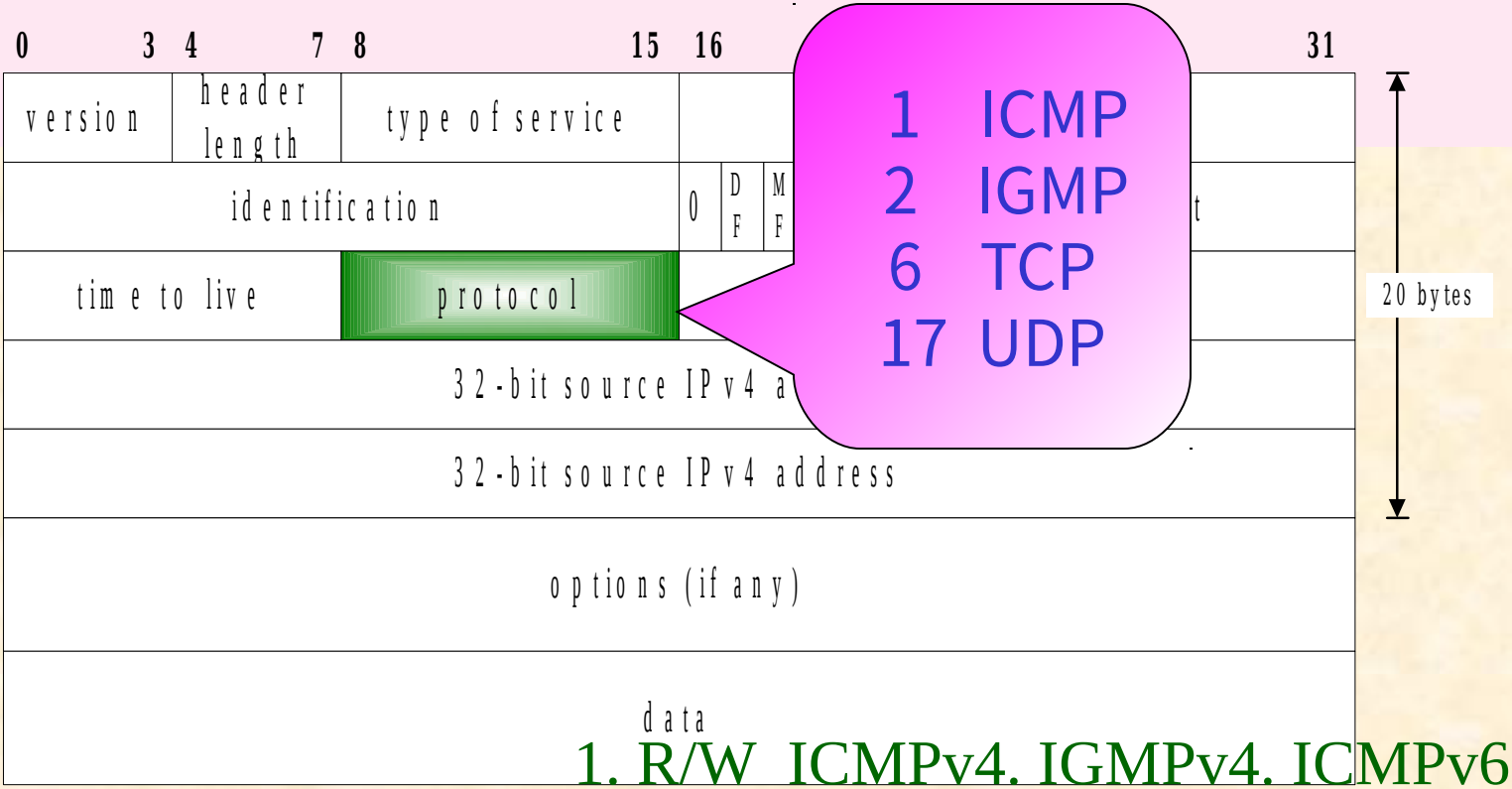
Raw Sockets

- Usage
- Creation. Output. Input
- Ping
- Traceroute

Raw Socket Usage



1. R/W ICMPv4. IGMPv4. ICMPv6
   ex. Ping
2. R/W other datagrams not processed by kernel
   ex. Gated => implement OSPF (protocol = 89)
3. Build one's own IPv4 Header
   ex. traceroute

Raw Socket Creation

Sockfd = socket (AF_INET, SOCK_RAW, *protocol*)

BIND

No port number concept

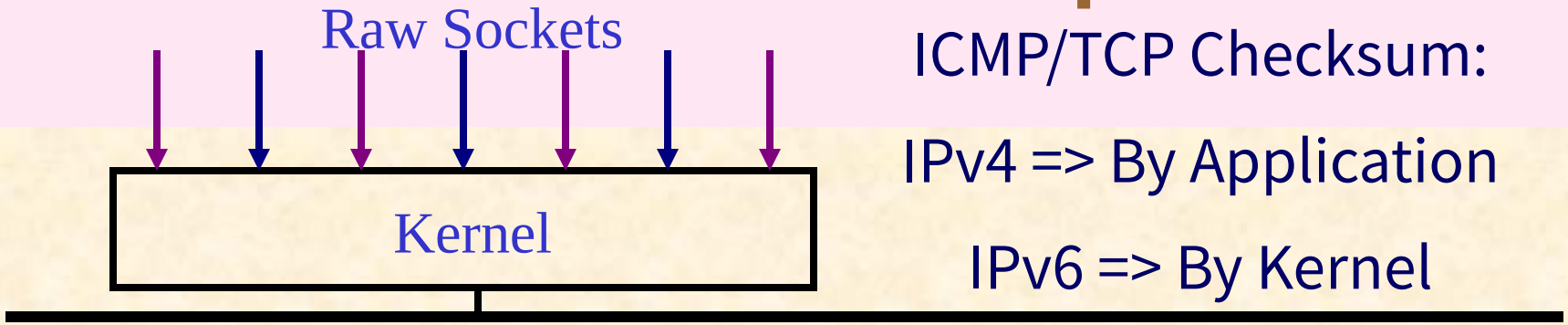
CONNECT

No port number concept

Set IP_HDRINCL socket option

If (setsockopt (sockfd, IPPROTO_IP, IP_HDRINCL, &on, sizeof (on))<0) error ;

Raw Socket Output



ICMP/TCP Checksum:

IPv4 => By Application

IPv6 => By Kernel

1. **Sendto / sendmsg** + destination IP

   connect => **write / writev / send**

2. Starting Address for the kernel to write

   If IP_HDRINCL is NOT set,

   Starting Addr. = **First byte following the IP header**

   If IP_HDRINCL is set,

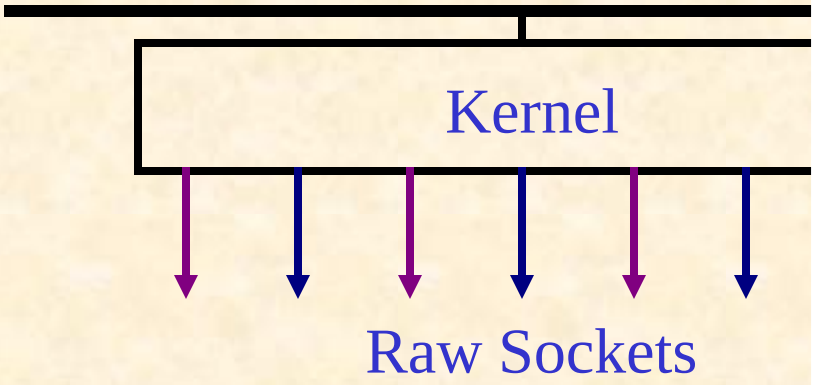
   Starting Addr. = **First byte of the IP header**

3. **Fragmentation** by kernel if the packet is too long

4. The kernel always calculates and stores IPv4 header checksum.

Raw Socket Input

1. UDP / TCP — Never pass to Raw Socket
2. Most ICMP — Kernel => Raw Socket
3. All IGMP — Kernel => Raw Socket
4. All IP datagram with unknown protocol field value — Kernel ~> Raw Socket
5. Fragment In — Reassemble then to Raw Socket



