/ipv4/net/ip_output.c

--IP Build and Send:

--IP HEADER

```
380         /* OK, we now build IP header. */
381         iph = (struct iphdr *) skb_push(skb, sizeof(struct iphdr) + (opt ?
opt->optlen : 0));
382         *((__u16 *)iph) = htons((4 << 12) | (5 << 8) | (sk->protinfo.af_inet.tos
& 0xff));
383         iph->tot_len = htons(skb->len);
384         if (ip_dont_fragment(sk, &rt->u.dst))
385                 iph->frag_off = htons(IP_DF);
386         else
387                 iph->frag_off = 0;
                        /*Assigning All the header Fields*/
388         iph->ttl      = sk->protinfo.af_inet.ttl;
389         iph->protocol = sk->protocol;
390         iph->saddr    = rt->rt_src;
391         iph->daddr    = rt->rt_dst;
                        /*Assigning the Buffer for the Header*/
392         skb->nh.iph   = iph;
```

-- IP checksum

```
/* Add an IP checksum. */

308     if (skb->len > rt->u.dst.pmtu)
309             goto fragment;
314         ip_send_check(iph);
315
316         skb->priority = sk->priority;
317         return skb->dst->output(skb);


89 __inline__ void ip_send_check(struct iphdr *iph)
90 {
91         iph->check = 0;
92         iph->check = ip_fast_csum((unsigned char *)iph, iph->ihl);
93 }
```

==========

--ip_queue_xmit() - net/ipv4/ip_output.c.

-- Checks the route:

```
339 int ip_queue_xmit(struct sk_buff *skb)
340 {
341         struct sock *sk = skb->sk;
342         struct ip_options *opt = sk->protinfo.af_inet.opt;
343         struct rtable *rt;
344         struct iphdr *iph;
345
.
.
349         rt = (struct rtable *) skb->dst;
350         if (rt != NULL)
351                 goto packet_routed;
352
353         /* Make sure we can route this packet. */
354         rt = (struct rtable *)__sk_dst_check(sk, 0);



packet_routed:
377         if (opt && opt-> is_strictroute && rt->rt_dst != rt->rt_gateway)
378                 goto no_route;


403 no_route:
404         IP_INC_STATS(IpOutNoRoutes);
405         kfree_skb(skb);
406         return -EHOSTUNREACH; /*Remember ERROR: Destination HOST
unreachable: Comes from here*/
```

-- IP_output: (ipv4/ip_output.c)

--Check for NAT

```
256 int ip_output(struct sk_buff *skb)
257 {
258 #ifdef CONFIG_IP_ROUTE_NAT
259         struct rtable *rt = (struct rtable*)skb->dst;
```

```
260 #endif
261
262         IP_INC_STATS(IpOutRequests);
263
264 #ifdef CONFIG_IP_ROUTE_NAT
265         if (rt->rt_flags&RTCF_NAT)
266                 ip_do_nat(skb);
267 #endif
268
269         return ip_finish_output(skb);
270 }
```

