is bound to the wildcard IP address, port 8888, without any socket options. (3) Go through the steps performed when another socket is bound to the wildcard IP address, port 8888, with the `SO_REUSEADDR` socket option.

22.4 What is the first ephemeral port number allocated by UDP?

22.5 When a process calls `bind`, which elements in the `sockaddr_in` structure must be filled in?

22.6 What happens if a process tries to `bind` a local broadcast address? What happens if a process tries to `bind` the limited broadcast address (255.255.255.255)?

23

UDP: User Datagram Protocol

23.1 Introduction

The User Datagram Protocol, or UDP, is a simple, datagram-oriented, transport-layer protocol: each output operation by a process produces exactly one UDP datagram, which causes one IP datagram to be sent.

A process accesses UDP by creating a socket of type `SOCK_DGRAM` in the Internet domain. By default the socket is termed *unconnected.* Each time the process sends a datagram it must specify the destination IP address and port number. Each time a datagram is received for the socket, the process can receive the source IP address and port number from the datagram.

We mentioned in Section 22.5 that a UDP socket can also be *connected* to one particular IP address and port number. This causes all datagrams written to the socket to go to that destination, and only datagrams arriving from that IP address and port number are passed to the process.

This chapter examines the implementation of UDP.

23.2 Code Introduction

There are nine UDP functions in a single C file and various UDP definitions in two headers, as shown in Figure 23.1.

Figure 23.2 shows the relationship of the six main UDP functions to other kernel functions. The shaded ellipses are the six functions that we cover in this chapter. We also cover three additional UDP functions that are called by some of these six functions.

File	Description
`netinet/udp.h`	`udphdr` structure definition
`netinet/udp_var.h`	other UDP definitions
`netinet/udp_usrreq.c`	UDP functions

Figure 23.1 Files discussed in this chapter.

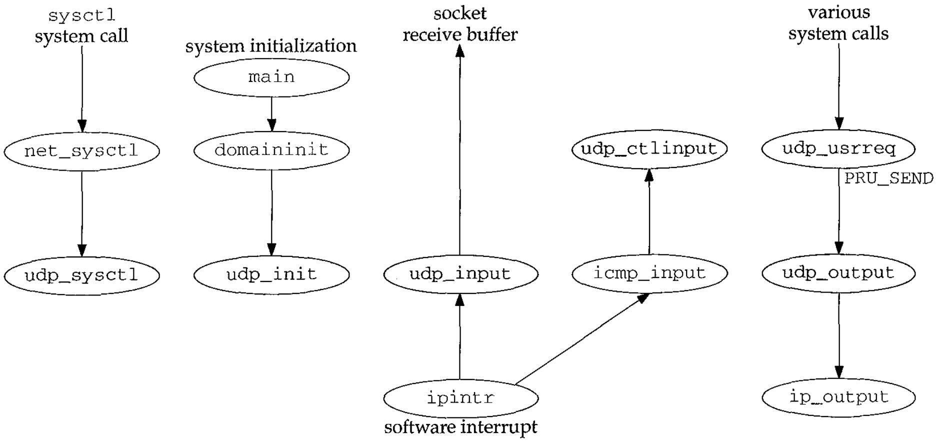

Figure 23.2 Relationship of UDP functions to rest of kernel.

Global Variables

Seven global variables are introduced in this chapter, which are shown in Figure 23.3.

Variable	Datatype	Description
`udb`	`struct inpcb`	head of the UDP PCB list
`udp_last_inpcb`	`struct inpcb *`	pointer to PCB for last received datagram: one-behind cache
`udpcksum`	`int`	flag for calculating and verifying UDP checksum
`udp_in`	`struct sockaddr_in`	holds sender's IP address and port on input
`udpstat`	`struct udpstat`	UDP statistics (Figure 23.4)
`udp_recvspace`	`u_long`	default size of socket receive buffer, 41,600 bytes
`udp_sendspace`	`u_long`	default size of socket send buffer, 9216 bytes

Figure 23.3 Global variables introduced in this chapter.

Statistics

Various UDP statistics are maintained in the global structure `udpstat`, described in Figure 23.4. We'll see where these counters are incremented as we proceed through the code.

`udpstat` member	Description	Used by SNMP
`udps_badlen`	#received datagrams with data length larger than packet	•
`udps_badsum`	#received datagrams with checksum error	•
`udps_fullsock`	#received datagrams not delivered because input socket full	
`udps_hdrops`	#received datagrams with packet shorter than header	•
`udps_ipackets`	total #received datagrams	•
`udps_noport`	#received datagrams with no process on destination port	•
`udps_noportbcast`	#received broadcast/multicast datagrams with no process on dest. port	•
`udps_opackets`	total #output datagrams	•
`udpps_pcbcachemiss`	#received input datagrams missing pcb cache	

Figure 23.4 UDP statistics maintained in the `udpstat` structure.

Figure 23.5 shows some sample output of these statistics, from the `netstat -s` command.

`netstat -s` output	`udpstat` member
`18,575,142 datagrams received`	`udps_ipackets`
`  0 with incomplete header`	`udps_hdrops`
`  18 with bad data length field`	`udps_badlen`
`  58 with bad checksum`	`udps_badsum`
`  84,079 dropped due to no socket`	`udps_noport`
`  446 broadcast/multicast datagrams dropped due to no socket`	`udps_noportbcast`
`  5,356 dropped due to full socket buffers`	`udps_fullsock`
`  18,485,185 delivered`	(see text)
`18,676,277 datagrams output`	`udps_opackets`

Figure 23.5 Sample UDP statistics.

The number of UDP datagrams delivered (the second from last line of output) is the number of datagrams received (`udps_ipackets`) minus the six variables that precede it in Figure 23.5.

SNMP Variables

Figure 23.6 shows the four simple SNMP variables in the UDP group and which counters from the `udpstat` structure implement that variable.

Figure 23.7 shows the UDP listener table, named `udpTable`. The values returned by SNMP for this table are taken from a UDP PCB, not the `udpstat` structure.

SNMP variable	udpstat member	Description
udpInDatagrams	udps_ipackets	#received datagrams delivered to processes
udpInErrors	udps_hdrops + udps_badsum + udps_badlen	#undeliverable UDP datagrams for reasons other than no application at destination port (e.g., UDP checksum error)
udpNoPorts	udps_noport + udps_noportbcast	#received datagrams for which no application process was at the destination port
udpOutDatagrams	udps_opackets	#datagrams sent

Figure 23.6 Simple SNMP variables in `udp` group.

UDP listener table, index = < *udpLocalAddress* >.< *udpLocalPort* >		
SNMP variable	PCB variable	Description
udpLocalAddress	inp_laddr	local IP address for this listener
udpLocalPort	inp_lport	local port number for this listener

Figure 23.7 Variables in UDP listener table: `udpTable`.

23.3 UDP `protosw` Structure

Figure 23.8 lists the protocol switch entry for UDP.

Member	inetsw[1]	Description
pr_type	*SOCK_DGRAM*	UDP provides datagram packet services
pr_domain	*&inetdomain*	UDP is part of the Internet domain
pr_protocol	*IPPROTO_UDP (17)*	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	*udp_input*	receives messages from IP layer
pr_output	*0*	not used by UDP
pr_ctlinput	*udp_ctlinput*	control input function for ICMP errors
pr_ctloutput	*ip_ctloutput*	respond to administrative requests from a process
pr_usrreq	*udp_usrreq*	respond to communication requests from a process
pr_init	*udp_init*	initialization for UDP
pr_fasttimo	*0*	not used by UDP
pr_slowtimo	*0*	not used by UDP
pr_drain	*0*	not used by UDP
pr_sysctl	*udp_sysctl*	for `sysctl`(8) system call

Figure 23.8 The UDP `protosw` structure.

We describe the five functions that begin with `udp_` in this chapter. We also cover a sixth function, `udp_output`, which is not in the protocol switch entry but is called by `udp_usrreq` when a UDP datagram is output.

23.4 UDP Header

The UDP header is defined as a `udphdr` structure. Figure 23.9 shows the C structure and Figure 23.10 shows a picture of the UDP header.

udp.h

```
struct udphdr {
    u_short uh_sport;           /* source port */
    u_short uh_dport;           /* destination port */
    short   uh_ulen;            /* udp length */
    u_short uh_sum;             /* udp checksum */
};
```

udp.h

Figure 23.9 `udphdr` structure.

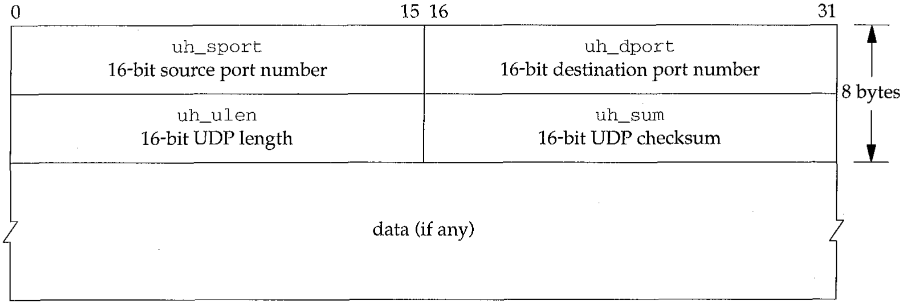

Figure 23.10 UDP header and optional data.

In the source code the UDP header is normally referenced as an IP header immediately followed by a UDP header. This is how `udp_input` processes received IP datagrams, and how `udp_output` builds outgoing IP datagrams. This combined IP/UDP header is a `udpiphdr` structure, shown in Figure 23.11.

udp_var.h

```
struct udpiphdr {
    struct ipovly ui_i;         /* overlaid ip structure */
    struct udphdr ui_u;         /* udp header */
};

#define ui_next     ui_i.ih_next
#define ui_prev     ui_i.ih_prev
#define ui_x1       ui_i.ih_x1
#define ui_pr       ui_i.ih_pr
#define ui_len      ui_i.ih_len
#define ui_src      ui_i.ih_src
#define ui_dst      ui_i.ih_dst
#define ui_sport    ui_u.uh_sport
#define ui_dport    ui_u.uh_dport
#define ui_ulen     ui_u.uh_ulen
#define ui_sum      ui_u.uh_sum
```

udp_var.h

Figure 23.11 `udpiphdr` structure: combined IP/UDP header.

The 20-byte IP header is defined as an `ipovly` structure, shown in Figure 23.12.

ip_var.h

```
struct ipovly {
    caddr_t ih_next, ih_prev;    /* for protocol sequence q's */
    u_char  ih_x1;               /* (unused) */
    u_char  ih_pr;               /* protocol */
    short   ih_len;              /* protocol length */
    struct in_addr ih_src;       /* source internet address */
    struct in_addr ih_dst;       /* destination internet address */
};
```

ip_var.h

Figure 23.12 `ipovly` structure.

Unfortunately this structure is not a real IP header, as shown in Figure 8.8. The size is the same (20 bytes) but the fields are different. We'll return to this discrepancy when we discuss the calculation of the UDP checksum in Section 23.6.

23.5 `udp_init` Function

The `domaininit` function calls UDP's initialization function (`udp_init`, Figure 23.13) at system initialization time.

udp_usrreq.c

```
void
udp_init()
{
    udb.inp_next = udb.inp_prev = &udb;
}
```

udp_usrreq.c

Figure 23.13 `udp_init` function.

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

The remainder of the `udb` PCB is initialized to 0, although the only other field used in this head PCB is `inp_lport`, the next UDP ephemeral port number to allocate. In the solution for Exercise 22.4 we mention that because this local port number is initialized to 0, the first ephemeral port number will be 1024.

23.6 `udp_output` Function

UDP output occurs when the application calls one of the five write functions: `send`, `sendto`, `sendmsg`, `write`, or `writev`. If the socket is connected, any of the five functions can be called, although a destination address cannot be specified with `sendto` or `sendmsg`. If the socket is unconnected, only `sendto` and `sendmsg` can be called, and a destination address must be specified. Figure 23.14 summarizes how these five write functions end up with `udp_output` being called, which in turn calls `ip_output`.

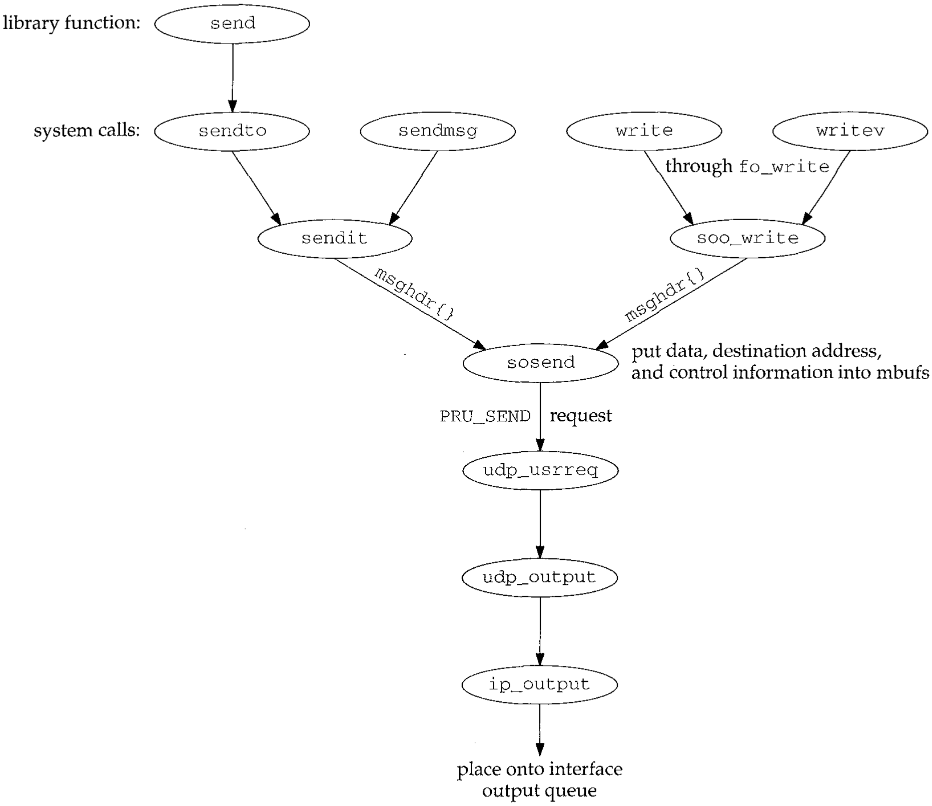

Figure 23.14 How the five write functions end up calling `udp_output`.

All five functions end up calling `sosend`, passing a pointer to a `msghdr` structure as an argument. The data to output is packaged into an mbuf chain and an optional destination address and optional control information are also put into mbufs by `sosend`. A `PRU_SEND` request is issued.

UDP calls the function `udp_output`, which we show the first half of in Figure 23.15. The four arguments are `inp`, a pointer to the socket Internet PCB; `m`, a pointer to the mbuf chain for output; `addr`, an optional pointer to an mbuf with the destination address packaged as a `sockaddr_in` structure; and `control`, an optional pointer to an mbuf with control information from `sendmsg`.

udp_usrreq.c

```
int
udp_output(inp, m, addr, control)
struct inpcb *inp;
struct mbuf *m;
struct mbuf *addr, *control;
{
    struct udpiphdr *ui;
    int     len = m->m_pkthdr.len;
    struct in_addr laddr;
    int     s, error = 0;

    if (control)
        m_freem(control);       /* XXX */

    if (addr) {
        laddr = inp->inp_laddr;
        if (inp->inp_faddr.s_addr != INADDR_ANY) {
            error = EISCONN;
            goto release;
        }
        /*
         * Must block input while temporarily connected.
         */
        s = splnet();
        error = in_pcbconnect(inp, addr);
        if (error) {
            splx(s);
            goto release;
        }
    } else {
        if (inp->inp_faddr.s_addr == INADDR_ANY) {
            error = ENOTCONN;
            goto release;
        }
    }
    /*
     * Calculate data length and get an mbuf for UDP and IP headers.
     */
    M_PREPEND(m, sizeof(struct udpiphdr), M_DONTWAIT);
    if (m == 0) {
        error = ENOBUFS;
        goto release;
    }


                /* remainder of function shown in Figure 23.20 */


  release:
    m_freem(m);
    return (error);
}
```

udp_usrreq.c

Figure 23.15 `udp_output` function: temporarily connect an unconnected socket.

Discard optional control information

333–344 Any optional control information is discarded by `m_freem`, without generating an error. UDP output does not use control information for any purpose.

> The comment `XXX` is because the control information is ignored without generating an error. Other protocols, such as the routing domain and TCP, generate an error if the process passes control information.

Temporarily connect an unconnected socket

345–359 If the caller specifies a destination address for the UDP datagram (`addr` is nonnull), the socket is temporarily connected to that destination address by `in_pcbconnect`. The socket will be disconnected at the end of this function. Before doing this connect, a check is made as to whether the socket is already connected, and, if so, the error `EISCONN` is returned. This is why a `sendto` that specifies a destination address on a connected socket returns an error.

Before the socket is temporarily connected, IP input processing is stopped by `splnet`. This is done because the temporary connect changes the foreign address, foreign port, and possibly the local address in the socket's PCB. If a received UDP datagram were processed while this PCB was temporarily connected, that datagram could be delivered to the wrong process. Setting the processor priority to `splnet` only stops a software interrupt from causing the IP input routine to be executed (Figure 1.12), it does not prevent the interface layer from accepting incoming packets and placing them onto IP's input queue.

> [Partridge and Pink 1993] note that this operation of temporarily connecting the socket is expensive and consumes nearly one-third of the cost of each UDP transmission.

The local address from the PCB is saved in `laddr` before temporarily connecting, because if it is the wildcard address it will be changed by `in_pcbconnect` when it calls `in_pcbbind`.