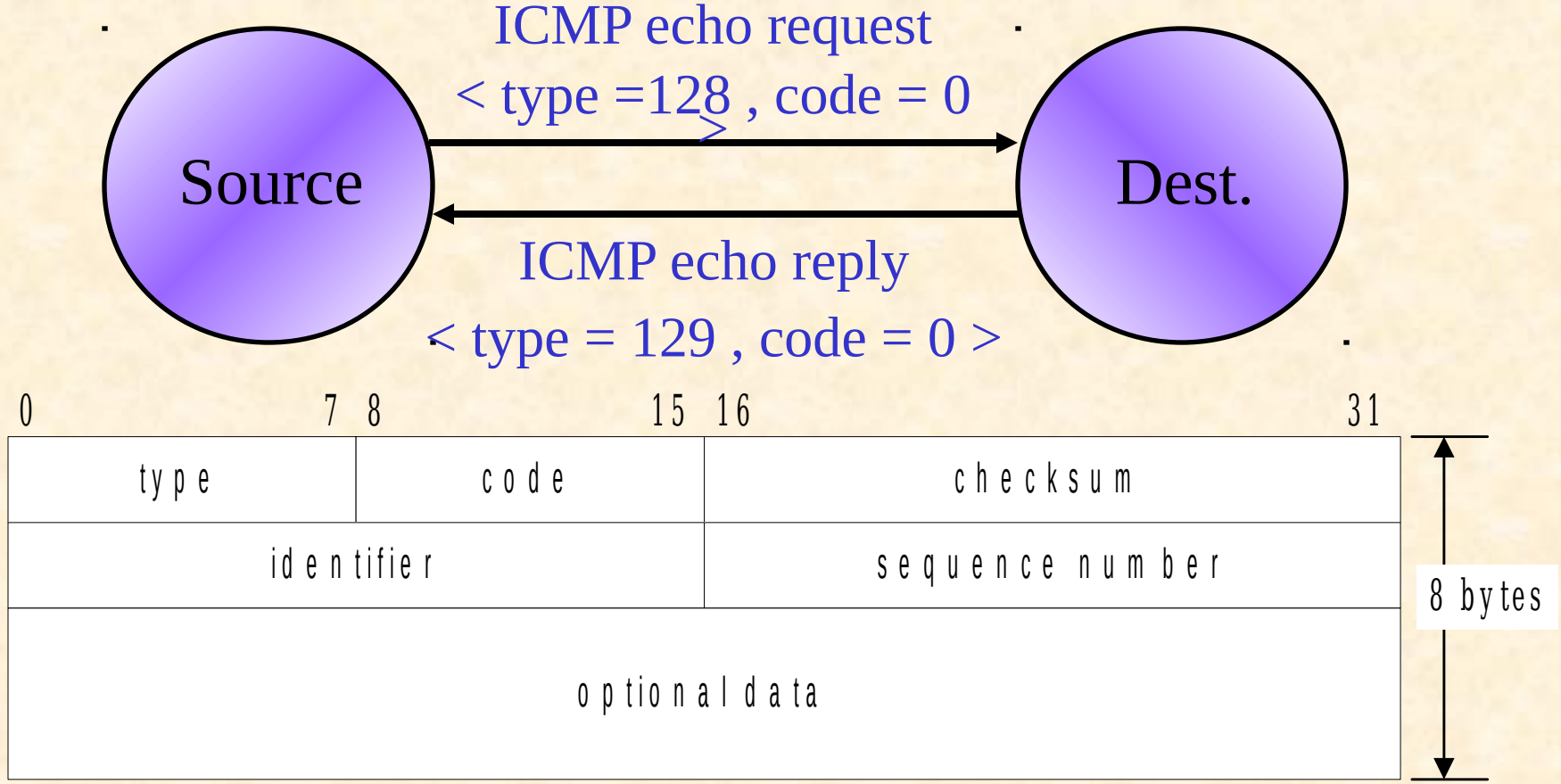
A copy of the IP datagram is delivered to each matching raw socket

- (1) protocol field must match
- (2) bind() addr. = dest. IP
- (3) connect() addr. = source IP

Ping Operation

```
Solaris # ping gemini.tuc.noao.edu
PING gemini.tuc.noao.edu (140.252.4.54):56 data bytes
64 bytes from 140.252.4.54: seq=0, ttl=248, rtt=37.542 ms
64 bytes from 140.252.4.54: seq=1, ttl=248, rtt=34.596 ms
64 bytes from 140.252.4.54: seq=2, ttl=248, rtt=29.204 ms
64 bytes from 140.252.4.54: seq=3, ttl=248, rtt=52.630 ms
```

Ping Operation



Format of ICMPv4 & ICMPv6 message

Overview of Ping

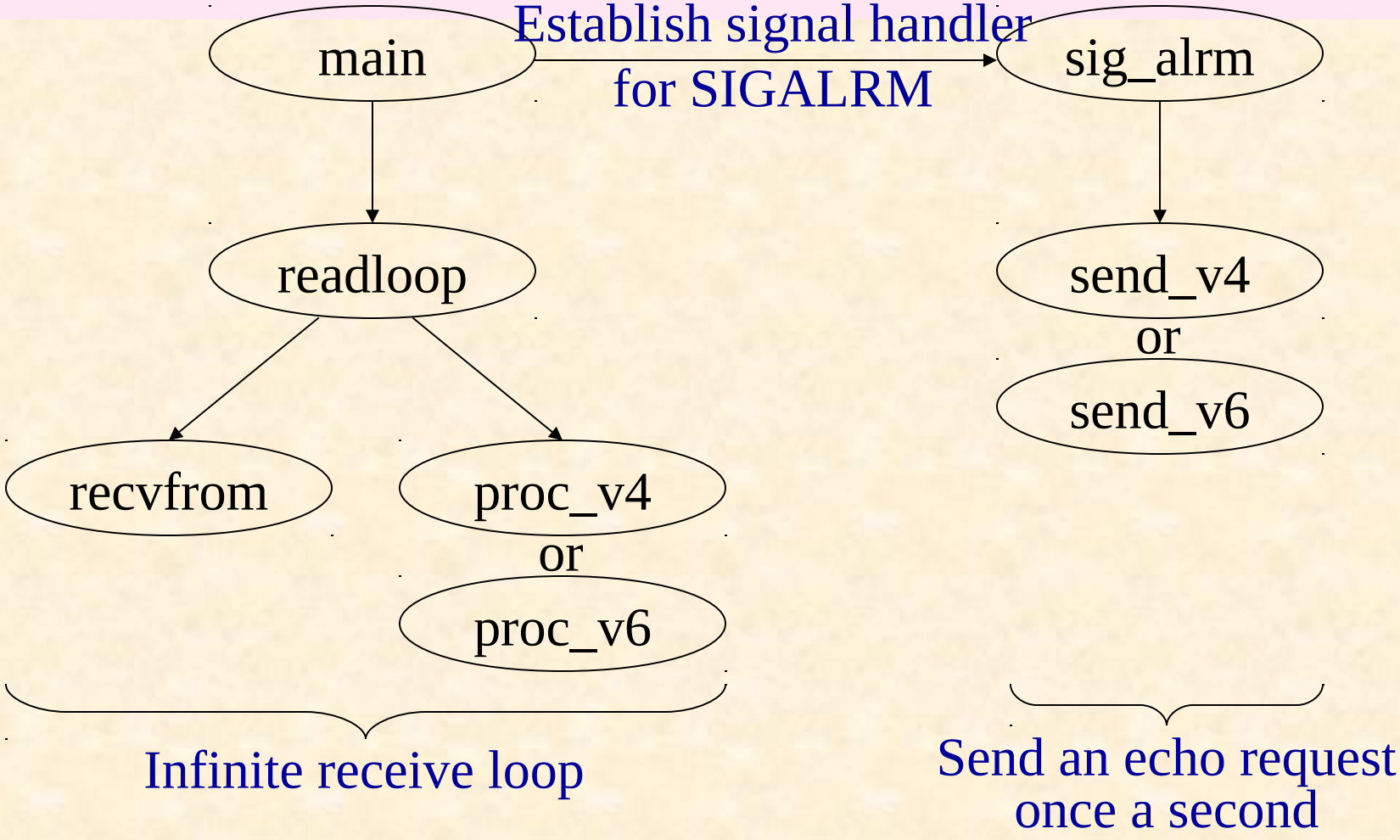


Figure 28.2 Sample output from our ping program.

```
freebsd % ping www.google.com
PING www.google.com (216.239.57.99): 56 data bytes
64 bytes from 216.239.57.99: seq=0, ttl=53, rtt=5.611 ms
64 bytes from 216.239.57.99: seq=1, ttl=53, rtt=5.562 ms
64 bytes from 216.239.57.99: seq=2, ttl=53, rtt=5.589 ms
64 bytes from 216.239.57.99: seq=3, ttl=53, rtt=5.910 ms

freebsd % ping www.kame.net
PING orange.kame.net (2001:200:0:4819:203:47ff:fea5:3085): 56 data bytes
64 bytes from 2001:200:0:4819:203:47ff:fea5:3085: seq=0, hlim=52, rtt=422.066 ms
64 bytes from 2001:200:0:4819:203:47ff:fea5:3085: seq=1, hlim=52, rtt=417.398 ms
```

