Linux Kernel Networking

Implementation and Theory

(Chapter 1~10)

Rami Rosen

Apress®

Contents

CHAPTER 1

Introduction

This book deals with the implementation of the Linux Kernel Networking stack and the theory behind it. You will find in the following pages an in-depth and detailed analysis of the networking subsystem and its architecture. I will not burden you with topics not directly related to networking, which you may encounter while reading kernel networking code (for example, locking and synchronization, SMP, atomic operations, and so on). There are plenty of resources about such topics. On the other hand, there are very few up-to-date resources that focus on kernel networking proper. By this I mean primarily describing the traversal of the packet in the Linux Kernel Networking stack and its interaction with various networking layers and subsystems—and how various networking protocols are implemented.

This book is also not a cumbersome, line-by-line code walkthrough. I focus on the essence of the implementation of each network layer and the theory guidelines and principles that led to this implementation. The Linux operating system has proved itself in recent years as a successful, reliable, stable, and popular operating system. And it seems that its popularity is growing steadily, in a wide variety of flavors, from mainframes, data centers, core routers, and web servers to embedded devices like wireless routers, set-top boxes, medical instruments, navigation equipment (like GPS devices), and consumer electronics devices. Many semiconductor vendors use Linux as the basis for their Board Support Packages (BSPs). The Linux operating system, which started as a project of a Finnish student named Linus Torvalds back in 1991, based on the UNIX operating system, proved to be a serious and reliable operating system and a rival for veteran proprietary operating systems.

Linux began as an Intel x86-based operating system but has been ported to a very wide range of processors, including ARM, PowerPC, MIPS, SPARC, and more. The Android operating system, based upon the Linux kernel, is common today in tablets and smartphones, and seems likely to gain popularity in the future in smart TVs. Apart from Android, Google has also contributed some kernel networking features that were merged into the mainline kernel.

Linux is an open source project, and as such it has an advantage over other proprietary operating systems: its source code is freely available under the General Public License (GPL). Other open source operating systems, like the different types of BSD, have much less popularity. I should also mention in this context the OpenSolaris project, based on the Common Development and Distribution License (CDDL). This project, started by Sun Microsystems, has not achieved the popularity that Linux has. Among the large community of active Linux developers, some contribute code on behalf of the companies they work for, and some contribute code voluntarily. All of the kernel development process is accessible via the kernel mailing lists. There is one central mailing list, the Linux Kernel Mailing List (LKML), and many subsystems have their own mailing lists. Contributing code is done via sending patches to the appropriate kernel mailing lists and to the maintainers, and these patches are discussed over the mailing lists.

The Linux Kernel Networking stack is a very important subsystem of the Linux kernel. It is quite difficult to find a Linux-based system, whether it is a desktop, a server, a mobile device or any other embedded device, that does not use any kind of networking. Even in the rare case when a machine doesn't have any hardware network devices, you will still be using networking (maybe unconsciously) when you use X-Windows, as X-Windows itself is based upon client-server networking. A wide range of projects are related to the Linux Networking stack, from core routers to small embedded devices. Some of these projects deal with adding vendor-specific features. For example, some hardware vendors implement Generic Segmentation Offload (GSO) in some network devices. GSO is a networking feature of the kernel network stack that divides a large packet into smaller ones in the Tx path. Many hardware vendors implement checksumming in hardware in their network devices. *Checksum* is a mechanism to verify that a packet was not damaged on transit by calculating some hash from the packet and attaching it to the packet. Many projects provide some security enhancements for Linux. Sometimes these enhancements require some changes in the networking subsystem, as you will see, for example, in Chapter 3, when discussing the Openwall GNU/*/Linux project. In the embedded device arena there are, for example, many wireless routers that are Linux based; one example is the WRT54GL Linksys router, which runs Linux. There is also an open source, Linux-based operating system that can run on this device (and on some other devices), named OpenWrt, with a large and active community of developers (see `https://openwrt.org/`). Learning about how the various protocols are implemented by the Linux Kernel Networking stack and becoming familiar with the main data structures and the main paths of a packet in it are essential to understanding it better.

The Linux Network Stack

There are seven logical networking layers according to the Open Systems Interconnection (OSI) model. The lowest layer is the physical layer, which is the hardware, and the highest layer is the application layer, where userspace software processes are running. Let's describe these seven layers:

1. *The physical layer:* Handles electrical signals and the low level details.
2. *The data link layer:* Handles data transfer between endpoints. The most common data link layer is Ethernet. The Linux Ethernet network device drivers reside in this layer.
3. *The network layer:* Handles packet forwarding and host addressing. In this book I discuss the most common network layers of the Linux Kernel Networking subsystem: IPv4 or IPv6. There are other, less common network layers which Linux implements, like DECnet, but they are not discussed.
4. *The protocol layer/transport layer:* Handles data sending between nodes. The TCP and UDP protocols are the best-known protocols.
5. *The session layer:* Handles sessions between endpoints.
6. *The presentation layer:* Handles delivery and formatting.
7. *The application layer:* Provides network services to end-user applications.

Figure 1-1 shows the seven layers according to the OSI model.

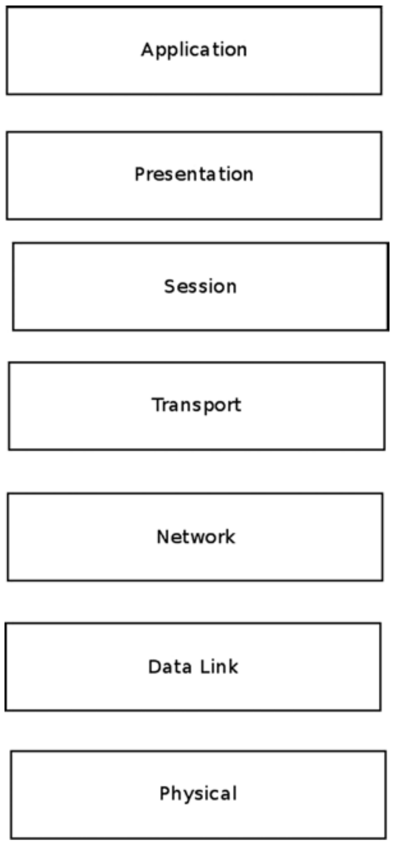


Figure 1-1. *The OSI seven-layer model*

Figure 1-2 shows the three layers that the Linux Kernel Networking stack handles. The L2, L3, and L4 layers in this figure correspond to the data link layer, the network layer, and the transport layer in the seven-layer model, respectively. The essence of the Linux kernel stack is passing incoming packets from L2 (the network device drivers) to L3 (the network layer, usually IPv4 or IPv6) and then to L4 (the transport layer, where you have, for example, TCP or UDP listening sockets) if they are for local delivery, or back to L2 for transmission when the packets should be forwarded. Outgoing packets that were locally generated are passed from L4 to L3 and then to L2 for actual transmission by the network device driver. Along this way there are many stages, and many things can happen. For example:

- The packet can be changed due to protocol rules (for example, due to an IPsec rule or to a NAT rule).
- The packet can be discarded.
- The packet can cause an error message to be sent.
- The packet can be fragmented.
- The packet can be defragmented.
- A checksum should be calculated for the packet.

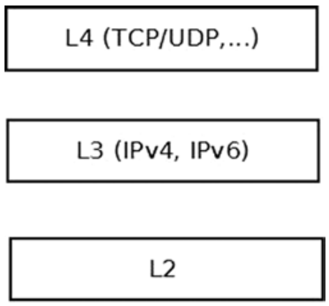


Figure 1-2. *The Linux Kernel Networking layers*

The kernel does not handle any layer above L4; those layers (the session, presentation, and application layers) are handled solely by userspace applications. The physical layer (L1) is also not handled by the Linux kernel.

If you feel overwhelmed, don't worry. You will learn a lot more about everything described here in a lot more depth in the following chapters.

The Network Device

The lower layer, Layer 2 (L2), as seen in Figure 1-2, is the link layer. The network device drivers reside in this layer. This book is not about network device driver development, because it focuses on the Linux kernel networking stack. I will briefly describe here the `net_device` structure, which represents a network device, and some of the concepts that are related to it. You should have a basic familiarity with the network device structure in order to better understand the network stack. Parameters of the device—like the size of MTU, which is typically 1,500 bytes for Ethernet devices—determine whether a packet should be fragmented. The `net_device` is a very large structure, consisting of device parameters like these:

- The IRQ number of the device.
- The MTU of the device.
- The MAC address of the device.
- The name of the device (like `eth0` or `eth1`).
- The flags of the device (for example, whether it is up or down).
- A list of multicast addresses associated with the device.
- The `promiscuity` counter (discussed later in this section).
- The features that the device supports (like GSO or GRO offloading).
- An object of network device callbacks (`net_device_ops` object), which consists of function pointers, such as for opening and stopping a device, starting to transmit, changing the MTU of the network device, and more.
- An object of `ethtool` callbacks, which supports getting information about the device by running the command-line `ethtool` utility.
- The number of Tx and Rx queues, when the device supports multiqueues.
- The timestamp of the last transmit of a packet on this device.
- The timestamp of the last reception of a packet on this device.

The following is the definition of some of the members of the `net_device` structure to give you a first impression:

```
struct net_device {
    unsigned int           irq;            /* device IRQ number    */
    . . .
    const struct net_device_ops *netdev_ops;
    . . .
    unsigned int           mtu;
    . . .
    unsigned int           promiscuity;
    . . .
    unsigned char          *dev_addr;
    . . .
};
(include/linux/netdevice.h)
```

Appendix A of the book includes a very detailed description of the `net_device` structure and most of its members. In that appendix you can see the `irq`, `mtu`, and other members mentioned earlier in this chapter.

When the `promiscuity` counter is larger than 0, the network stack does not discard packets that are not destined to the local host. This is used, for example, by packet analyzers ("sniffers") like `tcpdump` and `wireshark`, which open raw sockets in userspace and want to receive also this type of traffic. It is a counter and not a Boolean in order to enable opening several sniffers concurrently: opening each such sniffer increments the counter by 1. When a sniffer is closed, the `promiscuity` counter is decremented by 1; and if it reaches 0, there are no more sniffers running, and the device exits the promiscuous mode.

When browsing kernel networking core source code, in various places you will probably encounter the term NAPI (New API), which is a feature that most network device drivers implement nowadays. You should know what it is and why network device drivers use it.

New API (NAPI) in Network Devices

The old network device drivers worked in interrupt-driven mode, which means that for every received packet, there was an interrupt. This proved to be inefficient in terms of performance under high load traffic. A new software technique was developed, called New API (NAPI), which is now supported on almost all Linux network device drivers. NAPI was first introduced in the 2.5/2.6 kernel and was backported to the 2.4.20 kernel. With NAPI, under high load, the network device driver works in polling mode and not in interrupt-driven mode. This means that each received packet does not trigger an interrupt. Instead the packets are buffered in the driver, and the kernel polls the driver from time to time to fetch the packets. Using NAPI improves performance under high load. For sockets applications that need the lowest possible latency and are willing to pay a cost of higher CPU utilization, Linux has added a capability for Busy Polling on Sockets from kernel 3.11 and later. This technology is discussed in Chapter 14, in the "Busy Poll Sockets" section.

With your new knowledge about network devices under your belt, it is time to learn about the traversal of a packet inside the Linux Kernel Networking stack.

Receiving and Transmitting Packets

The main tasks of the network device driver are these:

- To receive packets destined to the local host and to pass them to the network layer (L3), and from there to the transport layer (L4)
- To transmit outgoing packets generated on the local host and sent outside, or to forward packets that were received on the local host

For each packet, incoming or outgoing, a lookup in the routing subsystem is performed. The decision about whether a packet should be forwarded and on which interface it should be sent is done based on the result of the lookup in the routing subsystem, which I describe in depth in Chapters 5 and 6. The lookup in the routing subsystem is not the only factor that determines the traversal of a packet in the network stack. For example, there are five points in the network stack where callbacks of the netfilter subsystem (often referred to as netfilter hooks) can be registered. The first netfilter hook point of a received packet is NF_INET_PRE_ROUTING, before a routing lookup was performed. When a packet is handled by such a callback, which is invoked by a macro named NF_HOOK(), it will continue its traversal in the networking stack according to the result of this callback (also called `verdict`). For example, if the `verdict` is NF_DROP, the packet will be discarded, and if the `verdict` is NF_ACCEPT, the packet will continue its traversal as usual. Netfilter hooks callbacks are registered by the `nf_register_hook()` method or by the `nf_register_hooks()` method, and you will encounter these invocations, for example, in various netfilter kernel modules. The kernel netfilter subsystem is the infrastructure for the well-known `iptables` userspace package. Chapter 9 describes the netfilter subsystem and the netfilter hooks, along with the connection tracking layer of netfilter.

Besides the netfilter hooks, the packet traversal can be influenced by the IPsec subsystem—for example, when it matches a configured IPsec policy. IPsec provides a network layer security solution, and it uses the ESP and the AH protocols. IPsec is mandatory according to IPv6 specification and optional in IPv4, though most operating systems, including Linux, implemented IPsec also in IPv4. IPsec has two modes of operation: transport mode and tunnel mode. It is used as a basis for many virtual private network (VPN) solutions, though there are also non-IPsec VPN solutions. You learn about the IPsec subsystem and about IPsec policies in Chapter 10, which also discusses the problems that occur when working with IPsec through a NAT, and the IPsec NAT traversal solution.

Still other factors can influence the traversal of the packet—for example, the value of the `ttl` field in the IPv4 header of a packet being forwarded. This `ttl` is decremented by 1 in each forwarding device. When it reaches 0, the packet is discarded, and an ICMPv4 message of "Time Exceeded" with "TTL Count Exceeded" code is sent back. This is done to avoid an endless journey of a forwarded packet because of some error. Moreover, each time a packet is forwarded successfully and the `ttl` is decremented by 1, the checksum of the IPv4 header should be recalculated, as its value depends on the IPv4 header, and the `ttl` is one of the IPv4 header members. Chapter 4, which deals with the IPv4 subsystem, talks more about this. In IPv6 there is something similar, but the hop counter in the IPv6 header is named `hop_limit` and not `ttl`. You will learn about this in Chapter 8, which deals with the IPv6 subsystem. You will also learn about ICMP in IPv4 and in IPv6 in Chapter 3, which deals with ICMP.

A large part of the book discusses the traversal of a packet in the networking stack, whether it is in the receive path (Rx path, also known as *ingress* traffic) or the transmit path (Tx path, also known as *egress* traffic). This traversal is complex and has many variations: large packets could be fragmented before they are sent; on the other hand, fragmented packets should be assembled (discussed in Chapter 4). Packets of different types are handled differently. For example, multicast packets are packets that can be processed by a group of hosts (as opposed to unicast packets, which are destined to a specified host). Multicast can be used, for example, in applications of streaming media in order to consume less network resources. Handling IPv4 multicast traffic is discussed in Chapter 4. You will also learn how a host joins and leaves a multicast group; in IPv4, the Internet Group Management Protocol (IGMP) protocol handles multicast membership. Yet there are cases when the host is configured as a multicast router, and multicast traffic should be forwarded and not delivered to the local host. These cases are more complex as they should be handled in conjunction with a userspace multicast routing daemon, like the `pimd` daemon or the `mrouted` daemon. These cases, which are called multicast routing, are discussed in Chapter 6.

