CHAPTER 4

IPv4

Chapter 3 deals with the implementation of the ICMP protocol in IPv4 and in IPv6. This chapter, which deals with the IPv4 protocol, shows how ICMP messages are used for reporting Internet protocol errors under certain circumstances. The IPv4 protocol (Internet Protocol version 4) is one of the core protocols of today's standards-based Internet and routes most of the traffic on the Internet. The base definition is in RFC 791, "Internet Protocol," from 1981. The IPv4 protocol provides an end-to-end connectivity between any two hosts. Another important function of the IP layer is forwarding packets (also called routing) and managing tables that store routing information. Chapters 5 and 6 discuss IPv4 routing. This chapter describes the IPv4 Linux implementation: receiving and sending IPv4 packets, including multicast packets, IPv4 forwarding, and handling IPv4 options. There are cases when the packet to be sent is bigger than the MTU of the outgoing interface; in such cases the packet should be fragmented into smaller fragments. When fragmented packets are received, they should be assembled into one big packet, which should be identical to the packet that was sent before it was fragmented. These are also important tasks of the IPv4 protocol discussed in this chapter.

Every IPv4 packet starts with an IP header, which is at least 20 bytes long. If IP options are used, the IPv4 header can be up to 60 bytes. After the IP header, there is the transport header (TCP header or UDP header, for example), and after it is the payload data. To understand the IPv4 protocol, you must first learn how the IPv4 header is built. In Figure 4-1 you can see the IPv4 header, which consists of two parts: the first part of 20 bytes (until the beginning of the options field in the IPv4 header) is the basic IPv4 header, and after it there is the IP options part, which can be from 0 to 40 bytes in length.

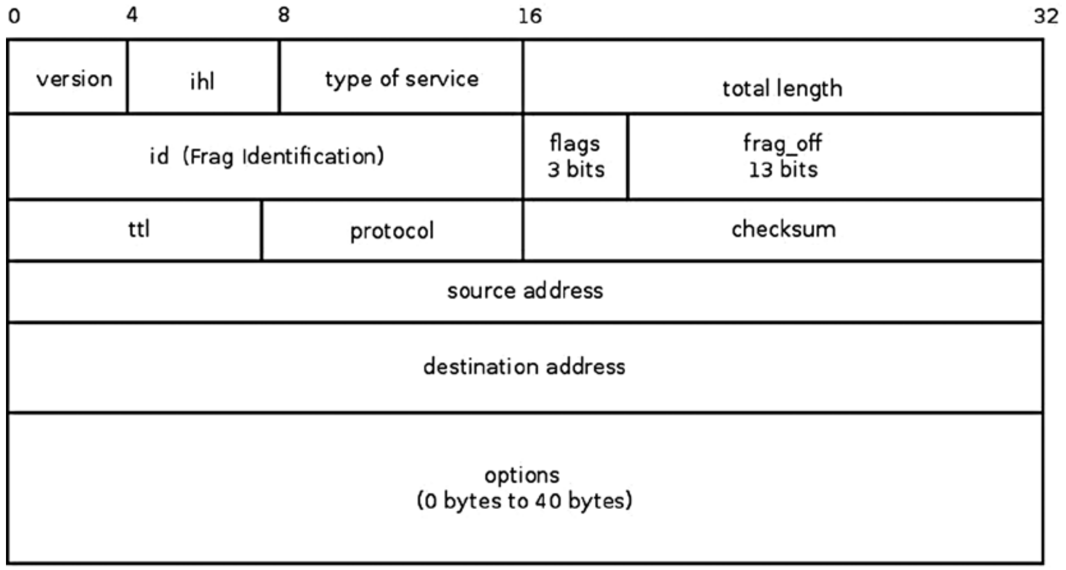

Figure 4-1. *IPv4 header*

IPv4 Header

The IPv4 header consists of information that defines how a packet should be handled by the kernel network stack: the protocol being used, the source and destination address, the checksum, the identification (`id`) of the packet that is needed for fragmentation, the `ttl` that helps avoiding packets being forwarded endlessly because of some error, and more. This information is stored in 13 members of the IPv4 header (the 14th member, IP Options, which is an extension to the IPv4 header, is optional). The various members of the IPv4 and the various IP options are described next. The IPv4 header is represented by the `iphdr` structure. Its members, which appear in Figure 4-1, are described in the next section. The IP options and their use are described in the "IP Options" section later in this chapter.

Figure 4-1 shows the IPv4 header. All members always exist—except for the last one, the IP options, which is optional. The content of the IPv4 members determines how it will be handled in the IPv4 network stack: the packet is discarded when there is some problem (for example, if the version, which is the first member, is not 4, or if the checksum is incorrect). Each IPv4 packet starts with IPv4 header, and after it there is the payload:

```
struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
    __u8    ihl:4,
            version:4;
#elif defined (__BIG_ENDIAN_BITFIELD)
    __u8    version:4,
            ihl:4;
#else
#error    "Please fix <asm/byteorder.h>"
#endif
    __u8          tos;
    __be16        tot_len;
    __be16        id;
    __be16        frag_off;
    __u8          ttl;
    __u8          protocol;
    __sum16       check;
    __be32        saddr;
    __be32        daddr;
    /*The options start here. */
};

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

The following is a description of the IPv4 header members:

- `ihl`: This stands for Internet Header Length. The length of the IPv4 header, measured in multiples of 4 bytes. The length of the IPv4 header is not fixed, as opposed to the header of IPv6, where the length is fixed (40 bytes). The reason is that the IPv4 header can include optional, varying length options. The minimum size of the IPv4 header is 20 bytes, when there are no options, and the maximum size is 60 bytes. The corresponding `ihl` values are 5 for minimum IPv4 header size, and 15 for the maximum size. The IPv4 header must be aligned to a 4-byte boundary.
- `version`: Should be 4.
- `tos`: The `tos` field of the IPv4 header was originally intended for Quality of Service (QoS) services; `tos` stands for Type of Service. Over the years this field took on a different meaning, as follows: RFC 2474 defines the Differentiated Services Field (DS Field) in the IPv4 and IPv6 headers, which is bits 0–5 of the `tos`. It is also named Differentiated Services Code Point (DSCP). RFC 3168 from 2001 defines the Explicit Congestion Notification (ECN) of the IP header; it is bits 6 and 7 of the `tos` field.
- `tot_len`: The total length, including the header, measured in bytes. Because `tot_len` is a 16-bit field, it can be up to 64KB. According to RFC 791, the minimum size is 576 bytes.
- `id`: Identification of the IPv4 header. The `id` field is important for fragmentation: when fragmenting an SKB, the `id` value of all the fragments of that SKB should be the same. Reassembling fragmented packets is done according to the `id` of the fragments.
- `frag_off`: The fragment offset, a 16-bit field. The lower 13 bits are the offset of the fragment. In the first fragment, the offset is 0. The offset is measured in units of 8 bytes. The higher 3 bits are the flags:
    - 001 is MF (More Fragments). It is set for all fragments, except the last one.
    - 010 is DF (Don't Fragment).
    - 100 is CE (Congestion).

  See the IP_MF, IP_DF, and IP_CE flags declaration in `include/net/ip.h`.
- `ttl`: Time To Live: this is a hop counter. Each forwarding node decreases the `ttl` by 1. When it reaches 0, the packet is discarded, and a time exceeded ICMPv4 message is sent back; this avoids packets from being forwarded endlessly, for this reason or another.
- `protocol`: The L4 protocol of the packet—for example, IPPROTO_TCP for TCP traffic or IPPROTO_UDP for UDP traffic (for a list of all available protocols see `include/linux/in.h`).
- `check`: The checksum (16-bit field). The checksum is calculated only over the IPv4 header bytes.
- `saddr`: Source IPv4 address, 32 bits.
- `daddr`: Destination IPv4 address, 32 bits.

In this section you have learned about the various IPv4 header members and their purposes. The initialization of the IPv4 protocol, which sets the callback to be invoked when receiving an IPv4 header, is discussed in the next section.

IPv4 Initialization

IPv4 packets are packets with Ethernet type 0x0800 (Ethernet type is stored in the first two bytes of the 14-byte Ethernet header). Each protocol should define a protocol handler, and each protocol should be initialized so that the network stack can handle packets that belong to this protocol. So that you understand what causes received IPv4 packets to be handled by IPv4 methods, this section describes the registration of the IPv4 protocol handler :

```
static struct packet    _type ip_packet_type __read_mostly = {
    .type = cpu_to_be16(ETH_P_IP),
    .func = ip_rcv,
};

static int __init inet_init(void)
{
  ...
  dev_add_pack(&ip_packet_type);
  ...
}

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

The `dev_add_pack()` method adds the `ip_rcv()` method as a protocol handler for IPv4 packets. These are packets with Ethernet type 0x0800 (ETH_P_IP, defined in `include/uapi/linux/if_ether.h`). The `inet_init()` method performs various IPv4 initializations and is called during the boot phase.

The main functionality of the IPv4 protocol is divided into the Rx (receive) path and the Tx (transmit) path. Now that you learned about the registration of the IPv4 protocol handler, you know which protocol handler manages IPv4 packets (the `ip_rcv` callback) and how this protocol handler is registered. You are ready now to start to learn about the IPv4 Rx path and how received IPv4 packets are handled. The Tx path is described in a later section, "Sending IPv4 Packets."

Receiving IPv4 Packets

The main IPv4 receive method is the `ip_rcv()` method, which is the handler for all IPv4 packets (including multicasts and broadcasts). In fact, this method consists mostly of sanity checks. The real work is done in the `ip_rcv_finish()` method it invokes. Between the `ip_rcv()` method and the `ip_rcv_finish()` method is the NF_INET_PRE_ROUTING netfilter hook, invoked by calling the NF_HOOK macro (see code snippet later in this section). In this chapter, you will encounter many invocations of the NF_HOOK macros—these are the netfilter hooks. The netfilter subsystem allows you to register callbacks in five points along the journey of a packet in the network stack. These points will be mentioned by their names shortly. The reason for adding the netfilter hooks is to enable loading the netfilter kernel modules at runtime. The NF_HOOK macro invokes the callbacks of a specified point, if such callbacks were registered. You might also encounter the NF_HOOK macro called NF_HOOK_COND, which is a variation of the NF_HOOK macro. In some places in the network stack, the NF_HOOK_COND macro includes a Boolean parameter (the last parameter), which must be `true` for the hook to be executed (Chapter 9 discusses netfilter hooks). Note that the netfilter hooks can discard the packet and in such a case it will not continue on its ordinary path. Figure 4-2 shows the receiving path (Rx) of a packet received by the network driver. This packet can either be delivered to the local machine or be forwarded to another host. It is the lookup in the routing table that determines which of these two options will take place.

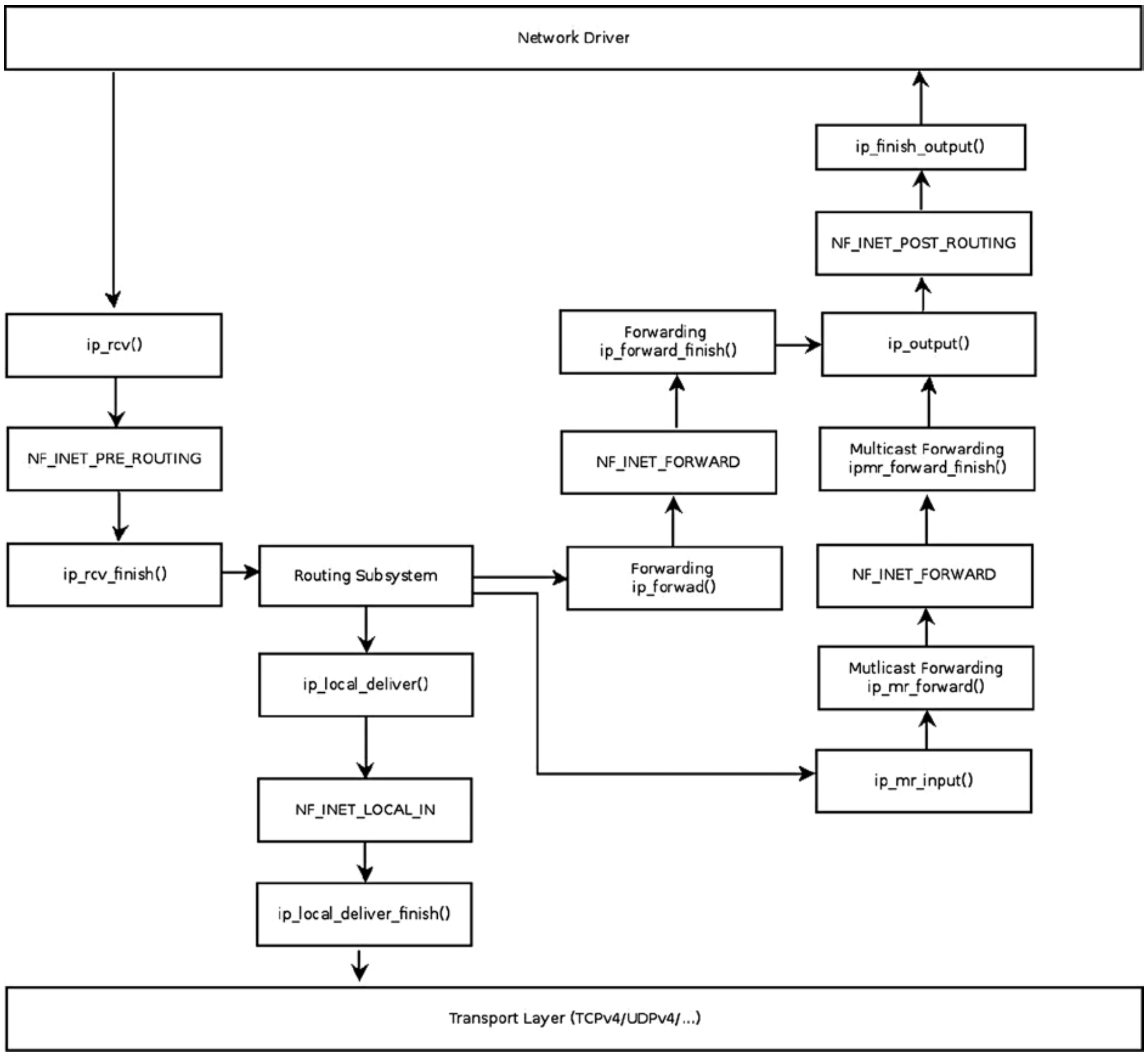

Figure 4-2. *Receiving IPv4 packets. For simplicity, the diagram does not include the fragmentation/defragmentation/options/IPsec methods*

