Routing of IP Output Packets

The UDP interface to the routing system

The \textit{RT\_TOS} macro retrieves the low order 5 bits from the \textit{tos} field of the \textit{struct sock}. These will be 0 unless set by \texttt{setsockopt()}. These include the \textit{DTRC} bits and the low order bit is the ONLINK bit.

\begin{verbatim}
#define RT_TOS(tos) ((tos)&IPTOS_TOS_MASK)
\end{verbatim}

501 \begin{verbatim}
tos = RT_TOS(sk->protinfo.af_inet.tos);
\end{verbatim}

The RTO\_ONLINK bit forces the destination (or next hop in case of a \textit{strict} source route) to be reachable in a single hop.

502 \begin{verbatim}
if (sk->localroute || (msg->msg_flags & MSG_DONTROUTE)||
      (ipc.opt && ipc.opt->is_strictroute)) {
  tos |= RTO_ONLINK;
  connected = 0;
}
\end{verbatim}

519 \begin{verbatim}
if (rt == NULL) {
  err = ip_route_output(&rt, daddr, ufh.saddr, tos,
                       ipc.oif);
\end{verbatim}

The \textit{ip\_route\_output} function

The \textit{ip\_route\_output()} function is a wrapper which constructs a \textit{rt\_key} structure and calls \textit{ip\_route\_output\_key}. Building a key on the stack with the \textit{iif} not specified is \textit{dangerous} and necessarily implies the \textit{iif} element is not used in output routing.

136 \begin{verbatim}
static inline int ip_route_output(struct rtable **rp, u32 daddr, u32 saddr, u32 tos, int oif)
{
  struct rt_key key = {dst:daddr, src:saddr, oif:oif, tos:tos };
  return ip_route_output_key(rp, &key);
}
\end{verbatim}
The `ip_route_output_key()` function

The `ip_route_output_key()` function is defined in `net/ipv4/route.c`.

```
1984 int ip_route_output_key(struct rtable **rp, const
    struct rt_key *key)
1985 {
1986    unsigned hash;
1987    struct rtable *rth;
1988
The `rt_key` structure passed as argument is to form the hash code that is used as an index into the table of `rt_hash_buckets` that was created during system initialization. Note the ugly “dual hash” that is carried out with the informal hashing of `src` and `oif` in the call!!

```
1989    hash = rt_hash_code(key->dst, key->src ^
    (key->oif << 5), key->tos);
```

The hash function is implemented by the inline function `rt_hash_code()`.

```
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 }
```
The route cache lookup

The hash code returned by the above function is used by ip_route_output_key to search in the respective hash queue of routing cache (rt_hash_table) to find an entry that matches the input key with respect to (dst, src, oif, tos). CONFIG_IP_ROUTE_FWMARK is an option to specify different routes for packets with different (netfilter) mark values. If this option is configured, the mark value is also used in matching. The last test forces the tos bits in the table and in the key to agree in the IPTOS_RT_MASK and RTO_ONLINK positions. Note that the iif problem is addressed by considering only those entries in the route cache for which iif is zero.

1991 read_lock_bh(&rt_hash_table[hash].lock);
1992 for (rth = rt_hash_table[hash].chain; rth;
    rth = rth->u.rt_next) {
1993     if  (rth->key.dst == key->dst &&
1994         rth->key.src == key->src &&
1995         rth->key.iif == 0 &&
1996         rth->key.oif == key->oif &&
1997 #ifdef CONFIG_IP_ROUTE_FWMARK
1998         rth->key.fwmark == key->fwmark &&
1999 #endif
2000         !((rth->key.tos ^ key->tos) &
2001            (IPTOS_RT_MASK | RTO_ONLINK))) {

Each time a routing cache entry is used, its time of last use should be updated so that the garbage collection procedure can identify entries that have not been used in a long time. The dst_hold() function simply increments the reference count (atomic_inc(&dst->__refcnt)). The element cannot be deleted while the refcnt is positive. When routing packets individually, this reference will be stored in the sk_buff and dropped by kfree_skb().

2002     rth->u.dst.lastuse = jiffies;
2003     dst_hold(&rth->u.dst);
2004     rth->u.dst.__use++;
2005     rt_cache_stat[smp_processor_id()].out_hit++;
2006     read_unlock_bh(&rt_hash_table[hash].lock);

Set argument "*rp" to point to this entry and return.

2007   *rp = rth;
2008   return 0;
2009 }
Failure to find a route cache element

Exit from the loop means that a route to desired destination was not cached. In this case it is necessary to call `ip_route_output_slow()` which tries to construct a new route cache element using the FIB.

```c
read_unlock_bh(&rt_hash_table[hash].lock);
return ip_route_output_slow(rp, key);
```
The ip_route_output_slow() function, defined in net/ipv4/route.c is the major route resolver. Given a "routing key" as an input parameter, this routine builds a new route cache entry and stores a pointer to it in the parameter **rp. A Linux route is defined by (dst, src, oif, iif, tos, scope).

```c
1690 int ip_route_output_slow(struct rtable **rp, const
   struct rt_key *oldkey)
1691 {
1692     struct rt_key key;
1693     struct fib_result res;
1694     unsigned flags = 0;
1695     struct rtable *rth;
1696     struct net_device *dev_out = NULL;
1697     unsigned hash;
1698     int free_res = 0;
1699     int err;
1700     u32 tos;
```

The function uses two important local variables: key is of struct rt_key, derived from the values pointed to by oldkey and is used to specify the characteristics of the desired route;

```c
48 struct rt_key
49 {
50     __u32 dst; /* Destination IP address */
51     __u32 src; /* Source IP address */
52     int iif; /* Input interface index */
53     int oif; /* Output interface index */
54 #ifdef CONFIG_IP_ROUTE_FWMARK
55     __u32 fwmark;
56 #endif
57     __u8 tos; /* Requested type of service */
58     __u8 scope; /* Host, LAN, site, universe */
59 };```
The \textit{fib\_result} structure

The variable \textit{res} has type \textit{struct fib\_result} and is later used in building the new routing cache entry.

```c
86 struct fib_result
87 {
88   unsigned char   prefixlen;
89   unsigned char   nh_sel;
90   unsigned char   type;
91   unsigned char   scope;
92   struct fib_info *fi;
93 #ifdef CONFIG_IP_MULTIPLE_TABLES
94   struct fib_rule *r;
95 #endif
96
```

The elements of the \textit{fib\_result} structure include:

- \textit{prefixlen} prefix length or equivalently the number of leading 1 bits in the subnet mask
- \textit{nh\_sel} Next hop (output dev index). This actually appears under `grep -r` to be one of the ever popular \textit{write only} variables!
- \textit{scope} An indication of the distance to the destination IP address (e.g. host, local network, site, universe). Higher scope values are more specific.
- \textit{type} type of address (LOCAL, UNICAST, BROADCAST, MULTICAST)
- \textit{fi} Pointer to the \textit{fib\_info} structure that contains protocol and hardware information specific to the output interface selected
- \textit{r} Pointer to a \textit{fib\_rule} structure used for policy based routing.

6
The fib_rule structure

The fib_rule structure is defined in net/ipv4/fib_rules.c. This structure is the key element defining the existence of a route with a given class of service between a specific source and destination address. It is not used unless CONFIG_IP_MULTIPLE_TABLES has been defined.

```c
struct fib_rule {
    struct fib_rule *r_next;
    atomic_t r_clntref;
    u32 r_preference;
    unsigned char r_table;
    unsigned char r_action;
    unsigned char r_dst_len;
    unsigned char r_src_len;
    u32 r_src;
    u32 r_srcmask;
    u32 r_dst;
    u32 r_dstmask;
    u32 r_srcmap;
    u8 r_flags;
    u8 r_tos;
    #ifdef CONFIG_IP_ROUTE_FWMARK
    u32 r_fwmark;
    #endif
    int r_ifindex;
    #ifdef CONFIG_NET_CLS_ROUTE
    __u32 r_tclassid;
    #endif
    char r_ifname[IFNAMSIZ];
    int r_dead;
};
```
Constructing the new route key

The function `ip_route_output_slow()` begins by constructing the new routing `key` structure. Manipulation of the `tos` field is somewhat strange. TOS related constants are defined as follows:

```c
#define IPTOS_TOS_MASK 0x1E
#define IPTOS_TOS(tos) ((tos) & IPTOS_TOS_MASK)
#define IPTOS_LOWDELAY 0x10
#define IPTOS_THROUGHPUT 0x08
#define IPTOS_RELIABILITY 0x04
#define IPTOS_MINCOST 0x02
#define IPTOS_RT_MASK (IPTOS_TOS_MASK & ~3)
#define RTO_ONLINK 0x01
```

