

IT Operation Workshop: Day-1

Fundamental of IP routing

Keyword: IP addressing, IP forwarding, Static routing, Zebra introduction

Yasuhiro Ohara
WIDE Project

2005/08/30

1 IP addressing

1.1 binary operation

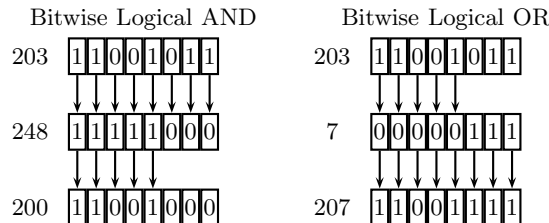


Figure 1: bitwise AND'ing and OR'ing

Logical AND operation is also known as "bit masking".

1.2 IP address

IP address is a binary number to identify a **network interface**. **IPv4 address notation** is a format of *A.B.C.D* where each octet is shown as a decimal, and each octet boundary is marked by "." (a dot).

The leftmost bit (bit 0) is called a **most significant bit** and the rightmost bit (bit 31) is called a **least significant bit**.

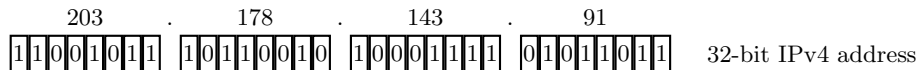


Figure 2: Example of an 32-bit IPv4 address

For routing purpose, IP address is divided into two parts: **network part** and **host part**. Generally network part of the IP address (called **IP prefix**) is used by routing protocols to locate the IP subnet (IP sub network, a minimum unit of network fragment in terms of IP routing). Once IP subnet is located, host part is used by ARP to identify individual host within the subnet.

To indicate the boundary between network part and host part, **netmask** is used. Example of netmask is illustrated in Figure 3. Both network part and host part must be contiguous, that is, there must be only one boundary between 1-bits and 0-bits in the netmask. Once the first 0-bit is reached in a netmask, all remaining bits must be 0¹. Today non-contiguous netmask is recognized as invalid. Netmask is both represented as decimal and hexadecimal, like 255.255.255.128 and 0xffff80 in Figure 3.

¹[1] intended to support non-contiguous netmask, but it never appeared in the real internet.

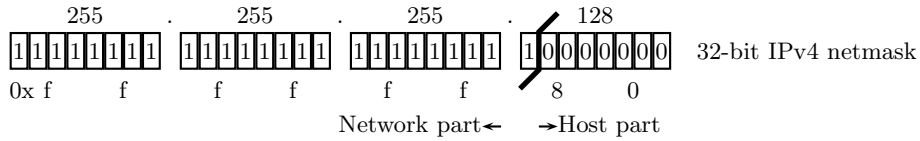


Figure 3: Example of an 32-bit IPv4 netmask (subnet mask)

Making bitwise AND operation on the IP address and the netmask yields **Network address** of the subnet. Network address of the subnet is an IP address with all host bits 0. *Broadcast address* of the subnet is an IP address with all host bits 1 and can be yielded by OR'ing with one's complement of the netmask. IP prefix is usually indicated with a network address and a indication of the size of network part (e.g. netmask).

More efficient way to describe IP prefix is to use **prefix length** rather than netmask. The format is *A.B.C.D/M* where the network address is followed by "/" (a slash) and a prefix length (*M*), which is a bit number of the network part.

Using the examples of Figure 2 and 3, the network address of the subnet is **203.178.143.0**, the broadcast address of the subnet is **203.178.143.127**, and the IP prefix of the subnet is **203.178.143.0/25**.

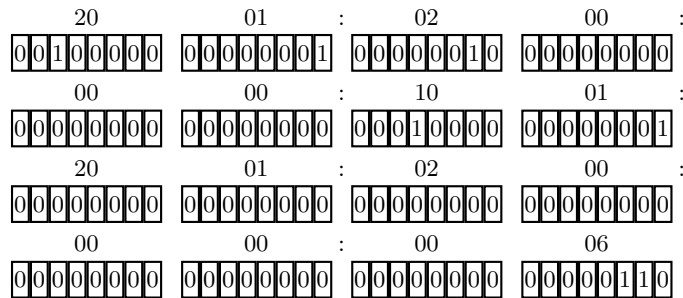


Figure 4: Example of an 128-bit IPv6 address: 2001:200:1001::6

IPv6 address is 128 bit long and the notation is represented by hexadecimal, each 16 bit (2 octets) segment separated by ":". In notation, leading 0 in each 16 bit segment can be omitted. Also in notation, a sequence of all-0 16 bit segments can be shortend into "::". ":" can occur only once in a address notation.

IPv6 address prefix is represented using prefix length, e.g. 2001:200:1001::/64. There is no netmask in IPv6.

1.3 aggregation

Individual network segment will typically be assigned the prefix in the range from /22 to /31². Managing all of those small pieces of address space individually will be a problem for IP routing entities in terms of memory consumption. Grouping some small pieces of address spaces into one big piece is called **aggregation**.

An IP prefix which indicates relatively smaller pieces of address space is called a **more specific** prefix. An IP prefix as a result of grouping some subnet prefixes is called **supernet**, and is called a **less specific** prefix.

Aggregated prefix will have shorter network part length (i.e. prefix length) than those of original prefixes.

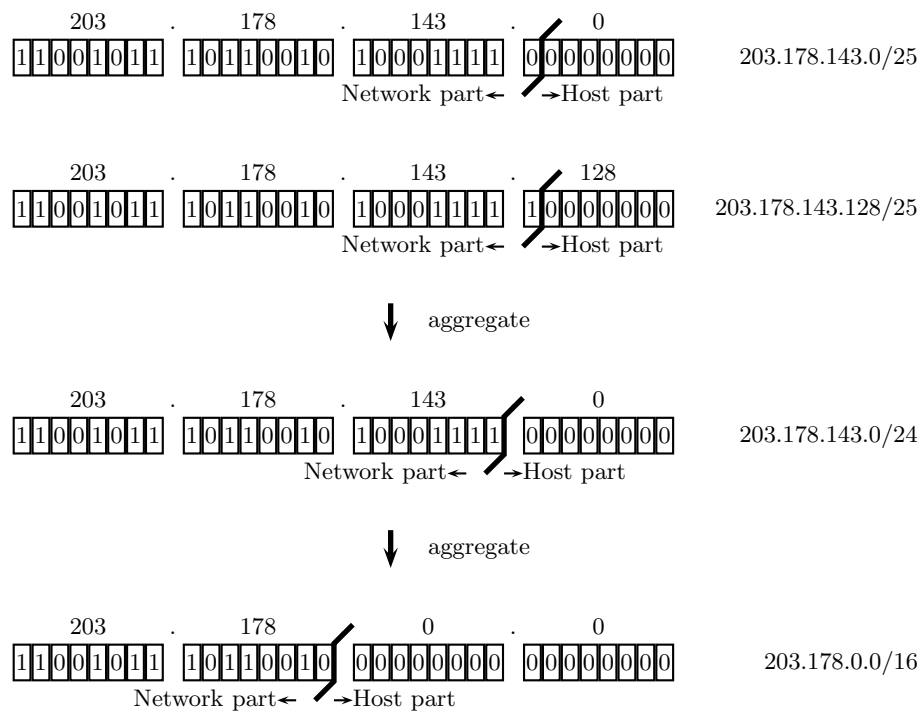


Figure 5: Example of aggregated prefix

In the example of Figure 5, 203.178.143.0/25 and 203.178.143.128/25 are aggregated into 203.178.143.0/24. Notice that two /25 prefixes are aggregated into one /24 prefix (which indicate the doubled size address space).