Figure 4-2 shows the paths for a received IPv4 packet. The packet is received by the IPv4 protocol handler, the `ip_rcv()` method (see the upper left side of the figure). First of all, a lookup in the routing subsystem should be performed, immediately after calling the `ip_rcv_finish()` method. The result of the routing lookup determines whether the packet is for local delivery to the local host or is to be forwarded (routing lookup is explained in Chapter 5). If the packet is destined for the local host, it will first reach the `ip_local_deliver()` method, and subsequently it will reach the `ip_local_deliver_finish()` method. When the packet is to be forwarded, it will be handled by the `ip_forward()` method. Some netfilter hooks appear in the figure, like NF_INET_PRE_ROUTING and NF_INET_LOCAL_IN. Note that multicast traffic is handled by the `ip_mr_input()` method, discussed in the "Receiving IPv4 Multicast Packets" section later in this chapter. The NF_INET_PRE_ROUTING, NF_INET_LOCAL_IN, NF_INET_FORWARD, and NF_INET_POST_ROUTING are four of the five entry points of the netfilter hooks. The fifth one, NF_INET_LOCAL_OUT, is mentioned in the "Sending IPv4 packets" section later in this chapter. These five entry points are defined in `include/uapi/linux/netfilter.h`. Note that the same `enum` for these five hooks is also used in IPv6; for example, in the `ipv6_rcv()` method, a hook is being registered on NF_INET_PRE_ROUTING (`net/ipv6/ip6_input.c`). Let's take a look at the `ip_rcv()` method:

```
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt, struct net_device
*orig_dev)
{
```

First some sanity checks are performed, and I mention some of them in this section. The length of the IPv4 header (`ihl`) is measured in multiples of 4 bytes. The IPv4 header must be at least 20 bytes in size, which means that the `ihl` size must be at least 5. The `version` should be 4 (for IPv4). If one of these conditions is not met, the packet is dropped and the statistics (IPSTATS_MIB_INHDRERRORS) are updated.

```
        if (iph->ihl < 5 || iph->version != 4)
                goto inhdr_error;
```

According to section 3.2.1.2 of RFC 1122, a host must verify the IPv4 header checksum on every received datagram and silently discard every datagram that has a bad checksum. This is done by calling the `ip_fast_csum()` method, which should return 0 on success. The IPv4 header checksum is calculated only over the IPv4 header bytes:

```
        if (unlikely(ip_fast_csum((u8 *)iph, iph->ihl)))
                goto inhdr_error;
```

Then the NF_HOOK macro is invoked:

```
         return NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING, skb, dev, NULL,
                        ip_rcv_finish);
```

When the registered netfilter hook method returns NF_DROP, it means that the packet should be dropped, and the packet traversal does not continue. When the registered netfilter hook returns NF_STOLEN, it means that the packet was taken over by the netfilter subsystem, and the packet traversal does not continue. When the registered netfilter hook returns NF_ACCEPT, the packet continues its traversal. There are other return values (also termed *verdicts*) from netfilter hooks, like NF_QUEUE, NF_REPEAT, and NF_STOP, which are not discussed in this chapter. (As mentioned earlier, netfilter hooks are discussed in Chapter 9.) Let's assume for a moment that there are no netfilter callbacks registered in the NF_INET_PRE_ROUTING entry point, so the NF_HOOK macro will not invoke any netfilter callbacks and the `ip_rcv_finish()` method will be invoked. Let's take a look at the `ip_rcv_finish()` method:

```
static int ip_rcv_finish(struct sk_buff *skb)
{
       const struct iphdr *iph = ip_hdr(skb);
       struct rtable *rt;
```

The `skb_dst()` method checks whether there is a `dst` object attached to the SKB; `dst` is an instance of `dst_entry` (`include/net/dst.h`) and represents the result of a lookup in the routing subsystem. The lookup is done according to the routing tables and the packet headers. The lookup in the routing subsystem also sets the `input` and /or the `output` callbacks of the `dst`. For example, if the packet is to be forwarded, the lookup in the routing subsystem will set the `input` callback to be `ip_forward()`. When the packet is destined to the local machine, the lookup in the routing subsystem will set the `input` callback to be `ip_local_deliver()`. For a multicast packet it can be `ip_mr_input()` under some conditions (I discuss multicast packets in the next section). The contents of the `dst` object determine how the packet will proceed in its journey; for example, when forwarding a packet, the decision about which `input` callback should be called when invoking dst_input(), or on which interface it should be transmitted, is taken according to the dst.(I discuss the routing subsystem in depth in the next chapter).

If there is no dst attached to the SKB, a lookup in the routing subsystem is performed by the ip_route_input_noref() method. If the lookup fails, the packet is dropped. Note that handling multicast packets is different than handling unicast packets (discussed in the section "Receiving IPv4 Multicast Packets" later in this chapter).

```
        ...
        if (!skb_dst(skb)) {
```

Perform a lookup in the routing subsystem:

```
                int err = ip_route_input_noref(skb, iph->daddr, iph->saddr,
                                               iph->tos, skb->dev);
                if (unlikely(err)) {
                    if (err == -EXDEV)
                        NET_INC_STATS_BH(dev_net(skb->dev),
                                         LINUX_MIB_IPRPFILTER);
                    goto drop;
                }
        }
```

■ **Note** The -EXDEV ("Crossdevice link") error is returned by the __fib_validate_source() method under certain circumstances when the Reverse Path Filter (RPF) is set. The RPF can be set via an entry in the procfs. In such cases the packet is dropped, the statistics (LINUX_MIB_IPRPFILTER) are updated, and the method returns NET_RX_DROP. Note that you can display the LINUX_MIB_IPRPFILTER counter by looking in the IPReversePathFilter column in the output of cat /proc/net/netstat.

Now a check is performed to see whether the IPv4 header includes options. Because the length of the IPv4 header (ihl) is measured in multiples of 4 bytes, if it is greater than 5 this means that it includes options, so the ip_rcv_options() method should be invoked to handle these options. Handling IP options is discussed in depth in the "IP Options" section later in this chapter. Note that the ip_rcv_options() method can fail, as you will shortly see. If it is a multicast entry or a broadcast entry, the IPSTATS_MIB_INMCAST statistics or the IPSTATS_MIB_INBCAST statistics is updated, respectively. Then the dst_input() method is invoked. This method in turn simply invokes the input callback method by calling skb_dst(skb)->input(skb):

```
    if (iph->ihl > 5 && ip_rcv_options(skb))
            goto drop;

    rt = skb_rtable(skb);
    if (rt->rt_type == RTN_MULTICAST) {
        IP_UPD_PO_STATS_BH(dev_net(rt->dst.dev), IPSTATS_MIB_INMCAST,
                skb->len);
    } else if (rt->rt_type == RTN_BROADCAST)
        IP_UPD_PO_STATS_BH(dev_net(rt->dst.dev), IPSTATS_MIB_INBCAST,
                skb->len);

    return dst_input(skb);
```

In this section you learned about the various stages in the reception of IPv4 packets: the sanity checks performed, the lookup in the routing subsystem, the ip_rcv_finish() method which performs the actual work. You also learned about which method is called when the packet should be forwarded and which method is called when the packet is for local delivery. IPv4 multicasting is a special case. Handling the reception of IPv4 multicast packets is discussed in the next section.

Receiving IPv4 Multicast Packets

The ip_rcv() method is also a handler for multicast packets. As mentioned earlier, after some sanity checks, it invokes the ip_rcv_finish() method, which performs a lookup in the routing subsystem by calling ip_route_input_noref(). In the ip_route_input_noref() method, first a check is performed to see whether the local machine belongs to a multicast group of the destination multicast address, by calling the ip_check_mc_rcu() method. If it is so, or if the local machine is a multicast router (CONFIG_IP_MROUTE is set), the ip_route_input_mc() method is invoked; let's take a look at the code:

```
int ip_route_input_noref(struct sk_buff *skb, __be32 daddr, __be32 saddr,
                         u8 tos, struct net_device *dev)
{
        int res;
        rcu_read_lock();
        . . .
        if (ipv4_is_multicast(daddr)) {
                struct in_device *in_dev = __in_dev_get_rcu(dev);
                if (in_dev) {
                        int our = ip_check_mc_rcu(in_dev, daddr, saddr,
                                                  ip_hdr(skb)->protocol);
                        if (our
#ifdef CONFIG_IP_MROUTE
                                ||
                            (!ipv4_is_local_multicast(daddr) &&
                             IN_DEV_MFORWARD(in_dev))
#endif
                           ) {
                                int res = ip_route_input_mc(skb, daddr, saddr,
                                                            tos, dev, our);
                                rcu_read_unlock();
                                return res;
                        }
                }
           . . .

        }
        . . .
```

Let's further look into the ip_route_input_mc() method. If the local machine belongs to a multicast group of the destination multicast address (the value of the variable our is 1), then the input callback of dst is set to be ip_local_deliver. If the local host is a multicast router and IN_DEV_MFORWARD(in_dev) is set, then the input callback of dst is set to be ip_mr_input. The ip_rcv_finish() method, which calls dst_input(skb), invokes thus either the ip_local_deliver() method or the ip_mr_input() method, according to the input callback of dst. The IN_DEV_MFORWARD macro checks the procfs multicast forwarding entry. Note that the procfs multicast forwarding entry, /proc/sys/net/ipv4/conf/all/mc_forwarding , is a read-only entry (as opposed to the IPv4 unicast procfs forwarding entry), so you cannot set it simply by running from the command line: echo 1 > /proc/sys/net/ipv4/conf/all/mc_forwarding. Starting the pimd daemon, for example, sets it to 1, and stopping the daemon sets it to 0. pimd is a lightweight standalone PIM-SM v2 multicast routing daemon. If you are interested in learning about multicast routing daemon implementation, you might want to look into the pimd source code in https://github.com/troglobit/pimd/:

```
static int ip_route_input_mc(struct sk_buff *skb, __be32 daddr, __be32 saddr,
                                 u8 tos, struct net_device *dev, int our)
 {
        struct rtable *rth;
        struct in_device *in_dev = __in_dev_get_rcu(dev);

       . . .

        if (our) {
                rth->dst.input= ip_local_deliver;
                rth->rt_flags |= RTCF_LOCAL;
        }

 #ifdef CONFIG_IP_MROUTE
        if (!ipv4_is_local_multicast(daddr) && IN_DEV_MFORWARD(in_dev))
                rth->dst.input = ip_mr_input;
 #endif
       . . .
```

The multicast layer holds a data structure called the Multicast Forwarding Cache (MFC). I don't discuss the details of the MFC or of the ip_mr_input() method here (I discuss them in Chapter 6). What is important in this context is that if a valid entry is found in the MFC, the ip_mr_forward() method is called. The ip_mr_forward() method performs some checks and eventually calls the ipmr_queue_xmit() method. In the ipmr_queue_xmit() method, the ttl is decreased, and the checksum is updated by calling the ip_decrease_ttl() method (the same is done in the ip_forward() method, as you will see later in this chapter). Then the ipmr_forward_finish() method is invoked by calling the NF_INET_FORWARD NF_HOOK macro (let's assume that there are no registered IPv4 netfilter hooks on NF_INET_FORWARD):

```
static void ipmr_queue_xmit(struct net *net, struct mr_table *mrt,
                             struct sk_buff *skb, struct mfc_cache *c, int vifi)
{
       . . .

       ip_decrease_ttl(ip_hdr(skb));
       ...
       NF_HOOK(NFPROTO_IPV4, NF_INET_FORWARD, skb, skb->dev, dev,
                       ipmr_forward_finish);
       return;

}
```

The `ipmr_forward_finish()` method is very short and is shown here in its entirety. All it does is update the statistics, call the `ip_forward_options()` method if there are options in the IPv4 header (IP options are described in the next section), and call the `dst_output()` method:

```
static inline int ipmr_forward_finish(struct sk_buff *skb)
{
        struct ip_options *opt = &(IPCB(skb)->opt);

        IP_INC_STATS_BH(dev_net(skb_dst(skb)->dev), IPSTATS_MIB_OUTFORWDATAGRAMS);
    IP_ADD_STATS_BH(dev_net(skb_dst(skb)->dev), IPSTATS_MIB_OUTOCTETS, skb->len);

        if (unlikely(opt->optlen))
                ip_forward_options(skb);

        return dst_output(skb);
}
```

This section discussed how receiving IPv4 multicast packets is handled. The `pimd` was mentioned as an example of a multicast routing daemon, which interacts with the kernel in multicast packet forwarding. The next section describes the various IP options, which enable using special features of the network stack, such as tracking the route of a packet, tracking timestamps of packets, specifying network nodes which a packet should traverse. I also discuss how these IP options are handled in the network stack.

IP Options

The IP options field of the IPv4 header is optional and is not often used for security reasons and because of processing overhead. Which options might be helpful? Suppose, for example, that your packets are being dropped by a certain firewall. You may be able to specify a different route with the Strict or Loose Source Routing options. Or if you want to find out the packets' path to some destination addresses, you can use the Record Route option.

The IPv4 header may contain zero, one, or more options. The IPv4 header size is 20 bytes when there are no options. The length of the IP options field can be 40 bytes at most. The reason the IPv4 maximum length is 60 bytes is because the IPv4 header length is a 4-bit field, which expresses the length in multiples of 4 bytes. Hence the maximum value of the field is 15, which gives an IPv4 maximum header length of 60 bytes. When using more than one option, options are simply concatenated one after the other. The IPv4 header must be aligned to a 4-byte boundary, so sometimes padding is needed. The following RFCs discuss IP options: 781 (Timestamp Option), 791, 1063, 1108, 1393 (Traceroute Using an IP Option), and 2113 (IP Router Alert Option). There are two forms of IP options:

- *Single byte option (option type)*: The "End of Option List" and "No Operation" are the only single byte options.
- *Multibyte option*: When using a multibyte option after the option type byte there are the following three fields:
  - *Length (1 byte)*: Length of the option in bytes.
  - *Pointer (1 byte)*: Offset from option start.
  - *Option data*: This is a space where intermediate hosts can store data, for example, timestamps or IP addresses.