Figure 28.4 `ping.h` **header.**

ping/ping.h

```
 1 #include    "unp.h"
 2 #include    <netinet/in_systm.h>
 3 #include    <netinet/ip.h>
 4 #include    <netinet/ip_icmp.h>

 5 #define BUFSIZE     1500

 6             /* globals */
 7 char    sendbuf[BUFSIZE];

 8 int     datalen;                /* #bytes of data following ICMP header */
 9 char   *host;
10 int     nsent;                  /* add 1 for each sendto() */
11 pid_t   pid;                    /* our PID */
12 int     sockfd;
13 int     verbose;

14             /* function prototypes */
15 void    init_v6(void);
16 void    proc_v4(char *, ssize_t, struct msghdr *, struct timeval *);
17 void    proc_v6(char *, ssize_t, struct msghdr *, struct timeval *);
18 void    send_v4(void);
```

```
19 void    send_v6(void);
20 void    readloop(void);
21 void    sig_alrm(int);
22 void    tv_sub(struct timeval *, struct timeval *);

23 struct proto {
24     void    (*fproc) (char *, ssize_t, struct msghdr *, struct timeval *);
25     void    (*fsend) (void);
26     void    (*finit) (void);
27     struct sockaddr *sasend;    /* sockaddr{} for send, from getaddrinfo */
28     struct sockaddr *sarecv;    /* sockaddr{} for receiving */
29     socklen_t salen;            /* length of sockaddr {}s */
30     int     icmpproto;          /* IPPROTO_xxx value for ICMP */
31 } *pr;

32 #ifdef IPV6

33 #include    <netinet/ip6.h>
34 #include    <netinet/icmp6.h>

35 #endif
```

Figure 28.5 `main` function.

ping/main.c

```
 1 #include     "ping.h"

 2 struct proto proto_v4 =
 3     { proc_v4, send_v4, NULL, NULL, NULL, 0, IPPROTO_ICMP };

 4 #ifdef  IPV6
 5 struct proto proto_v6 =
 6     { proc_v6, send_v6, NULL, NULL, 0, IPPROTO_ICMPV6 };
 7 #endif

 8 int     datalen = 56;   /* data that goes with ICMP echo request */

 9 int
10 main(int argc, char **argv)
11 {
12     int     c;
13     struct addrinfo *ai;
14     char   *h;
```

```
15      opterr = 0;                     /* don't want getopt() writing to stderr
16      while ( (c = getopt (argc, argv, "v") ) != -1) {
17          switch (c) {
18          case 'v':
19              verbose++;
20              break;

21           case '?':
22               err_quit ("unrecognized option: %c", c);
23           }
24      }

25      if  (optind != argc - 1)
26          err_quit ("usage: ping [ -v ] <hostname>");
27      host = argv [optind];

28      pid = getpid() & 0xffff;      /* ICMP ID field is 16 bits */
29      Signal(SIGALRM, sig_alrm);

30      ai = Host_serv (host, NULL, 0, 0);

31      h = Sock_ntop_host(ai->ai_addr, ai->ai_addrlen);
32      printf ("PING %s (%s): %d data bytes\n",
33              ai->ai_canonname ? ai->ai_canonname : h, h, datalen);
```

```
34          /* initialize  according to protocol */
35      if (ai->ai_family == AF_INET) {
36            pr = &proto_v4;
37  #ifdef   IPV6
38      } else if (ai->ai_family == AF_INET6) {
39          pr = &proto_v6;
40          if (IN6_IS_ADDR_V4MAPPED (&(((struct sockaddr_in6 *)
41                                  ai->ai_addr)->sin6_addr)))
42              err_quit ("cannot ping IPv4-mapped IPv6 address");
43  #endif
44      } else
45          err_quit ("unknown address family %d", ai->ai_family);

46      pr->sasend = ai->ai_addr;
47      pr->sacrecv = Calloc (1, ai->ai_addrlen);
48      pr->salen = ai->ai_addrlen);

49      readloop();

50      exit(0);
51  }
```

Figure 28.6 `readloop` function.

ping/readloop.c

```
 1 #include     "ping.h"

 2 void
 3 readloop(void)
 4 {
 5     int      size;
 6     char     recvbuf[BUFSIZE];
 7     char     controlbuf[BUFSIZE];
 8     struct msghdr msg;
 9     struct iovec iov;
10     ssize_t n;
11     struct timeval tval;

12     sockfd = Socket(pr->sasend->sa_family, SOCK_RAW, pr->icmpproto);
13     setuid(getuid());            /* don't need special permissions any more */
14     if (pr->finit)
15         (*pr->finit) ();

16     size = 60 * 1024;            /* OK if setsockopt fails */
17     setsockopt (sockfd, SOL_SOCKET, SO_RCVBUF, &size, sizeof (size));

18     sig_alrm (SIGALRM);          /* send first packet */
```

```
19      iov.iov_base = recvbuf;
20      iov.iov_len = sizeof (recvbuf);
21      msg.msg_name = pr->sarecv;
22      msg.msg_iov = &iov;
23      msg.msg_iovlen = 1;
24      msg.msg_control = controlbuf;
25      for ( ; ; ) {
26          msg.msg_namelen = pr->salen;
27          msg.msg_controllen = sizeof (controlbuf);
28          n = recvmsg (sockfd, &msg, 0);
29          if (n < o) {
30              if (errno == EINTR)
31                  continue;
32              else
33                  err_sys("recvmsg error");
34          }
35          Gettimeofday (&tval, NULL);
36          (*pr->fproc) (recvbuf, n, &msg, &tval);
37      }
38  }
```

Figure 28.7 `tv_sub` function: subtracts two `timeval` structures.

lib/tv_sub.c

```
 1 #include     "unp.h"

 2 void
 3 tv_sub (struct timeval *out, struct timeval *in)
 4 {
 5     if ((out->tv_usec -= in->tv_usec) < 0) {     /* out -= in */
 6         --out->tv_sec;
 7         out->tv_usec += 1000000;
 8     }
 9     out->tv_sec -= in->tv_sec;
10 }
```

Figure 28.9. Headers, pointers, and lengths in processing ICMPv4 reply.

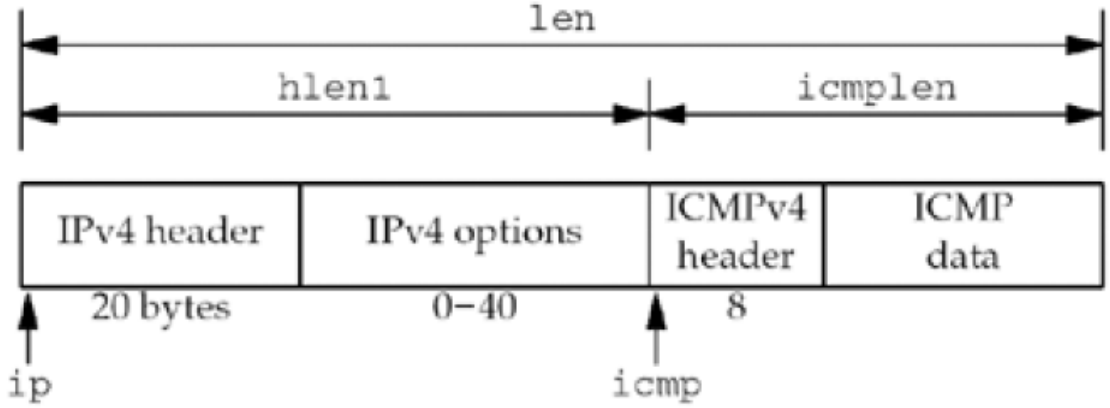


Figure 28.8 `proc_v4` **function: processes ICMPv4 message.**

ping/proc_v4.c

```
 1 #include     "ping.h"

 2 void
 3 proc_v4 (char *ptr, ssize_t len, struct msghdr *msg, struct timeval *tvrecv)
 4 {
 5     int     hlenl, icmplen;
 6     double  rtt;
 7     struct ip *ip;
 8     struct icmp *icmp;
 9     struct timeval *tvsend;

10     ip = (struct ip *) ptr;       /* start of IP header */
11     hlenl = ip->ip_hl << 2;       /* length of IP header */
12     if (ip->ip_p != IPPROTO_ICMP)
```

```
13          return;                     /* not ICMP */

14      icmp = (struct icmp *) (ptr + hlenl);   /* start of ICMP header */
15      if ( (icmplen = len - hlenl) < 8)
16          return;                     /* malformed packet */

17      if (icmp->icmp_type == ICMP_ECHOREPLY) {
18          if (icmp->icmp_id != pid)
19              return;                 /* not a response to our ECHO_REQUEST */
20          if (icmplen < 16)
21              return;                 /* not enough data to use */

22          tvsend = (struct  timeval  *) icmp->icmp_data;
23          tv_sub (tvrecv, tvsend);
24          rtt = tvrecv->tv_sec * 1000.0 + tvrecv->tv_usec / 1000.0;

25          printf ("%d bytes from %s: seq=%u, ttl=%d, rtt=%.3f ms\n",
26                  icmplen, Sock_ntop_host (pr->sarecv, pr->salen),
27                  icmp->icmp_seq, ip->ip_ttl, rtt);

28      } else if  (verbose) {
29          printf (" %d bytes from %s: type = %d, code = %d\n",
30                  icmplen, Sock_ntop_host (pr->sarecv, pr->salen),
31                  icmp->icmp_type, icmp->icmp_code);
32      }
33  }
```

Figure 28.13 `sig_alrm` function: `SIGALRM` signal handler.

