Input Routing

Determining the next hop with *ip_route_input()*

The *ip_route_input()* function is defined in net/ipv4/route.c. If first tries to find a suitable destination structure in the route cache and if that fails it invokes *ip_route_input_slow()* to perform a FIB lookup. For input routing the *iif* is the *actual* interface upon which the packet arrived and the *oif* is coerced to 0.

```
1622 int ip_route_input(struct sk_buff *skb, u32 daddr, u32
              saddr, u8 tos, struct net_device *dev)
1624 {
1625     struct rtable * rth;
1626     unsigned        hash;
1627     int iif = dev->ifindex;
1628
1629     tos &= IPTOS_RT_MASK;
```

The *rt_hash_code()* function returns the hash code that is used as an index into the route cache.

```
1630     hash = rt_hash_code(daddr, saddr ^ (iif << 5), tos);
```

The hash function is implemented by the inline function *rt_hash_code()*. The code is derived from the source and destination addresses, the input interface index and the type of service.

```
 203 static __inline__ unsigned rt_hash_code(u32 daddr,
                                      u32 saddr, u8 tos)
 204 {
 205     unsigned hash = ((daddr & 0xF0F0F0F0) >> 4) |
 206                    ((daddr & 0x0F0F0F0F) << 4);
 207     hash ^= saddr ^ tos;
 208     hash ^= (hash >> 16);
 209     return (hash ^ (hash >> 8)) & rt_hash_mask;
 210 }
```

Route cache lookup

The *hash* code returned by the above function is used by *ip_route_input* to identify the proper chain in the *rt_hash_table* structure. In contrast to output routing, the *oif* is forced to zero here and the *iif* on which the packet arrived is used. The *tos* value used here is the *tos* that was carried in the IP header of the arriving packet.

```
1632     read_lock(&rt_hash_table[hash].lock);
1633     for (rth = rt_hash_table[hash].chain; rth; rth =
              rth->u.rt_next) {
1634          if (rth->key.dst == daddr &&
1635               rth->key.src == saddr &&
1636               rth->key.iif == iif &&
1637               rth->key.oif == 0 &&
1638 #ifdef CONFIG_IP_ROUTE_FWMARK
1639               rth->key.fwmark == skb->nfmark &&
1640 #endif
1641               rth->key.tos == tos) {
```

Route cache hit

On finding a match, the time of last use for this entry is updated. The *dst_hold()* function simply increments the reference count *(atomic_inc(&dst->__refcnt))*

```
1642               rth->u.dst.lastuse = jiffies;
1643               dst_hold(&rth->u.dst);
1644               rth->u.dst.__use++;
1645               rt_cache_stat[smp_processor_id()].in_hit++;
1646               read_unlock(&rt_hash_table[hash].lock);
```

Set *skb->dst* to this entry and return.

```
1647               skb->dst = (struct dst_entry*)rth;
1648               return 0;
1649          }
1650     }
```

Falling out of the loop means a route couldn't be found in the route cache.

```
1651     read_unlock(&rt_hash_table[hash].lock);
```

Receipt of mulitcast packet

Reaching this point in *ip_route_input()* implies that a suitable routing element was not present in the route cache. If the destination is a multicast address, it is necessary to determine whether the interface on which this packet was received belongs to this multicast group. The comment below describes how multicast routing is complicated by broken or deficient multicast filters on many ethernet cards.

```
1653     /*   Multicast recognition logic is moved from route
              cache to here.  The problem was that too many
              Ethernet cards have broken/missing hardware
              multicast filters :-( As result the host on
              multicasting network acquires a lot of useless
              route cache entries, sort of SDR messages from all
              the world. Now we try to get rid of them. Really,
              provided software IP multicast filter is organized
              reasonably (at least, hashed), it does not result
              in a slowdown comparing with route cache reject
              entries. Note, that multicast routers are not
              affected, because route cache entry is  created
              eventually.
1663      */
1664     if (MULTICAST(daddr)) {
1665          struct in_device *in_dev;
1666
1667          read_lock(&inetdev_lock);
```

Each *net_device* that supports IP traffic must also an associate *struct in_device.* The *__in_dev_get()* function returns its address.

```
1668          if ((in_dev = __in_dev_get(dev)) != NULL) {
```

The *ip_check_mc()* function, defined in net/ipv4/igmp.c, returns true if the interface is a member of the multicast group identified by *daddr.*

```
1669               int our = ip_check_mc(in_dev, daddr);
```

Handling mulitcast inputs

If the the destination was a multicast address and the interface was a member of the associated group and several configuration constraints are met, then the packet is sent to *ip_route_input_mc()* for routing.CONFIG_IP_MROUTE is an option to allow routing of IP packets that have several destination addresses. IN_DEV_MFORWARD is a macro defined in include/linux/inetdevice.h.

```
  40 #define IN_DEV_MFORWARD(in_dev)(ipv4_devconf.mc_forwarding
                            && (in_dev)->cnf.mc_forwarding)

1670               if (our
1671 #ifdef CONFIG_IP_MROUTE
1672                   || (!LOCAL_MCAST(daddr)
                            && IN_DEV_MFORWARD(in_dev))
1673 #endif
1674                   ){
1675                        read_unlock(&inetdev_lock);
1676                        return ip_route_input_mc(skb,
                                          daddr, saddr,
1677                                             tos, dev, our);
1678               }
1679          }
1680          read_unlock(&inetdev_lock);
1681          return -EINVAL;
1682     }
```

Route cache miss

Reaching this point implies the packet was routeable neither through the routing cache nor as a multicast. The *ip_route_input_slow()* function must be called to try to route via the FIB.

```
1683     return ip_route_input_slow(skb, daddr, saddr, tos, dev);
1684 }
```