²Assigning /31 subnet eliminates the use of the network address and the broadcast address. see [2]

Lower half of the Figure 5 shows further aggregation from /24 to /16. Just shortening prefix length (and thus making the prefix indicate bigger space) is also called aggregation. It can be interpreted that in the example a real prefix 203.178.143.0/24 and virtual 254 prefixes ranging from 203.178.143.1/24 to 203.178.143.255/24 are aggregated into one big prefix 203.178.143.0/16. It is important to understand that other prefixes (e.g. 203.178.143.100/24 and 203.178.143.200/24, to name a few) are also fall into the aggregated range.

1.4 bestmatch/longest match

If an IP address falls into the range of an IP prefix, the prefix is said to **match** the IP address, and is called "matching prefix".

When there are prefixes ranging from shorter to longer, a routing lookup (for an IP address) may match several prefixes. To prioritize those matching prefixes, the concept of "**bestmatch**" or "**longest match**" is introduced.

Each bit in IP prefix is tested against the same bit in the IP address. The prefix that matches the maximum bit with the IP address is the bestmatching (longest matching) prefix. Figure 6 shows an example of IP prefix matching against an IP address (203.178.143.91).

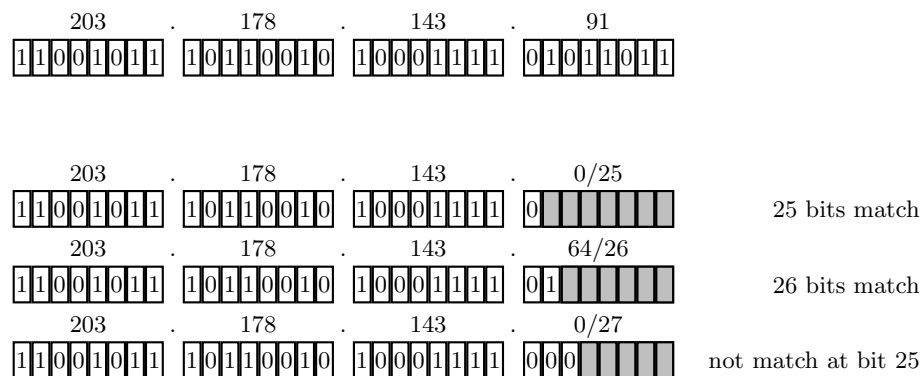


Figure 6: Example of prefix match

There are three prefixes, 203.178.143.0/25, 203.178.143.64/26 and 203.178.143.0/27. The bestmatching prefix for 203.178.143.91 is 203.178.143.64/26 with 26 bits matching. 203.178.143.0/25 also matches, but the matching bits are less. 203.178.143.0/27 does not match 203.178.143.91 because bit 25 is different with 203.178.143.91.

2 IP forwarding

Summary of ip_input, ip_forward, ip_output are given in appendix.

Process of forwarding IP packet(s) (Router, Interface, Switch/Bridge/Hub)

