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 resememblence 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.
*/

int ip_route_input_slow(struct sk_buff *skb, u32
        daddr, u32 saddr, u8 tos, struct net_device *dev)
{
        struct rt_key           key;
        struct fib_result       res;
        struct in_device        *in_dev = in_dev_get(dev);
        struct in_device        *out_dev = NULL;
        unsigned                flags = 0;
        u32                     itag = 0;
        struct rtable           *rth;
        unsigned                hash;
        u32                     spec_dst;
        int                     err = -EINVAL;
        int                     free_res = 0;

```

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

```
        /* IP on this device is disabled. */

        if (!in_dev)
             goto out;
```

A key is constructed for lookup into the FIB.

```
    key.dst          = daddr;
    key.src          = saddr;
    key.tos          = tos;
#ifdef CONFIG_IP_ROUTE_FWMARK
    key.fwmark       = skb->nfmark;
#endif
    key.iif          = dev->ifindex;
    key.oif          = 0;
    key.scope        = RT_SCOPE_UNIVERSE;
```

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.

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

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.

```

    /*   Check for the most weird martians, which can
         be not detected by fib_lookup.
     */

    if (MULTICAST(saddr) || BADCLASS(saddr) ||
         LOOPBACK(saddr))
         goto martian_source;
```

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, but the rationale for this choice is unclear.

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

Zero valued source addresses are invalid unless the destination is also zero.

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

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

When the destination is a badclass/loopback/zeronet address, an error is also returned.

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

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. Recall that 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 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

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

When 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.

```
              if (!IN_DEV_FORWARD(in_dev))
                   goto e_inval;
              goto no_route;
        }
        free_res = 1; /* remember to free the res struct */

```

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

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

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

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

    if (1) {
         u32 src_map = saddr;
         if (res.r)
              src_map = fib_rules_policy(saddr, &res,
                                                  &flags);

         if (res.type == RTN_NAT) {
              key.dst = fib_rules_map_destination(daddr,
                        &res);
              fib_res_put(&res);
              free_res = 0;
              if (fib_lookup(&key, &res))
                   goto e_inval;
              free_res = 1;
              if (res.type != RTN_UNICAST)
                   goto e_inval;
              flags |= RTCF_DNAT;
         }
         key.src = src_map;
    }
#endif
```

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.

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

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.

```
    if (res.type == RTN_LOCAL) {
         int result;

         result = fib_validate_source(saddr, daddr,
                        tos, loopback_dev.ifindex,
                        dev, &spec_dst, &itag);
         if (result < 0)
              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 actually owned by this host and the appropriate routing cache flag is set.

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

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.[1] 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."

```
         spec_dst = daddr;
         goto local_input;
    }
```

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

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.

```
    if (!IN_DEV_FORWARD(in_dev))
         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.

```
    if (res.type != RTN_UNICAST)
         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.

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

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.

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

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

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

```

Also as noted previously, if the value returned by fib_validate_source() is positive, then the source IP address is owned by this host.

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

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. Note that a system acting as a router apparently will never receive a redirect even if it might improve routing.

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

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!

```
int inet_addr_onlink(struct in_device *in_dev, u32 a,
                   u32 b)
{
    read_lock(&in_dev->lock);
    for_primary_ifa(in_dev) {
         if (inet_ifa_match(a, ifa)) {
              if (!b || inet_ifa_match(b, ifa)) {
                   read_unlock(&in_dev->lock);
                   return 1;
              }
         }
    } endfor_ifa(in_dev);
    read_unlock(&in_dev->lock);
    return 0;
}
```

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

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. RTCF_DNAT ..??

```
    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.
    */
         if (out_dev == in_dev && !(flags & RTCF_DNAT))
            goto e_inval;
    }
```

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

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

The parameter, ipv4_dst_ops, is declared and initialized in net/ipv4/route.c.

```
struct dst_ops ipv4_dst_ops = {
    family:                 AF_INET,
    protocol:               __constant_htons(ETH_P_IP),
    gc:                     rt_garbage_collect,
    check:                  ipv4_dst_check,
    reroute:                ipv4_dst_reroute,
    destroy:                ipv4_dst_destroy,
    negative_advice:        ipv4_negative_advice,
    link_failure:           ipv4_link_failure,
    entry_size:             sizeof(struct rtable),
};
```