The same rules apply to the destination address that would apply if the process called `connect`, since `in_pcbconnect` is called for both cases.

360–364 If the process doesn't specify a destination address, and the socket is not connected, `ENOTCONN` is returned.

Prepend IP and UDP headers

366–373 `M_PREPEND` allocates room for the IP and UDP headers in front of the data. Figure 1.8 showed one scenario, assuming there is not room in the first mbuf on the chain for the 28 bytes of header. Exercise 23.1 details the other possible scenarios. The flag `M_DONTWAIT` is specified because if the socket is temporarily connected, IP processing is blocked, and `M_PREPEND` should not block.

> Earlier Berkeley releases incorrectly specified `M_WAIT` here.

Prepending IP/UDP Headers and Mbuf Clusters

There is a subtle interaction between the `M_PREPEND` macro and mbuf clusters. If the user data is placed into a cluster by `sosend`, then 56 bytes (`max_hdr` from Figure 7.17) are left unused at the beginning of the cluster, allowing room for the Ethernet, IP, and UDP headers. This is to prevent `M_PREPEND` from allocating another mbuf just to hold these headers. `M_PREPEND` calls `M_LEADINGSPACE` to calculate how much space is available at the beginning of the mbuf:

```
#define M_LEADINGSPACE(m) \
    ((m)->m_flags & M_EXT ? /* (m)->m_data - (m)->m_ext.ext_buf */ 0 : \
        (m)->m_flags & M_PKTHDR ? (m)->m_data - (m)->m_pktdat : \
        (m)->m_data - (m)->m_dat)
```

The code that correctly calculates the amount of room at the front of a cluster is commented out, and the macro always returns 0 if the data is in a cluster. This means that when the user data is in a cluster, `M_PREPEND` always allocates a new mbuf for the protocol headers instead of using the room allocated for this purpose by `sosend`.

> The reason for commenting out the correct code in `M_LEADINGSPACE` is that the cluster might be shared (Section 2.9), and, if it is shared, using the space before the user's data in the cluster could wipe out someone else's data.
>
> With UDP data, clusters are not shared, since `udp_output` does not save a copy of the data. TCP, however, saves a copy of the data in its send buffer (waiting for the data to be acknowledged), and if the data is in a cluster, it is shared. But `tcp_output` doesn't call `M_LEADINGSPACE`, because `sosend` leaves room for only 56 bytes at the beginning of the cluster for datagram protocols. `tcp_output` always calls `MGETHDR` instead, to allocate an mbuf for the protocol headers.

UDP Checksum Calculation and Pseudo-Header

Before showing the last half of `udp_output` we describe how UDP fills in some of the fields in the IP/UDP headers, calculates the UDP checksum, and passes the IP/UDP headers and the data to IP for output. The way this is done with the `ipovly` structure is tricky.

Figure 23.16 shows the 28-byte IP/UDP headers that are built by `udp_output` in the first mbuf in the chain pointed to by `m`. The unshaded fields are filled in by `udp_output` and the shaded fields are filled in by `ip_output`. This figure shows the format of the headers as they appear on the wire.

The UDP checksum is calculated over three areas: (1) a 12-byte pseudo-header containing fields from the IP header, (2) the 8-byte UDP header, and (3) the UDP data. Figure 23.17 shows the 12 bytes of pseudo-header used for the checksum computation, along with the UDP header. The UDP header used for the checksum calculation is identical to the UDP header that appears on the wire (Figure 23.16).

The following three facts are used in computing the UDP checksum. (1) The third 32-bit word in the pseudo-header (Figure 23.17) looks similar to the third 32-bit word in the IP header (Figure 23.16): two 8-bit values and a 16-bit value. (2) The order of the three 32-bit values in the pseudo-header is irrelevant. Actually, the computation of the Internet checksum does not depend on the order of the 16-bit values that are used (Section 8.7). (3) Including additional 32-bit words of 0 in the checksum computation has no effect.

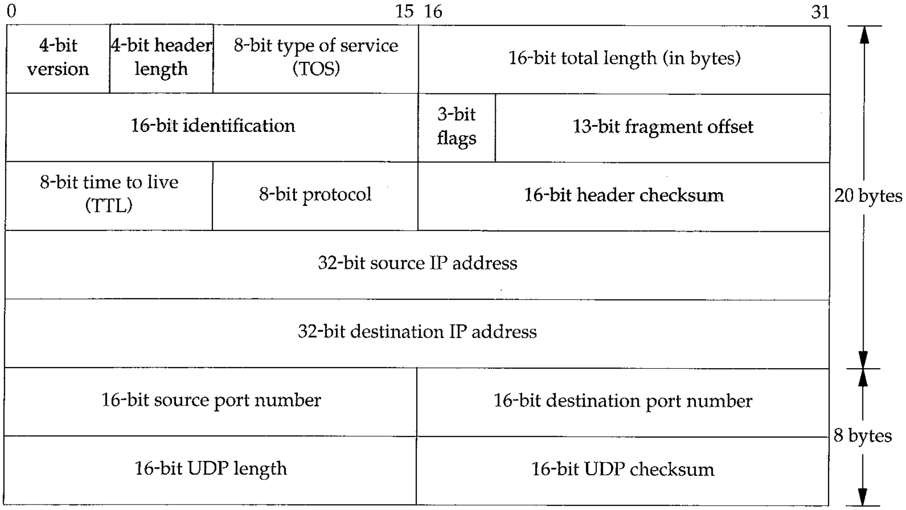

Figure 23.16 IP/UDP headers: unshaded fields filled in by UDP; shaded fields filled in by IP.

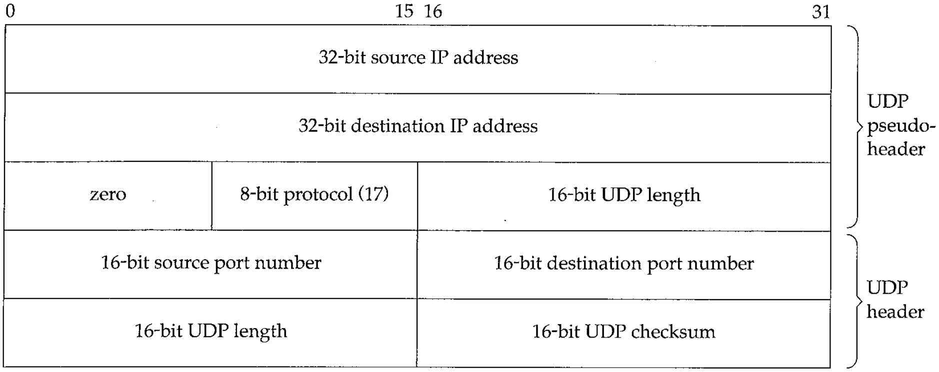

Figure 23.17 Pseudo-header used for checksum computation and UDP header.

`udp_output` takes advantage of these three facts and fills in the fields in the `udpiphdr` structure (Figure 23.11), which we depict in Figure 23.18. This structure is contained in the first mbuf in the chain pointed to by the argument `m`.

The last three 32-bit words in the 20-byte IP header (the five members `ui_x1`, `ui_pr`, `ui_len`, `ui_src`, and `ui_dst`) are used as the pseudo-header for the checksum computation. The first two 32-bit words in the IP header (`ui_next` and `ui_prev`) are also used in the checksum computation, but they're initialized to 0, and don't affect the checksum.

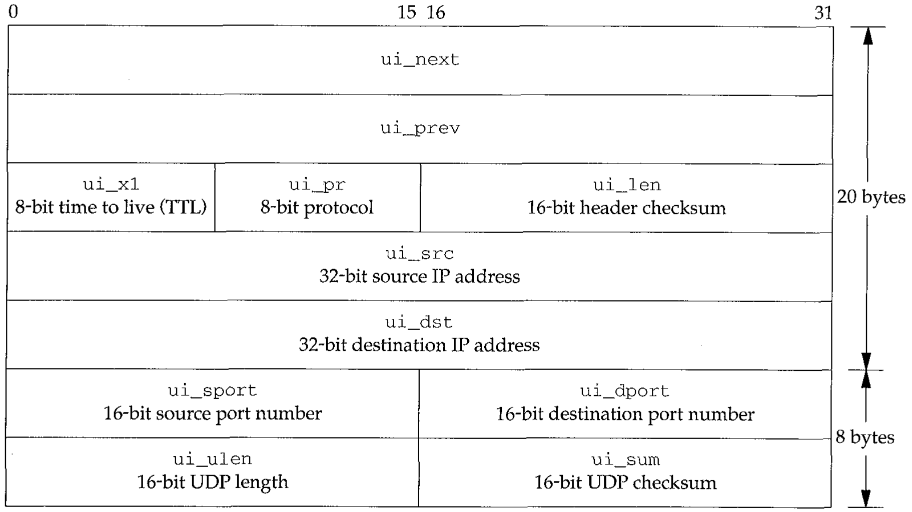

Figure 23.18 `udpiphdr` structure used by `udp_output`.

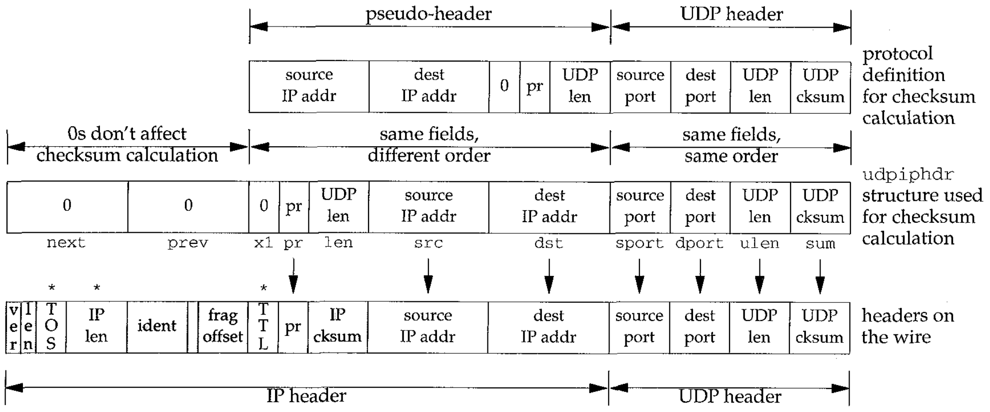

Figure 23.19 Operations to fill in IP/UDP headers and calculate UDP checksum.

Figure 23.19 summarizes the operations we've described.

1. The top picture shown in Figure 23.19 is the protocol definition of the pseudo-header, which corresponds to Figure 23.17.

2. The middle picture is the `udpiphdr` structure that is used in the source code, which corresponds to Figure 23.11. (To make the figure readable, the prefix `ui_` has been left off all the members.) This is the structure built by `udp_output` in the first mbuf and then used to calculate the UDP checksum.
3. The bottom picture shows the IP/UDP headers that appear on the wire, which corresponds to Figure 23.16. The seven fields with an arrow above are filled in by `udp_output` before the checksum computation. The three fields with an asterisk above are filled in by `udp_output` after the checksum computation. The remaining six shaded fields are filled in by `ip_output`.

Figure 23.20 shows the last half of the `udp_output` function.

udp_usrreq.c

```
    /*
     * Fill in mbuf with extended UDP header
     * and addresses and length put into network format.
     */
    ui = mtod(m, struct udpiphdr *);
    ui->ui_next = ui->ui_prev = 0;
    ui->ui_x1 = 0;
    ui->ui_pr = IPPROTO_UDP;
    ui->ui_len = htons((u_short) len + sizeof(struct udphdr));
    ui->ui_src = inp->inp_laddr;
    ui->ui_dst = inp->inp_faddr;
    ui->ui_sport = inp->inp_lport;
    ui->ui_dport = inp->inp_fport;
    ui->ui_ulen = ui->ui_len;

    /*
     * Stuff checksum and output datagram.
     */
    ui->ui_sum = 0;
    if (udpcksum) {
        if ((ui->ui_sum = in_cksum(m, sizeof(struct udpiphdr) + len)) == 0)
                    ui->ui_sum = 0xffff;
    }
    ((struct ip *) ui)->ip_len = sizeof(struct udpiphdr) + len;
    ((struct ip *) ui)->ip_ttl = inp->inp_ip.ip_ttl;    /* XXX */
    ((struct ip *) ui)->ip_tos = inp->inp_ip.ip_tos;    /* XXX */
    udpstat.udps_opackets++;
    error = ip_output(m, inp->inp_options, &inp->inp_route,
              inp->inp_socket->so_options & (SO_DONTROUTE | SO_BROADCAST),
                      inp->inp_moptions);

    if (addr) {
        in_pcbdisconnect(inp);
        inp->inp_laddr = laddr;
        splx(s);
    }
    return (error);
```

udp_usrreq.c

Figure 23.20 udp_output function: fill in headers, calculate checksum, pass to IP.

Prepare pseudo-header for checksum computation

374–387 All the members in the `udpiphdr` structure (Figure 23.18) are set to their respective values. The local and foreign sockets from the PCB are already in network byte order, but the UDP length must be converted to network byte order. The UDP length is the number of bytes of data (`len`, which can be 0) plus the size of the UDP header (8). The UDP length field appears twice in the UDP checksum calculation: `ui_len` and `ui_ulen`. One of them is redundant.

Calculate checksum

388–395 The checksum is calculated by first setting it to 0 and then calling `in_cksum`. If UDP checksums are disabled (a bad idea—see Section 11.3 of Volume 1), 0 is sent as the checksum. If the calculated checksum is 0, 16 one bits are stored in the header instead of 0. (In one's complement arithmetic, all one bits and all zero bits are both considered 0.) This allows the receiver to distinguish between a UDP packet without a checksum (the checksum field is 0) versus a UDP packet with a checksum whose value is 0 (the checksum is 16 one bits).

> The variable `udpcksum` (Figure 23.3) normally defaults to 1, enabling UDP checksums. The kernel can be compiled for 4.2BSD compatibility, which initializes `udpcksum` to 0.

Fill in UDP length, TTL, and TOS

396–398 The pointer `ui` is cast to a pointer to a standard IP header (`ip`), and three fields in the IP header are set by UDP. The IP length field is set to the amount of data in the UDP datagram, plus 28, the size of the IP/UDP headers. Notice that this field in the IP header is stored in host byte order, not network byte order like the rest of the multibyte fields in the header. `ip_output` converts it to network byte order before transmission.

The TTL and TOS fields in the IP header are then set from the values in the socket's PCB. These values are defaulted by UDP when the socket is created, but can be changed by the process using `setsockopt`. Since these three fields—IP length, TTL, and TOS—are not part of the pseudo-header and not used in the UDP checksum computation, they must be set after the checksum is calculated but before `ip_output` is called.

Send datagram

400–402 `ip_output` sends the datagram. The second argument, `inp_options`, are IP options the process can set using `setsockopt`. These IP options are placed into the IP header by `ip_output`. The third argument is a pointer to the cached route in the PCB, and the fourth argument is the socket options. The only socket options that are passed to `ip_output` are `SO_DONTROUTE` (bypass the routing tables) and `SO_BROADCAST` (allow broadcasting). The final argument is a pointer to the multicast options for this socket.

Disconnect temporarily connected socket

403–407 If the socket was temporarily connected, `in_pcbdisconnect` disconnects the socket, the local IP address is restored in the PCB, and the interrupt level is restored to its saved value.

23.7 `udp_input` Function

UDP output is driven by a process calling one of the five write functions. The functions shown in Figure 23.14 are all called directly as part of the system call. UDP input, on the other hand, occurs when IP input receives an IP datagram on its input queue whose protocol field specifies UDP. IP calls the function `udp_input` through the `pr_input` function in the protocol switch table (Figure 8.15). Since IP input is at the software interrupt level, `udp_input` also executes at this level. The goal of `udp_input` is to place the UDP datagram onto the appropriate socket's buffer and wake up any process blocked for input on that socket.

We'll divide our discussion of the `udp_input` function into three sections:

1. the general validation that UDP performs on the received datagram,
2. processing UDP datagrams destined for a unicast address: locating the appropriate PCB and placing the datagram onto the socket's buffer, and
3. processing UDP datagrams destined for a broadcast or multicast address: the datagram may be delivered to multiple sockets.

This last step is new with the support of multicasting in Net/3, but consumes almost one-third of the code.

General Validation of Received UDP Datagram

Figure 23.21 shows the first section of UDP input.

55–65 The two arguments to `udp_input` are `m`, a pointer to an mbuf chain containing the IP datagram, and `iphlen`, the length of the IP header (including possible IP options).

Discard IP options

67–76 If IP options are present they are discarded by `ip_stripoptions`. As the comments indicate, UDP should save a copy of the IP options and make them available to the receiving process through the `IP_RECVOPTS` socket option, but this isn't implemented yet.