Structure of Routing table

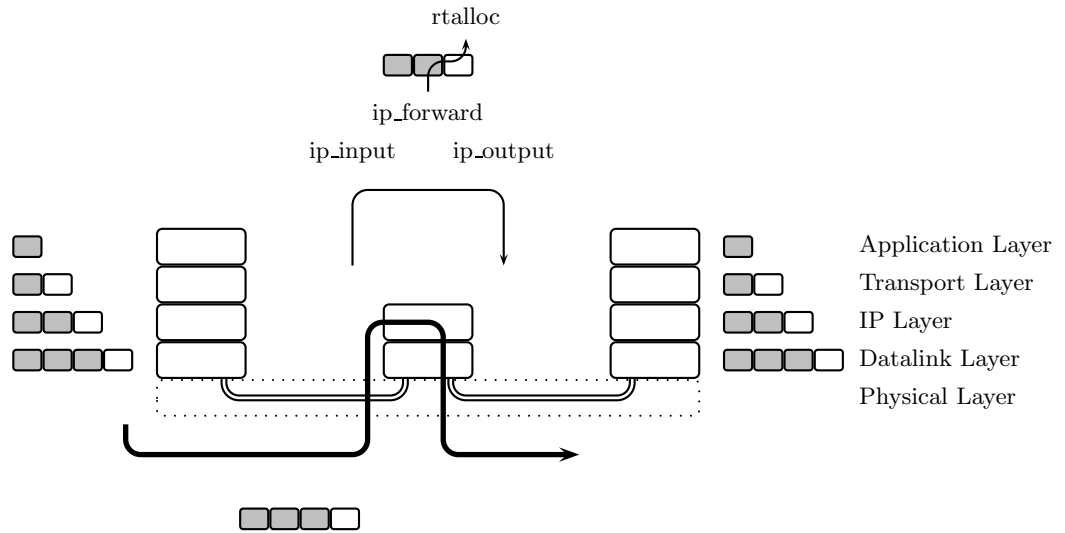


Figure 7: IP forwarding model

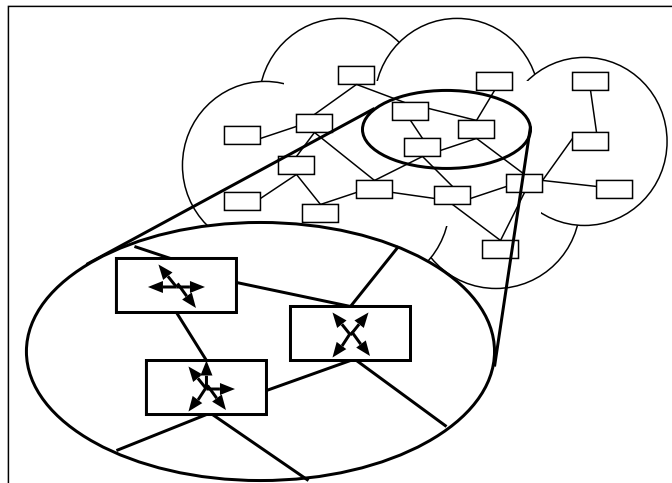


Figure 8: Role of routing table

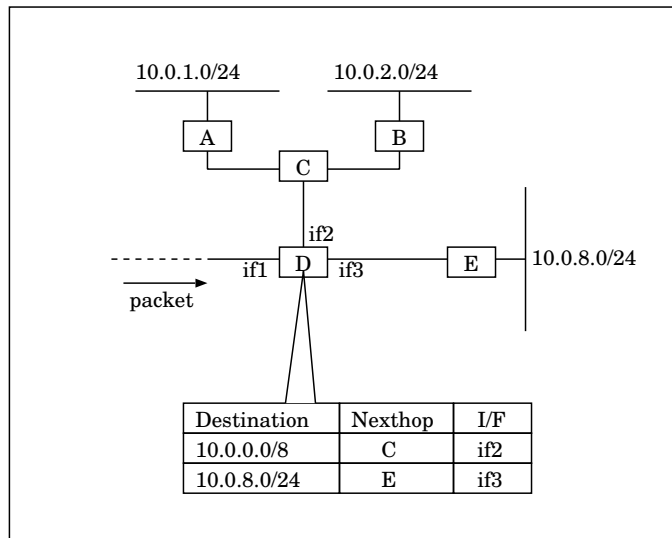


Figure 9: Example of best match

3 Static routing

1. What is static routing ?
2. How to see routes using netstat (1)

Table 1: Route Flags in netstat

1	RTF_PROTO1	H	RTF_HOST
2	RTF_PROTO2	L	RTF_LLINFO
B	RTF_BLACKHOLE	M	RTF_MODIFIED
C	RTF_CLONING	R	RTF_REJECT
c	RTF_CLONED	S	RTF_STATIC
D	RTF_DYNAMIC	U	RTF_UP
G	RTF_GATEWAY	X	RTF_XRESOLVE

3. How to add routes using route (8)
4. Monitoring routes

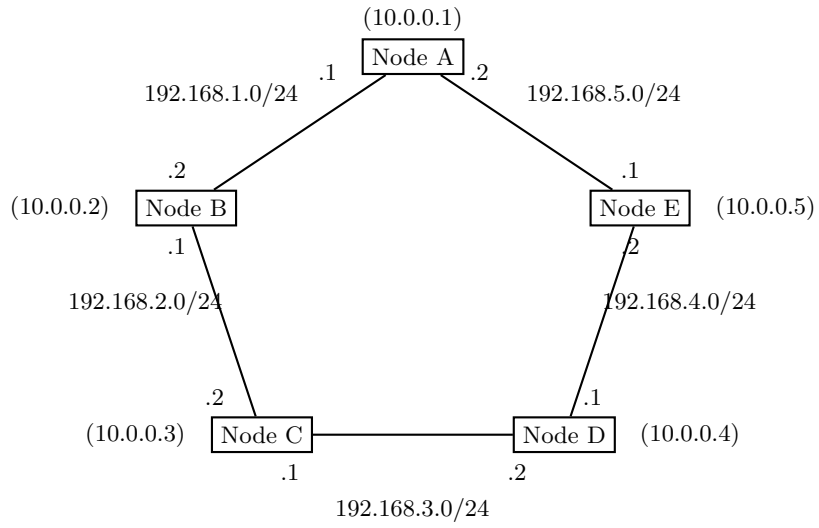


Figure 10: Configure a network

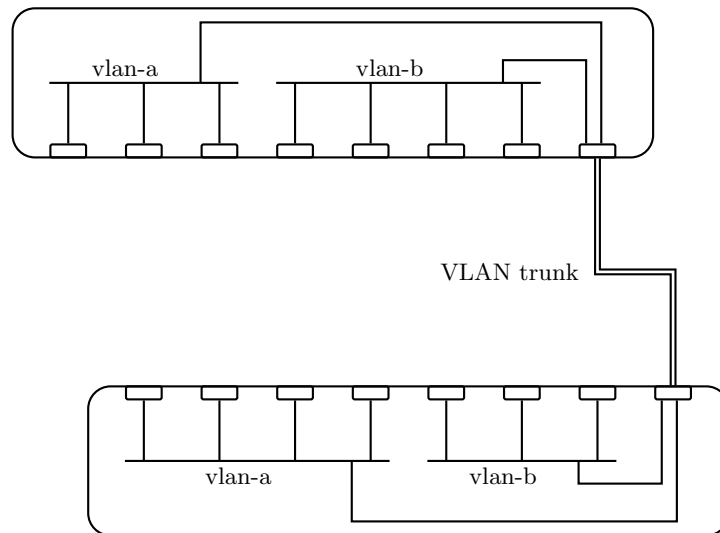


Figure 11: concept of a VLAN

4 Configure a network

5 VLAN

References

- [1] J.C. Mogul and J. Postel. Internet Standard Subnetting Procedure. RFC 950, IETF, August 1985.
- [2] A. Retana, R. White, V. Fuller, and D. McPherson. Using 31-Bit Prefixes on IPv4 Point-to-Point Links. RFC 3021, IETF, December 2000.

A Additional glossary

VLSM Variable Length Subnet Mask

CIDR Classless Inter Domain Routing

Class C A /24 prefix or that range of address

B ip_input

NetBSD 2.0.2 sys/netinet/ip_input.c (revision: 1.197.2.1):

```
478 /*
479  * Ip input routine.  Checksum and byte swap header.  If fragmented
480  * try to reassemble.  Process options.  Pass to next level.
481  */
482 void
483 ip_input(struct mbuf *m)
484 {
485     struct ip *ip = NULL;
486     struct ipq *fp;
487     struct in_ifaddr *ia;
488     struct ifaddr *ifa;
489     struct ipqent *ipqe;
490     int hlen = 0, mff, len;
491     int downmatch;
492     int checkif;
493     int srcrt = 0;
494     u_int hash;
495
496     :
497     :
498
534     ip = mtod(m, struct ip *);
535     if (ip->ip_v != IPVERSION) {
536         ipstat.ips_badvers++;
537         goto bad;
538     }
539     hlen = ip->ip_hl << 2;
540     if (hlen < sizeof(struct ip)) {          /* minimum header length */
541         ipstat.ips_badhlen++;
542         goto bad;
543     }
544     if (hlen > m->m_len) {
545         if ((m = m_pullup(m, hlen)) == 0) {
546             ipstat.ips_badhlen++;
547             return;
548         }
549         ip = mtod(m, struct ip *);
```

```

550     }
551
552     /*
553     * RFC1122: packets with a multicast source address are
554     * not allowed.
555     */
556     if (IN_MULTICAST(ip->ip_src.s_addr)) {
557         ipstat.ips_badaddr++;
558         goto bad;
559     }
560
561     /* 127/8 must not appear on wire - RFC1122 */
562     if ((ntohl(ip->ip_dst.s_addr) >> IN_CLASSA_NSHIFT) == IN_LOOPBACKNET ||
563         (ntohl(ip->ip_src.s_addr) >> IN_CLASSA_NSHIFT) == IN_LOOPBACKNET) {
564         if ((m->m_pkthdr.rcvif->if_flags & IFF_LOOPBACK) == 0) {
565             ipstat.ips_badaddr++;
566             goto bad;
567         }
568     }
569
570     switch (m->m_pkthdr.csum_flags &
571            ((m->m_pkthdr.rcvif->if_csum_flags_rx & M_CSUM_IPv4) |
572             M_CSUM_IPv4_BAD)) {
573
574     :
575     :
576
577     default:
578         /* Must compute it ourselves. */
579         INET_CSUM_COUNTER_INCR(&ip_swcs);
580         if (in_cksum(m, hlen) != 0)
581             goto bad;
582         break;
583     }
584
585     /* Retrieve the packet length. */
586     len = ntohs(ip->ip_len);
587
588     /*
589     * Check for additional length bogosity
590     */
591     if (len < hlen) {
592         ipstat.ips_badlen++;
593         goto bad;
594     }
595
596     /*
597     * Check that the amount of data in the buffers
598     * is as at least much as the IP header would have us expect.
599     * Trim mbufs if longer than we expect.

