CHAPTER 5

The IPv4 Routing Subsystem

Chapter 4 discussed the IPv4 subsystem. In this chapter and the next I discuss one of the most important Linux subsystems, the routing subsystem, and its implementation in Linux. The Linux routing subsystem is used in a wide range of routers—from home and small office routers, to enterprise routers (which connect organizations or ISPs) and core high speed routers on the Internet backbone. It is impossible to imagine the modern world without these devices. The discussion in these two chapters is limited to the IPv4 routing subsystem, which is very similar to the IPv6 implementation. This chapter is mainly an introduction and presents the main data structures that are used by the IPv4 routing subsystem, like the routing tables, the Forwarding Information Base (FIB) info and the FIB alias, the FIB TRIE and more. (TRIE is not an acronym, by the way, but it is derived from the word *retrieval*). The TRIE is a data structure, a special tree that replaced the FIB hash table. You will learn how a lookup in the routing subsystem is performed, how and when ICMP Redirect messages are generated, and about the removal of the routing cache code. Note that the discussion and the code examples in this chapter relate to kernel 3.9, except for two sections where a different kernel version is explicitly mentioned.

Forwarding and the FIB

One of the important goals of the Linux Networking stack is to forward traffic. This is relevant especially when discussing core routers, which operate in the Internet backbone. The Linux IP stack layer, responsible for forwarding packets and maintaining the forwarding database, is called the routing subsystem. For small networks, management of the FIB can be done by a system administrator, because most of the network topology is static. When discussing core routers, the situation is a bit different, as the topology is dynamic and there is a vast amount of ever-changing information. In this case, management of the FIB is done usually by userspace routing daemons, sometimes in conjunction with special hardware enhancements. These userspace daemons usually maintain routing tables of their own, which sometimes interact with the kernel routing tables.

Let's start with the basics: what is routing? Take a look at a very simple forwarding example: you have two Ethernet Local Area Networks, LAN1 and LAN2. On LAN1 you have a subnet of 192.168.1.0/24, and on LAN2 you have a subnet of 192.168.2.0/24. There is a machine between these two LANs, which will be called a "forwarding router." There are two Ethernet network interface cards (NICs) in the forwarding router. The network interface connected to LAN1 is `eth0` and has an IP address of 192.168.1.200, and the network interface connected to LAN2 is `eth1` and has an IP address of 192.168.2.200, as you can see in Figure 5-1. For the sake of simplicity, let's assume that no firewall daemon runs on the forwarding router. You start sending traffic from LAN1, which is destined to LAN2. The process of forwarding incoming packets, which are sent from LAN1 and which are destined to LAN2 (or vice versa), according to data structures that are called routing tables, is called *routing*. I discuss this process and the routing table data structures in this chapter and in the next as well.

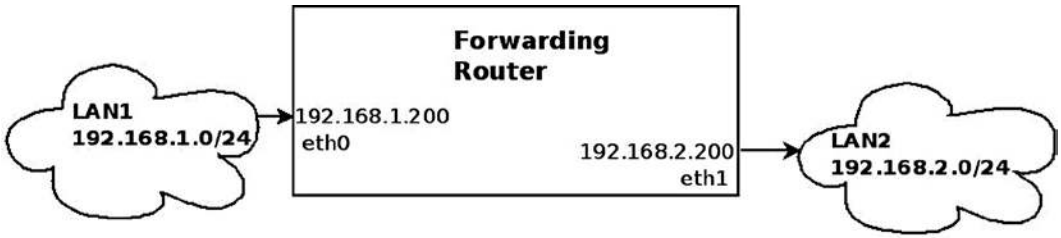

Figure 5-1. *Forwarding packets between two LANs*

In Figure 5-1, packets that arrive on `eth0` from LAN1, which are destined to LAN2, are forwarded via `eth1` as the outgoing device. In this process, the incoming packets move from Layer 2 (the link layer) in the kernel networking stack, to Layer 3, the network layer, in the forwarding router machine. As opposed to the case where the traffic is destined to the forwarding router machine ("Traffic to me"), however, there is no need to move the packets to Layer 4 (the transport layer) because this traffic in not intended to be handled by any Layer 4 transport socket. This traffic should be forwarded. Moving to Layer 4 has a performance cost, which is better to avoid whenever possible. This traffic is handled in Layer 3, and, according to the routing tables configured on the forwarding router machine, packets are forwarded on `eth1` as the outgoing interface (or rejected).

Figure 5-2 shows the three network layers handled by the kernel that were mentioned earlier.

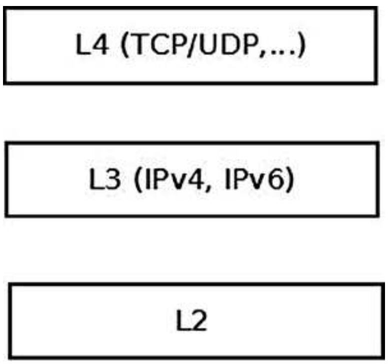

Figure 5-2. *The three layers that are handled by the networking kernel stack*

Two additional terms that I should mention here, which are commonly used in routing, are *default gateway* and *default route*. When you are defining a default gateway entry in a routing table, every packet that is not handled by the other routing entries (if there are such entries) must be forwarded to it, regardless of the destination address in the IP header of this packet. The default route is designated as 0.0.0.0/0 in Classless Inter-Domain Routing (CIDR) notation. As a simple example, you can add a machine with an IPv4 address of 192.168.2.1 as a default gateway as follows:

```
ip route add default via 192.168.2.1
```

Or, when using the `route` command, like this:

```
route add default gateway 192.168.2.1
```

In this section you learned what forwarding is and saw a simple example illustrating how packets are forwarded between two LANs. You also learned what a default gateway is and what a default route is, and how to add them. Now that you know the basic terminology and what forwarding is, let's move on and see how a lookup in the routing subsystem is performed.

Performing a Lookup in the Routing Subsystem

A lookup in the routing subsystem is done for each packet, both in the Rx path and in the Tx path. In kernels prior to 3.6, each lookup, both in the Rx path and in the Tx path, consisted of two phases: a lookup in the routing cache and, in case of a cache miss, a lookup in the routing tables (I discuss the routing cache at the end of this chapter, in the "IPv4 Routing Cache" section). A lookup is done by the `fib_lookup()` method. When the `fib_lookup()` method finds a proper entry in the routing subsystem, it builds a `fib_result` object, which consists of various routing parameters, and it returns 0. I discuss the `fib_result` object in this section and in other sections of this chapter. Here is the `fib_lookup()` prototype:

```
int fib_lookup(struct net *net, const struct flowi4 *flp, struct fib_result *res)
```

The `flowi4` object consists of fields that are important to the IPv4 routing lookup process, including the destination address, source address, Type of Service (TOS), and more. In fact the `flowi4` object defines the key to the lookup in the routing tables and should be initialized prior to performing a lookup with the `fib_lookup()` method. For IPv6 there is a parallel object named `flowi6`; both are defined in `include/net/flow.h`. The `fib_result` object is built in the IPv4 lookup process. The `fib_lookup()` method first searches the local FIB table. If the lookup fails, it performs a lookup in the main FIB table (I describe these two tables in the next section, "FIB tables"). After a lookup is successfully done, either in the Rx path or the Tx path, a `dst` object is built (an instance of the `dst_entry` structure, the destination cache, defined in `include/net/dst.h`). The `dst` object is embedded in a structure called `rtable`, as you will soon see. The `rtable` object, in fact, represents a routing entry which can be associated with an SKB. The most important members of the `dst_entry` object are two callbacks named `input` and `output`. In the routing lookup process, these callbacks are assigned to be the proper handlers according to the routing lookup result. These two callbacks get only an SKB as a parameter:

```
struct dst_entry {
    ...
    int  (*input)(struct sk_buff *);
    int  (*output)(struct sk_buff *);
    ...
}
```

The following is the `rtable` structure; as you can see, the `dst` object is the first object in this structure:

```
struct rtable {
    struct dst_entry  dst;

    int               rt_genid;
    unsigned int      rt_flags;
    __u16             rt_type;
    __u8              rt_is_input;
    __u8              rt_uses_gateway;

    int               rt_iif;

    /* Info on neighbour */
    __be32            rt_gateway;
```

```
    /* Miscellaneous cached information */
    u32             rt_pmtu;

    struct list_head  rt_uncached;
};
```