To better understand the packet traversal, you must learn about how a packet is represented in the Linux kernel. The `sk_buff` structure represents an incoming or outgoing packet, including its headers (`include/linux/skbuff.h`). I refer to an `sk_buff` object as SKB in many places along this book, as this is the common way to denote `sk_buff` objects (SKB stands for *socket buffer*). The socket buffer (`sk_buff`) structure is a large structure—I will only discuss a few members of this structure in this chapter.

The Socket Buffer

The sk_buff structure is described in depth in Appendix A. I recommend referring to this appendix when you need to know more about one of the SKB members or how to use the SKB API. Note that when working with SKBs, you must adhere to the SKB API. Thus, for example, when you want to advance the skb->data pointer, you do not do it directly, but with the skb_pull_inline() method or the skb_pull() method (you will see an example of this later in this section). And if you want to fetch the L4 header (transport header) from an SKB, you do it by calling the skb_transport_header() method. Likewise if you want to fetch the L3 header (network header), you do it by calling the skb_network_header() method, and if you want to fetch the L2 header (MAC header), you do it by calling the skb_mac_header() method. These three methods get an SKB as a single parameter.

Here is the (partial) definition of the sk_buff structure:

```
struct sk_buff {
    . . .
    struct sock             *sk;
    struct net_device       *dev;
    . . .
    __u8                    pkt_type:3,
    . . .
    __be16                  protocol;
    . . .
    sk_buff_data_t          tail;
    sk_buff_data_t          end;
    unsigned char           *head,
                            *data;

    sk_buff_data_t          transport_header;
    sk_buff_data_t          network_header;
    sk_buff_data_t          mac_header;
    . . .

};
(include/linux/skbuff.h)
```

When a packet is received on the wire, an SKB is allocated by the network device driver, typically by calling the netdev_alloc_skb() method (or the dev_alloc_skb() method, which is a legacy method that calls the netdev_alloc_skb() method with the first parameter as NULL). There are cases along the packet traversal where a packet can be discarded, and this is done by calling kfree_skb() or dev_kfree_skb(), both of which get as a single parameter a pointer to an SKB. Some members of the SKB are determined in the link layer (L2). For example, the pkt_type is determined by the eth_type_trans() method, according to the destination Ethernet address. If this address is a multicast address, the pkt_type will be set to PACKET_MULTICAST; if this address is a broadcast address, the pkt_type will be set to PACKET_BROADCAST; and if this address is the address of the local host, the pkt_type will be set to PACKET_HOST. Most Ethernet network drivers call the eth_type_trans() method in their Rx path. The eth_type_trans() method also sets the protocol field of the SKB according to the ethertype of the Ethernet header. The eth_type_trans() method also advances the data pointer of the SKB by 14 (ETH_HLEN), which is the size of an Ethernet header, by calling the skb_pull_inline() method. The reason for this is that the skb->data should point to the header of the layer in which it currently resides. When the packet was in L2, in the network device driver Rx path, skb->data pointed to the L2 (Ethernet) header; now that the packet is going to be moved to Layer 3, immediately after the call to the eth_type_trans() method, skb->data should point to the network (L3) header, which starts immediately after the Ethernet header (see Figure 1-3).

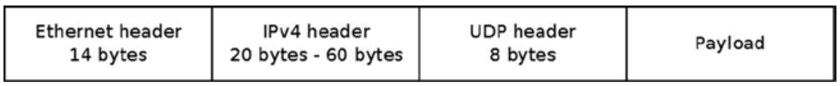


Figure 1-3. *An IPv4 packet*

The SKB includes the packet headers (L2, L3, and L4 headers) and the packet payload. In the packet traversal in the network stack, a header can be added or removed. For example, for an IPv4 packet generated locally by a socket and transmitted outside, the network layer (IPv4) adds an IPv4 header to the SKB. The IPv4 header size is 20 bytes as a minimum. When adding IP options, the IPv4 header size can be up to 60 bytes. IP options are described in Chapter 4, which discusses the IPv4 protocol implementation. Figure 1-3 shows an example of an IPv4 packet with L2, L3, and L4 headers. The example in Figure 1-3 is a UDPv4 packet. First is the Ethernet header (L2) of 14 bytes. Then there's the IPv4 header (L3) of a minimal size of 20 bytes up to 60 bytes, and after that is the UDPv4 header (L4), of 8 bytes. Then comes the payload of the packet.

Each SKB has a `dev` member, which is an instance of the `net_device` structure. For incoming packets, it is the incoming network device, and for outgoing packets it is the outgoing network device. The network device attached to the SKB is sometimes needed to fetch information which might influence the traversal of the SKB in the Linux Kernel Networking stack. For example, the MTU of the network device may require fragmentation, as mentioned earlier. Each transmitted SKB has a `sock` object associated to it (`sk`). If the packet is a forwarded packet, then `sk` is NULL, because it was not generated on the local host.

Each received packet should be handled by a matching network layer protocol handler. For example, an IPv4 packet should be handled by the `ip_rcv()` method, and an IPv6 packet should be handled by the `ipv6_rcv()` method. You will learn about the registration of the IPv4 protocol handler with the `dev_add_pack()` method in Chapter 4, and about the registration of the IPv6 protocol handler also with the `dev_add_pack()` method in Chapter 8. Moreover, I will follow the traversal of incoming and outgoing packets both in IPv4 and in IPv6. For example, in the `ip_rcv()` method, mostly sanity checks are performed, and if everything is fine the packet proceeds to an NF_INET_PRE_ROUTING hook callback, if such a callback is registered, and the next step, if it was not discarded by such a hook, is the `ip_rcv_finish()` method, where a lookup in the routing subsystem is performed. A lookup in the routing subsystem builds a destination cache entry (`dst_entry` object). You will learn about the `dst_entry` and about the `input` and `output` callback methods associated with it in Chapters 5 and 6, which describe the IPv4 routing subsystem.

In IPv4 there is a problem of limited address space, as an IPv4 address is only 32 bit. Organizations use NAT (discussed in Chapter 9) to provide local addresses to their hosts, but the IPv4 address space still diminishes over the years. One of the main reasons for developing the IPv6 protocol was that its address space is huge compared to the IPv4 address space, because the IPv6 address length is 128 bit. But the IPv6 protocol is not only about a larger address space. The IPv6 protocol includes many changes and additions as a result of the experience gained over the years with the IPv4 protocol. For example, the IPv6 header has a fixed length of 40 bytes as opposed to the IPv4 header, which is variable in length (from a minimum of 20 bytes to 60 bytes) due to IP options, which can expand it. Processing IP options in IPv4 is complex and quite heavy in terms of performance. On the other hand, in IPv6 you cannot expand the IPv6 header at all (it is fixed in length, as mentioned). Instead there is a mechanism of extension headers which is much more efficient than the IP options in IPv4 in terms of performance. Another notable change is with the ICMP protocol; in IPv4 it was used only for error reporting and for informative messages. In IPv6, the ICMP protocol is used for many other purposes: for Neighbour Discovery (ND), for Multicast Listener Discovery (MLD), and more. Chapter 3 is dedicated to ICMP (both in IPv4 and IPv6). The IPv6 Neighbour Discovery protocol is described in Chapter 7, and the MLD protocol is discussed in Chapter 8, which deals with the IPv6 subsystem.

As mentioned earlier, received packets are passed by the network device driver to the network layer, which is IPv4 or IPv6. If the packets are for local delivery, they will be delivered to the transport layer (L4) for handling by listening sockets. The most common transport protocols are UDP and TCP, discussed in Chapter 11, which discusses Layer 4, the transport layer. This chapter also covers two newer transport protocols, the Stream Control Transmission Protocol (SCTP) and the Datagram Congestion Control Protocol (DCCP). Both SCTP and DCCP adopted some TCP features and some UDP features, as you will find out. The SCTP protocol is known to be used in conjunction with the Long Term Evolution (LTE) protocol; the DCCP has not been tested so far in larger-scale Internet setups.

Packets generated by the local host are created by Layer 4 sockets—for example, by TCP sockets or by UDP sockets. They are created by a userspace application with the Sockets API. There are two main types of sockets: **datagram** sockets and **stream** sockets. These two types of sockets and the POSIX-based socket API are also discussed in Chapter 11, where you will also learn about the kernel implementation of sockets (`struct socket`, which provides an interface to userspace, and `struct sock`, which provides an interface to Layer 3). The packets generated locally are passed to the network layer, L3 (described in Chapter 4, in the section "Sending IPv4 Packets") and then are passed to the network device driver (L2) for transmission. There are cases when fragmentation takes place in Layer 3, the network layer, and this is also discussed in chapter 4.

Every Layer 2 network interface has an L2 address that identifies it. In the case of Ethernet, this is a 48-bit address, the MAC address which is assigned for each Ethernet network interface, provided by the manufacturer, and said to be unique (though you should consider that the MAC address for most network interfaces can be changed by userspace commands like `ifconfig` or `ip`). Each Ethernet packet starts with an Ethernet header, which is 14 bytes long. It consists of the Ethernet type (2 bytes), the source MAC address (6 bytes), and the destination MAC address (6 bytes). The Ethernet type value is 0x0800, for example, for IPv4, or 0x86DD for IPv6. For each outgoing packet, an Ethernet header should be built. When a userspace socket sends a packet, it specifies its destination address (it can be an IPv4 or an IPv6 address). This is not enough to build the packet, as the destination MAC address should be known. Finding the MAC address of a host based on its IP address is the task of the neighbouring subsystem, discussed in Chapter 7. Neighbor Discovery is handled by the ARP protocol in IPv4 and by the NDISC protocol in IPv6. These protocols are different: the ARP protocol relies on sending broadcast requests, whereas the NDISC protocol relies on sending ICMPv6 requests, which are in fact multicast packets. Both the ARP protocol and the NDSIC protocol are also discussed in Chapter 7.

The network stack should communicate with the userspace for tasks such as adding or deleting routes, configuring neighboring tables, setting IPsec policies and states, and more. The communication between userspace and the kernel is done with netlink sockets, described in Chapter 2. The `iproute2` userspace package, based on netlink sockets, is also discussed in Chapter 2, as well as the generic netlink sockets and their advantages.

The wireless subsystem is discussed in Chapter 12. This subsystem is maintained separately, as mentioned earlier; it has a `git` tree of its own and a mailing list of its own. There are some unique features in the wireless stack that do not exist in the ordinary network stack, such as power save mode (which is when a station or an access point enters a sleep state). The Linux wireless subsystem also supports special topologies, like Mesh network, ad-hoc network, and more. These topologies sometimes require using special features. For example, Mesh networking uses a routing protocol called Hybrid Wireless Mesh Protocol (HWMP), discussed in Chapter 12. This protocol works in Layer 2 and deals with MAC addresses, as opposed to the IPV4 routing protocol. Chapter 12 also discusses the mac80211 framework, which is used by wireless device drivers. Another very interesting feature of the wireless subsystem is the block acknowledgment mechanism in IEEE 802.11n, also discussed in Chapter 12.

In recent years InfiniBand technology has gained in popularity with enterprise datacenters. InfiniBand is based on a technology called Remote Direct Memory Access (RDMA). The RDMA API was introduced to the Linux kernel in version 2.6.11. In Chapter 13 you will find a good explanation about the Linux Infiniband implementation, the RDMA API, and its fundamental data structures.

Virtualization solutions are also becoming popular, especially due to projects like Xen or KVM. Also hardware improvements, like VT-x for Intel processors or AMD-V for AMD processors, have made virtualization more efficient. There is another form of virtualization, which may be less known but has its own advantages. This virtualization is based on a different approach: process virtualization. It is implemented in Linux by namespaces. There is currently support for six namespaces in Linux, and there could be more in the future. The namespaces feature is already used by projects like Linux Containers (`http://lxc.sourceforge.net/`) and Checkpoint/Restore In Userspace (CRIU). In order to support namespaces, two system calls were added to the kernel: `unshare()` and `setns()`; and six new flags were added to the CLONE_* flags, one for each type of namespace. I discuss namespaces and network namespaces in particular in Chapter 14. Chapter 14 also deals with the Bluetooth subsystem and gives a brief overview about the PCI subsystem, because many network device drivers are PCI devices. I do not delve into the PCI subsystem internals, because that is out of the scope of this book. Another interesting subsystem discussed in Chapter 14 is the IEEE 8012.15.4, which is for low-power and low-cost devices. These devices are sometimes mentioned in conjunction with the *Internet of Things* (IoT) concept, which involves connecting IP-enabled embedded devices to IP networks. It turns out that using IPv6 for these devices might be a good idea. This solution is termed IPv6 over Low Power Wireless Personal Area Networks (6LoWPAN). It has its own challenges, such as expanding the IPv6 Neighbour Discovery protocol to be suitable for such devices, which occasionally enter sleep mode (as opposed to ordinary IPv6 networks). These changes to the IPv6 Neighbour Discovery protocol have not been implemented yet, but it is interesting to consider the theory behind these changes. Apart from this, in Chapter 14 there are sections about other advanced topics like NFC, cgroups, Android, and more.

To better understand the Linux Kernel Network stack or participate in its development, you must be familiar with how its development is handled.

The Linux Kernel Networking Development Model

The kernel networking subsystem is very complex, and its development is quite dynamic. Like any Linux kernel subsystem, the development is done by `git` patches that are sent over a mailing list (sometimes over more than one mailing list) and that are eventually accepted or rejected by the maintainer of that subsystem. Learning about the Kernel Networking Development Model is important for many reasons. To better understand the code, to debug and solve problems in Linux Kernel Networking–based projects, to implement performance improvements and optimizations patches, or to implement new features, in many cases you need to learn many things such as the following:

- How to apply a patch
- How to read and interpret a patch
- How to find which patches could cause a given problem
- How to revert a patch
- How to find which patches are relevant to some feature
- How to adjust a project to an older kernel version (backporting)
- How to adjust a project to a newer kernel version (upgrading)
- How to clone a `git` tree
- How to rebase a `git` tree
- How to find out in which kernel version a specified `git` patch was applied

There are cases when you need to work with new features that were just added, and for this you need to know how to work with the latest, bleeding-edge tree. And there are cases when you encounter some bug or you want to add some new feature to the network stack, and you need to prepare a patch and submit it. The Linux Kernel Networking subsystem, like the other parts of the kernel, is managed by `git`, a source code management (SCM) system, developed by Linus Torvalds. If you intend to send patches for the mainline kernel, or if your project is managed by `git`, you must learn to use the `git` tool.

Sometimes you may even need to install a `git` server for development of local projects. Even if you are not intending to send any patches, you can use the `git` tool to retrieve a lot of information about the code and about the history of the development of the code. There are many available resources on the web about `git`; I recommend the free online book *Pro Git*, by Scott Chacon, available at `http://git-scm.com/book`. If you intend to submit your patches to the mainline, you must adhere to some strict rules for writing, checking, and submitting patches so that your patch will be applied. Your patch should conform to the kernel coding style and should be tested. You also need to be patient, as sometimes even a trivial patch can be applied only after several days. I recommend learning to configure a host for using the `git send-email` command to submit patches (though submitting patches can be done with other mail clients, even with the popular Gmail webmail client). There are plenty of guides on the web about how to use `git` to prepare and send kernel patches. I also recommend reading `Documentation/SubmittingPatches` and `Documentation/CodingStyle` in the kernel tree before submitting your first patch.