ping/sig_alrm.c

```
1 #include     "ping.h"

2 void
3 sig_alrm (int signo)
4 {
5     (*pr->fsend) ();

6     alarm(1);
7     return;
8 }
```

Figure 28.14 `send_v4` function: builds an ICMPv4 echo request message and sends it.

ping/send_v4.c

```
 1 #include     "ping.h"

 2 void
 3 send_v4 (void)
 4 {
 5     int     len;
 6     struct icmp *icmp;

 7     icmp = (struct icmp *) sendbuf;
 8     icmp->icmp_type = ICMP_ECHO;
 9     icmp->icmp_code = 0;
10     icmp->icmp_id = pid;
11     icmp->icmp-seq = nsent++;
12     memset (icmp->icmp_data, 0xa5, datalen); /* fill with pattern */
13     Gettimeofday ((struct timeval *) icmp->icmp_data, NULL);

14     len = 8 + datalen;           /* checksum ICMP header and data */
15     icmp->icmp_cksum = 0;
16     icmp->icmp_cksum = in_cksum ((u_short *) icmp, len);

17     Sendto (sockfd, sendbuf, len, 0, pr->sasend, pr->salen);
18 }
```

Figure 28.15 `in_cksum` function: calculate the Internet checksum.

libfree/in_cksum.c

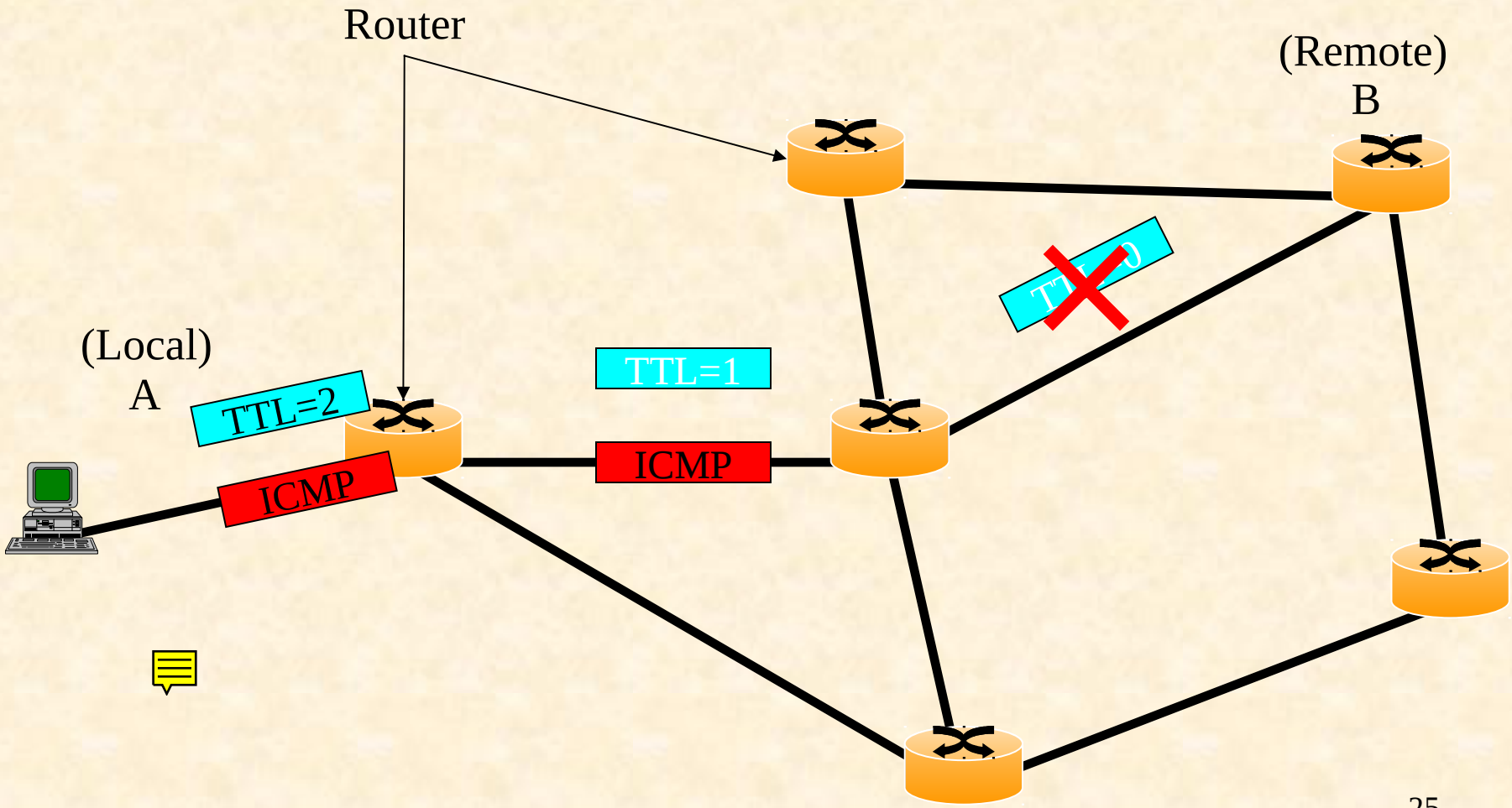
```
 1 uint16_t
 2 in_cksum (uint16_t * addr, int len)
 3 {
 4     int     nleft = len;
 5     uint32_t sum = 0;
 6     uint16_t *w = addr;
 7     uint16_t answer = 0;

 8     /*
 9      * Our algorithm is simple, using a 32 bit accumulator (sum), we add
10      * sequential 16 bit words to it, and at the end, fold back all the
11      * carry bits from the top 16 bits into the lower 16 bits.
12      */
13     while (nleft > 1) {
14         sum += *w++;
15         nleft -= 2;
16     }
```

```
17          /* mop up an odd byte, if necessary */
18      if (nleft == 1) {
19          * (unsigned char *) (&answer) = * (unsigned char *) w;
20          sum += answer;
21      }

22          /* add back carry outs from top 16 bits to low 16 bits */
23      sum = (sum >> 16) + (sum & 0xffff); /* add hi 16 to low 16 */
24      sum += (sum >> 16);     /* add carry */
25      answer = ~sum;      /* truncate to 16 bits */
26      return (answer);
27  }
```

Traceroute with TTL Mechanism



Example of Traceroute

Solaris # **traceroute gemini.tuc.noao.edu**

traceroute to gemini.tuc.noao.edu (140. 252. 3. 54): 30 hops max, 12 data bytes

1 gw.kohala.com (206.62.226.62) 3.839ms 3.595ms 3.722ms

2 tuc -1 -s1 -9 .rtd.net (206.85.40.73) 40.014ms 21.078ms 18.826ms

3 frame -gw.ttn.ep.net (198.32.152.9) 39.283ms 24.598ms 50.037ms

•

•

•

7 gemini.tuc.noao.edu (140.252.3.54) 70.476ms 43.555ms 88.716ms

Figure 28.21. Headers, pointers, and lengths in processing ICMPv4 err

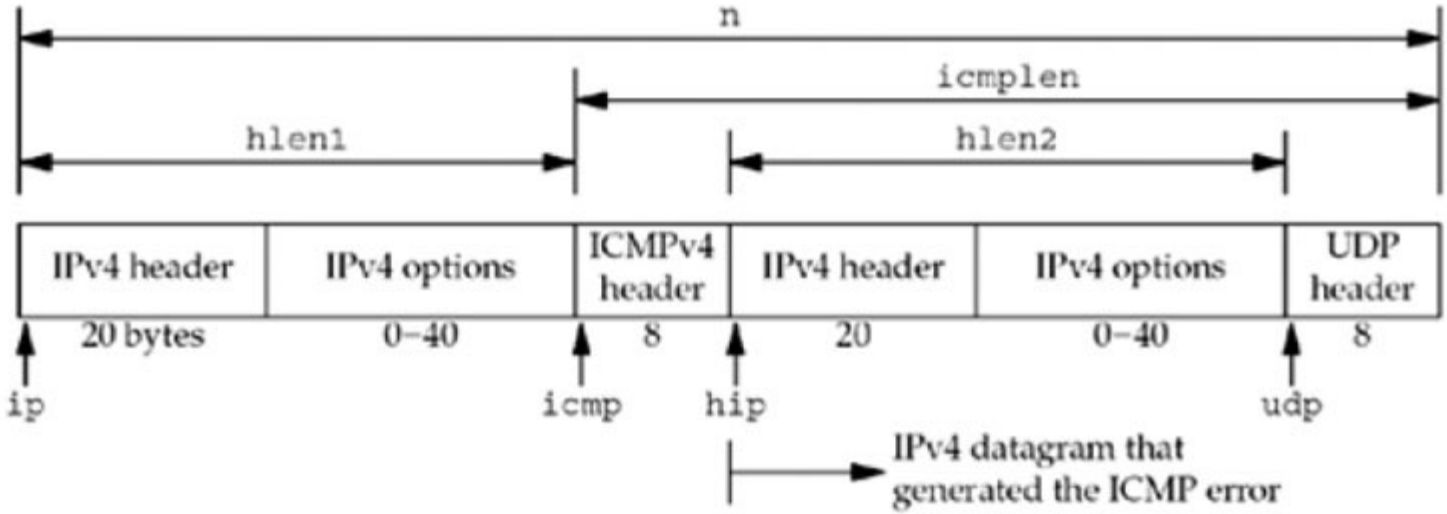


Figure 28.17 `trace.h` header.