(include/net/route.h)

The following is a description of the members of the rtable structure:

- rt_flags: The rtable object flags; some of the important flags are mentioned here:
    - RTCF_BROADCAST: When set, the destination address is a broadcast address. This flag is set in the __mkroute_output() method and in the ip_route_input_slow() method.
    - RTCF_MULTICAST: When set, the destination address is a multicast address. This flag is set in the ip_route_input_mc() method and in the __mkroute_output() method.
    - RTCF_DOREDIRECT: When set, an ICMPv4 Redirect message should be sent as a response for an incoming packet. Several conditions should be fulfilled for this flag to be set, including that the input device and the output device are the same and the corresponding procfs send_redirects entry is set. There are more conditions, as you will see later in this chapter. This flag is set in the __mkroute_input() method.
    - RTCF_LOCAL: When set, the destination address is local. This flag is set in the following methods: ip_route_input_slow(), __mkroute_output(), ip_route_input_mc() and __ip_route_output_key(). Some of the RTCF_XXX flags can be set simultaneously. For example, RTCF_LOCAL can be set when RTCF_BROADCAST or RTCF_MULTICAST are set. For the complete list of RTCF_ XXX flags, look in include/uapi/linux/in_route.h. Note that some of them are unused.
- rt_is_input: A flag that is set to 1 when this is an input route.
- rt_uses_gateway: Gets a value according to the following:
    - When the nexthop is a gateway, rt_uses_gateway is 1.
    - When the nexthop is a direct route, rt_uses_gateway is 0.
- rt_iif: The ifindex of the incoming interface. (Note that the rt_oif member was removed from the rtable structure in kernel 3.6; it was set to the oif of the specified flow key, but was used in fact only in one method).
- rt_pmtu: The Path MTU (the smallest MTU along the route).

  Note that in kernel 3.6, the fib_compute_spec_dst() method was added, which gets an SKB as a parameter. This method made the rt_spec_dst member of the rtable structure unneeded, and rt_spec_dst was removed from the rtable structure as a result. The fib_compute_spec_dst() method is needed in special cases, such as in the icmp_reply() method, when replying to the sender using its source address as a destination for the reply.

For incoming unicast packets destined to the local host, the input callback of the dst object is set to ip_local_deliver(), and for incoming unicast packets that should be forwarded, this input callback is set to ip_forward(). For a packet generated on the local machine and sent away, the output callback is set to be ip_output(). For a multicast packet, the input callback can be set to ip_mr_input() (under some conditions which are not detailed in this chapter). There are cases when the `input` callback is set to be `ip_error()`, as you will see later in the PROHIBIT rule example in this chapter. Let's take a look in the `fib_result` object:

```
struct fib_result {
        unsigned char    prefixlen;
        unsigned char    nh_sel;
        unsigned char    type;
        unsigned char    scope;
        u32              tclassid;
        struct fib_info  *fi;
        struct fib_table *table;
        struct list_head *fa_head;
};
```

(`include/net/ip_fib.h`)

- `prefixlen`: The prefix length, which represents the netmask. Its values are in the range 0 to 32. It is 0 when using the default route. When adding, for example, a routing entry by `ip route add 192.168.2.0/24 dev eth0`, the `prefixlen` is 24, according to the netmask which was specified when adding the entry. The `prefixlen` is set in the `check_leaf()` method (`net/ipv4/fib_trie.c`).
- `nh_sel`: The nexthop number. When working with one nexthop only, it is 0. When working with Multipath Routing, there can be more than one nexthop. The nexthop objects are stored in an array in the routing entry (inside the `fib_info` object), as discussed in the next section.
- `type`: The `type` of the `fib_result` object is the most important field because it determines in fact how to handle the packet: whether to forward it to a different machine, deliver it locally, discard it silently, discard it with replying with an ICMPv4 message, and so on. The type of the `fib_result` object is determined according to the packet content (most notably the destination address) and according to routing rules set by the administrator, routing daemons, or a Redirect message. You will see how the `type` of the `fib_result` object is determined in the lookup process later in this chapter and in the next. The two most common types of the `fib_result` objects are the RTN_UNICAST type, which is set when the packet is for forwarding via a gateway or a direct route, and the RTN_LOCAL type, which is set when the packet is for the local host. Other types you will encounter in this book are the RTN_BROADCAST type, for packets that should be accepted locally as broadcasts, the RTN_MULTICAST type, for multicast routes, the RTN_UNREACHABLE type, for packets which trigger sending back an ICMPv4 "Destination Unreachable" message, and more. There are 12 route types in all. For a complete list of all available route types, see `include/uapi/linux/rtnetlink.h`.
- `fi`: A pointer to a `fib_info` object, which represents a routing entry. The `fib_info` object holds a reference to the nexthop (`fib_nh`). I discuss the FIB info structure in the section "FIB Info" later in this chapter.
- `table`: A pointer to the FIB table on which the lookup is done. It is set in the `check_leaf()` method (`net/ipv4/fib_trie.c`).
- `fa_head`: A pointer to a `fib_alias` list (a list of `fib_alias` objects associated with this route); optimization of routing entries is done when using `fib_alias` objects, which avoids creating a separate `fib_info` object for each routing entry, regardless of the fact that there are other `fib_info` objects which are very similar. All FIB aliases are sorted by `fa_tos` descending and `fib_priority` (metric) ascending. Aliases whose `fa_tos` is 0 are the last and can match any TOS. I discuss the `fib_alias` structure in the section "FIB Alias" later in this chapter.

In this section you learned how a lookup in the routing subsystem is performed. You also found out about important data structures that relate to the routing lookup process, like `fib_result` and `rtable`. The next section discusses how the FIB tables are organized.

FIB Tables

The main data structure of the routing subsystem is the routing table, which is represented by the `fib_table` structure. A routing table can be described, in a somewhat simplified way, as a table of entries where each entry determines which nexthop should be chosen for traffic destined to a subnet (or to a specific IPv4 destination address). This entry has other parameters, of course, discussed later in this chapter. Each routing entry contains a `fib_info` object (`include/net/ip_fib.h`), which stores the most important routing entry parameters (but not all, as you will see later in this chapter). The `fib_info` object is created by the `fib_create_info()` method (`net/ipv4/fib_semantics.c`) and is stored in a hash table named `fib_info_hash`. When the route uses `prefsrc`, the `fib_info` object is added also to a hash table named `fib_info_laddrhash`.

There is a global counter of `fib_info` objects named `fib_info_cnt` which is incremented when creating a `fib_info` object, by the `fib_create_info()` method, and decremented when freeing a `fib_info` object, by the `free_fib_info()` method. The hash table is dynamically resized when it grows over some threshold. A lookup in the `fib_info_hash` hash table is done by the `fib_find_info()` method (it returns NULL when not finding an entry). Serializing access to the `fib_info` members is done by a spinlock named `fib_info_lock`. Here's the `fib_table` structure:

```
struct fib_table {
        struct hlist_node       tb_hlist;
        u32                     tb_id;
        int                     tb_default;
        int                     tb_num_default;
        unsigned long           tb_data[0];
};
```

(`include/net/ip_fib.h`)

- `tb_id`: The table identifier. For the main table, `tb_id` is 254 (RT_TABLE_MAIN), and for the local table, `tb_id` is 255 (RT_TABLE_LOCAL). I talk about the main table and the local table soon—for now, just note that when working without Policy Routing, only these two FIB tables, the main table and the local table, are created in boot.
- `tb_num_default:` The number of the default routes in the table. The `fib_trie_table()` method, which creates a table, initializes `tb_num_default` to 0. Adding a default route increments `tb_num_default` by 1, by the `fib_table_insert()` method. Deleting a default route decrements `tb_num_default` by 1, by the `fib_table_delete()` method.
- `tb_data[0]` : A placeholder for a routing entry (`trie`) object.

This section covered how a FIB table is implemented. Next you will learn about the FIB info, which represents a single routing entry.