RTO_ONLINK is a flag that indicates the destination is no more than one hop away and reachable via a link layer protocol. The `IPTOS_RT_MASK` disables both `IPTOS_MINCOST` and `RTO_ONLINK`. Even though the RTO_ONLINK by is not carried in the `tos` field of the key, we will see that it is carried in the `scope` element of the new key.

```c
tos = oldkey->tos & (IPTOS_RT_MASK | RTO_ONLINK);
key.dst = oldkey->dst;
key.src = oldkey->src;
key.tos = tos & IPTOS_RT_MASK;
```

Setting up the `iif` and `oif`

The input interface identifier is forced to that of the loopback device. The variable `loopback_dev` is an instance of `struct net_device` and is globally defined in `drivers/net/Space.c`. The value of the `ifindex` field is a unique identifier assigned to the interface at initialization time.

```c
key.iif = loopback_dev.ifindex;
key.oif = oldkey->oif;
```

CONFIG_IP_ROUTE_FWMARK is an option to specify different route for packets with different (netfilter) mark values.

```c
#ifdef CONFIG_IP_ROUTE_FWMARK
key.fwmark = oldkey->fwmark;
#endif
```
Route scope assignment

The value of `key.scope` is an indication of the distance from the destination. Here there are only two possible choices, and they depend on the setting of RTO_ONLINK. If RTO_ONLINK is set then the scope must be RT_SCOPE_LINK. Otherwise it is RT_SCOPE_UNIVERSE. Thus the scope attribute of the new key does reflect the setting of the RTO_ONLINK bit in the `tos` field of the old key.

```
1711    key.scope = (tos & RTO_ONLINK) ? RT_SCOPE_LINK:
1712                        RT_SCOPE_UNIVERSE;
```

As described more fully in the kernel comments below and subsequent data definitions it is clear that a wider range of possible scopes is intended and that the higher the value of scope the more specific the target routing domain.

```
``Really not a scope, but sort of distance to the destination. NOWHERE are reserved for non-existing dests, HOST is our local addresses, LINK are dests on directly attached link and UNIVERSE is everywhere in the Universe. Intermediate values are also possible f.e. interior routes could be assigned a value between UNIVERSE and LINK."``

RT_SCOPE_LINK, RT_SCOPE_UNIVERSE stand for on-link routes and global routes respectively and are defined in `include/linux/rtnetlink.h`.

```
155 enum rt_scope_t
156 {
157    RT_SCOPE_UNIVERSE=0,
158    /* User defined values */
159    RT_SCOPE_SITE=200,
160    RT_SCOPE_LINK=253,
161    RT_SCOPE_HOST=254,
162    RT_SCOPE_NOWHERE=255
163    }
```

Initializing the `fib_info` pointer

The `fibinfo` pointer in the results structure is initialized to NULL.

```
1713    res.fi          = NULL;
```
The multiple tables option

CONFIG_IP_MULTIPLE_TABLES is an option that allows the Linux router to be able to take the packet's source address into account when making routing decision. (Normally, a router decides what to do with a received packet based solely on the packet's final destination address.)

The routing tables are referred to as "classes". Currently, the number of classes is limited to 255, of which three classes are builtin:\footnote{\url{http://lxr.linux.no/source/Documentation/networking/policy-routing.txt}}:

\begin{align*}
\text{RT\_CLASS\_LOCAL} & = 255 & \text{- local interface addresses, broadcasts, nat addresses} \\
\text{RT\_CLASS\_MAIN} & = 254 & \text{- all normal routes are put here by default.} \\
\text{RT\_CLASS\_DEFAULT} & = 253 & \text{- If the ip\_fib\_model == 1, then normal default routes are put there. If the ip\_fib\_model == 2, all gateway routes are put there.}
\end{align*}

\begin{verbatim}
1714 #ifdef CONFIG_IP_MULTIPLE_TABLES
1715     res.r           = NULL;
1716 #endif
\end{verbatim}

This facility is disabled by default and normally only two tables LOCAL and MAIN are used.
Validating non-zero source addresses

If the source address is non-zero, it must not be of type MULTICAST, BADCLASS or ZERONET (these macros are defined in `include/linux/in.h`) and it must map to some physical interface that is on this host, but not necessarily the one specified by `oldkey.iif`.

182  
183  
184  
185  
186  
1718  
1719  
1720  
1721  
1722  
1723  
11
Finding an interface with the specified source address

The `ip_dev_find()` function looks up the IP source address in the local table and returns a pointer to the struct `net_device` associated with the source address. This function is defined in `net/ipv4/fib_frontend.c`.

```c
1725 /* It is equivalent to inet_addr_type(saddr)==RTN_LOCAL */
1726     dev_out = ip_dev_find(oldkey->src);
```

On return from `ip_find_dev()` to `ip_route_output_slow()`, If the value of `dev_out` is NULL, then there is no usable network interface associated with the source IP address. The comment below discusses why it is not necessary that the device found here actually map to the output interface specified by the caller. He actually probably means `key.oif == dev_out->oif`.

```c
1727 if (dev_out == NULL)
1728     goto out;
1729 1730 /* I removed check for oif == dev_out->oif here.
1731 It was wrong by three reasons:
1732 1. ip_dev_find(saddr) can return wrong iface, if saddr
1733 is assigned to multiple interfaces.
1734 2. Moreover, we are allowed to send packets with saddr
1735 of another iface. --ANK
1736 */
```
Routing of multicasts or broadcasts for which a source address was specified

Since \( oif = 0 \) means unspecified, what is happening here is a coerced conversion of a multicast and broadcast destination addresses to use the output interface associated with the device that was returned. In addition to the factors discussion below, it is also the case that proper multicast addresses must be associated with a specific interface.

```c
1738     if (oldkey->oif == 0
1739     && (MULTICAST(oldkey->dst) ||
1740         oldkey->dst == 0xFFFFFFFF)) {
1741     /* Special hack: user can direct multicasts
1742        and limited broadcast via necessary interface
1743        without fiddling with IP_MULTICAST_IF or IP_PKTINFO.
1744        This hack is not just for fun, it allows
1745        vic,vat and friends to work.
1746        They bind socket to loopback, set ttl to zero
1747        and expect that it will work.
1748        From the viewpoint of routing cache they are broken,
1749        because we are not allowed to build multicast path
1750        with loopback source addr (look, routing cache
1751        cannot know, that ttl is zero, so that packet
1752        will not leave this host and route is valid).
1753        Luckily, this hack is good workaround.
1754     */
1755     key.oif = dev_out->ifindex;
1756     goto make_route;
1757 }
```

Release the device by invoking the `dev_put()` function defined in `include/linux/netdevice.h`. Note that for unicasts the value of `dev_out` is reset to NULL undoing the effect of this code block!

```c
1758     if (dev_out)
1759         dev_put(dev_out);
1760     dev_out = NULL;
1761     } /* end if (oldkey->src) */
```
Handling a specific output interface specification

If an output interface index is specified, it is necessary to see if the interface really exists and if it is also possible to associate a source IP address with it. This process starts with an attempt to retrieve a pointer to the associated `struct net_device`. A return value of NULL indicates the device is not found. If the device exists, its reference count is incremented, and the pointer is safe until `dev_put` is called to release it.

```c
1762     if (oldkey->oif) {
1763         dev_out = dev_get_by_index(oldkey->oif);
1764         err = -ENODEV;
1765         if (dev_out == NULL)
1766            goto out;
```

The IPV4 specific data is retrieved by the `in_dev_get()` function which is defined in `include/linux/inetdevice.h`. This call returns the `void *ip_ptr` element of the `net_device` structure. This pointer points to an instance of `struct in_device`. Each `net_device` that supports IPV4 also has an associated `struct in_device` that carries the IPV4 dependencies of the device layer. An important element of the `in_device` is the `ifa_list` pointer. This pointer is the root of a list of `struct ifa_list` elements.