77–88 If the length of the first mbuf on the mbuf chain is less than 28 bytes (the size of the IP header plus the UDP header), `m_pullup` rearranges the mbuf chain so that at least 28 bytes are stored contiguously in the first mbuf.

udp_usrreq.c

```
void
udp_input(m, iphlen)
struct mbuf *m;
int     iphlen;
{
    struct ip *ip;
    struct udphdr *uh;
    struct inpcb *inp;
    struct mbuf *opts = 0;
    int     len;
    struct ip save_ip;

    udpstat.udps_ipackets++;

    /*
     * Strip IP options, if any; should skip this,
     * make available to user, and use on returned packets,
     * but we don't yet have a way to check the checksum
     * with options still present.
     */
    if (iphlen > sizeof(struct ip)) {
        ip_stripoptions(m, (struct mbuf *) 0);
        iphlen = sizeof(struct ip);
    }
    /*
     * Get IP and UDP header together in first mbuf.
     */
    ip = mtod(m, struct ip *);
    if (m->m_len < iphlen + sizeof(struct udphdr)) {
        if ((m = m_pullup(m, iphlen + sizeof(struct udphdr))) == 0) {
            udpstat.udps_hdrops++;
            return;
        }
        ip = mtod(m, struct ip *);
    }
    uh = (struct udphdr *) ((caddr_t) ip + iphlen);

    /*
     * Make mbuf data length reflect UDP length.
     * If not enough data to reflect UDP length, drop.
     */
    len = ntohs((u_short) uh->uh_ulen);
    if (ip->ip_len != len) {
        if (len > ip->ip_len) {
            udpstat.udps_badlen++;
            goto bad;
        }
        m_adj(m, len - ip->ip_len);
        /* ip->ip_len = len; */
    }
    /*
     * Save a copy of the IP header in case we want to restore
     * it for sending an ICMP error message in response.
     */
    save_ip = *ip;
```

```
107     /*
108      * Checksum extended UDP header and data.
109      */
110     if (udpcksum && uh->uh_sum) {
111         ((struct ipovly *) ip)->ih_next = 0;
112         ((struct ipovly *) ip)->ih_prev = 0;
113         ((struct ipovly *) ip)->ih_x1 = 0;
114         ((struct ipovly *) ip)->ih_len = uh->uh_ulen;
115         if (uh->uh_sum = in_cksum(m, len + sizeof(struct ip))) {
116             udpstat.udps_badsum++;
117             m_freem(m);
118             return;
119         }
120     }
```

udp_usrreq.c

Figure 23.21 `udp_input` function: general validation of received UDP datagram.

Verify UDP length

89–101 There are two lengths associated with a UDP datagram: the length field in the IP header (`ip_len`) and the length field in the UDP header (`uh_ulen`). Recall that `ipintr` subtracted the length of the IP header from `ip_len` before calling `udp_input` (Figure 10.11). The two lengths are compared and there are three possibilities:

1. `ip_len` equals `uh_ulen`. This is the common case.
2. `ip_len` is greater than `uh_ulen`. The IP datagram is too big, as shown in Figure 23.22.

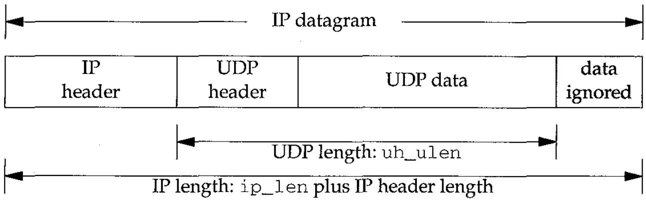

Figure 23.22 UDP length too small.

The code believes the smaller of the two lengths (the UDP header length) and `m_adj` removes the excess bytes of data from the end of the datagram. In the code the second argument to `m_adj` is negative, which we said in Figure 2.20 trims data from the end of the mbuf chain. It is possible in this scenario that the UDP length field has been corrupted. If so, the datagram will probably be discarded shortly, assuming the sender calculated the UDP checksum, that this checksum detects the error, and that the receiver verifies the checksum. The IP length field should be correct since it was verified by IP against the amount of data received from the interface, and the IP length field is covered by the mandatory IP header checksum.

3. `ip_len` is less than `uh_ulen`. The IP datagram is smaller than possible, given the length in the UDP header. Figure 23.23 shows this case.

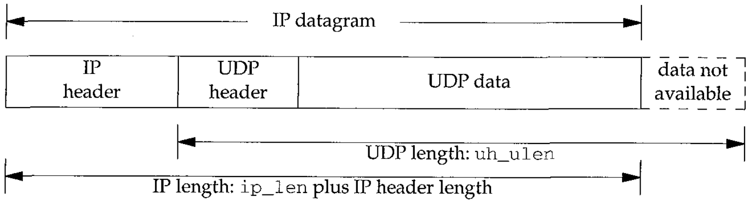

Figure 23.23 UDP length too big.

Something is wrong and the datagram is discarded. There is no other choice here: if the UDP length field has been corrupted, it can't be detected with the UDP checksum. The correct UDP length is needed to calculate the checksum.

As we've said, the UDP length is redundant. In Chapter 28 we'll see that TCP does not have a length field in its header—it uses the IP length field, minus the lengths of the IP and TCP headers, to determine the amount of data in the datagram. Why does the UDP length field exist? Possibly to add a small amount of error checking, since UDP checksums are optional.

Save copy of IP header and verify UDP checksum

102–106 `udp_input` saves a copy of the IP header before verifying the checksum, because the checksum computation wipes out some of the fields in the original IP header.

110 The checksum is verified only if UDP checksums are enabled for the kernel (`udpcksum`), and if the sender calculated a UDP checksum (the received checksum is nonzero).

This test is incorrect. If the sender calculated a checksum, it should be verified, regardless of whether outgoing checksums are calculated or not. The variable `udpcksum` should only specify whether outgoing checksums are calculated. Unfortunately many vendors have copied this incorrect test, although many vendors today finally ship their kernels with UDP checksums enabled by default.

111–120 Before calculating the checksum, the IP header is referenced as an `ipovly` structure (Figure 23.18) and the fields are initialized as described in the previous section when the UDP checksum is calculated by `udp_output`.

At this point special code is executed if the datagram is destined for a broadcast or multicast IP address. We defer this code until later in the section.

Demultiplexing Unicast Datagrams

Assuming the datagram is destined for a unicast address, Figure 23.24 shows the code that is executed.

udp_usrreq.c

```
           /* demultiplex broadcast & multicast datagrams (Figure 23.26) */

206     /*
207      * Locate pcb for unicast datagram.
208      */
209     inp = udp_last_inpcb;
210     if (inp->inp_lport != uh->uh_dport ||
211         inp->inp_fport != uh->uh_sport ||
212         inp->inp_faddr.s_addr != ip->ip_src.s_addr ||
213         inp->inp_laddr.s_addr != ip->ip_dst.s_addr) {

214         inp = in_pcblookup(&udb, ip->ip_src, uh->uh_sport,
215                            ip->ip_dst, uh->uh_dport, INPLOOKUP_WILDCARD);
216         if (inp)
217             udp_last_inpcb = inp;
218         udpstat.udpps_pcbcachemiss++;
219     }
220     if (inp == 0) {
221         udpstat.udps_noport++;
222         if (m->m_flags & (M_BCAST | M_MCAST)) {
223             udpstat.udps_noportbcast++;
224             goto bad;
225         }
226         *ip = save_ip;
227         ip->ip_len += iphlen;
228         icmp_error(m, ICMP_UNREACH, ICMP_UNREACH_PORT, 0, 0);
229         return;
230     }
```

udp_usrreq.c

Figure 23.24 udp_input function: demultiplex unicast datagram.

Check one-behind cache

206–209 UDP maintains a pointer to the last Internet PCB for which it received a datagram, `udp_last_inpcb`. Before calling `in_pcblookup`, which might have to search many PCBs on the UDP list, the foreign and local addresses and ports of that last PCB are compared against the received datagram. This is called a *one-behind cache* [Partridge and Pink 1993], and it is based on the assumption that the next datagram received has a high probability of being destined for the same socket as the last received datagram [Mogul 1991]. This cache was introduced with the 4.3BSD Tahoe release.

210–213 The order of the four comparisons between the cached PCB and the received datagram is intentional. If the PCBs don't match, the comparisons should stop as soon as possible. The highest probability is that the destination port numbers are different—this is therefore the first test. The lowest probability of a mismatch is between the local addresses, especially on a host with just one interface, so this is the last test.

Unfortunately this one-behind cache, as coded, is practically useless [Partridge and Pink 1993]. The most common type of UDP server binds only its well-known port, leaving its local address, foreign address, and foreign port wildcarded. The most common type of UDP client does not connect its UDP socket; it specifies the destination address for each datagram using `sendto`. Therefore most of the time the three values in the PCB `inp_laddr`, `inp_faddr`, and `inp_fport` are wildcards. In the cache comparison the four values in the received datagram are never wildcards, meaning the cache entry will compare equal with the received datagram only when the PCB has all four local and foreign values specified to nonwildcard values. This happens only for a connected UDP socket.

> On the system `bsdi`, the counter `udpps_pcbcachemiss` was 41,253 and the counter `udps_ipackets` was 42,485. This is less than a 3% cache hit rate.
>
> The `netstat -s` command prints most of the fields in the `udpstat` structure (Figure 23.5). Unfortunately the Net/3 version, and most vendor's versions, never print `udpps_pcbcachemiss`. If you want to see the value, use a debugger to examine the variable in the running kernel.

Search all UDP PCBs

214–218 Assuming the comparison with the cached PCB fails, `in_pcblookup` searches for a match. The `INPLOOKUP_WILDCARD` flag is specified, allowing a wildcard match. If a match is found, the pointer to the PCB is saved in `udp_last_inpcb`, which we said is a cache of the last received UDP datagram's PCB.

Generate ICMP port unreachable error

220–230 If a matching PCB is not found, UDP normally generates an ICMP port unreachable error. First the `m_flags` for the received mbuf chain is checked to see if the datagram was sent to a link-level broadcast or multicast destination address. It is possible to receive an IP datagram with a unicast IP address that was sent to a broadcast or multicast link-level address, but an ICMP port unreachable error must not be generated. If it is OK to generate the ICMP error, the IP header is restored to its received value (`save_ip`) and the IP length is also set back to its original value.

> This check for a link-level broadcast or multicast address is redundant. `icmp_error` also performs this check. The only advantage in this redundant check is to maintain the counter `udps_noportbcast` in addition to the counter `udps_noport`.
>
> The addition of `iphlen` back into `ip_len` is a bug. `icmp_error` will also do this, causing the IP length field in the IP header returned in the ICMP error to be 20 bytes too large. You can tell if a system has this bug by adding a few lines of code to the Traceroute program (Chapter 8 of Volume 1) to print this field in the ICMP port unreachable that is returned when the destination host is finally reached.

Figure 23.25 is the next section of processing for a unicast datagram, delivering the datagram to the socket corresponding to the destination PCB.

udp_usrreq.c

```
    /*
     * Construct sockaddr format source address.
     * Stuff source address and datagram in user buffer.
     */
    udp_in.sin_port = uh->uh_sport;
    udp_in.sin_addr = ip->ip_src;

    if (inp->inp_flags & INP_CONTROLOPTS) {
        struct mbuf **mp = &opts;

        if (inp->inp_flags & INP_RECVDSTADDR) {
            *mp = udp_saveopt((caddr_t) & ip->ip_dst,
                              sizeof(struct in_addr), IP_RECVDSTADDR);
            if (*mp)
                mp = &(*mp)->m_next;
        }
#ifdef notyet
        /* IP options were tossed above */
        if (inp->inp_flags & INP_RECVOPTS) {
            *mp = udp_saveopt((caddr_t) opts_deleted_above,
                              sizeof(struct in_addr), IP_RECVOPTS);
            if (*mp)
                mp = &(*mp)->m_next;
        }
        /* ip_srcroute doesn't do what we want here, need to fix */
        if (inp->inp_flags & INP_RECVRETOPTS) {
            *mp = udp_saveopt((caddr_t) ip_srcroute(),
                              sizeof(struct in_addr), IP_RECVRETOPTS);
            if (*mp)
                mp = &(*mp)->m_next;
        }
#endif
    }
    iphlen += sizeof(struct udphdr);
    m->m_len -= iphlen;
    m->m_pkthdr.len -= iphlen;
    m->m_data += iphlen;
    if (sbappendaddr(&inp->inp_socket->so_rcv, (struct sockaddr *) &udp_in,
                     m, opts) == 0) {
        udpstat.udps_fullsock++;
        goto bad;
    }
    sorwakeup(inp->inp_socket);
    return;

  bad:
    m_freem(m);
    if (opts)
        m_freem(opts);
}
```

udp_usrreq.c

Figure 23.25 udp_input function: deliver unicast datagram to socket.

Return source IP address and source port

231–236 The source IP address and source port number from the received IP datagram are stored in the global `sockaddr_in` structure `udp_in`. This structure is passed as an argument to `sbappendaddr` later in the function.

Using a global to hold the IP address and port number is OK because `udp_input` is single threaded. When this function is called by `ipintr` it processes the received datagram completely before returning. Also, `sbappendaddr` copies the socket address structure from the global into an mbuf.

`IP_RECVDSTADDR` socket option

237–244 The constant `INP_CONTROLOPTS` is the combination of the three socket options that the process can set to cause control information to be returned through the `recvmsg` system call for a UDP socket (Figure 22.5). The `IP_RECVDSTADDR` socket option returns the destination IP address from the received UDP datagram as control information. The function `udp_saveopt` allocates an mbuf of type `MT_CONTROL` and stores the 4-byte destination IP address in the mbuf. We show this function in Section 23.8.

> This socket option appeared with 4.3BSD Reno and was intended for applications such as TFTP, the Trivial File Transfer Protocol, that should not respond to client requests that are sent to a broadcast address. Unfortunately, even if the receiving application uses this option, it is nontrivial to determine if the destination IP address is a broadcast address or not (Exercise 23.6).
>
> When the multicasting changes were added in 4.4BSD, this code was left in only for datagrams destined for a unicast address. We'll see in Figure 23.26 that this option is not implemented for datagrams sent to a broadcast of multicast address. This defeats the purpose of the option!

Unimplemented socket options

245–260 This code is commented out because it doesn't work. The intent of the `IP_RECVOPTS` socket option is to return the IP options from the received datagram as control information, and the intent of `IP_RECVRETOPTS` socket option is to return source route information. The manipulation of the `mp` variable by all three `IP_RECV` socket options is to build a linked list of up to three mbufs that are then placed onto the socket's buffer by `sbappendaddr`. The code shown in Figure 23.25 only returns one option as control information, so the `m_next` pointer of that mbuf is always a null pointer.

Append data to socket's receive queue

262–272 At this point the received datagram (the mbuf chain pointed to by `m`), is ready to be placed onto the socket's receive queue along with a socket address structure representing the sender's IP address and port (`udp_in`), and optional control information (the destination IP address, the mbuf pointed to by `opts`). This is done by `sbappendaddr`. Before calling this function, however, the pointer and lengths of the first mbuf on the chain are adjusted to ignore the IP and UDP headers. Before returning, `sorwakeup` is called for the receiving socket to wake up any processes asleep on the socket's receive queue.

Error return

273–276 If an error is encountered during UDP input processing, `udp_input` jumps to the label `bad`. The mbuf chain containing the datagram is released, along with the mbuf chain containing any control information (if present).

Demultiplexing Multicast and Broadcast Datagrams

We now return to the portion of `udp_input` that handles datagrams sent to a broadcast or multicast IP address. The code is shown in Figure 23.26.

121–138 As the comments indicate, these datagrams are delivered to *all* sockets that match, not just a single socket. The inadequacy of the UDP interface that is mentioned refers to the inability of a process to receive asynchronous errors on a UDP socket (notably ICMP port unreachables) unless the socket is connected. We described this in Section 22.11.

139–145 The source IP address and port number are saved in the global `sockaddr_in` structure `udp_in`, which is passed to `sbappendaddr`. The mbuf chain's length and data pointer are updated to ignore the IP and UDP headers.

146–164 The large `for` loop scans each UDP PCB to find all matching PCBs. `in_pcblookup` is not called for this demultiplexing because it returns only one PCB, whereas the broadcast or multicast datagram may be delivered to more than one PCB.

If the local port in the PCB doesn't match the destination port from the received datagram, the entry is ignored. If the local address in the PCB is not the wildcard, it is compared to the destination IP address and the entry is skipped if they're not equal. If the foreign address in the PCB is not a wildcard, it is compared to the source IP address and if they match, the foreign port must also match the source port. This last test assumes that if the socket is connected to a foreign IP address it must also be connected to a foreign port, and vice versa. This is the same logic we saw in `in_pcblookup`.

165–177 If this is not the first match found (`last` is nonnull), a copy of the datagram is placed onto the receive queue for the previous match. Since `sbappendaddr` releases the mbuf chain when it is done, a copy is first made by `m_copy`. Any processes waiting for this data are awakened by `sorwakeup`. A pointer to this matching `socket` structure is saved in `last`.

This use of the variable `last` avoids calling `m_copy` (an expensive operation since an entire mbuf chain is copied) unless there are multiple recipients for a given datagram. In the common case of a single recipient, the `for` loop just sets `last` to the single matching PCB, and when the loop terminates, `sbappendaddr` places the mbuf chain onto the socket's receive queue—a copy is not made.

178–188 If this matching socket doesn't have either the `SO_REUSEPORT` or the `SO_REUSEADDR` socket option set, then there's no need to check for additional matches and the loop is terminated. The datagram is placed onto the single socket's receive queue in the call to `sbappendaddr` outside the loop.