traceroute/trace.h

```
 1 #include     "unp.h"
 2 #include    <netinet/in_systm.h>


 3 #include    <netinet/ip.h>
 4 #include    <netinet/ip_icmp.h>
 5 #include    <netinet/udp.h>

 6 #define BUFSIZE      1500

 7 struct rec {                          /* of outgoing UDP data */
 8     u_short rec_seq;                 /* sequence number */
 9     u_short rec_ttl;                 /* TTL packet left with */
10     struct timeval rec_tv;           /* time packet left */
11 };

12              /* globals */
13 char     recvbuf [BUFSIZE];
14 char     sendbuf [BUFSIZE];
```

```
15 int     datalen;                    /* # bytes of data following ICMP header *
16 char   *host;
17 u_short sport, dport;
18 int     nsent;                      /* add 1 for each sendto () */
19 pid_t   pid;                        /* our PID */
20 int     probe, nprobes;
21 int     sendfd, recvfd;             /* send on UDP sock, read on raw ICMP sock
22 int     ttl, max_ttl;
23 int     verbose;

24             /* function prototypes */
25 const char *icmpcode_v4 (int);
26 const char *icmpcode_v6 (int);
27 int     recv_v4 (int, struct timeval *);
28 int     recv_v6 (int, struct timeval *);
29 void    sig_alrm (int);
30 void    traceloop (void);
31 void    tv_sub (struct timeval *, struct timeval *);
```

```
32 struct proto {
33     const char *(*icmpcode) (int);
34     int     (*recv) (int, struct timeval *);
35     struct sockaddr *sasend;    /* sockaddr{} for send, from getaddrinfo */
36     struct sockaddr *sarecv;    /* sockaddr{} for receiving */
37     struct sockaddr *salast;    /* last sockaddr{} for receiving */
38     struct sockaddr *sabind;    /* sockaddr{} for binding source port */
39     socklen_t salen;            /* length of sockaddr{}s */
40     int     icmpproto;          /* IPPROTO_xxx value for ICMP */
41     int     ttllevel;           /* setsockopt () level to set TTL */
42     int     ttloptname;         /* setsockopt () name to set TTL */
43 } *pr;

44 #ifdef IPV6

45 #include     <netinet/ip6.h>
46 #include     <netinet/icmp6.h>

47 #endif
```

Figure 28.18 `main` function for `traceroute` program.

traceroute/main.c

```
 1 #include     "trace.h"

 2 struct proto proto_v4 = { icmpcode_v4, recv_v4, NULL, NULL, NULL, NULL, 0,
 3     IPPROTO_ICMP, IPPROTO_IP, IP_TTL
 4 };

 5 #ifdef IPV6
 6 struct proto proto_v6 = { icmpcode_v6, recv_v6, NULL, NULL, NULL, NULL, 0,
 7     IPPROTO_ICMPV6, IPPROTO_IPV6, IPV6_UNICAST_HOPS
 8 };
 9 #endif

10 int     datalen = sizeof (struct rec); /* defaults */
11 int     max_ttl = 30;
12 int     nprobes = 3;
13 u_short dport = 32768 + 666;
```

```
14 int
15 main(int argc, char **argv)
16 {
17      int     c;

18      struct addrinfo *ai;
19      char    *h;

20      opterr = 0;                     /* don't want getopt () writing to stderr */
21      while ( (c = getopt (argc, argv, "m:v")) != -1) {
22          switch (c) {
23          case 'm':
24              if ( (max_ttl = atoi (optarg)) <= 1)
25                  err_quit ("invalid -m value");
26              break;

27          case 'v':
28              verbose++;
29              break;

30          case '?':
31              err_quit ("unrecognized option: %c", c);
32          }
33      }
```

```
34      if (optind != argc - 1)
35          err_quit ("usage: traceroute [ -m <maxttl> -v ] <hostname>");
36      host = argv [optind];

37      pid = getpid();
38      Signal (SIGALRM, sig_alrm);

39      ai = Host_serv (host, NULL, 0, 0);

40      h = Sock_ntop_host (ai->ai_addr, ai->ai_addrlen);
41      printf ("traceroute to %s (%s) : %d hops max, %d data bytes\n",
42              ai->ai_canonname ? ai->ai_canonname : h, h, max_ttl, datalen);

43      /* initialize according to protocol */
44      if (ai->ai_family == AF_INET) {
45          pr = &proto_v4;
46  #ifdef  IPV6
47      } else if (ai->ai_family == AF_INET6) {
48          pr = &proto_v6;
49          if (IN6_IS_ADDR_V4MAPPED
50              (&(((struct sockaddr_in6 *) ai->ai_addr)->sin6_addr)))
51              err_quit ("cannot traceroute IPv4-mapped IPv6 address");
52  #endif
```

```
46 #ifdef  IPV6
47     } else if (ai->ai_family == AF_INET6) {
48         pr = &proto_v6;
49         if (IN6_IS_ADDR_V4MAPPED
50             (&(((struct sockaddr_in6 *) ai->ai_addr)->sin6_addr)))
51             err_quit ("cannot traceroute IPv4-mapped IPv6 address");
52 #endif
53     } else
54         err_quit ("unknown address family %d", ai->ai_family);

55     pr->sasend = ai->ai_addr; /* contains destination address */
56     pr->sarecv = Calloc (1, ai->ai_addrlen);
57     pr->salast = Calloc (1, ai->ai_addrlen);
58     pr->sabind = Calloc (1, ai->ai_addrlen);
59     pr->salen = ai->ai_addrlen;

60     traceloop();

61     exit (0);
62 }
```

Figure 28.19 `traceloop` function: main processing loop.

traceroute/traceloop.c

```
 1 #include     "trace.h"

 2 void
 3 traceloop(void)
 4 {
 5     int      seq, code, done;
 6     double   rtt;
 7     struct rec *rec;
 8     struct timeval tvrecv;

 9     recvfd = Socket (pr->sasend->sa_family, SOCK_RAW, pr->icmpproto);
10     setuid (getuid());           /* don't need special permissions anymore */

11 #ifdef  IPV6
12     if (pr->sasend->sa_family == AF_INET6 && verbose == 0) {
13         struct icmp6_filter myfilt;
14         ICMP6_FILTER_SETBLOCKALL (&myfilt);
15         ICMP6_FILTER_SETPASS (ICMP6_TIME_EXCEEDED, &myfilt);
16         ICMP6_FILTER_SETPASS (ICMP6_DST_UNREACH, &myfilt);
17         setsockopt (recvfd, IPPROTO_IPV6, ICMP6_FILTER,
18                     &myfilt, sizeof (myfilt));
19     }
20 #endif
```

```
21      sendfd = Socket (pr->sasend->sa_family, SOCK_DGRAM, 0);

22      pr->sabind->sa_family = pr->sasend->sa_family;
23      sport = (getpid() & 0xffff) | 0x8000; /* our source UDP port # */
24      sock_set_port (pr->sabind, pr->salen; htons (sport));
25      Bind (sendfd, pr->sabind, pr->salen);

26      sig_alrm (SIGALRM);

27      seq = 0;
28      done = 0;
29      for (ttl = 1; ttl <= max_ttl && done == 0; ttl++) {
30          Setsockopt (sendfd, pr->ttllevel, pr->ttloptname, &ttl, sizeof (int))
31          bzero (pr->salast, pr->salen);

32          printf ("%2d ", ttl);
33          fflush (stdout);

34          for (probe = 0; probe < nprobes; probe++) {
35              rec = (struct rec *) sendbuf;
36              rec->rec_seq = ++seq;
37              rec->rec_ttl = ttl;
38              Gettimeofday (&rec->rec_tv, NULL);

39              sock_set_port (pr->sasend, pr->salen, htons (dport + seq));
40              Sendto (sendfd, sendbuf, datalen, 0, pr->sasend, pr->salen),
```

```
41              if ( (code = (*pr->recv) (seq, &tvrecv)) == -3)
42                  printf (" *"); /* timeout, no reply */
43              else {
44                  char    str [NI_MAXHOST];
45                  if (sock_cmp_addr (pr->sarecv, pr->salast, pr->salen) != 0) {
46                  if (getnameinfo (pr->sarecv, pr->salen, str, sizeof (str),
47                                   NULL, 0, 0) == 0)
48                      printf (" %s (%s)", str,
49                              Sock_ntop_host (pr->sarecv, pr->salen));
50                  else
51                      printf (" %s", Sock_ntop_host (pr->sarecv, pr->salen));
52                  memcpy (pr->salast, pr->sarecv, pr->salen);
53                  {
54          tv_sub (&tvrecv, &rec->rec_tv);
55          rtt = tvrecv.tv_sec * 1000.0 + tvrecv.tv_usec / 1000.0;
56          printf (" %.3f ms", rtt);
```

```
57          if (code == -1) /* port unreachable; at destination */
58              done++;
59          else if (code >= 0)
60              printf (" (ICMP %s)", (*pr->icmpcode) (code));
61        }
62        fflush (stdout);
63      }
64      printf ("\n");
65    }
66  }
```