Establishing multicast membership

The *mc_list* element of *struct in_device* points to a linked list of the *ip_mc_list* structures that describes the multicast groups of which the network interface is a member.[1]

```
761 int ip_check_mc(struct in_device *in_dev, u32 mc_addr)
762 {
763     struct ip_mc_list *im;
765     read_lock(&in_dev->lock);
766     for (im=in_dev->mc_list; im; im=im->next) {
767          if (im->multiaddr == mc_addr) {
768               read_unlock(&in_dev->lock);
769               return 1;
770          }
771     }
772     read_unlock(&in_dev->lock);
773     return 0;
774 }
```

1 http://www.tldp.org/HOWTO/Multicast-HOWTO-7.html

Input Routing Via the FIB

When a suitable route cache entry is not found, the *ip_route_input_slow()* function, defined in *net/ipv4/route.c*, attempts to find a FIB entry that can be used. If it succeeds, a new route cache entry will have been created. The organization of this function bears resemeblence to some Fortran code written by the writer of these notes in the mid 1960's.

```
     /*
          NOTE. We drop all the packets that has local source
          addresses because every properly looped back packet must
          have correct destination already attached by output
          routine.

          Such approach solves two big problems:
               1. Not simplex devices are handled properly.
               2. IP spoofing attempts are filtered with 100% of
               guarantee.
     */

 1312 int ip_route_input_slow(struct sk_buff *skb, u32
          daddr, u32 saddr, u8 tos, struct net_device *dev)
 1314 {
 1315     struct rt_key        key;
 1316     struct fib_result    res;
 1317     struct in_device     *in_dev = in_dev_get(dev);
 1318     struct in_device     *out_dev = NULL;
 1319     unsigned             flags = 0;
 1320     u32                  itag = 0;
 1321     struct rtable        *rth;
 1322     unsigned             hash;
 1323     u32                  spec_dst;
 1324     int                  err = -EINVAL;
 1325     int                  free_res = 0;
 1326
```

If IP is not supported on the *net_device* on which the packet arrived, then the packet must be dropped.

```
 1327     /* IP on this device is disabled. */
 1328
 1329     if (!in_dev)
 1330          goto out;
```

Route key construction

A *key* is constructed for lookup into the FIB. Note that *key.oif* is coerced to 0 here just as *key.iif* was in output routing. The *scope* is unconditionally set to RT_SCOPE_UNIVERSE, and the values of *tos, saddr,* and *daddr* are obtained from the IP header of the input packet being routed.

```
1332      key.dst           = daddr;
1333      key.src           = saddr;
1334      key.tos           = tos;
1335 #ifdef CONFIG_IP_ROUTE_FWMARK
1336      key.fwmark        = skb->nfmark;
1337 #endif
1338      key.iif           = dev->ifindex;
1339      key.oif           = 0;
1340      key.scope         = RT_SCOPE_UNIVERSE;
```

Hash key construction

A hash value is derived from the destination address, source address, input interface index and type of service. Note that the value of *hash* is used for cache lookups and should not be confused with the value of *key* which is used for FIB lookups. The value computed here *is not used until near the end of the routine* where it is used to identify the proper hash queue into which to insert a the newly created *struct rtable* entry.

```
1342      hash = rt_hash_code(daddr, saddr ^ (key.iif << 5),
                               tos);
```

Source address filtering (phase 1)

When the source address is a multicast/badclass/loopback address, an error is returned straightaway. The term *martian* is commonly used to refer to an IP address that appears to be defective or spoofed in some way.

Exercise for final exam: Describe how a packet whose source and dest IP address are a local ethernet address is routed.

```
1343
1344      /*   Check for the most weird martians, which can
               be not detected by fib_lookup.
1346       */
1347
1348      if (MULTICAST(saddr) || BADCLASS(saddr) ||
               LOOPBACK(saddr))
1349           goto martian_source;
```

Broadcast destination addresses

If the packet has a broadcast destination address, a jump is taken to the broadcast input handler. When both source and destination addresses are NULL, the packet is considered to have been broadcast. Hosts using *bootp* or *dhcp* may have to use a source address of 0, but it is not clear why they would need to use a 0 destination as opposed to the standard broadcast address 0xffffffff.

```
1351      if (daddr == 0xFFFFFFFF || (saddr == 0 && daddr == 0))
1352              goto brd_input;
```

Additional source filtering

Zero valued source addresses are invalid unless the destination is also zero (in which case the packet was already handled as a broadcast).

```
1354      /*   Accept zero addresses only to limited broadcast;
               I even do not know to fix it or not. Waiting for
               complains :-)

          */
1357      if (ZERONET(saddr))
1358              goto martian_source;
```

Destination address filtering

When the destination is a badclass/loopback/zeronet address, an error is also returned. A *martian_destination* is *not* an unknown destination. It is a semantically illegal destination. Unknown destinations are represented by route cache entries but *martians* are not.

```
1360      if (BADCLASS(daddr) || ZERONET(daddr)||LOOPBACK(daddr))
1361              goto martian_destination;
```

FIB lookup

After the source and destination addresses are validated, the FIB is searched in an attempt to resolve the key constructed earlier. Recall that when class based routing is not in effect, that the *fib_lookup()* function attempts a lookup in both the *local* and *main* tables in that order. If the *local* table lookup succeeds the *main* table will not be searched. The following are the criteria for success in lookup.