```c
1767     if (__in_dev_get(dev_out) == NULL) {
1768         dev_put(dev_out);
1769         goto out;        /* Wrong error code */
1770     }
1771```
Local multicasts and broadcasts with non-zero oif

If the destination address is a LOCAL multicast address (0xE00000xx) or broadcast address, the source address is set to an IP address associated with the specified output device. Recall that dev_out is a pointer to the struct net_device associated with the explicitly specified output interface. The call to inet_select_address() will return the ifa_local associated with the first interface that is found associated with the net_device that has scope no more restrictive (numerically less than or equal to) than LINK. The use of RT_SCOPE_LINK seems a bit unusual here. It turns out that this scope is used only for LOCAL MCAST and BCAST. For UCAST destinations the scope will be set to RT_SCOPE_HOST when inet_select_address() is called.

```
1772   if (LOCAL_MCAST(oldkey->dst) ||
1773       oldkey->dst == 0xFFFFFFFF) {
1774       if (!key.src)
1775           key.src = inet_select_addr(dev_out, 0,
1776                           RT_SCOPE_LINK);
1777           goto make_route;
1778   }
```

**key.oif specified and key.src not specified**

Recall that this code block is executed only if the routing key specified an output interface and that the objective is to find an IP source address that is in some sense compatible with the specified output device. We just dispensed with local multicast and broadcast destination addresses. If the destination is general MULTICAST, then the address is selected from the output device using the key’s scope. If the destination is unspecified, the scope RT_SCOPE_HOST is passed to inet_select_addr().

```
1778   if (!key.src) {
1779       if (MULTICAST(oldkey->dst))
1780           key.src = inet_select_addr(dev_out, 0,
1781                           key.scope);
1782       else if (!oldkey->dst)
1783           key.src = inet_select_addr(dev_out, 0,
1784                           RT_SCOPE_HOST);
1785   }
1786 /* if (oldkey->oif) */
1787```
Handling unspecified destination

If the destination address is unspecified, the destination is set to the source address (which is presumably on this machine). If the source is also NULL then they are both set to the loopback address.

```
1788   if (!key.dst) {
1789       key.dst = key.src;
1790   
1791       if (!key.dst)
1792         key.dst= key.src = htonl(INADDR_LOOPBACK);
```

If an output device is held, because an output interface was specified, then it must be returned here because the loopback device must be used instead of the one specified.

```
1792   if (dev_out)
1793     dev_put(dev_out);
```

Use loopback device for sending packet to this machine.

```
1794   dev_out = &loopback_dev;
1795   dev_hold(dev_out);
1796   key.oif = loopback_dev.ifindex;
1797   res.type = RTN_LOCAL;
1798   flags |= RTCF_LOCAL;
1799   goto make_route;
1800 }
```
Building a route to a specified destination address

Finally, the function `fib_lookup()` defined in `include/net/ip_fib.h` is invoked to try to resolve the destination address. As we will see it will try to resolve the route first in the `local_table` and if that doesn't work, it will try the `main table`.

1802  if (fib_lookup(&key, &res)) {
1803       res.fi = NULL;

Falling into this block implies that the `fib_lookup` failed. If an output interface was specified, it is still possible to send the packet as described in the comment below.

Apparently, routing tables are wrong. Assume, that the destination is on link. WHY? -- DW. Because we are allowed to send to iface even if it has NO routes and NO assigned addresses. When oif is specified, routing ables are looked up with only one purpose: to catch if destination is gatewayed, rather than direct. Moreover, if MSG_DONTROUTE is set, we send packet, ignoring both routing tables and ifaddr state. --ANK

1804  if (oldkey->oif) {
1805      if (key.src == 0)
1806          key.src = inet_select_addr(dev_out, 0,
1807                                      RT_SCOPE_LINK);
1808          res.type = RTN_UNICAST;
1809          goto make_route;
1810  }

Reaching this point indicates that the FIB lookup failed and no output interface was specified. Its not clear how `dev_out` could be held under these conditions, but just to be safe it is checked.

1829  if (dev_out)
1830      dev_put(dev_out);
1831  err = -ENETUNREACH;
1832  goto out;
1833  }
Successful return from `fib_lookup`

Arrival here implies that the lookup succeeded. Thus `res` now points to a dynamically allocated `fib_result` which must be freed before returning to prevent a memory leak.

```c
1834    free_res = 1;
1835
```

It's not clear how the type could be NAT and why that is bad.

```c
1836    if (res.type == RTN_NAT)
1837        goto e_inval;
1838
```

Local route types

If this packet is routed locally (RTN_LOCAL), the destination and source host are the same and the loopback device should be used.

```c
1839    if (res.type == RTN_LOCAL) {
1840        if (!key.src)
1841            key.src = key.dst;
1842        if (dev_out)
1843            dev_put(dev_out);
1844        dev_out = &loopback_dev;
1845        dev_hold(dev_out);
1846        key.oif = dev_out->ifindex;
1847        if (res.fi)
1848            fib_info_put(res.fi);
1849        res.fi = NULL;
1850        flags |= RTCF_LOCAL;
1851        goto make_route;
1852    }
1853
```

Release the `fib_info` reference with `fib_info_put()`.

```c
1854    if (res.fi->fib_nhs > 1 && key.oif == 0)
1855        fib_select_multipath(&key, &res);
1856    else
1857        #endif
1858```

18
Default route selection

If the prefix length is 0 (implying default route), and the type is UNICAST, and no output interface index was specified then its necessary to select among (possibly multiple) default routes. No one would ever believe how hard this will be!

```c
1859     if(!res.prefixlen && res.type == RTN_UNICAST && !key.oif)
1860         fib_select_default(&key, &res);
```

Source address remains unspecified

If the source IP address remains NULL, an attempt is made to derive the source address from the fib_prefsrc field of the fib_info structure. If that field is also NULL, then our old friend inet_select_addr() is asked to recover it. The fib_info (see fib_waco.pdf) will normally have a non-zero preferred source field which is its first ifaddr.

If it does not, the FIB_RES_GW() IP address is passed to inet_select_addr(). Although this address should be on another host, the address matching logic in inet_select_address matches with respect to the netmask associated with the interface. Therefore, if my gateway is 130.127.48.1 and one of my interfaces owns the address 130.127.48.128/23 that address will be correctly selected as the source address for the outgoing packet. If inet_select_address() can't match the specified address it will search all net_devices for an interface with an ifa whose scope is <= res->scope of the proper scope and return the first one of those that it finds. The ifa scope of real interfaces is always 0 and the scope of the loopback interface is always 254.

```c
1861     if (!key.src)
1862         key.src = FIB_RES_PREFSRC(res);
```

FIB_RES_PREFSRC is a macro defined in include/net/ip_fib.h

```c
111 #define FIB_RES_PREFSRC(res)((res).fi->fib_prefsrc ? :
       __fib_res_prefsrc(&res))
624 u32 __fib_res_prefsrc(struct fib_result *res)
625 {
626     return inet_select_addr(FIB_RES_DEV(*res),
       FIB_RES_GW(*res), res->scope);
627 }
```
If a net device is held in `dev_out`, release it here.

```c
if (dev_out)
    dev_put(dev_out);
```

Set the value of `key.oif` from the `net_device` pointed to by the `fib_info` structure than lives in the `res` structure.

```c
dev_out = FIB_RES_DEV(res);
dev_hold(dev_out);
key.oif = dev_out->ifindex;
```

### Installing the route in the route cache

Before the route cache element is created it is necessary to clean up some issues pertaining to broadcast and multicast routes. First, it is ensured that if the source address is a loopback address then the selected output device carries the `IFF_LOOPBACK` flag.

```c
make_route:
if (LOOPBACK(key.src) && !((dev_out->flags&IFF_LOOPBACK))
    goto e_inval;
if (key.dst == 0xFFFFFFFF)
    res.type = RTN_BROADCAST;
else if (MULTICAST(key.dst))
    res.type = RTN_MULTICAST;
else if (BADCLASS(key.dst) || ZERONET(key.dst))
    goto e_inval;
if (dev_out->flags & IFF_LOOPBACK)
    flags |= RTCF_LOCAL;
```
If the result type is BROADCAST, then any fib_info structure that is held is released.

```
1885     if (res.type == RTN_BROADCAST) {
1886         flags |= RTCF_BROADCAST | RTCF_LOCAL;
1887         if (res.fi) {
1888             fib_info_put(res.fi);
1889             res.fi = NULL;
1890         }
1891     } else if (res.type == RTN_MULTICAST) {
1892         flags |= RTCF_MULTICAST | RTCF_LOCAL;
1893         read_lock(&inetdev_lock);
1894         if (!__in_dev_get(dev_out) ||
1895             !ip_check_mc(__in_dev_get(dev_out),
1896             oldkey->dst))
1897             flags &= ~RTCF_LOCAL;
1898         read_unlock(&inetdev_lock);
1899         /* If multicast route do not exist use
1900            default one, but do not gateway in
1901            this case. Yes, it is hack.
1902         */
1903         if (res.fi && res.prefixlen < 4) {
1904             fib_info_put(res.fi);
1905             res.fi = NULL;
1906     }
```
Creating the new route cache entry

1908 rth = dst_alloc(&ipv4_dst_ops);
1909 if (!rth)
1910 goto e_nobufs;
1911
1912 atomic_set(&rth->u.dst.__refcnt, 1);

Copy (most of) the elements of the old key structure that was used to create the route to the key structure embedded the rth. The rth->key structure will be used in subsequent route cache lookups and must match the input key.