FIB Info

A routing entry is represented by a `fib_info` structure. It consists of important routing entry parameters, such as the outgoing network device (`fib_dev`), the priority (`fib_priority`), the routing protocol identifier of this route (`fib_protocol`), and more. Let's take a look at the `fib_info` structure:

```
struct fib_info {
    struct hlist_node    fib_hash;
    struct hlist_node    fib_lhash;
    struct net          *fib_net;
    int                 fib_treeref;
    atomic_t            fib_clntref;
    unsigned int        fib_flags;
    unsigned char       fib_dead;
    unsigned char       fib_protocol;
    unsigned char       fib_scope;
    unsigned char       fib_type;
    __be32              fib_prefsrc;
    u32                 fib_priority;
    u32                 *fib_metrics;
#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]
    int                 fib_nhs;
#ifdef CONFIG_IP_ROUTE_MULTIPATH
    int                 fib_power;
#endif
    struct rcu_head     rcu;
    struct fib_nh       fib_nh[0];
#define fib_dev         fib_nh[0].nh_dev
};
```

(`include/net/ip_fib.h`)

- `fib_net`: The network namespace the `fib_info` object belongs to.
- `fib_treeref`: A reference counter that represents the number of `fib_alias` objects which hold a reference to this `fib_info` object. This reference counter is incremented in the `fib_create_info()` method and decremented in the `fib_release_info()` method. Both methods are in `net/ipv4/fib_semantics.c`.
- `fib_clntref`: A reference counter that is incremented by the `fib_create_info()` method (`net/ipv4/fib_semantics.c`) and decremented by the `fib_info_put()` method (`include/net/ip_fib.h`). If, after decrementing it by 1 in the `fib_info_put()` method, it reaches zero, than the associated `fib_info` object is freed by the `free_fib_info()` method.
- `fib_dead`: A flag that indicates whether it is permitted to free the `fib_info` object with the `free_fib_info()` method; `fib_dead` must be set to 1 before calling the `free_fib_info()` method. If the `fib_dead` flag is not set (its value is 0), then it is considered alive, and trying to free it with the `free_fib_info()` method will fail.

- `fib_protocol`: The routing protocol identifier of this route. When adding a routing rule from userspace without specifying the routing protocol ID, the `fib_protocol` is assigned to be RTPROT_BOOT. The administrator may add a route with the "proto static" modifier, which indicates that the route was added by an administrator; this can be done, for example, like this: `ip route add proto static 192.168.5.3 via 192.168.2.1`. The `fib_protocol` can be assigned one of these flags:

  - RTPROT_UNSPEC: An error value.
  - RTPROT_REDIRECT: When set, the routing entry was created as a result of receiving an ICMP Redirect message. The RTPROT_REDIRECT protocol identifier is used only in IPv6.
  - RTPROT_KERNEL: When set, the routing entry was created by the kernel (for example, when creating the local IPv4 routing table, explained shortly).
  - RTPROT_BOOT: When set, the admin added a route without specifying the "proto static" modifier.
  - RTPROT_STATIC: Route installed by system administrator.
  - RTPROT_RA: Don't misread this— this protocol identifier is not for Router Alert; it is for RDISC/ND Router Advertisements, and it is used in the kernel by the IPv6 subsystem only; see: `net/ipv6/route.c`. I discuss it in Chapter 8.

  The routing entry could also be added by userspace routing daemons, like ZEBRA, XORP, MROUTED, and more. Then it will be assigned the corresponding value from a list of protocol identifiers (see the RTPROT_XXX definitions in `include/uapi/linux/rtnetlink.h`). For example, for the XORP daemon it will be RTPROT_XORP. Note that these flags (like RTPROT_KERNEL or RTPROT_STATIC) are also used by IPv6, for the parallel field (the `rt6i_protocol` field in the `rt6_info` structure; the `rt6_info` object is the IPv6 parallel to the `rtable` object).

- `fib_scope`: The scope of the destination address. In short, scopes are assigned to addresses and routes. Scope indicates the distance of the host from other nodes. The `ip address show` command shows the scopes of all configured IP addresses on a host. The `ip route show` command displays the scopes of all the route entries of the main table. A scope can be one of these:

  - host (RT_SCOPE_HOST): The node cannot communicate with the other network nodes. The loopback address has scope host.
  - global (RT_SCOPE_UNIVERSE): The address can be used anywhere. This is the most common case.
  - link (RT_SCOPE_LINK): This address can be accessed only from directly attached hosts.
  - site (RT_SCOPE_SITE): This is used in IPv6 only (I discuss it in Chapter 8).
  - nowhere (RT_SCOPE_NOWHERE): Destination doesn't exist.

  When a route is added by an administrator without specifying a scope, the `fib_scope` field is assigned a value according to these rules:

  - global scope (RT_SCOPE_UNIVERSE): For all gatewayed unicast routes.
  - scope link (RT_SCOPE_LINK): For direct unicast and broadcast routes.
  - scope host (RT_SCOPE_HOST): For local routes.

- `fib_type`: The type of the route. The `fib_type` field was added to the `fib_info` structure as a key to make sure there is differentiation among `fib_info` objects by their type. The `fib_type` field was added to the `fib_info struct` in kernel 3.7. Originally this type was stored only in the `fa_type` field of the FIB alias object (`fib_alias`). You can add a rule to block traffic according to a specified category, for example, by: `ip route add prohibit 192.168.1.17 from 192.168.2.103`.
  - The `fib_type` of the generated `fib_info` object is RTN_PROHIBIT.
  - Sending traffic from 192.168.2.103 to 192.168.1.17 results in an ICMPv4 message of "Packet Filtered" (ICMP_PKT_FILTERED).
- `fib_prefsrc`: There are cases when you want to provide a specific source address to the lookup key. This is done by setting `fib_prefsrc.`
- `fib_priority`: The priority of the route, by default, is 0, which is the highest priority. The higher the value of the priority, the lower the priority is. For example, a priority of 3 is lower than a priority of 0, which is the highest priority. You can configure it, for example, with the `ip` command, in one of the following ways:
  - `ip route add 192.168.1.10 via 192.168.2.1 metric 5`
  - `ip route add 192.168.1.10 via 192.168.2.1 priority 5`
  - `ip route add 192.168.1.10 via 192.168.2.1 preference 5`

  Each of these three commands sets the `fib_priority` to 5; there is no difference at all between them. Moreover, the `metric` parameter of the `ip route` command is not related in any way to the `fib_metrics` field of the `fib_info` structure.
- `fib_mtu`, `fib_window`, `fib_rtt`, and `fib_advmss` simply give more convenient names to commonly used elements of the `fib_metrics` array.

  `fib_metrics` is an array of 15 (RTAX_MAX) elements consisting of various metrics. It is initialized to be `dst_default_metrics` in `net/core/dst.c`. Many metrics are related to the TCP protocol, such as the Initial Congestion Window (`initcwnd`) metric. Table 5-1, at the end of the chapter shows all the available metrics and displays whether each is a TCP-related metric or not.

  From userspace, the TCPv4 `initcwnd` metric can be set thus, for example:

  ```
  ip route add 192.168.1.0/24 initcwnd 35
  ```

  There are metrics which are not TCP specific—for example, the `mtu` metric, which can be set from userspace like this:

  ```
  ip route add 192.168.1.0/24 mtu 800
  ```

  or like this:

  ```
  ip route add 192.168.1.0/24 mtu lock 800
  ```

The difference between the two commands is that when specifying the modifier `lock`, no path MTU discovery will be tried. When not specifying the modifier `lock`, the MTU may be updated by the kernel due to Path MTU discovery. For more about how this is implemented, see the `__ip_rt_update_pmtu()` method, in `net/ipv4/route.c`:

```
static void __ip_rt_update_pmtu(struct rtable *rt, struct flowi4 *fl4, u32 mtu)
{
```

Avoiding Path MTU update when specifying the `mtu lock` modifier is achieved by calling the `dst_metric_locked()` method :

```
. . .
if (dst_metric_locked(dst, RTAX_MTU))
      return;
. . .
}
```

- `fib_nhs`: The number of nexthops. When Multipath Routing (CONFIG_IP_ROUTE_MULTIPATH) is not set, it cannot be more than 1. The Multipath Routing feature sets multiple alternative paths for a route, possibly assigning different weights to these paths. This feature provides benefits such as fault tolerance, increased bandwidth, or improved security (I discuss it in Chapter 6).
- `fib_dev`: The network device that will transmit the packet to the nexthop.
- `fib_nh[0]`: The `fib_nh[0]` member represents the nexthop. When working with Multipath Routing, you can define more than one nexthop in a route, and in this case there is an array of nexthops. Defining two nexthop nodes can be done like this, for example: `ip route add default scope global nexthop dev eth0 nexthop dev eth1`.