189–197 If `last` is null at the end of the loop, no matches were found. An ICMP error is not generated because the datagram was sent to a broadcast or multicast IP address.

udp_usrreq.c

```
    if (IN_MULTICAST(ntohl(ip->ip_dst.s_addr)) ||
        in_broadcast(ip->ip_dst, m->m_pkthdr.rcvif)) {
        struct socket *last;
        /*
         * Deliver a multicast or broadcast datagram to *all* sockets
         * for which the local and remote addresses and ports match
         * those of the incoming datagram.  This allows more than
         * one process to receive multi/broadcasts on the same port.
         * (This really ought to be done for unicast datagrams as
         * well, but that would cause problems with existing
         * applications that open both address-specific sockets and
         * a wildcard socket listening to the same port -- they would
         * end up receiving duplicates of every unicast datagram.
         * Those applications open the multiple sockets to overcome an
         * inadequacy of the UDP socket interface, but for backwards
         * compatibility we avoid the problem here rather than
         * fixing the interface.  Maybe 4.5BSD will remedy this?)
         */

        /*
         * Construct sockaddr format source address.
         */
        udp_in.sin_port = uh->uh_sport;
        udp_in.sin_addr = ip->ip_src;
        m->m_len -= sizeof(struct udpiphdr);
        m->m_data += sizeof(struct udpiphdr);
        /*
         * Locate pcb(s) for datagram.
         * (Algorithm copied from raw_intr().)
         */
        last = NULL;
        for (inp = udb.inp_next; inp != &udb; inp = inp->inp_next) {
            if (inp->inp_lport != uh->uh_dport)
                continue;
            if (inp->inp_laddr.s_addr != INADDR_ANY) {
                if (inp->inp_laddr.s_addr !=
                    ip->ip_dst.s_addr)
                    continue;
            }
            if (inp->inp_faddr.s_addr != INADDR_ANY) {
                if (inp->inp_faddr.s_addr !=
                    ip->ip_src.s_addr ||
                    inp->inp_fport != uh->uh_sport)
                    continue;
            }
            if (last != NULL) {
                struct mbuf *n;

                if ((n = m_copy(m, 0, M_COPYALL)) != NULL) {
                    if (sbappendaddr(&last->so_rcv,
                                     (struct sockaddr *) &udp_in,
                                     n, (struct mbuf *) 0) == 0) {
                        m_freem(n);
                        udpstat.udps_fullsock++;
```

```
173                     } else
174                         sorwakeup(last);
175                 }
176             }
177             last = inp->inp_socket;
178             /*
179              * Don't look for additional matches if this one does
180              * not have either the SO_REUSEPORT or SO_REUSEADDR
181              * socket options set.  This heuristic avoids searching
182              * through all pcbs in the common case of a non-shared
183              * port.  It assumes that an application will never
184              * clear these options after setting them.
185              */
186             if ((last->so_options & (SO_REUSEPORT | SO_REUSEADDR) == 0))
187                 break;
188         }
189         if (last == NULL) {
190             /*
191              * No matching pcb found; discard datagram.
192              * (No need to send an ICMP Port Unreachable
193              * for a broadcast or multicast datgram.)
194              */
195             udpstat.udps_noportbcast++;
196             goto bad;
197         }
198         if (sbappendaddr(&last->so_rcv, (struct sockaddr *) &udp_in,
199                          m, (struct mbuf *) 0) == 0) {
200             udpstat.udps_fullsock++;
201             goto bad;
202         }
203         sorwakeup(last);
204         return;
205     }
```

udp_usrreq.c

Figure 23.26 udp_input function: demultiplexing of broadcast and multicast datagrams.

198–204 The final matching entry (which could be the only matching entry) has the original datagram (m) placed onto its receive queue. After sorwakeup is called, udp_input returns, since the processing the broadcast or multicast datagram is complete.

The remainder of the function (shown previously in Figure 23.24) handles unicast datagrams.

Connected UDP Sockets and Multihomed Hosts

There is a subtle problem when using a connected UDP socket to exchange datagrams with a process on a multihomed host. Datagrams from the peer may arrive with a different source IP address and will not be delivered to the connected socket.

Consider the example shown in Figure 23.27.

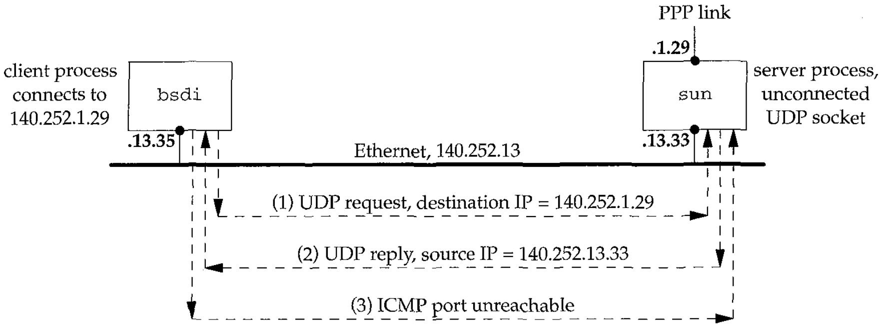

Figure 23.27 Example of connected UDP socket sending datagram to a multihomed host.

Three steps take place.

1. The client on `bsdi` creates a UDP socket and connects it to 140.252.1.29, the PPP interface on `sun`, not the Ethernet interface. A datagram is sent on the socket to the server.

   The server on `sun` receives the datagram and accepts it, even though it arrives on an interface that differs from the destination IP address. (`sun` is acting as a router, so whether it implements the weak end system model or the strong end system model doesn't matter.) The datagram is delivered to the server, which is waiting for client requests on an unconnected UDP socket.

2. The server sends a reply, but since the reply is being sent on an unconnected UDP socket, the source IP address for the reply is chosen by the kernel based on the outgoing interface (140.252.13.33). The destination IP address in the request is not used as the source address for the reply.

   When the reply is received by `bsdi` it is not delivered to the client's connected UDP socket since the IP addresses don't match.

3. `bsdi` generates an ICMP port unreachable error since the reply can't be demultiplexed. (This assumes that there is not another process on `bsdi` eligible to receive the datagram.)

The problem in this example is that the server does not use the destination IP address from the request as the source IP address of the reply. If it did, the problem wouldn't exist, but this solution is nontrivial—see Exercise 23.10. We'll see in Figure 28.16 that a TCP server uses the destination IP address from the client as the source IP address from the server, if the server has not explicitly bound a local IP address to its socket.

23.8 `udp_saveopt` Function

If a process specifies the `IP_RECVDSTADDR` socket option, to receive the destination IP address from the received datagram `udp_saveopt` is called by `udp_input`:

```
*mp = udp_saveopt((caddr_t) &ip->ip_dst, sizeof(struct in_addr),
                  IP_RECVDSTADDR);
```

Figure 23.28 shows this function.

udp_usrreq.c

```
278 /*
279  * Create a "control" mbuf containing the specified data
280  * with the specified type for presentation with a datagram.
281  */
282 struct mbuf *
283 udp_saveopt(p, size, type)
284 caddr_t p;
285 int     size;
286 int     type;
287 {
288     struct cmsghdr *cp;
289     struct mbuf *m;

290     if ((m = m_get(M_DONTWAIT, MT_CONTROL)) == NULL)
291         return ((struct mbuf *) NULL);
292     cp = (struct cmsghdr *) mtod(m, struct cmsghdr *);
293     bcopy(p, CMSG_DATA(cp), size);
294     size += sizeof(*cp);
295     m->m_len = size;
296     cp->cmsg_len = size;
297     cp->cmsg_level = IPPROTO_IP;
298     cp->cmsg_type = type;
299     return (m);
300 }
```

udp_usrreq.c

Figure 23.28 `udp_saveopt` function: create mbuf with control information.

278–289 The arguments are `p`, a pointer to the information to be stored in the mbuf (the destination IP address from the received datagram); `size`, its size in bytes (4 in this example, the size of an IP address); and `type`, the type of control information (`IP_RECVDSTADDR`).

290–299 An mbuf is allocated, and since the code is executing at the software interrupt layer, `M_DONTWAIT` is specified. The pointer `cp` points to the data portion of the mbuf, and it is cast into a pointer to a `cmsghdr` structure (Figure 16.14). The IP address is copied from the IP header into the data portion of the `cmsghdr` structure by `bcopy`. The length of the mbuf is then set (to 16 in this example), followed by the remainder of the `cmsghdr` structure. Figure 23.29 shows the final state of the mbuf.

The `cmsg_len` field contains the length of the `cmsghdr` structure (12) plus the size of the `cmsg_data` field (4 for this example). If the application calls `recvmsg` to receive the control information, it must go through the `cmsghdr` structure to determine the type and length of the `cmsg_data` field.

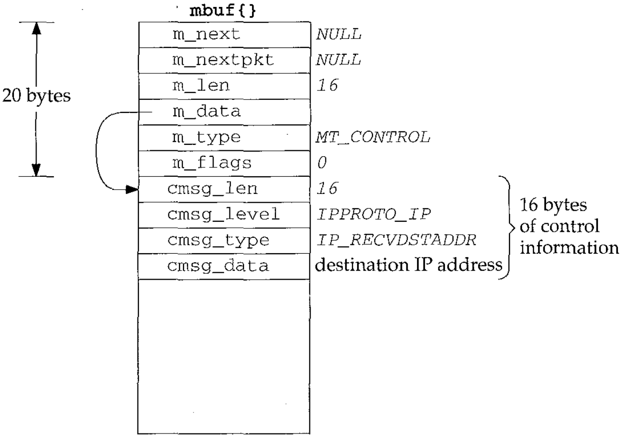

Figure 23.29 Mbuf containing destination address from received datagram as control information.

23.9 `udp_ctlinput` Function

When `icmp_input` receives an ICMP error (destination unreachable, parameter problem, redirect, source quench, and time exceeded) the corresponding protocol's `pr_ctlinput` function is called:

```
if (ctlfunc = inetsw[ ip_protox[icp->icmp_ip.ip_p] ].pr_ctlinput)
    (*ctlfunc)(code, (struct sockaddr *)&icmpsrc, &icp->icmp_ip);
```

For UDP, Figure 22.32 showed that the function `udp_ctlinput` is called. We show this function in Figure 23.30.

314–322 The arguments are `cmd`, one of the `PRC_`*xxx* constants from Figure 11.19; `sa`, a pointer to a `sockaddr_in` structure containing the source IP address from the ICMP message; and `ip`, a pointer to the IP header that caused the error. For the destination unreachable, parameter problem, source quench, and time exceeded errors, the pointer `ip` points to the IP header that caused the error. But when `udp_ctlinput` is called by `pfctlinput` for redirects (Figure 22.32), `sa` points to a `sockaddr_in` structure containing the destination address that should be redirected, and `ip` is a null pointer. There is no loss of information in this final case, since we saw in Section 22.11 that a redirect is applied to all TCP and UDP sockets connected to the destination address. The nonnull third argument is needed, however, for other errors, such as a port unreachable, since the protocol header following the IP header contains the unreachable port.

323–325 If the error is not a redirect, and either the `PRC_`*xxx* value is too large or there is no error code in the global array `inetctlerrmap`, the ICMP error is ignored. To understand this test we need to review what happens to a received ICMP message.

1. `icmp_input` converts the ICMP type and code into a `PRC_`*xxx* error code.
2. The `PRC_`*xxx* error code is passed to the protocol's control-input function.

udp_usrreq.c

```
314 void
315 udp_ctlinput(cmd, sa, ip)
316 int     cmd;
317 struct sockaddr *sa;
318 struct ip *ip;
319 {
320     struct udphdr *uh;
321     extern struct in_addr zeroin_addr;
322     extern u_char inetctlerrmap[];

323     if (!PRC_IS_REDIRECT(cmd) &&
324         ((unsigned) cmd >= PRC_NCMDS || inetctlerrmap[cmd] == 0))
325         return;
326     if (ip) {
327         uh = (struct udphdr *) ((caddr_t) ip + (ip->ip_hl << 2));
328         in_pcbnotify(&udb, sa, uh->uh_dport, ip->ip_src, uh->uh_sport,
329                      cmd, udp_notify);
330     } else
331         in_pcbnotify(&udb, sa, 0, zeroin_addr, 0, cmd, udp_notify);
332 }
```

udp_usrreq.c

Figure 23.30 `udp_ctlinput` function: process received ICMP errors.

3. The Internet protocols (TCP and UDP) map the `PRC_`*xxx* error code into one of the Unix `errno` values using `inetctlerrmap`, and this value is returned to the process.

Figures 11.1 and 11.2 summarize this processing of ICMP messages.

Returning to Figure 23.30, we can see what happens to an ICMP source quench that arrives in response to a UDP datagram. `icmp_input` converts the ICMP message into the error `PRC_QUENCH` and `udp_ctlinput` is called. But since the `errno` column for this ICMP error is blank in Figure 11.2, the error is ignored.

326–331 The function `in_pcbnotify` notifies the appropriate PCBs of the ICMP error. If the third argument to `udp_ctlinput` is nonnull, the source and destination UDP ports from the datagram that caused the error are passed to `in_pcbnotify` along with the source IP address.

`udp_notify` Function

The final argument to `in_pcbnotify` is a pointer to a function that `in_pcbnotify` calls for each PCB that is to receive the error. The function for UDP is `udp_notify` and we show it in Figure 23.31.

301–313 The `errno` value, the second argument to this function, is stored in the socket's `so_error` variable. By setting this socket variable, the socket becomes readable and writable if the process calls `select`. Any processes waiting to receive or send on the socket are then awakened to receive the error.

udp_usrreq.c

```
static void
udp_notify(inp, errno)
struct inpcb *inp;
int     errno;
{
    inp->inp_socket->so_error = errno;
    sorwakeup(inp->inp_socket);
    sowwakeup(inp->inp_socket);
}
```

udp_usrreq.c

Figure 23.31 `udp_notify` function: notify process of an asynchronous error.

23.10 `udp_usrreq` Function

The protocol's user-request function is called for a variety of operations. We saw in Figure 23.14 that a call to any one of the five write functions on a UDP socket ends up calling UDP's user-request function with a request of `PRU_SEND`.

Figure 23.32 shows the beginning and end of `udp_usrreq`. The body of the `switch` is discussed in separate figures following this figure. The function arguments are described in Figure 15.17.

udp_usrreq.c

```
int
udp_usrreq(so, req, m, addr, control)
struct socket *so;
int     req;
struct mbuf *m, *addr, *control;
{
    struct inpcb *inp = sotoinpcb(so);
    int     error = 0;
    int     s;

    if (req == PRU_CONTROL)
        return (in_control(so, (int) m, (caddr_t) addr,
                           (struct ifnet *) control));
    if (inp == NULL && req != PRU_ATTACH) {
        error = EINVAL;
        goto release;
    }
    /*
     * Note: need to block udp_input while changing
     * the udp pcb queue and/or pcb addresses.
     */
    switch (req) {


                                   /* switch cases */
```

```
522     default:
523         panic("udp_usrreq");
524     }

525   release:
526     if (control) {
527         printf("udp control data unexpectedly retained\n");
528         m_freem(control);
529     }
530     if (m)
531         m_freem(m);
532     return (error);
533 }
```

udp_usrreq.c

Figure 23.32 Body of `udp_usrreq` function.

417–428 The `PRU_CONTROL` request is from the `ioctl` system call. The function `in_control` processes the request completely.

429–432 The socket pointer was converted to the PCB pointer when `inp` was declared at the beginning of the function. The only time a null PCB pointer is allowed is when a new socket is being created (`PRU_ATTACH`).

433–436 The comment indicates that whenever entries are being added to or deleted from UDP's PCB list, the code must be protected by `splnet`. This is done because `udp_usrreq` is called as part of a system call, and it doesn't want to be interrupted by UDP input (called by IP input, which is called as a software interrupt) while it is modifying the doubly linked list of PCBs. UDP input is also blocked while modifying the local or foreign addresses or ports in a PCB, to prevent a received UDP datagram from being delivered incorrectly by `in_pcblookup`.

We now discuss the individual `case` statements. The `PRU_ATTACH` request, shown in Figure 23.33, is from the `socket` system call.

438–447 If the socket structure already points to a PCB, `EINVAL` is returned. `in_pcballoc` allocates a new PCB, adds it to the front of UDP's PCB list, and links the socket structure and the PCB to each other.

448–450 `soreserve` reserves buffer space for a receive buffer and a send buffer for the socket. As noted in Figure 16.7, `soreserve` just enforces system limits; the buffer space is not actually allocated. The default values for the send and receive buffer sizes are 9216 bytes (`udp_sendspace`) and 41,600 bytes (`udp_recvspace`). The former allows for a maximum UDP datagram size of 9200 bytes (to hold 8 Kbytes of data in an NFS packet), plus the 16-byte `sockaddr_in` structure for the destination address. The latter allows for 40 1024-byte datagrams to be queued at one time for the socket. The process can change these defaults by calling `setsockopt`.

451–452 There are two fields in the prototype IP header in the PCB that the process can change by calling `setsockopt`: the TTL and the TOS. The TTL defaults to 64 (`ip_defttl`) and the TOS defaults to 0 (normal service), since the PCB is initialized to 0 by `in_pcballoc`.

udp_usrreq.c

```
438     case PRU_ATTACH:
439         if (inp != NULL) {
440             error = EINVAL;
441             break;
442         }
443         s = splnet();
444         error = in_pcballoc(so, &udb);
445         splx(s);
446         if (error)
447             break;
448         error = soreserve(so, udp_sendspace, udp_recvspace);
449         if (error)
450             break;
451         ((struct inpcb *) so->so_pcb)->inp_ip.ip_ttl = ip_defttl;
452         break;

453     case PRU_DETACH:
454         udp_detach(inp);
455         break;
```

udp_usrreq.c

Figure 23.33 `udp_usrreq` function: `PRU_ATTACH` and `PRU_DETACH` requests.

453–455 The `close` system call issues the `PRU_DETACH` request. The function `udp_detach`, shown in Figure 23.34, is called. This function is also called later in this section for the `PRU_ABORT` request.

udp_usrreq.c

```
534 static void
535 udp_detach(inp)
536 struct inpcb *inp;
537 {
538     int     s = splnet();

539     if (inp == udp_last_inpcb)
540         udp_last_inpcb = &udb;
541     in_pcbdetach(inp);
542     splx(s);
543 }
```

udp_usrreq.c

Figure 23.34 `udp_detach` function: delete a UDP PCB.

If the last-received PCB pointer (the one-behind cache) points to the PCB being detached, the cache pointer is set to the head of the UDP list (`udb`). The function `in_pcbdetach` removes the PCB from UDP's list and releases the PCB.

Returning to `udp_usrreq`, a `PRU_BIND` request is the result of the `bind` system call and a `PRU_LISTEN` request is the result of the `listen` system call. Both are shown in Figure 23.35.

456–460 All the work for a `PRU_BIND` request is done by `in_pcbbind`.