Figure 28.20 `recv_v4` function: reads and processes ICMPv4 messages.

traceroute/recv_v4.c

```
 1 #include     "trace.h"

 2 extern int gotalarm;

 3 /*
 4  * Return: -3 on timeout
 5  *         -2 on ICMP time exceeded in transit (caller keeps going)
 6  *         -1 on ICMP port unreachable (caller is done)
 7  *       >= 0 return value is some other ICMP unreachable code
 8  */
 9 int
10 recv_v4(int seq, struct timeval *tv)
11 {
12     int     hlen1, hlen2, icmplen, ret;
13     socklen_t len;
14     ssize_t n;
15     struct ip *ip, *hip;
16     struct icmp *icmp;
17     struct udphdr *udp;
```

```
18      gotalarm = 0;
19      alarm(3);
20      for ( ; ; ) {
21          if (gotalarm)
22              return (-3);        /* alarm expried */
23          len = pr->salen;
24          n = recvfrom (recvfd, recvbuf, sizeof (recvbuf), 0, pr->sarecv, &len);
25          if (n < 0) {
26              if (errno == EINTR)
27                  continue;
28              else
29                   err_sys ("recvfrom error");
30           }

31           ip = (struct ip *) recvbuf; /* start of IP header */
32           hlen1 = ip->ip_hl << 2; /* length of IP header */

33           icmp = (struct icmp *) (recvbuf + hlen1); /* start of ICMP header */
34           if ( (icmplen = n - hlen1) < 8)
35               continue;     /* not enough to look at ICMP header */

36           if (icmp->icmp_type == ICMP_TIMXCEED &&
37               icmp->icmp_code == ICMP_TIMXCEED_INTRANS) {
38               if (icmplen < 8 + sizeof (struct ip))
39               continue;          /* not enough data to look at inner IP */
```

```
40          hip = (struct ip *) (recvbuf + hlen1 + 8);
41          hlen2 = hip->ip_hl << 2;
42          if (icmplen < 8 + hlen2 + 4)
43              continue;           /* not enough data to look at UDP ports */

44          udp = (struct udphdr *) (recvbuf + hlen1 + 8 + hlen2);
45          if (hip->ip_p == IPPROTO_UDP &&
46              udp->uh_sport == htons (sport) &&
47              udp->uh_dport == htons (dport + seq)) {
48              ret = -2;           /* we hit an intermediate router */
49              break;
50          }

51      } else if (icmp->icmp_type == ICMP_UNREACH) {
52          if (icmplen <8 + sizeof (struct ip))
53               continue;          /* not enough data to look at inner IP */

54          hip = (struct ip *) (recvbuf + hlen1 + 8);
55          hlen2 = hip->ip_hl << 2;
56          if (icmplen < 8 + hlen2 + 4)
57              continue;          /* not enough data to look at UDP ports */

58          udp = (struct udphdr *) (recvbuf + hlen1 + 8 + hlen2);
59          if (hip->ip_p == IPPROTO_UDP &&
```

```
60              udp->uh_sport == htons (sport) &&
61              udp->uh_dport == htons (dport + seq)) {
62              if (icmp->icmp_code == ICMP_UNREACH_PORT)
63                  ret = -1;    /* have reached destination */
64              else
65                  ret = icmp->icmp_code; /* 0, 1, 2, ... */
66              break;
67          }
68      }
69      if (verbose) {
70          printf (" (from %s: type = %d, code = %d)\n",
71                  Sock_ntop_host (pr->sarecv, pr->salen),
72                  icmp->icmp_type, icmp->icmp_code);
73      }
74      /* Some other ICMP error, recvfrom() again */
75      }
76      alarm(0);                       /* don't leave alarm running */
77      Gettimeofday (tv, NULL);        /* get time of packet arrival */
78      return (ret);
79  }
```

Figure 28.23 `sig_alrm` function.

traceroute/sig_alrm.c

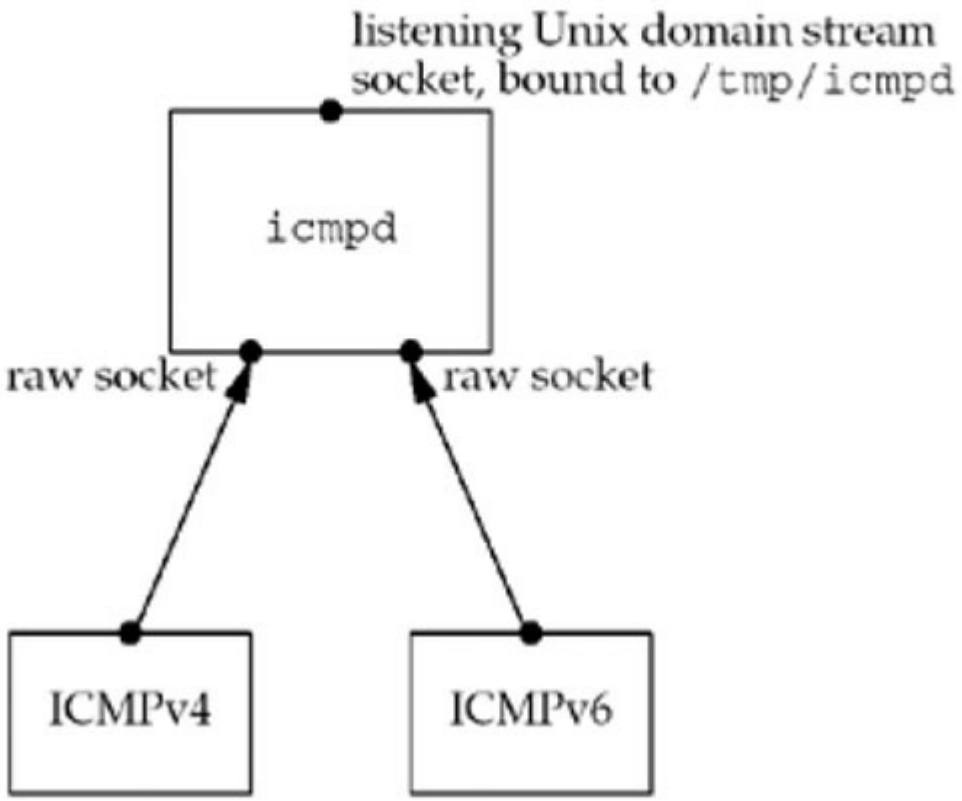
```
1 #include     "trace.h"

2 int      gotalarm;

3 void
4 sig_alrm(int signo)

5 {
6      gotalarm = 1:          /* set flag to note that alarm occurred */
7      return;                /* and interrupt the recvfrom() */
8  }
```

Figure 28.26. `icmpd` daemon: initial sockets created.