As mentioned, when the `fib_type` is RTN_PROHIBIT, an ICMPv4 message of "Packet Filtered" (ICMP_PKT_FILTERED) is sent. How is it implemented? An array named `fib_props` consists of 12 (RTN_MAX) elements (defined in `net/ipv4/fib_semantics.c`). The index of this array is the route type. The available route types, such as RTN_PROHIBIT or RTN_UNICAST, can be found in `include/uapi/linux/rtnetlink.h`. Each element in the array is an instance of `struct fib_prop`; the `fib_prop` structure is a very simple structure:

```
struct fib_prop {
        int     error;
        u8      scope;
  };
```

(`net/ipv4/fib_lookup.h`)

For every route type, the corresponding `fib_prop` object contains the `error` and the `scope` for that route. For example, for the RTN_UNICAST route type (gateway or direct route), which is a very common route, the error value is 0, which means that there is no error, and the scope is RT_SCOPE_UNIVERSE. For the RTN_PROHIBIT route type (a rule which a system administrator configures in order to block traffic), the error is –EACCES, and the scope is RT_SCOPE_UNIVERSE:

```
const struct fib_prop fib_props[RTN_MAX + 1] = {
 . . .
        [RTN_PROHIBIT] = {
                .error  = -EACCES,
                .scope  = RT_SCOPE_UNIVERSE,
        },

. . .
```

Table 5-2 at the end of this chapter shows all available route types, their error codes, and their scopes.

When you configure a rule like the one mentioned earlier, by `ip route add prohibit 192.168.1.17 from 192.168.2.103`—and when a packet is sent from 192.168.2.103 to 192.168.1.17, what happens is the following: a lookup in the routing tables is performed in the Rx path. When a corresponding entry, which is in fact a leaf in the FIB TRIE, is found, the `check_leaf()` method is invoked. This method accesses the `fib_props` array with the route type of the packet as an index (`fa->fa_type`):

```
static int check_leaf(struct fib_table *tb, struct trie *t, struct leaf *l,
                      t_key key,  const struct flowi4 *flp,
                      struct fib_result *res, int fib_flags)
{
    . . .
       fib_alias_accessed(fa);
       err = fib_props[fa->fa_type].error;
       if (err) {
               . . .
               return err;
                }
    . . .
```

Eventually, the `fib_lookup()` method, which initiated the lookup in the IPv4 routing subsystem, returns an error of –EACCES (in our case). It propagates all the way back from `check_leaf()` via `fib_table_lookup()` and so on until it returns to the method which triggered this chain, namely the `fib_lookup()` method. When the `fib_lookup()` method returns an error in the Rx path, it is handled by the `ip_error()` method. According to the error, an action is taken. In the case of –EACCES, an ICMPv4 of destination unreachable with code of Packet Filtered (ICMP_PKT_FILTERED) is sent back, and the packet is dropped.

This section covered the FIB info, which represents a single routing entry. The next section discusses caching in the IPv4 routing subsystem (not to be confused with the IPv4 routing cache, which was removed from the network stack, and is discussed in the “IPv4 Routing Cache” section at the end of this chapter).

Caching

Caching the results of a routing lookup is an optimization technique that improves the performance of the routing subsystem. The results of a routing lookup are usually cached in the nexthop (`fib_nh`) object; when the packet is not a unicast packet or `realms` are used (the packet `itag` is not 0), the results are not cached in the nexthop. The reason is that if all types of packets are cached, then the same nexthop can be used by different kinds of routes—that should be avoided. There are some minor exceptions to this which I do not discuss in this chapter. Caching in the Rx and the Tx path are performed as follows:

- In the Rx path, caching the `fib_result` object in the nexthop (`fib_nh`) object is done by setting the `nh_rth_input` field of the nexthop (`fib_nh`) object.
- In the Tx path, caching the `fib_result` object in the nexthop (`fib_nh`) object is done by setting the `nh_pcpu_rth_output` field of the nexthop (`fib_nh`) object.
- Both `nh_rth_input` and `nh_pcpu_rth_output` are instances of the `rtable` structure.
- Caching the `fib_result` is done by the `rt_cache_route()` method both in the Rx and the Tx paths (`net/ipv4/route.c`).
- Caching of Path MTU and ICMPv4 redirects is done with FIB exceptions.

For performance, the nh_pcpu_rth_output is a per-CPU variable, meaning there is a copy for each CPU of the output dst entry. Caching is used almost always. The few exceptions are when an ICMPv4 Redirect message is sent, or itag (tclassid) is set, or there is not enough memory.

In this section you have learned how caching is done using the nexthop object. The next section discusses the fib_nh structure, which represents the nexthop, and the FIB nexthop exceptions.

Nexthop (fib_nh)

The fib_nh structure represents the nexthop. It consists of information such as the outgoing nexthop network device (nh_dev), outgoing nexthop interface index (nh_oif), the scope (nh_scope), and more. Let's take a look:

```
struct fib_nh {
    struct net_device       *nh_dev;
    struct hlist_node       nh_hash;
    struct fib_info         *nh_parent;
    unsigned int            nh_flags;
    unsigned char           nh_scope;
#ifdef CONFIG_IP_ROUTE_MULTIPATH
    int                     nh_weight;
    int                     nh_power;
#endif
#ifdef CONFIG_IP_ROUTE_CLASSID
    __u32                   nh_tclassid;
#endif
    int                     nh_oif;
    __be32                  nh_gw;
    __be32                  nh_saddr;
    int                     nh_saddr_genid;
    struct rtable __rcu * __percpu *nh_pcpu_rth_output;
    struct rtable __rcu     *nh_rth_input;
    struct fnhe_hash_bucket *nh_exceptions;
};
```

(include/net/ip_fib.h)

The nh_dev field represents the network device (net_device object) on which traffic to the nexthop will be transmitted. When a network device associated with one or more routes is disabled, a NETDEV_DOWN notification is sent. The FIB callback for handling this event is the fib_netdev_event() method; it is the callback of the fib_netdev_notifier notifier object, which is registered in the ip_fib_init() method by calling the register_netdevice_notifier() method (notification chains are discussed in Chapter 14). The fib_netdev_event() method calls the fib_disable_ip() method upon receiving a NETDEV_DOWN notification. In the fib_disable_ip() method, the following steps are performed:

- First, the fib_sync_down_dev() method is called (net/ipv4/fib_semantics.c). In the fib_sync_down_dev() method, the RTNH_F_DEAD flag of the nexthop flags (nh_flags) is set and the FIB info flags (fib_flags) is set.
- The routes are flushed by the fib_flush() method.
- The rt_cache_flush() method and the arp_ifdown() method are invoked. The arp_ifdown() method is not on any notifier chain.