The dst_alloc() function is defined in net/core/dev.c.

```
void * dst_alloc(struct dst_ops * ops)
{
    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(). Need to include rt_garbage_collect ..

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

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

```
struct rtable
{
    union
    {
         struct dst_entry dst;
         struct rtable *rt_next;
    } u;

    unsigned       rt_flags;
    unsigned       rt_type;

    __u32          rt_dst; /* Path destination     */
    __u32          rt_src; /* Path source          */
    int            rt_iif;

/* Info on neighbour */
    __u32          rt_gateway;

/* Cache lookup keys */
    struct rt_key  key;

/* Miscellaneous cached information */
    __u32 rt_spec_dst; /* RFC1122 specific destination */
    struct inet_peer *peer; /* long-living peer info */

#ifdef CONFIG_IP_ROUTE_NAT
    __u32     rt_src_map;
    __u32     rt_dst_map;
#endif
};
```

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.

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

Unfortunately no commentary accompanies these definitions.

```
#define RTCF_NOTIFY     0x00010000
#define RTCF_DIRECTDST  0x00020000
#define RTCF_REDIRECTED 0x00040000
#define RTCF_TPROXY     0x00080000

#define RTCF_FAST       0x00200000
#define RTCF_MASQ       0x00400000
#define RTCF_SNAT       0x00800000
#define RTCF_DOREDIRECT 0x01000000
#define RTCF_DIRECTSRC  0x04000000
#define RTCF_DNAT       0x08000000
#define RTCF_BROADCAST  0x10000000
#define RTCF_MULTICAST  0x20000000
#define RTCF_REJECT     0x40000000
#define RTCF_LOCAL      0x80000000
```

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

```
enum
{
    RTN_UNSPEC,
    RTN_UNICAST,        /* Gateway or direct route      */
    RTN_LOCAL,          /* Accept locally               */
    RTN_BROADCAST,      /* Accept locally as broadcast,
                           send as broadcast */
    RTN_ANYCAST,        /* Accept locally as broadcast,
                           but send as unicast */
    RTN_MULTICAST,      /* Multicast route              */
    RTN_BLACKHOLE,      /* Drop                         */
    RTN_UNREACHABLE,    /* Destination is unreachable   */
    RTN_PROHIBIT,       /* Administratively prohibited  */
    RTN_THROW,          /* Not in this table            */
    RTN_NAT,            /* Translate this address       */
    RTN_XRESOLVE,       /* Use external resolver        */
};
```

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.

```
    dst = kmem_cache_alloc(ops->kmem_cachep,
                           SLAB_ATOMIC);
    if (!dst)
         return NULL;
    memset(dst, 0, ops->entry_size);
    dst->ops = ops;
    dst->lastuse = jiffies;
    dst->input = dst_discard;
    dst->output = dst_blackhole;
    atomic_inc(&dst_total);
    atomic_inc(&ops->entries);
    return dst;
}
```

Back in 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.

```
    atomic_set(&rth->u.dst.__refcnt, 1);
    rth->u.dst.flags= DST_HOST;
    rth->key.dst     = daddr;
    rth->rt_dst      = daddr;
    rth->key.tos     = tos;
#ifdef CONFIG_IP_ROUTE_FWMARK
    rth->key.fwmark = skb->nfmark;
#endif
    rth->key.src     = saddr;
    rth->rt_src      = saddr;
    rth->rt_gateway = daddr;
#ifdef CONFIG_IP_ROUTE_NAT
    rth->rt_src_map = key.src;
    rth->rt_dst_map = key.dst;
    if (flags&RTCF_DNAT)
         rth->rt_gateway = key.dst;
#endif
    rth->rt_iif      =
    rth->key.iif     = dev->ifindex;
    rth->u.dst.dev   = out_dev->dev;
    dev_hold(rth->u.dst.dev);
    rth->key.oif     = 0;
    rth->rt_spec_dst= spec_dst;
```

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 faciltate the processing of future packets.

```
     rth->u.dst.input = ip_forward;
     rth->u.dst.output = ip_output;

     rt_set_nexthop(rth, &res, itag);
```

What transpires next 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.