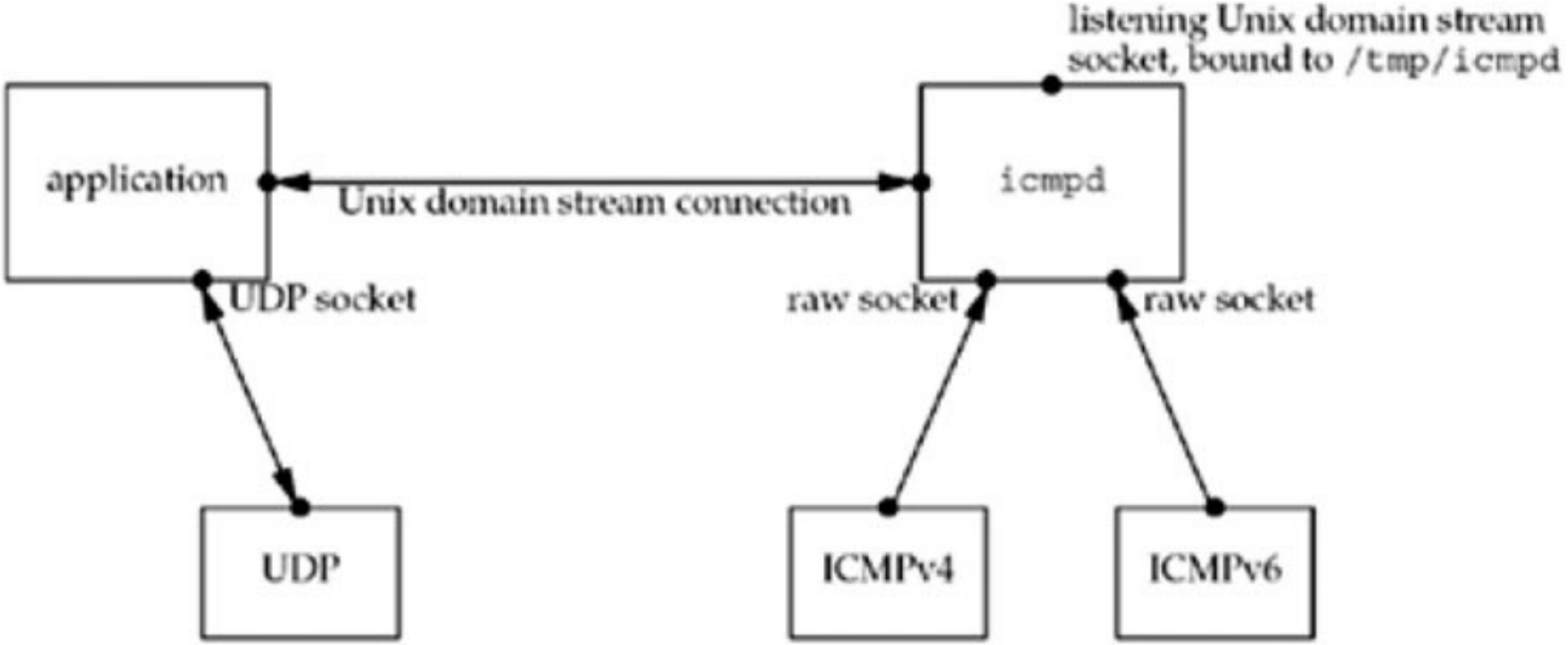

listening Unix domain stream
socket, bound to /tmp/icmpd
application
Unix domain stream connection
icmpd
UDP socket
raw socket
raw socket
UDP
ICMPv4
ICMPv6

Figure 28.28. Passing UDP socket to daemon across Unix domain connection.

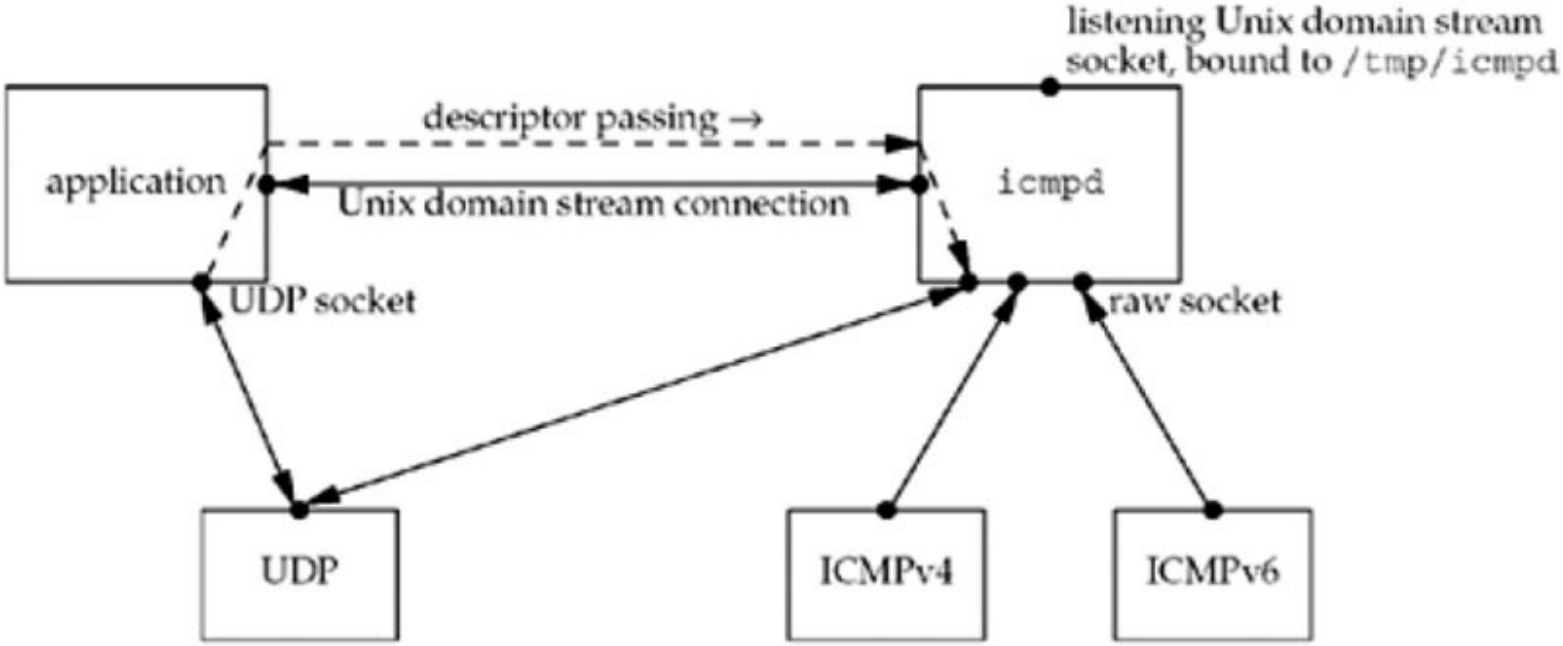


Figure 28.29 `unpicmpd.h` header.

icmpd/unpicmpd.h

```
 1 #ifndef __unpicmp_h
 2 #define __unpicmp_h

 3 #include   "unp.h"

 4 #define ICMPD_PATH     "/tmp/icmpd"     /* server's well-known pathname */

 5 struct icmpd_err {
 6     int      icmpd_errno;      /* EHOSTUNREACH, EMSGSIZE, ECONNREFUSED */
 7     char     icmpd_type;       /* actual ICMPv[46] type */
 8     char     icmpd_code;       /* actual ICMPv[46] code */
 9     socklen_t icmpd_len;       /* length of sockaddr{} that follows */
10     struct sockaddr_storage icmpd_dest;  /* sockaddr_storage handles any size
11 };
12 #endif  /* __unpicmp_h */
```

Figure 28.30. `icmpd_errno` mapping from ICMPv4 and ICMPv6 errors.

icmpd_errno	ICMPv4 error	ICMPv6 error
ECONNREFUSED	port unreachable	port unreachable
EMSGSIZE	fragmentation needed but DF set	packet too big
EHOSTUNREACH	time exceeded	time exceeded
EHOSTUNREACH	source quench	
EHOSTUNREACH	All other destination unreachables	All other destination unreachables

Figure 28.32 Last half of `dg_cli` application.

icmpd/dgcli01.c

```
24      while (Fgets(sendline, MAXLINE, fp) ! = NULL) {
25          Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

26          tv.tv_sec = 5;
27          tv.tv_usec = 0;
28          FD_SET(sockfd, &rset);
29          FD_SET(icmpfd, &rset);
30          if ( (n = Select(maxfdpl, &rset,  NULL, NULL, &tv)) == 0) {
31              fprintf(stderr,   socket timeout\n);
32              continue;
33          }

34          if (FD_ISSET(sockfd,  &rset)) {
35              n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
36              recvline[n] = 0;      /* null terminate */
37              Fputs(recvline, stdout);
38          }
```

```
39          if (FD_ISSET(icmpfd, &rset)) {
40              if ( (n = Read(icmpfd, &icmpd_err, sizeof(icmpd_err))) == 0)
41                  err_quit ("ICMP daemon terminated");
42              else if (n ! = sizeof(icmpd_err))
43                  err_quit("n = %d, expected %d", n, sizeof(icmpd_err));
44              printf("ICMP error: dest = %s, %s, type = %d, code = %d\n",
45                      Sock_ntop(&icmpd_err.icmpd_dest, icmpd_err.icmpd_len),
46                      strerror(icmpd_err.icmpd_errno),
47                      icmpd_err.icmpd_type, icmpd_err.icmpd_code);
48          }
49      }
50  }
```

Figure 28.33 `icmpd.h` header for `icmpd` daemon.

icmpd/icmpd.h

```
 1 #include     "unpicmpd.h"
 2 struct client {
 3     int     connfd;             /* Unix domain stream socket to client */
 4     int     family;             /* AF_INET or AF_INET6 */
 5     int     lport;              /* local port bound to client's UDP socket */
 6     /* network byte ordered */
 7 } client [FD_SETSIZE];

 8                             /* globals */
 9 int    fd4, fd6, listenfd, maxi, maxfd, nready;
10 fd_set rset, allset;
11 struct sockaddr_un cliaddr;

12                 /* function prototypes */
13 int     readable_conn (int);
14 int     readable_listen (void);
15 int     readable_v4 (void);
16 int     readable_v6 (void);
```

Figure 28.34 First half of `main` function: creates sockets.