FIB Nexthop Exceptions

FIB nexthop exceptions were added in kernel 3.6 to handle cases when a routing entry is changed not as a result of a userspace action, but as a result of an ICMPv4 Redirect message or as a result of Path MTU discovery. The hash key is the destination address. The FIB nexthop exceptions are based on a 2048 entry hash table; reclaiming (freeing hash entries) starts at a chain depth of 5. Each nexthop object (`fib_nh`) has a FIB nexthop exceptions hash table, `nh_exceptions` (an instance of the `fnhe_hash_bucket` structure). Let's take a look at the `fib_nh_exception` structure:

```
struct fib_nh_exception {
    struct fib_nh_exception __rcu    *fnhe_next;
    __be32                           fnhe_daddr;
    u32                              fnhe_pmtu;
    __be32                           fnhe_gw;
    unsigned long                    fnhe_expires;
    struct rtable __rcu              *fnhe_rth;
    unsigned long                    fnhe_stamp;
};
```

(`include/net/ip_fib.h`)

The `fib_nh_exception` objects are created by the `update_or_create_fnhe()` method (`net/ipv4/route.c`). Where are FIB nexthop exceptions generated? The first case is when receiving an ICMPv4 Redirect message ("Redirect to Host") in the `__ip_do_redirect()` method. The "Redirect to Host" message includes a new gateway. The `fnhe_gw` field of the `fib_nh_exception` is set to be the new gateway when creating the FIB nexthop exception object (in the `update_or_create_fnhe()` method):

```
static void __ip_do_redirect(struct rtable *rt, struct sk_buff *skb, struct flowi4 *fl4,
                 bool kill_route)
{
  ...
  __be32 new_gw = icmp_hdr(skb)->un.gateway;
  ...
  update_or_create_fnhe(nh, fl4->daddr, new_gw, 0, 0);
  ...

}
```

The second case of generating FIB nexthop exceptions is when the Path MTU has changed, in the `__ip_rt_update_pmtu()` method. In such a case, the `fnhe_pmtu` field of the `fib_nh_exception` object is set to be the new MTU when creating the FIB nexthop exception object (in the `update_or_create_fnhe()` method). PMTU value is expired if it was not updated in the last 10 minutes (`ip_rt_mtu_expires`). This period is checked on every `dst_mtu()` call via the `ipv4_mtu()` method, which is a `dst->ops->mtu` handler. The `ip_rt_mtu_expires`, which is by default 600 seconds, can be configured via the `procfs` entry `/proc/sys/net/ipv4/route/mtu_expires`:

```
static void __ip_rt_update_pmtu(struct rtable *rt, struct flowi4 *fl4, u32 mtu)
{
    . . .
    if (fib_lookup(dev_net(dst->dev), fl4, &res) == 0) {
        struct fib_nh *nh = &FIB_RES_NH(res);

        update_or_create_fnhe(nh, fl4->daddr, 0, mtu,
                      jiffies + ip_rt_mtu_expires);
    }
    . . .
}
```

■ **Note** FIB nexthop exceptions are used in the Tx path. Starting with Linux 3.11, they are also used in the Rx path. As a result, instead of `fnhe_rth`, there are `fnhe_rth_input` and `fnhe_rth_output`.

Since kernel 2.4, Policy Routing is supported. With Policy Routing, the routing of a packet depends not only on the destination address, but on several other factors, such as the source address or the TOS. The system administrator can add up to 255 routing tables.

Policy Routing

When working without Policy Routing (CONFIG_IP_MULTIPLE_TABLES is not set), two routing tables are created: the local table and the main table. The main table id is 254 (RT_TABLE_MAIN), and the local table id is 255 (RT_TABLE_LOCAL). The local table contains routing entries of local addresses. These routing entries can be added to the local table only by the kernel. Adding routing entries to the main table (RT_TABLE_MAIN) is done by a system administrator (via `ip route add`, for example). These tables are created by the `fib4_rules_init()` method of `net/ipv4/fib_frontend.c`. These tables were called `ip_fib_local_table` and `ip_fib_main_table` in kernels prior to 2.6.25, but they were removed in favor of using unified access to the routing tables with the `fib_get_table()` method with appropriate argument. By *unified access*, I mean that access to the routing tables is done in the same way, with the `fib_get_table()` method, both when Policy Routing support is enabled and when it is disabled. The `fib_get_table()` method gets only two arguments: the network namespace and the table id. Note that there is a different method with the same name, `fib4_rules_init()`, for the Policy Routing case, in `net/ipv4/fib_rules.c`, which is invoked when working with Policy Routing support. When working with Policy Routing support (CONFIG_IP_MULTIPLE_TABLES is set), there are three initial tables (local, main, and default), and there can be up to 255 routing tables. I talk more about Policy Routing in Chapter 6. Access to the main routing table can be done as follows:

- By a system administrator command (using `ip route` or `route`):
  - Adding a route by `ip route add` is implemented by sending RTM_NEWROUTE message from userspace, which is handled by the `inet_rtm_newroute()` method. Note that a route is not necessarily always a rule that permits traffic. You can also add a route that blocks traffic, for example, by `ip route add prohibit 192.168.1.17 from 192.168.2.103.` As a result of applying this rule, all packets sent from 192.168.2.103 to 192.168.1.17 will be blocked.
  - Deleting a route by `ip route del` is implemented by sending RTM_DELROUTE message from userspace, which is handled by the `inet_rtm_delroute()` method.
  - Dumping a routing table by `ip route show` is implemented by sending RTM_GETROUTE message from userspace, which is handled by the `inet_dump_fib()` method.

  Note that `ip route show` displays the main table. For displaying the local table, you should run `ip route show table local`.

  - Adding a route by `route add` is implemented by sending SIOCADDRT IOCTL, which is handled by the `ip_rt_ioctl()` method (`net/ipv4/fib_frontend.c`).
  - Deleting a route by `route del` is implemented by sending SIOCDELRT IOCTL, which is handled by the `ip_rt_ioctl()` method (`net/ipv4/fib_frontend.c`).

- By userspace routing daemons which implement routing protocols like BGP (Border Gateway Protocol), EGP (Exterior Gateway Protocol), OSPF (Open Shortest Path First), or others. These routing daemons run on core routers, which operate in the Internet backbone, and can handle hundreds of thousands of routes.

I should mention here that routes that were changed as a result of an ICMPv4 REDIRECT message or as a result of Path MTU discovery are cached in the nexthop exception table, discussed shortly. The next section describes the FIB alias, which helps in routing optimizations.

FIB Alias (fib_alias)

There are cases when several routing entries to the same destination address or to the same subnet are created. These routing entries differ only in the value of their TOS. Instead of creating a `fib_info` for each such route, a `fib_alias` object is created. A `fib_alias` is smaller, which reduces memory consumption. Here is a simple example of creating 3 `fib_alias` objects:

```
ip route add 192.168.1.10 via 192.168.2.1 tos 0x2
ip route add 192.168.1.10 via 192.168.2.1 tos 0x4
ip route add 192.168.1.10 via 192.168.2.1 tos 0x6
```

Let's take a look at the `fib_alias` structure definition:

```
struct fib_alias {
        struct list_head        fa_list;
        struct fib_info         *fa_info;
        u8                      fa_tos;
        u8                      fa_type;
        u8                      fa_state;
        struct rcu_head         rcu;
};
```

(`net/ipv4/fib_lookup.h`)

Note that there was also a scope field in the `fib_alias` structure (`fa_scope`), but it was moved in kernel 2.6.39 to the `fib_info` structure.

The `fib_alias` object stores routes to the same subnet but with different parameters. You can have one `fib_info` object which will be shared by many `fib_alias` objects. The `fa_info` pointer in all these `fib_alias` objects, in this case, will point to the same shared `fib_info` object. In Figure 5-3, you can see one `fib_info` object which is shared by three `fib_alias` objects, each with a different `fa_tos`. Note that the reference counter value of the `fib_info` object is 3 (`fib_treeref`).