```
enum
{
     RTAX_UNSPEC,
     RTAX_LOCK,
     RTAX_MTU,
     RTAX_WINDOW,
     RTAX_RTT,
     RTAX_RTTVAR,
     RTAX_SSTHRESH,
     RTAX_CWND,
     RTAX_ADVMSS,
     RTAX_REORDERING,
};
#define RTAX_MAX RTAX_REORDERING
```

```
        unsigned  fib_metrics[RTAX_MAX];
#define fib_mtu           fib_metrics[RTAX_MTU-1]
#define fib_window        fib_metrics[RTAX_WINDOW-1]
#define fib_rtt           fib_metrics[RTAX_RTT-1]
#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!

```
     unsigned            mxlock;
     unsigned            pmtu;
     unsigned            window;
     unsigned            rtt;
     unsigned            rttvar;
     unsigned            ssthresh;
     unsigned            cwnd;
     unsigned            advmss;
     unsigned            reordering;
```

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.

```
static void rt_set_nexthop(struct rtable *rt, struct
                    fib_result *res, u32 itag)
{
    struct fib_info *fi = res->fi;

    if (fi) {
         if (FIB_RES_GW(*res) &&
              FIB_RES_NH(*res).nh_scope == RT_SCOPE_LINK)
              rt->rt_gateway = FIB_RES_GW(*res);
              memcpy(&rt->u.dst.mxlock, fi->fib_metrics,
                   sizeof(fi->fib_metrics));
         if (fi->fib_mtu == 0) {
              rt->u.dst.pmtu =  rt->u.dst.dev->mtu;
              if (rt->u.dst.mxlock & (1 << RTAX_MTU)
                   && rt->rt_gateway != rt->rt_dst &&
                   rt->u.dst.pmtu > 576)
                   rt->u.dst.pmtu = 576;
         }
#ifdef CONFIG_NET_CLS_ROUTE
    rt->u.dst.tclassid =
                   FIB_RES_NH(*res).nh_tclassid;
#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.

```
    } else
         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. Since this section of the code is handling a forwarded packet, its not clear why it is necessary to set up advmss.

```
    if (rt->u.dst.pmtu > IP_MAX_MTU)
         rt->u.dst.pmtu = IP_MAX_MTU;
    if (rt->u.dst.advmss == 0)
         rt->u.dst.advmss = max_t(unsigned int,
                   rt->u.dst.dev->mtu - 40,
                             ip_rt_min_advmss);
    if (rt->u.dst.advmss > 65535 - 40)
         rt->u.dst.advmss = 65535 - 40;

```

```
#ifdef CONFIG_NET_CLS_ROUTE
#ifdef CONFIG_IP_MULTIPLE_TABLES
    set_class_tag(rt, fib_rules_tclass(res));
#endif
    set_class_tag(rt, itag);
#endif
    rt->rt_type = res->type;
}
```

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

```
    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.

```
#ifdef CONFIG_NET_FASTROUTE
    if (netdev_fastroute &&
         !(flags&(RTCF_NAT|RTCF_MASQ|
         RTCF_DOREDIRECT))) {
         struct net_device *odev = rth->u.dst.dev;
         if (odev != dev &&
                  dev->accept_fastpath &&
                  odev->mtu >= dev->mtu &&
                  dev->accept_fastpath(dev,
         &rth->u.dst) == 0)
              rth->rt_flags |= RTCF_FAST;
    }
#endif
```

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.

```
intern:
    err = rt_intern_hash(hash, rth,
                               (struct rtable**)&skb->dst);
done:
        in_dev_put(in_dev);
        if (out_dev)
                in_dev_put(out_dev);
        if (free_res)
                fib_res_put(&res);
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 postive value this host owns the source address.

```
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 packets destined for this host.

The handling of broadcast packets merges 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;
```

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 which also has the value 0. In this case inet_select_addr() will just return the IP address of the first configured interface associated with the struct net_device.

```
no_route:
    rt_cache_stat[smp_processor_id()].in_no_route++;
    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.

```
    res.type = RTN_UNREACHABLE;
    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.