461–463 The `PRU_LISTEN` request is invalid for a connectionless protocol—it is used only by connection-oriented protocols.

udp_usrreq.c

```
456     case PRU_BIND:
457         s = splnet();
458         error = in_pcbbind(inp, addr);
459         splx(s);
460         break;

461     case PRU_LISTEN:
462         error = EOPNOTSUPP;
463         break;
```

udp_usrreq.c

Figure 23.35 `udp_usrreq` function: `PRU_BIND` and `PRU_LISTEN` requests.

We mentioned earlier that a UDP application, either a client or server (normally a client), can call `connect`. This fixes the foreign IP address and port number that this socket can send to or receive from. Figure 23.36 shows the `PRU_CONNECT`, `PRU_CONNECT2`, and `PRU_ACCEPT` requests.

udp_usrreq.c

```
464     case PRU_CONNECT:
465         if (inp->inp_faddr.s_addr != INADDR_ANY) {
466             error = EISCONN;
467             break;
468         }
469         s = splnet();
470         error = in_pcbconnect(inp, addr);
471         splx(s);
472         if (error == 0)
473             soisconnected(so);
474         break;

475     case PRU_CONNECT2:
476         error = EOPNOTSUPP;
477         break;

478     case PRU_ACCEPT:
479         error = EOPNOTSUPP;
480         break;
```

udp_usrreq.c

Figure 23.36 `udp_usrreq` function: `PRU_CONNECT`, `PRU_CONNECT2`, and `PRU_ACCEPT` requests.

464–474 If the socket is already connected, `EISCONN` is returned. The socket should never be connected at this point, because a call to `connect` on an already-connected UDP socket generates a `PRU_DISCONNECT` request before this `PRU_CONNECT` request. Otherwise `in_pcbconnect` does all the work. If no errors are encountered, `soisconnected` marks the socket structure as being connected.

475–477 The `socketpair` system call issues the `PRU_CONNECT2` request, which is defined only for the Unix domain protocols.

478–480 The `PRU_ACCEPT` request is from the `accept` system call, which is defined only for connection-oriented protocols.

The `PRU_DISCONNECT` request can occur in two cases for a UDP socket:

1. When a connected UDP socket is closed, `PRU_DISCONNECT` is called before `PRU_DETACH`.
2. When a `connect` is issued on an already-connected UDP socket, `soconnect` issues the `PRU_DISCONNECT` request before the `PRU_CONNECT` request.

Figure 23.37 shows the `PRU_DISCONNECT` request.

udp_usrreq.c

```
481     case PRU_DISCONNECT:
482         if (inp->inp_faddr.s_addr == INADDR_ANY) {
483             error = ENOTCONN;
484             break;
485         }
486         s = splnet();
487         in_pcbdisconnect(inp);
488         inp->inp_laddr.s_addr = INADDR_ANY;
489         splx(s);
490         so->so_state &= ~SS_ISCONNECTED;    /* XXX */
491         break;
```

udp_usrreq.c

Figure 23.37 `udp_usrreq` function: `PRU_DISCONNECT` request.

If the socket is not already connected, `ENOTCONN` is returned. Otherwise `in_pcbdisconnect` sets the foreign IP address to 0.0.0.0 and the foreign port to 0. The local address is also set to 0.0.0.0, since this PCB variable could have been set by `connect`.

A call to `shutdown` specifying that the process has finished sending data generates the `PRU_SHUTDOWN` request, although it is rare for a process to issue this system call for a UDP socket. Figure 23.38 shows the `PRU_SHUTDOWN`, `PRU_SEND`, and `PRU_ABORT` requests.

udp_usrreq.c

```
492     case PRU_SHUTDOWN:
493         socantsendmore(so);
494         break;

495     case PRU_SEND:
496         return (udp_output(inp, m, addr, control));

497     case PRU_ABORT:
498         soisdisconnected(so);
499         udp_detach(inp);
500         break;
```

udp_usrreq.c

Figure 23.38 `udp_usrreq` function: `PRU_SHUTDOWN`, `PRU_SEND`, and `PRU_ABORT` requests.

492–494 `socantsendmore` sets the socket's flags to prevent any future output.

495–496 In Figure 23.14 we showed how the five write functions ended up calling `udp_usrreq` with a `PRU_SEND` request. `udp_output` sends the datagram. `udp_usrreq` returns, to avoid falling through to the label `release` (Figure 23.32), since the mbuf chain containing the data (`m`) must not be released yet. IP output appends this mbuf chain to the appropriate interface output queue, and the device driver will release the mbuf when the data has been transmitted.

The only buffering of UDP output within the kernel is on the interface's output queue. If there is room in the socket's send buffer for the datagram and destination address, `sosend` calls `udp_usrreq`, which we see calls `udp_output`. We saw in Figure 23.20 that `ip_output` is then called, which calls `ether_output` for an Ethernet, placing the datagram onto the interface's output queue (if there is room). If the process calls `sendto` faster than the interface can transmit the datagrams, `ether_output` can return `ENOBUFS`, which is returned to the process.

497–500 A `PRU_ABORT` request should never be generated for a UDP socket, but if it is, the socket is disconnected and the PCB detached.

The `PRU_SOCKADDR` and `PRU_PEERADDR` requests are from the `getsockname` and `getpeername` system calls, respectively. These two requests, and the `PRU_SENSE` request, are shown in Figure 23.39.

udp_usrreq.c

```
501     case PRU_SOCKADDR:
502         in_setsockaddr(inp, addr);
503         break;

504     case PRU_PEERADDR:
505         in_setpeeraddr(inp, addr);
506         break;

507     case PRU_SENSE:
508         /*
509          * fstat: don't bother with a blocksize.
510          */
511         return (0);
```

udp_usrreq.c

Figure 23.39 `udp_usrreq` function: `PRU_SOCKADDR`, `PRU_PEERADDR`, and `PRU_SENSE` requests.

501–506 The functions `in_setsockaddr` and `in_setpeeraddr` fetch the information from the PCB, storing the result in the `addr` argument.

507–511 The `fstat` system call generates the `PRU_SENSE` request. The function returns OK, but doesn't return any other information. We'll see later that TCP returns the size of the send buffer as the `st_blksize` element of the `stat` structure.

The remaining seven `PRU_`*xxx* requests, shown in Figure 23.40, are not supported for a UDP socket.

udp_usrreq.c

```
    case PRU_SENDOOB:
    case PRU_FASTTIMO:
    case PRU_SLOWTIMO:
    case PRU_PROTORCV:
    case PRU_PROTOSEND:
        error = EOPNOTSUPP;
        break;

    case PRU_RCVD:
    case PRU_RCVOOB:
        return (EOPNOTSUPP);    /* do not free mbuf's */
```

udp_usrreq.c

Figure 23.40 `udp_usrreq` function: unsupported requests.

There is a slight difference in how the last two are handled because `PRU_RCVD` doesn't pass a pointer to an mbuf as an argument (`m` is a null pointer) and `PRU_RCVOOB` passes a pointer to an mbuf for the protocol to fill in. In both cases the error is immediately returned, without breaking out of the `switch` and releasing the mbuf chain. With `PRU_RCVOOB` the caller releases the mbuf that it allocated.

23.11 `udp_sysctl` Function

The `sysctl` function for UDP supports only a single option, the UDP checksum flag. The system administrator can enable or disable UDP checksums using the `sysctl`(8) program. Figure 23.41 shows the `udp_sysctl` function. This function calls `sysctl_int` to fetch or set the value of the integer `udpcksum`.

udp_usrreq.c

```
udp_sysctl(name, namelen, oldp, oldlenp, newp, newlen)
int     *name;
u_int    namelen;
void    *oldp;
size_t  *oldlenp;
void    *newp;
size_t   newlen;
{
    /* All sysctl names at this level are terminal. */
    if (namelen != 1)
        return (ENOTDIR);

    switch (name[0]) {
    case UDPCTL_CHECKSUM:
        return (sysctl_int(oldp, oldlenp, newp, newlen, &udpcksum));
    default:
        return (ENOPROTOOPT);
    }
    /* NOTREACHED */
}
```

udp_usrreq.c

Figure 23.41 `udp_sysctl` function.

23.12 Implementation Refinements

UDP PCB Cache

In Section 22.12 we talked about some general features of PCB searching and how the code we've seen uses a linear search of the protocol's PCB list. We now tie this together with the one-behind cache used by UDP in Figure 23.24.

The problem with the one-behind cache occurs when the cached PCB contains wildcard values (for either the local address, foreign address, or foreign port): the cached value never matches any received datagram. One solution tested in [Partridge and Pink 1993] is to modify the cache to not compare wildcarded values. That is, instead of comparing the foreign address in the PCB with the source address in the datagram, compare these two values only if the foreign address in the PCB is not a wildcard.

There's a subtle problem with this approach [Partridge and Pink 1993]. Assume there are two sockets bound to local port 555. One has the remaining three elements wildcarded, while the other has connected to the foreign address 128.1.2.3 and the foreign port 1600. If we cache the first PCB and a datagram arrives from 128.1.2.3, port 1600, we can't ignore comparing the foreign addresses just because the cached value has a wildcarded foreign address. This is called *cache hiding*. The cached PCB has hidden another PCB that is a better match in this example.

To get around cache hiding requires more work when a new entry is added to or deleted from the cache. Those PCBs that hide other PCBs cannot be cached. This is not a problem, however, because the normal scenario is to have one socket per local port. The example we just gave with two sockets bound to local port 555, while possible (especially on a multihomed host), is rare.

The next enhancement tested in [Partridge and Pink 1993] is to also remember the PCB of the last datagram sent. This is motivated by [Mogul 1991], who shows that half of all datagrams received are replies to the last datagram that was sent. Cache hiding is a problem here also, so PCBs that would hide other PCBs are not cached.

The results of these two caches shown in [Partridge and Pink 1993] on a general-purpose system measured for around 100,000 received UDP datagrams show a 57% hit rate for the last-received PCB cache and a 30% hit rate for the last-sent PCB cache. The amount of CPU time spent in `udp_input` is more than halved, compared to the version with no caching.

These two caches still depend on a certain amount of locality: that with a high probability the UDP datagram that just arrived is either from the same peer as the last UDP datagram received or from the peer to whom the last datagram was sent. The latter is typical for request–response applications that send a datagram and wait for a reply. [McKenney and Dove 1992] show that some applications, such as data entry into an online transaction processing (OLTP) system, don't yield the high cache hit rates that [Partridge and Pink 1993] observed. As we mentioned in Section 22.12, placing the PCBs onto hash chains provided an order of magnitude improvement over the last-received and last-sent caches for a system with thousands of OLTP connections.

UDP Checksum

The next area for improving the implementation is to combine the copying of data between the process and the kernel with the calculation of the checksum. In Net/3, each byte of data is processed twice during an output operation: once when copied from the process into an mbuf (the function `uiomove`, which is called by `sosend`), and again when the UDP checksum is calculated (by the function `in_cksum`, which is called by `udp_output`). This happens on input as well as output.

[Partridge and Pink 1993] modified the UDP output processing from what we showed in Figure 23.14 so that a UDP-specific function named `udp_sosend` is called instead of `sosend`. This new function calculates the checksum of the UDP header and the pseudo-header in-line (instead of calling the general-purpose function `in_cksum`) and then copies the data from the process into an mbuf chain using a special function named `in_uiomove` (instead of the general-purpose `uiomove`). This new function copies the data *and* updates the checksum. The amount of time spent copying the data and calculating the checksum is reduced with this technique by about 40 to 45%.

On the receive side the scenario is different. UDP calculates the checksum of the UDP header and the pseudo-header, removes the UDP header, and queues the data for the appropriate socket. When the application reads the data, a special version of `soreceive` (called `udp_soreceive`) completes the calculation of the checksum while copying the data into the user's buffer. If the checksum is in error, however, the error is not detected until the entire datagram has been copied into the user's buffer. In the normal case of a blocking socket, `udp_soreceive` just waits for the next datagram to arrive. But if the socket is nonblocking, the error `EWOULDBLOCK` must be returned if another datagram is not ready to be passed to the process. This implies two changes in the socket interface for a nonblocking read from a UDP socket:

1. The `select` function can indicate that a nonblocking UDP socket is readable, yet the error `EWOULDBLOCK` is unexpectedly returned by one of the read functions if the checksum fails.
2. Since a checksum error is detected after the datagram has been copied into the user's buffer, the application's buffer is changed even though no data is returned by the read.

Even with a blocking socket, if the datagram with the checksum error contains 100 bytes of data and the next datagram without an error contains 40 bytes of data, `recvfrom` returns a length of 40, but the 60 bytes that follow in the user's buffer have also been modified.

[Partridge and Pink 1993] compare the timings for a copy versus a copy-with-checksum for six different computers. They show that the checksum is calculated for free during the copy operation on many architectures. This occurs when memory access speeds and CPU processing speeds are mismatched, as is true for many current RISC processors.

23.13 Summary

UDP is a simple, connectionless protocol, which is why we cover it before looking at TCP. UDP output is simple: IP and UDP headers are prepended to the user's data, as much of the header is filled in as possible, and the result is passed to `ip_output`. The only complication is calculating the UDP checksum, which involves prepending a pseudo-header just for the checksum computation. We'll encounter a similar pseudo-header for the calculation of the TCP checksum in Chapter 26.

When `udp_input` receives a datagram, it first performs a general validation (the length and checksum); the processing then differs depending on whether the destination IP address is a unicast address or a broadcast or multicast address. A unicast datagram is delivered to at most one process, but a broadcast or multicast datagram can be delivered to multiple processes. A one-behind cache is maintained for unicast datagrams, which maintains a pointer to the last Internet PCB for which a UDP datagram was received. We saw, however, that because of the prevalence of wildcard addressing with UDP applications, this cache is practically useless.

The `udp_ctlinput` function is called to handle received ICMP messages, and the `udp_usrreq` function handles the `PRU_`*xxx* requests from the socket layer.

Exercises

23.1 List the five types of mbuf chains that `udp_output` passes to `ip_output`. (*Hint*: look at `sosend`.)

23.2 What happens to the answer for the previous exercise when the process specifies IP options for the outgoing datagram?

23.3 Does a UDP client need to call `bind`? Why or why not?

23.4 What happens to the processor priority level in `udp_output` if the socket is unconnected and the call to `M_PREPEND` in Figure 23.15 fails?

23.5 `udp_output` does not check for a destination port of 0. Is it possible to send a UDP datagram with a destination port of 0?

23.6 Assuming the `IP_RECVDSTADDR` socket option worked when a datagram was sent to a broadcast address, how can you then determine if this address is a broadcast address?

23.7 Who releases the mbuf that `udp_saveopt` (Figure 23.28) allocates?

23.8 How can a process disconnect a connected UDP socket? That is, the process calls `connect` and exchanges datagrams with that peer, and then the process wants to disconnect the socket, allowing it to call `sendto` and send a datagram to some other host.

23.9 In our discussion of Figure 22.25 we noted that a UDP application that calls `connect` with a foreign IP address of 255.255.255.255 actually sends datagrams out the primary interface with a destination IP address corresponding to the broadcast address of that interface. What happens if a UDP application uses an unconnected socket instead, calling `sendto` with a destination address of 255.255.255.255?

23.10 After discussing the problem with Figure 23.27, we mentioned that this problem would not exist if the server used the destination IP address from the request as the source IP address of the reply. Explain how the server could do this.

23.11 Implement changes to allow a process to perform path MTU discovery using UDP: the process must be able to set the "don't fragment" bit in the resulting IP datagram and be told if the corresponding ICMP destination unreachable error is received.

23.12 Does the variable `udp_in` need to be global?

23.13 Modify `udp_input` to save the IP options and make them available to the receiver with the `IP_RECVOPTS` socket option.

23.14 Fix the one-behind cache in Figure 23.24.

23.15 Fix `udp_input` to implement the `IP_RECVOPTS` and `IP_RETOPTS` socket options.

23.16 Fix `udp_input` so that the `IP_RECVDSTADDR` socket option works for datagrams sent to a broadcast or multicast address.

24

TCP: Transmission Control Protocol

24.1 Introduction

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

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

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

24.2 Code Introduction

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

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

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

Figure 24.1 Files discussed in the TCP chapters.

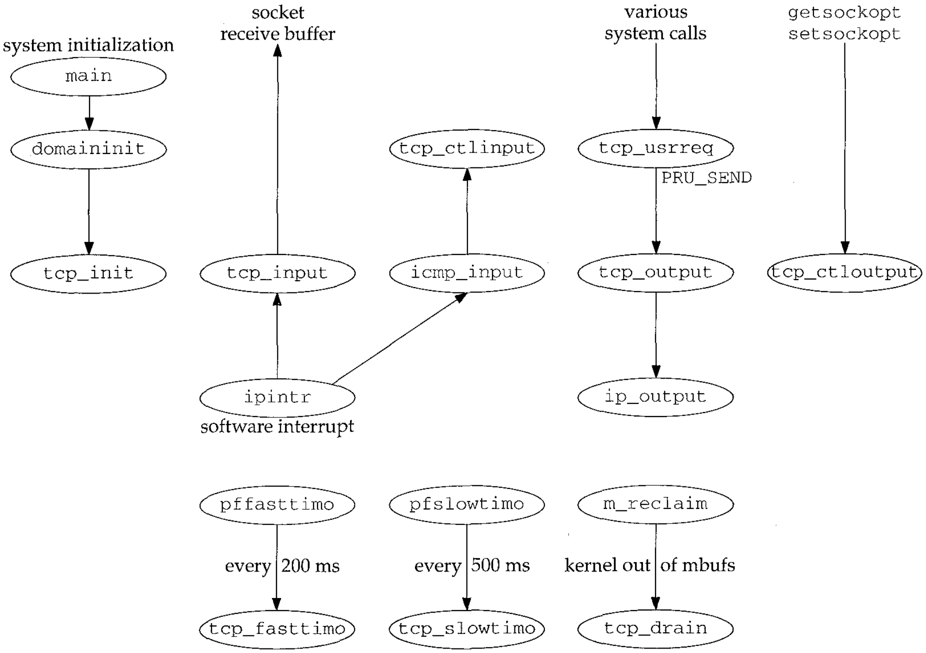

Figure 24.2 Relationship of TCP functions to rest of the kernel.