In Figure 4-3 the Option type is shown.

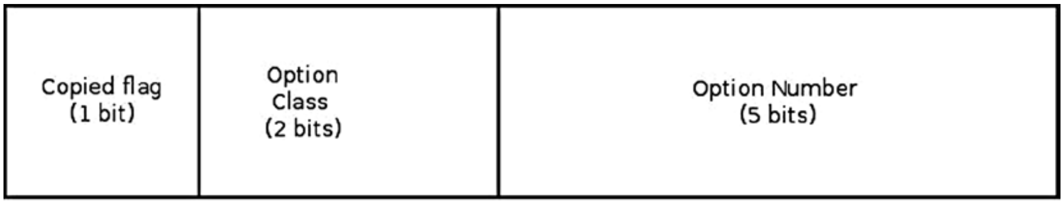

Figure 4-3. *Option type*

When set, copied flag means that the option should be copied in all fragments. When it is not set, the option should be copied only in the first fragment. The IPOPT_COPIED macro checks whether the copied flag of a specified IP option is set. It is used in the ip_options_fragment() method for detecting options which may not be copied and for inserting IPOPT_NOOP instead. The ip_options_fragment() method is discussed later in this section.

The option class can be one of the following 4 values:

- 00: control class (IPOPT_CONTROL)
- 01: reserved1 (IPOPT_RESERVED1)
- 10: debugging and measurement (IPOPT_MEASUREMENT)
- 11: reserved2 (IPOPT_RESERVED2)

In the Linux network stack, only the IPOPT_TIMESTAMP option belongs to the debugging and measurement class. All the other options are control classes.

The Option Number specifies an option by a unique number; possible values are 0–31, but not all are used by the Linux kernel.

Table 4-1 shows all options according to their Linux symbol, option number, option class, and copied flag.

Table 4-1. *Options Table*

Linux Symbol	Option Number	Class	Copied Flag	Description
IPOPT_END	0	0	0	End of Option List
IPOPT_NOOP	1	0	0	No Operation
IPOPT_SEC	2	0	1	Security
IPOPT_LSRR	3	0	1	Loose Source Record Route
IPOPT_TIMESTAMP	4	2	0	Timestamp
IPOPT_CIPSO	6	0	1	Commercial Internet Protocol Security Option
IPOPT_RR	7	0	0	Record Route
IPOPT_SID	8	0	1	Stream ID
IPOPT_SSRR	9	0	1	Strict Source Record Route
IPOPT_RA	20	0	1	Router Alert

The option names (IPOPT_*) declarations are in `include/uapi/linux/ip.h`.

The Linux network stack does not include all the IP options. For a full list, see `www.iana.org/assignments/ip-parameters/ip-parameters.xml`.

I will describe the five options shortly, and then describe the Timestamp Option and the Record Route option in depth:

- *End of Option List (IPOPT_END)*: 1-byte option used to indicate the end of the options field. This is a single zero byte option (all its bits are '0'). There can be no IP options after it.
- *No Operation (IPOPT_NOOP)*: 1-byte option is used for internal padding, which is used for alignment.
- *Security (IPOPT_SEC)*: This option provides a way for hosts to send security, handling restrictions, and TCC (closed user group) parameters. See RFC 791 and RFC 1108. Initially intended to be used by military applications.
- *Loose Source Record Route (IPOPT_LSRR)*: This option specifies a list of routers that the packet should traverse. Between each two adjacent nodes in the list there can be intermediate routers which do not appear in the list, but the order should be kept.
- *Commercial Internet Protocol Security Option (IPOPT_CIPSO)*: CIPSO is an IETF draft that has been adopted by several vendors. It deals with a network labeling standard. CIPSO labeling of a socket means adding the CIPSO IP options to all packets leaving the system through that socket. This option is validated upon reception of the packet. For more info about the CIPSO option, see `Documentation/netlabel/draft-ietf-cipso-ipsecurity-01.txt` and `Documentation/netlabel/cipso_ipv4.txt`.

Timestamp Option

Timestamp (IPOPT_TIMESTAMP): The Timestamp option is specified in RFC 781, "A Specification of the Internet Protocol (IP) Timestamp Option." This option stores timestamps of hosts along the packet route. The stored timestamp is a 32-bit timestamp in milliseconds since midnight UTC of the current day. In addition, it can also store the addresses of all hosts in the packet route or timestamps of only selected hosts along the route. The maximum Timestamp option length is 40. The Timestamp option is not copied for fragments; it is carried only in the first fragment. The Timestamp option begins with three bytes of option type, length, and pointer (offset). The higher 4 bits of the fourth byte are the overflow counter, which is incremented in each hop where there is no available space to store the required data. When the overflow counter exceeds 15, an ICMP message of Parameter Problem is sent back. The lower 4 bits is the flag. The value of the flag can be one of the following:

- *0*: Timestamp only (IPOPT_TS_TSONLY)
- *1*: Timestamps and addresses (IPOPT_TS_TSANDADDR)
- *3*: Timestamps of specified hops only (IPOPT_TS_PRESPEC)

■ **Note** You can use the command-line `ping` utility with the Timestamp option and with the three subtypes mentioned earlier:

```
ping -T tsonly     (IPOPT_TS_TSONLY)

ping -T tsandaddr  (IPOPT_TS_TSANDADDR)

ping -T tsprespec  (IPOPT_TS_PRESPEC)
```

Figure 4-4 shows the Timestamp option with timestamp only (the IPOPT_TS_TSONLY flag is set). Each router on the path adds its IPv4 address. When there is no more space, the overflow counter is incremented.

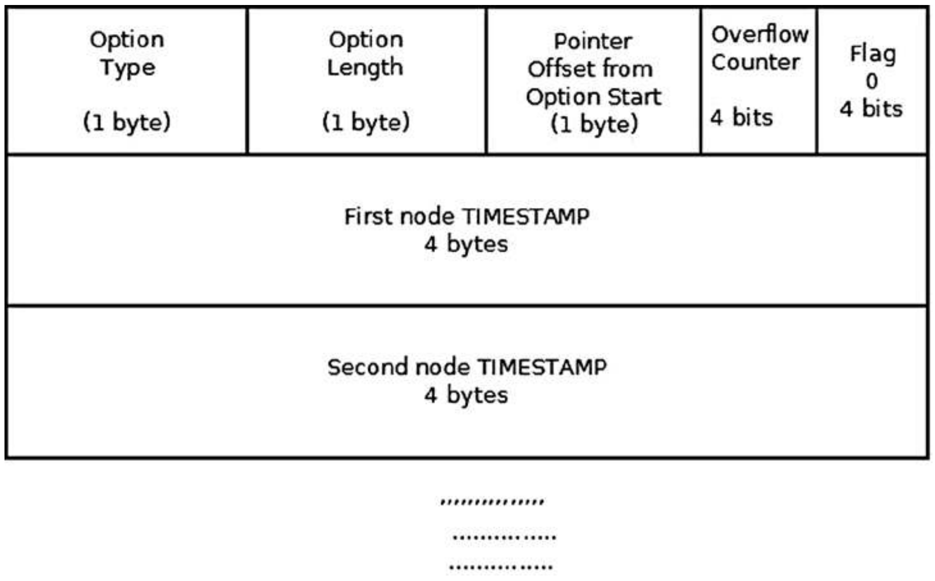
Option Type (1 byte)
Option Length (1 byte)
Pointer Offset from Option Start (1 byte)
Overflow Counter 4 bits
Flag 0 4 bits
First node TIMESTAMP 4 bytes
Second node TIMESTAMP 4 bytes

Figure 4-4. *Timestamp option (with timestamp only, flag = 0)*

Figure 4-5 shows the Timestamp option with timestamps and addresses (the IPOPT_TS_TSANDADDR flag is set). Each router on the path adds its IPv4 address and its timestamp. Again, when there is no more space, the overflow counter is incremented.

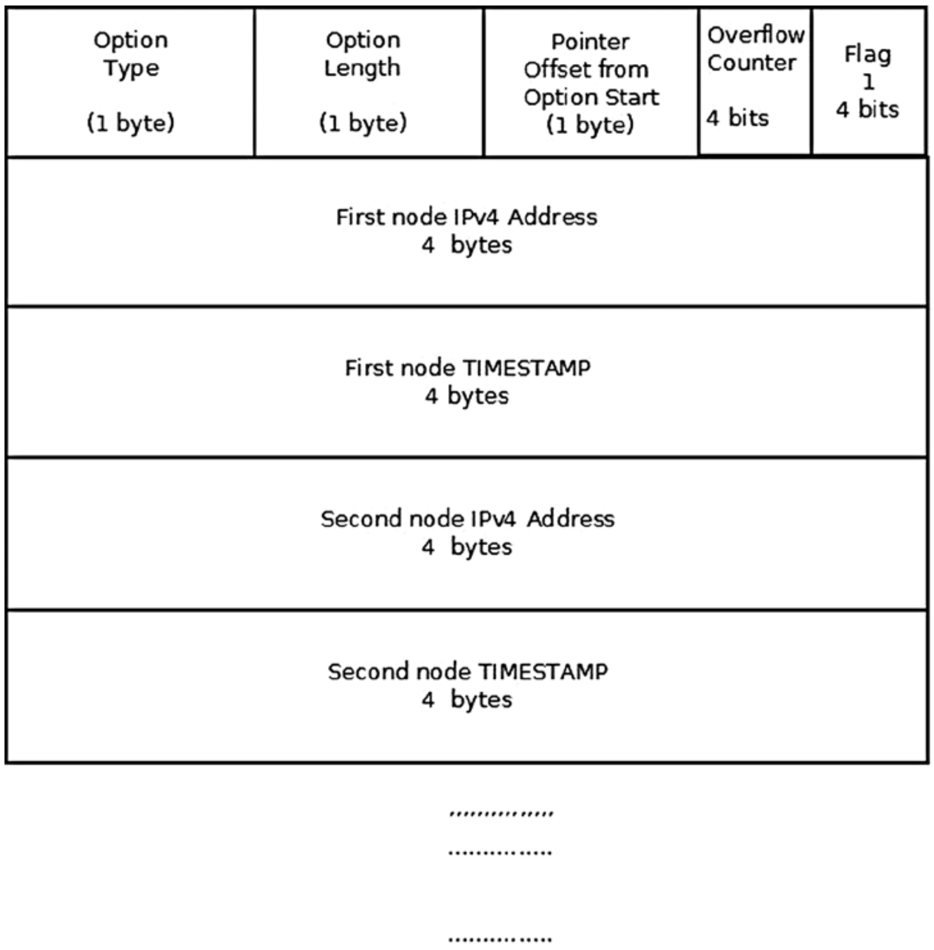
Option Type (1 byte)
Option Length (1 byte)
Pointer Offset from Option Start (1 byte)
Overflow Counter 4 bits
Flag 1 4 bits
First node IPv4 Address 4 bytes
First node TIMESTAMP 4 bytes
Second node IPv4 Address 4 bytes
Second node TIMESTAMP 4 bytes

Figure 4-5. *Timestamp option (with timestamps and addresses, flag = 1)*

Figure 4-6 shows the Timestamp option with timestamps (the IPOPT_TS_PRESPEC flag is set). Each router on the path adds its timestamp only if it is in the pre-specified list. Again, when there is no more space, the overflow counter is incremented.

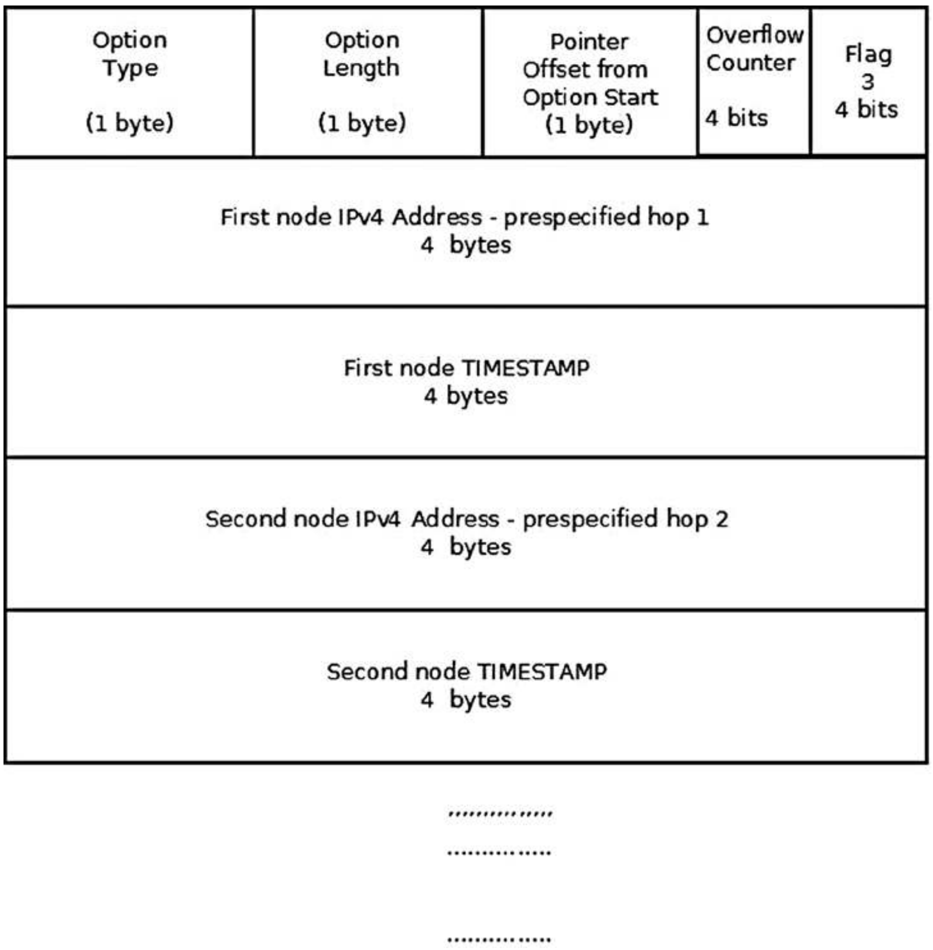
Option Type (1 byte)
Option Length (1 byte)
Pointer Offset from Option Start (1 byte)
Overflow Counter 4 bits
Flag 3 4 bits
First node IPv4 Address - prespecified hop 1 4 bytes
First node TIMESTAMP 4 bytes
Second node IPv4 Address - prespecified hop 2 4 bytes
Second node TIMESTAMP 4 bytes

Figure 4-6. *Timestamp option (with timestamps of specified hops only, flag = 3)*

Record Route Option

Record Route (IPOPT_RR): The route of a packet is recorded. Each router on the way adds its address (see Figure 4-7). The length is set by the sending device. The command-line utility `ping -R` uses the Record Route IP Option. Note that the IPv4 header is only large enough for nine such routes (or even less, if more options are used). When the header is full and there is no room to insert an additional address, the datagram is forwarded without inserting the address to the IP options. See section 3.1, RFC 791.