```

```

605     * Drop packet if shorter than we expect.
606     */
607     if (m->m_pkthdr.len < len) {
608         ipstat.ips_tooshort++;
609         goto bad;
610     }
611     if (m->m_pkthdr.len > len) {
612         if (m->m_len == m->m_pkthdr.len) {
613             m->m_len = len;
614             m->m_pkthdr.len = len;
615         } else
616             m_adj(m, len - m->m_pkthdr.len);
617     }
618
619     :
620     :
621
622     /*
623     * Process options and, if not destined for us,
624     * ship it on. ip_dooptions returns 1 when an
625     * error was detected (causing an icmp message
626     * to be sent and the original packet to be freed).
627     */
628     ip_nhops = 0;                /* for source routed packets */
629     if (hlen > sizeof (struct ip) && ip_dooptions(m))
630         return;
631
632     /*
633     * Enable a consistency check between the destination address
634     * and the arrival interface for a unicast packet (the RFC 1122
635     * strong ES model) if IP forwarding is disabled and the packet
636     * is not locally generated.
637     *
638     * XXX - Checking also should be disabled if the destination
639     * address is ipnat'ed to a different interface.
640     *
641     * XXX - Checking is incompatible with IP aliases added
642     * to the loopback interface instead of the interface where
643     * the packets are received.
644     *
645     * XXX - We need to add a per ifaddr flag for this so that
646     * we get finer grain control.
647     */
648     checkif = ip_checkinterface && (ipforwarding == 0) &&
649         (m->m_pkthdr.rcvif != NULL) &&
650         ((m->m_pkthdr.rcvif->if_flags & IFF_LOOPBACK) == 0);
651
652     /*
653     * Check our list of addresses, to see if the packet is for us.
654     *

```

```

705     * Traditional 4.4BSD did not consult IFF_UP at all.
706     * The behavior here is to treat addresses on !IFF_UP interface
707     * as not mine.
708     */
709     downmatch = 0;
710     LIST_FOREACH(ia, &IN_IFADDR_HASH(ip->ip_dst.s_addr), ia_hash) {
711         if (in_hosteq(ia->ia_addr.sin_addr, ip->ip_dst)) {
712             if (checkif && ia->ia_ifp != m->m_pkthdr.rcvif)
713                 continue;
714             if ((ia->ia_ifp->if_flags & IFF_UP) != 0)
715                 break;
716             else
717                 downmatch++;
718         }
719     }
720     if (ia != NULL)
721         goto ours;
722     if (m->m_pkthdr.rcvif->if_flags & IFF_BROADCAST) {
723         TAILQ_FOREACH(ifa, &m->m_pkthdr.rcvif->if_addrlist, ifa_list) {
724             if (ifa->ifa_addr->sa_family != AF_INET)
725                 continue;
726             ia = ifatoia(ifa);
727             if (in_hosteq(ip->ip_dst, ia->ia_broadaddr.sin_addr) ||
728                 in_hosteq(ip->ip_dst, ia->ia_netbroadcast) ||
729                 /*
730                  * Look for all-0's host part (old broadcast addr),
731                  * either for subnet or net.
732                  */
733                 ip->ip_dst.s_addr == ia->ia_subnet ||
734                 ip->ip_dst.s_addr == ia->ia_net)
735                 goto ours;
736             /*
737              * An interface with IP address zero accepts
738              * all packets that arrive on that interface.
739              */
740             if (in_nullhost(ia->ia_addr.sin_addr))
741                 goto ours;
742         }
743     }
744     if (IN_MULTICAST(ip->ip_dst.s_addr)) {
745         struct in_multi *inm;
746
747         :
748         :
749
750         /*
751          * See if we belong to the destination multicast group on the
752          * arrival interface.
753          */
754         IN_LOOKUP_MULTI(ip->ip_dst, m->m_pkthdr.rcvif, inm);

```

```

791         if (inm == NULL) {
792             ipstat.ips_cantforward++;
793             m_freem(m);
794             return;
795         }
796         goto ours;
797     }
798     if (ip->ip_dst.s_addr == INADDR_BROADCAST ||
799         in_nullhost(ip->ip_dst))
800         goto ours;
801
802     /*
803     * Not for us; forward if possible and desirable.
804     */
805     if (ipforwarding == 0) {
806         ipstat.ips_cantforward++;
807         m_freem(m);
808     } else {
809         /*
810         * If ip_dst matched any of my address on !IFF_UP interface,
811         * and there's no IFF_UP interface that matches ip_dst,
812         * send icmp unreachable. Forwarding it will result in in-kernel
813         * forwarding loop till TTL goes to 0.
814         */
815         if (downmatch) {
816             icmp_error(m, ICMP_UNREACH, ICMP_UNREACH_HOST, 0, 0);
817             ipstat.ips_cantforward++;
818             return;
819         }
820
821         :
822         :
823
824         ip_forward(m, srcrt);
825     }
826     return;
827
828 ours:
829     /*
830     * If offset or IP_MF are set, must reassemble.
831     * Otherwise, nothing need be done.
832     * (We could look in the reassembly queue to see
833     * if the packet was previously fragmented,
834     * but it's not worth the time; just let them time out.)
835     */
836     if (ip->ip_off & ~htons(IP_DF|IP_RF)) {
837         if (M_READONLY(m)) {
838             if ((m = m_pullup(m, hlen)) == NULL) {
839                 ipstat.ips_toosmall++;
840                 goto bad;
841             }