And I recommended using the following PERL scripts:

- `scripts/checkpatch.pl` to check the correctness of a patch
- `scripts/get_maintainer.pl` to find out to which maintainers a patch should be sent

One of the most important resources of information is the Kernel Networking Development mailing list, `netdev`: `netdev@vger.kernel.org`, archived at `www.spinics.net/lists/netdev`. This is a high volume list. Most of the posts are patches and Request for Comments (RFCs) for new code, along with comments and discussions about patches. This mailing list handles the Linux Kernel Networking stack and network device drivers, except for cases when dealing with a subsystem that has a specific mailing list and a specific `git` repository (such as the wireless subsystem, discussed in Chapter 12). Development of the `iproute2` and the `ethtool` userspace packages is also handled in the `netdev` mailing list. It should be mentioned here that not every networking subsystem has a mailing list of its own; for example, the IPsec subsystem (discussed in Chapter 10), does not have a mailing list, nor does the IEEE 802.15.4 subsystem (Chapter 14). Some networking subsystems have their own specific `git` tree, maintainer, and mailing list, such as the wireless mailing list and the Bluetooth mailing list. From time to time the maintainers of these subsystems send a pull request for their `git` trees over the `netdev` mailing list. Another source of information is `Documentation/networking` in the kernel tree. It has a lot of information in many files about various networking topics, but keep in mind that the file that you find there is not always up to date.

The Linux Kernel Networking subsystem is maintained in two `git` repositories. Patches and RFCs are sent to the `netdev` mailing list for both repositories. Here are the two `git` trees:

- *net:* `http://git.kernel.org/?p=linux/kernel/git/davem/net.git`: for fixes to existing code already in the mainline tree
- *net-next:* `http://git.kernel.org/?p=linux/kernel/git/davem/net-next.git`: new code for the future kernel release

From time to time the maintainer of the networking subsystem, David Miller, sends pull requests for mainline for these `git` trees to Linus over the LKML. You should be aware that there are periods of time, during merge with mainline, when the net-next `git` tree is closed, and no patches should be sent. An announcement about when this period starts and another one when it ends is sent over the `netdev` mailing list.

■ **Note** This book is based on kernel 3.9. All the code snippets are from this version, unless explicitly specified otherwise. The kernel tree is available from `www.kernel.org` as a `tar` file. Alternatively, you can download a kernel `git` tree with `git clone` (for example, using the URLs of the `git net` tree or the `git net-next` tree, which were mentioned earlier, or other `git` kernel repositories). There are plenty of guides on the Internet covering how to configure, build, and boot a Linux kernel. You can also browse various kernel versions online at `http://lxr.free-electrons.com/`. This website lets you follow where each method and each variable is referenced; moreover, you can navigate easily with a click of a mouse to previous versions of the Linux kernel. In case you are working with your own version of a Linux kernel tree, where some changes were made locally, you can locally install and configure a Linux Cross-Referencer server (LXR) on a local Linux machine. See `http://lxr.sourceforge.net/en/index.shtml`.

Summary

This chapter is a short introduction to the Linux Kernel Networking subsystem. I described the benefits of using Linux, a popular open source project, and the Kernel Networking Development Model. I also described the network device structure (`net_device`) and the socket buffer structure (`sk_buff`), which are the two most fundamental structures of the networking subsystem. You should refer to Appendix A for a detailed description of almost all the members of these structures and their uses. This chapter covered other important topics related to the traversal of a packet in the kernel networking stack, such as the lookup in the routing subsystem, fragmentation and defragmentation, protocol handler registration, and more. Some of these protocols are discussed in later chapters, including IPv4, IPv6, ICMP4 and ICMP6, ARP, and Neighbour Discovery. Several important subsystems, including the wireless subsystem, the Bluetooth subsystem, and the IEEE 812.5.4 subsystem, are also covered in later chapters. Chapter 2 starts the journey in the kernel network stack with netlink sockets, which provide a way for bidirectional communication between the userspace and the kernel, and which are talked about in several other chapters.

CHAPTER 2

Netlink Sockets

Chapter 1 discusses the roles of the Linux kernel networking subsystem and the three layers in which it operates. The netlink socket interface appeared first in the 2.2 Linux kernel as AF_NETLINK socket. It was created as a more flexible alternative to the awkward IOCTL communication method between userspace processes and the kernel. The IOCTL handlers cannot send asynchronous messages to userspace from the kernel, whereas netlink sockets can. In order to use IOCTL, there is another level of complexity: you need to define IOCTL numbers. The operation model of netlink is quite simple: you open and register a netlink socket in userspace using the socket API, and this netlink socket handles bidirectional communication with a kernel netlink socket, usually sending messages to configure various system settings and getting responses back from the kernel.

This chapter describes the netlink protocol implementation and API and discusses its advantages and drawbacks. I also talk about the new generic netlink protocol, discuss its implementation and its advantages, and give some illustrative examples using the `libnl` library. I conclude with a discussion of the socket monitoring interface.

The Netlink Family

The netlink protocol is a socket-based Inter Process Communication (IPC) mechanism, based on RFC 3549, “Linux Netlink as an IP Services Protocol.” It provides a bidirectional communication channel between userspace and the kernel or among some parts of the kernel itself. Netlink is an extension of the standard socket implementation. The netlink protocol implementation resides mostly under `net/netlink`, where you will find the following four files:

- `af_netlink.c`
- `af_netlink.h`
- `genetlink.c`
- `diag.c`

Apart from them, there are a few header files. In fact, the `af_netlink` module is the most commonly used; it provides the netlink kernel socket API, whereas the `genetlink` module provides a new generic netlink API with which it should be easier to create netlink messages. The `diag` monitoring interface module (`diag.c`) provides an API to dump and to get information about the netlink sockets. I discuss the `diag` module later in this chapter in the section “Socket monitoring interface.”

I should mention here that theoretically netlink sockets can be used to communicate between two userspace processes, or more (including sending multicast messages), though this is usually not used, and was not the original goal of netlink sockets. The UNIX domain sockets provide an API for IPC, and they are widely used for communication between two userspace processes.

Netlink has some advantages over other ways of communication between userspace and the kernel. For example, there is no need for polling when working with netlink sockets. A userspace application opens a socket and then calls `recvmsg()`, and enters a blocking state if no messages are sent from the kernel; see, for example, the `rtnl_listen()` method of the `iproute2` package (`lib/libnetlink.c`). Another advantage is that the kernel can be the initiator of sending asynchronous messages to userspace, without any need for the userspace to trigger any action (for example, by calling some IOCTL or by writing to some `sysfs` entry). Yet another advantage is that netlink sockets support multicast transmission.

You create netlink sockets from userspace with the `socket()` system call. The netlink sockets can be SOCK_RAW sockets or SOCK_DGRAM sockets.

Netlink sockets can be created in the kernel or in userspace; kernel netlink sockets are created by the `netlink_kernel_create()` method; and userspace netlink sockets are created by the `socket()` system call. Creating a netlink socket from userspace or from the kernel creates a `netlink_sock` object. When the socket is created from userspace, it is handled by the `netlink_create()` method. When the socket is created in the kernel, it is handled by `__netlink_kernel_create()`; this method sets the NETLINK_KERNEL_SOCKET flag. Eventually both methods call `__netlink_create()` to allocate a socket in the common way (by calling the `sk_alloc()` method) and initialize it. Figure 2-1 shows how a netlink socket is created in the kernel and in userspace.

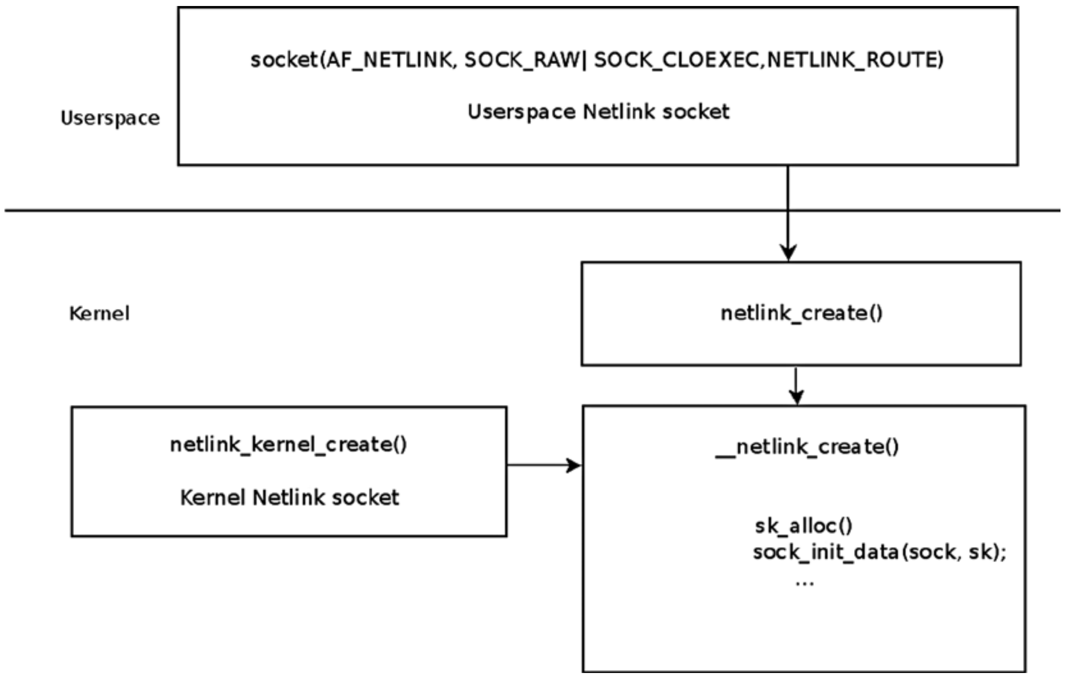


Figure 2-1. *Creating a netlink socket in the kernel and in userspace*

You can create a netlink socket from userspace in a very similar way to ordinary BSD-style sockets, like this, for example: `socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE)`. Then you should create a `sockaddr_nl` object (instance of the netlink socket address structure), initialize it, and use the standard BSD sockets API (such as `bind()`, `sendmsg()`, `recvmsg()`, and so on). The `sockaddr_nl` structure represents a netlink socket address in userspace or in the kernel.

Netlink socket libraries provide a convenient API to netlink sockets. I discuss them in the next section.

Netlink Sockets Libraries

I recommend you use the `libnl` API to develop userspace applications, which send or receive data by netlink sockets. The `libnl` package is a collection of libraries providing APIs to the netlink protocol-based Linux kernel interfaces. The `iproute2` package uses the `libnl` library, as mentioned. Besides the core library (`libnl`), it includes support for the generic netlink family (`libnl-genl`), routing family (`libnl-route`), and netfilter family (`libnl-nf`). The package was developed mostly by Thomas Graf (www.infradead.org/~tgr/libnl/). I should mention here also that there is a library called libmnl, which is a minimalistic userspace library oriented to netlink developers. The libmnl library was mostly written by Pablo Neira Ayuso, with contributions from Jozsef Kadlecsik and Jan Engelhardt. (http://netfilter.org/projects/libmnl/).

The sockaddr_nl Structure

Let's take a look at the sockaddr_nl structure, which represents a netlink socket address:

```
struct sockaddr_nl {
    __kernel_sa_family_t    nl_family;    /* AF_NETLINK              */
    unsigned short          nl_pad;       /* zero                    */
    __u32                   nl_pid;       /* port ID                 */
    __u32                   nl_groups;    /* multicast groups mask   */
};
```

(include/uapi/linux/netlink.h)

- nl_family: Should always be AF_NETLINK.
- nl_pad: Should always be 0.
- nl_pid: The unicast address of a netlink socket. For kernel netlink sockets, it should be 0. Userspace applications sometimes set the nl_pid to be their process id (pid). In a userspace application, when you set nl_pid explicitly to 0, or don't set it at all, and afterwards call bind(), the kernel method netlink_autobind() assigns a value to nl_pid. It tries to assign the process id of the current thread. If you're creating two sockets in userspace, then you are responsible that their nl_pids are unique in case you don't call bind. Netlink sockets are not used only for networking; other subsystems, such as SELinux, audit, uevent, and others, use netlink sockets. The rtnelink sockets are netlink sockets specifically used for networking; they are used for routing messages, neighbouring messages, link messages, and more networking subsystem messages.
- nl_groups: The multicast group (or multicast group mask).

The next section discusses the iproute2 and the older net-tools packages. The iproute2 package is based upon netlink sockets, and you'll see an example of using netlink sockets in iproute2 in the section "Adding and deleting a routing entry in a routing table", later in this chapter. I mention the net-tools package, which is older and might be deprecated in the future, to emphasize that as an alternative to iproute2, it has less power and less abilities.

Userspace Packages for Controlling TCP/IP Networking

There are two userspace packages for controlling TCP/IP networking and handling network devices: net-tools and iproute2. The iproute2 package includes commands like the following:

- ip: For management of network tables and network interfaces
- tc: For traffic control management
- ss: For dumping socket statistics
- lnstat: For dumping linux network statistics
- bridge: For management of bridge addresses and devices

The iproute2 package is based mostly on sending requests to the kernel from userspace and getting replies back over netlink sockets. There are a few exceptions where IOCTLs are used in iproute2. For example, the ip tuntap command uses IOCTLs to add/remove a TUN/TAP device. If you look at the TUN/TAP software driver code, you'll find that it defines some IOCTL handlers, but it does not use the rtnetlink sockets. The net-tools package is based on IOCTLs and includes known commands like these:

- ifconifg
- arp
- route
- netstat
- hostname
- rarp

Some of the advanced functionalities of the iproute2 package are not available in the net-tools package.

The next section discusses kernel netlink sockets—the core engine of handling communication between userspace and the kernel by exchanging netlink messages of different types. Learning about kernel netlink sockets is essential for understanding the interface that the netlink layer provides to userspace.

Kernel Netlink Sockets

You create several netlink sockets in the kernel networking stack. Each kernel socket handles messages of different types: so for example, the netlink socket, which should handle NETLINK_ROUTE messages, is created in rtnetlink_net_init():

```
static int __net_init rtnetlink_net_init(struct net *net) {
    ...
    struct netlink_kernel_cfg cfg = {
        .groups     = RTNLGRP_MAX,
        .input        = rtnetlink_rcv,
        .cb_mutex    = &rtnl_mutex,
        .flags        = NL_CFG_F_NONROOT_RECV,
    };

    sk = netlink_kernel_create(net, NETLINK_ROUTE, &cfg);
    ...
}
```

Note that the rtnetlink socket is aware of network namespaces; the network namespace object (struct net) contains a member named rtnl (rtnetlink socket). In the rtnetlink_net_init() method, after the rtnetlink socket was created by calling netlink_kernel_create(), it is assigned to the rtnl pointer of the corresponding network namespace object.