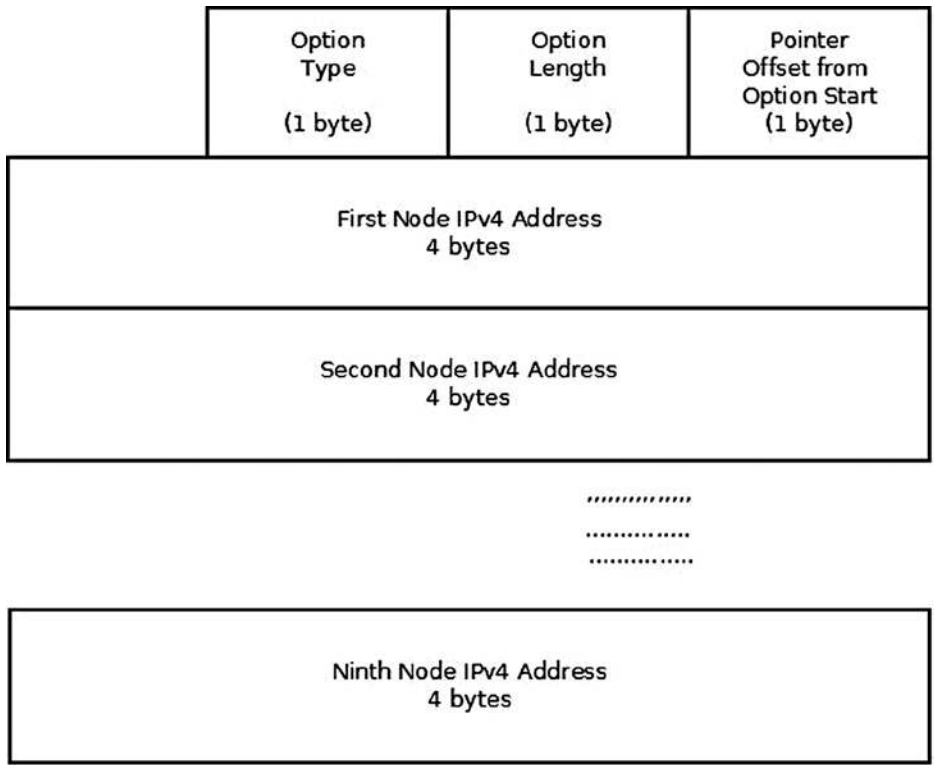
Option Type (1 byte)
Option Length (1 byte)
Pointer Offset from Option Start (1 byte)
First Node IPv4 Address 4 bytes
Second Node IPv4 Address 4 bytes
Ninth Node IPv4 Address 4 bytes

Figure 4-7. *Record Route option*

Though `ping -R` uses the Record Route IP Option, in many cases, if you will try it, you will not get the expected result of all the network nodes along the way, because for security reasons many network nodes ignore this IP option. The `manpage` of `ping` mentions this explicitly. From `man ping`:

```
. . .
-R
Includes the RECORD_ROUTE option in the ECHO_REQUEST packet and displays the route buffer on
returned packets.
. . .
Many hosts ignore or discard this option.
. . .
```

- *Stream ID (IPOPT_SID)*: This option provides a way for the 16-bit SATNET stream identifier to be carried through networks that do not support the stream concept.
- *Strict Source Record Route (IPOPT_SSRR)*: This option specifies a list of routers that the packet should traverse. The order should be kept, and no changes in traversal are permitted. Many routers block the Loose Source Record Route (LSRR) and Strict Source Record Route (SSRR) options because of security reasons.
- *Router Alert (IPOPT_RA)*: The IP Router Alert option can be used to notify transit routers to more closely examine the contents of an IP packet. This is useful, for example, for new protocols but requires relatively complex processing in routers along the path. Specified in RFC 2113, "IP Router Alert Option."

IP options are represented in Linux by the `ip_options` structure:

```
struct ip_options {
        __be32          faddr;
        __be32          nexthop;
        unsigned char   optlen;
        unsigned char   srr;
        unsigned char   rr;
        unsigned char   ts;
        unsigned char   is_strictroute:1,
        srr_is_hit:1,
        is_changed:1,
        rr_needaddr:1,
        ts_needtime:1,
        ts_needaddr:1;
        unsigned char   router_alert;
        unsigned char   cipso;
        unsigned char   __pad2;
        unsigned char   __data[0];
};
```

(include/net/inet_sock.h)

Here are short descriptions of the members of the IP options structure:

- `faddr`: Saved first hop address. Set in `ip_options_compile()` when handling loose and strict routing, when the method was not invoked from the Rx path (SKB is NULL).
- `nexthop`: Saved nexthop address in LSRR and SSRR.
- `optlen`: The option length, in bytes. Cannot exceed 40 bytes.
- `is_strictroute`: A flag specifing usage of strict source route. The flag is set in the `ip_options_compile()` method when parsing strict route option type (IPOPT_SSRR); note that it is not set for loose route (IPOPT_LSRR).
- `srr_is_hit`: A flag specifing that the packet destination `addr` was the local host The `srr_is_hit` flag is set in `ip_options_rcv_srr()`.
- `is_changed`: IP checksum is not valid anymore (the flag is set when one of the IP options is changed).
- `rr_needaddr`: Need to record IPv4 address of the outgoing device. The flag is set for the Record Route option (IPOPT_RR).
- `ts_needtime`: Need to record timestamp. The flag is set for these flags of the Timestamp IP Option: IPOPT_TS_TSONLY, IPOPT_TS_TSANDADDR and IPOPT_TS_PRESPEC (see a detailed explanation about the difference between these flags later in this section).
- `ts_needaddr`: Need to record IPv4 address of the outgoing device. This flag is set only when the IPOPT_TS_TSANDADDR flag is set, and it indicates that the IPv4 address of each node along the route of the packet should be added.
- `router_alert`: Set in the `ip_options_compile()` method when parsing a router alert option (`IPOPT_RR`).
- `__data[0]`: A buffer to store options that are received from userspace by `setsockopt()`.

See `ip_options_get_from_user()` and `ip_options_get_finish() (net/ipv4/ip_options.c)`.

Let's take a look at the ip_rcv_options() method:

```
static inline bool ip_rcv_options(struct sk_buff *skb)
{
         struct ip_options *opt;
         const struct iphdr *iph;
         struct net_device *dev = skb->dev;
        . . .
```

Fetch the IPv4 header from the SKB:

```
         iph = ip_hdr(skb);
```

Fetch the ip_options object from the inet_skb_parm object which is associated to the SKB:

```
         opt = &(IPCB(skb)->opt);
```

Calculate the expected options length:

```
         opt->optlen = iph->ihl*4 - sizeof(struct iphdr);
```

Call the ip_options_compile() method to build an ip_options object out of the SKB:

```
         if (ip_options_compile(dev_net(dev), opt, skb)) {
                 IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_INHDRERRORS);
                 goto drop;
         }
```

When the ip_options_compile() method is called in the Rx path (from the ip_rcv_options() method), it parses the IPv4 header of the specified SKB and builds an ip_options object out of it, according to the IPv4 header content, after verifying the validity of the options. The ip_options_compile() method can also be invoked from the ip_options_get_finish() method when getting options from userspace via the setsockopt() system call with IPPROTO_IP and IP_OPTIONS. In this case, data is copied from userspace into opt->data, and the third parameter for ip_options_compile(), the SKB, is NULL; the ip_options_compile() method builds the ip_options object in such a case from opt->__data. If some error is found while parsing the options, and it is in the Rx path (the ip_options_compile() method was invoked from ip_rcv_options()), a "Parameter Problem" ICMPv4 message (ICMP_PARAMETERPROB) is sent back. An error with the code -EINVAL is returned in case of error, regardless of how the method was invoked. Naturally, it is more convenient to work with the ip_options object than with the raw IPv4 header, because access to the IP options fields is much simpler this way. In the Rx path, the ip_options object that the ip_options_compile() method builds is stored in the control buffer (cb) of the SKB; this is done by setting the opt object to &(IPCB(skb)->opt). The IPCB(skb) macro is defined like this:

```
#define IPCB(skb) ((struct inet_skb_parm*)((skb)->cb))
```

And the inet_skb_parm structure (which includes an ip_options object) is defined like this:

```
struct inet_skb_parm {
        struct ip_options       opt;            /* Compiled IP options          */
        unsigned char           flags;
        u16                     frag_max_size;
};

(include/net/ip.h)
```

So &(IPCB(skb)->opt points to the ip_options object inside the inet_skb_parm object. I will not delve into all the small, tedious technical details of parsing the IPv4 header in the ip_options_compile() method in this book, because there is an abundance of such details and they are self-explanatory. I will discuss briefly how the ip_options_compile() parses some single byte options, like IPOPT_END and IPOPT_NOOP, and some more complex options like IPOPT_RR and IPOPT_TIMESTAMP in the Rx path and show some examples of which checks are done in this method and how it is implemented in the following code snippet:

```
int ip_options_compile(struct net *net, struct ip_options *opt, struct sk_buff *skb)
{

         ...
         unsigned char *pp_ptr = NULL;
         struct rtable *rt = NULL;
         unsigned char *optptr;
         unsigned char *iph;
         int optlen, l;
```

For starting the parsing process, the optptr pointer should point to the start of the IP options object and iterate over all the options in a loop. For the Rx path (when the ip_options_compile() method is invoked from the ip_rcv_options() method), the SKB that was received in the ip_rcv() method is passed as a parameter to ip_options_compile() and, needless to say, cannot be NULL. In such a case, the IP options start immediately after the initial fixed size (20 bytes) of the IPv4 header. When the ip_options_compile() was invoked from ip_options_get_finish(), the optptr pointer was set to opt->__data, because the ip_options_get_from_user() method copied the options that were sent from userspace into opt->__data. To be accurate, I should mention that if alignment is needed, the ip_options_get_finish() method also writes into opt->__data (it writes IPOPT_END in the proper place).

```
         if (skb != NULL) {
             rt = skb_rtable(skb);
             optptr = (unsigned char *)&(ip_hdr(skb)[1]);
         } else
             optptr = opt->__data;
```

In this case, iph = ip_hdr(skb) cannot be used instead, because the case when SKB is NULL should be considered. The following assignment is correct also for the non-Rx path:

```
        iph = optptr - sizeof(struct iphdr);
```

The variable l is initialized to be the options length (it can be 40 bytes at most). It is decremented by the length of the current option in each iteration of the following for loop:

```
        for (l = opt->optlen; l > 0; ) {
            switch (*optptr) {
```

If an IPOPT_END option is encountered, it indicates that this is the end of the options list—there must be no other option after it. In such a case you write IPOPT_END for each byte which is different than IPOPT_END until the end of the options list. The is_changed Boolean flag should also be set, because it indicates that the IPv4 header was changed (and as a result, recalculation of checksum is pending—there is no justification for calculating the checksum right now or inside the for loop, because there might be other changes in the IPv4 header during the loop):

```
                case IPOPT_END:
                  for (optptr++, l--; l>0; optptr++, l--) {
                     if (*optptr != IPOPT_END) {
                         *optptr = IPOPT_END;
                         opt->is_changed = 1;
                     }
                  }
         goto eol;
```

If an option type of No Operation (IPOPT_NOOP), which is a single byte option, is encountered, simply decrement `l` by 1, increment `optptr` by 1, and move forward to the next option type:

```
                case IPOPT_NOOP:
                  l--;
                  optptr++;
                  continue;
            }
```

`Optlen` is set to be the length of the option that is read (as `optptr[1]` holds the option length):

```
            optlen = optptr[1];
```

The No Operation (IPOPT_NOOP) option and the End of Option List (IPOPT_END) option are the only single byte options. All other options are multibyte options and must have at least two bytes (option type and option length). Now a check is made that there are at least two option bytes and the option list length was not exceeded. If there was some error, the `pp_ptr` pointer is set to point to the source of the problem and exit the loop. If it is in the Rx path, an ICMPv4 message of "Parameter Problem" is sent back, passing as a parameter the offset where the problem occurred, so that the other side can analyze the problem:

```
            if (optlen<2 || optlen>l) {
                pp_ptr = optptr;
                goto error;
            }
            switch (*optptr) {
                case IPOPT_SSRR:
                case IPOPT_LSRR:
                ...
                case IPOPT_RR:
```

The option length of the Record Route option must be at least 3 bytes: option type, option length, and pointer (offset):

```
                  if (optlen < 3) {
                      pp_ptr = optptr + 1;
                      goto error;
                  }
```

The option pointer offset of the Record Route option must be at least 4 bytes, since the space reserved for the address list must start after the three initial bytes (option type, option length, and pointer):

```
                if (optptr[2] < 4) {
                        pp_ptr = optptr + 2;
                        goto error;
                }
                if (optptr[2] <= optlen) {
```

If the offset (`optptr[2]`) plus the three initial bytes exceeds the option length, there is an error:

```
                    if (optptr[2]+3 > optlen) {
                         pp_ptr = optptr + 2;
                         goto error;
                    }
                    if (rt) {
                        spec_dst_fill(&spec_dst, skb);
```

Copy the IPv4 address to the Record Route buffer:

```
                        memcpy(&optptr[optptr[2]-1], &spec_dst, 4);
```

Set the `is_changed` Boolean flag, which indicates that the IPv4 header was changed (recalculation of checksum is pending):

```
                        opt->is_changed = 1;
                    }
```

Increment the pointer (offset) by 4 for the next address in the Record Route buffer (each IPv4 address is 4 bytes):

```
                    optptr[2] += 4;
```

Set the `rr_needaddr` flag (this flag is checked in the `ip_forward_options()` method):

```
                    opt->rr_needaddr = 1;
                }
                opt->rr = optptr - iph;
                break;

                     case IPOPT_TIMESTAMP:
                       ...
```