```

```

892         }
893         ip = mtod(m, struct ip *);
894     }
895
896     /*
897     * Look for queue of fragments
898     * of this datagram.
899     */
900     IPQ_LOCK();
901     hash = IPREASS_HASH(ip->ip_src.s_addr, ip->ip_id);
902     /* XXX LIST_FOREACH(fp, &ipq[hash], ipq_q) */
903     for (fp = LIST_FIRST(&ipq[hash]); fp != NULL;
904         fp = LIST_NEXT(fp, ipq_q)) {
905         if (ip->ip_id == fp->ipq_id &&
906             in_hosteq(ip->ip_src, fp->ipq_src) &&
907             in_hosteq(ip->ip_dst, fp->ipq_dst) &&
908             ip->ip_p == fp->ipq_p)
909             goto found;
910     }
911     fp = 0;
912 found:
913
914     /*
915     * Adjust ip_len to not reflect header,
916     * set ipqe_mff if more fragments are expected,
917     * convert offset of this to bytes.
918     */
919     ip->ip_len = htons(ntohs(ip->ip_len) - hlen);
920     mff = (ip->ip_off & htons(IP_MF)) != 0;
921     if (mff) {
922         /*
923         * Make sure that fragments have a data length
924         * that's a non-zero multiple of 8 bytes.
925         */
926         if (ntohs(ip->ip_len) == 0 ||
927             (ntohs(ip->ip_len) & 0x7) != 0) {
928             ipstat.ips_badfrags++;
929             IPQ_UNLOCK();
930             goto bad;
931         }
932     }
933     ip->ip_off = htons((ntohs(ip->ip_off) & IP_OFFMASK) << 3);
934
935     /*
936     * If datagram marked as having more fragments
937     * or if this is not the first fragment,
938     * attempt reassembly; if it succeeds, proceed.
939     */
940     if (mff || ip->ip_off != htons(0)) {

```

```

942             ipstat.ips_fragments++;
943             ipqe = pool_get(&ipqent_pool, PR_NOWAIT);
944             if (ipqe == NULL) {
945                 ipstat.ips_rcvmemdrop++;
946                 IPQ_UNLOCK();
947                 goto bad;
948             }
949             ipqe->ipqe_mff = mff;
950             ipqe->ipqe_m = m;
951             ipqe->ipqe_ip = ip;
952             m = ip_reass(ipqe, fp, &ipq[hash]);
953             if (m == 0) {
954                 IPQ_UNLOCK();
955                 return;
956             }
957             ipstat.ips_reassembled++;
958             ip = mtod(m, struct ip *);
959             hlen = ip->ip_hl << 2;
960             ip->ip_len = htons(ntohs(ip->ip_len) + hlen);
961         } else
962             if (fp)
963                 ip_freef(fp);
964         IPQ_UNLOCK();
965     }

:
:

1019     /*
1020     * Switch out to protocol's input routine.
1021     */
1022     #if IFA_STATS
1023         if (ia && ip)
1024             ia->ia_ifa.ifa_data.ifad_inbytes += ntohs(ip->ip_len);
1025     #endif
1026     ipstat.ips_delivered++;
1027     {
1028         int off = hlen, nh = ip->ip_p;
1029
1030         (*inetsw[ip_protox[nh]].pr_input)(m, off, nh);
1031         return;
1032     }
1033 bad:
1034     m_freem(m);
1035     return;
1036
1037 badcsum:
1038     ipstat.ips_badsum++;
1039     m_freem(m);
1040 }

```

C ip_forward

NetBSD 2.0.2 sys/netinet/ip_input.c (revision: 1.197.2.1):

```
1817 /*
1818  * Forward a packet.  If some error occurs return the sender
1819  * an icmp packet.  Note we can't always generate a meaningful
1820  * icmp message because icmp doesn't have a large enough repertoire
1821  * of codes and types.
1822  *
1823  * If not forwarding, just drop the packet.  This could be confusing
1824  * if ipforwarding was zero but some routing protocol was advancing
1825  * us as a gateway to somewhere.  However, we must let the routing
1826  * protocol deal with that.
1827  *
1828  * The srcrt parameter indicates whether the packet is being forwarded
1829  * via a source route.
1830  */
1831 void
1832 ip_forward(m, srcrt)
1833     struct mbuf *m;
1834     int srcrt;
1835 {
1836     struct ip *ip = mtod(m, struct ip *);
1837     struct sockaddr_in *sin;
1838     struct rtable *rt;
1839     int error, type = 0, code = 0;
1840     struct mbuf *mcopy;
1841     n_long dest;
1842     struct ifnet *destifp;
1843 #if defined(IPSEC) || defined(FAST_IPSEC)
1844     struct ifnet dummyifp;
1845 #endif
1846
1847     /*
1848      * We are now in the output path.
1849      */
1850     MCLAIM(m, &ip_tx_mowner);
1851
1852     /*
1853      * Clear any in-bound checksum flags for this packet.
1854      */
1855     m->m_pkthdr.csum_flags = 0;
1856
1857     dest = 0;
1858 #ifdef DIAGNOSTIC
1859     if (ipprintfs)
1860         printf("forward: src %2.2x dst %2.2x ttl %x\n",
1861             ntohl(ip->ip_src.s_addr),
1862             ntohl(ip->ip_dst.s_addr), ip->ip_ttl);
```

```

1863 #endif
1864     if (m->m_flags & (M_BCAST|M_MCAST) || in_canforward(ip->ip_dst) == 0) {
1865         ipstat.ips_cantforward++;
1866         m_freem(m);
1867         return;
1868     }
1869     if (ip->ip_ttl <= IPTTLDEC) {
1870         icmp_error(m, ICMP_TIMXCEED, ICMP_TIMXCEED_INTRANS, dest, 0);
1871         return;
1872     }
1873     ip->ip_ttl -= IPTTLDEC;
1874
1875     sin = satoip(&ipforward_rt.ro_dst);
1876     if ((rt = ipforward_rt.ro_rt) == 0 ||
1877         !in_hosteq(ip->ip_dst, sin->sin_addr)) {
1878         if (ipforward_rt.ro_rt) {
1879             RTFREE(ipforward_rt.ro_rt);
1880             ipforward_rt.ro_rt = 0;
1881         }
1882         sin->sin_family = AF_INET;
1883         sin->sin_len = sizeof(struct sockaddr_in);
1884         sin->sin_addr = ip->ip_dst;
1885
1886         rtalloc(&ipforward_rt);
1887         if (ipforward_rt.ro_rt == 0) {
1888             icmp_error(m, ICMP_UNREACH, ICMP_UNREACH_HOST, dest, 0);
1889             return;
1890         }
1891         rt = ipforward_rt.ro_rt;
1892     }
1893
1894     /*
1895     * Save at most 68 bytes of the packet in case
1896     * we need to generate an ICMP message to the src.
1897     * Pullup to avoid sharing mbuf cluster between m and mcopy.
1898     */
1899     mcopy = m_copym(m, 0, imin(ntohs(ip->ip_len), 68), M_DONTWAIT);
1900     if (mcopy)
1901         mcopy = m_pullup(mcopy, ip->ip_hl << 2);
1902
1903     /*
1904     * If forwarding packet using same interface that it came in on,
1905     * perhaps should send a redirect to sender to shortcut a hop.
1906     * Only send redirect if source is sending directly to us,
1907     * and if packet was not source routed (or has any options).
1908     * Also, don't send redirect if forwarding using a default route
1909     * or a route modified by a redirect.
1910     */
1911     if (rt->rt_ifp == m->m_pkthdr.rcvif &&
1912         (rt->rt_flags & (RTF_DYNAMIC|RTF_MODIFIED)) == 0 &&