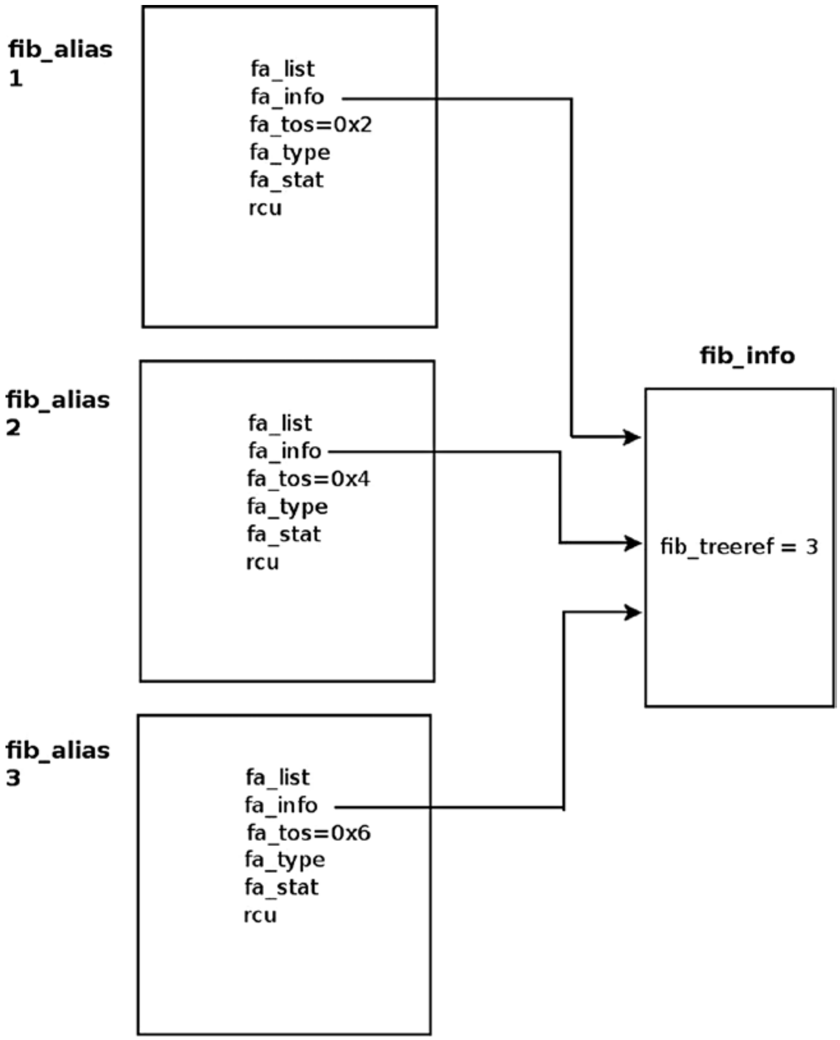

Figure 5-3. *A fib_info which is shared by three fib_alias objects. Each fib_alias object has a different fa_tos value*

Let's take a look at what happens when you try to add a key for which a `fib_node` was already added before (as in the earlier example with the three TOS values 0x2, 0x4, and 0x6); suppose you had created the first rule with TOS of 0x2, and now you create the second rule, with TOS of 0x4.

A `fib_alias` object is created by the `fib_table_insert()` method, which is the method that handles adding a routing entry:

```
int fib_table_insert(struct fib_table *tb, struct fib_config *cfg)
 {
         struct trie *t = (struct trie *) tb->tb_data;
         struct fib_alias *fa, *new_fa;
         struct list_head *fa_head = NULL;
         struct fib_info *fi;
      . . .
```

First, a `fib_info` object is created. Note that in the `fib_create_info()` method, after allocating and creating a `fib_info` object, a lookup is performed to check whether a similar object already exists by calling the `fib_find_info()` method. If such an object exists, it will be freed, and the reference counter of the object that was found (`ofi` in the code snippet you will shortly see) will be incremented by 1:

```
fi = fib_create_info(cfg);
```

Let's take a look at the code snippet in the `fib_create_info()` method mentioned earlier; for creating the second TOS rule, the `fib_info` object of the first rule and the `fib_info` object of the second rule are identical. You should remember that the TOS field exists in the `fib_alias` object but not in the `fib_info` object:

```
struct fib_info *fib_create_info(struct fib_config *cfg)
{
    struct fib_info *fi = NULL;
    struct fib_info *ofi;
    . . .
    fi = kzalloc(sizeof(*fi)+nhs*sizeof(struct fib_nh), GFP_KERNEL);
    if (fi == NULL)
            goto failure;
    . . .
link_it:
        ofi = fib_find_info(fi);
```

If a similar object is found, free the `fib_info` object and increment the `fib_treeref` reference count:

```
        if (ofi) {
                fi->fib_dead = 1;
                free_fib_info(fi);
                ofi->fib_treeref++;
                return ofi;
        }
    . . .
}
```

Now a check is performed to find out whether there is an alias to the `fib_info` object; in this case, there will be no alias because the TOS of the second rule is different than the TOS of the first rule:

```
     l = fib_find_node(t, key);
     fa = NULL;

     if (l) {
             fa_head = get_fa_head(l, plen);
             fa = fib_find_alias(fa_head, tos, fi->fib_priority);
     }

if (fa && fa->fa_tos == tos &&
    fa->fa_info->fib_priority == fi->fib_priority) {
    . . .
       }
```

Now a `fib_alias` is created, and its `fa_info` pointer is assigned to point the `fib_info` of the first rule that was created:

```
new_fa = kmem_cache_alloc(fn_alias_kmem, GFP_KERNEL);
if (new_fa == NULL)
    goto out;

new_fa->fa_info = fi;
    . . .
```

Now that I have covered the FIB Alias, you are ready to look at the ICMPv4 redirect message, which is sent when there is a suboptimal route.

ICMPv4 Redirect Message

There are cases when a routing entry is suboptimal. In such cases, an ICMPv4 redirect message is sent. The main criterion for a suboptimal entry is that the input device and the output device are the same. But there are more conditions that should be fulfilled so that an ICMPv4 redirect message is sent, as you will see in this section. There are four codes of ICMPv4 redirect message:

- ICMP_REDIR_NET: Redirect Net
- ICMP_REDIR_HOST: Redirect Host
- ICMP_REDIR_NETTOS: Redirect Net for TOS
- ICMP_REDIR_HOSTTOS: Redirect Host for TOS

Figure 5-4 shows a setup where there is a suboptimal route. There are three machines in this setup, all on the same subnet (192.168.2.0/24) and all connected via a gateway (192.168.2.1). The AMD server (192.168.2.200) added the Windows server (192.168.2.10) as a gateway for accessing 192.168.2.7 (the laptop) by `ip route add 192.168.2.7 via 192.168.2.10`. The AMD server sends traffic to the laptop, for example, by `ping 192.168.2.7`. Because the default gateway is `192.168.2.10`, the traffic is sent to `192.168.2.10`. The Windows server detects that this is a suboptimal route, because the AMD server could send directly to 192.168.2.7, and sends back to the AMD server an ICMPv4 redirect message with ICMP_REDIR_HOST code.

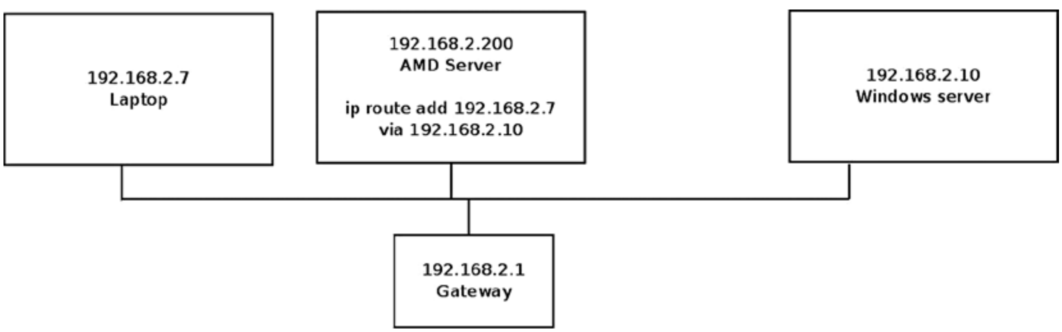

Figure 5-4. *Redirect to Host (ICMP_REDIR_HOST), a simple setup*

Now that you have a better understanding of redirects, let's look at how an ICMPv4 message is generated.