The option length for Timestamp option must be at least 4 bytes: option type, option length, pointer (offset), and the fourth byte is divided into two fields: the higher 4 bits are the overflow counter, which is incremented in each hop where there is no available space to store the required data, and the lower 4 bits are the flag: timestamp only, timestamp and address, and timestamp by a specified hop:

```
                       if (optlen < 4) {
                             pp_ptr = optptr + 1;
                             goto error;
                       }
```

`optptr[2]` is the pointer (offset). Because, as stated earlier, each Timestamp option starts with 4 bytes, it implies that the pointer (offset) must be at least 5:

```
                if (optptr[2] < 5) {
                        pp_ptr = optptr + 2;
                        goto error;
                }
                if (optptr[2] <= optlen) {
                        unsigned char *timeptr = NULL;
                        if (optptr[2]+3 > optptr[1]) {
                                pp_ptr = optptr + 2;
                                goto error;
                        }
```

In the switch command, the value of `optptr[3]&0xF` is checked. It is the flag (4 lower bits of the fourth byte) of the Timestamp option:

```
                        switch (optptr[3]&0xF) {
                              case IPOPT_TS_TSONLY:
                                if (skb)
                                        timeptr = &optptr[optptr[2]-1];
                                opt->ts_needtime = 1;
```

For the Timestamp option with timestamps only flag (IPOPT_TS_TSONLY), 4 bytes are needed; so the pointer (offset) is incremented by 4:

```
                                optptr[2] += 4;
                                break;

                              case IPOPT_TS_TSANDADDR:
                                if (optptr[2]+7 > optptr[1]) {
                                        pp_ptr = optptr + 2;
                                        goto error;
                                }
                                if (rt)  {
                                         spec_dst_fill(&spec_dst, skb);
                                         memcpy(&optptr[optptr[2]-1],
                                                &spec_dst, 4);
                                         timeptr = &optptr[optptr[2]+3];
                                }
                                opt->ts_needaddr = 1;
                                opt->ts_needtime = 1;
```

For the Timestamp option with timestamps and addresses flag (IPOPT_TS_TSANDADDR), 8 bytes are needed; so the pointer (offset) is incremented by 8:

```
                                optptr[2] += 8;
                                break;

                              case IPOPT_TS_PRESPEC:
                                if (optptr[2]+7 > optptr[1]) {
                                        pp_ptr = optptr + 2;
                                        goto error;
                                }
                                {
                                 __be32 addr;
                                 memcpy(&addr, &optptr[optptr[2]-1], 4);
                                    if (inet_addr_type(net,addr) == RTN_UNICAST)
                                       break;
                                 if (skb)
                                      timeptr = &optptr[optptr[2]+3];
                                }
                                opt->ts_needtime = 1;
```

For the Timestamp option with timestamps and pre-specified hops flag (IPOPT_TS_PRESPEC), 8 bytes are needed, so the pointer (offset) is incremented by 8:

```
                                optptr[2] += 8;
                                break;
                              default:
                                ...
                         }
                   ...
```

After the `ip_options_compile()` method has built the `ip_options` object, strict routing is handled. First, a check is performed to see whether the device supports source routing. This means that the `/proc/sys/net/ipv4/conf/all/accept_source_route` is set, and the `/proc/sys/net/ipv4/conf/<deviceName>/accept_source_route` is set. If these conditions are not met, the packet is dropped:

```
        . . .
    if (unlikely(opt->srr)) {
        struct in_device *in_dev = __in_dev_get_rcu(dev);

        if (in_dev) {
                if (!IN_DEV_SOURCE_ROUTE(in_dev)) {
                . . .
                            goto drop;
                }
        }

        if (ip_options_rcv_srr(skb))
                goto drop;
    }
```

Let's take a look at the `ip_options_rcv_srr()` method (again, I will focus on the important points, not little details). The list of source route addresses is iterated over. During the parsing process some sanity checks are made in the loop to see if there are errors. When the first nonlocal address is encountered, the loop is exited, and the following actions take place:

- Set the `srr_is_hit` flag of the IP option object (`opt->srr_is_hit = 1`).
- Set `opt->nexthop` to be the nexthop address that was found.
- Set the `opt->is_changed` flag to 1.

The packet should be forwarded. When the method `ip_forward_finish()` is reached, the `ip_forward_options()` method is called. In this method, if the `srr_is_hit` flag of the IP option object is set, the `daddr` of the ipv4 header is changed to be `opt->nexthop`, the offset is incremented by 4 (to point to the next address in the source route addresses list), and—because the IPv4 header was changed—the checksum is recalculated by calling the `ip_send_check()` method.

IP Options and Fragmentation

When describing the option type in the beginning of this section, I mentioned a `copied` flag in the option type byte which indicates whether or not to copy the option when forwarding a fragmented packet. Handling IP options in fragmentation is done by the `ip_options_fragment()` method, which is invoked from the method which prepares fragments, `ip_fragment()`. It is called only for the first fragment. Let's take a look at the `ip_options_fragment()` method, which is very simple:

```
void ip_options_fragment(struct sk_buff *skb)
{
        unsigned char *optptr = skb_network_header(skb) + sizeof(struct iphdr);
        struct ip_options *opt = &(IPCB(skb)->opt);
        int  l = opt->optlen;
        int  optlen;
```

The `while` loop simply iterates over the options, reading each option type. `optptr` is a pointer to the option list (which starts at the end of the 20 first bytes of the IPv4 header). `l` is the size of the option list, which is being decremented by 1 in each loop iteration:

```
        while (l > 0) {
                switch (*optptr) {
```

When the option type is IPOPT_END, which terminates the option string, it means that reading the options is finished:

```
                case IPOPT_END:
                        return;

                case IPOPT_NOOP:
```

When the `option type` is IPOPT_NOOP, used for padding between options, the `optptr` pointer is incremented by 1, `l` is decremented, and the next option is processed:

```
                        l--;
                        optptr++;
                        continue;
                }
```

Perform a sanity check on the option length:

```
                optlen = optptr[1];
                if (optlen<2 || optlen>l)
                  return;
```

Check whether the option should be copied; if not, simply put one or several IPOPT_NOOP options instead of it with the `memset()` function. The number of IPOPT_NOOP bytes that `memset()` writes is the size of the option that was read, namely `optlen`:

```
                if (!IPOPT_COPIED(*optptr))
                        memset(optptr, IPOPT_NOOP, optlen);
```

Now go to the next option:

```
                l -= optlen;
                optptr += optlen;        }
```

IPOPT_TIMESTAMP and IPOPT_RR are options for which the `copied` flag is 0 (see Table 4-1). They are replaced by IPOPT_NOOP in the loop you saw earlier, and their relevant fields in the IP option object are reset to 0:

```
        opt->ts = 0;
        opt->rr = 0;
        opt->rr_needaddr = 0;
        opt->ts_needaddr = 0;
        opt->ts_needtime = 0;
}
```

(net/ipv4/ip_options.c)

In this section you have learned how the `ip_rcv_options()` handles the reception of packets with IP options and how IP options are parsed by the `ip_options_compile()` method. Fragmentation in IP options was also discussed. The next section covers the process of building IPv4 options, which involves setting the IP options of an IPv4 header based on a specified `ip_options` object.

Building IP Options

The `ip_options_build()` method can be thought of as the reverse of the `ip_options_compile()` method you saw earlier in this chapter. It takes an `ip_options` object as an argument and writes its content to the IPv4 header. Let's take a look at it:

```
void ip_options_build(struct sk_buff *skb, struct ip_options *opt,
                      __be32 daddr, struct rtable *rt, int is_frag)
{
        unsigned char *iph = skb_network_header(skb);

        memcpy(&(IPCB(skb)->opt), opt, sizeof(struct ip_options));
        memcpy(iph+sizeof(struct iphdr), opt->__data, opt->optlen);
        opt = &(IPCB(skb)->opt);
        if (opt->srr)
                memcpy(iph+opt->srr+iph[opt->srr+1]-4, &daddr, 4);

        if (!is_frag) {
                if (opt->rr_needaddr)
                        ip_rt_get_source(iph+opt->rr+iph[opt->rr+2]-5, skb, rt);
                if (opt->ts_needaddr)
                        ip_rt_get_source(iph+opt->ts+iph[opt->ts+2]-9, skb, rt);
                if (opt->ts_needtime) {
                        struct timespec tv;
                        __be32 midtime;
                        getnstimeofday(&tv);
                        midtime = htonl((tv.tv_sec % 86400) *
                                         MSEC_PER_SEC + tv.tv_nsec / NSEC_PER_MSEC);
                        memcpy(iph+opt->ts+iph[opt->ts+2]-5, &midtime, 4);
                }
                return;
        }
        if (opt->rr) {
                memset(iph+opt->rr, IPOPT_NOP, iph[opt->rr+1]);
                opt->rr = 0;
                opt->rr_needaddr = 0;
        }
        if (opt->ts) {
                memset(iph+opt->ts, IPOPT_NOP, iph[opt->ts+1]);
                opt->ts = 0;
                opt->ts_needaddr = opt->ts_needtime = 0;
        }
}
```

The `ip_forward_options()` method handles forwarding fragmented packets (`net/ipv4/ip_options.c`). In this method the Record Route and Strict Record route options are handled, and the `ip_send_check()` method is invoked to calculate the checksum for packets whose IPv4 header was changed (the `opt->is_changed` flag is set) and to reset the `opt->is_changed` flag to 0. The IPv4 Tx path—namely, how packets are sent—is discussed in the next section.

My discussion on the Rx path is finished. The next section talks about the Tx path—what happens when IPv4 packets are sent.

Sending IPv4 Packets

The IPv4 layer provides the means for the layer above it, the transport layer (L4), to send packets by passing these packets to the link layer (L2). I discuss how that is implemented in this section, and you'll see some differences between handling transmission of TCPv4 packets in IPv4 and handling transmission of UDPv4 packets in IPv4. There are two main methods for sending IPv4 packets from Layer 4, the transport layer: The first one is the `ip_queue_xmit()` method, used by the transport protocols that handle fragmentation by themselves, like TCPv4. The `ip_queue_xmit()` method is not the only transmission method used by TCPv4, which uses also the `ip_build_and_send_pkt()` method, for example, to send SYN ACK messages (see the tcp_v4_send_synack() method implementation in net/ipv4/tcp_ipv4.c). The second method is the ip_append_data() method, used by the transport protocols that do not handle fragmentation, like the UDPv4 protocol or the ICMPv4 protocol. The ip_append_data() method does not send any packet—it only prepares the packet. The ip_push_pending_frames() method is for actually sending the packet, and it is used by ICMPv4 or raw sockets, for example. Calling ip_push_pending_frames() actually starts the transmission process by calling the ip_send_skb() method, which eventually calls the ip_local_out() method. The ip_push_pending_frames() method was used for carrying out the transmission in UDPv4 prior to kernel 2.6.39; with the new ip_finish_skb API in 2.6.39, the ip_send_skb() method is used instead. Both methods are implemented in net/ipv4/ip_output.c.

There are cases where the dst_output() method is called directly, without using the ip_queue_xmit() method or the ip_append_data() method; for example, when sending with a raw socket which uses IP_HDRINCL socket option, there is no need to prepare an IPv4 header. Userspace applications that build an IPv4 by their own use the IPv4 IP_HDRINCL socket option. For example, the well-known ping of iputils and nping of nmap both enable the user to set the ttl of the IPv4 header like this:

```
ping -ttl ipDestAddress
```

or:

```
nping -ttl ipDestAddress
```

Sending packets by raw sockets whose IP_HDRINCL socket option is set is done like this:

```
static int raw_send_hdrinc(struct sock *sk, struct flowi4 *fl4,
               void *from, size_t length,
               struct rtable **rtp,
               unsigned int flags)
{
        ...
        err = NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_OUT, skb, NULL,
              rt->dst.dev, dst_output);
        ...
}
```

Figure 4-8 shows the paths for sending IPv4 packets from the transport layer.

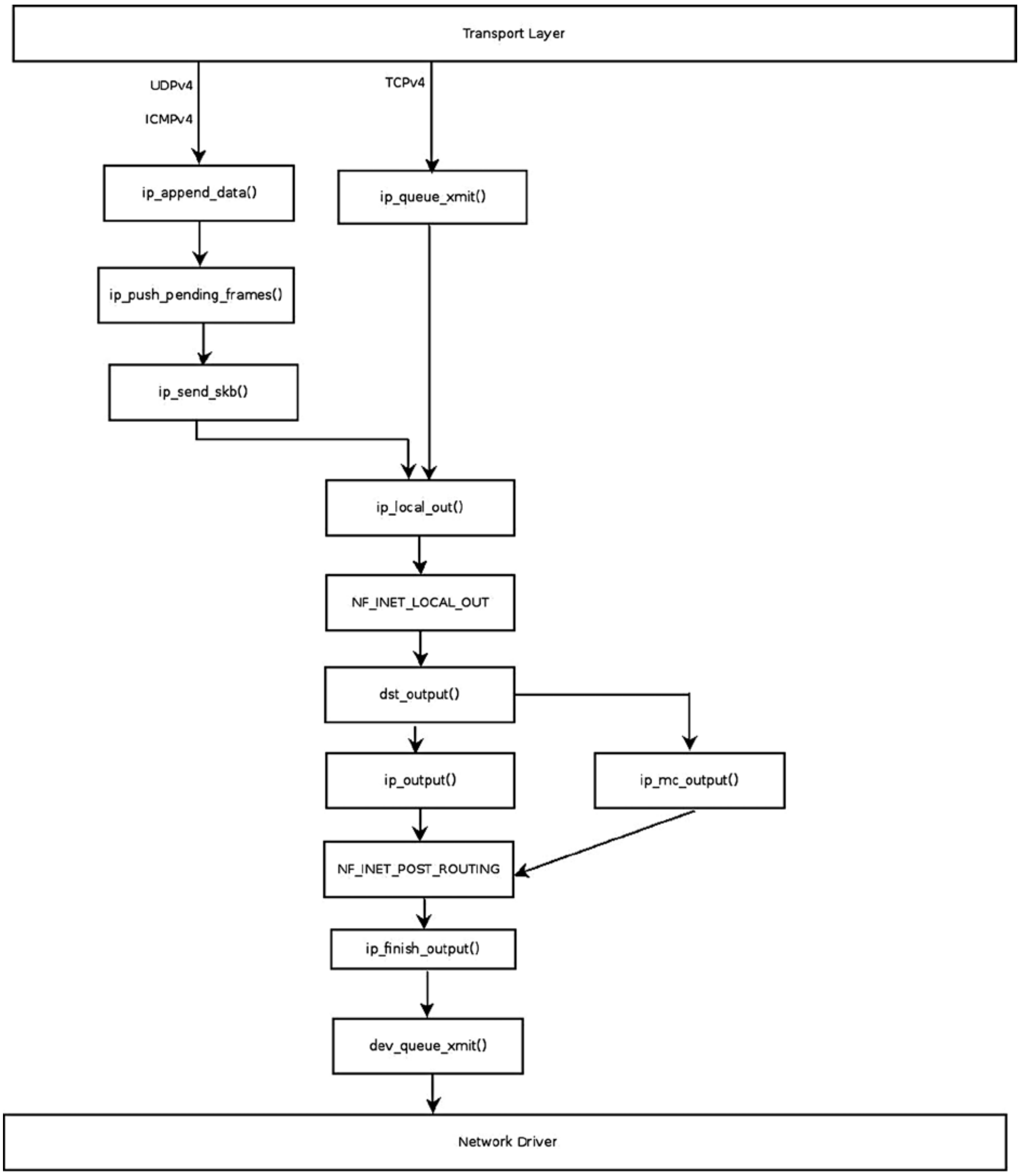

Figure 4-8. *Sending IPv4 packets*

In figure 4-8 you can see the different paths for transmitted packets that come from the transport layer (L4); these packets are handled by the `ip_queue_xmit()` method or by the `ip_append_data()` method.