1913 rth->u.dst.flags = DST_HOST;
1914 rth->key.dst = oldkey->dst;
1915 rth->key.tos = tos;
1916 rth->key.src = oldkey->src;
1917 rth->key.iif = 0;
1918 rth->key.oif = oldkey->oif;
1919 #ifdef CONFIG_IP_ROUTE_FWMARK
1920 rth->key.fwmark = oldkey->fwmark;
1921 #endif

Copy the elements used to route the packet to the rt_ fields of the route cache element. These are the elements that are actually used in building and routing the packet.

1922 rth->rt_dst = key.dst;
1923 rth->rt_src = key.src;
1924 #ifdef CONFIG_IP_ROUTE_NAT
1925 rth->rt_dst_map = key.dst;
1926 rth->rt_src_map = key.src;
1927 #endif
1928 rth->rt_iif = oldkey->oif ? : dev_out->ifindex;
1929 rth->u.dst.dev = dev_out;
1930 dev_hold(dev_out);
1931 rth->rt_gateway = key.dst;
1932 rth->rt_spec_dst = key.src;

Setup the function that will be used to transmit the packet.

1934 rth->u.dst.output= ip_output;
1935
1936 rt_cache_stat[smp_processor_id()].out_slow_tot++;
If the flags indicate that this route terminates on this machine, then the input handler is set to `ip_local_deliver`.

```c
1938     if (flags & RTCF_LOCAL) {
1939         rth->u.dst.input = ip_local_deliver;
1940         rth->rt_spec_dst = key.dst;
1941     }
1942     if (flags & (RTCF_BROADCAST | RTCF_MULTICAST)) {
1943         rth->rt_spec_dst = key.src;
1944         if (flags & RTCF_LOCAL &&
1945             !(dev_out->flags & IFF_LOOPBACK)) {
1946             rth->u.dst.output = ip_mc_output;
1947             rt_cache_stat[smp_processor_id()].out_slow_mc++;
1948         }
1949     }
1950 #ifdef CONFIG_IP_MROUTE
1951     if (res.type == RTN_MULTICAST) {
1952         struct in_device *in_dev = in_dev_get(dev_out);
1953         if (in_dev) {
1954             if (IN_DEV_MFORWARD(in_dev) &&
1955                 !LOCAL_MCAST(oldkey->dst)) {
1956                 rth->u.dst.input = ip_mr_input;
1957                 rth->u.dst.output = ip_mc_output;
1958             }
1959         }
1960     }
1961 #endif
```

CONFIG_IP_MROUTE option is used if you want your machine to act as a router for IP packets that have multicast destination addresses.
The `rt_set_nexthop()` defined in `net/ipv4/route.c` sets next neighbour parameters including `pmtu` and `mss`.

```c
1963(rt_set_nexthop(rth, &res, 0);
```

On return to `ip_route_output_slow()`, use the source address, destination address, and tos to determine and return a hash value by invoking the `rt_hash_code()` function defined in `net/ipv4/route.c`. We had visited this function earlier in UDP connect and was called by the `ip_route_output_key()` function.

```c
1965 rth->rt_flags = flags;
1967 hash = rt_hash_code(oldkey->dst, oldkey->src ^ (oldkey->oif << 5), tos);
```

The hash code returned is used by `rt_intern_hash()` function to search in the respective hash queue of routing cache (`rt_hash_table`) to find an entry that matches the entry that was just created. The `rp` parameter was passed in to `ip_route_output_slow()` as the location at which a pointer to the new route cache entry should be returned.

```c
1968 err = rt_intern_hash(hash, rth, rp);
```

References to `fib_info` or `net_device` structures are released before returning.

```c
1969 done:
1970 if (free_res) fib_res_put(&res);
1971 if (dev_out) dev_put(dev_out);
1974 out: return err;
1975 done:}
1976 e_inval:
1977 err = -EINVAL;
1978 goto done;
1979 e_nobufs:
1980 err = -ENOBUFS;
1981 goto done;
1982 }
```
To summarize, the `ip_route_output_slow()` function does the following:

- Creates a route key structure.
- If the source address is specified, calls `ip_dev_find()` to determine the output device.
- If the `oif` is specified, use `dev_get_by_index` to retrieve output device and select source addr (if the dest address was not NULL).
- If the destination address is not specified, set up loopback
- Calls `fib_lookup()` to find route to destination.
- Allocates memory for new routing cache entry and initializes it.
- Calls `rt_set_nexthop()` to set up destination.
- Returns `rt_intern_hash()`, which creates a new route in the routing cache and creates a neighbour structure for the route.
Finding a *net_device* associated with a local IP address

The input parameter here is the *source* IP address associated with the route being setup.

```c
145 struct net_device *ip_dev_find(u32 addr)
146 {
147     struct rt_key key;
148     struct fib_result res;
149     struct net_device *dev = NULL;
150
151     memset(&key, 0, sizeof(key));
152     key.dst = addr;
153     #ifdef CONFIG_IP_MULTIPLE_TABLES
154         res.r = NULL;
155     #endif
156
157     if (!local_table ||
158         local_table->tb_lookup(local_table, &key, &res)) {
159         return NULL;
160     }
161     if (res.type != RTN_LOCAL)
162         goto out;
```

The first step in the process is to determine if the specified IP address actually exists in the local table. The variable *local_table* is a reference to the statically defined local table.

```c
ip_fib.h:
#define local_table (fib_tables[RT_TABLE_LOCAL])
```

Since the *source* address is being processed, it is necessary that the returned route type be RTN_LOCAL. This seems like one convoluted way to find if a host owns a particular IP address! If the route is not RTN_LOCAL, a jump is made to the tag *out* bypassing the code which normally sets up the return value, *dev*. The value of *dev* was initialized to NULL, and a return value of NULL will cause *ip_route_output_slow* to return failure.

```c
160     if (res.type != RTN_LOCAL)
161         goto out;
```
FIB_RES_DEV, a macro defined in include/net/ip_fib.h, extracts the struct netdevice pointer from the fib_info pointer contained in the results structure. Note that dev->refcnt is incremented here. Where the corresponding decrement occurs is not clear at present.

```
113 #define FIB_RES_DEV(res) (FIB_RES_NH(res).nh_dev)
106 #define FIB_RES_NH(res)  ((res).fi->fib_nh[0])
```

```
162    dev = FIB_RES_DEV(res);
163    if (dev)
164        atomic_inc(&dev->refcnt);
165```

The fib_res_put() function releases the reference to the fib_res structure.

```
166 out:
167    fib_res_put(&res);
168    return dev;
169 }
```

What is not well understood here is how routes dynamically become ``dead'' or come to have reference counts of 0. The best guess at the moment is that the fib_info structure is held by all but its creator for a very short interval of time. Nevertheless, it would be possible that whatever owned and normally keeps the reference count at 1 tried to delete the route while we owned it here. Thus when we release it, it really should go away, but qui sait.

```
268 static inline void fib_res_put(struct fib_result *res)
269 {
270    if (res->fi)
271        fib_info_put(res->fi);  
272    ifdef CONFIG_IP_MULTIPLE_TABLES
273    if (res->r)
274        fib_rule_put(res->r);  
275    endif
276 }
```
The FIB lookup mechanism

Since the destination may be on this host as well as elsewhere in the Internet, the fib_lookup() function calls tb_lookup() on both the local table and the main table. Both tb_lookup functions resolve to fn_hash_lookup which was encountered earlier. Since fn_hash_lookup() returns 0 on success and non-zero on failure. The operation fails only if both lookups fail. Theoretically, at least, the lookup should not succeed in both tables but if it does, it would appear that the local table has precedence.

155 static inline int fib_lookup(const struct rt_key *key,  
       struct fib_result *res)  
156 {  
157     if ((local_table->tb_lookup(local_table, key, res) &&  
158         main_table->tb_lookup(main_table, key, res))  
159     return -ENETUNREACH;  
160     return 0;  
161 }
Route table lookup with *fn_hash_lookup*

The call to `local_table->tb_lookup()` is a reference to the `fn_hash_lookup()` function. This function is used to determine if the destination entity identified by `key` exists in the specified table. All the `fib_tables` are searched by `zone` where a routing `zone` is the set of routing destinations that have the same length prefix (or equivalently netmask). The `fn_hash_lookup()` searches the specified table, starting with the most specific zone netmask looking for a match. The most specific existing zone is pointed by the `fn_zone_list` variable.