Let's look in `netlink_kernel_create()` prototype:

```
struct sock *netlink_kernel_create(struct net *net, int unit, struct netlink_kernel_cfg *cfg)
```

- The first parameter (`net`) is the network namespace.
- The second parameter is the netlink protocol (for example, NETLINK_ROUTE for rtnetlink messages, or NETLINK_XFRM for IPsec or NETLINK_AUDIT for the audit subsystem). There are over 20 netlink protocols, but their number is limited by 32 (MAX_LINKS). This is one of the reasons for creating the generic netlink protocol, as you'll see later in this chapter. The full list of netlink protocols is in `include/uapi/linux/netlink.h`.
- The third parameter is a reference to `netlink_kernel_cfg`, which consists of optional parameters for the netlink socket creation:

```
struct netlink_kernel_cfg {
    unsigned int    groups;
    unsigned int    flags;
    void        (*input)(struct sk_buff *skb);
    struct mutex    *cb_mutex;
    void        (*bind)(int group);
};
(include/uapi/linux/netlink.h)
```

The `groups` member is for specifying a multicast group (or a mask of multicast groups). It's possible to join a multicast group by setting `nl_groups` of the `sockaddr_nl` object (you can also do this with the `nl_join_groups()` method of `libnl`). However, in this way you are limited to joining only 32 groups. Since kernel version 2.6.14, you can use the NETLINK_ADD_MEMBERSHIP/ NETLINK_DROP_MEMBERSHIP socket option to join/leave a multicast group, respectively. Using the socket option enables you to join a much higher number of groups. The `nl_socket_add_memberships()/nl_socket_drop_membership()` methods of `libnl` use this socket option.

The `flags` member can be NL_CFG_F_NONROOT_RECV or NL_CFG_F_NONROOT_SEND.

When CFG_F_NONROOT_RECV is set, a non-superuser can bind to a multicast group; in `netlink_bind()` there is the following code:

```
static int netlink_bind(struct socket *sock, struct sockaddr *addr,
                        int addr_len)
 {
  ...
  if (nladdr->nl_groups) {
         if (!netlink_capable(sock, NL_CFG_F_NONROOT_RECV))
                        return -EPERM;
    }
```

For a non-superuser, if the NL_CFG_F_NONROOT_RECV is not set, then when binding to a multicast group the `netlink_capable()` method will return 0, and you get –EPRM error.

When the NL_CFG_F_NONROOT_SEND flag is set, a non-superuser is allowed to send multicasts.

The `input` member is for a callback; when the `input` member in `netlink_kernel_cfg` is NULL, the kernel socket won't be able to receive data from userspace (sending data from the kernel to userspace is possible, though). For the rtnetlink kernel socket, the `rtnetlink_rcv()` method was declared to be the `input` callback; as a result, data sent from userspace over the rtnelink socket will be handled by the `rtnetlink_rcv()` callback.

For uevent kernel events, you need only to send data from the kernel to userspace; so, in lib/kobject_uevent.c, you have an example of a netlink socket where the input callback is undefined:

```
static int uevent_net_init(struct net *net)
{
    struct uevent_sock *ue_sk;
    struct netlink_kernel_cfg cfg = {
        .groups    = 1,
        .flags    = NL_CFG_F_NONROOT_RECV,
    };

    ...
    ue_sk->sk = netlink_kernel_create(net, NETLINK_KOBJECT_UEVENT, &cfg);
    ...
}
(lib/kobject_uevent.c)
```

The mutex (cb_mutex) in the netlink_kernel_cfg object is optional; when not defining a mutex, you use the default one, cb_def_mutex (an instance of a mutex structure; see net/netlink/af_netlink.c). In fact, most netlink kernel sockets are created without defining a mutex in the netlink_kernel_cfg object. For example, the uevent kernel netlink socket (NETLINK_KOBJECT_UEVENT), mentioned earlier. Also, the audit kernel netlink socket (NETLINK_AUDIT) and other netlink sockets don't define a mutex. The rtnetlink socket is an exception—it uses the rtnl_mutex. Also the generic netlink socket, discussed in the next section, defines a mutex of its own: genl_mutex.

The netlink_kernel_create() method makes an entry in a table named nl_table by calling the netlink_insert() method. Access to the nl_table is protected by a read write lock named nl_table_lock; lookup in this table is done by the netlink_lookup() method, specifying the protocol and the port id. Registration of a callback for a specified message type is done by rtnl_register(); there are several places in the networking kernel code where you register such callbacks. For example, in rtnetlink_init() you register callbacks for some messages, like RTM_NEWLINK (creating a new link), RTM_DELLINK (deleting a link), RTM_GETROUTE (dumping the route table), and more. In net/core/neighbour.c, you register callbacks for RTM_NEWNEIGH messages (creating a new neighbour), RTM_DELNEIGH (deleting a neighbour), RTM_GETNEIGHTBL message (dumping the neighbour table), and more. I discuss these actions in depth in Chapters 5 and 7. You also register callbacks to other types of messages in the FIB code (ip_fib_init()), in the multicast code (ip_mr_init()), in the IPv6 code, and in other places.

The first step you should take to work with a netlink kernel socket is to register it. Let's take a look at the rtnl_register() method prototype:

```
extern void rtnl_register(int protocol, int msgtype,
                  rtnl_doit_func,
                  rtnl_dumpit_func,
                  rtnl_calcit_func);
```

The first parameter is the protocol family (when you don't aim at a specific protocol, it is PF_UNSPEC); you'll find a list of all the protocol families in include/linux/socket.h.

The second parameter is the netlink message type, like RTM_NEWLINK or RTM_NEWNEIGH. These are private netlink message types which the rtnelink protocol added. The full list of message types is in include/uapi/linux/rtnetlink.h.

The last three parameters are callbacks: doit, dumpit, and calcit. The callbacks are the actions you want to perform for handling the message, and you usually specify only one callback.

The doit callback is for actions like addition/deletion/modification; the dumpit callback is for retrieving information, and the calcit callback is for calculation of buffer size. The rtnetlink module has a table named rtnl_msg_handlers. This table is indexed by protocol number. Each entry in the table is a table in itself, indexed by message type. Each element in the table is an instance of rtnl_link, which is a structure that consists of pointers for these three callbacks. When registering a callback with rtnl_register(), you add the specified callback to this table.

Registering a callback is done like this, for example: `rtnl_register(PF_UNSPEC, RTM_NEWLINK, rtnl_newlink, NULL, NULL) in net/core/rtnetlink.c`. This adds `rtnl_newlink` as the `doit` callback for RTM_NEWLINK messages in the corresponding `rtnl_msg_handlers` entry.

Sending of rtnelink messages is done with `rtmsg_ifinfo()`. For example, in `dev_open()` you create a new link, so you call: `rtmsg_ifinfo(RTM_NEWLINK, dev, IFF_UP|IFF_RUNNING);` in the `rtmsg_ifinfo()` method, first the `nlmsg_new()` method is called to allocate an `sk_buff` with the proper size. Then two objects are created: the netlink message header (`nlmsghdr`) and an `ifinfomsg` object, which is located immediately after the netlink message header. These two objects are initialized by the `rtnl_fill_ifinfo()` method. Then `rtnl_notify()` is called to send the packet; sending the packet is actually done by the generic netlink method, `nlmsg_notify()` (in `net/netlink/af_netlink.c`). Figure 2-2 shows the stages of sending rtnelink messages with the `rtmsg_ifinfo()` method.

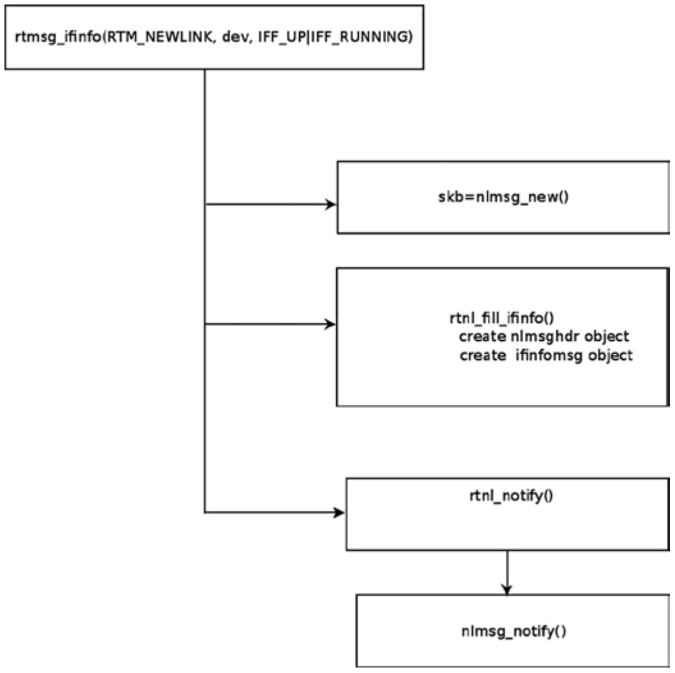


Figure 2-2. *Sending of rtnelink messages with the `rtmsg_ifinfo()` method*

The next section is about netlink messages, which are exchanged between userspace and the kernel. A netlink message always starts with a netlink message header, so your first step in learning about netlink messages will be to study the netlink message header format.

The Netlink Message Header

A netlink message should obey a certain format, specified in RFC 3549, "Linux Netlink as an IP Services Protocol", section 2.2, "Message Format." A netlink message starts with a fixed size netlink header, and after it there is a payload. This section describes the Linux implementation of the netlink message header.

The netlink message header is defined by `struct nlmsghdr` in `include/uapi/linux/netlink.h`:

```
struct nlmsghdr
{
  __u32 nlmsg_len;
  __u16 nlmsg_type;
  __u16 nlmsg_flags;
  __u32 nlmsg_seq;
  __u32 nlmsg_pid;
};
(include/uapi/linux/netlink.h)
```

Every netlink packet starts with a netlink message header, which is represented by `struct nlmsghdr`. The length of `nlmsghdr` is 16 bytes. It contains five fields:

- `nlmsg_len` is the length of the message including the header.
- `nlmsg_type` is the message type; there are four basic netlink message header types:
    - NLMSG_NOOP: No operation, message must be discarded.
    - NLMSG_ERROR: Error occurred.
    - NLMSG_DONE: A multipart message is terminated.
    - NLMSG_OVERRUN: Overrun notification: error, data was lost.

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

      However, families can add netlink message header types of their own. For example, the rtnetlink protocol family adds message header types such as RTM_NEWLINK, RTM_DELLINK, RTM_NEWROUTE, and a lot more (see `include/uapi/linux/rtnetlink.h`). For a full list of the netlink message header types that were added by the rtnelink family with detailed explanation on each, see: `man 7 rtnetlink`. Note that message type values smaller than NLMSG_MIN_TYPE (0x10) are reserved for control messages and may not be used.
- `nlmsg_flags` field can be as follows:
    - NLM_F_REQUEST: When it's a request message.
    - NLM_F_MULTI: When it's a multipart message. Multipart messages are used for table dumps. Usually the size of messages is limited to a page (PAGE_SIZE). So large messages are divided into smaller ones, and each of them (except the last one) has the NLM_F_MULTI flag set. The last message has the NLMSG_DONE flag set.
    - NLM_F_ACK: When you want the receiver of the message to reply with ACK. Netlink ACK messages are sent by the `netlink_ack()` method (`net/netlink/af_netlink.c`).
    - NLM_F_DUMP: Retrieve information about a table/entry.
    - NLM_F_ROOT: Specify the tree root.
    - NLM_F_MATCH: Return all matching entries.
    - NLM_F_ATOMIC: This flag is deprecated.

      The following flags are modifiers for creation of an entry:
    - NLM_F_REPLACE: Override existing entry.
    - NLM_F_EXCL: Do not touch entry, if it exists.
    - NLM_F_CREATE: Create entry, if it does not exist.

- NLM_F_APPEND: Add entry to end of list.
- NLM_F_ECHO: Echo this request.

  I've shown the most commonly used flags. For a full list, see `include/uapi/linux/netlink.h`.

- `nlmsg_seq` is the sequence number (for message sequences). Unlike some Layer 4 transport protocols, there is no strict enforcement of the sequence number.
- `nlmsg_pid` is the sending port id. When a message is sent from the kernel, the `nlmsg_pid` is 0. When a message is sent from userspace, the `nlmsg_pid` can be set to be the process id of that userspace application which sent the message.

  Figure 2-3 shows the netlink message header.

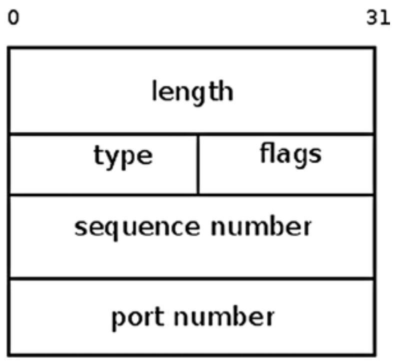


Figure 2-3. *nlmsg header*

After the header comes the payload. The payload of netlink messages is composed of a set of attributes which are represented in Type-Length-Value (TLV) format. With TLV, the type and length are fixed in size (typically 1–4 bytes), and the value field is of variable size. The TLV representation is used also in other places in the networking code—for example, in IPv6 (see RFC 2460). TLV provides flexibility which makes future extensions easier to implement. Attributes can be nested, which enables complex tree structures of attributes.

Each netlink attribute header is defined by `struct nlattr`:

```
struct nlattr {
   __u16   nla_len;
   __u16   nla_type;
};
(include/uapi/linux/netlink.h)
```

- `nla_len`: The size of the attribute in bytes.
- `nla_type`: The attribute type. The value of `nla_type` can be, for example, NLA_U32 (for a 32-bit unsigned integer), NLA_STRING for a variable length string, NLA_NESTED for a nested attribute, NLA_UNSPEC for arbitrary type and length, and more. You can find the list of available types in `include/net/netlink.h`.

Every netlink attribute must be aligned by a 4-byte boundary (NLA_ALIGNTO).

Each family can define an attribute validation policy, which represents the expectations regarding the received attributes. This validation policy is represented by the nla_policy object. In fact, the nla_policy struct has exactly the same content as struct nlattr:

```
 struct nla_policy {
   u16  type;
   u16  len;
};
(include/uapi/linux/netlink.h)
```

The attribute validation policy is an array of nla_policy objects; this array is indexed by the attribute number. For each attribute (except the fixed-length attributes), if the value of len in the nla_policy object is 0, no validation should be performed. If the attribute is one of the string types (such as NLA_STRING), len should be the maximum length of the string, without the terminating NULL byte. If the attribute type is NLA_UNSPEC or unknown, len should be set to the exact length of the attribute's payload. If the attribute type is NLA_FLAG, len is unused. (The reason is that the presence of the attribute itself implies a value of true, and the absence of the attribute implies a value of false).

Receiving a generic netlink message in the kernel is handled by genl_rcv_msg(). In case it is a dump request (when the NLM_F_DUMP flag is set), you dump the table by calling the netlink_dump_start() method. If it's not a dump request, you parse the payload by the nlmsg_parse() method. The nlmsg_parse() method performs attribute validation by calling validate_nla() (lib/nlattr.c). If there are attributes with a type exceeding maxtype, they will be silently ignored for backwards compatibility. In case validation fails, you don't continue to the next step in genl_rcv_msg() (which is running the doit() callback), and the genl_rcv_msg() returns an error code.

The next section describes the NETLINK_ROUTE messages, which are the most commonly used messages in the networking subsystem.

NETLINK_ROUTE Messages

The rtnetlink (NETLINK_ROUTE) messages are not limited to the networking routing subsystem: there are neighbouring subsystem messages as well, interface setup messages, firewalling message, netlink queuing messages, policy routing messages, and many other types of rtnetlink messages, as you'll see in later chapters.