icmpd/icmpd.c

```
 1 #include     "icmpd.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     i, sockfd;
 6     struct sockaddr_un sun;

 7     if (argc != 1)
 8         err_quit ("usage: icmpd");

 9     maxi = -1;                    /* index into client [] array */
10     for (i = 0; i < FD_SETSIZE; i++)
11         client [i] .connfd = -1;   /* -1 indicates available entry */
12     FD_ZERO (&allset);

13     fd4 = Socket (AF_INET, SOCK_RAW, IPPROTO_ICMP);
14     FD_SET (fd4, &allset);
15     maxfd = fd4;
```

```
16 #ifdef IPV6
17     fd6 = Socket (AF_INET6, SOCK_RAW, IPPROTO_ICMPV6);
18     FD_SET (fd6, &allset);
19     maxfd = max (maxfd, fd6);
20 #endif

21     listenfd = Socket (AF_UNIX, SOCK_STREAM, 0);
22     sun.sun_family = AF_LOCAL;
23     strcpy (sun.sun_path, ICMPD_PATH);
24     unlink (ICMPD_PATH);
25     Bind (listenfd, (SA *) &sun, sizeof (sun));
26     Listen (listenfd, LISTENQ);
27     FD_SET (listenfd, &allset);
28     maxfd = max (maxfd, listenfd);
```

Figure 28.35 Second half of `main` function: handles readable descriptor.

icmpd/icmpd.c

```
29      for ( ; ; ) {
30          rset = allset;
31          nready = Select (maxfd + 1, &rset, NULL, NULL, NULL);

32          if (FD_ISSET (listenfd, &rset))
33              if (readable_listen () <= 0)
34                  continue;

35          if (FD_ISSET (fd4, &rset))
36              if (readable_v4 () <= 0)
37                  continue;
```

```
38 #ifdef IPV6
39         if (FD_ISSET (fd6, &rset))
40               if (readable_v6 () <= 0)
41                   continue;
42 #endif

43           for (i = 0; i <= maxi; i++) { /* check all clients for data */
44               if ( (sockfd = client [i] .connfd) < 0)
45                   continue;
46           if (FD_ISSET (sockfd, &rset))
47               if (readable_conn (i) <= 0)
48                   break;     /* no more readable descriptors */
49           }
50     }
51     exit (0);
52 }
```

Figure 28.36 Handle new client connections.

icmpd/readable_listen.c

```
 1 #include   "icmpd.h"

 2 int
 3 readable_listen (void)
 4 {
 5     int     i, connfd;
 6     socklen_t clilen;

 7     clilen = sizeof (cliaddr);
 8     connfd = Accept (listenfd, (SA *) &cliaddr, &clilen);

 9         /* find first available client [] structure */
10     for (i = 0; i < FD_SETSIZE; i++)
11         if (client [i] .connfd < 0) {
12             client [i] .connfd = connfd; /* save descriptor */
13             break;
14          }
```

```
15      if (i == FD_SETSIZE) {
16          close (connfd);             /* can't handle new client, */
17          return (--nready);          /* rudely close the new connection */
18      }
19      printf ("new connection, i = %d, connfd = %d\n", i, connfd);

20      FD_SET (connfd, &allset);   /* add new descriptor to set */
21      if (connfd > maxfd)
22          maxfd = connfd;             /* for select () */
23      if (i > maxi)
24          maxi = i;                   /* max index in client [] array */

25      return (--nready);
26  }
```

Figure 28.37 Read data and possible descriptor from client.

icmpd/readable_conn.c

```
 1 #include     "icmpd.h"

 2 int
 3 readable_conn(int i)
 4 {
 5     int     unixfd, recvfd;
 6     char    c;
 7     ssize_t n;
 8     socklen_t len;
 9     struct sockaddr_storage ss;

10     unixfd = client [i] .connfd;
11     recvfd = -1;
12     if ( (n = Read_fd (unixfd, &c, 1, &recvfd)) == 0) {
13         err_msg ("client %d terminated, recvfd = %d", i, recvfd);
14         goto clientdone;          /* client probably terminated */
15     }

16         /* data from client; should be descriptor */
17     if (recvfd < 0) {
18         err_msg ("read_fd did not return descriptor");
19         goto clienterr;
20     }
```

Figure 28.38 Get port number that client has bound to its UDP socket.

icmpd/readable_conn.c

```
21      len = sizeof (ss);
22      if (getsockname (recvfd, (SA *) &ss, &len) < 0) {
23          err_ret ("getsockname error");
24          goto clienterr;
25      }

26      client[i].family = ss.ss_family;
27      if ((client[i].lport = sock_get_port ((SA *) &ss, len)) == 0) {
28          client[i].lport = sock_bind_wild (recvfd, client[i].family);
29          if (client[i].lport <= 0) {
30              err_ret ("error binding ephemeral port");
31              goto clienterr;
32          }
33      }
34      Write (unixfd, "1", 1);          /* tell client all OK */
35      Close (recvfd);                 /* all done with client's UDP socket */
36      return  (--nready);
```

```
37    clienterr:
38      Write (unixfd, "0", 1);          /* tell client error occurred */
39      clientdone:
40        Close (unixfd);
41        if (recvfd >= 0)
42            Close (recvfd);
43        FD_CLR (unixfd, &allset);
44        client[i].connfd = -1;
45        return (--nready);
46  }
```

Figure 28.39 Handle received ICMPv4 datagram, first half.

icmpd/readable_v4.c

```
 1 #include     "icmpd.h"
 2 #include     <netinet/in_systm.h>
 3 #include     <netinet/ip.h>
 4 #include     <netinet/ip_icmp.h>
 5 #include     <netinet/udp.h>

 6 int
 7 readable_v4 (void)
 8 {
 9     int      i, hlen1, hlen2, icmplen, sport;
10     char     buf[MAXLINE];
11     char     srcstr [INET_ADDRSTRLEN], dststr[INET_ADDRSTRLEN];
12     ssize_t  n;
13     socklen _ t len;
14     struct  ip *ip, *hip;
15     struct  icmp *icmp;
16     struct  udphdr *udp;
17     struct  sockaddr_in from, dest;
18     struct  icmpd_err icmpd_err;
```

```
19      len =  sizeof (from);
20      n =  Recvfrom(fd4, buf, MAXLINE, 0, (SA *) &from, &len);

21      printf("%d bytes ICMPv4 from %s:", n, Sock_ntop_host ((SA *) &from, len));

22      ip = (struct ip *) buf;      /* start of IP header */
23      hlen1 = ip->ip_hl << 2;      /* length of IP header */

24      icmp = (struct icmp *) (buf + hlen1);      /* start of ICMP header */
25      if ( (icmplen = n - hlen1) < 8)
26           err_quit("icmplen (%d) < 8", icmplen);

27      printf(" type = %d, code = %d\n", icmp->icmp_type, icmp->icmp_code);
```

Figure 28.40 Handle received ICMPv4 datagram, second half.

icmpd/readable_v4.c

```
28      if (icmp->icmp_type == ICMP_UNREACH ||
29          icmp->icmp_type == ICMP_TIMXCEED ||
30          icmp->icmp_type == ICMP_SOURCEQUENCH) {
31          if (icmplen < 8 + 20 + 8)
32              err_quit("icmplen (%d) < 8 + 20 + 8", icmplen);
33          hip = (struct ip *) (buf + hlen1 + 8);
34          hlen2 = hip->ip_hl << 2;
35          printf("\tsrcip = %s, dstip = %s, proto = %d\n",
36                 Inet_ntop(AF_INET, &hip->ip_src, srcstr, sizeof(srcstr)),
37                 Inet_ntop(AF_INET, &hip->ip_dst, dststr, sizeof(dststr)),
38                 hip->ip_p);
39          if (hip->ip_p == IPPROTO_UDP) {
40              udp = (struct udphdr *) (buf + hlen1 + 8 + hlen2);
41              sport = udp->uh_sport;

42                  /* find client's Unix domain socket, send headers */
43              for (i = 0; i <= maxi; i++) {
44                  if (client[i].connfd >= 0 &&
45                      client[i].family == AF_INET &&
46                      client[i].lport == sport) {
```

```
47                      bzero(&dest, sizeof(dest));
48                      dest.sin_family = AF_INET;
49 #ifdef  HAVE_SOCKADDR_SA_LEN
50                      dest.sin_len = sizeof(dest);
51 #endif
52                      memcpy(&dest.sin_addr, &hip->ip_dst,
53                             sizeof(struct in_addr));
54                      dest.sin_port = udp->uh_dport;

55                      icmpd_err.icmpd_type = icmp->icmp_type;
56                      icmpd_err.icmpd_code = icmp->icmp_code;
57                      icmpd_err.icmpd_len = sizeof(struct sockaddr_in);
58                      memcpy(&icmpd_err.icmpd_dest, &dest, sizeof(dest));

59                          /* convert type & code to reasonable errno value */
60                      icmpd_err.icmpd_errno = EHOSTUNREACH;     /* default */
61                      if (icmp->icmp_type == ICMP_UNREACH) {
62                          if (icmp->icmp_code == ICMP_UNREACH_PORT)
63                              icmpd_err.icmpd_errno = ECONNREFUSED;
64                          else if (icmp->icmp_code == ICMP_UNREACH_NEEDFRAG)
65                              icmpd_err.icmpd_errno = EMSGSIZE;
66                      }
67                      Write(client[i].connfd, &icmpd_err, sizeof(icmpd_err));
68                  }
69              }
70          }
71      }
72      return (--nready);
73 }
```