```c
static int
fn_hash_lookup (struct fib_table *tb, const struct rt_key *key, struct fib_result *res)
{
    int err;
    struct fn_zone *fz;
    struct fn_hash *t = (struct fn_hash*)tb->tb_data;

    for (fz = t->fn_zone_list; fz; fz = fz->fz_next) {
        struct fib_node *f;
        fn_key_t k = fz_key(key->dst, fz);
    }
```

This outer loop processes every non-empty zone associated with the `fib_table` in longest prefix first order. A new key is needed for each zone, because the key is the route `prefix`.

```c
read_lock(&fib_hash_lock);
for (fz = t->fn_zone_list; fz; fz = fz->fz_next) {
    struct fib_node *f;
    fn_key_t k = fz_key(key->dst, fz);
```

The `fz_key()` function, defined in `fib_hash.c`, builds a test key by and-ing the address with the zone's netmask. The structures `fn_key_t` and `fn_hash_idx_t` are simply unsigned integers representing IP prefixes and hash table indices respectively.

```c
typedef struct {
    u32 datum;
} fn_key_t;

typedef struct {
    u32 datum;
} fn_hash_idx_t;
```
static __inline__ fn_key_t fz_key(u32 dst, struct fn_zone *fz)
{
    fn_key_t k;
    k.datum = dst & FZ_MASK(fz);
    return k;
}

FZ_MASK is a macro defined in fib_hash.c

#define FZ_MASK(fz)     ((fz)->fz_mask)
On returning to `fn_hash_lookup()`, this inner loop traverses the list of `fib_node` structures associated with the hash bucket of the routing key searching for the first key match. To initiate this process `fz_chain()` is called to retrieve the address of the first `fib_node` in the chain.

It performs the hash function `fn_hash()` and ANDs this value with the zone's `fz_hashmask` to get an index into the zone's hash table of nodes. The syntax of this function is a bit dense. Note that `fn_hash` returns `fn_hash_idx_t` which was shown above to be a ``structure" consisting of a single unsigned int member called datum. That value is used as an index into the hash table structure associated with the routing zone yielding the required pointer to the `struct fib_node`.

```c
for (f = fz_chain(k, fz); f; f = f->fn_next) {

static __inline__ struct fib_node * fz_chain(fn_key_t key, struct fn_zone *fz)
135 { return fz->fz_hash[fn_hash(key, fz).datum];
136 }

static __inline__ fn_hash_idx_t fn_hash(fn_key_t key, struct fn_zone *fz)
110 { u32 h = ntohl(key.datum)>>(32 - fz->fz_order);
111     h ^= (h>>20);
112     h ^= (h>>10);
113     h ^= (h>>5);
114     h &= FZ_HASHMASK(fz);
115
116     return *(fn_hash_idx_t*)&h;
117 }
```

`FZ_HASHMASK` is a macro defined in `fib_hash.c`

```c
#define FZ_HASHMASK(fz) ((fz)->fz_hashmask)
```

`fn_hash_idx_t` is a structure containing the address as its element.

```c
typedef struct {
    u32 datum;
} fn_hash_idx_t;
```

```c
117    return *(fn_hash_idx_t*)&h;
118 }
```

31
Matching input key to \texttt{fib\_node} key

The first action of the inner loop is to compare search key with the key of the \texttt{struct fib\_node}. Recall that the variable \texttt{k} is an instance of \texttt{fn\_key\_t}, a structure of the single element \texttt{datum}, whose value was previously set to the target IP address and'ed with the netmask associated with the zone. From this we can infer that the value of \texttt{f->fn\_key} is the network address or CIDR network prefix associated with the routing table entity associated with this node. The nodes on any hash queue are apparently sorted in ascending order by prefix, and \texttt{fn\_key\_leq()} will return 1 if \texttt{k} < \texttt{fn\_key}. Therefore, if they do not match and if the search key value is greater than that of the node key, the search continues on to the next node. (Consider search key = 10 and node keys \{5, 7, 9, 11, 12\}. At the point we see that 10 \leq 11, it is known that the target key is not in this list.)

```c
281       if (!fn_key_eq(k, f->fn_key))
282               if (fn_key_leq(k, f->fn_key))
283                   break;
284               else
285                   continue;
286           }
```
Verifying *tos* OK

Arriving here implies that there has been a match. CONFIG_IP_ROUTE_TOS makes use of TOS value as routing key and so if there is a *tos* associated with the *fib_node* and it is not equal to the *tos* of the key, the match is discarded and the search continues.

287  #ifdef CONFIG_IP_ROUTE_TOS
288     if (f->fn_tos && f->fn_tos != key->tos)
289         continue;
290  #endif

The state information of the *fib_node* is updated and tested for Zombie status. Zombie nodes are considered non-usable and likely relate to deleted routes or dead interfaces. Very little state information is present in *fib_nodes*. Only 2 bits are defined:

80  #define FN_S_ZOMBIE 1
81  #define FN_S_ACCESSED 2

Verifying route alive

291         f->fn_state |= FN_S_ACCESSED;
292
293         if (f->fn_state & FN_S_ZOMBIE)
294             continue;
Scope testing

Recall that higher values of scope means more specific or constrained routing. Thus the node scope is required to be at least as specific as the requested route scope. If the fib_node scope is less than that of the scope of the key, then this node is also not usable. For ip_route_output_slow(), the value of key->scope will be RT_SCOPE_UNIVERSE (0) unless the RTO_ONLINK flag was set. In that case it will be RT_SCOPE_LINK (253). The scope value stored in the fib_node is 254 for local host addresses and 253 for link, broadcast, and multicast addresses.

```
295     if (f->fn_scope < key->scope)
296         continue;
297```

Semantic testing

Finally the fib_semantic_match() function is called to ensure that this fib_node is usable within the semantic constraints imposed by the route key.

```
298     err = fib_semantic_match(f->fn_type, 
                      FIB_INFO(f), key, res);
```

On return from fib_semantic_match(), if the source address was found to be acceptable, the res structure is filled with the type and scope elements copied from the fib_node structure and the prefix length is copied from the fn_zone structure.

```
299     if (err == 0) {
300         res->type = f->fn_type;
301         res->scope = f->fn_scope;
302         res->prefixlen = fz->fz_order;
303         goto out;
304     }
305     if (err < 0)
306         goto out;
307 }
308 err = 1;
309 out:
310     read_unlock(&fib_hash_lock);
311     return err;
312 }
313 ```
The **fib_semantic_match** function

The **fib_semantic_match()** function is defined in `net/ipv4/fib_semantics.c`. Its mission is to ensure that the candidate **fib_node** appears to represent an acceptable route. The tests include:

- ensuring that the route type is acceptable
- ensuring that the associated **fib_info's** view of the next hop is that it is alive,
- the **fib_nh's** view of the next hop is that its alive, and
- if the output interface is specified in the routing key, it is the same interface as the one associated with the next hop structure.

The **fib_props** table is a static table that maps values of route type (**RTN_**) that is contained in a **fib_node** to an error code and scope. Thus, **fib_semantic_match()** begins by ensuring that the value of **fn_type** that is passed in routeable. The route type is established when the route is created but may be dynamically adjusted if it is found that the route doesn't work.

```c
enum {
    RTN_UNSPEC,            /* Gateway or direct route */
    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 */
};
```
Validating the route type

The route type, as enumerated above, is used as an index into this table to recover an error code and actual route scope. An error code of 0 indicates that the route is usable.

```c
static struct {
    int     error;
    u8      scope;
} fib_props[RTA_MAX+1] = {
    { 0, RT_SCOPE_NOWHERE},         /* RTN_UNSPEC */
    { 0, RT_SCOPE_UNIVERSE},        /* RTN_UNICAST */
    { 0, RT_SCOPE_HOST},            /* RTN_LOCAL */
    { 0, RT_SCOPE_LINK},            /* RTN_BROADCAST */
    { 0, RT_SCOPE_LINK},            /* RTN_ANYCAST */
    { 0, RT_SCOPE_UNIVERSE},        /* RTN_MULTICAST */
    { -EINVAL, RT_SCOPE_UNIVERSE},  /* RTN_BLACKHOLE */
    { -EHOSTUNREACH, RT_SCOPE_UNIVERSE},/* RTN_UNREACHABLE */
    { -EACCES, RT_SCOPE_UNIVERSE},  /* RTN_PROHIBIT */
    { -EAGAIN, RT_SCOPE_UNIVERSE},  /* RTN_THROW */
    { 0, RT_SCOPE_HOST},            /* RTN_NAT */
    { -EINVAL, RT_SCOPE_NOWHERE},   /* RTN_NAT */
    { -EINVAL, RT_SCOPE_NOWHERE}    /* RTN_XRESOLVE */
};
```

If the type is acceptable, the remainder of the operation proceeds.