```

```

1913         !in_nullhost(satosin(rt_key(rt))->sin_addr) &&
1914         ipsendredirects && !srcrt) {
1915             if (rt->rt_ifa &&
1916                 (ip->ip_src.s_addr & ifatoia(rt->rt_ifa)->ia_subnetmask) ==
1917                 ifatoia(rt->rt_ifa)->ia_subnet) {
1918                 if (rt->rt_flags & RTF_GATEWAY)
1919                     dest = satosin(rt->rt_gateway)->sin_addr.s_addr;
1920                 else
1921                     dest = ip->ip_dst.s_addr;
1922                 /*
1923                  * Router requirements says to only send host
1924                  * redirects.
1925                  */
1926                 type = ICMP_REDIRECT;
1927                 code = ICMP_REDIRECT_HOST;
1928 #ifdef DIAGNOSTIC
1929                 if (ipprintfs)
1930                     printf("redirect (%d) to %x\n", code,
1931                         (u_int32_t)dest);
1932 #endif
1933             }
1934         }
1935
1936         error = ip_output(m, (struct mbuf *)0, &ipforward_rt,
1937             (IP_FORWARDING | (ip_directedbcst ? IP_ALLOWBROADCAST : 0)),
1938             (struct ip_moptions *)NULL, (struct socket *)NULL);
1939
1940         if (error)
1941             ipstat.ips_cantforward++;
1942         else {
1943             ipstat.ips_forward++;
1944             if (type)
1945                 ipstat.ips_redirectsent++;
1946             else {
1947                 if (mcopy) {
1948 #ifdef GATEWAY
1949                     if (mcopy->m_flags & M_CANFASTFWD)
1950                         ipflow_create(&ipforward_rt, mcopy);
1951 #endif
1952                     m_freem(mcopy);
1953                 }
1954                 return;
1955             }
1956         }
1957         if (mcopy == NULL)
1958             return;
1959         destifp = NULL;
1960
1961         switch (error) {
1962

```

```

1963         case 0:                                     /* forwarded, but need redirect */
1964             /* type, code set above */
1965             break;
1966
1967         case ENETUNREACH:                             /* shouldn't happen, checked above */
1968         case EHOSTUNREACH:
1969         case ENETDOWN:
1970         case EHOSTDOWN:
1971         default:
1972             type = ICMP_UNREACH;
1973             code = ICMP_UNREACH_HOST;
1974             break;
1975
1976         case EMSGSIZE:
1977             type = ICMP_UNREACH;
1978             code = ICMP_UNREACH_NEEDFRAG;
1979 #if !defined(IPSEC) && !defined(FAST_IPSEC)
1980             if (ipforward_rt.ro_rt)
1981                 destifp = ipforward_rt.ro_rt->rt_ifp;
1982 #else
1983             /*
1984             * If the packet is routed over IPsec tunnel, tell the
1985             * originator the tunnel MTU.
1986             *     tunnel MTU = if MTU - sizeof(IP) - ESP/AH hdrsiz
1987             * XXX quickhack!!!
1988             */
1989             if (ipforward_rt.ro_rt) {
1990                 struct secpolicy *sp;
1991                 int ipsecerror;
1992                 size_t ipsechdr;
1993                 struct route *ro;
1994
1995                 sp = ipsec4_getpolicybyaddr(mcopy,
1996                     IPSEC_DIR_OUTBOUND, IP_FORWARDING,
1997                     &ipsecerror);
1998
1999                 if (sp == NULL)
2000                     destifp = ipforward_rt.ro_rt->rt_ifp;
2001             else {
2002                 /* count IPsec header size */
2003                 ipsechdr = ipsec4_hdrsiz(mcopy,
2004                     IPSEC_DIR_OUTBOUND, NULL);
2005
2006                 /*
2007                 * find the correct route for outer IPv4
2008                 * header, compute tunnel MTU.
2009                 *
2010                 * XXX BUG ALERT
2011                 * The "dummyifp" code relies upon the fact
2012                 * that icmp_error() touches only ifp->if_mtu.

```

```

2013         */
2014         /*XXX*/
2015         destifp = NULL;
2016         if (sp->req != NULL
2017             && sp->req->sav != NULL
2018             && sp->req->sav->sah != NULL) {
2019             ro = &sp->req->sav->sah->sa_route;
2020             if (ro->ro_rt && ro->ro_rt->rt_ifp) {
2021                 dummyifp.if_mtu =
2022                     ro->ro_rt->rt_rmx.rmx_mtu ?
2023                     ro->ro_rt->rt_rmx.rmx_mtu :
2024                     ro->ro_rt->rt_ifp->if_mtu;
2025                 dummyifp.if_mtu -= ipsechdr;
2026                 destifp = &dummyifp;
2027             }
2028         }
2029
2030 #ifdef      IPSEC
2031             key_freesp(sp);
2032 #else
2033             KEY_FREESP(&sp);
2034 #endif
2035         }
2036     }
2037 #endif /*IPSEC*/
2038     ipstat.ips_cantfrag++;
2039     break;
2040
2041     case ENOBUFS:
2042 #if 1
2043         /*
2044          * a router should not generate ICMP_SOURCEQUENCH as
2045          * required in RFC1812 Requirements for IP Version 4 Routers.
2046          * source quench could be a big problem under DoS attacks,
2047          * or if the underlying interface is rate-limited.
2048          */
2049         if (mcopy)
2050             m_freem(mcopy);
2051         return;
2052 #else
2053         type = ICMP_SOURCEQUENCH;
2054         code = 0;
2055         break;
2056 #endif
2057     }
2058     icmp_error(mcopy, type, code, dest, destifp);
2059 }