The NETLINK_ROUTE messages can be divided into families:

- LINK (network interfaces)
- ADDR (network addresses)
- ROUTE (routing messages)
- NEIGH (neighbouring subsystem messages)
- RULE (policy routing rules)
- QDISC (queueing discipline)
- TCLASS (traffic classes)
- ACTION (packet action API, see net/sched/act_api.c)
- NEIGHTBL (neighbouring table)
- ADDRLABEL (address labeling)

Each of these families has three types of messages: for creation, deletion, and retrieving information. So, for routing messages, you have the RTM_NEWROUTE message type for creating a route, the RTM_DELROUTE message type for deleting a route, and the RTM_GETROUTE message type for retrieving a route. With LINK messages there is, apart from the three methods for creation, deletion and information retrieval, an additional message for modifying a link: RTM_SETLINK.

There are cases in which an error occurs, and you send an error message as a reply. The netlink error message is represented by the `nlmsgerr struct`:

```
struct nlmsgerr {
    int        error;
    struct nlmsghdr msg;
};
(include/uapi/linux/netlink.h)
```

In fact, as you can see in Figure 2-4, the netlink error message is built from a netlink message header and an error code. When the error code is not 0, the netlink message header of the original request which caused the error is appended after the error code field.

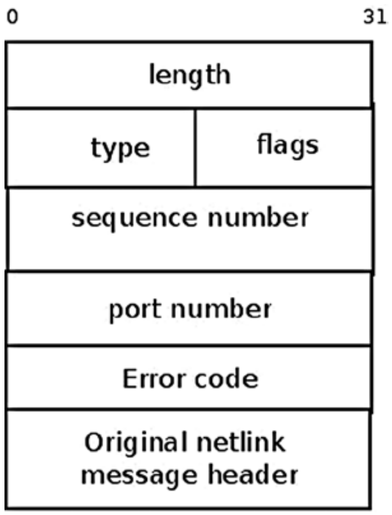


Figure 2-4. *Netlink error message*

If you send a message that was constructed erroneously (for example, the `nlmsg_type` is not valid) then a netlink error message is sent back, and the error code is set according to the error that occurred. For example, when the `nlmsg_type` is not valid (a negative value, or a value higher than the maximum value permitted) the error code is set to –EOPNOTSUPP. See the `rtnetlink_rcv_msg()` method in `net/core/rtnetlink.c`. In error messages, the sequence number is set to be the sequence number of the request that caused the error.

The sender can request to get an ACK for a netlink message. This is done by setting the netlink message header type (`nlmsg_type`) to be NLM_F_ACK. When the kernel sends an ACK, it uses an error message (the netlink message header type of this message is set to be NLMSG_ERROR) with an error code of 0. In this case, the original netlink header of the request is not appended to the error message. For implementation details, see the `netlink_ack()` method implementation in `net/netlink/af_netlink.c`.

After learning about NETLINK_ROUTE messages, you're ready to look at an example of adding and deleting a routing entry in a routing table using NETLINK_ROUTE messages.

Adding and Deleting a Routing Entry in a Routing Table

Behind the scenes, let's see what happens in the kernel in the context of netlink protocol when adding and deleting a routing entry. You can add a routing entry to the routing table by running, for example, the following:

```
ip route add 192.168.2.11 via 192.168.2.20
```

This command sends a netlink message from userspace (RTM_NEWROUTE) over an rtnetlink socket for adding a routing entry. The message is received by the rtnetlink kernel socket and handled by the `rtnetlink_rcv()` method. Eventually, adding the routing entry is done by invoking `inet_rtm_newroute()` in `net/ipv4/fib_frontend.c`. Subsequently, insertion into the Forwarding Information Base (FIB), which is the routing database, is accomplished with the `fib_table_insert()` method; however, inserting into the routing table is not the only task of `fib_table_insert()`. You should notify all listeners who performed registration for RTM_NEWROUTE messages. How? When inserting a new routing entry, you call the `rtmsg_fib()` method with RTM_NEWROUTE. The `rtmsg_fib()` method builds a netlink message and sends it by calling `rtnl_notify()` to notify all listeners who are registered to the RTNLGRP_IPV4_ROUTE group. These RTNLGRP_IPV4_ROUTE listeners can be registered in the kernel as well as in userspace (as is done in `iproute2`, or in some userspace routing daemons, like `xorp`). You'll see shortly how userspace daemons of `iproute2` can subscribe to various rtnelink multicast groups.

When deleting a routing entry, something quite similar happens. You can delete the routing entry earlier by running the following:

```
ip route del 192.168.2.11
```

That command sends a netlink message from userspace (RTM_DELROUTE) over an rtnetlink socket for deleting a routing entry. The message is again received by the rtnetlink kernel socket and handled by the `rtnetlink_rcv()` callback. Eventually, deleting the routing entry is done by invoking `inet_rtm_delroute()` callback in `net/ipv4/fib_frontend.c`. Subsequently, deletion from the FIB is done with `fib_table_delete()`, which calls `rtmsg_fib()`, this time with the RTM_DELROUTE message.

You can monitor networking events with `iproute2 ip` command like this:

```
ip monitor route
```

For example, if you open one terminal and run `ip monitor route` there, and then open another terminal and run `ip route add 192.168.1.10 via 192.168.2.200`, on the first terminal you'll see this line: `192.168.1.10 via 192.168.2.200 dev em1`. And when you run, on the second terminal, `ip route del 192.168.1.10`, on the first terminal the following text will appear: `Deleted 192.168.1.10 via 192.168.2.200 dev em1`.

Running `ip monitor route` runs a daemon that opens a netlink socket and subscribes to the RTNLGRP_IPV4_ROUTE multicast group. Now, adding/deleting a route, as done in this example, will result in this: the message that was sent with `rtnl_notify()` will be received by the daemon and displayed on the terminal.

You can subscribe to other multicast groups in this way. For example, to subscribe to the RTNLGRP_LINK multicast group, run `ip monitor link`. This daemon receives netlink messages from the kernel—when adding/deleting a link, for example. So if you open one terminal and run `ip monitor link`, and then open another terminal and add a VLAN interface by `vconfig add eth1 200`, on the first terminal you'll see lines like this:

```
4: eth1.200@eth1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN
    link/ether 00:e0:4c:53:44:58 brd ff:ff:ff:ff:ff:ff
```

And if you will add a bridge on the second terminal by `brctl addbr mybr`, on the first terminal you'll see lines like this:

```
5: mybr: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN
    link/ether a2:7c:be:62:b5:b6 brd ff:ff:ff:ff:ff:ff
```

You've seen what a netlink message is and how it is created and handled. You've seen how netlink sockets are handled. Next you'll learn why the generic netlink family (introduced in kernel 2.6.15) was created, and you'll learn about its Linux implementation.

Generic Netlink Protocol

One of the drawbacks of the netlink protocol is that the number of protocol families is limited to 32 (MAX_LINKS). This is one of the main reasons that the generic netlink family was created—to provide support for adding a higher number of families. It acts as a netlink multiplexer and works with a single netlink family (`NETLINK_GENERIC`). The generic netlink protocol is based on the netlink protocol and uses its API.

To add a netlink protocol family, you should add a protocol family definition in `include/linux/netlink.h`. But with generic netlink protocol, there is no need for that. The generic netlink protocol is also intended to be used in other subsystems besides networking, because it provides a general purpose communication channel. For example, it's used also by the acpi subsystem (see the definition of `acpi_event_genl_family` in `drivers/acpi/event.c`), by the task stats code (see `kernel/taskstats.c`), by the thermal events code, and more.

The generic netlink kernel socket is created by the `netlink_kernel_create()` method like this:

```
static int __net_init genl_pernet_init(struct net *net) {
    ..
           struct netlink_kernel_cfg cfg = {
                 .input          = genl_rcv,
                 .cb_mutex       = &genl_mutex,
                 .flags          = NL_CFG_F_NONROOT_RECV,
         };
         net->genl_sock = netlink_kernel_create(net, NETLINK_GENERIC, &cfg);
 ...
 }
(net/netlink/genetlink.c)
```

Note that, like the netlink sockets described earlier, the generic netlink socket is also aware of network namespaces; the network namespace object (`struct net`) contains a member named `genl_sock` (a generic netlink socket). As you can see, the network namespace `genl_sock` pointer is assigned in the `genl_pernet_init()` method.

The `genl_rcv()` method is defined to be the `input` callback of the `genl_sock` object, which was created earlier by the `genl_pernet_init()` method. As a result, data sent from userspace over generic netlink sockets is handled in the kernel by the `genl_rcv()` callback.

You can create a generic netlink userspace socket with the `socket()` system call, though it is better to use the `libnl-genl` API (discussed later in this section).

Immediately after creating the generic netlink kernel socket, register the controller family (`genl_ctrl`):

```
static struct genl_family genl_ctrl = {
         .id = GENL_ID_CTRL,
         .name = "nlctrl",
         .version = 0x2,
         .maxattr = CTRL_ATTR_MAX,
         .netnsok = true,
};

static int __net_init genl_pernet_init(struct net *net) {
...
err = genl_register_family_with_ops(&genl_ctrl, &genl_ctrl_ops, 1)
...
```

The genl_ctrl has a fixed id of 0x10 (GENL_ID_CTRL); it is in fact the only instance of genl_family that's initialized with a fixed id; all other instances are initialized with GENL_ID_GENERATE as an id, which subsequently is replaced by a dynamically assigned value.

There is support for registering multicast groups in generic netlink sockets by defining a genl_multicast_group object and calling genl_register_mc_group(); for example, in the Near Field Communication (NFC) subsystem, you have the following:

```
static struct genl_multicast_group nfc_genl_event_mcgrp = {
         .name = NFC_GENL_MCAST_EVENT_NAME,
 };

int __init nfc_genl_init(void)
{
...
 rc = genl_register_mc_group(&nfc_genl_family, &nfc_genl_event_mcgrp);
...
}
(net/nfc/netlink.c)
```

The name of a multicast group should be unique, because it is the primary key for lookups.

In the multicast group, the id is also generated dynamically when registering a multicast group by calling the find_first_zero_bit() method in genl_register_mc_group(). There is only one multicast group, the notify_grp, that has a fixed id, GENL_ID_CTRL.

To work with generic netlink sockets in the kernel, you should do the following:

- Create a genl_family object and register it by calling genl_register_family().
- Create a genl_ops object and register it by calling genl_register_ops().

Alternatively, you can call genl_register_family_with_ops() and pass to it a genl_family object, an array of genl_ops, and its size. This method will first call genl_register_family() and then, if successful, will call genl_register_ops() for each genl_ops element of the specified array of genl_ops.

The genl_register_family() and genl_register_ops() as well as the genl_family and genl_ops are defined in include/net/genetlink.h.

The wireless subsystem uses generic netlink sockets:

```
int nl80211_init(void)
{
    int err;
    err = genl_register_family_with_ops(&nl80211_fam,
        nl80211_ops, ARRAY_SIZE(nl80211_ops));
...
}
(net/wireless/nl80211.c)
```

The generic netlink protocol is used by some userspace packages, such as the hostapd package and the iw package. The hostapd package (http://hostap.epitest.fi) provides a userspace daemon for wireless access point and authentication servers. The iw package is for manipulating wireless devices and their configuration (see http://wireless.kernel.org/en/users/Documentation/iw).

The iw package is based on nl80211 and the libnl library. Chapter 12 discusses nl80211 in more detail. The old userspace wireless package is called wireless-tools and is based on sending IOCTLs.

Here are the genl_family and genl_ops definitions in nl80211:

```
static struct genl_family nl80211_fam = {
    .id         = GENL_ID_GENERATE, /* don't bother with a hardcoded ID */
    .name       = "nl80211",    /* have users key off the name instead */
    .hdrsize    = 0,        /* no private header */
    .version    = 1,        /* no particular meaning now */
    .maxattr    = NL80211_ATTR_MAX,
    .netnsok    = true,
    .pre_doit   = nl80211_pre_doit,
    .post_doit  = nl80211_post_doit,
};
```

- name: Must be a unique name.
- id: id is GENL_ID_GENERATE in this case, which is in fact 0. GENL_ID_GENERATE tells the generic netlink controller to assign the channel a unique channel number when you register the family with genl_register_family(). The genl_register_family() assigns an id in the range 16 (GENL_MIN_ID, which is 0x10) to 1023 (GENL_MAX_ID).
- hdrsize: Size of a private header.
- maxattr: NL80211_ATTR_MAX, which is the maximum number of attributes supported.

  The nl80211_policy validation policy array has NL80211_ATTR_MAX elements (each attribute has an entry in the array):
- netnsok: true, which means the family can handle network namespaces.
- pre_doit: A hook that's called before the doit() callback.
- post_doit: A hook that can, for example, undo locking or any required private tasks after the doit() callback.

  You can add a command or several commands with the genl_ops structure. Let's take a look at the definition of genl_ops struct and then at its usage in nl80211:

```
struct genl_ops {
   u8                       cmd;
   u8                       internal_flags;
   unsigned int             flags;
   const struct nla_policy *policy;
   int                     (*doit)(struct sk_buff *skb,
                                    struct genl_info *info);
   int                    (*dumpit)(struct sk_buff *skb,
                                      struct netlink_callback *cb);
   int                    (*done)(struct netlink_callback *cb);
   struct list_head        ops_list;
};
```

- cmd: Command identifier (the genl_ops struct defines a single command and its doit/dumpit handlers).
- internal_flags: Private flags which are defined and used by the family. For example, in nl80211, there are many operations that define internal flags (such as NL80211_FLAG_NEED_NETDEV_UP, NL80211_FLAG_NEED_RTNL, and more). The nl80211 pre_doit() and post_doit() callbacks perform actions according to these flags. See net/wireless/nl80211.
- flags: Operation flags. Values can be the following:
    - GENL_ADMIN_PERM: When this flag is set, it means that the operation requires the CAP_NET_ADMIN privilege; see the genl_rcv_msg() method in net/netlink/genetlink.c.
    - GENL_CMD_CAP_DO: This flag is set if the genl_ops struct implements the doit() callback.
    - GENL_CMD_CAP_DUMP: This flag is set if the genl_ops struct implements the dumpit() callback.
    - GENL_CMD_CAP_HASPOL: This flag is set if the genl_ops struct defines attribute validation policy (nla_policy array).
- policy : Attribute validation policy is discussed later in this section when describing the payload.
- doit: Standard command callback.
- dumpit: Callback for dumping.
- done: Completion callback for dumps.
- ops_list: Operations list.

```
static struct genl_ops nl80211_ops[] = {
    {

    ...
      {
        .cmd = NL80211_CMD_GET_SCAN,
        .policy = nl80211_policy,
        .dumpit = nl80211_dump_scan,
      },
    ...
}
```

Note that either a doit or a dumpit callback must be specified for every element of genl_ops (nl80211_ops in this case) or the function will fail with -EINVAL.

This entry in genl_ops adds the nl80211_dump_scan() callback as a handler of the NL80211_CMD_GET_SCAN command. The nl80211_policy is an array of nla_policy objects and defines the expected datatype of the attributes and their length.