Global Variables

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

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

Figure 24.3 Global variables introduced in the following chapters.

Statistics

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

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

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

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

Figure 24.4 TCP statistics maintained in the `tcpstat` structure.

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

Figure 24.5 Sample TCP statistics.

SNMP Variables

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

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

SNMP variable	`tcpstat` members or constant	Description
`tcpRtoAlgorithm`	4	algorithm used to calculate retransmission timeout value: 1 = none of the following, 2 = a constant RTO, 3 = MIL-STD-1778 Appendix B, 4 = Van Jacobson's algorithm.
`tcpRtoMin`	1000	minimum retransmission timeout value, in milliseconds
`tcpRtoMax`	64000	maximum retransmission timeout value, in milliseconds
`tcpMaxConn`	-1	maximum #TCP connections (-1 if dynamic)
`tcpActiveOpens`	`tcps_connattempt`	#transitions from CLOSED to SYN_SENT states
`tcpPassiveOpens`	`tcps_accepts`	#transitions from LISTEN to SYN_RCVD states
`tcpAttemptFails`	`tcps_conndrops`	#transitions from SYN_SENT or SYN_RCVD to CLOSED, plus #transitions from SYN_RCVD to LISTEN
`tcpEstabResets`	`tcps_drops`	#transitions from ESTABLISHED or CLOSE_WAIT states to CLOSED
`tcpCurrEstab`	(see text)	#connections currently in ESTABLISHED or CLOSE_WAIT states
`tcpInSegs`	`tcps_rcvtotal`	total #segments received
`tcpOutSegs`	`tcps_sndtotal -` `tcps_sndrexmitpack`	total #segments sent, excluding those containing only retransmitted bytes
`tcpRetransSegs`	`tcps_sndrexmitpack`	total #retransmitted segments

`tcpInErrs`	`tcps_rcvbadsum +`
	`tcps_rcvbadoff +`
	`tcps_rcvshort`
		total #segments received with an error

`tcpOutRsts`	(not implemented)	total #segments sent with RST flag set

Figure 24.6 Simple SNMP variables in `tcp` group.

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

Figure 24.7 Variables in TCP listener table: `tcpTable`.

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

24.3 TCP `protosw` Structure

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

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

Figure 24.8 The TCP `protosw` structure.

24.4 TCP Header

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

tcp.h

```
struct tcphdr {
    u_short th_sport;           /* source port */
    u_short th_dport;           /* destination port */
    tcp_seq th_seq;             /* sequence number */
    tcp_seq th_ack;             /* acknowledgement number */
#if BYTE_ORDER == LITTLE_ENDIAN
    u_char  th_x2:4,            /* (unused) */
            th_off:4;           /* data offset */
#endif
#if BYTE_ORDER == BIG_ENDIAN
    u_char  th_off:4,           /* data offset */
            th_x2:4;            /* (unused) */
#endif
    u_char  th_flags;           /* ACK, FIN, PUSH, RST, SYN, URG */
    u_short th_win;             /* advertised window */
    u_short th_sum;             /* checksum */
    u_short th_urp;             /* urgent offset */
};
```

tcp.h

Figure 24.9 `tcphdr` structure.

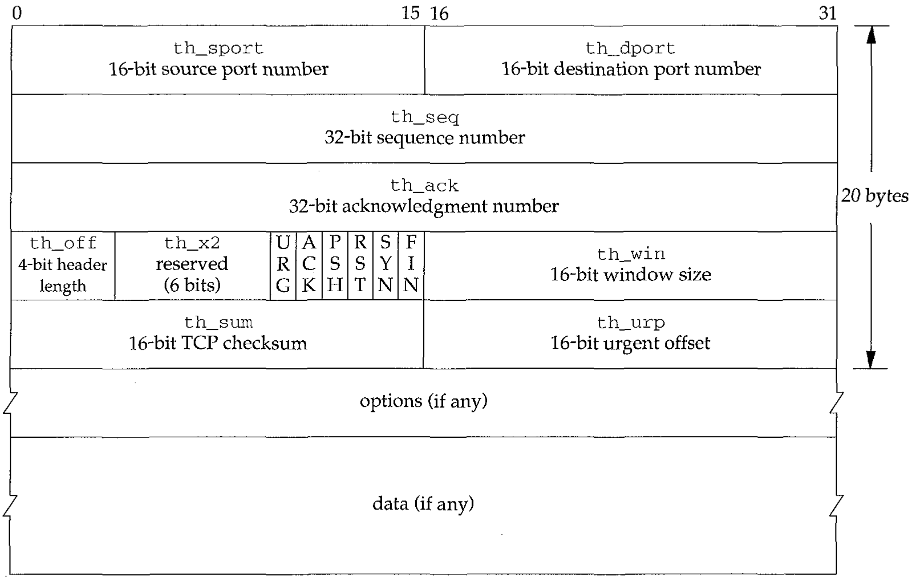

Figure 24.10 TCP header and optional data.

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

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

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

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

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

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

Figure 24.11 `th_flags` values.

tcpip.h

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

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

tcpip.h

Figure 24.12 `tcpiphdr` structure: combined IP/TCP header.

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

24.5 TCP Control Block

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

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

tcp_var.h

```
struct tcpcb {
    struct tcpiphdr *seg_next;  /* reassembly queue of received segments */
    struct tcpiphdr *seg_prev;  /* reassembly queue of received segments */
    short   t_state;            /* connection state (Figure 24.16) */
    short   t_timer[TCPT_NTIMERS];  /* tcp timers (Chapter 25) */
    short   t_rxtshift;         /* log(2) of rexmt exp. backoff */
    short   t_rxtcur;           /* current retransmission timeout (#ticks) */
    short   t_dupacks;          /* #consecutive duplicate ACKs received */
    u_short t_maxseg;           /* maximum segment size to send */
    char    t_force;            /* 1 if forcing out a byte (persist/OOB) */
    u_short t_flags;            /* (Figure 24.14) */
    struct tcpiphdr *t_template;    /* skeletal packet for transmit */
    struct inpcb *t_inpcb;      /* back pointer to internet PCB */
/*
 * The following fields are used as in the protocol specification.
 * See RFC783, Dec. 1981, page 21.
 */
/* send sequence variables */
    tcp_seq snd_una;            /* send unacknowledged */
    tcp_seq snd_nxt;            /* send next */
    tcp_seq snd_up;             /* send urgent pointer */
    tcp_seq snd_wl1;            /* window update seg seq number */
    tcp_seq snd_wl2;            /* window update seg ack number */
    tcp_seq iss;                /* initial send sequence number */
    u_long  snd_wnd;            /* send window */
/* receive sequence variables */
    u_long  rcv_wnd;            /* receive window */
    tcp_seq rcv_nxt;            /* receive next */
    tcp_seq rcv_up;             /* receive urgent pointer */
    tcp_seq irs;                /* initial receive sequence number */
/*
 * Additional variables for this implementation.
 */
/* receive variables */
    tcp_seq rcv_adv;            /* advertised window by other end */
/* retransmit variables */
    tcp_seq snd_max;            /* highest sequence number sent;
                                 * used to recognize retransmits */
/* congestion control (slow start, source quench, retransmit after loss) */
    u_long  snd_cwnd;           /* congestion-controlled window */
    u_long  snd_ssthresh;       /* snd_cwnd size threshhold for slow start
                                 * exponential to linear switch */
/*
 * transmit timing stuff.  See below for scale of srtt and rttvar.
 * "Variance" is actually smoothed difference.
 */
    short   t_idle;             /* inactivity time */
    short   t_rtt;              /* round-trip time */
    tcp_seq t_rtseq;            /* sequence number being timed */
    short   t_srtt;             /* smoothed round-trip time */
    short   t_rttvar;           /* variance in round-trip time */
    u_short t_rttmin;           /* minimum rtt allowed */
    u_long  max_sndwnd;         /* largest window peer has offered */
```

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

tcp_var.h

Figure 24.13 `tcpcb` structure: TCP control block.

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

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

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

Figure 24.14 `t_flags` values.

24.6 TCP State Transition Diagram

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

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

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

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

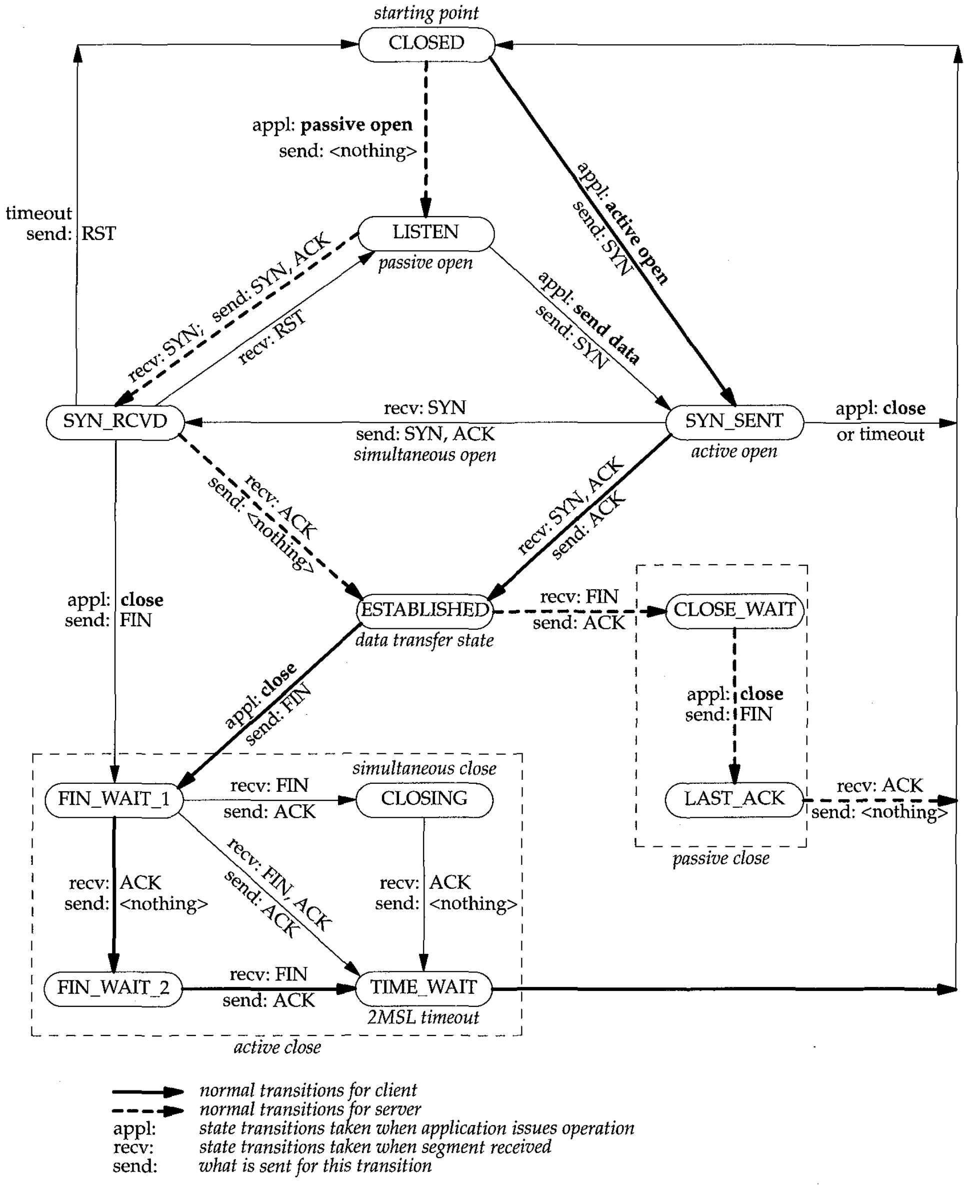

Figure 24.15 TCP state transition diagram.

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

Figure 24.16 `t_state` values.

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

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

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

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

Half-Close

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

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

24.7 TCP Sequence Numbers

Every byte of data exchanged across a TCP connection, along with the SYN and FIN flags, is assigned a 32-bit *sequence number*. The sequence number field in the TCP header (Figure 24.10) contains the sequence number of the first byte of data in the segment. The *acknowledgment number* field in the TCP header contains the next sequence number that the sender of the ACK expects to receive, which acknowledges all data bytes through the acknowledgment number minus 1. In other words, the acknowledgment number is the *next* sequence number expected by the sender of the ACK. The acknowledgment number is valid only if the ACK flag is set in the header. We'll see that TCP always sets the ACK flag except for the first SYN sent by an active open (the SYN_SENT state; see `tcp_outflags[2]` in Figure 24.16) and in some RST segments.

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

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

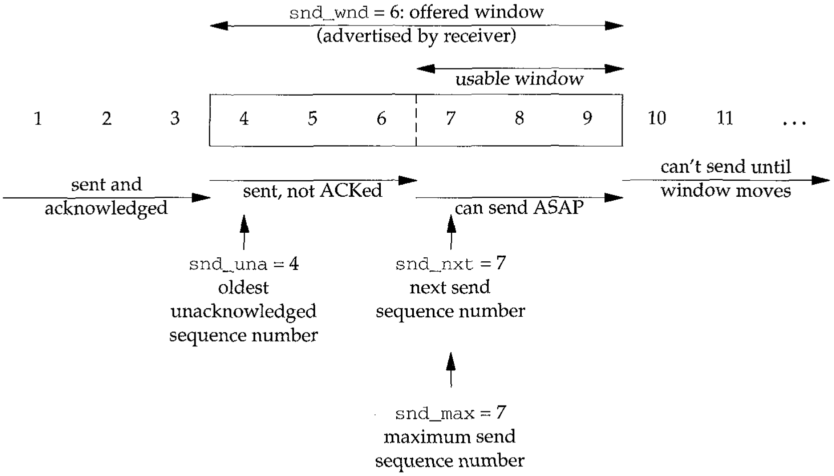

Figure 24.17 Example of send sequence space.

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

`snd_una` < acknowledgment field <= `snd_max`

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

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

```
snd_nxt < snd_max
```

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

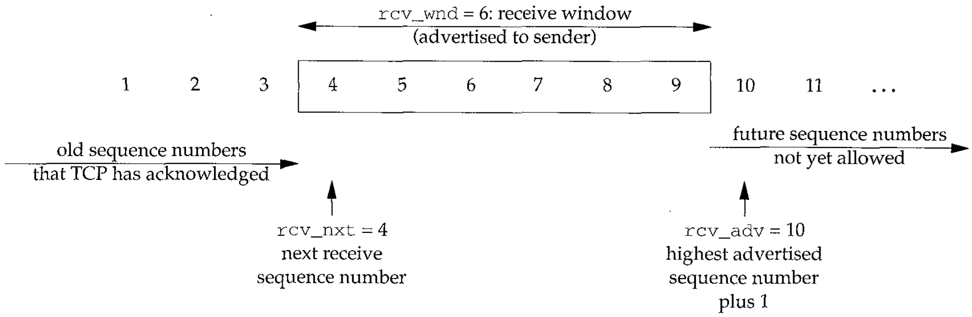

Figure 24.18 Example of receive sequence space.

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

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

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

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

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

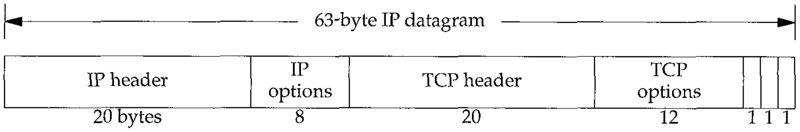

Figure 24.19 TCP segment transmitted as an IP datagram.

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

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

Figure 24.20 Values of variables corresponding to Figure 24.19.

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

Modular Arithmetic with Sequence Numbers

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

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

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

```
typedef  u_long  tcp_seq;
```

The four macros shown in Figure 24.21 compare sequence numbers.

tcp_seq.h

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

tcp_seq.h

Figure 24.21 Macros for TCP sequence number comparison.

Example—Sequence Number Comparisons

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

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

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

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

Figure 24.22 Example using 3-bit sequence numbers.

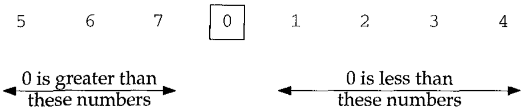

Figure 24.23 TCP sequence number comparisons for 3-bit sequence numbers.

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

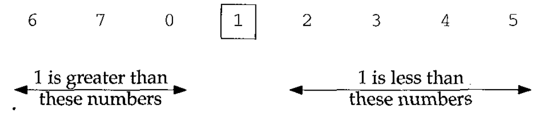

Figure 24.24 TCP sequence number comparisons for 3-bit sequence numbers.

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

Figure 24.25 Another way to visualize Figures 24.23 and 24.24.

With regard to TCP, these sequence number comparisons determine whether a given sequence number is in the future or in the past (a retransmission). For example, using Figure 24.24, if TCP is expecting sequence number 1 and sequence number 6 arrives, since 6 is less than 1 using the sequence number arithmetic we showed, the data byte is considered a retransmission of a previously received data byte and is discarded. But if sequence number 5 is received, since it is greater than 1 it is considered a future data byte and is saved by TCP, awaiting the arrival of the missing bytes 2, 3, and 4 (assuming byte 5 is within the receive window).