- `key->dst == fn->fn_key` with respect to the zone's prefix length
- `f->fn_state & FN_S_ZOMBIE) == 0`
- `f->fn_scope >= key->scope` which is RT_SCOPE_UNIVERSE (0) here.
- `fi->fib_flags & RTNH_F_DEAD == 0`
- `nh->nh_flags & RTNH_F_DEAD == 0`
- `!key->oif || key->oif == nh->nh_oif` and *key->oif == 0* here

```
1363     /*
1364      *      Now we are ready to route packet.
1365      */
1366     if ((err = fib_lookup(&key, &res)) != 0) {
```

FIB lookup failure

If the FIB lookup fails, the routing process must be aborted and the packet dropped. If the input device is not configured to support forwarding, there is nothing more to be done. *If forwarding is enabled, a jump to no_route is taken* where an entry with type set to RTN_UNREACHABLE is added to the routing cache. This action will make it *unnecessary to repeat the FIB lookup* in the likely case of the arrival of additional unrouteable packets with the same destination.

```
1367          if (!IN_DEV_FORWARD(in_dev))
1368               goto e_inval;
1369          goto no_route;
1370     }
1371     free_res = 1;  /* remember to free the res struct */
1372
```

A per processor count of routes resolved in the FIB is maintained and is incremented here.

```
1373     rt_cache_stat[smp_processor_id()].in_slow_tot++;
```

NAT lookup

CONFIG_IP_ROUTE_NAT is an option to enable fast network address translation. We do not consider the details of NAT support here.

```
1375 #ifdef CONFIG_IP_ROUTE_NAT
         /*   Policy is applied before mapping destination,
              but rerouting after map should be made with
              old source.
          */

1380     if (1) {
1381          u32 src_map = saddr;
1382          if (res.r)
1383               src_map = fib_rules_policy(saddr, &res,
                                                             &flags);
1384
1385          if (res.type == RTN_NAT) {
1386               key.dst = fib_rules_map_destination(daddr,
                              &res);
1387               fib_res_put(&res);
1388               free_res = 0;
1389               if (fib_lookup(&key, &res))
1390                    goto e_inval;
1391               free_res = 1;
1392               if (res.type != RTN_UNICAST)
1393                    goto e_inval;
1394               flags |= RTCF_DNAT;
1395          }
1396          key.src = src_map;
1397     }
1398 #endif
```

Successful FIB lookups

Reaching this point indicates that the FIB lookup succeded. Therefore, delivery to the destination is thought to be possible, but before delivery can take place several tests involving the RTN and the legitimacy of the source address must be performed. If result is of type RTN_BROADCAST, the packet is processed as a broadcast directed to this system.

```
1400      if (res.type == RTN_BROADCAST)
1401           goto brd_input;
```

Source address filtering (phase 3)

If the result is of type RTN_LOCAL , the packet is destined for this host. However, the FIB is used to validate source address before the packet is accepted. The objective here is to determine if this host has a route back to the the source.

The *fib_validate_source()* function returns

- a negative value if no route exists to *saddr,*
- a positive value if *saddr* is on *this LAN*, and
- 0 if s*addr* is routeable but uses a gatewayed route.

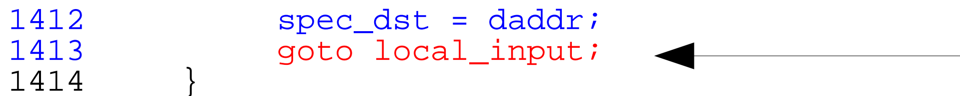
```
1403      if (res.type == RTN_LOCAL) {
1404           int result;
1405           result = fib_validate_source(saddr, daddr,
                              tos, loopback_dev.ifindex,
                              dev, &spec_dst, &itag);
1408           if (result < 0)
1409                goto martian_source;
```

When the value of *result* is positive, the scope of the sender is RT_SCOPE_HOST. This indicates that the source address is on this LAN.

```
1410            if (result)
1411                flags |= RTCF_DIRECTSRC;
```

The broken *spec_dst* address parameter

Note that the value *spec_dst* that is filled in by *fib_validate_source* is not used. Instead *spec_dst* is unconditionally set to *daddr* which was passed as an input parameter to this routine. This action is in accordance with RFC 791 Section 3.2.[2] which states that: ``The specific-destination address is defined to be the destination address in the IP header unless the header contains a broadcast or multicast address, in which case the specific-destination is an IP address assigned to the physical interface on which the datagram arrived."



```
1412            spec_dst = daddr;
1413            goto local_input;
1414        }
```

2 http://www.freesoft.org/CIE/RFC/1122/34.htm

Forwarding packets

At this point it is ensured that the final destination of this packet is not on this host. If the interface on which it is arrived is not configured for forwarding, then the packet must be dropped.

```
1416      if (!IN_DEV_FORWARD(in_dev))
1417           goto e_inval;
```

For the packet to be forwarded it is necessary that the route type be RTN_UNICAST. Other route types (e.g., RTN_BLACKHOLE) also cause the packet to be dropped.

```
1418      if (res.type != RTN_UNICAST)
1419           goto martian_destination;
```

CONFIG_IP_ROUTE_MULTIPATH is an option that may be used to specify several alternative paths for certain packets. The *fib_select_multipath()* function considers all these paths to be of equal cost and chooses one of them in a non-deterministic fashion when a packet is to be routed.

```
1421 #ifdef CONFIG_IP_ROUTE_MULTIPATH
1422      if (res.fi->fib_nhs > 1 && key.oif == 0)
1423           fib_select_multipath(&key, &res);
1424 #endif
```

Recovering the *net_device* for the outgoing next hop

A pointer to the *struct in_device* associated with the output *struct net_device* onto which the packet is to be forwarded is obtained here.