Let's start with the `ip_queue_xmit()` method, which is the simpler method of the two:

```
int ip_queue_xmit(struct sk_buff *skb, struct flowi *fl)
    . . .
    /* Make sure we can route this packet. */
    rt = (struct rtable *)__sk_dst_check(sk, 0);
```

The `rtable` object is the result of a lookup in the routing subsystem. First I discuss the case where the `rtable` instance is NULL and you need to perform a lookup in the routing subsystem. If the strict routing option flag is set, the destination address is set to be the first address of the IP options:

```
    if (rt == NULL) {
        __be32 daddr;

        /* Use correct destination address if we have options. */
        daddr = inet->inet_daddr;
        if (inet_opt && inet_opt->opt.srr)
            daddr = inet_opt->opt.faddr;
```

Now a lookup in the routing subsystem is performed with the `ip_route_output_ports()` method: if the lookup fails, the packet is dropped, and an error of –EHOSTUNREACH is returned:

```
        /* If this fails, retransmit mechanism of transport layer will
         * keep trying until route appears or the connection times
         * itself out.
         */
        rt = ip_route_output_ports(sock_net(sk), fl4, sk,
                       daddr, inet->inet_saddr,
                       inet->inet_dport,
                       inet->inet_sport,
                       sk->sk_protocol,
                       RT_CONN_FLAGS(sk),
                       sk->sk_bound_dev_if);
        if (IS_ERR(rt))
            goto no_route;
        sk_setup_caps(sk, &rt->dst);
    }
    skb_dst_set_noref(skb, &rt->dst);
    . . .
```

If the lookup succeeds, but both the `is_strictroute` flag in the options and the `rt_uses_gateway` flag in the routing entry are set, the packet is dropped, and an error of –EHOSTUNREACH is returned:

```
    if (inet_opt && inet_opt->opt.is_strictroute && rt->rt_uses_gateway)
        goto no_route;
```

Now the IPv4 header is being built. You should remember that the packet arrived from Layer 4, where skb->data pointed to the transport header. The skb->data pointer is moved back by the skb_push() method; the offset needed to move it back is the size of the IPv4 header plus the size of the IP options list (optlen), if IP options are used:

```
/* OK, we know where to send it, allocate and build IP header. */
skb_push(skb, sizeof(struct iphdr) + (inet_opt ? inet_opt->opt.optlen : 0));
```

Set the L3 header (skb->network_header) to point to skb->data:

```
skb_reset_network_header(skb);
iph = ip_hdr(skb);
*((__be16 *)iph) = htons((4 << 12) | (5 << 8) | (inet->tos & 0xff));
if (ip_dont_fragment(sk, &rt->dst) && !skb->local_df)
    iph->frag_off = htons(IP_DF);
else
    iph->frag_off = 0;
iph->ttl      = ip_select_ttl(inet, &rt->dst);
iph->protocol = sk->sk_protocol;
ip_copy_addrs(iph, fl4);
```

The options length (optlen) is divided by 4, and the result is added to the IPv4 header length (iph->ihl) because the IPv4 header is measured in multiples of 4 bytes. Then the ip_options_build() method is invoked to build the options in the IPv4 header based on the content of the specified IP options. The last parameter of the ip_options_build() method, is_frag, specifies that there are no fragments. The ip_options_build() method was discussed in the "IP Option" section earlier in this chapter.

```
if (inet_opt && inet_opt->opt.optlen) {
iph->ihl += inet_opt->opt.optlen >> 2;
ip_options_build(skb, &inet_opt->opt, inet->inet_daddr, rt, 0);
}
```

Set the id in the IPv4 header:

```
ip_select_ident_more(iph, &rt->dst, sk,
         (skb_shinfo(skb)->gso_segs ?: 1) - 1);

skb->priority = sk->sk_priority;
skb->mark = sk->sk_mark;
```

Send the packet:

```
res = ip_local_out(skb);
```

Before discussing the ip_append_data() method, I want to mention a callback which is a parameter to the ip_append_data() method: the getfrag() callback. The getfrag() method is a callback to copy the actual data from userspace into the SKB. In UDPv4, the getfrag() callback is set to be the generic method, ip_generic_getfrag(). In ICMPv4, the getfrag() callback is set to be a protocol-specific method, icmp_glue_bits(). Another issue I should mention here is the UDPv4 corking feature. The UDP_CORK socket option was added in kernel 2.5.44; when this option is enabled, all data output on this socket is accumulated into a single datagram that is transmitted when the option is disabled. You can enable and disable this socket option with the setsockopt() system call; see man 7 udp. In kernel 2.6.39, a lockless transmit fast path was added to the UDPv4 implementation. With this addition, when the corking feature is not used, the socket lock is not used. So when the UDP_CORK socket option is set (with the `setsockopt()` system call), or the MSG_MORE flag is set, the `ip_append_data()` method is invoked. And when the UDP_CORK socket option is not set, another path in the `udp_sendmsg()` method is used, which does not hold the socket lock and is faster as a result, and the `ip_make_skb()` method is invoked. Calling the `ip_make_skb()` method is similar to the `ip_append_data()` and the `ip_push_pending_frames()` methods rolled into one, except that it does not send the SKB produced. Sending the SKB is carried out by the `ip_send_skb()` method.

Let's take a look now at the `ip_append_data()` method:

```
int ip_append_data(struct sock *sk, struct flowi4 *fl4,
                   int getfrag(void *from, char *to, int offset, int len,
                               int odd, struct sk_buff *skb),
                   void *from, int length, int transhdrlen,
                   struct ipcm_cookie *ipc, struct rtable **rtp,
                   unsigned int flags)
{
        struct inet_sock *inet = inet_sk(sk);
        int err;
```

If the MSG_PROBE flag us used, it means that the caller is interested only in some information (usually MTU, for PMTU discovery), so there is no need to actually send the packet, and the method returns 0:

```
        if (flags&MSG_PROBE)
                return 0;
```

The value of `transhdrlen` is used to indicate whether it is a first fragment or not. The `ip_setup_cork()` method creates a cork IP options object if it does not exist and copies the IP options of the specified `ipc` (`ipcm_cookie` object) to the cork IP options:

```
        if (skb_queue_empty(&sk->sk_write_queue)) {
                err = ip_setup_cork(sk, &inet->cork.base, ipc, rtp);
                if (err)
                        return err;
        } else {
                transhdrlen = 0;
        }
```

The real work is done by the `__ip_append_data()` method; this is a long and a complex method, and I can't delve into all its details. I will mention that there are two different ways to handle fragments in this method, according to whether the network device supports Scatter/Gather (NETIF_F_SG) or not. When the NETIF_F_SG flag is set, `skb_shinfo(skb)->frags` is used, whereas when the NETIF_F_SG flag is not set, `skb_shinfo(skb)->frag_list` is used. There is also a different memory allocation when the MSG_MORE flag is set. The MSG_MORE flag indicates that soon another packet will be sent. Since Linux 2.6, this flag is also supported for UDP sockets.

```
        return __ip_append_data(sk, fl4, &sk->sk_write_queue, &inet->cork.base,
                                sk_page_frag(sk), getfrag,
                                from, length, transhdrlen, flags);
}
```

In this section you have learned about the Tx path—how sending IPv4 packets is implemented. When the packet length is higher than the network device MTU, the packet can't be sent as is. The next section covers fragmentation in the Tx path and how it is handled.

Fragmentation

The network interface has a limit on the size of a packet. Usually in 10/100/1000 Mb/s Ethernet networks, it is 1500 bytes, though there are network interfaces that allow using an MTU of up to 9K (called *jumbo frames*). When sending a packet that is larger than the MTU of the outgoing network card, it should be broken into smaller pieces. This is done within the `ip_fragment()` method (`net/ipv4/ip_output.c`). Received fragmented packets should be reassembled into one packet. This is done by the `ip_defrag()` method, (`net/ipv4/ip_fragment.c`), discussed in the next section, "Defragmentation."

Let's take a look first at the `ip_fragment()` method. Here's its prototype:

```
int ip_fragment(struct sk_buff *skb, int (*output)(struct sk_buff *))
```

The `output` callback is the method of transmission to be used. When the `ip_fragment()` method is invoked from `ip_finish_output()`, the `output` callback is the `ip_finish_output2()` method. There are two paths in the `ip_fragment()` method: the fast path and the slow path. The fast path is for packets where the `frag_list` of the SKB is not NULL, and the slow path is for packets that do not meet this condition.

First a check is performed to see whether fragmentation is permitted, and if not, a "Destination Unreachable" ICMPv4 message with code of fragmentation needed is sent back to the sender, the statistics (IPSTATS_MIB_FRAGFAILS) are updated, the packet is dropped, and an error code of –EMSGSIZE is returned:

```
int ip_fragment(struct sk_buff *skb, int (*output)(struct sk_buff *))
        {
        unsigned int mtu, hlen, left, len, ll_rs;
        . . .
        struct rtable *rt = skb_rtable(skb);
        int err = 0;

        dev = rt->dst.dev;

        . . .

        iph = ip_hdr(skb);

        if (unlikely(((iph->frag_off & htons(IP_DF)) && !skb->local_df) ||
              (IPCB(skb)->frag_max_size &&
               IPCB(skb)->frag_max_size > dst_mtu(&rt->dst)))) {
           IP_INC_STATS(dev_net(dev), IPSTATS_MIB_FRAGFAILS);
           icmp_send(skb, ICMP_DEST_UNREACH, ICMP_FRAG_NEEDED,
                     htonl(ip_skb_dst_mtu(skb)));
           kfree_skb(skb);
           return -EMSGSIZE;
        }
        . . .
        . . .
```

The next section discusses the fast path in fragmentation and its implementation.

Fast Path

Now let's look into the fast path. First a check is performed to see whether the packet should be handled in the fast path by calling the `skb_has_frag_list()` method, which simply checks that `skb_shinfo(skb)->frag_list` is not NULL; if it is NULL, some sanity checks are made, and if something is not valid, the fallback to the slow path mechanism is activated (simply by calling `goto slow_path`). Then an IPv4 header is built for the first fragment. The `frag_off` of this IPv4 header is set to be `htons(IP_MF)`, which indicates more fragments ahead. The `frag_off` field of the IPv4 header is a 16-bit field; the lower 13 bits are the fragment offset, and the higher 3 bits are the flags. For the first fragment, the offset should be 0, and the flag should be IP_MF (More Fragments). For all other fragments except the last one, the IP_MF flag should be set, and the lower 13 bits should be the fragment offset (measured in units of 8 bytes). For the last fragment, the IP_MF flag should not be set, but the lower 13 bits will still hold the fragment offset.

Here's how to set `hlen` to the IPv4 header size in bytes:

```
hlen = iph->ihl * 4;
. . .
if (skb_has_frag_list(skb)) {
    struct sk_buff *frag, *frag2;
    int first_len = skb_pagelen(skb);
    . . .
    err     = 0;
    offset = 0;
    frag = skb_shinfo(skb)->frag_list;
```

set `skb_shinfo(skb)->frag_list` to NULL by `skb_frag_list_init(skb)`:

```
    skb_frag_list_init(skb);
    skb->data_len = first_len - skb_headlen(skb);
    skb->len = first_len;
    iph->tot_len = htons(first_len);
```

Set the IP_MF (More Fragments) flag for the first fragment:

```
    iph->frag_off = htons(IP_MF);
```

Because the value of some IPv4 header fields were changed, the checksum needs to be recalculated:

```
    ip_send_check(iph);
```

Now take a look at the loop that traverses `frag_list` and builds fragments:

```
    for (;;) {
       /* Prepare header of the next frame,
        * before previous one went down. */
       if (frag) {
         frag->ip_summed = CHECKSUM_NONE;
         skb_reset_transport_header(frag);
```

The `ip_fragment()` was invoked from the transport layer (L4), so `skb->data` points to the transport header. The `skb->data` pointer should be moved back by `hlen` bytes so that it will point to the IPv4 header (`hlen` is the size of the IPv4 header in bytes):

```
         __skb_push(frag, hlen);
```

Set the L3 header (`skb->network_header`) to point to `skb->data`:

```
            skb_reset_network_header(frag);
```

Copy the IPv4 header which was created into the L3 network header; in the first iteration of this `for` loop, it is the header which was created outside the loop for the first fragment:

```
            memcpy(skb_network_header(frag), iph, hlen);
```

Now the IPv4 header and its `tot_len` of the next frag are initialized:

```
            iph = ip_hdr(frag);
            iph->tot_len = htons(frag->len);
```

Copy various SKB fields (like `pkt_type`, `priority`, `protocol`) from SKB into `frag`:

```
            ip_copy_metadata(frag, skb);
```

Only for the first fragment (where the offset is 0) should the `ip_options_fragment()` method be called:

```
            if (offset == 0)
                ip_options_fragment(frag);
            offset += skb->len - hlen;
```

The `frag_off` field of the IPv4 header is measured in multiples of 8 bytes, so divide the offset by 8:

```
            iph->frag_off = htons(offset>>3);
```

Each fragment, except the last one, should have the IP_MF flag set:

```
            if (frag->next != NULL)
                iph->frag_off |= htons(IP_MF);
```

The value of some IPv4 header fields were changed, so the checksum should be recalculated:

```
            /* Ready, complete checksum */
            ip_send_check(iph);
          }
```

Now send the fragment with the `output` callback. If sending it succeeded, increment IPSTATS_MIB_FRAGCREATES. If there was an error, exit the loop:

```
          err = output(skb);

          if (!err)
             IP_INC_STATS(dev_net(dev), IPSTATS_MIB_FRAGCREATES);
          if (err || !frag)
             break;
```

Fetch the next SKB:

```
        skb = frag;
        frag = skb->next;
        skb->next = NULL;
```

The following closing bracket is the end of the for loop:

```
    }
```

The for loop is terminated, and the return value of the last call to output(skb) should be checked. If it is successful, the statistics (IPSTATS_MIB_FRAGOKS) are updated, and the method returns 0:

```
if (err == 0) {
     IP_INC_STATS(dev_net(dev), IPSTATS_MIB_FRAGOKS);
     return 0;
 }
```

If the last call to output(skb) failed in one of the loop iterations, including the last one, the SKBs are freed, the statistics (IPSTATS_MIB_FRAGFAILS) are updated, and the error code (err) is returned:

```
while (frag) {
     skb = frag->next;
     kfree_skb(frag);
     frag = skb;
 }
 IP_INC_STATS(dev_net(dev), IPSTATS_MIB_FRAGFAILS);
 return err;
```

You should now have a good understanding of the fast path in fragmentation and how it is implemented.