Generating an ICMPv4 Redirect Message

An ICMPv4 Redirect message is sent when there is some suboptimal route. The most notable condition for a suboptimal route is that the input device and the output device are the same, but there are some more conditions which should be met. Generating an ICMPv4 Redirect message is done in two phases:

- In the `__mkroute_input()` method: Here the RTCF_DOREDIRECT flag is set if needed.
- In the `ip_forward()` method: Here the ICMPv4 Redirect message is actually sent by calling the `ip_rt_send_redirect()` method.

```
static int __mkroute_input(struct sk_buff *skb,
                   const struct fib_result *res,
                   struct in_device *in_dev,
                   __be32 daddr, __be32 saddr, u32 tos)
{
    struct rtable *rth;
    int err;
    struct in_device *out_dev;
    unsigned int flags = 0;
    bool do_cache;
```

All of the following conditions should be sustained so that the RTCF_DOREDIRECT flag is set:

- The input device and the output device are the same.
- The `procfs` entry, `/proc/sys/net/ipv4/conf/<deviceName>/send_redirects`, is set.
- Either this outgoing device is a shared media or the source address (`saddr`) and the nexthop gateway address (`nh_gw`) are on the same subnet:

```
if (out_dev == in_dev && err && IN_DEV_TX_REDIRECTS(out_dev) &&
  (IN_DEV_SHARED_MEDIA(out_dev) ||
   inet_addr_onlink(out_dev, saddr, FIB_RES_GW(*res)))) {

  flags |= RTCF_DOREDIRECT;
  do_cache = false;
}
  . . .
```

Setting the `rtable` object flags is done by:

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

}
```

Sending the ICMPv4 Redirect message is done in the second phase, by the `ip_forward()` method:

```
int ip_forward(struct sk_buff *skb)
{
    struct iphdr          *iph;    /* Our header */
    struct rtable         *rt;     /* Route we use */
    struct ip_options     *opt     = &(IPCB(skb)->opt);
```

Next a check is performed to see whether the RTCF_DOREDIRECT flag is set, whether an IP option of strict route does not exist (see chapter 4), and whether it is not an IPsec packet. (With IPsec tunnels, the input device of the tunneled packet can be the same as the decapsulated packet outgoing device; see http://lists.openwall.net/netdev/2007/08/24/29):

```
if (rt->rt_flags&RTCF_DOREDIRECT && !opt->srr && !skb_sec_path(skb))
    ip_rt_send_redirect(skb);
```

In the `ip_rt_send_redirect()` method, the ICMPv4 Redirect message is actually sent. The third parameter is the IP address of the advised new gateway, which will be 192.168.2.7 in this case (The address of the laptop):

```
void ip_rt_send_redirect(struct sk_buff *skb)
  {
      . . .
      icmp_send(skb, ICMP_REDIRECT, ICMP_REDIR_HOST,
            rt_nexthop(rt, ip_hdr(skb)->daddr))
      . . .
  }
```

(net/ipv4/route.c)

Receiving an ICMPv4 Redirect Message

For an ICMPv4 Redirect message to be processed, it should pass some sanity checks. Handling an ICMPv4 Redirect message is done by the `__ip_do_redirect()` method:

```
static void __ip_do_redirect(struct rtable *rt, struct sk_buff *skb, struct flowi4
    *fl4,bool kill_route)
{
    __be32 new_gw = icmp_hdr(skb)->un.gateway;
    __be32 old_gw = ip_hdr(skb)->saddr;
    struct net_device *dev = skb->dev;
    struct in_device *in_dev;
    struct fib_result res;
    struct neighbour *n;
    struct net *net;
      . . .
```

Various checks are performed, such as that the network device is set to accept redirects. The redirect is rejected if necessary:

```
if (rt->rt_gateway != old_gw)
    return;

in_dev = __in_dev_get_rcu(dev);
if (!in_dev)
    return;

net = dev_net(dev);
if (new_gw == old_gw || !IN_DEV_RX_REDIRECTS(in_dev) ||
    ipv4_is_multicast(new_gw) || ipv4_is_lbcast(new_gw) ||
```

```
    ipv4_is_zeronet(new_gw))
    goto reject_redirect;

if (!IN_DEV_SHARED_MEDIA(in_dev)) {
    if (!inet_addr_onlink(in_dev, new_gw, old_gw))
        goto reject_redirect;
    if (IN_DEV_SEC_REDIRECTS(in_dev) && ip_fib_check_default(new_gw, dev))
        goto reject_redirect;
} else {
    if (inet_addr_type(net, new_gw) != RTN_UNICAST)
        goto reject_redirect;
}
```

A lookup in the neighboring subsystem is performed; the key to the lookup is the address of the advised gateway, new_gw, which was extracted from the ICMPv4 message in the beginning of this method:

```
n = ipv4_neigh_lookup(&rt->dst, NULL, &new_gw);
if (n) {
    if (!(n->nud_state & NUD_VALID)) {
        neigh_event_send(n, NULL);
    } else {
           if (fib_lookup(net, fl4, &res) == 0) {
              struct fib_nh *nh = &FIB_RES_NH(res);
```

Create / update a FIB nexthop exception, specifying the IP address of an advised gateway (new_gw):

```
               update_or_create_fnhe(nh, fl4->daddr, new_gw,
                              0, 0);
            }
            if (kill_route)
                rt->dst.obsolete = DST_OBSOLETE_KILL;
            call_netevent_notifiers(NETEVENT_NEIGH_UPDATE, n);
        }
        neigh_release(n);
    }
    return;

reject_redirect:
      . . .