The FIB_RES_DEV(*res*) macro, *res.fi->fib_nh[(res).nh_sel].nh_dev,* extracts the *struct net_device* pointer from the next hop array of type *struct fib_nh* that is embedded in the *struct fib_info* associated with the selected route.

```
1425      out_dev = in_dev_get(FIB_RES_DEV(res));
1426      if (out_dev == NULL) {
1427           if (net_ratelimit())
1428                printk(KERN_CRIT "Bug in
           ip_route_input_slow(). "
                          "Please, report\n");
1430           goto e_inval;
1431      }
```

Source validation before forwarding

As seen earlier in the case of local delivery, it is necessary to validate the source address before forwarding the packet.

```
1433      err = fib_validate_source(saddr, daddr, tos,
                                    FIB_RES_OIF(res), dev,
1434                                     &spec_dst, &itag);
1435      if (err < 0)
1436           goto martian_source;
1437
```

Also as noted previously, if the value returned by *fib_validate_source()* is positive, then the source IP address is directly reachable on this LAN.

```
1438      if (err)
1439           flags |= RTCF_DIRECTSRC;
```

ICMP redirects

When a packet is to be retransmitted on the interface upon which it was received (and some additional constraints are met), the RTCF_DOREDIRECT flag is set in the routing cache element. Setting this flag causes an ICMP redirect packet to be returned to the system that originated the packet.

```
1441      if (out_dev == in_dev && err && !(flags &
                    (RTCF_NAT | RTCF_MASQ)) &&
1442                (IN_DEV_SHARED_MEDIA(out_dev) ||
1443                inet_addr_onlink(out_dev, saddr,
                         FIB_RES_GW(res))))
1444           flags |= RTCF_DOREDIRECT;
```

The function *ip_route_input_slow()* might also be called from *arp_rcv().* Hence the protocol type is verified before creating a routing cache entry.

```
1446      if (skb->protocol != __constant_htons(ETH_P_IP)) {
          /*   Not IP (i.e. ARP). Do not create route,
               if it is invalid for proxy arp. DNAT
               routes are always valid.
1449      */
1450           if (out_dev == in_dev && !(flags & RTCF_DNAT))
1451              goto e_inval;
1452      }
```

Creation of the new route cache entry for forwarded packets

Allocate a routing cache destination entry (*struct dst_entry*) for the packet to be forwarded.

```
1454      rth = dst_alloc(&ipv4_dst_ops);
1455      if (!rth)
1456           goto e_nobufs;
```

On return to *ip_route_input_slow()*, initialization of the destination entry is completed. Recall that the *struct rtable* is a union which consists of a *struct dst_entry* and a *struct rtable **. Therefore the *struct dst_entry* pointer returned by *dst_alloc()* may be interchangeably used as a *struct rtable* pointer.

```
1458      atomic_set(&rth->u.dst.__refcnt, 1);
1459      rth->u.dst.flags= DST_HOST;
1460      rth->key.dst     = daddr;
1461      rth->rt_dst      = daddr;
1462      rth->key.tos     = tos;
1463 #ifdef CONFIG_IP_ROUTE_FWMARK
1464      rth->key.fwmark = skb->nfmark;
1465 #endif
1466      rth->key.src     = saddr;
1467      rth->rt_src      = saddr;
1468      rth->rt_gateway = daddr;
1469 #ifdef CONFIG_IP_ROUTE_NAT
1470      rth->rt_src_map = key.src;
1471      rth->rt_dst_map = key.dst;
1472      if (flags&RTCF_DNAT)
1473           rth->rt_gateway = key.dst;
1474 #endif
1475      rth->rt_iif      =
1476      rth->key.iif     = dev->ifindex;
1477      rth->u.dst.dev   = out_dev->dev;
1478      dev_hold(rth->u.dst.dev);
1479      rth->key.oif     = 0;
1480      rth->rt_spec_dst= spec_dst;
```

Setting up handlers for forwarded packets

Since this section of the code is processing packets forwarded packets for which this host is an intermediate host, the *input* function pointer of the destination entry is set to *ip_forward()* and the output function pointer is set to *ip_output()*. It should be remembered that this routing cache element is being set up to facilitate the processing of *future* packets.

```
1482      rth->u.dst.input = ip_forward;
1483      rth->u.dst.output = ip_output;
1484
1485      rt_set_nexthop(rth, &res, itag);
```

On return from *rt_set_next_hop(),* the routing of the packet to be forwarded concludes here in *ip_route_input_slow().*

```
1487      rth->rt_flags = flags;
```

CONFIG_NET_FASTROUTE is an option to allow direct NIC-to-NIC data transfer on a local network, which is fast.

```
1489 #ifdef CONFIG_NET_FASTROUTE
1490      if (netdev_fastroute &&
               !(flags&(RTCF_NAT|RTCF_MASQ|
               RTCF_DOREDIRECT))) {
1491           struct net_device *odev = rth->u.dst.dev;
1492           if (odev != dev &&
1493                     dev->accept_fastpath &&
1494                     odev->mtu >= dev->mtu &&
1495                     dev->accept_fastpath(dev,
               &rth->u.dst) == 0)
1496                rth->rt_flags |= RTCF_FAST;
1497      }
1498 #endif
```

The exit code

Finally, the newly constructed entry is added to routing cache, the FIB table and any device references held are released, and a return is made to the caller. Jumps are made *back to done:* from several spots in the code.

```
1500 intern:
1501     err = rt_intern_hash(hash, rth,
                                  (struct rtable**)&skb->dst);
1502 done:
1503     in_dev_put(in_dev);
1504     if (out_dev)
1505          in_dev_put(out_dev);
1506     if (free_res)
1507          fib_res_put(&res);
1508 out: return err;
```

Handling of broadcast messages destined for this host.

A jump from line *1401* to the tag *brd_input* was effected when it was determined that the packet carried a legitimate broadcast address. Here the protocol type and source address are validated and specific, local destination address is obtained. As noted earlier, if *fib_validate_source()* returns a positive value this host owns the source address. Throughout this function, *dev* points to the *net_device* on which the packet arrived.

```
1510 brd_input:
1511     if (skb->protocol !=__constant_htons(ETH_P_IP))
1512          goto e_inval;

1514     if (ZERONET(saddr))
1515          spec_dst = inet_select_addr(dev, 0, RT_SCOPE_LINK);
1516     else {
1517          err = fib_validate_source(saddr, 0, tos, 0,
                       dev, &spec_dst, &itag);
1519          if (err < 0)
1520               goto martian_source;
1521          if (err)
1522               flags |= RTCF_DIRECTSRC;
1523     }

1524     flags |= RTCF_BROADCAST;
1525     res.type = RTN_BROADCAST;
1526     rt_cache_stat[smp_processor_id()].in_brd++;
```

Handling of unicast *and broadcast* packets destined for this host.