```

D ip_output

NetBSD 2.0.2 sys/netinet/ip_output.c (revision: 1.130):

```
159 /*
160  * IP output.  The packet in mbuf chain m contains a skeletal IP
161  * header (with len, off, ttl, proto, tos, src, dst).
162  * The mbuf chain containing the packet will be freed.
163  * The mbuf opt, if present, will not be freed.
164  */
165 int
166 #if __STDC__
167 ip_output(struct mbuf *m0, ...)
168 #else
169 ip_output(m0, va_alist)
170         struct mbuf *m0;
171         va_dcl
172 #endif
173 {
174     struct ip *ip;
175     struct ifnet *ifp;
176     struct mbuf *m = m0;
177     int hlen = sizeof (struct ip);
178     int len, error = 0;
179     struct route iproute;
180     struct sockaddr_in *dst;
181     struct in_ifaddr *ia;
182     struct mbuf *opt;
183     struct route *ro;
184     int flags, sw_csum;
185     int *mtu_p;
186     u_long mtu;
187     struct ip_moptions *imo;
188     struct socket *so;
189     va_list ap;
190 #ifdef IPSEC
191     struct secpolicy *sp = NULL;
192 #endif /*IPSEC*/
193 #ifdef FAST_IPSEC
194     struct inpcb *inp;
195     struct m_tag *mtag;
196     struct secpolicy *sp = NULL;
197     struct tdb_ident *tdbi;
198     int s;
199 #endif
200     u_int16_t ip_len;
201
202     len = 0;
203     va_start(ap, m0);
204     opt = va_arg(ap, struct mbuf *);
```

```

205     ro = va_arg(ap, struct route *);
206     flags = va_arg(ap, int);
207     imo = va_arg(ap, struct ip_moptions *);
208     so = va_arg(ap, struct socket *);
209     if (flags & IP_RETURNMTU)
210         mtu_p = va_arg(ap, int *);
211     else
212         mtu_p = NULL;
213     va_end(ap);
214
215     MCLAIM(m, &ip_tx_mowner);
216 #ifdef FAST_IPSEC
217     if (so != NULL && so->so_proto->pr_domain->dom_family == AF_INET)
218         inp = (struct inpcb *)so->so_pcb;
219     else
220         inp = NULL;
221 #endif /* FAST_IPSEC */
222
223 #ifdef DIAGNOSTIC
224     if ((m->m_flags & M_PKTHDR) == 0)
225         panic("ip_output no HDR");
226 #endif
227     if (opt) {
228         m = ip_insertoptions(m, opt, &len);
229         if (len >= sizeof(struct ip))
230             hlen = len;
231     }
232     ip = mtod(m, struct ip *);
233     /*
234     * Fill in IP header.
235     */
236     if ((flags & (IP_FORWARDING|IP_RAWOUTPUT)) == 0) {
237         ip->ip_v = IPVERSION;
238         ip->ip_off = htons(0);
239         ip->ip_id = ip_newid();
240         ip->ip_hl = hlen >> 2;
241         ipstat.ips_localout++;
242     } else {
243         hlen = ip->ip_hl << 2;
244     }
245     /*
246     * Route packet.
247     */
248     if (ro == 0) {
249         ro = &iproute;
250         bzero((caddr_t)ro, sizeof (*ro));
251     }
252     dst = satoSin(&ro->ro_dst);
253     /*
254     * If there is a cached route,

```

```

255     * check that it is to the same destination
256     * and is still up.  If not, free it and try again.
257     * The address family should also be checked in case of sharing the
258     * cache with IPv6.
259     */
260     if (ro->ro_rt && ((ro->ro_rt->rt_flags & RTF_UP) == 0 ||
261         dst->sin_family != AF_INET ||
262         !in_hosteq(dst->sin_addr, ip->ip_dst))) {
263         RTFREE(ro->ro_rt);
264         ro->ro_rt = (struct rtable *)0;
265     }
266     if (ro->ro_rt == 0) {
267         bzero(dst, sizeof(*dst));
268         dst->sin_family = AF_INET;
269         dst->sin_len = sizeof(*dst);
270         dst->sin_addr = ip->ip_dst;
271     }
272     /*
273     * If routing to interface only,
274     * short circuit routing lookup.
275     */
276     if (flags & IP_ROUTETOIF) {
277         if ((ia = ifatoia(ifa_ifwithladdr(sintosa(dst)))) == 0) {
278             ipstat.ips_noroute++;
279             error = ENETUNREACH;
280             goto bad;
281         }
282         ifp = ia->ia_ifp;
283         mtu = ifp->if_mtu;
284         ip->ip_ttl = 1;
285     } else if ((IN_MULTICAST(ip->ip_dst.s_addr) ||
286         ip->ip_dst.s_addr == INADDR_BROADCAST) &&
287         imo != NULL && imo->imo_multicast_ifp != NULL) {
288         ifp = imo->imo_multicast_ifp;
289         mtu = ifp->if_mtu;
290         IFP_TO_IA(ifp, ia);
291     } else {
292         if (ro->ro_rt == 0)
293             rtalloc(ro);
294         if (ro->ro_rt == 0) {
295             ipstat.ips_noroute++;
296             error = EHOSTUNREACH;
297             goto bad;
298         }
299         ia = ifatoia(ro->ro_rt->rt_ifa);
300         ifp = ro->ro_rt->rt_ifp;
301         if ((mtu = ro->ro_rt->rt_rmx.rmx_mtu) == 0)
302             mtu = ifp->if_mtu;
303         ro->ro_rt->rt_use++;
304         if (ro->ro_rt->rt_flags & RTF_GATEWAY)

```

```

305         dst = satosin(ro->ro_rt->rt_gateway);
306     }
307     if (IN_MULTICAST(ip->ip_dst.s_addr) ||
308         (ip->ip_dst.s_addr == INADDR_BROADCAST)) {
309         struct in_multi *inm;
310
311         m->m_flags |= (ip->ip_dst.s_addr == INADDR_BROADCAST) ?
312             M_BCAST : M_MCAST;
313         /*
314          * IP destination address is multicast. Make sure "dst"
315          * still points to the address in "ro". (It may have been
316          * changed to point to a gateway address, above.)
317          */
318         dst = satosin(&ro->ro_dst);
319         /*
320          * See if the caller provided any multicast options
321          */
322         if (imo != NULL)
323             ip->ip_ttl = imo->imo_multicast_ttl;
324         else
325             ip->ip_ttl = IP_DEFAULT_MULTICAST_TTL;
326
327         /*
328          * if we don't know the outgoing ifp yet, we can't generate
329          * output
330          */
331         if (!ifp) {
332             ipstat.ips_noroute++;
333             error = ENETUNREACH;
334             goto bad;
335         }
336
337         /*
338          * If the packet is multicast or broadcast, confirm that
339          * the outgoing interface can transmit it.
340          */
341         if (((m->m_flags & M_MCAST) &&
342             (ifp->if_flags & IFF_MULTICAST) == 0) ||
343             ((m->m_flags & M_BCAST) &&
344             (ifp->if_flags & (IFF_BROADCAST|IFF_POINTOPOINT)) == 0)) {
345             ipstat.ips_noroute++;
346             error = ENETUNREACH;
347             goto bad;
348         }
349         /*
350          * If source address not specified yet, use an address
351          * of outgoing interface.
352          */
353         if (in_nullhost(ip->ip_src)) {
354             struct in_ifaddr *ia;