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

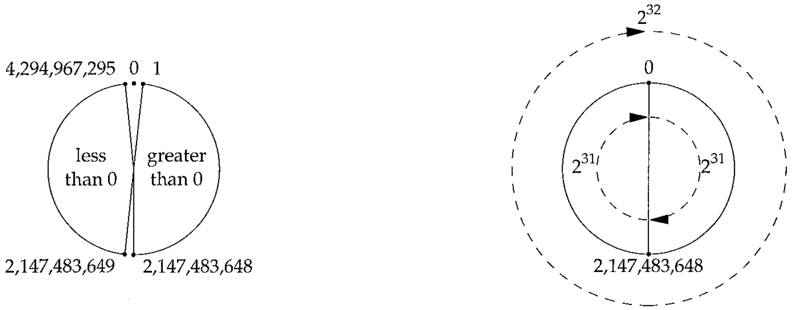

Figure 24.26 Comparisons against 0, using 32-bit sequence numbers.

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

24.8 `tcp_init` Function

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

tcp_subr.c

```
43 void
44 tcp_init()
45 {

46     tcp_iss = 1;                /* wrong */
47     tcb.inp_next = tcb.inp_prev = &tcb;

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

tcp_subr.c

Figure 24.27 `tcp_init` function.

Set initial send sequence number (ISS)

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

Initialize linked list of TCP Internet PCBs

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

Calculate maximum protocol header length

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

MSL and Quiet Time Concept

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

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

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

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

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

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

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

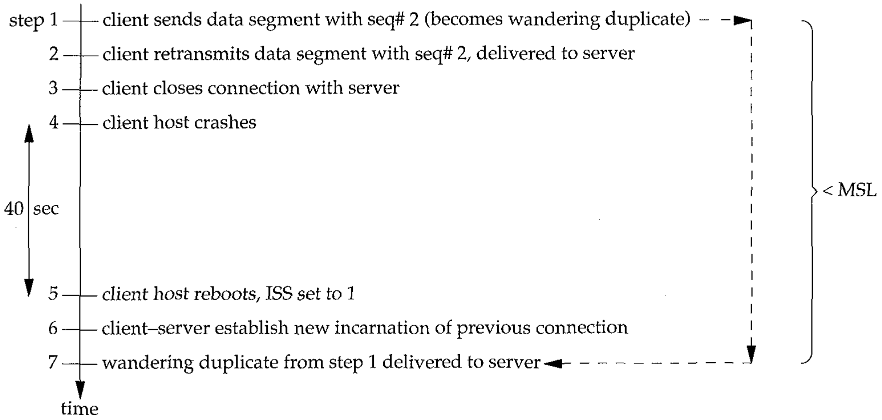

Figure 24.28 Example of old segment delivered to new incarnation of a connection.

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

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

24.9 Summary

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

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

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

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

Exercises

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

24.2 Is the kernel panic in `tcp_init` reasonable?

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

25

TCP Timers

25.1 Introduction

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

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

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

4. A *persist* timer is set when the other end of a connection advertises a window of 0, stopping TCP from sending data. Since window advertisements from the other end are not sent reliably (that is, ACKs are not acknowledged, only data is acknowledged), there's a chance that a future window update, allowing TCP to send some data, can be lost. Therefore, if TCP has data to send and the other end advertises a window of 0, the persist timer is set and when it expires, 1 byte of data is sent to see if the window has opened. Like the retransmission timer, the persist timer value is calculated dynamically, based on the round-trip time. The value of this is bounded by TCP to be between 5 and 60 seconds.
5. A *keepalive* timer can be set by the process using the `SO_KEEPALIVE` socket option. If the connection is idle for 2 hours, the keepalive timer expires and a special segment is sent to the other end, forcing it to respond. If the expected response is received, TCP knows that the other host is still up, and TCP won't probe it again until the connection is idle for another 2 hours. Other responses to the keepalive probe tell TCP that the other host has crashed and rebooted. If no response is received to a fixed number of keepalive probes, TCP assumes that the other end has crashed, although it can't distinguish between the other end being down (i.e., it crashed and has not yet rebooted) and a temporary lack of connectivity to the other end (i.e., an intermediate router or phone line is down).
6. A *FIN_WAIT_2* timer. When a connection moves from the FIN_WAIT_1 state to the FIN_WAIT_2 state (Figure 24.15) *and* the connection cannot receive any more data (implying the process called `close`, instead of taking advantage of TCP's half-close with `shutdown`), this timer is set to 10 minutes. When this timer expires it is reset to 75 seconds, and when it expires the second time the connection is dropped. The purpose of this timer is to avoid leaving a connection in the FIN_WAIT_2 state forever, if the other end never sends a FIN. (We don't show this timeout in Figure 24.15.)
7. A *TIME_WAIT* timer, often called the *2MSL* timer. The term *2MSL* means twice the MSL, the maximum segment lifetime defined in Section 24.8. It is set when a connection enters the TIME_WAIT state (Figure 24.15), that is, when the connection is actively closed. Section 18.6 of Volume 1 describes the reasoning for the 2MSL wait state in detail. The timer is set to 1 minute (Net/3 uses an MSL of 30 seconds) when the connection enters the TIME_WAIT state and when it expires, the TCP control block and Internet PCB are deleted, allowing that socket pair to be reused.

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

25.2 Code Introduction

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

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

Figure 25.1 Indexes into the `t_timer` array.

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

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

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

Figure 25.2 Implementation of the seven TCP timers.

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

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

Figure 25.3 Fundamental timer values for the implementation.

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

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

Figure 25.4 Timer constants.

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

tcp_timer.h

```
#define TCPT_RANGESET(tv, value, tvmin, tvmax) { \
    (tv) = (value); \
    if ((tv) < (tvmin)) \
        (tv) = (tvmin); \
    else if ((tv) > (tvmax)) \
        (tv) = (tvmax); \
}
```

tcp_timer.h

Figure 25.5 `TCPT_RANGESET` macro.

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

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

25.3 `tcp_canceltimers` Function

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

tcp_timer.c

```
void
tcp_canceltimers(tp)
struct tcpcb *tp;
{
    int     i;

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

tcp_timer.c

Figure 25.6 `tcp_canceltimers` function.

25.4 `tcp_fasttimo` Function

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

tcp_timer.c

```
void
tcp_fasttimo()
{
    struct inpcb *inp;
    struct tcpcb *tp;
    int     s = splnet();

    inp = tcb.inp_next;
    if (inp)
        for (; inp != &tcb; inp = inp->inp_next)
            if ((tp = (struct tcpcb *) inp->inp_ppcb) &&
                (tp->t_flags & TF_DELACK)) {
                tp->t_flags &= ~TF_DELACK;
                tp->t_flags |= TF_ACKNOW;
                tcpstat.tcps_delack++;
                (void) tcp_output(tp);
            }
    splx(s);
}
```

tcp_timer.c

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

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

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

25.5 `tcp_slowtimo` Function

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

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

Check each timer counter in all TCP control blocks

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

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

Check if TCP control block has been deleted

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

Count idle time

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

tcp_timer.c

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

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

tcp_timer.c

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

Increment RTT counter

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

Increment initial send sequence number

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

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

Increment RFC 1323 timestamp value

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

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

25.6 `tcp_timers` Function

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

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

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

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

tcp_timer.c

```
struct tcpcb *
tcp_timers(tp, timer)
struct tcpcb *tp;
int     timer;
{
    int     rexmt;

    switch (timer) {


                                        /* switch cases */


    }
    return (tp);
}
```

tcp_timer.c

Figure 25.9 `tcp_timers` function: general organization.

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

FIN_WAIT_2 and 2MSL Timers

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

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

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

tcp_timer.c

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

tcp_timer.c

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

2MSL timer

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

FIN_WAIT_2 timer

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

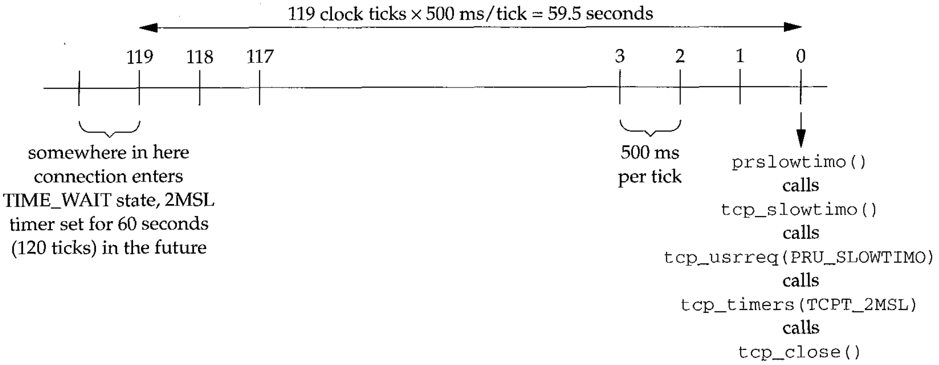

Figure 25.11 Setting and expiration of 2MSL timer in TIME_WAIT state.

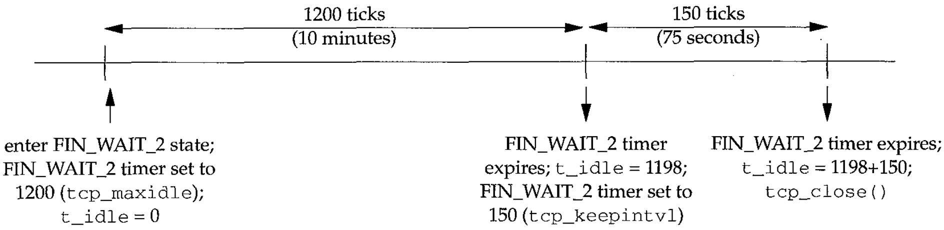

Figure 25.12 FIN_WAIT_2 timer to avoid infinite wait in FIN_WAIT_2 state.

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

The reason for the 75-second timeout after the first 10-minute timeout is as follows: a connection in the FIN_WAIT_2 state is not dropped until the connection has been idle for *more than* 10 minutes. There's no reason to test `t_idle` until at least 10 minutes have expired, but once this time has passed, the value of `t_idle` is checked every 75 seconds. Since a duplicate segment could be received, say a duplicate of the ACK that moved the connection from the FIN_WAIT_1 state to the FIN_WAIT_2 state, the 10-minute wait is restarted when the segment is received (since `t_idle` will be set to 0).

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

Persist Timer

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

tcp_timer.c

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

tcp_timer.c

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

Force window probe segment

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

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

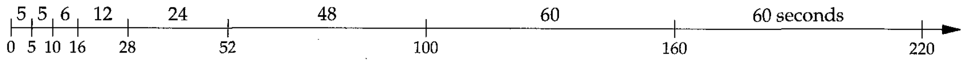

Figure 25.14 Time line of persist timer when probing a zero window.

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

Connection Establishment and Keepalive Timers

TCP's `TCPT_KEEP` counter implements two timers:

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

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

Connection-establishment timer expires after 75 seconds

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

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

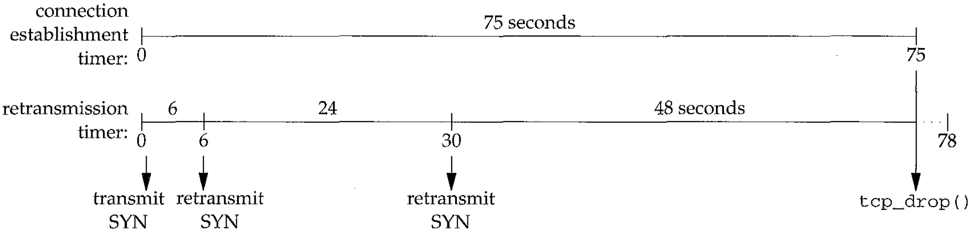

Figure 25.15 Connection-establishment timer and retransmission timer after SYN is sent.

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

tcp_timer.c

```
221         /*
222          * Keep-alive timer went off; send something
223          * or drop connection if idle for too long.
224          */
225     case TCPT_KEEP:
226         tcpstat.tcps_keeptimeo++;
227         if (tp->t_state < TCPS_ESTABLISHED)
228             goto dropit;        /* connection establishment timer */

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

tcp_timer.c

Figure 25.16 tcp_timers function: expiration of keepalive timer.

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

Keepalive timer expires after 2 hours of idle time

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

Drop connection when no response

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

Send a keepalive probe

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

Figure 25.17 summarizes this use of the keepalive timer.

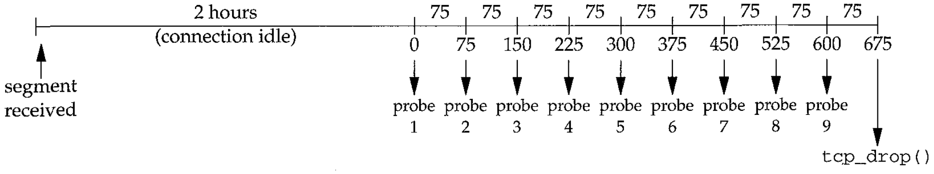

Figure 25.17 Summary of keepalive timer to detect unreachability of other end.

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

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

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

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

Reset keepalive timer

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

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

25.7 Retransmission Timer Calculations

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

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

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

$$delta = nticks - srtt$$

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

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

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

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

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

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

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

Figure 25.19 Control block variables for calculation of retransmission timer.

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

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

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

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

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

Figure 25.20 Multipliers and shifts for RTT estimators.

25.8 `tcp_newtcpcb` Function

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

tcp_subr.c

```
struct tcpcb *
tcp_newtcpcb(inp)
struct inpcb *inp;
{
    struct tcpcb *tp;

    tp = malloc(sizeof(*tp), M_PCB, M_NOWAIT);
    if (tp == NULL)
        return ((struct tcpcb *) 0);
    bzero((char *) tp, sizeof(struct tcpcb));
    tp->seg_next = tp->seg_prev = (struct tcpiphdr *) tp;
    tp->t_maxseg = tcp_mssdflt;
    tp->t_flags = tcp_do_rfc1323 ? (TF_REQ_SCALE | TF_REQ_TSTMP) : 0;
    tp->t_inpcb = inp;
    /*
     * Init srtt to TCPTV_SRTTBASE (0), so we can tell that we have no
     * rtt estimate.  Set rttvar so that srtt + 2 * rttvar gives
     * reasonable initial retransmit time.
     */
    tp->t_srtt = TCPTV_SRTTBASE;
    tp->t_rttvar = tcp_rttdflt * PR_SLOWHZ << 2;
    tp->t_rttmin = TCPTV_MIN;
    TCPT_RANGESET(tp->t_rxtcur,
                  ((TCPTV_SRTTBASE >> 2) + (TCPTV_SRTTDFLT << 2)) >> 1,
                  TCPTV_MIN, TCPTV_REXMTMAX);

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

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

tcp_subr.c

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

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

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

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

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

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

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

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

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

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

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

Substituting the two scaling relationships

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

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

we get

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

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

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

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

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

25.9 `tcp_setpersist` Function

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

tcp_output.c

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

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

tcp_output.c

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

Check retransmission timer not enabled

493–499 A check is made that the retransmission timer is not enabled when the persist timer is about to be set, since the two timers are mutually exclusive: if data is being sent, the other side must be advertising a nonzero window, but the persist timer is being set only if the advertised window is 0.

Calculate RTO

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

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

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

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

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

Apply exponential backoff

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

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

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

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

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

25.10 `tcp_xmit_timer` Function

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

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

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

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

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

```
if (ts_present)
    tcp_xmit_timer(tp, tcp_now - ts_ecr + 1);

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

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

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

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

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

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

Update smoothed estimators

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

tcp_input.c

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

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

tcp_input.c

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

1326–1327 The smoothed RTT estimator is updated using the equation

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

Since the gain g is $\frac{1}{8}$, this equation is

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

which is

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

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

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

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

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

which is

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

This final equation corresponds to the C code.

Initialize smoothed estimators on first RTT measurement

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

$$srtt = nticks + 1$$

or

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

which is

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

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

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

which is

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

or

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

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

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

substituting for *rttvar* gives us

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

which is indeed

$$RTO = 3 \times srtt$$

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

tcp_input.c

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

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

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

tcp_input.c

Figure 25.24 `tcp_xmit_timer` function: final part.

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

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

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

This is the now-familiar equation

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

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

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

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

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

Clear soft error variable

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

25.11 Retransmission Timeout: `tcp_timers` Function

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

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

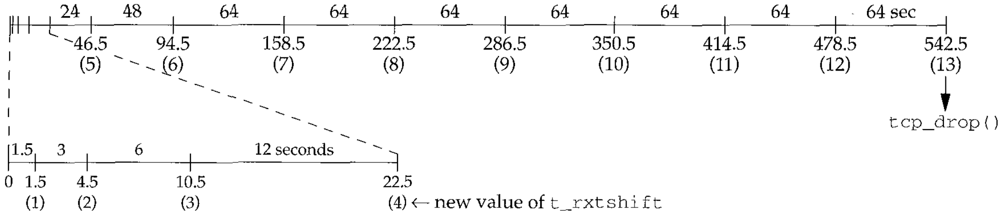

Figure 25.25 Summary of retransmission timer when sending data.

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

RFC 793 recommended that an open of a new connection, active or passive, allow a parameter specifying the total timeout period for data sent by TCP. This is the total amount of time TCP will try to send a given segment before giving up and terminating the connection. The recommended default was 5 minutes.

RFC 1122 requires that an application must be able to specify a parameter for a connection giving either the total number of retransmissions or the total timeout value for data sent by TCP. This parameter can be specified as "infinity," meaning TCP never gives up, allowing, perhaps, an interactive user the choice of when to give up.

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

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

tcp_timer.c

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

tcp_timer.c

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

Increment shift count

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

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

Drop connection

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

Calculate new RTO

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

Ask IP to find a new route

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

Clear estimators

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

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

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

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

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

Force retransmission of oldest unacknowledged data

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

Karn's algorithm

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

Slow Start and Congestion Avoidance

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

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

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

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

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

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

tcp_timer.c

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

tcp_timer.c

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

Accuracy

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

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

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

25.12 An RTT Example

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

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

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

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

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

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

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

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

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

Figure 25.28 Values of RTT variables and estimators during example.

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

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

25.13 Summary

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

TCP maintains the following seven timers for each connection:

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

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

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

Exercises

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

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

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

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

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

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

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

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

26

TCP Output

26.1 Introduction

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

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

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

26.2 `tcp_output` Overview

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

Is an ACK expected from the other end?

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

Go back to slow start

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

Send more than one segment

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

26.3 Determine if a Segment Should be Sent

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

tcp_output.c

```
int
tcp_output(tp)
struct tcpcb *tp;
{
    struct socket *so = tp->t_inpcb->inp_socket;
    long    len, win;
    int     off, flags, error;
    struct mbuf *m;
    struct tcpiphdr *ti;
    u_char  opt[MAX_TCPOPTLEN];
    unsigned optlen, hdrlen;
    int     idle, sendalot;

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

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


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

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

  send:


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


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

tcp_output.c

Figure 26.1 tcp_output function: overview.