When running a scan command from userspace, for example by iw dev wlan0 scan, you send from userspace a generic netlink message whose command is NL80211_CMD_GET_SCAN over a generic netlink socket. Messages are sent by the nl_send_auto_complete() method or by nl_send_auto() in the newer libnl versions. nl_send_auto() fills the missing bits and pieces in the netlink message header. If you don't require any of the automatic message completion functionality, you can use nl_send() directly.

The message is handled by the nl80211_dump_scan() method, which is the dumpit callback for this command (net/wireless/nl80211.c). There are more than 50 entries in the nl80211_ops object for handling commands, including NL80211_CMD_GET_INTERFACE, NL80211_CMD_SET_INTERFACE, NL80211_CMD_START_AP, and so on.

To send commands to the kernel, a userspace application should know the family id. The family name is known in the userspace, but the family id is unknown in the userspace because it's determined only in runtime in the kernel. To get the family id, the userspace application should send a generic netlink CTRL_CMD_GETFAMILY request to the kernel. This request is handled by the ctrl_getfamily() method. It returns the family id as well as other information, such as the operations the family supports. Then the userspace can send commands to the kernel specifying the family id that it got in the reply. I discuss this more in the next section.

Creating and Sending Generic Netlink Messages

A generic netlink message starts with a netlink header, followed by the generic netlink message header, and then there is an optional user specific header. Only after all that do you find the optional payload, as you can see in Figure 2-5.

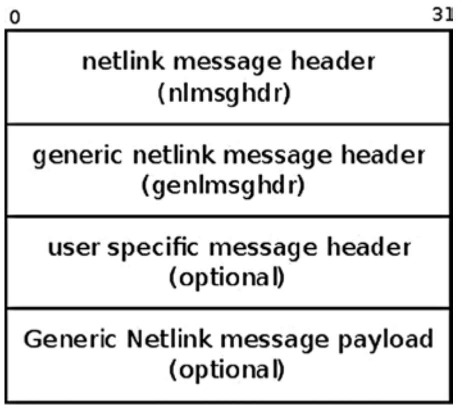

0
31
netlink message header (nlmsghdr)
generic netlink message header (genlmsghdr)
user specific message header (optional)
Generic Netlink message payload (optional)

Figure 2-5. *Generic netlink message.*

This is the generic netlink message header:

```
struct genlmsghdr {
   __u8    cmd;
   __u8    version;
  __u16    reserved;
};
(include/uapi/linux/genetlink.h)
```

- `cmd` is a generic netlink message type; each generic family that you register adds its own commands. For example, for the `nl80211_fam` family mentioned above, the commands it adds (like NL80211_CMD_GET_INTERFACE) are represented by the `nl80211_commands enum`. There are more than 60 commands (see `include/linux/nl80211.h`).
- `version` can be used for versioning support. With `nl80211` it is 1, with no particular meaning. The version member allows changing the format of a message without breaking backward compatibility.
- `reserved` is for future use.

Allocating a buffer for a generic netlink message is done by the following method:

```
sk_buff *genlmsg_new(size_t payload, gfp_t flags)
```

This is in fact a wrapper around `nlmsg_new()`.

After allocating a buffer with `genlmsg_new()`, the `genlmsg_put()` is called to create the generic netlink header, which is an instance of `genlmsghdr`. You send a unicast generic netlink message with `genlmsg_unicast()`, which is in fact a wrapper around `nlmsg_unicast()`. You can send a multicast generic netlink message in two ways:

- `genlmsg_multicast()`: This method sends the message to the default network namespace, `net_init`.
- `genlmsg_multicast_allns()`: This method sends the message to all network namespaces.

(All prototypes of the methods mentioned in this section are in `include/net/genetlink.h`.)

You can create a generic netlink socket from userspace like this: `socket(AF_NETLINK, SOCK_RAW, NETLINK_GENERIC)`; this call is handled in the kernel by the `netlink_create()` method, like an ordinary, non-generic netlink socket, as you saw in the previous section. You can use the socket API to perform further calls like `bind()` and `sendmsg()` or `recvmsg()`; however, using the `libnl` library instead is recommended.

`libnl-genl` provides generic netlink API, for management of controller, family, and command registration. With `libnl-genl`, you can call `genl_connect()` to create a local socket file descriptor and bind the socket to the NETLINK_GENERIC netlink protocol.

Let's take a brief look at what happens in a short typical userspace-kernel session when sending a command to the kernel via generic netlink sockets using the `libnl` library and the `libnl-genl` library.

The `iw` package uses the `libnl-genl` library. When you run a command like `iw dev wlan0 list`, the following sequence occurs (omitting unimportant details):

```
state->nl_sock = nl_socket_alloc()
```

Allocate a socket (note the use here of `libnl` core API and not the generic netlink family (`libnl-genl`) yet.

```
genl_connect(state->nl_sock)
```

Call `socket()` with NETLINK_GENERIC and call `bind()` on this socket; the `genl_connect()` is a method of the `libnl-genl` library.

```
genl_ctrl_resolve(state->nl_sock, "nl80211");
```

This method resolves the generic netlink family name (`"nl80211"`) to the corresponding numeric family identifier. The userspace application must send its subsequent messages to the kernel, specifying this id.

The `genl_ctrl_resolve()` method calls `genl_ctrl_probe_by_name()`, which in fact sends a generic netlink message to the kernel with the CTRL_CMD_GETFAMILY command.

In the kernel, the generic netlink controller ("nlctrl") handles the CTRL_CMD_GETFAMILY command by the ctrl_getfamily() method and returns the family id to userspace. This id was generated when the socket was created.

■ **Note** You can get various parameters (such as generated id, header size, max attributes, and more) of all the registered generic netlink families with the userspace tool genl (of iproute2) by running genl ctrl list.

You're now ready to learn about the socket monitoring interface, which lets you get information about sockets. The socket monitoring interface is used in userspace tools like ss, which displays socket information and statistics for various socket types, and in other projects, as you'll see in the next section.

Socket Monitoring Interface

The sock_diag netlink sockets provide a netlink-based subsystem that can be used to get information about sockets. This feature was added to the kernel to support checkpoint/restore functionality for Linux in userspace (CRIU). To support this functionality, additional data about sockets was needed. For example, /procfs doesn't say which are the peers of a UNIX domain socket (AF_UNIX), and this info is needed for checkpoint/restore support. This additional data is not exported via /proc, and to make changes to procfs entries isn't always desirable because it might break userspace applications. The sock_diag netlink sockets give an API which enables access to this additional data. This API is used in the CRIU project as well as in the ss util. Without the sock_diag, after *checkpointing* a process (saving the state of a process to the filesystem), you can't reconstruct its UNIX domain sockets because you don't know who the peers are.

To support the monitoring interface used by the ss tool, a netlink-based kernel socket is created (NETLINK_SOCK_DIAG). The ss tool, which is part of the iproute2 package, enables you to get socket statistics in a similar way to netstat. It can display more TCP and state information than other tools.

You create a netlink kernel socket for sock_diag like this:

```
static int __net_init diag_net_init(struct net *net)
{
    struct netlink_kernel_cfg cfg = {
        .input    = sock_diag_rcv,
    };

    net->diag_nlsk = netlink_kernel_create(net, NETLINK_SOCK_DIAG, &cfg);
    return net->diag_nlsk == NULL ? -ENOMEM : 0;
}
(net/core/sock_diag.c)
```

The sock_diag module has a table of sock_diag_handler objects named sock_diag_handlers. This table is indexed by the protocol number (for the list of protocol numbers, see include/linux/socket.h).

The sock_diag_handler struct is very simple:

```
struct sock_diag_handler {
__u8 family;
int (*dump)(struct sk_buff *skb, struct nlmsghdr *nlh);
};
(net/core/sock_diag.c)
```

Each protocol that wants to add a socket monitoring interface entry to this table first defines a handler and then calls `sock_diag_register()`, specifying its handler. For example, for UNIX sockets, there is the following in `net/unix/diag.c`:

The first step is definition of the handler:

```
static const struct sock_diag_handler unix_diag_handler = {
    .family = AF_UNIX,
    .dump = unix_diag_handler_dump,
};
```

The second step is registration of the handler:

```
static int __init unix_diag_init(void)
{
    return sock_diag_register(&unix_diag_handler);
}
```

Now, with `ss -x` or `ss --unix`, you can dump the statistics that are gathered by the UNIX `diag` module. In quite a similar way, there are `diag` modules for other protocols, such as UDP (`net/ipv4/udp_diag.c`), TCP (`net/ipv4/tcp_diag.c`), DCCP (`/net/dccp/diag.c`), and AF_PACKET (`net/packet/diag.c`).

There's also a `diag` module for the netlink sockets themselves. The `/proc/net/netlink` entry provides information about the netlink socket (`netlink_sock` object) like the `portid`, `groups`, the inode number of the socket, and more. If you want the details, dumping `/proc/net/netlink` is handled by `netlink_seq_show()` in `net/netlink/af_netlink.c`. There are some `netlink_sock` fields which `/proc/net/netlink` doesn't provide—for example, `dst_group` or `dst_portid` or groups above 32. For this reason, the netlink socket monitoring interface was added (`net/netlink/diag.c`). You should be able to use the `ss` tool of `iproute2` to read netlink sockets information. The netlink `diag` code can be built also as a kernel module.

Summary

This chapter covered netlink sockets, which provide a mechanism for bidirectional communication between the userspace and the kernel and are widely used by the networking subsystem. You've seen some examples of netlink sockets usage. I also discussed netlink messages, how they're created and handled. Another important subject the chapter dealt with is the generic netlink sockets, including their advantages and their usage. The next chapter covers the ICMP protocol, including its usage and its implementation in IPv4 and IPv6.

Quick Reference

I conclude this chapter with a short list of important methods of the netlink and generic netlink subsystems. Some of them were mentioned in this chapter:

int netlink_rcv_skb(struct sk_buff *skb, int (*cb)(struct sk_buff *, struct nlmsghdr *))

This method handles receiving netlink messages. It's called from the input callback of netlink families (for example, in the `rtnetlink_rcv()` method for the rtnetlink family, or in the `sock_diag_rcv()` method for the `sock_diag` family. The method performs sanity checks, like making sure that the length of the netlink message header does not exceed the permitted max length (NLMSG_HDRLEN). It also avoids invoking the specified callback in case that the message is a control message. In case the ACK flag (NLM_F_ACK) is set, it sends an error message by invoking the `netlink_ack()` method.

struct sk_buff *netlink_alloc_skb(struct sock *ssk, unsigned int size, u32 dst_portid, gfp_t gfp_mask)

This method allocates an SKB with the specified size and `gfp_mask`; the other parameters (`ssk`, `dst_portid`) are used when working with memory mapped netlink IO (NETLINK_MMAP). This feature is not discussed in this chapter, and is located here: `net/netlink/af_netlink.c`.

struct netlink_sock *nlk_sk(struct sock *sk)

This method returns the `netlink_sock` object, which has an `sk` as a member, and is located here: `net/netlink/af_netlink.h`.

struct sock *netlink_kernel_create(struct net *net, int unit, struct netlink_kernel_cfg *cfg)

This method creates a kernel netlink socket.

struct nlmsghdr *nlmsg_hdr(const struct sk_buff *skb)

This method returns the netlink message header pointed to by `skb->data`.

struct nlmsghdr *__nlmsg_put(struct sk_buff *skb, u32 portid, u32 seq, int type, int len, int flags)

This method builds a netlink message header according to the specified parameters, and puts it in the `skb`, and is located here: `include/linux/netlink.h`.

struct sk_buff *nlmsg_new(size_t payload, gfp_t flags)

This method allocates a new netlink message with the specified message payload by calling `alloc_skb()`. If the specified payload is 0, `alloc_skb()` is called with `NLMSG_HDRLEN` (after alignment with the NLMSG_ALIGN macro).

int nlmsg_msg_size(int payload)

This method returns the length of a netlink message (message header length and payload), not including padding.

void rtnl_register(int protocol, int msgtype, rtnl_doit_func doit, rtnl_dumpit_func dumpit, rtnl_calcit_func calcit)

This method registers the specified rtnetlink message type with the three specified callbacks.

static int rtnetlink_rcv_msg(struct sk_buff *skb, struct nlmsghdr *nlh)

This method processes an rtnetlink message.

static int rtnl_fill_ifinfo(struct sk_buff *skb, struct net_device *dev, int type, u32 pid, u32 seq, u32 change, unsigned int flags, u32 ext_filter_mask)

This method creates two objects: a netlink message header (`nlmsghdr`) and an `ifinfomsg` object, located immediately after the netlink message header.

void rtnl_notify(struct sk_buff *skb, struct net *net, u32 pid, u32 group, struct nlmsghdr *nlh, gfp_t flags)

This method sends an rtnetlink message.

int genl_register_mc_group(struct genl_family *family, struct genl_multicast_group *grp)

This method registers the specified multicast group, notifies the userspace, and returns 0 on success or a negative error code. The specified multicast group must have a name. The multicast group id is generated dynamically in this method by the `find_first_zero_bit()` method for all multicast groups, except for `notify_grp`, which has a fixed id of 0x10 (GENL_ID_CTRL).

void genl_unregister_mc_group(struct genl_family *family, struct genl_multicast_group *grp)

This method unregisters the specified multicast group and notifies the userspace about it. All current listeners on the group are removed. It's not necessary to unregister all multicast groups before unregistering the family—unregistering the family causes all assigned multicast groups to be unregistered automatically.

int genl_register_ops(struct genl_family *family, struct genl_ops *ops)

This method registers the specified operations and assigns them to the specified family. Either a `doit()` or a `dumpit()` callback must be specified or the operation will fail with -EINVAL. Only one operation structure per command identifier may be registered. It returns 0 on success or a negative error code.

int genl_unregister_ops(struct genl_family *family, struct genl_ops *ops)

This method unregisters the specified operations and unassigns them from the specified family. The operation blocks until the current message processing has finished and doesn't start again until the unregister process has finished. It's not necessary to unregister all operations before unregistering the family—unregistering the family causes all assigned operations to be unregistered automatically. It returns 0 on success or a negative error code.

int genl_register_family(struct genl_family *family)

This method registers the specified family after validating it first. Only one family may be registered with the same family name or identifier. The family id may equal GENL_ID_GENERATE, causing a unique id to be automatically generated and assigned.

int genl_register_family_with_ops(struct genl_family *family, struct genl_ops *ops, size_t n_ops)

This method registers the specified family and operations. Only one family may be registered with the same family name or identifier. The family id may equal GENL_ID_GENERATE, causing a unique id to be automatically generated and assigned. Either a `doit` or a `dumpit` callback must be specified for every registered operation or the function will fail. Only one operation structure per command identifier may be registered. This is equivalent to calling `genl_register_family()` followed by `genl_register_ops()` for every operation entry in the table, taking care to unregister the family on the error path. The method returns 0 on success or a negative error code.

int genl_unregister_family(struct genl_family *family)

This method unregisters the specified family and returns 0 on success or a negative error code.

void *genlmsg_put(struct sk_buff *skb, u32 portid, u32 seq, struct genl_family *family, int flags, u8 cmd)

This method adds a generic netlink header to a netlink message.

int genl_register_family(struct genl_family *family) int genl_unregister_family(struct genl_family *family)

This method registers/unregisters a generic netlink family.

int genl_register_ops(struct genl_family *family, struct genl_ops *ops) int genl_unregister_ops(struct genl_family *family, struct genl_ops *ops)

This method registers/unregisters generic netlink operations.

void genl_lock(void) void genl_unlock(void)

This method locks/unlocks the generic netlink mutex (`genl_mutex`). Used for example in `net/l2tp/l2tp_netlink.c`.