Slow Path

Let's now take a look at how to implement the slow path in fragmentation:

```
    . . .

    iph = ip_hdr(skb);

    left = skb->len - hlen;         /* Space per frame */
    . . .

    while (left > 0) {
            len = left;
            /* IF: it doesn't fit, use 'mtu' - the data space left */
            if (len > mtu)
                    len = mtu;
```

Each fragment (except the last one) should be aligned on a 8-byte boundary:

```
            if (len < left) {
                    len &= ~7;
            }
```

Allocate an SKB:

```
            if ((skb2 = alloc_skb(len+hlen+ll_rs, GFP_ATOMIC)) == NULL) {
                    NETDEBUG(KERN_INFO "IP: frag: no memory for new fragment!\n");
                    err = -ENOMEM;
                    goto fail;
            }

            /*
             *      Set up data on packet
             */
```

Copy various SKB fields (like pkt_type, priority, protocol) from skb into skb2:

```
            ip_copy_metadata(skb2, skb);
            skb_reserve(skb2, ll_rs);
            skb_put(skb2, len + hlen);
            skb_reset_network_header(skb2);
            skb2->transport_header = skb2->network_header + hlen;

            /*
             *      Charge the memory for the fragment to any owner
             *      it might possess
             */

            if (skb->sk)
                    skb_set_owner_w(skb2, skb->sk);

            /*
             *      Copy the packet header into the new buffer.
             */

            skb_copy_from_linear_data(skb, skb_network_header(skb2), hlen);

            /*
             *      Copy a block of the IP datagram.
             */
            if (skb_copy_bits(skb, ptr, skb_transport_header(skb2), len))
                    BUG();
            left -= len;

            /*
             *      Fill in the new header fields.
             */
            iph = ip_hdr(skb2);
```

frag_off is measured in multiples of 8 bytes, so divide the offset by 8:

```
            iph->frag_off = htons((offset >> 3));
            . . .
```

Handle options only once for the first fragment:

```
            if (offset == 0)
                    ip_options_fragment(skb);
```

The MF flag (More Fragments) should be set on any fragment but the last:

```
            if (left > 0 || not_last_frag)
                    iph->frag_off |= htons(IP_MF);
            ptr += len;
            offset += len;

            /*
             *      Put this fragment into the sending queue.
             */
            iph->tot_len = htons(len + hlen);
```

Because the value of some IPv4 header fields were changed, the checksum should be recalculated:

```
            ip_send_check(iph);
```

Now send the fragment with the output callback. If sending it succeeded, increment IPSTATS_MIB_FRAGCREATES. If there was an error, then free the packet, update the statistics (IPSTATS_MIB_FRAGFAILS), and return the error code:

```
            err = output(skb2);
            if (err)
                    goto fail;

            IP_INC_STATS(dev_net(dev), IPSTATS_MIB_FRAGCREATES);
    }
```

Now the while (left > 0) loop has terminated, and the consume_skb() method is invoked to free the SKB, the statistics (IPSTATS_MIB_FRAGOKS) are updated, and the value of err is returned:

```
    consume_skb(skb);
    IP_INC_STATS(dev_net(dev), IPSTATS_MIB_FRAGOKS);
    return err;
```

This section dealt with the implementation of slow path in fragmentation, and this ends the discussion of fragmentation in the Tx path. Remember that received fragmented packets, which are received on a host, should be reconstructed again so that applications can handle the original packet. The next section discusses defragmentation—the opposite of fragmentation.

Defragmentation

Defragmentation is the process of reassembling all the fragments of a packet, which all have the same id in the IPv4 header, into one buffer. The main method that handles defragmentation in the Rx path is ip_defrag() (net/ipv4/ip_fragment.c), which is called from ip_local_deliver(). There are other places where defragmentation might be needed, such as in firewalls, where the content of the packet should be known in order to be able to inspect it. In the ip_local_deliver() method, the ip_is_fragment() method is invoked to check whether the packet is fragmented; if it is, the ip_defrag() method is invoked. The ip_defrag() method has two arguments: the first is the SKB and the second is a 32-bit field which indicates the point where the method was invoked. Its value can be the following:

- IP_DEFRAG_LOCAL_DELIVER when it was called from ip_local_deliver().
- IP_DEFRAG_CALL_RA_CHAIN when it was called from ip_call_ra_chain().
- IP_DEFRAG_VS_IN or IP_DEFRAG_VS_FWD or IP_DEFRAG_VS_OUT when it was called from IPVS.

For a full list of possible values for the second argument of ip_defrag(), look in the ip_defrag_users enum definition in include/net/ip.h.

Let's look at the ip_defrag() invocation in ip_local_deliver():

```
int ip_local_deliver(struct sk_buff *skb)
{
    /*
     *    Reassemble IP fragments.
     */

    if (ip_is_fragment(ip_hdr(skb))) {
        if (ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER))
            return 0;
    }

    return NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_IN, skb, skb->dev, NULL,
               ip_local_deliver_finish);
}
```

(net/ipv4/ip_input.c)

The ip_is_fragment() is a simple helper method that takes as a sole argument the IPv4 header and returns true when it is a fragment, like this:

```
static inline bool ip_is_fragment(const struct iphdr *iph)
{
         return (iph->frag_off & htons(IP_MF | IP_OFFSET)) != 0;
}
```

(include/net/ip.h)

The ip_is_fragment() method returns true in either of two cases (or both):

- The IP_MF flag is set.
- The fragment offset is not 0.

Thus it will return true on all fragments:

- On the first fragment, where frag_off is 0 but the IP_MF flag is set.
- On the last fragment, where frag_off is not 0 but the IP_MF flag is not set.
- On all other fragments, where frag_off is not 0 and the IP_MF flag is set.

The implementation of defragmentation is based on a hash table of ipq objects. The hash function (ipqhashfn) has four arguments: fragment id, source address, destination address, and protocol:

```
struct ipq {
        struct inet_frag_queue q;

        u32             user;
        __be32          saddr;
        __be32          daddr;
        __be16          id;
        u8              protocol;
        u8              ecn; /* RFC3168 support */
        int             iif;
        unsigned int    rid;
        struct inet_peer    *peer;
};
```

Note that the logic of IPv4 defragmentation is shared with its IPv6 counterpart. So, for example, the inet_frag_queue structure and methods like the inet_frag_find() method and the inet_frag_evictor() method are not specific to IPv4; they are also used in IPv6 (see net/ipv6/reassembly.c and net/ipv6/nf_conntrack_reasm.c).

The ip_defrag() method is quite short. First it makes sure there is enough memory by calling the ip_evictor() method. Then it tries to find an ipq for the SKB by calling the ip_find() method; if it does not find one, it creates an ipq object. The ipq object that the ip_find() method returns is assigned to a variable named qp (a pointer to an ipq object). Then it calls the ip_frag_queue() method to add the fragment to a linked list of fragments (qp->q.fragments). The addition to the list is done according to the fragment offset, because the list is sorted by the fragment offset. After all fragments of an SKB were added, the ip_frag_queue() method calls the ip_frag_reasm() method to build a new packet from all its fragments. The ip_frag_reasm() method also stops the timer (of ip_expire()) by calling the ipq_kill() method. If there was some error, and the size of the new packet exceeds the highest permitted size (which is 65535), the ip_frag_reasm() method updates the statistics (IPSTATS_MIB_REASMFAILS) and returns -E2BIG. If the call to skb_clone() method in ip_frag_reasm() fails, it returns –ENOMEM. The IPSTATS_MIB_REASMFAILS statistics is updated in this case as well. Constructing a packet from all its fragments should be done in a specified time interval. If it's not completed within that interval, the ip_expire() method will send an ICMPv4 message of "Time Exceeded" with "Fragment Reassembly Time Exceeded" code. The defragmentation time interval can be set by the following procfs entry: /proc/sys/net/ipv4/ipfrag_time. It is 30 seconds by default.

Let's take a look at the ip_defrag() method:

```
int ip_defrag(struct sk_buff *skb, u32 user)
{
        struct ipq *qp;
        struct net *net;

        net = skb->dev ? dev_net(skb->dev) : dev_net(skb_dst(skb)->dev);
        IP_INC_STATS_BH(net, IPSTATS_MIB_REASMREQDS);

        /* Start by cleaning up the memory. */
        ip_evictor(net);
        /* Lookup (or create) queue header */
        if ((qp = ip_find(net, ip_hdr(skb), user)) != NULL) {
                int ret;

                spin_lock(&qp->q.lock);
                ret = ip_frag_queue(qp, skb);
                spin_unlock(&qp->q.lock);
                ipq_put(qp);
                return ret;
        }

        IP_INC_STATS_BH(net, IPSTATS_MIB_REASMFAILS);
        kfree_skb(skb);
        return -ENOMEM;
}
```

Before looking at the `ip_frag_queue()` method, consider the following macro, which simply returns the `ipfrag_skb_cb` object which is associated with the specified SKB:

```
#define FRAG_CB(skb)    ((struct ipfrag_skb_cb *)((skb)->cb))
```

Now let's look at the `ip_frag_queue()` method. I will not describe all the details because the method is very complicated and takes into account problems that might arise from overlapping (overlapping fragments may occur due to retransmissions). In the following snippet, `qp->q.len` is set to be the total length of the packet, including all its fragments; when the IP_MF flag is not set, this means that this is the last fragment:

```
static int ip_frag_queue(struct ipq *qp, struct sk_buff *skb)
{
        struct sk_buff *prev, *next;
        . . .
        /* Determine the position of this fragment. */
        end = offset + skb->len - ihl;
        err = -EINVAL;

        /* Is this the final fragment? */
        if ((flags & IP_MF) == 0) {
                /* If we already have some bits beyond end
                 * or have different end, the segment is corrupted.
                 */
                if (end < qp->q.len ||
                    ((qp->q.last_in & INET_FRAG_LAST_IN) && end != qp->q.len))
                        goto err;
                qp->q.last_in |= INET_FRAG_LAST_IN;
                qp->q.len = end;
        } else {
           . . .
        }
```

Now the location for adding the fragment is found by looking for the first place which is after the fragment offset (the linked list of fragments is ordered by offset):

```
        . . .
        prev = NULL;
        for (next = qp->q.fragments; next != NULL; next = next->next) {
               if (FRAG_CB(next)->offset >= offset)
                       break;  /* bingo! */
               prev = next;
        }
```

Now, prev points to where to add the new fragment if it is not NULL. Skipping handling overlapping and some other checks, let's continue to the insertion of the fragment into the list:

```
        FRAG_CB(skb)->offset = offset;
        /* Insert this fragment in the chain of fragments. */
        skb->next = next;
        if (!next)
            qp->q.fragments_tail = skb;
        if (prev)
            prev->next = skb;
        else
            qp->q.fragments = skb;
        . . .
        qp->q.meat += skb->len;
```

Note that the qp->q.meat is incremented by skb->len for each fragment. As mentioned earlier, qp->q.len is the total length of all fragments, and when it is equal to qp->q.meat, it means that all fragments were added and should be reassembled into one packet with the ip_frag_reasm() method.