The handling of broadcast packets merges here with the handling of unicast packets destined for this host. As noted previously, in the allocation of a new routing cache element, the void pointer returned by *dst_alloc()* may be interchangeably used as a pointer to either *struct rtable* or the embedded *struct dst_entry.*

```
1528 local_input:
1529     rth = dst_alloc(&ipv4_dst_ops);
1530     if (!rth)
1531             goto e_nobufs;
1532
```

Since the destination is now known to be this host the *output* function in the cache entry is set to *ip_rt_bug().*

```
1533     rth->u.dst.output= ip_rt_bug;
1534
1535     atomic_set(&rth->u.dst.__refcnt, 1);
1536     rth->u.dst.flags= DST_HOST;
1537     rth->key.dst     = daddr;
1538     rth->rt_dst      = daddr;
1539     rth->key.tos     = tos;

1540 #ifdef CONFIG_IP_ROUTE_FWMARK
1541     rth->key.fwmark = skb->nfmark;
1542 #endif
1543     rth->key.src     = saddr;
1544     rth->rt_src      = saddr;
1545 #ifdef CONFIG_IP_ROUTE_NAT
1546     rth->rt_dst_map = key.dst;
1547     rth->rt_src_map = key.src;
1548 #endif
1549 #ifdef CONFIG_NET_CLS_ROUTE
1550         rth->u.dst.tclassid = itag;
1551 #endif
1552     rth->rt_iif      =
1553     rth->key.iif     = dev->ifindex;
1554     rth->u.dst.dev   = &loopback_dev;
1555     dev_hold(rth->u.dst.dev);
1556     rth->key.oif     = 0;
1557     rth->rt_gateway = daddr;
1558     rth->rt_spec_dst= spec_dst;
```

Setting up the packet handler

Finally, if the FIB lookup did not return a *fib_result* with route type set to *RTN_UNREACHABLE* , the *input* member of the destination entry is set to *ip_local_deliver()*. It is not intuitively clear why a destination address owned by this system would ever be considered unreachable. However, code on the next page indicates that when an unknown destination is encountered, a routing cache entry bearing a local address is created and the route type is set to unreachable.

```
1559     rth->u.dst.input= ip_local_deliver;
1560     rth->rt_flags = flags | RTCF_LOCAL;
1561     if (res.type == RTN_UNREACHABLE) {
1562          rth->u.dst.input= ip_error;
1563          rth->u.dst.error= -err;
1564          rth->rt_flags   &= ~RTCF_LOCAL;
1565         }
1566     rth->rt_type     = res.type;
1567     goto intern; /* Jump back to line 1500 */
```

Handling FIB Lookup Failure

A jump to *no_route* occurs if forwarding is enabled on the input device on which the packet arrived but the destination address is not found in the FIB. The *inet_select_addr()* function is called to obtain an IP address associated with the device on which the packet arrived. The *dst* parameter is set to *0* and the *scope* parameter set to *RT_SCOPE_UNIVERSE.* In this case *inet_select_addr()* will just return the IP address of the first configured interface associated with the *struct net_device.*

```
1569 no_route:
1570     rt_cache_stat[smp_processor_id()].in_no_route++;
1571     spec_dst = inet_select_addr(dev, 0,
                              RT_SCOPE_UNIVERSE);
```

A jump back to *local_input* is now made for the purpose of setting up an unreachable destination entry in the routing cache. The packet will eventually be dropped by *ip_error*.

```
1572     res.type = RTN_UNREACHABLE;
1573     goto local_input;
```

Handling Martian Destination Addresses

According to RFC 1812 invalid (martian) destination addresses should be logged and not added to the routing cache. The device name, destination address and source address of the packet are logged and -EINVAL is returned.

```
1578 martian_destination:
1579     rt_cache_stat[smp_processor_id()].in_martian_dst++;
1580 #ifdef CONFIG_IP_ROUTE_VERBOSE
1581     if (IN_DEV_LOG_MARTIANS(in_dev) &&
             net_ratelimit())
1582             printk(KERN_WARNING "martian destination
             %u.%u.%u.%u from "
1583                     "%u.%u.%u.%u, dev %s\n",
1584                     NIPQUAD(daddr), NIPQUAD(saddr),
             dev->name);
1585 #endif

1586 e_inval:
1587         err = -EINVAL;
1588         goto done;

1590 e_nobufs:
1591         err = -ENOBUFS;
1592         goto done;
```

Handling Martian Source Addresses

When an invalid source address is detected the device name, destination address, source address and "hardware header" of a packet are logged. The *net_ratelimit()* constraints the rate at which the messages are logged.

```
1594 martian_source:
1595
1596      rt_cache_stat[smp_processor_id()].in_martian_src++;

1597 #ifdef CONFIG_IP_ROUTE_VERBOSE
1598     if (IN_DEV_LOG_MARTIANS(in_dev) &&
             net_ratelimit()) {
         /*
             RFC1812 recommendation, if source is martian,
             the only hint is MAC header.
          */
1603         printk(KERN_WARNING "martian source
                       %u.%u.%u.%u from "
1604                   "%u.%u.%u.%u, on dev %s\n",
1605                   NIPQUAD(daddr), NIPQUAD(saddr),
                       dev->name);
1606         if (dev->hard_header_len) {
1607              int i;
1608              unsigned char *p = skb->mac.raw;
1609              printk(KERN_WARNING "ll header: ");
1610              for (i = 0; i < dev->hard_header_len;
                       i++, p++) {
1611                   printk("%02x", *p);
1612                   if (i < (dev->hard_header_len - 1))
1613                        printk(":");
1614                   }
1615                   printk("\n");
1616              }
1617         }
1618 #endif
1619     goto e_inval;
1620 }
```