net/ipv4/ip_nat_dumb.c, line 47

```
46 int
47 ip_do_nat(struct sk_buff *skb)
48 {
                                /*Over here encapsulating IP within an IP */
49         struct rtable *rt = (struct rtable*)skb->dst;
50         struct iphdr *iph = skb->nh.iph;
51         u32 odaddr = iph->daddr;
52         u32 osaddr = iph->saddr;
53         u16     check;
54
55         IPCB(skb)->flags |= IPSKB_TRANSLATED;
56
57         /* Rewrite IP header */
58         iph->daddr = rt->rt_dst_map;
59         iph->saddr = rt->rt_src_map;
60         iph->check = 0;
61         iph->check = ip_fast_csum((unsigned char *)iph, iph->ihl);
.
.
.
/*rewriting the some of the protocol headers*/
/*Checking what kind of protocol, if icmp from icmp.c*/
switch(iph->protocol) {
69                 case IPPROTO_TCP:
.
.
.
case IPPROTO_UDP:
.
case IPPROTO_ICMP:
.
.
```

```
.
}
161         return NET_RX_SUCCESS;
}

==========================================================
====


-- ip_finish_output()

184 __inline__ int ip_finish_output(struct sk_buff *skb)
185 {
                                /*Netdevice if the next device, ready to receive
packet*/
186         struct net_device *dev = skb->dst->dev;
187
188         skb->dev = dev;
189         skb->protocol = htons(ETH_P_IP);
190
191         return NF_HOOK(PF_INET, NF_IP_POST_ROUTING, skb, NULL,
dev,
192                         ip_finish_output2);
193 }


/linux/netfilter_ipv4.h
48 /* Packets about to hit the wire. */
49 #define NF_IP_POST_ROUTING      4


/ipv4/ip_output.c
160 static inline int ip_finish_output2(struct sk_buff *skb)
161 {
162         struct dst_entry *dst = skb->dst;
163         struct hh_cache *hh = dst->hh;
.
.
if (hh) {
.
.
173                 skb_push(skb, hh->hh_len);
174                 return hh->hh_output(skb);
175         } else if (dst->neighbour)
176                 return dst->neighbour->output(skb);
.
```

```
}

 ========================================
-- Examines Packet for error:

/net/ipv4/ip_input.c : ip_rcv()


379      int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct
packet_type *pt)
380 {
381         struct iphdr *iph;
382
383         /* When the interface is in promisc. mode, drop all the crap
384          * that it receives, do not try to analyse it.
385          */

                    /*Check: Packet belong to me :Drop the Crap!!*/
386         if (skb->pkt_type == PACKET_OTHERHOST)
387                 goto drop;
.
.
/*
400          *      RFC1122: 3.1.2.2 MUST silently discard any IP frame that fails
the checksum.
401          *
402          *      Is the datagram acceptable?
403          *
404          *      1.      Length at least the size of an ip header
405          *      2.      Version of 4
406          *      3.      Checksums correctly. [Speed optimisation for later, skip
loopback checksums]
407          *      4.      Doesn't have a bogus length
408          */
409
410         if (iph->ihl < 5 || iph->version != 4)
411                 goto inhdr_error;
412
415
.
.
             /*Checks for checksum*/

418         if (ip_fast_csum((u8 *)iph, iph->ihl) != 0)
419                 goto inhdr_error;
420
```

```
421         {
422                 __u32 len = ntohs(iph->tot_len);
423                 if (skb->len < len || len < (iph->ihl<<2))
424                         goto inhdr_error;
.
.
434                 }
435         }
436
437         return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL,
438                        ip_rcv_finish);
439

440 inhdr_error:
441         IP_INC_STATS_BH(IpInHdrErrors);
442 drop:
443         kfree_skb(skb);
444 out:
445         return NET_RX_DROP;
446 }
```

/*packet travel inside linux networking – finishing receive*/
/net/ipv4/ip_input.c

```
306 static inline int ip_rcv_finish(struct sk_buff *skb)
307 {
308         struct net_device *dev = skb->dev;
309         struct iphdr *iph = skb->nh.iph;
310
311         /*
312          *      Initialise the virtual path cache for the packet.
                It describes      how the packet travels inside Linux networking.
314          */

                        /*Passing the packet to the routing routine*/

316                 if (ip_route_input(skb, iph->daddr, iph->saddr, iph->tos, dev))

317                         goto drop;
.
.
```

/*IP Forwarding: Packet Belong to other Host*/

```
367         return skb->dst->input(skb); /*[ = IP Forwarding]*/


371 drop:
372         kfree_skb(skb);
373         return NET_RX_DROP;
374 }
```

Let's Check out Routing: Interesting to know how it is done:

--net/ipv4/route.c, line 1637

```
1637   int ip_route_input(struct sk_buff *skb, u32 daddr, u32 saddr,
1638                 u8 tos, struct net_device *dev)
1639 {
1640         struct rtable * rth;
.
.
1647         read_lock(&rt_hash_table[hash].lock);
1648         for (rth = rt_hash_table[hash].chain; rth; rth = rth->u.rt_next) {
1649               if (rth->key.dst == daddr &&
1650                   rth->key.src == saddr &&
1651                   rth->key.iif == iif &&
1652                   rth->key.oif == 0 &&
.
.
1661                      read_unlock(&rt_hash_table[hash].lock);
1662                      skb->dst = (struct dst_entry*)rth;
/*Checking above the ENTRY if found return or Do Routing*/
1663                      return 0;
1664               }
1665         }


.
1697         }
1698         return ip_route_input_slow(skb, daddr, saddr, tos, dev);
1699 }


.
.
/* Go to ip_route_output_slow() to do major routing functions:*/
```

```
1705 int ip_route_output_slow(struct rtable **rp, const struct rt_key
*oldkey)
1706 {
1707         struct rt_key key;
1708         struct fib_result res;
1709         unsigned flags = 0;
1710         struct rtable *rth;
1711         struct net_device *dev_out = NULL;
1712         unsigned hash;
1713         int free_res = 0;
1714         int err;
1715         u32 tos;
.
.
1803         if (!key.dst) {
1804                 key.dst = key.src;
1805                 if (!key.dst)
1806                         key.dst = key.src = htonl(INADDR_LOOPBACK);
.
.
                 goto make_route;
1815         }
```

/*Start to make New Entry*/
net/ipv4/route.c

```
1886 make_route:
--Checks for like MULTICAST,BROADCAST, NAT

if (key.dst == 0xFFFFFFFF)
1891                 res.type = RTN_BROADCAST;
1892         else if (MULTICAST(key.dst))
1893                 res.type = RTN_MULTICAST;
1894         else if (BADCLASS(key.dst) || ZERONET(key.dst))

/*allocating buffer for Routing enrty*/
1923         rth = dst_alloc(&ipv4_dst_ops);
1924         if (!rth)
1925                 goto e_nobufs;
1926
.
.
```

```
1937        rth->rt_dst     = key.dst;
1938        rth->rt_src     = key.src;
1939 #ifdef CONFIG_IP_ROUTE_NAT (defined:/usr/src/linux/.config file)
1940        rth->rt_dst_map = key.dst;
1941        rth->rt_src_map = key.src;

.
.
 /*calls rt_set_nexthop() to find next destination*/
1978        rt_set_nexthop(rth, &res, 0);
1979
1980        rth->rt_flags = flags;

/*DONE*/}

1994 e_nobufs:
1995        err = -ENOBUFS;
1996        goto done;
```

```
62 struct rtable
63 {
64        union
65        {
66              struct dst_entry       dst;
67              struct rtable          *rt_next;
68        } u;
69
70        unsigned              rt_flags;
71        unsigned              rt_type;
72
73        __u32                 rt_dst; /* Path destination     */
74        __u32                 rt_src; /* Path source         */
75        int                   rt_iif;
76
77        /* Info on neighbour */
78        __u32                 rt_gateway;
79
80        /* Cache lookup keys */
81        struct rt_key         key;
82
83        /* Miscellaneous cached information */
```

```
84        __u32                 rt_spec_dst; /* RFC1122 specific destination */
85        struct inet_peer        *peer; /* long-living peer info */
86
87 #ifdef CONFIG_IP_ROUTE_NAT
88        __u32                 rt_src_map;
89        __u32                 rt_dst_map;
90 #endif
91 };
```