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

tcp_output.c

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

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

tcp_output.c

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

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

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

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

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

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

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

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

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

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

tcp_output.c

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

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

tcp_output.c

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

Calculate amount of data to send

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

Check for window shrink

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

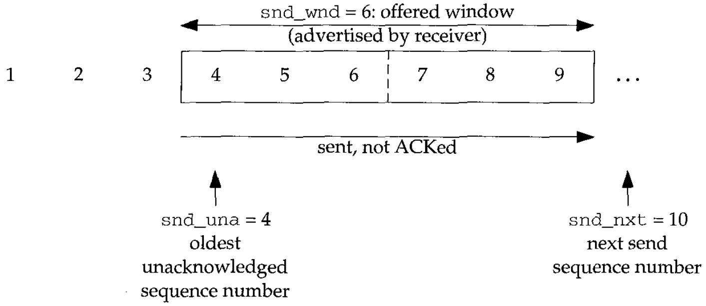

Figure 26.4 Send buffer after bytes 4 through 9 are sent.

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

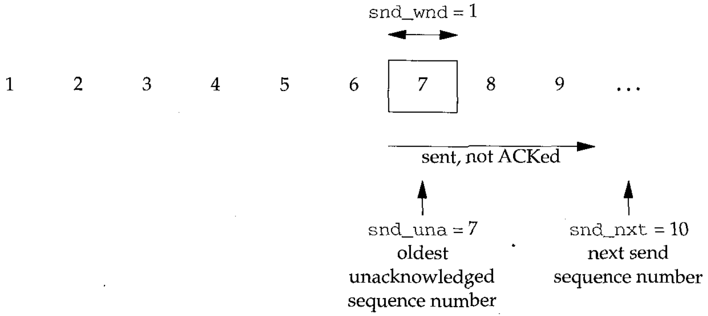

Figure 26.5 Send buffer after receiving acknowledgment of bytes 4 through 6.

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

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

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

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

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

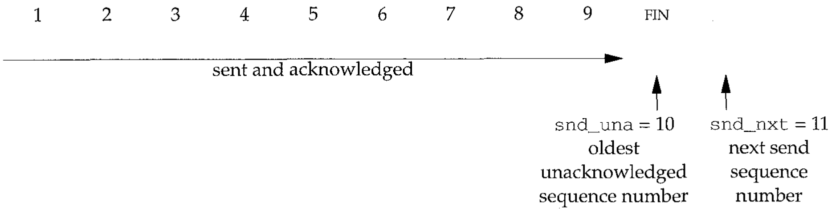

Figure 26.6 Bytes 1 through 9 have been sent and acknowledged, and then connection is closed.

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

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

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

Enter persist state

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

Send one segment at a time

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

Turn off FIN flag if send buffer not emptied

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

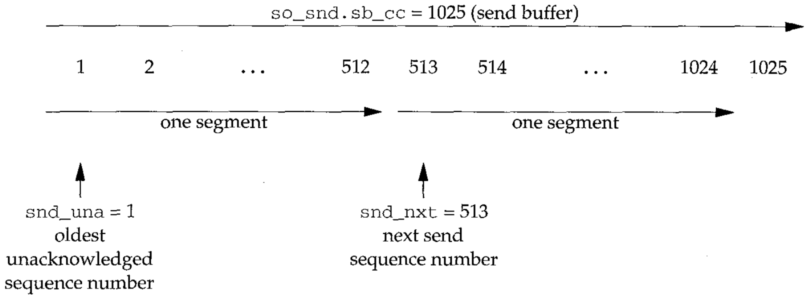

Figure 26.7 Example of send buffer not being emptied when FIN is set.

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

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

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

Calculate window advertisement

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

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

Sender silly window avoidance

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

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

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

tcp_output.c

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

tcp_output.c

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

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

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

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

154–168 The expression

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

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

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

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

tcp_output.c

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

169         if (adv >= (long) (2 * tp->t_maxseg))
170             goto send;
171         if (2 * adv >= (long) so->so_rcv.sb_hiwat)
172             goto send;
173     }
```

tcp_output.c

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

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

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

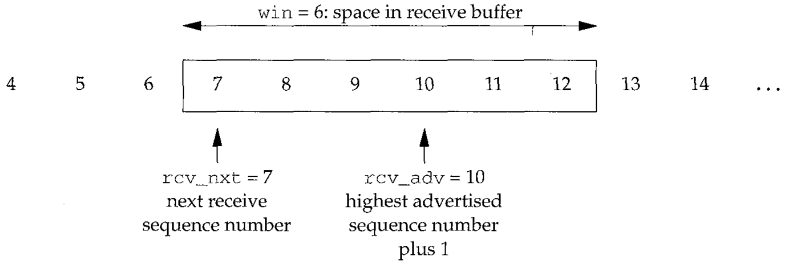

Figure 26.10 Transition from Figure 24.18 after bytes 4, 5, and 6 are received.

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

169–170 If the window has opened by two or more segments, a window update is sent. When data is received as full-sized segments, this code causes every other received segment to be acknowledged: TCP's ACK-every-other-segment property. (We show an example of this shortly.)

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

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

tcp_output.c

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

tcp_output.c

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

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

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

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

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

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

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

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

tcp_output.c

```
    /*
     * TCP window updates are not reliable, rather a polling protocol
     * using 'persist' packets is used to ensure receipt of window
     * updates.  The three 'states' for the output side are:
     *  idle                not doing retransmits or persists
     *  persisting          to move a small or zero window
     *  (re)transmitting    and thereby not persisting
     *
     * tp->t_timer[TCPT_PERSIST]
     *      is set when we are in persist state.
     * tp->t_force
     *      is set when we are called to send a persist packet.
     * tp->t_timer[TCPT_REXMT]
     *      is set when we are retransmitting
     * The output side is idle when both timers are zero.
     *
     * If send window is too small, there is data to transmit, and no
     * retransmit or persist is pending, then go to persist state.
     * If nothing happens soon, send when timer expires:
     * if window is nonzero, transmit what we can,
     * otherwise force out a byte.
     */
    if (so->so_snd.sb_cc && tp->t_timer[TCPT_REXMT] == 0 &&
        tp->t_timer[TCPT_PERSIST] == 0) {
        tp->t_rxtshift = 0;
        tcp_setpersist(tp);
    }
    /*
     * No reason to send a segment, just return.
     */
    return (0);
```

tcp_output.c

Figure 26.12 `tcp_output` function: enter persist state.

Example

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

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

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

Example

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

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

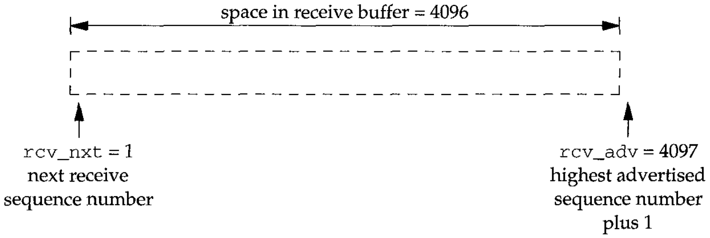

Figure 26.13 Receiver advertising a window of 4096.

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

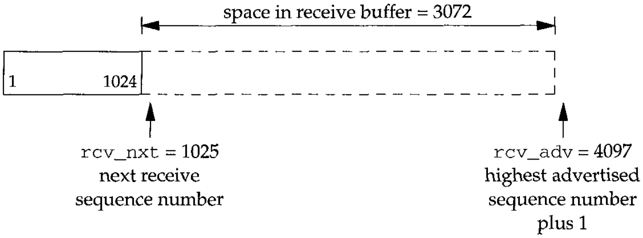

Figure 26.14 Transition from Figure 26.13 after bytes 1–1024 received.

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

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

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

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

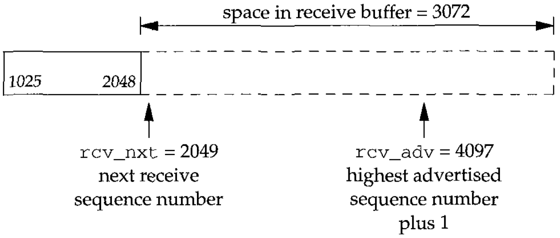

Figure 26.15 Transition from Figure 26.14 after bytes 1025–2048 received.

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

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

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

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

26.4 TCP Options

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

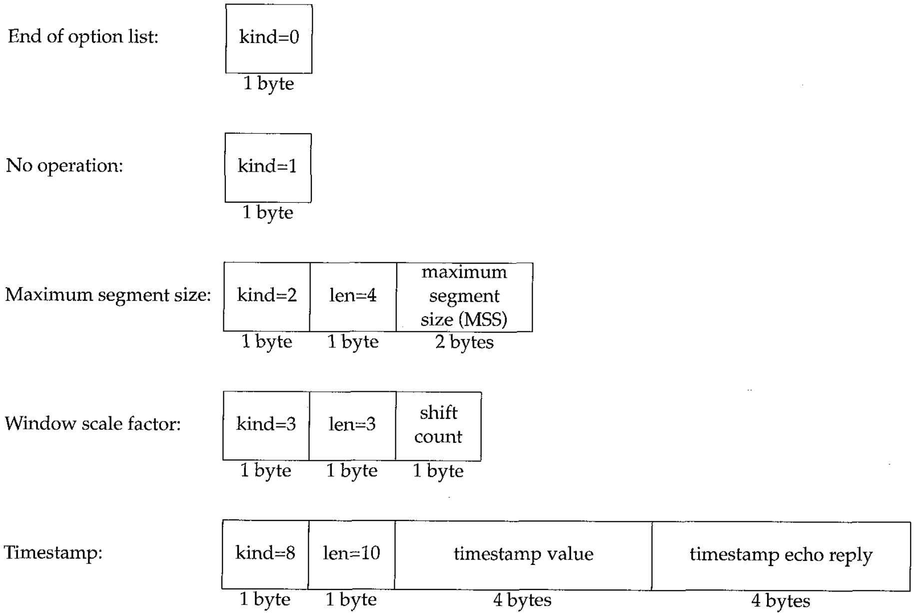

Figure 26.16 TCP options supported by Net/3.

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

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

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

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

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

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

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

26.5 Window Scale Option

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

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

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

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

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

26.6 Timestamp Option

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

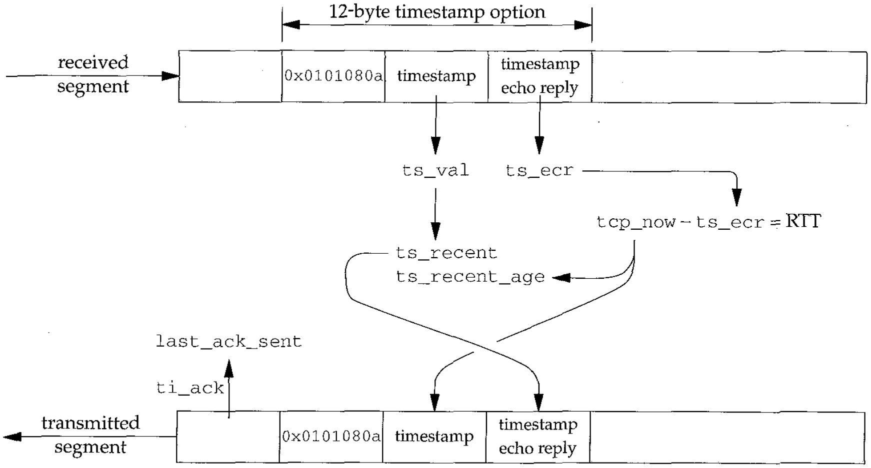

Figure 26.17 Summary of variables used with timestamp option.

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

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

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

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

In an outgoing segment, the first 4 bytes of the timestamp option are set to `0x0101080a`. This is the recommended value from Appendix A of RFC 1323. The 2 bytes of 1 are NOPs from Figure 26.16, followed by a *kind* of 8 and a *len* of 10, which identify the timestamp option. By placing two NOPs in front of the option, the two 32-bit timestamps in the option and the data that follows are aligned on 32-bit boundaries. Also, we show the received timestamp option in Figure 26.17 with the recommended 12-byte format (which Net/3 always generates), but the code that processes received options (Figure 28.10) does not require this format. The 10-byte format shown in Figure 26.16, without two preceding NOPs, is handled fine on input (but see Exercise 28.4).

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

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

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

Which Timestamp to Echo, RFC 1323 Algorithm

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

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

which is implemented in C as shown in Figure 26.18.

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

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

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

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

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

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

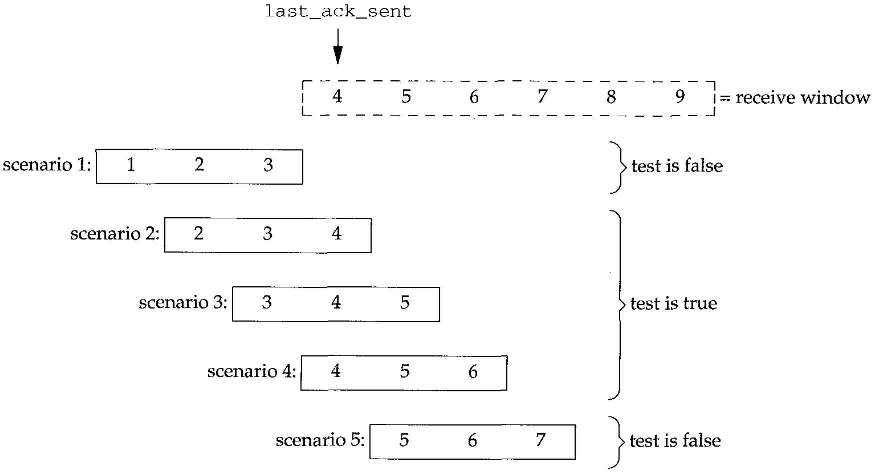

Figure 26.19 Example receive window and five different scenarios of received segment.

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

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

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

Which Timestamp to Echo, Corrected Algorithm

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

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

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

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

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

Timestamps and Delayed ACKs

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

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

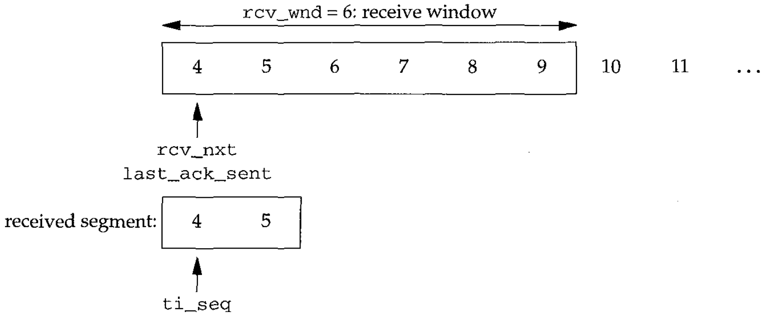

Figure 26.21 Receive sequence space when segment with bytes 4 and 5 arrives.

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

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

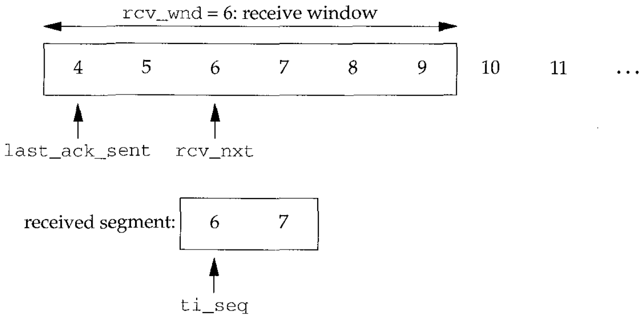

Figure 26.22 Receive sequence space when segment with bytes 6 and 7 arrives.

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

26.7 Send a Segment

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

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

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

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

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

tcp_output.c

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

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

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

tcp_output.c

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

Build MSS option

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

Should window scale option be sent?

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

Build window scale option

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

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

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

tcp_output.c

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

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

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

tcp_output.c

Figure 26.24 `tcp_output` function: finish sending options.

Should timestamp option be sent?

253–261 If the following three conditions are all true, a timestamp option is sent: (1) TCP is configured to request the timestamp option, (2) the segment being formed does not contain the RST flag, and (3) either this is an active open (i.e., `flags` specifies the SYN flag but not the ACK flag) or TCP has received a timestamp from the other end (`TF_RCVD_TSTMP`). Unlike the MSS and window scale options, a timestamp option can be sent with every segment once both ends agree to use the option.

Build timestamp option

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

Check if options have overflowed segment

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

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

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

Update statistics

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

Allocate an mbuf for IP and TCP headers

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

Copy data into mbuf

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

tcp_output.c

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

tcp_output.c

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

Set PSH flag

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