Now you can see how and where reassembly takes place: (reassembly is done by calling the ip_frag_reasm() method):

```
    if (qp->q.last_in == (INET_FRAG_FIRST_IN | INET_FRAG_LAST_IN) &&
        qp->q.meat == qp->q.len) {
        unsigned long orefdst = skb->_skb_refdst;

        skb->_skb_refdst = 0UL;
        err = ip_frag_reasm(qp, prev, dev);
        skb->_skb_refdst = orefdst;
        return err;
    }
```

Let's take a look at the ip_frag_reasm() method:

```
static int ip_frag_reasm(struct ipq *qp, struct sk_buff *prev,
                         struct net_device *dev)
{
        struct net *net = container_of(qp->q.net, struct net, ipv4.frags);
        struct iphdr *iph;
        struct sk_buff *fp, *head = qp->q.fragments;
        int len;
      ...
```

```
        /* Allocate a new buffer for the datagram. */
        ihlen = ip_hdrlen(head);
        len = ihlen + qp->q.len;

        err = -E2BIG;
        if (len > 65535)
               goto out_oversize;
        ...
        skb_push(head, head->data - skb_network_header(head));
```

Forwarding

The main handler for forwarding a packet is 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);
```

I should describe why Large Receive Offload (LRO) packets are dropped in forwarding. LRO is a performance-optimization technique that merges packets together, creating one large SKB, before they are passed to higher network layers. This reduces CPU overhead and thus improves the performance. Forwarding a large SKB, which was built by LRO, is not acceptable because it will be larger than the outgoing MTU. Therefore, when LRO is enabled the SKB is freed and the method returns NET_RX_DROP. Generic Receive Offload (GRO) design included forwarding ability, but LRO did not:

```
    if (skb_warn_if_lro(skb))
        goto drop;
```

If the router_alert option is set, the ip_call_ra_chain() method should be invoked to handle the packet. When calling setsockopt() with IP_ROUTER_ALERT on a raw socket, the socket is added to a global list named ip_ra_chain (see include/net/ip.h). The ip_call_ra_chain() method delivers the packet to all raw sockets. You might wonder why is the packet delivered to all raw sockets and not to a single raw socket? In raw sockets there are no ports on which the sockets listen, as opposed to TCP or UDP.

If the pkt_type—which was determined by the eth_type_trans() method, which should be called from the network driver, and which is discussed in Appendix A—is not PACKET_HOST, the packet is discarded:

```
    if (IPCB(skb)->opt.router_alert && ip_call_ra_chain(skb))
        return NET_RX_SUCCESS;

    if (skb->pkt_type != PACKET_HOST)
        goto drop;
```

The ttl (Time To Live) field of the IPv4 header is a counter which is decreased by 1 in each forwarding device. If the ttl reaches 0, that is an indication that the packet should be dropped and that a corresponding time exceeded ICMPv4 message with "TTL Count Exceeded" code should be sent:

```
    if (ip_hdr(skb)->ttl <= 1)
        goto too_many_hops;. . .
        . . .
```

```
too_many_hops:
    /* Tell the sender its packet died... */
    IP_INC_STATS_BH(dev_net(skb_dst(skb)->dev), IPSTATS_MIB_INHDRERRORS);
    icmp_send(skb, ICMP_TIME_EXCEEDED, ICMP_EXC_TTL, 0);
    . . .
```

Now a check is performed if both the strict route flag (`is_strictroute`) is set and the `rt_uses_gateway` flag is set; in such a case, strict routing cannot be applied, and a "Destination Unreachable" ICMPv4 message with "Strict Routing Failed" code is sent back:

```
    rt = skb_rtable(skb);

    if (opt->is_strictroute && rt->rt_uses_gateway)
        goto sr_failed;
    . . .
sr_failed:
    icmp_send(skb, ICMP_DEST_UNREACH, ICMP_SR_FAILED, 0);
    goto drop;
    . . .
```

Now a check is performed to see whether the length of the packet is larger than the outgoing device MTU. If it is, that means the packet is not permitted to be sent as it is. Another check is performed to see whether the DF (Don't Fragment) field in the IPv4 header is set and whether the `local_df` flag in the SKB is not set. If these conditions are met, it means that when the packet reaches the `ip_output()` method, it will not be fragmented with the `ip_fragment()` method. This means the packet cannot be sent as is, and it also cannot be fragmented; so a destination unreachable ICMPv4 message with "Fragmentation Needed" code is sent back, the packet is dropped, and the statistics (IPSTATS_MIB_FRAGFAILS) are updated:

```
      if (unlikely(skb->len > dst_mtu(&rt->dst) &&
          !skb_is_gso(skb) && (ip_hdr(skb)->frag_off & htons(IP_DF)))
            && !skb->local_df) {
      IP_INC_STATS(dev_net(rt->dst.dev), IPSTATS_MIB_FRAGFAILS);
      icmp_send(skb, ICMP_DEST_UNREACH, ICMP_FRAG_NEEDED,
              htonl(dst_mtu(&rt->dst)));
      goto drop;    }
```

Because the `ttl` and checksum of the IPv4 header are going to be changed, a copy of the SKB should be kept:

```
        /* We are about to mangle packet. Copy it! */
         if (skb_cow(skb, LL_RESERVED_SPACE(rt->dst.dev)+rt->dst.header_len))
                 goto drop;
         iph = ip_hdr(skb);
```

As mentioned earlier, each node that forwards the packet should decrease the `ttl`. As a result of the `ttl` change, the checksum is also updated accordingly in the `ip_decrease_ttl()` method:

```
         /* Decrease ttl after skb cow done */
         ip_decrease_ttl(iph);
```

Now a redirect ICMPv4 message is sent back. If the RTCF_DOREDIRECT flag of the routing entry is set then a "Redirect To Host" code is used for this message (I discuss ICMPv4 redirect messages in Chapter 5).

```
        /*
         *      We now generate an ICMP HOST REDIRECT giving the route
         *      we calculated.
         */
        if (rt->rt_flags&RTCF_DOREDIRECT && !opt->srr && !skb_sec_path(skb))
                ip_rt_send_redirect(skb);
```

The `skb->priority` in the Tx path is set to be the socket priority (`sk->sk_priority`)—see, for example, the `ip_queue_xmit()` method. The socket priority, in turn, can be set by calling the `setsockopt()` system call with SOL_SOCKET and SO_PRIORITY. However, when forwarding the packet, there is no socket attached to the SKB. So, in the `ip_forward()` method, the `skb->priority` is set according to a special table called `ip_tos2prio`. This table has 16 entries (see `include/net/route.h`).

```
        skb->priority = rt_tos2priority(iph->tos);
```

Now, assuming that there are no netfilter NF_INET_FORWARD hooks, the `ip_forward_finish()` method is invoked:

```
        return NF_HOOK(NFPROTO_IPV4, NF_INET_FORWARD, skb, skb->dev,
                       rt->dst.dev, ip_forward_finish);
```

In `ip_forward_finish()`, the statistics are updated, and we check that the IPv4 packet includes IP options. If it does, the `ip_forward_options()` method is invoked to handle the options. If it does not have options, the `dst_output()` method is called. The only thing this method does is invoke `skb_dst(skb)->output(skb)`:

```
static int ip_forward_finish(struct sk_buff *skb)
     {
     struct ip_options *opt  = &(IPCB(skb)->opt);

     IP_INC_STATS_BH(dev_net(skb_dst(skb)->dev), IPSTATS_MIB_OUTFORWDATAGRAMS);

     IP_ADD_STATS_BH(dev_net(skb_dst(skb)->dev), IPSTATS_MIB_OUTOCTETS, skb->len);


     if (unlikely(opt->optlen))
             ip_forward_options(skb);

     return dst_output(skb);
     }
```

In this section you learned about the methods for forwarding packets (`ip_forward()` and `ip_forward_finish()`), about cases when a packet is discarded in forwarding, about cases when an ICMP redirect is sent, and more.

Summary

This chapter dealt with the IPv4 protocol—how an IPv4 packet is built, the IPv4 header structure and IP options, and how they are handled. You learned how the IPv4 protocol handler is registered. You also learned about the Rx path (how the reception of IPv4 packets is handled) and about the Tx path in IPv4 (how the transmission of IPv4 packets is handled). There are cases when packets are larger than the network interface MTU, and as a result they can't be sent without being fragmented on the sender side and later defragmented on the receiver side. You learned about the implementation of fragmentation in IPv4 (including how the slow path and the fast path are implemented and when they are used) and the implementation of defragmentation in IPv4. The chapter also covered IPv4 forwarding—sending an incoming packet on a different network interface without passing it to the upper layer. And you saw some examples of when a packet is discarded in the forwarding process and when an ICMP redirect is sent. The next chapter discusses the IPv4 routing subsystem. The "Quick Reference" section that follows covers the top methods that are related to the topics discussed in this chapter, ordered by their context.

Quick Reference

I conclude this chapter with a short list of important methods and macros of the IPv4 subsystem that were mentioned in this chapter.

Methods

The following is a short list of important methods of the IPv4 layer, which were mentioned in this chapter.

int ip_queue_xmit(struct sk_buff *skb, struct flowi *fl);

This method moves packets from L4 (the transport layer) to L3 (the network layer), invoked for example from TCPv4.

int ip_append_data(struct sock *sk, struct flowi4 *fl4, int getfrag(void *from, char *to, int offset, int len, int odd, struct sk_buff *skb), void *from, int length, int transhdrlen, struct ipcm_cookie *ipc, struct rtable **rtp, unsigned int flags);

This method moves packets from L4 (the transport layer) to L3 (the network layer); invoked for example from UDPv4 when working with corked UDP sockets and from ICMPv4.

struct sk_buff *ip_make_skb(struct sock *sk, struct flowi4 *fl4, int getfrag(void *from, char *to, int offset, int len, int odd, struct sk_buff *skb), void *from, int length, int transhdrlen, struct ipcm_cookie *ipc, struct rtable **rtp, unsigned int flags);

This method was added in kernel 2.6.39 for enabling lockless transmit fast path to the UDPv4 implementation; called when not using the UDP_CORK socket option.

int ip_generic_getfrag(void *from, char *to, int offset, int len, int odd, struct sk_buff *skb);

This method is a generic method for copying data from userspace into the specified `skb`.

static int icmp_glue_bits(void *from, char *to, int offset, int len, int odd, struct sk_buff *skb);

This method is the ICMPv4 `getfrag` callback. The ICMPv4 module calls the `ip_append_data()` method with `icmp_glue_bits()` as the `getfrag` callback.

int ip_options_compile(struct net *net,struct ip_options *opt, struct sk_buff *skb);

This method builds an `ip_options` object by parsing IP options.

void ip_options_fragment(struct sk_buff *skb);

This method fills the options whose copied flag is not set with NOOPs and resets the corresponding fields of these IP options. Invoked only for the first fragment.

void ip_options_build(struct sk_buff *skb, struct ip_options *opt, __be32 daddr, struct rtable *rt, int is_frag);

This method takes the specified `ip_options` object and writes its content to the IPv4 header. The last parameter, `is_frag`, is in practice 0 in all invocations of the `ip_options_build()` method.

void ip_forward_options(struct sk_buff *skb);

This method handles IP options forwarding.

int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt, struct net_device *orig_dev);

This method is the main Rx handler for IPv4 packets.

ip_rcv_options(struct sk_buff *skb);

This method is the main method for handling receiving a packet with options.

int ip_options_rcv_srr(struct sk_buff *skb);

This method handles receiving a packet with strict route option.

int ip_forward(struct sk_buff *skb);

This method is the main handler for forwarding IPv4 packets.

static void ipmr_queue_xmit(struct net *net, struct mr_table *mrt, struct sk_buff *skb, struct mfc_cache *c, int vifi);

This method is the multicast transmission method.

static int raw_send_hdrinc(struct sock *sk, struct flowi4 *fl4, void *from, size_t length, struct rtable **rtp, unsigned int flags);

This method is used by raw sockets for transmission when the IPHDRINC socket option is set. It calls the `dst_output()` method directly.

int ip_fragment(struct sk_buff *skb, int (*output)(struct sk_buff *));

This method is the main fragmentation method.

int ip_defrag(struct sk_buff *skb, u32 user);

This method is the main defragmentation method. It processes an incoming IP fragment. The second parameter, `user`, indicates where this method was invoked from. For a full list of possible values for the second parameter, look in the `ip_defrag_users enum` definition in `include/net/ip.h`.

bool skb_has_frag_list(const struct sk_buff *skb);

This method returns `true` if `skb_shinfo(skb)->frag_list` is not NULL. The method `skb_has_frag_list()` was named `skb_has_frags()` in the past, and was renamed `skb_has_frag_list()` in kernel 2.6.37. (The reason was that the name was confusing.) SKBs can be fragmented in two ways: via a page array (called `skb_shinfo(skb)->frags[]`) and via a list of SKBs (called `skb_shinfo(skb)->frag_list`). Because `skb_has_frags()` tests the latter, its name is confusing because it sounds more like it's testing the former.

int ip_local_deliver(struct sk_buff *skb);

This method handles delivering packets to Layer 4.

int ip_options_get_from_user(struct net *net, struct ip_options_rcu **optp, unsigned char __user *data, int optlen);

This method handles setting options from userspace by the `setsockopt()` system call with IP_OPTIONS.

bool ip_is_fragment(const struct iphdr *iph);

This method returns `true` if the packet is a fragment.

int ip_decrease_ttl(struct iphdr *iph);

This method decrements the `ttl` of the specified IPv4 header by 1 and, because one of the IPv4 header fields had changed (`ttl`), recalculates the IPv4 header checksum.

int ip_build_and_send_pkt(struct sk_buff *skb, struct sock *sk, __be32 saddr, __be32 daddr, struct ip_options_rcu *opt);

This method is used by TCPv4 to send SYN ACK. See the `tcp_v4_send_synack()` method in `net/ipv4/tcp_ipv4.c`.

int ip_mr_input(struct sk_buff *skb);

This method handles incoming multicast packets.

int ip_mr_forward(struct net *net, struct mr_table *mrt, struct sk_buff *skb, struct mfc_cache *cache, int local);

This method forwards multicast packets.

bool ip_call_ra_chain(struct sk_buff *skb);

This method handles the Router Alert IP option.

Macros

This section mentions some macros from this chapter that deal with mechanisms encountered in the IPv4 stack, such as fragmentation, netfilter hooks, and IP options.

IPCB(skb)

This macro returns the `inet_skb_parm` object which `skb->cb` points to. It is used to access the `ip_options` object stored in the `inet_skb_parm` object (`include/net/ip.h`).

FRAG_CB(skb)

This macro returns the `ipfrag_skb_cb` object which `skb->cb` points to (`net/ipv4/ip_fragment.c`).

int NF_HOOK(uint8_t pf, unsigned int hook, struct sk_buff *skb, struct net_device *in, struct net_device *out, int (*okfn)(struct sk_buff *))

This macro is the netilter hook; the first parameter, `pf`, is the protocol family; for IPv4 it is NFPROTO_IPV4, and for IPv6 it is NFPROTO_IPV6. The second parameter is one of the five netfilter hook points in the network stack; these five points are defined in `include/uapi/linux/netfilter.h` and can be used both by IPv4 and IPv6. The `okfn` callback is to be called if there is no hook registered or if the registered netfilter hook does not discard or reject the packet.

int NF_HOOK_COND(uint8_t pf, unsigned int hook, struct sk_buff *skb, struct net_device *in, struct net_device *out, int (*okfn)(struct sk_buff *), bool cond)

This macro is same as the `NF_HOOK()` macro, but with an additional Boolean parameter, `cond`, which must be `true` so that the netfilter hook will be called.

IPOPT_COPIED()

This macro returns the copied flag of the option type.