Allocation of the new route cache entry

The *dst_alloc()* function is defined in net/core/dev.c. The parameter, *ipv4_dst_ops,* is declared and initialized in net/ipv4/route.c.

```
141 struct dst_ops ipv4_dst_ops = {
142     family:                  AF_INET,
143     protocol:                __constant_htons(ETH_P_IP),
144     gc:                      rt_garbage_collect,
145     check:                   ipv4_dst_check,
146     reroute:                 ipv4_dst_reroute,
147     destroy:                 ipv4_dst_destroy,
148     negative_advice:         ipv4_negative_advice,
149     link_failure:            ipv4_link_failure,
150     entry_size:              sizeof(struct rtable),
151 };

 95 void * dst_alloc(struct dst_ops * ops)
 96 {
 97     struct dst_entry * dst;
```

If the number of entries in the routing cache exceeds the threshold established at system initialization time, then the garbage collection function, which was also set a boot time to point to *rt_garbage_collect()* is called. The threshold value, *ipv4_dst_ops.gc_thresh,* was set to (*rt_hash_mask* + *1*) in *ip_rt_init()* which was called by *ip_init()*.

```
 99     if (ops->gc && atomic_read(&ops->entries) >
                  ops->gc_thresh) {
100         if (ops->gc())
101             return NULL;
102     }
```

As described earlier the *struct rtable* consists of a *struct dst_entry* followed by a few fields.

```
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 };
```

The meaning of the individual bits of *rt_flag* are defined here. The high order half of the *rt_flags* word is mapped by the *RTCF_* values defined below.

```
50
51 #define RTF_UP          0x0001  /* route usable  */
52 #define RTF_GATEWAY     0x0002  /* dest is a gateway */
53 #define RTF_HOST        0x0004  /* host entry */
54 #define RTF_REINSTATE   0x0008  /* reinstate after tmout*/
55 #define RTF_DYNAMIC     0x0010  /* created dyn. (by redrct)*/
56 #define RTF_MODIFIED    0x0020  /* modified dyn. (by redrct)*/
57 #define RTF_MTU         0x0040  /* specific MTU for route  */
58 #define RTF_MSS         RTF_MTU /* Compatibility :-( */
59 #define RTF_WINDOW      0x0080  /* per route window clamping
*/
60 #define RTF_IRTT        0x0100  /* Initial round trip time  */
61 #define RTF_REJECT      0x0200  /* Reject route */
```

Unfortunately no commentary accompanies these definitions.

```
12 #define RTCF_NOTIFY     0x00010000
13 #define RTCF_DIRECTDST  0x00020000
14 #define RTCF_REDIRECTED 0x00040000
15 #define RTCF_TPROXY     0x00080000
16
17 #define RTCF_FAST       0x00200000
18 #define RTCF_MASQ       0x00400000
19 #define RTCF_SNAT       0x00800000
20 #define RTCF_DOREDIRECT 0x01000000
21 #define RTCF_DIRECTSRC  0x04000000
22 #define RTCF_DNAT       0x08000000
23 #define RTCF_BROADCAST  0x10000000
24 #define RTCF_MULTICAST  0x20000000
25 #define RTCF_REJECT     0x40000000
26 #define RTCF_LOCAL      0x80000000
27
```

These are the possible value for *rt_type*. Unlike *rt_flags* these are mutually exclusive and thus enumerated instead of bit mapped.

```
100 enum
101 {
102     RTN_UNSPEC,
103     RTN_UNICAST,        /* Gateway or direct route      */
104     RTN_LOCAL,          /* Accept locally               */
105     RTN_BROADCAST,      /* Accept locally as broadcast,
106                            send as broadcast */
107     RTN_ANYCAST,        /* Accept locally as broadcast,
108                            but send as unicast */
109     RTN_MULTICAST,      /* Multicast route              */
110     RTN_BLACKHOLE,      /* Drop                         */
111     RTN_UNREACHABLE,    /* Destination is unreachable   */
112     RTN_PROHIBIT,       /* Administratively prohibited  */
113     RTN_THROW,          /* Not in this table            */
114     RTN_NAT,            /* Translate this address       */
115     RTN_XRESOLVE,       /* Use external resolver        */
116 };
117
```

Continuing in *dst_alloc() a* new *dst_entry* structure is allocated from the slab cache *(ipv4_dst_ops.kmem_cachep)* and initialized to its default state.

```
103     dst = kmem_cache_alloc(ops->kmem_cachep,
                               SLAB_ATOMIC);
104     if (!dst)
105          return NULL;
106     memset(dst, 0, ops->entry_size);
107     dst->ops = ops;
108     dst->lastuse = jiffies;
109     dst->input = dst_discard;
110     dst->output = dst_blackhole;
111     atomic_inc(&dst_total);
112     atomic_inc(&ops->entries);
113     return dst;
114 }
```

Completing the route cache entry

What transpires here involves copying routing information from the *fib_info* structure to the *dst_entry* structure and is a real mess. The enumeration below identifies elements of the route metrics array that are carried in the *fib_metrics* array of the *fib_info* structure.

```
261 enum
262 {
263     RTAX_UNSPEC,
265     RTAX_LOCK,
267     RTAX_MTU,
269     RTAX_WINDOW,
271     RTAX_RTT,
273     RTAX_RTTVAR,
275     RTAX_SSTHRESH,
277     RTAX_CWND,
279     RTAX_ADVMSS,
281     RTAX_REORDERING,
283 };
284 #define RTAX_MAX RTAX_REORDERING
```

```
 68      unsigned  fib_metrics[RTAX_MAX];
 69 #define fib_mtu         fib_metrics[RTAX_MTU-1]
 70 #define fib_window      fib_metrics[RTAX_WINDOW-1]
 71 #define fib_rtt         fib_metrics[RTAX_RTT-1]
 72 #define fib_advmss      fib_metrics[RTAX_ADVMSS-1]
```