```c
    int err = fib_props[type].error;
    if (err == 0) {
        if (fi->fib_flags & RTNH_F_DEAD)
            return 1;
```
The *fib_info* structure is next connected to the results structure, and then route type dependent processing occurs.

```c
578   res->fi = fi;
579
580 #ifdef CONFIG_IP_ROUTE_NAT
581     switch (type) {
582       case RTN_NAT:
583         FIB_RES_RESET(*res);
584         atomic_inc(&fi->fib_clntref);
585         return 0;
586     #endif
```

Only the NAT type route is distinguished for the purposes of route semantics.

```c
587     case RTN_UNICAST:
588     case RTN_LOCAL:
589     case RTN_BROADCAST:
590     case RTN_ANYCAST:
591     case RTN_MULTICAST:
```

Check if a next hop is feasible from this node. The macros used in this loop depend upon whether or not multipath routing is enabled. If not, there can be only one next hop associated with a *fib_info* structure.

```c
57   #ifdef CONFIG_IP_ROUTE_MULTIPATH
58   #define for_nexthops(fi) { int nhsel; const struct fib_nh *nh; \n59     for (nhsel=0, nh = (fi)->fib_nh; nhsel < (fi)->fib_nhs; nh++, nhsel++)
60     #else /* CONFIG_IP_ROUTE_MULTIPATH */
61     /* Hope, that gcc will optimize it to get rid of dummy loop */
62     #define for_nexthops(fi) {int nhsel=0; const struct fib_nh *nh = (fi)->fib_nh;
63     for (nhsel=0; nhsel < 1; nhsel++)
64     #endif /* CONFIG_IP_RO
```

```c
```
Evaluating the health of the next hop.

As noted on the previous page the behavior of `for_nexthops` is dependent on whether MULTIPATH routing is enabled. Without MULTIPATH a `struct fib_info` will contain only one `struct fib_nh`.

```
for_nexthops(fi) {
    if (nh->nh_flags & RTNH_F_DEAD)
        continue;
}
```

If the route key requires a specific output interface and that is not the output interface associated with this `fib_nh` then the route is not usable. Note that there is a subtle difference between this situation and the earlier case in which there was a mismatch between the source IP address and the `oif`. The `break` is taken if the route is usable.

```
if (!key->oif || key->oif == nh->nh_oif)
    break;
```

The `CONFIG_IP_ROUTE_MULTIPATH` option allows the routing tables to specify alternative paths to travel for a given packet. The router considers all these paths to be of equal "cost" and chooses one of them in a non-deterministic fashion when selecting a route. How is this done??

```
#define CONFIG_IP_ROUTE_MULTIPATH
if (nhsel < fi->fib_nhs) {
    res->nh_sel = nhsel;
    atomic_inc(&fi->fib_clntref);
    return 0;
} else
```

For non multi-path routing, this is the success return point. The loop will have been exited via the `break` and so `nhsel` will remain 0. The reference counter of the `fib_info` structure is incremented here.

```
if (nhsel < 1) {
    atomic_inc(&fi->fib_clntref);
    return 0;
}
```

38
This *endfor* is misleading. The actual loop ended at line 597. This closes the block in which the local variables preceding the *for* loop are declared.

```
610          endfor_nexthops(fi);
```

Falling out of the loop implies no *fib_nh* with acceptable semantics was found.

```
611          res->fi = NULL;
612          return 1;
613          default:
614          res->fi = NULL;
615          printk(KERN_DEBUG "impossible 102\n");
616          return -EINVAL;
617      }
618  }
619  return err;
620 }
```

The *fib_clntref* is a reference counter and when its value reaches zero, the *struct fib_info* is deleted. In this context *fib_clntref* was incremented in the function *fib_semantic_match()*.

```
262 static inline void fib_info_put(struct fib_info *fi)
263 {
264      if (atomic_dec_and_test(&fi->fib_clntref))
265          free_fib_info(fi);
266 }
```

```
106 void free_fib_info(struct fib_info *fi)
107 { 
108      if (fi->fib_dead == 0) {
109          printk("Freeing alive fib_info %p\n", fi);
110          return;
111      }
```

Unless multipath routing is enabled, *change_nexthops()* will cause the enclosed block to be executed exactly one time and this *fib_info* structure's claim on the *net_device* will be dropped.

```
112      change_nexthops(fi) {
113          if (nh->nh_dev)
114              dev_put(nh->nh_dev);
115          nh->nh_dev = NULL;
116      } endfor_nexthops(fi);
117      fib_info_cnt--;
118      kfree(fi);
119  }
```
Release a *fib_rule* structure.

```c
152 void fib_rule_put(struct fib_rule *r) {
153     if (atomic_dec_and_test(&r->r_clntref)) {
154         if (r->r_dead)
155             kfree(r);
156         else
157             printk("Freeing alive rule %p\n", r);
158     }
159 }
160 }
```
IP specific device structures

The \texttt{in_dev_get()} function returns a pointer to the \texttt{struct in_device} that is associated with a given \texttt{net_device}.

\begin{verbatim}
133 __in_dev_get(const struct net_device *dev)
134 {
135    return (struct in_device*)dev->ip_ptr;
136 }
137
26 struct in_device
27 {
28    struct net_device *dev;
29    atomic_t refcnt;
30    rwlock_t lock;
31    int dead;
32    struct in_ifaddr *ifa_list; /* IP ifaddr chain */
33    struct ip_mc_list *mc_list; /* IP mcst filter chain */
34    unsigned long mr_v1_seen;
35    struct neigh_parms *arp_parms;
36    struct ipv4_devconf cnf;
37 }
\end{verbatim}

Each physical \texttt{net_device} may be assigned alias IP addresses and names (eth0:1 eth0:2, .. etc). The name is stored in \texttt{ifa_label}. Each alias is represented by an instance of the \texttt{struct in_ifaddr}. The distinction between \texttt{ifa_local} and \texttt{ifa_address} is not well understood. Empirical analysis of "normal" network configurations fails to disclose any instances in which \texttt{ifa_local} and \texttt{ifa_address} differ.

\begin{verbatim}
60 struct in_ifaddr
61 {
62    struct in_ifaddr *ifa_next;
63    struct in_device *ifa_dev;
64    u32 ifa_local;
65    u32 ifa_address;
66    u32 ifa_mask;
67    u32 ifa_broadcast;
68    u32 ifa_anycast;
69    unsigned char ifa_scope;
70    unsigned char ifa_flags;
71    unsigned char ifa_prefixlen;
72    char ifa_label[IFNAMSIZ];
73 }
\end{verbatim}
Interface creation

When a new interface is created by the `inet_rtm_newaddr(struct sk_buff *skb, struct nlmsghdr *nlh, void *arg)` function in `net/ipv4/devinet.c`, the two addresses are set to the values passed in via the netlinks protocol message (don't ask).

if (rta[IFA_ADDRESS-1] == NULL)
    rta[IFA_ADDRESS-1] = rta[IFA_LOCAL-1];
memcpy(&ifa->ifa_local, RTA_DATA(rta[IFA_LOCAL-1]), 4);
memcpy(&ifa->ifa_address,RTA_DATA(rta[IFA_ADDRESS-1]),4);
ifa->ifa_prefixlen = ifm->ifa_prefixlen;

Selecting an IP address

When the destination address is LOCAL multicast or broadcast, the `inet_select_addr()` function, defined in `net/ipv4/devinet.c`, returns a local address associated with the specified output device. In this case dev points to the output device, the dst address is NULL and the scope is RT_SCOPE_LINK. The return value is the selected IP address or is NULL upon failure.

u32 inet_select_addr(const struct net_device *dev, u32 dst, int scope) {
    u32 addr = 0;
    struct in_device *in_dev;
    read_lock(&inetdev_lock);
    in_dev = __in_dev_get(dev);
    if (in_dev == NULL) {
        read_unlock(&inetdev_lock);
        return 0;
    }
}
Identifying acceptable scope

At this point in_dev points to a valid in_device structure. The for_primary_ifa macro runs the interface address chain associated with the in_device. Recall that routing scope values are ordered with the most specific scope (i.e. this host) having the highest value.

For broadcasts or unspecified destination addresses, the scope passed was RT_SCOPE_LINK. For other cases, the scope was inherited from the key and thus could be RT_SCOPE_UNIVERSE or RT_SCOPE_LINK depending upon whether or not RTO_ONLINK was specified.

Thus interfaces having a more specific address scope (HOST or NOWHERE) are rejected. In practice (see fib_waco.pdf) values of ifa_scope appear to always be either 254 for loopback addresses or zero for IP host or network addresses. Thus in the scope matching logic below, physical interfaces are always acceptable and the loopback interface is acceptable only if the input scope is also HOST.