```

```

355
356         IFP_TO_IA(ifp, ia);
357         if (!ia) {
358             error = EADDRNOTAVAIL;
359             goto bad;
360         }
361         ip->ip_src = ia->ia_addr.sin_addr;
362     }
363
364     IN_LOOKUP_MULTI(ip->ip_dst, ifp, inm);
365     if (inm != NULL &&
366         (imo == NULL || imo->imo_multicast_loop)) {
367         /*
368          * If we belong to the destination multicast group
369          * on the outgoing interface, and the caller did not
370          * forbid loopback, loop back a copy.
371          */
372         ip_mloopback(ifp, m, dst);
373     }
374
375     :
376     :
377
378     /*
379     * Multicasts with a time-to-live of zero may be looped-
380     * back, above, but must not be transmitted on a network.
381     * Also, multicasts addressed to the loopback interface
382     * are not sent -- the above call to ip_mloopback() will
383     * loop back a copy if this host actually belongs to the
384     * destination group on the loopback interface.
385     */
386     if (ip->ip_ttl == 0 || (ifp->if_flags & IFF_LOOPBACK) != 0) {
387         m_freem(m);
388         goto done;
389     }
390
391     goto sendit;
392 }
393 #ifndef notdef
394     /*
395     * If source address not specified yet, use address
396     * of outgoing interface.
397     */
398     if (in_nullhost(ip->ip_src))
399         ip->ip_src = ia->ia_addr.sin_addr;
400 #endif
401
402     /*
403     * packets with Class-D address as source are not valid per
404     * RFC 1112
405     */

```

```

425     */
426     if (IN_MULTICAST(ip->ip_src.s_addr)) {
427         ipstat.ips_odropped++;
428         error = EADDRNOTAVAIL;
429         goto bad;
430     }
431
432     /*
433     * Look for broadcast address and
434     * and verify user is allowed to send
435     * such a packet.
436     */
437     if (in_broadcast(dst->sin_addr, ifp)) {
438         if ((ifp->if_flags & IFF_BROADCAST) == 0) {
439             error = EADDRNOTAVAIL;
440             goto bad;
441         }
442         if ((flags & IP_ALLOWBROADCAST) == 0) {
443             error = EACCES;
444             goto bad;
445         }
446         /* don't allow broadcast messages to be fragmented */
447         if (ntohs(ip->ip_len) > ifp->if_mtu) {
448             error = EMSGSIZE;
449             goto bad;
450         }
451         m->m_flags |= M_BCAST;
452     } else
453         m->m_flags &= ~M_BCAST;
454
455     sendit:
456     /*
457     * If we're doing Path MTU Discovery, we need to set DF unless
458     * the route's MTU is locked.
459     */
460     if ((flags & IP_MTUDISC) != 0 && ro->ro_rt != NULL &&
461         (ro->ro_rt->rt_rmx.rmx_locks & RTV_MTU) == 0)
462         ip->ip_off |= htons(IP_DF);
463
464     /* Remember the current ip_len */
465     ip_len = ntohs(ip->ip_len);
466
467     :
468     :
469
470     m->m_pkthdr.csum_flags |= M_CSUM_IPv4;
471     sw_csum = m->m_pkthdr.csum_flags & ~ifp->if_csum_flags_tx;
472     /*
473     * If small enough for mtu of path, can just send directly.
474     */

```

```

748         if (ip_len <= mtu) {
749 #if IFA_STATS
750             /*
751              * search for the source address structure to
752              * maintain output statistics.
753              */
754             INADDR_TO_IA(ip->ip_src, ia);
755             if (ia)
756                 ia->ia_ifa.ifa_data.ifad_outbytes += ip_len;
757 #endif
758             /*
759              * Always initialize the sum to 0! Some HW assisted
760              * checksumming requires this.
761              */
762             ip->ip_sum = 0;
763
764             /*
765              * Perform any checksums that the hardware can't do
766              * for us.
767              *
768              * XXX Does any hardware require the {th,uh}_sum
769              * XXX fields to be 0?
770              */
771             if (sw_csum & M_CSUM_IPv4) {
772                 ip->ip_sum = in_cksum(m, hlen);
773                 m->m_pkthdr.csum_flags &= ~M_CSUM_IPv4;
774             }
775             if (sw_csum & (M_CSUM_TCPv4|M_CSUM_UDPv4)) {
776                 in_delayed_cksum(m);
777                 m->m_pkthdr.csum_flags &= ~(M_CSUM_TCPv4|M_CSUM_UDPv4);
778             }
779
780 #ifndef IPSEC
781             /* clean ipsec history once it goes out of the node */
782             ipsec_delaux(m);
783 #endif
784             error = (*ifp->if_output)(ifp, m, sintosa(dst), ro->ro_rt);
785             goto done;
786         }
787
788         /*
789          * We can't use HW checksumming if we're about to
790          * to fragment the packet.
791          *
792          * XXX Some hardware can do this.
793          */
794         if (m->m_pkthdr.csum_flags & (M_CSUM_TCPv4|M_CSUM_UDPv4)) {
795             in_delayed_cksum(m);
796             m->m_pkthdr.csum_flags &= ~(M_CSUM_TCPv4|M_CSUM_UDPv4);
797         }

```

```

798
799      /*
800      * Too large for interface; fragment if possible.
801      * Must be able to put at least 8 bytes per fragment.
802      */
803      if (ntohs(ip->ip_off) & IP_DF) {
804          if (flags & IP_RETURNMTU)
805              *mtu_p = mtu;
806          error = EMSGSIZE;
807          ipstat.ips_cantfrag++;
808          goto bad;
809      }
810
811      error = ip_fragment(m, ifp, mtu);
812      if (error) {
813          m = NULL;
814          goto bad;
815      }
816
817      for (; m; m = m0) {
818          m0 = m->m_nextpkt;
819          m->m_nextpkt = 0;
820          if (error == 0) {
821              #if IFA_STATS
822                  /*
823                  * search for the source address structure to
824                  * maintain output statistics.
825                  */
826                  INADDR_TO_IA(ip->ip_src, ia);
827                  if (ia) {
828                      ia->ia_ifa.ifa_data.ifad_outbytes +=
829                          ntohs(ip->ip_len);
830                  }
831              #endif
832              #ifdef IPSEC
833                  /* clean ipsec history once it goes out of the node */
834                  ipsec_delaux(m);
835              #endif
836                  KASSERT((m->m_pkthdr.csum_flags &
837                      (M_CSUM_UDPv4 | M_CSUM_TCPv4)) == 0);
838                  error = (*ifp->if_output)(ifp, m, sintosa(dst),
839                      ro->ro_rt);
840              } else
841                  m_freem(m);
842          }
843
844          if (error == 0)
845              ipstat.ips_fragmented++;
846      done:
847          if (ro == &iproute && (flags & IP_ROUTETOIF) == 0 && ro->ro_rt) {

```

```
848             RTFREE(ro->ro_rt);
849             ro->ro_rt = 0;
850         }
851
852 #ifdef IPSEC
853     if (sp != NULL) {
854         KEYDEBUG(KEYDEBUG_IPSEC_STAMP,
855             printf("DP ip_output call free SP:%p\n", sp));
856         key_freesp(sp);
857     }
858 #endif /* IPSEC */
859 #ifdef FAST_IPSEC
860     if (sp != NULL)
861         KEY_FREESP(&sp);
862 #endif /* FAST_IPSEC */
863
864     return (error);
865 bad:
866     m_freem(m);
867     goto done;
868 }
```