Alas, the corresponding elements of the *struct dst_entry* are not defined as an array but instead explicitly declared as individual variables. This practice makes it necessary to ensure manually that the definitions remain in sync, but no warning to that effect is present anywhere in the code!

```
 39     unsigned            mxlock;
 40     unsigned            pmtu;
 41     unsigned            window;
 42     unsigned            rtt;
 43     unsigned            rttvar;
 44     unsigned            ssthresh;
 45     unsigned            cwnd;
 46     unsigned            advmss;
 47     unsigned            reordering;
```

The *rt_set_nexthop()* function

The *rt_set_nexthop()* function is defined in net/ipv4/route.c. Its mission is to copy required elements of the *fib_info* structure into their counter parts in the *struct rtable.*

```
1180 static void rt_set_nexthop(struct rtable *rt, struct
                    fib_result *res, u32 itag)
1181 {
1182     struct fib_info *fi = res->fi;
1183
1184     if (fi) {
1185         if (FIB_RES_GW(*res) &&
1186             FIB_RES_NH(*res).nh_scope == RT_SCOPE_LINK)
1187             rt->rt_gateway = FIB_RES_GW(*res);
1188             memcpy(&rt->u.dst.mxlock, fi->fib_metrics,
1189                  sizeof(fi->fib_metrics));
1190         if (fi->fib_mtu == 0) {
1191             rt->u.dst.pmtu =  rt->u.dst.dev->mtu;
1192             if (rt->u.dst.mxlock & (1 << RTAX_MTU)
                     && rt->rt_gateway != rt->rt_dst &&
1194                 rt->u.dst.pmtu > 576)
1195                 rt->u.dst.pmtu = 576;
1196         }
1197 #ifdef CONFIG_NET_CLS_ROUTE
1198     rt->u.dst.tclassid =
                    FIB_RES_NH(*res).nh_tclassid;
1199 #endif
```

The else block handles the case in which the *fib_result* has no corresponding *fib_info*. The conditions under which this might occur are not clear. Here the *mtu* is inherited from the *net_device.*

```
1200     } else
1201         rt->u.dst.pmtu  = rt->u.dst.dev->mtu;
```

Here it is ensured that the *mtu* does not exceed the maximum *mtu*, and the initial maximum segment size for TCP sessions is set to *mtu-40.* The value 40 is the sum of the sizes of standard TCP and IP headers.

```
1203     if (rt->u.dst.pmtu > IP_MAX_MTU)
1204         rt->u.dst.pmtu = IP_MAX_MTU;
1205     if (rt->u.dst.advmss == 0)
1206         rt->u.dst.advmss = max_t(unsigned int,
                          rt->u.dst.dev->mtu - 40,
1207                                ip_rt_min_advmss);
1208     if (rt->u.dst.advmss > 65535 - 40)
1209         rt->u.dst.advmss = 65535 - 40;
1210
1211 #ifdef CONFIG_NET_CLS_ROUTE
1212 #ifdef CONFIG_IP_MULTIPLE_TABLES
1213     set_class_tag(rt, fib_rules_tclass(res));
1214 #endif
1215     set_class_tag(rt, itag);
1216 #endif
1217     rt->rt_type = res->type;
1218 }
```

The *inet_addr_onlink()* function is defined in net/ipv4/devinet.c. Here the parameter *a* is the source address for the packet, and *b* is the IP address of the next hop gateway. The function *inet_ifa_match(),* defined in include/linux/inetdevice.h returns true if the IP address associated with the interface matches the parameter address *with respect to the netmask* of the interface. Therefore, the test here is essentially verifying that the next hop gateway is in the same broadcast domain as the interface which *must* be the case if it is possible to use the gateway at all!

```
187 int inet_addr_onlink(struct in_device *in_dev, u32 a,
                           u32 b)
188 {
189     read_lock(&in_dev->lock);
190     for_primary_ifa(in_dev) {
191          if (inet_ifa_match(a, ifa)) {
192               if (!b || inet_ifa_match(b, ifa)) {
193                    read_unlock(&in_dev->lock);
194                    return 1;
195               }
196          }
197     } endfor_ifa(in_dev);
198     read_unlock(&in_dev->lock);
199     return 0;
200 }
```

```
 88 static __inline__ int inet_ifa_match(u32 addr, struct
                                          in_ifaddr *ifa)
 89 {
 90     return !((addr^ifa->ifa_address)&ifa->ifa_mask);
 91 }
```

Validation of the Source IP Address

The *fib_validate_source()* function is defined in net/ipv4/fib_frontend.c. Its mission is *to determine if a route exists back to the sender of a received packet*. Additional functions include determining on exactly which "logical" interface this packet arrived, calculating the "specific destination" address, and ensuring that the packet arrived on the expected physical interface. At entry here, *dev* always points to the *net_device* on which the packet arrived.

```
206 int fib_validate_source(u32 src, u32 dst, u8 tos, int
        oif, struct net_device *dev, u32 *spec_dst, u32 *itag)
208 {
209         struct in_device    *in_dev;
210         struct rt_key       key;
211         struct fib_result   res;
212         int                 no_addr, rpf;
213         int                 ret;
```

Key construction

The first step is to construct the FIB lookup key. Since the source address is being validated, the value of *key.dst* is set to *src*. The *oif* parameter will be the *loopback* interface unless this packet is being *forwarded through this host*. This will be the *iif* on the reverse path.

```
215         key.dst = src;
216         key.src = dst;
217         key.tos = tos;
218         key.oif = 0;
219         key.iif = oif;
220         key.scope = RT_SCOPE_UNIVERSE;
221
```

The *no_addr* and *rpf (receive packet filter)* flags.

When both IP and the device have received packet filtering enabled, incoming packets, whose routing table entry for their source address doesn't match an IP address bound to the network interface on which they arrived rejected. This procedure can prevent some forms of IP-spoofing.[3]

The IN_DEV_RPFILTER macro, defined in include/linux/inetdevice.h, is a macro which determines whether receive packet filtering is enabled.