```
730    read_lock(&in_dev->lock);
731    for_primary_ifa(in_dev) {
732        if (ifa->ifa_scope > scope)
733            continue;
```
IP address matching

The address matching logic is with respect to the network mask associated with the `in_ifaddr` structure. If the value of `dst` (which in this code is the value of `key.src`) that was passed in was 0, `!dst` is true and the value of `addr` is set to the `ifa_local` field of the interface. Note that the address matching test is against `ifa_address`, but if a match occurs `addr` is set to `ifa_local`.

```c
if (!dst || inet_ifa_match(dst, ifa)) {
    addr = ifa->ifa_local;
    break;
}
```

For the control path we are investigating it appears that `addr` should always be non-zero here and thus a return should take place.

```c
if (addr)
    return addr;
```
Acceptable address not found

If control should reach here, it indicates that *dst* was non-zero and didn't match the *ifa_address* field of any interface address structure associated with the device. *dev_base* is a global variable pointing to the list of all instances of *struct net_device*. Here the selection criterion appears to be finding an interface whose scope is *not* LINK and whose scope is numerically less than or equal to the scope that was passed in.

```
746 747 /* Not loopback addresses on loopback should be preferred
748     in this case. It is important that lo is the 1st intf
749     in dev_base list. */
750  read_lock(&dev_base_lock);
751  read_lock(&inetdev_lock);
752  for (dev = dev_base; dev; dev = dev->next) {
753       if ((in_dev=__in_dev_get(dev)) == NULL)
754           continue;
755       read_lock(&in_dev->lock);
756       for_primary_ifa(in_dev) {
757         if (ifa->ifa_scope != RT_SCOPE_LINK &&
758                ifa->ifa_scope <= scope) {
759           read_unlock(&in_dev->lock);
760           read_unlock(&inetdev_lock);
761           read_unlock(&dev_base_lock);
762           return ifa->ifa_local;
763         }
764       } endfor_ifa(in_dev);
765       read_unlock(&in_dev->lock);
766   } read_unlock(&inetdev_lock);
767   read_unlock(&dev_base_lock);
768  }
769 770 771

Return failure if an acceptable address cannot be found.

```

```
772  return 0;
773 }
774

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

```
**Default route selection**

The `fib_select_default()` function is defined in `include/net/ip_fib.h`. The function returns immediately if the initial `if` condition is false. The `FIB_RES_GW()` macro will return the `nh_gw` IP address from the `fib_nh` structure pointed to by the `fib_info` structure that is accessed via the `fib_result` structure that is passed in. So the call to `tb_select_default()` occurs only if there is already is a known gateway address and that gateway in on link.

As noted earlier, three different entities, the node, the next hop, and the interface all have scope values. The value of `nh_scope` appears to be the most specific, having the value 254 for all local interface entries and local net entries in the main table. *It does appear to have a 253 value for those table entries that do specify routing through a gateway either to a remote net or the default route.*

```
163 static inline void fib_select_default(const struct rt_key *key, struct fib_result *res) {
164     if (FIB_RES_GW(*res) &&
165         FIB_RES_NH(*res).nh_scope == RT_SCOPE_LINK) {
166         main_table->tb_select_default(main_table, key, res);
167     }
```
The \textit{fn_hash_select_default} function

This extremely nasty function may or may not override the \textit{fib_info} in the result structure. The basic idea appears to be to try to find a \textit{fib_info} whose next hop is known to be REACHABLE. If that is not possible, it tries to use VALID next hops in a round robin type way.

340 static void
341 fn_hash_select_default(struct fib_table *tb,
342    const struct rt_key *key, struct fib_result *res)
343 {
344     int order, last_idx;
345     struct fib_node *f;
346     struct fib_info *fi = NULL;
347     struct fib_info *last_resort;
348     struct fn_hash *t = (struct fn_hash*)tb->tb_data;
349     struct fn_zone *fz = t->fn_zones[0];
350     if (fz == NULL)
351         return;
352     last_idx = -1;
353     last_resort = NULL;
354     order = -1;
355     read_lock(&fib_hash_lock);
The main search loop

Iterate through all the nodes for the order zero zone. Needless to say this implies the existence of more than one default route. To successfully find something here requires finding a nh_scope of RT_SCOPE_LINK. Through this loop it will be the case that fi points to the fib_info that is associated with the previous fib_node. Gatewayed routes appear to have NODE scope of universe but a NH scope of link (253) as shown in this example from fib_waco.

NODE at c74b8d80. Key 0a060000 tos 0 type 1 scope 0 state 0
NH at c57d342c flg 00 scope 253 oif 3 gw 0a080006

```
for (f = fz->fz_hash[0]; f; f = f->fn_next) {
    struct fib_info *next_fi = FIB_INFO(f);

    if ((f->fn_state & FN_S_ZOMBIE) ||
        f->fn_scope != res->scope ||
        f->fn_type != RTN_UNICAST)
        continue;

    if (next_fi->fib_priority > res->fi->fib_priority)
        break;

    if (!next_fi->fib_nh[0].nh_gw ||
        next_fi->fib_nh[0].nh_scope!= RT_SCOPE_LINK)
        continue;

    f->fn_state |= FN_S_ACCESSED;
```

Early out for priority

Since we saw earlier that fib_nodes seemed to be linked in key order and not according to the priority of the associated fib_info, it is not clear why a high fib_priority causes an early exit from the loop when there might be a still higher priority route later on in the list.

Ensure scope acceptable

To be usable the fib_info must have an associated nh_gw with RT_SCOPE_LINK. All gatewayed routes will satisfy this requirement and the default route must be gatewayed.

```
if (!next_fi->fib_nh[0].nh_gw ||
    next_fi->fib_nh[0].nh_scope!= RT_SCOPE_LINK)
    continue;

f->fn_state |= FN_S_ACCESSED;
```
Table \( fi \) not equal \( res->fi \) exit.

The first time through this loop \( fi \) will be NULL. The loop is exited if the \( fib_info \) that is under consideration is not the one pointed the \( fib_result \) structure. Normally, one would expect that \( res->fi \) points to the first \( fi \) and so the loop will not be exited.

\[
\begin{align*}
&372 \text{ if } (fi == \text{NULL}) \{ \\
&373 \quad \text{if } (next_fi != res->fi) \\
&374 \quad \quad \text{break;}
\end{align*}
\]

Validating ARP state of previous \( fi \).

If this is not the first iteration of the loop, the value of \( fi \) will not be NULL, and the \( fib\text{-}detect\_death() \) function is called to see if the IP address is still in the ARP cache. A return code of 0 means that either:

1. a NUD\_REACHABLE ARP cache entry was matched to the IP address and
2. that a NUD\_VALID neighbour was found or and the current value of \( order \) is not the same as the global variable \( fn\_hash\_last\_dflt \). The variable \( fn\_hash\_last\_dflt \) keeps the relative position on the zone 0 list of the \( fib\_node \) that was last used to satisfy a default route.

\[
\begin{align*}
&375 \quad \text{else if } (!\text{fib}\_detect\_death(fi, order,} \\
&376 \quad \quad \quad \text{&last\_resort, &last\_idx)) \{ \\
&377 \quad \quad \quad \text{if } (res->fi) \\
&378 \quad \quad \quad \quad \text{fib\_info\_put(res->fi);} \\
&379 \quad \quad \quad \quad \text{res->fi }= \text{fi;} \\
&380 \quad \quad \quad \quad \text{atomic\_inc(&fi->fib\_clntref);} \\
&381 \quad \quad \quad \quad \text{fn\_hash\_last\_dflt }= \text{order;} \\
&382 \quad \quad \quad \text{goto out;}
\end{align*}
\]

Update \( fi \) and \( order \)

On the first iteration of the loop \( fi \) is NULL and this block is executed if \( next\_fi == res\_fi \). On subsequent iterations it will be executed if \( fib\text{-}detect\_death() \) returns 1. In these latter cases it is conceivably possible that \( fi == next\_fi \) already because two \( fib\_nodes \) may share a \( fib\_info \).

\[
\begin{align*}
&383 \quad \text{fi }= \text{next\_fi;} \\
&384 \quad \text{order++;}
\end{align*}
\]
Break out/fall out of lookup loop

This point will be reached only if

1. the \textit{break} at line 374 were taken on the first iteration of the loop (break out) or
2. there was only a single \textit{fn} and the \textit{next\_fi == res\_fi} (fall out), or
3. there were multiple \textit{fn}'s and \textit{fib\_detect\_death()} returned 1 on all calls (fall out)
4. the \textit{break} for priority at line 367 were take.

The \textit{if} below handles the first two cases and could also apply to case (4) if the break were taken on the first or second iteration of the loop! In case (1) \textit{(order == -1)} and \textit{(fi == NULL)}. In case (2) \textit{(order == 0)} and \textit{\textit{fi == next\_fi = res\_fi}}. In this case since the first \textit{fib\_node} in the chain is being used, \textit{fn\_hash\_default} is set back to -1. In this and only this case it appears that a call to \textit{fib\_detect\_death()} will not be made!

\begin{verbatim}
387     if (order<=0 || fi==NULL) {
388         fn_hash_last_dflt = -1;
389         goto out;
390     }
391
This block will be executed in case (3) above and case(4) when the number of iterations of the loop exceeds 2. In case (3) since \textit{fi} is set to \textit{next\_fi} at the bottom of the loop, here \textit{fi} points to the \textit{fib\_info} associated with the last \textit{fn} in the hash chain. If this \textit{fib\_info} is found acceptable by \textit{fib\_detect\_death()} it will be used.

392     if (!fib\_detect\_death(fi, order, &last\_resort, &last\_idx)) {
393         if (res->fi)
394             fib\_info\_put(res->fi);
395         res->fi = fi;
396         atomic\_inc(&fi->fib\_clntref);
397         fn\_hash\_last\_dflt = order;
398         goto out;
399     }
400
\end{verbatim}
fib_detect_death() returns non-zero in case

Reaching this point means that all calls fib_detect_death() returned 1. If fib_detect_death() found a last resort fib_info it will be used.