CHAPTER 3

Internet Control Message Protocol (ICMP)

Chapter 2 discusses the netlink sockets implementation and how netlink sockets are used as a communication channel between the kernel and userspace. This chapter deals with the ICMP protocol, which is a Layer 4 protocol. Userspace applications can use the ICMP protocol (to send and receive ICMP packets) by using the sockets API (the best-known example is probably the `ping` utility). This chapter discusses how these ICMP packets are handled in the kernel and gives some examples.

The ICMP protocol is used primarily as a mandatory mechanism for sending error and control messages about the network layer (L3). The protocol enables getting feedback about problems in the communication environment by sending ICMP messages. These messages provide error handling and diagnostics. The ICMP protocol is relatively simple but is very important for assuring correct system behavior. The basic definition of ICMPv4 is in RFC 792, "Internet Control Message Protocol." This RFC defines the goals of the ICMPv4 protocol and the format of various ICMPv4 messages. I also mention in this chapter RFC 1122 ("Requirements for Internet Hosts—Communication Layers") which defines some requirements about several ICMP messages; RFC 4443, which defines the ICMPv6 protocol; and RFC 1812, which defines requirements for routers. I also describe which types of ICMPv4 and ICMPv6 messages exist, how they are sent, and how they are processed. I cover ICMP sockets, including why they were added and how they are used. Keep in mind that the ICMP protocol is also used for various security attacks; for example, the Smurf Attack is a denial-of-service attack in which large numbers of ICMP packets with the intended victim's spoofed source IP are sent as broadcasts to a computer network using an IP broadcast address.

ICMPv4

ICMPv4 messages can be classified into two categories: error messages and information messages (they are termed "query messages" in RFC 1812). The ICMPv4 protocol is used in diagnostic tools like `ping` and `traceroute`. The famous `ping` utility is in fact a userspace application (from the `iputils` package) which opens a raw socket and sends an ICMP_ECHO message and should get back an ICMP_REPLY message as a response. `Traceroute` is a utility to find the path between a host and a given destination IP address. The `traceroute` utility is based on setting varying values to the Time To Live (TTL), which is a field in the IP header representing the hop count. The `traceroute` utility takes advantage of the fact that a forwarding machine will send back an ICMP_TIME_EXCEED message when the TTL of the packet reaches 0. The `traceroute` utility starts by sending messages with a TTL of 1, and with each received ICMP_DEST_UNREACH with code ICMP_TIME_EXCEED as a reply, it increases the TTL by 1 and sends again to the same destination. It uses the returned ICMP "Time Exceeded" messages to build a list of the routers that the packets traverse, until the destination is reached and returns an ICMP "Echo Reply" message. Traceroute uses the UDP protocol by default. The ICMPv4 module is `net/ipv4/icmp.c`. Note that ICMPv4 cannot be built as a kernel module.

ICMPv4 Initialization

ICMPv4 initialization is done in the `inet_init()` method, which is invoked in boot phase. The `inet_init()` method invokes the `icmp_init()` method, which in turn calls the `icmp_sk_init()` method to create a kernel ICMP socket for sending ICMP messages and to initialize some ICMP `procfs` variables to their default values. (You will encounter some of these `procfs` variables later in this chapter.)

Registration of the ICMPv4 protocol, like registration of other IPv4 protocols, is done in `inet_init()`:

```
static const struct net_protocol icmp_protocol = {
    .handler        =  icmp_rcv,
    .err_handler    =  icmp_err,
    .no_policy      =  1,
    .netns_ok       =  1,
};
```

(`net/ipv4/af_inet.c`)

- `icmp_rcv`: The `handler` callback. This means that for incoming packets whose protocol field in the IP header equals IPPROTO_ICMP (0x1), `icmp_rcv()` will be invoked.
- `no_policy`: This flag is set to 1, which implies that there is no need to perform IPsec policy checks; for example, the `xfrm4_policy_check()` method is not called in `ip_local_deliver_finish()` because the `no_policy` flag is set.
- `netns_ok`: This flag is set to 1, which indicates that the protocol is aware of network namespaces. Network namespaces are described in Appendix A, in the `net_device` section. The `inet_add_protocol()` method will fail for protocols whose `netns_ok` field is 0 with an error of `-EINVAL`.

```
static int __init inet_init(void) {
. . .
    if (inet_add_protocol(&icmp_protocol, IPPROTO_ICMP) < 0)
        pr_crit("%s: Cannot add ICMP protocol\n", __func__);
. . .

int __net_init icmp_sk_init(struct net *net)
{
    . . .
    for_each_possible_cpu(i) {
        struct sock *sk;

        err = inet_ctl_sock_create(&sk, PF_INET,
                        SOCK_RAW, IPPROTO_ICMP, net);
        if (err < 0)
            goto fail;

                net->ipv4.icmp_sk[i] = sk;
              . . .
                sock_set_flag(sk, SOCK_USE_WRITE_QUEUE);
        inet_sk(sk)->pmtudisc = IP_PMTUDISC_DONT;
    }
    . . .

}
```

In the `icmp_sk_init()` method, a raw ICMPv4 socket is created for each CPU and is kept in an array. The current `sk` can be accessed with the `icmp_sk(struct net *net)` method. These sockets are used in the `icmp_push_reply()` method. The ICMPv4 `procfs` entries are initialized in the `icmp_sk_init()` method; I mention them in this chapter and summarize them in the "Quick Reference" section at the end of this chapter. Every ICMP packet starts with an ICMPv4 header. Before discussing how ICMPv4 messages are received and transmitted, the following section describes the ICMPv4 header, so that you better understand how ICMPv4 messages are built.

ICMPv4 Header

The ICMPv4 header consists of type (8 bits), code (8 bits), and checksum (16 bits), and a 32 bits variable part member (its content varies based on the ICMPv4 type and code), as you can see in Figure 3-1. After the ICMPv4 header comes the payload, which should include the IPv4 header of the originating packet and a part of its payload. According to RFC 1812, it should contain as much of the original datagram as possible without the length of the ICMPv4 datagram exceeding 576 bytes. This size is in accordance to RFC 791, which specifies that "All hosts must be prepared to accept datagrams of up to 576 octets."

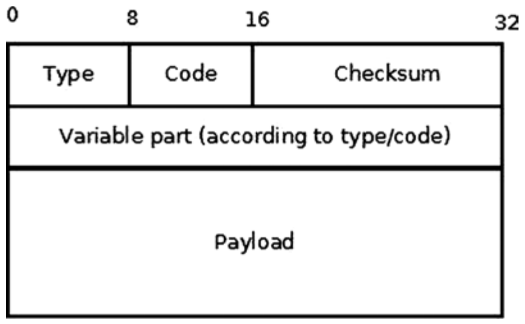


Figure 3-1. *The ICMPv4 header*

The ICMPv4 header is represented by `struct icmphdr`:

```
struct icmphdr {
  __u8        type;
  __u8        code;
  __sum16    checksum;
  union {
    struct {
        __be16    id;
        __be16    sequence;
    } echo;
    __be32    gateway;
    struct {
        __be16    __unused;
        __be16    mtu;
    } frag;
  } un;
};
```

(include/uapi/linux/icmp.h)

You'll find the current complete list of assigned ICMPv4 message type numbers and codes at www.iana.org/assignments/icmp-parameters/icmp-parameters.xml.

The ICMPv4 module defines an array of `icmp_control` objects, named `icmp_pointers`, which is indexed by ICMPv4 message type. Let's take a look at the `icmp_control` structure definition and at the `icmp_pointers` array:

```
struct icmp_control {
    void (*handler)(struct sk_buff *skb);
    short error;        /* This ICMP is classed as an error message */
};

static const struct icmp_control icmp_pointers[NR_ICMP_TYPES+1];
```

NR_ICMP_TYPES is the highest ICMPv4 type, which is 18.

(include/uapi/linux/icmp.h)

The error field of the `icmp_control` objects of this array is 1 only for error message types, like the "Destination Unreachable" message (ICMP_DEST_UNREACH), and it is 0 (implicitly) for information messages, like echo (ICMP_ECHO). Some handlers are assigned to more than one type. Next I discuss handlers and the ICMPv4 message types they manage.

`ping_rcv()` handles receiving a ping reply (ICMP_ECHOREPLY). The `ping_rcv()` method is implemented in the ICMP sockets code, `net/ipv4/ping.c`. In kernels prior to 3.0, in order to send ping, you had to create a raw socket in userspace. When receiving a reply to a ping (ICMP_ECHOREPLY message), the raw socket that sent the ping processed it. In order to understand how this is implemented, let's take a look in `ip_local_deliver_finish()`, which is the method which handles incoming IPv4 packets and passes them to the sockets which should process them:

```
static int ip_local_deliver_finish(struct sk_buff *skb)
{
    . . .
        int protocol = ip_hdr(skb)->protocol;
        const struct net_protocol *ipprot;
        int raw;

    resubmit:
        raw = raw_local_deliver(skb, protocol);
        ipprot = rcu_dereference(inet_protos[protocol]);
            if (ipprot != NULL) {
                    int ret;
                    . . .
                    ret = ipprot->handler(skb);
                    . . .
```

(net/ipv4/ip_input.c)

When the `ip_local_deliver_finish()` method receives an ICMP_ECHOREPLY packet, it first tries to deliver it to a listening raw socket, which will process it. Because a raw socket that was opened in userspace handles the ICMP_ECHOREPLY message, there is no need to do anything further with it. So when the `ip_local_deliver_finish()` method receives ICMP_ECHOREPLY, the `raw_local_deliver()` method is invoked first to process it by a raw socket, and afterwards the `ipprot->handler(skb)` is invoked (this is the `icmp_rcv()` callback in the case of ICMPv4 packet). And because the packet was already processed by a raw socket, there is nothing more to do with it. So the packet is discarded silently by calling the `icmp_discard()` method, which is the handler for ICMP_ECHOREPLY messages.

When the ICMP sockets ("ping sockets") were integrated into the Linux kernel in kernel 3.0, this was changed. Ping sockets are discussed in the "ICMP Sockets ("Ping Sockets")" section later in this chapter. In this context I should note that with ICMP sockets, the sender of ping can be also *not a raw socket*. For example, you can create a socket like this: `socket (PF_INET, SOCK_DGRAM, PROT_ICMP)` and use it to send ping packets. This socket is not a raw socket. As a result, the echo reply is not delivered to any raw socket, since there is no corresponding raw socket which listens. To avoid this problem, the ICMPv4 module handles receiving ICMP_ECHOREPLY messages with the `ping_rcv()` callback. The ping module is located in the IPv4 layer (`net/ipv4/ping.c`). Nevertheless, most of the code in `net/ipv4/ping.c` is a dual-stack code (intended for both IPv4 and IPv6). As a result, the `ping_rcv()` method also handles ICMPV6_ECHO_REPLY messages for IPv6 (see `icmpv6_rcv()` in `net/ipv6/icmp.c`). I talk more about ICMP sockets later in this chapter.

`icmp_discard()` is an empty handler used for nonexistent message types (message types whose numbers are without corresponding declarations in the header file) and for some messages that do not need any handling, for example ICMP_TIMESTAMPREPLY. The ICMP_TIMESTAMP and the ICMP_TIMESTAMPREPLY messages are used for time synchronization; the sender sends the `originate timestamp` in an ICMP_TIMESTAMP request; the receiver sends ICMP_TIMESTAMPREPLY with three timestamps: the originating timestamp which was sent by the sender of the timestamp request, as well as a receive timestamp and a transmit timestamp. There are more commonly used protocols for time synchronization than ICMPv4 timestamp messages, like the Network Time Protocol (NTP). I should also mention the Address Mask request (ICMP_ADDRESS), which is normally sent by a host to a router in order to obtain an appropriate subnet mask. Recipients should reply to this message with an address mask reply message. The ICMP_ADDRESS and the ICMP_ADDRESSREPLY messages, which were handled in the past by the `icmp_address()` method and by the `icmp_address_reply()` method, are now handled also by `icmp_discard()`. The reason is that there are other ways to get the subnet masks, such as with DHCP.

`icmp_unreach()` handles ICMP_DEST_UNREACH, ICMP_TIME_EXCEED, ICMP_PARAMETERPROB, and ICMP_QUENCH message types.

An `ICMP_DEST_UNREACH` message can be sent under various conditions. Some of these conditions are described in the "Sending ICMPv4 Messages: Destination Unreachable" section in this chapter.

An `ICMP_TIME_EXCEEDED` message is sent in two cases:

In `ip_forward()`, each packet decrements its TTL. According to RFC 1700, the recommended TTL for the IPv4 protocol is 64. If the TTL reaches 0, this is indication that the packet should be dropped because probably there was some loop. So, if the TTL reaches 0 in `ip_forward()`, the `icmp_send()` method is invoked:

```
icmp_send(skb, ICMP_TIME_EXCEEDED, ICMP_EXC_TTL, 0);

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

In such a case, an ICMP_TIME_EXCEEDED message with code ICMP_EXC_TTL is sent, the SKB is freed, the `InHdrErrors` SNMP counter (IPSTATS_MIB_INHDRERRORS) is incremented, and the method returns NET_RX_DROP.

In `ip_expire()`, the following occurs when a timeout of a fragment exists:

```
icmp_send(head, ICMP_TIME_EXCEEDED, ICMP_EXC_FRAGTIME, 0);

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

An ICMP_PARAMETERPROB message is sent when parsing the options of an IPv4 header fails, in the `ip_options_compile()` method or in the `ip_options_rcv_srr()` method (`net/ipv4/ip_options.c`). The options are an optional, variable length field (up to 40 bytes) of the IPv4 header. IP options are discussed in Chapter 4.

An ICMP_QUENCH message type is in fact deprecated. According to RFC 1812, section 4.3.3.3 (Source Quench): "A router SHOULD NOT originate ICMP Source Quench messages", and also, "A router MAY ignore any ICMP Source Quench messages it receives." The ICMP_QUENCH message was intended to reduce congestion, but it turned out that this is an ineffective solution.

`icmp_redirect()` handles ICMP_REDIRECT messages; according to RFC 1122, section 3.2.2.2, hosts should not send an ICMP redirect message; redirects are to be sent only by gateways. `icmp_redirect()` handles ICMP_REDIRECT messages. In the past, `icmp_redirect()` called `ip_rt_redirect()`, but an `ip_rt_redirect()`

invocation is not needed anymore as the protocol handlers now all properly propagate the redirect back into the routing code. In fact, in kernel 3.6, the `ip_rt_redirect()` method was removed. So the `icmp_redirect()` method first performs sanity checks and then calls `icmp_socket_deliver()`, which delivers the packet to the raw sockets and invokes the protocol error handler (in case it exists). Chapter 6 discusses ICMP_REDIRECT messages in more depth.

`icmp_echo()` handles echo ("ping") requests (ICMP_ECHO) by sending echo replies (ICMP_ECHOREPLY) with `icmp_reply()`. If case `net->ipv4.sysctl_icmp_echo_ignore_all` is set, a reply will not be sent. For configuring ICMPv4 `procfs` entries, see the "Quick Reference" section at the end of this chapter, and also `Documentation/networking/ip-sysctl.txt`.