(net/ipv4/route.c)
```

Now that we've covered how a received ICMPv4 message is handled, we can next tackle the IPv4 routing cache and the reasons for its removal.

IPv4 Routing Cache

In kernels prior to 3.6, there was an IPv4 routing cache with a garbage collector. The IPv4 routing cache was removed in kernel 3.6 (around July 2012). The FIB TRIE / FIB hash was a choice in the kernel for years, but not as the default. Having the FIB TRIE made it possible to remove the IPv4 routing cache, as it had Denial of Service (DoS) issues. FIB TRIE (also known as LC-trie) is the longest matching prefix lookup algorithm that performs better than FIB hash for large routing tables. It consumes more memory and is more complex, but since it performs better, it made the removal of the routing cache feasible. The FIB TRIE code was in the kernel for a long time before it was merged, but it was not the default. The main reason for the removal of the IPv4 routing cache was that launching DoS attacks against it was easy because the IPv4 routing cache created a cache entry for each unique flow. Basically that meant that by sending packets to random destinations, you could generate an unlimited amount of routing cache entries.

Merging the FIB TRIE entailed the removal of the routing cache and of some of the cumbersome FIB hash tables and of the routing cache garbage collector methods. This chapter discusses the routing cache very briefly. Because the novice reader may wonder what it is needed for, note that in the Linux-based software industry, in commercial distributions like RedHat Enterprise, the kernels are fully maintained and fully supported for a very long period of time (RedHat, for example, gives support for its distributions for up to seven years). So it is very likely that some readers will be involved in projects based on kernels prior to 3.6, where you will find the routing cache and the FIB hash-based routing tables. Delving into the theory and implementation details of the FIB TRIE data structure is beyond the scope of this book. To learn more, I recommend the article "TRASH—A dynamic LC-trie and hash data structure" by Robert Olsson and Stefan Nilsson, `www.nada.kth.se/~snilsson/publications/TRASH/trash.pdf`.

Note that with the IPv4 routing cache implementation, there is a single cache, regardless of how many routing tables are used (there can be up to 255 routing tables when using Policy Routing). Note that there was also support for IPv4 Multipath Routing cache, but it was removed in kernel 2.6.23, in 2007. In fact, it never did work very well and never got out of the experimental state.

For kernels prior to the 3.6 kernel, where the FIB TRIE is not yet merged, the lookup in the IPv4 routing subsystem was different: access to routing tables was preceded by access to the routing cache, the tables were organized differently, and there was a routing cache garbage collector, which was both asynchronous (periodic timer) and synchronous (activated under specific conditions, for example when the number of the cache entries exceeded some threshold). The cache was basically a big hash with the IP flow source address, destination address, and TOS as a key, associated with all flow-specific information like neighbor entry, PMTU, redirect, TCPMSS info, and so on. The benefit here is that cached entries were fast to look up and contained all the information needed by higher layers.

■ **Note** The following two sections ("Rx Path" and "Tx Path") refer to the 2.6.38 kernel.

Rx Path

In the Rx path, first the `ip_route_input_common()` method is invoked. This method performs a lookup in the IPv4 routing cache, which is much quicker than the lookup in the IPv4 routing tables. Lookup in these routing tables is based on the Longest Prefix Match (LPM) search algorithm. With the LPM search, the most specific table entry—the one with the highest subnet mask—is called the Longest Prefix Match. In case the lookup in the routing cache fails ("cache miss"), a lookup in the routing tables is being performed by calling the `ip_route_input_slow()` method. This method calls the `fib_lookup()` method to perform the actual lookup. Upon success, it calls the `ip_mkroute_input()` method which (among other actions) inserts the routing entry into the routing cache by calling the `rt_intern_hash()` method.

Tx Path

In the Tx path, first the `ip_route_output_key()` method is invoked. This method performs a lookup in the IPv4 routing cache. In case of a cache miss, it calls the `ip_route_output_slow()` method, which calls the `fib_lookup()` method to perform a lookup in the routing subsystem. Subsequently, upon success, it calls the `ip_mkroute_output()` method which (among other actions) inserts the routing entry into the routing cache by calling the `rt_intern_hash()` method.

Summary

This chapter covered various topics of the IPv4 routing subsystem. The routing subsystem is essential for handling both incoming and outgoing packets. You learned about various topics like forwarding, lookup in the routing subsystem, organization of the FIB tables, Policy Routing and the routing subsystem, and ICMPv4 Redirect message. You also learned about optimization which is gained with the FIB alias and the fact that the routing cache was removed, and why. The next chapter covers advanced topics of the IPv4 routing subsystem.

Quick Reference

I conclude this chapter with a short list of important methods, macros, and tables of the IPv4 routing subsystem, along with a short explanation about routing flags.

■ **Note** The IPv4 routing subsystem is implemented in these modules under `net/ipv4`: `fib_frontend.c`, `fib_trie.c`, `fib_semantics.c`, `route.c`.

The `fib_rules.c` module implements Policy Routing and is compiled only when CONFIG_IP_MULTIPLE_TABLES is set. Among the most important header files are `fib_lookup.h`, `include/net/ip_fib.h`, and `include/net/route.h`.

The destination cache (`dst`) implementation is in `net/core/dst.c` and in `include/net/dst.h`.

CONFIG_IP_ROUTE_MULTIPATH should be set for Multipath Routing Support.

Methods

This section lists the methods that were mentioned in this chapter.

int fib_table_insert(struct fib_table *tb, struct fib_config *cfg);

This method inserts an IPv4 routing entry to the specified FIB table (`fib_table` object), based on the specified `fib_config` object.

int fib_table_delete(struct fib_table *tb, struct fib_config *cfg);

This method deletes an IPv4 routing entry from the specified FIB table (`fib_table` object), based on the specified `fib_config` object.

struct fib_info *fib_create_info(struct fib_config *cfg);

This method creates a `fib_info` object derived from the specified `fib_config` object.

void free_fib_info(struct fib_info *fi);

This method frees a `fib_info` object in condition that it is not alive (the `fib_dead` flag is not 0) and decrements the global `fib_info` objects counter (`fib_info_cnt`).

void fib_alias_accessed(struct fib_alias *fa);

This method sets the `fa_state` flag of the specified `fib_alias` to be FA_S_ACCESSED. Note that the only `fa_state` flag is FA_S_ACCESSED.

void ip_rt_send_redirect(struct sk_buff *skb);

This method sends an ICMPV4 Redirect message, as a response to a suboptimal path.

void __ip_do_redirect(struct rtable *rt, struct sk_buff *skb, struct flowi4*fl4, bool kill_route);

This method handles receiving an ICMPv4 Redirect message.

void update_or_create_fnhe(struct fib_nh *nh, __be32 daddr, __be32 gw, u32 pmtu, unsigned long expires);

This method creates a FIB nexthop exception table (`fib_nh_exception`) in the specified nexthop object (`fib_nh`), if it does not already exist, and initializes it. It is invoked when there should be a route update due to ICMPv4 redirect or due to PMTU discovery.

u32 dst_metric(const struct dst_entry *dst, int metric);

This method returns a metric of the specified `dst` object.

struct fib_table *fib_trie_table(u32 id);

This method allocates and initializes a FIB TRIE table.

struct leaf *fib_find_node(struct trie *t, u32 key);

This method performs a TRIE lookup with the specified key. It returns a `leaf` object upon success, or NULL in case of failure.

Macros

This section is a list of macros of the IPv4 routing subsystem, some of which were mentioned in this chapter.

FIB_RES_GW()

This macro returns the `nh_gw` field (nexthop gateway address) associated with the specified `fib_result` object.

FIB_RES_DEV()

This macro returns the `nh_dev` field (Next hop `net_device` object) associated with the specified `fib_result` object.

FIB_RES_OIF()

This macro returns the `nh_oif` field (nexthop output interface index) associated with the specified `fib_result` object.

FIB_RES_NH()

This macro returns the nexthop (`fib_nh` object) of the `fib_info` of the specified `fib_result` object. When Multipath Routing is set, you can have multiple nexthops; the value of `nh_sel` field of the specified `fib_result` object is taken into account in this case, as an index to the array of the nexthops which is embedded in the `fib_info` object.

(`include/net/ip_fib.h`)

IN_DEV_FORWARD()

This macro checks whether the specified network device (`in_device` object) supports IPv4 forwarding.

IN_DEV_RX_REDIRECTS()

This macro checks whether the specified network device (`in_device` object) supports accepting ICMPv4 Redirects.

IN_DEV_TX_REDIRECTS()

This macro checks whether the specified network device (`in_device` object) supports sending ICMPv4 Redirects.

IS_LEAF()

This macro checks whether the specified tree node is a leaf.

IS_TNODE()

This macro checks whether the specified tree node is an internal node (`trie` node or `tnode`).

change_nexthops()

This macro iterates over the nexthops of the specified `fib_info` object (`net/ipv4/fib_semantics.c`).

Tables

There are 15 (RTAX_MAX) metrics for routes. Some of them are TCP related, and some are general. Table 5-1 shows which of these metrics are related to TCP.

Table 5-1. *Route Metrics*

Linux Symbol	TCP Metric (Y/N)
RTAX_UNSPEC	N
RTAX_LOCK	N
RTAX_MTU	N
RTAX_WINDOW	Y
RTAX_RTT	Y
RTAX_RTTVAR	Y
RTAX_SSTHRESH	Y
RTAX_CWND	Y
RTAX_ADVMSS	Y
RTAX_REORDERING	Y
RTAX_HOPLIMIT	N
RTAX_INITCWND	Y
RTAX_FEATURES	N
RTAX_RTO_MIN	Y
RTAX_INITRWND	Y

(`include/uapi/linux/rtnetlink.h`)

Table 5-2 shows the error value and the scope of all the route types.

Table 5-2. *Route Types*

Linux Symbol	Error	Scope
RTN_UNSPEC	0	RT_SCOPE_NOWHERE
RTN_UNICAST	0	RT_SCOPE_UNIVERSE
RTN_LOCAL	0	RT_SCOPE_HOST
RTN_BROADCAST	0	RT_SCOPE_LINK
RTN_ANYCAST	0	RT_SCOPE_LINK
RTN_MULTICAST	0	RT_SCOPE_UNIVERSE
RTN_BLACKHOLE	-EINVAL	RT_SCOPE_UNIVERSE
RTN_UNREACHABLE	-EHOSTUNREACH	RT_SCOPE_UNIVERSE
RTN_PROHIBIT	-EACCES	RT_SCOPE_UNIVERSE
RTN_THROW	-EAGAIN	RT_SCOPE_UNIVERSE
RTN_NAT	-EINVAL	RT_SCOPE_NOWHERE
RTN_XRESOLVE	-EINVAL	RT_SCOPE_NOWHERE

Route Flags

When running the `route -n` command, you get an output that shows the route flags. Here are the flag values and a short example of the output of `route -n`:

U (Route is up)

H (Target is a host)

G (Use gateway)

R (Reinstate route for dynamic routing)

D (Dynamically installed by daemon or redirect)

M (Modified from routing daemon or redirect)

A (Installed by addrconf)

! (Reject route)

Table 5-3 shows an example of the output of running `route -n` (the results are organized into a table form):

Table 5-3. *Kernel IP Routing Table*

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
169.254.0.0	0.0.0.0	255.255.0.0	U	1002	0	0	eth0
192.168.3.0	192.168.2.1	255.255.255.0	UG	0	0	0	eth1