Let's Look into IP_Forwarding:
/net/ipv4/ip_forward.c:

```
73 int ip_forward(struct sk_buff *skb)
74 {
75        struct net_device *dev2;        /* Output device */
76        struct iphdr *iph;      /* Our header */
77        struct rtable *rt;      /* Route we use */
78        struct ip_options * opt = &(IPCB(skb)->opt);
79        unsigned short mtu;
.
.
/*Packet Does not belong to any HOST : drop it*/
84        if (skb->pkt_type != PACKET_HOST)
85              goto drop;
86
87        skb->ip_summed = CHECKSUM_NONE;
88
89        /*
90         *      According to the RFC, we must first decrease the TTL field. If
91         *      that reaches zero, we must reply an ICMP control message
telling
92         *      that the packet's lifetime expired.
93         */
94
95        iph = skb->nh.iph;
                              /*Check TTL*/

96        rt = (struct rtable*)skb->dst;
97
98        if (iph->ttl <= 1)
99              goto too_many_hops;
100
```

```
160 too_many_hops:
161         /* Tell the sender its packet died... */
162         icmp_send(skb, ICMP_TIME_EXCEEDED, ICMP_EXC_TTL, 0);
                    /*Defined in net/ipv4/icmp.c, line 391 */

<=
101         if (opt->is_strictroute && rt->rt_dst != rt->rt_gateway)
102               goto sr_failed; /*send ICMP*/
103
104         /*
105          *      Having picked a route we can now send the frame out
106          *      after asking the firewall permission to do so.
107          */
.
110         dev2 = rt->u.dst.dev;
111         mtu = rt->u.dst.pmtu;
.
.
126         ip_decrease_ttl(iph);

=>
/*Defined in include/net/ip.h, line 173 , Just decreasing ttl Field in IP header,
Interesting!!*/
172 static inline
173 int ip_decrease_ttl(struct iphdr *iph)
174 {
.
.
178         return --iph->ttl;
179 }
*****************************

153 sr_failed:
154         /*
155          *      Strict routing permits no gatewaying
156          */
157         icmp_send(skb, ICMP_DEST_UNREACH, ICMP_SR_FAILED, 0);
158         goto drop;
159
```

Let's Look into ARP:
/net/ipv4/arp.c:

```
487 void arp_send(int type, int ptype, u32 dest_ip,
488             struct net_device *dev, u32 src_ip,
489             unsigned char *dest_hw, unsigned char *src_hw,
```

```
490             unsigned char *target_hw)
491 {
492         struct sk_buff *skb;
493         struct arphdr *arp;
494         unsigned char *arp_ptr;
495
496         /*
497          *      No arp on this interface.
498          */
499
500         if (dev->flags&IFF_NOARP)
501                 return;

503         /*
504          *      Allocate a buffer
505          */
506
507         skb = alloc_skb(sizeof(struct arphdr)+ 2*(dev->addr_len+4)
508                         + dev->hard_header_len + 15, GFP_ATOMIC);
509                       if ( skb == NULL)
510                         return;
.
                        /*Fills in ARP Header*/
514         arp = (struct arphdr *) skb_put(skb,sizeof(struct arphdr) + 2*(dev-
>addr_len+4));
515         skb->dev = dev;
516         skb->protocol = htons (ETH_P_ARP);
517         if (src_hw == NULL)
518                 src_hw = dev->dev_addr;
519         if (dest_hw == NULL)
520                 dest_hw = dev->broadcast;
.
.
539         switch (dev->type) {
540         default:
541                 arp->ar_hrd = htons(dev->type);
542                 arp->ar_pro = htons(ETH_P_IP);
543                 break;
544 /*Checking for the device type*/
545 #if defined(CONFIG_AX25) || defined(CONFIG_AX25_MODULE)
546         case ARPHRD_AX25:
547                 arp->ar_hrd = htons(ARPHRD_AX25);
548                 arp->ar_pro = htons(AX25_P_IP);
549                 break;
550
551 #if defined(CONFIG_NETROM) || defined(CONFIG_NETROM_MODULE)
```

```
552         case ARPHRD_NETROM:
553                 arp->ar_hrd = htons(ARPHRD_NETROM);
554                 arp->ar_pro = htons(AX25_P_IP);
555                 break;
556 #endif
557 #endif


590         /* Send it off, maybe filter it using firewalling first.  */

591         NF_HOOK(NF_ARP, NF_ARP_OUT, skb, NULL, dev,
dev_queue_xmit);
592         return;
```