This is another of the infamous "thin" places. The packet will have *already* been deemed unroutable if the input interface didn't support IP, but its possible that the interface *does* support IP but *does not* have an IP address.

```
 41 #define IN_DEV_RPFILTER(in_dev)
        (ipv4_devconf.rp_filter && (in_dev)->cnf.rp_filter)

222     no_addr = rpf = 0;
223     read_lock(&inetdev_lock);
224     in_dev = __in_dev_get(dev);
225     if (in_dev) {
226          no_addr = in_dev->ifa_list == NULL;
227          rpf = IN_DEV_RPFILTER(in_dev);
228     }
229     read_unlock(&inetdev_lock);
230
231     if (in_dev == NULL)
232          goto e_inval;
```

3 http://lxr.linux.no/source/Documentation/Configure.help#L5036

Searching the FIB for the Source of the Packet.

Attempt to look up the source address in the FIB. A return code of *NULL* indicates success, and on success, *res* points to a filled in results structure. On failure it is necessary to visit the *last_resort.*

```
234     if (fib_lookup(&key, &res))
235         goto last_resort;
```

We have seen earlier that broadcast and multicast *source* addresses are considered *martian*. Therefore, since a *source* address is being validated, only unicast route types are legitimate. Note that RTN_LOCAL is also illegal.

```
236     if (res.type != RTN_UNICAST)
237         goto e_inval_res;
```

Identifiying the *spec_dst*

The FIB_RES_PREF_SRC macro uses the *prefsrc* field of the *fib_info* structure if it is not NULL. If that field *is* NULL, *inet_select_addr()* is used to obtain an IP address associated with the *net_device* that is assocated with the *fib_nh* structure that is contained in the *fib_info.* This effectively sets the *spec_dst* to the IP address associated with the outgoing interface for the return path and would appear to be in contradiction to the comment at the bottom of page 5. This doesn't matter though because (on page 5 at least) the *spec_dst* returned by this routine is ignored. The *fib_combine_itag()* function has effect only when CONFIG_NET_CLS_ROUTE is defined.

```
238     *spec_dst = FIB_RES_PREFSRC(res);
239     fib_combine_itag(itag, &res);
```

The *FIB_RES_DEV(res)* macro, defined in include/net/ip_fib.h, returns the device that should be used for the next outgoing hop associated with this FIB entry.

```
#define FIB_RES_DEV(res) (FIB_RES_NH(res).nh_dev)
```

Since the key that was used to obtain the *res* pointer used the remote source address of this packet, the device on which the packet arrived should be the next hop device for a transmission back to the source. If multipath routing is enabled and there are multiple possible next hops, *(res.fi->fib_nhs > 1),* the code does not bother to ensure that the device upon which the packet was received is included in the possible outgoing next hops.

```
240 #ifdef CONFIG_IP_ROUTE_MULTIPATH
241     if(FIB_RES_DEV(res) == dev || res.fi->fib_nhs > 1)
242 #else
```

If the device upon which the packet was received is consistent with the perceived next hop, the value returned is dependent upon whether the scope of the FIB entry is RT_SCOPE_HOST. This is the "normal" return point. For local LAN addresses (130.127.48.0/23) the scope of the fib node is LINK but the scope of the next hop is HOST. Thus 0 will be returned here if the reverse path is gatewayed and 1 if the sender is directly attached.

```
243     if (FIB_RES_DEV(res) == dev)
244 #endif
245     {
246         ret = FIB_RES_NH(res).nh_scope >=
                                     RT_SCOPE_HOST;
247         fib_res_put(&res);
248         return ret;
249     }
250     fib_res_put(&res);
```

Device mismatch

If the network device (*dev*) doesn't match, the action taken depends upon the state of *no_addr* and *rpf*. The value of *no_addr* will be 1 only if the interface on which the packet arrived does not have an associated IP address.

```
251     if (no_addr)
252             goto last_resort;
```

If receive packet filtering is enabled on the device on which the packet arrived and device on which the packet arrived was not the expected device, a jump is made to the point at which -EINVAL is returned to the caller.

```
253     if (rpf)
254             goto e_inval;
```

If an acceptable device has not been found, adjust the *key* by explicitly encoding the *oif* with the index of the device on which the packet was received and *retry* the fib lookup. If it fails, success is returned! This presumably means that the route back has device affinity, but that the packet didn't arrive on the expected device but since *rpf* is false this situation is acceptable.

```
255     key.oif = dev->ifindex;
256
257     ret = 0;
258     if (fib_lookup(&key, &res) == 0) {
```

However, if it succeeds, and the route type is UNICAST, the value returned is dependent upon the scope of the next hop. Recall that a ret value of 0 means on the same LAN and 1 means gatewayed. If the route type is not UNICAST success will also be returned.

```
259             if (res.type == RTN_UNICAST) {
260                     *spec_dst = FIB_RES_PREFSRC(res);
261                     ret = FIB_RES_NH(res).nh_scope >=
                                        RT_SCOPE_HOST;
262             }
263             fib_res_put(&res);
264     }


265     return ret;
```

The last resort

Arrival at the *last resort* implies that either the FIB lookup failed or that the interface on which the packet arrived does not have an IP address. That is a fatal problem if *rpf* is enabled on the device, but if not the route will be assumed gatewayed and, when passed an input parameter of 0, *inet_select_addr* will return an IP address if *any* device has one.

```
267 last_resort:
268     if (rpf)
269          goto e_inval;
270     *spec_dst = inet_select_addr(dev, 0,
                            RT_SCOPE_UNIVERSE);
271     *itag = 0;
272     return 0;
273
274 e_inval_res:
275     fib_res_put(&res);
276 e_inval:
277     return -EINVAL;
278 }
```