```
  if (last_idx >= 0) {
      if (res->fi)           
          fib_info_put(res->fi);
      res->fi = last_resort;
      if (last_resort)       
          atomic_inc(&last_resort->fib_clntref);
  }
  fn_hash_last_dflt = last_idx;
  read_unlock(&fib_hash_lock);
```
The `fib_detect_death()` function

The `fib_detect_death()` function attempts to lookup the IP address specified in `nh_gw` in the ARP cache and it may or may not update the `last_idx` and `last_resort`.

```c
317 static int fib_detect_death(struct fib_info *fi, int order,
318                               struct fib_info **last_resort, int *last_idx)
319 {
320   struct neighbour *n;
321   int state = NUD_NONE;
322   n = neigh_lookup(&arp_tbl, &fi->fib_nh[0].nh_gw, fi->fib_dev);

After returning from `neigh_lookup()` that returns a pointer to an entry in the neighbour table, the state of the neighbor is checked. On return, if an ARP cache entry was found its `state` is saved.

```c
324   if (n) {
325     state = n->nud_state;
326     neigh_release(n);
327   }
```

Reachable is the strongest possible state. It means that an ARP request has been recently sent and recently responded to.

```c
328   if (state==NUD_REACHABLE)
329     return 0;
```

`NUD_VALID` is a weaker composite state that includes `NUD_REACHABLE` and some other states that are entered when an ARP cache entry has timed out. If the `state` is valid and the `order` indicates this `fib_info` is not the one most recently used then it is accepted. Otherwise if the `state` is valid or it is the case that both the `*last_resort` pointer has not been set yet and this `fib_node` is farther along in the list than the last one used in default routing, then this `fib_info` is saved as a `last_resort` route.

```c
330   if ((state & NUD_VALID) && order != fn_hash_last_dflt)
331     return 0;
332   if ((state & NUD_VALID) || (*last_idx<0 && order > fn_hash_last_dflt)) {
334     *last_resort = fi;
335     *last_idx = order;
336   }
337   return 1;
338 }
```
Establishing route parameters

```c
1180 static void rt_set_nexthop(struct rtable *rt, struct
1181       fib_result *res, u32 itag)
1182 {
1183   struct fib_info *fi = res->fi;
1184
The bulk of this code seems to be attempting to address potential problems associated with missing
or invalid elements in the fib_info structure.

```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));
```fib_mtu is actually a macro referencing the RTAX_MTU element of the fib_metrics array. If the
value is zero it is copied from the net device. Oddly, it appears that rt->u.dst.pmtu has not been
previously set in this module... so it is also set in the else clause!

```if (fi->fib_mtu == 0) {
```1190   if (rt->u.dst.pmtu = rt->u.dst.dev->mtu;
1191     if (rt->u.dst.mxlock & (1 << RTAX_MTU) &&
1192         rt->rt_gateway != rt->rt_dst &&
1193         rt->u.dst.pmtu > 576)
1194       rt->u.dst.pmtu = 576;
1195   }
```1197 ifdef CONFIG_NET_CLS_ROUTE
1198   rt->u.dst.tclassid = FIB_RES_NH(*res).nh_tclassid;
1199 #endif
```1200 } else
```1201   rt->u.dst.pmtu = rt->u.dst.dev->mtu;
1202 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,
1207     rt->u.dst.dev->mtu - 40,
1208     ip_rt_min_advmss);
```

53
if (rt->u.dst.advmss > 65535 - 40)
rt->u.dst.advmss = 65535 - 40;
#endif
#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;
}

Inserting the entry into the hash queue

The hash code returned is used by rt_intern_hash() function to search in the respective hash queue of routing cache (rt_hash_table) to find an entry that matches the entry that was just created. The rp parameter was passed in to ip_route_output_slow() as the location at which a pointer to the new route cache entry should be returned.

rt_intern_hash(unsigned hash, struct rtable *rt, struct rtable **rp) {
    struct rtable *rth, **rthp;
    unsigned long now = jiffies;
    int attempts = !in_softirq();
    restart:
    rthp = &rt_hash_table[hash].chain;
    write_lock_bh(&rt_hash_table[hash].lock);
This loop appears to be looking for the possible case that the route already exists! This could conceivably occur due to race conditions involving multiple callers of `ip_route_output()`. If an existing entry with the same key is found, the existing entry is used and the newly created one is dropped.

```c
while ((rth = *rthp) != NULL) {
    if (memcmp(&rth->key, &rt->key, sizeof(rt->key)) == 0) {
        /* Put it first */
        *rthp = rth->u.rt_next;
        rth->u.rt_next = rt_hash_table[hash].chain;
        rt_hash_table[hash].chain = rth;
        rth->u.dst.__use++;
        dst_hold(&rth->u.dst);
        rth->u.dst.lastuse = now;
        write_unlock_bh(&rt_hash_table[hash].lock);
        rt_drop(rt);
        *rp = rth;
        return 0;
    }
}

/* Try to bind route to arp only if it is output route or unicast forwarding path. */
if (rt->rt_type == RTN_UNICAST || rt->key.iif == 0) {
    int err = arp_bind_neighbour(&rt->u.dst);
    if (err) {
        write_unlock_bh(&rt_hash_table[hash].lock);
        if (err != -ENOBUFS) {
            rt_drop(rt);
            return err;
        }
    }
}
```

Update the reference count and the last use of the existing entry.
/* Neighbour tables are full and nothing can be released. Try to shrink route cache, it is most likely it holds some neighbour records. */

if (attempts-- > 0) {
    int saved_elasticity = ip_rt_gc_elasticity;
    int saved_int = ip_rt_gc_min_interval;
    ip_rt_gc_elasticity = 1;
    ip_rt_gc_min_interval = 0;
    rt_garbage_collect();
    ip_rt_gc_min_interval = saved_int;
    ip_rt_gc_elasticity = saved_elasticity;
    goto restart;
}

if (net_ratelimit())
    printk("Neighbour table overflow.\n");
    rt_drop(rt);
    return -ENOBUFS;
}

Here the new route is inserted at the head of the hash queue.

rt->u.rt_next = rt_hash_table[hash].chain;
#if RT_CACHE_DEBUG >= 2
    if (rt->u.rt_next) {
        struct rtable *trt;
        printk("rt_cache @%02x: %u.%u.%u.%u", hash,
           NIPQUAD(rt->rt_dst));
        for (trt = rt->u.rt_next; trt;
             trt = trt->u.rt_next)
            printk(" . %u.%u.%u.%u", NIPQUAD(trt->rt_dst));
        printk("\n");
    }
#endif
    rt_hash_table[hash].chain = rt;
    write_unlock_bh(&rt_hash_table[hash].lock);
    *rp = rt;
    return 0;