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

e_inval:
        err = -EINVAL;
        goto done;

e_nobufs:
        err = -ENOBUFS;
        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.

```
martian_source:

rt_cache_stat[smp_processor_id()].in_martian_src++;

#ifdef CONFIG_IP_ROUTE_VERBOSE
    if (IN_DEV_LOG_MARTIANS(in_dev) &&
         net_ratelimit()) {
    /*
         RFC1812 recommendation, if source is martian,
         the only hint is MAC header.
     */
         printk(KERN_WARNING "martian source
                   %u.%u.%u.%u from "
                   "%u.%u.%u.%u, on dev %s\n",
                   NIPQUAD(daddr), NIPQUAD(saddr),
                   dev->name);
         if (dev->hard_header_len) {
              int i;
              unsigned char *p = skb->mac.raw;
              printk(KERN_WARNING "ll header: ");
              for (i = 0; i < dev->hard_header_len;
                   i++, p++) {
                   printk("%02x", *p);
                   if (i < (dev->hard_header_len - 1))
                        printk(":");
                   }
                   printk("\n");
              }
         }
#endif
    goto e_inval;
}
```

Validation of the Source IP Address

The fib_validate_source() function is defined in net/ipv4/fib_frontend.c. Its mission is to determine that 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.

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

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.

```
        key.dst = src;
        key.src = dst;
        key.tos = tos;
        key.oif = 0;
        key.iif = oif;
        key.scope = RT_SCOPE_UNIVERSE;

        no_addr = rpf = 0;
        read_lock(&inetdev_lock);
```

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

Each device that supports IPv4 traffic has an associated structure of type struct in_device that contains IPv4 specific data for the device. The variable no_addr is a flag set that is to one when ifa_list is empty. How could this happen? The ifa_list is a list of structures defining the addresses associated with a struct in_device which is in turn associated with a struct net_device.

```
struct in_ifaddr
{
    struct in_ifaddr    *ifa_next;
    struct in_device    *ifa_dev;
    u32                 ifa_local;
    u32                 ifa_address;
    u32                 ifa_mask;
    u32                 ifa_broadcast;
    u32                 ifa_anycast;
    unsigned char       ifa_scope;
    unsigned char       ifa_flags;
    unsigned char       ifa_prefixlen;
    char                ifa_label[IFNAMSIZ];
};
```

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.[2]

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

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

```
    in_dev = __in_dev_get(dev);
    if (in_dev) {
         no_addr = in_dev->ifa_list == NULL;
         rpf = IN_DEV_RPFILTER(in_dev);
    }
    read_unlock(&inetdev_lock);

    if (in_dev == NULL)
         goto e_inval;
```

2 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.

```
if (fib_lookup(&key, &res))
     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.

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

Obtain the specific destination address. 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.

```
*spec_dst = FIB_RES_PREFSRC(res);
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.

```
#ifdef CONFIG_IP_ROUTE_MULTIPATH
     if(FIB_RES_DEV(res) == dev || res.fi->fib_nhs > 1)
#else
     if (FIB_RES_DEV(res) == dev)
#endif
     {
```

If the device upon which the packet was received is consistent with the perceived next hop, and the scope of the FIB entry is RT_SCOPE_HOST release the FIB entry return success now.

```
          ret = FIB_RES_NH(res).nh_scope >=
                                        RT_SCOPE_HOST;
          fib_res_put(&res);
          return ret;
     }
     fib_res_put(&res);
```

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.

```
    if (no_addr)
        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.

```
    if (rpf)
        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! However, if it succeeds, and the route type is UNICAST, then success is returned only if the scope of the next hop is RT_SCOPE_HOST.

```
    key.oif = dev->ifindex;

    ret = 0;
    if (fib_lookup(&key, &res) == 0) {
        if (res.type == RTN_UNICAST) {
            *spec_dst = FIB_RES_PREFSRC(res);
            ret = FIB_RES_NH(res).nh_scope >=
                               RT_SCOPE_HOST;
        }
        fib_res_put(&res);
    }
    return ret;
```

We reach here when we couldn't match network interface. Determine specific destination address and return. What will inet_select_address() return here?

```
last_resort:
    if (rpf)
        goto e_inval;
    *spec_dst = inet_select_addr(dev, 0,
                          RT_SCOPE_UNIVERSE);
    *itag = 0;
    return 0;

e_inval_res:
    fib_res_put(&res);
e_inval:
    return -EINVAL;
}
```