`icmp_timestamp()` handles ICMP Timestamp requests (ICMP_TIMESTAMP) by sending ICMP_ TIMESTAMPREPLY with `icmp_reply()`.

Before discussing sending ICMP messages by the `icmp_reply()` method and by the `icmp_send()` method, I should describe the `icmp_bxm` ("ICMP build xmit message") structure, which is used in both methods:

```
struct icmp_bxm {
    struct sk_buff *skb;
    int offset;
    int data_len;

    struct {
        struct icmphdr icmph;
        __be32          times[3];
    } data;
    int head_len;
    struct ip_options_data replyopts;
};
```

- `skb`: For the `icmp_reply()` method, this `skb` is the request packet; the `icmp_param` object (instance of `icmp_bxm`) is built from it (in the `icmp_echo()` method and in the `icmp_timestamp()` method). For the `icmp_send()` method, this `skb` is the one that triggered sending an ICMPv4 message due to some conditions; you will see several examples of such messages in this section.
- `offset`: Difference (offset) between `skb_network_header(skb)` and `skb->data`.
- `data_len`: ICMPv4 packet payload size.
- `icmph`: The ICMP v4 header.
- `times[3]`: Array of three timestamps, filled in `icmp_timestamp()`.
- `head_len`: Size of the ICMPv4 header (in case of `icmp_timestamp()`, there are additional 12 bytes for the timestamps).
- `replyopts`: An `ip_options` data object. IP options are optional fields after the IP header, up to 40 bytes. They enable advanced features like strict routing/loose routing, record routing, time stamping, and more. They are initialized with the `ip_options_echo()` method. Chapter 4 discusses IP options.

Receiving ICMPv4 Messages

The `ip_local_deliver_finish()` method handles packets for the local machine. When getting an ICMP packet, the method delivers the packet to the raw sockets that had performed registration of ICMPv4 protocol. In the `icmp_rcv()` method, first the `InMsgs` SNMP counter (ICMP_MIB_INMSGS) is incremented. Subsequently, the checksum correctness is verified. If the checksum is not correct, two SNMP counters are incremented, `InCsumErrors` and `InErrors` (ICMP_MIB_CSUMERRORS and ICMP_MIB_INERRORS, respectively), the SKB is freed, and the method returns 0. The `icmp_rcv()` method does not return an error in this case. In fact, the `icmp_rcv()` method always returns 0; the reason for returning 0 in case of checksum error is that no special thing should be done when receiving an erroneous ICMP message except to discard it; when a protocol handler returns a negative error, another attempt to process the packet is performed, and it is not needed in this case. For more details, refer to the implementation of the `ip_local_deliver_finish()` method. Then the ICMP header is examined in order to find its type; the corresponding `procfs` message type counter is incremented (each ICMP message type has a `procfs` counter), and a sanity check is performed to verify that it is not higher than the highest permitted value (NR_ICMP_TYPES). According to section 3.2.2 of RFC 1122, if an ICMP message of unknown type is received, it must be silently discarded. So if the message type is out of range, the `InErrors` SNMP counter (ICMP_MIB_INERRORS) is incremented, and the SKB is freed.

In case the packet is a broadcast or a multicast, and it is an ICMP_ECHO message or an ICMP_TIMESTAMP message, there is a check whether broadcast/multicast echo requests are permitted by reading the variable `net->ipv4.sysctl_icmp_echo_ignore_broadcasts`. This variable can be configured via `procfs` by writing to `/proc/sys/net/ipv4/icmp_echo_ignore_broadcasts`, and by default its value is 1. If this variable is set, the packet is dropped silently. This is done according to section 3.2.2.6 of RFC 1122: "An ICMP Echo Request destined to an IP broadcast or IP multicast address MAY be silently discarded." And according to section 3.2.2.8 of this RFC, "An ICMP Timestamp Request message to an IP broadcast or IP multicast address MAY be silently discarded." Then a check is performed to detect whether the type is allowed for broadcast/multicast (ICMP_ECHO, ICMP_TIMESTAMP, ICMP_ADDRESS, and ICMP_ADDRESSREPLY). If it is not one of these message types, the packet is dropped and 0 is returned. Then according to its type, the corresponding entry in the `icmp_pointers` array is fetched and the appropriate handler is called. Let's take a look in the ICMP_ECHO entry in the `icmp_control` dispatch table:

```
static const struct icmp_control icmp_pointers[NR_ICMP_TYPES + 1] = {
...
  [ICMP_ECHO] = {
        .handler = icmp_echo,
    },
...
}
```

So when receiving a ping (the type of the message is "Echo Request," ICMP_ECHO), it is handled by the `icmp_echo()` method. The `icmp_echo()` method changes the type in the ICMP header to be ICMP_ECHOREPLY and sends a reply by calling the `icmp_reply()` method. Apart from `ping`, the only other ICMP message which requires a response is the timestamp message (ICMP_TIMESTAMP); it is handled by the `icmp_timestamp()` method, which, much like in the ICMP_ECHO case, changes the type to ICMP_TIMESTAMPREPLY and sends a reply by calling the `icmp_reply()` method. Sending is done by `ip_append_data()` and by `ip_push_pending_frames()`. Receiving a ping reply (ICMP_ECHOREPLY) is handled by the `ping_rcv()` method.

You can disable replying to pings with the following:

```
echo 1 >  /proc/sys/net/ipv4/icmp_echo_ignore_all
```

There are some callbacks that handle more than one ICMP type. The `icmp_discard()` callback, for example, handles ICMPv4 packets whose type is not handled by the Linux ICMPv4 implementation, and messages like ICMP_TIMESTAMPREPLY, ICMP_INFO_REQUEST , ICMP_ADDRESSREPLY, and more.

Sending ICMPv4 Messages: "Destination Unreachable"

There are two methods for sending an ICMPv4 message: the first is the `icmp_reply()` method, which is sent as a response for two types of ICMP requests, ICMP_ECHO and ICMP_TIMESTAMP. The second one is the `icmp_send()` method, where the local machine initiates sending an ICMPv4 message under certain conditions (described in this section). Both these methods eventually invoke `icmp_push_reply()` for actually sending the packet. The `icmp_reply()` method is called as a response to an ICMP_ECHO message from the `icmp_echo()` method, and as a response to an ICMP_TIMESTAMP message from the `icmp_timestamp()` method. The `icmp_send()` method is invoked from many places in the IPv4 network stack—for example, from netfilter, from the forwarding code (`ip_forward.c`), from tunnels like `ipip` and `ip_gre`, and more.

This section looks into some of the cases when a "Destination Unreachable" message is sent (the type is ICMP_DEST_UNREACH).

Code 2: ICMP_PROT_UNREACH (Protocol Unreachable)

When the protocol of the IP header (which is an 8-bit field) is a nonexistent protocol, an ICMP_DEST_UNREACH/ICMP_PROT_UNREACH is sent back to the sender because there is no protocol handler for such a protocol (the protocol handler array is indexed by the protocol number, so for nonexistent protocols there will be no handler). By *nonexistent* protocol I mean either that because of some error indeed the protocol number of the IPv4 header does not appear in the protocol number list (which you can find in `include/uapi/linux/in.h`, for IPv4), or that the kernel was built without support for that protocol, and, as a result, this protocol is not registered and there is no entry for it in the protocol handlers array. Because such a packet can't be handled, an ICMPv4 message of "Destination Unreachable" should be replied back to the sender; the ICMP_PROT_UNREACH code in the ICMPv4 reply signifies the cause of the error, "protocol is unreachable." See the following:

```
static int ip_local_deliver_finish(struct sk_buff *skb)
  {
    ...
    int protocol = ip_hdr(skb)->protocol;
    const struct net_protocol *ipprot;
    int raw;

resubmit:
    raw = raw_local_deliver(skb, protocol);

    ipprot = rcu_dereference(inet_protos[protocol]);
    if (ipprot != NULL) {
        ...
    } else {
    if (!raw) {
    if (xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
            IP_INC_STATS_BH(net, IPSTATS_MIB_INUNKNOWNPROTOS);
            icmp_send(skb, ICMP_DEST_UNREACH,ICMP_PROT_UNREACH, 0);
              }
        ...
  }
```

(`net/ipv4/ip_input.c`)

In this example, a lookup in the `inet_protos` array by protocol is performed; and because no entry was found, this means that the protocol is not registered in the kernel.

Code 3: ICMP_PORT_UNREACH ("Port Unreachable")

When receiving UDPv4 packets, a matching UDP socket is searched for. If no matching socket is found, the checksum correctness is verified. If it is wrong, the packet is dropped silently. If it is correct, the statistics are updated and a "Destination Unreachable"/"Port Unreachable" ICMP message is sent back:

```
int __udp4_lib_rcv(struct sk_buff *skb, struct udp_table *udptable, int proto)
{
        struct sock *sk;
        ...
        sk = __udp4_lib_lookup_skb(skb, uh->source, uh->dest, udptable)
        ...
        if (sk != NULL) {
        ...
        }

        /* No socket. Drop packet silently, if checksum is wrong */
    if (udp_lib_checksum_complete(skb))
        goto csum_error;

        UDP_INC_STATS_BH(net, UDP_MIB_NOPORTS, proto == IPPROTO_UDPLITE);
        icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PORT_UNREACH, 0);
        ...
        }
...

}
```

(net/ipv4/udp.c)

A lookup is being performed by the `__udp4_lib_lookup_skb()` method, and if there is no socket, the statistics are updated and an ICMP_DEST_UNREACH message with ICMP_PORT_UNREACH code is sent back.

Code 4: ICMP_FRAG_NEEDED

When forwarding a packet with a length larger than the MTU of the outgoing link, if the don't fragment (DF) bit in the IPv4 header (IP_DF) is set, the packet is discarded and an ICMP_DEST_UNREACH message with ICMP_FRAG_NEEDED code is sent back to the sender:

```
int ip_forward(struct sk_buff *skb)
{
        ...
        struct rtable *rt;      /* Route we use */
        ...
        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;
        }
        ...
}
```

(net/ipv4/ip_forward.c)

Code 5: ICMP_SR_FAILED

When forwarding a packet with the strict routing option and gatewaying set, a "Destination Unreachable" message with ICMP_SR_FAILED code is sent back, and the packet is dropped:

```
int ip_forward(struct sk_buff *skb)
 {
         struct ip_options *opt  = &(IPCB(skb)->opt);
         ...
         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;
}
```

(net/ipv4/ip_forward.c)

For a full list of all IPv4 "Destination Unreachable" codes, see Table 3-1 in the "Quick Reference" section at the end of this chapter. Note that a user can configure some rules with the `iptables` REJECT target and the `--reject-with` qualifier, which can send "Destination Unreachable" messages according to the selection; more in the "Quick Reference" section at the end of this chapter.

Both the `icmp_reply()` and the `icmp_send()` methods support rate limiting; they call `icmpv4_xrlim_allow()`, and if the rate limiting check allows sending the packet (the `icmpv4_xrlim_allow()` returns `true`), they send the packet. It should be mentioned here that rate limiting is not performed automatically on all types of traffic. Here are the conditions under which rate limiting check will not be performed:

- The message type is unknown.
- The packet is of PMTU discovery.
- The device is a loopback device.
- The ICMP type is not enabled in the rate mask.

If all these conditions are not matched, rate limiting is performed by calling the `inet_peer_xrlim_allow()` method. You'll find more info about rate mask in the "Quick Reference" section at the end of this chapter.

Let's look inside the `icmp_send()` method. First, this is its prototype:

```
void icmp_send(struct sk_buff *skb_in, int type, int code, __be32 info)
```

skb_in is the SKB which caused the invocation of the icmp_send() method, type and code are the ICMPv4 message type and code, respectively. The last parameter, info, is used in the following cases:

- For the ICMP_PARAMETERPROB message type it is the offset in the IPv4 header where the parsing problem occurred.
- For the ICMP_DEST_UNREACH message type with ICMP_FRAG_NEEDED code, it is the MTU.
- For the ICMP_REDIRECT message type with ICMP_REDIR_HOST code, it is the IP address of the destination address in the IPv4 header of the provoking SKB.

When further looking into the icmp_send() method, first there are some sanity checks. Then multicast/broadcast packets are rejected. A check of whether the packet is a fragment is performed by inspecting the frag_off field of the IPv4 header. If the packet is fragmented, an ICMPv4 message is sent, but only for the first fragment. According to section 4.3.2.7 of RFC 1812, an ICMP error message must not be sent as the result of receiving an ICMP error message. So first a check is performed to find out whether the ICMPv4 message to be sent is an error message, and if it is so, another check is performed to find out whether the provoking SKB contained an error ICMPv4 message, and if so, then the method returns without sending the ICMPv4 message. Also if the type is an unknown ICMPv4 type (higher than NR_ICMP_TYPES), the method returns without sending the ICMPv4 message, though this isn't specified explicitly by the RFC. Then the source address is determined according to the value of net->ipv4.sysctl_icmp_errors_use_inbound_ifaddr value (more details in the "Quick Reference" section at the end of this chapter). Then the ip_options_echo() method is invoked to copy the IP options of the IPv4 header of the invoking SKB. An icmp_bxm object (icmp_param) is being allocated and initialized, and a lookup in the routing subsystem is performed with the icmp_route_lookup() method. Then the icmp_push_reply() method is invoked.

Let's take a look at the icmp_push_reply() method, which actually sends the packet. The icmp_push_reply() first finds the socket on which the packet should be sent by calling:

```
sk = icmp_sk(dev_net((*rt)->dst.dev));
```

The dev_net() method returns the network namespace of the outgoing network device. (The dev_net() method and network namespaces are discussed in chapter 14 and in Appendix A.) Then, the icmp_sk() method fetches the socket (because in SMP there is a socket per CPU). Then the ip_append_data() method is called to move the packet to the IP layer. If the ip_append_data() method fails, the statistics are updated by incrementing the ICMP_MIB_OUTERRORS counter and the ip_flush_pending_frames() method is called to free the SKB. I discuss the ip_append_data() method and the ip_flush_pending_frames() method in Chapter 4.

Now that you know all about ICMPv4, it's time to move on to ICMPv6.

ICMPv6

ICMPv6 has many similarities to ICMPv4 when it comes to reporting errors in the network layer (L3). There are additional tasks for ICMPv6 which are not performed in ICMPv4. This section discusses the ICMPv6 protocol, its new features (which are not implemented in ICMPv4), and the features which are similar. ICMPv6 is defined in RFC 4443. If you delve into ICMPv6 code you will probably encounter, sooner or later, comments that mention RFC 1885. In fact, RFC 1885, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6)," is the base ICMPv6 RFC. It was obsoleted by RFC 2463, which was in turn obsoleted by RFC 4443. The ICMPv6 implementation is based upon IPv4, but it is more complicated; the changes and additions that were added are discussed in this section.

The ICMPv6 protocol has a next header value of 58, according to RFC 4443, section 1 (Chapter 8 discusses IPv6 next headers). ICMPv6 is an integral part of IPv6 and must be fully implemented by every IPv6 node. Apart from error handling and diagnostics, ICMPv6 is used for the Neighbour Discovery (ND) protocol in IPv6, which replaces and enhances functions of ARP in IPv4, and for the Multicast Listener Discovery (MLD) protocol, which is the counterpart of the IGMP protocol in IPv4, shown in Figure 3-2.

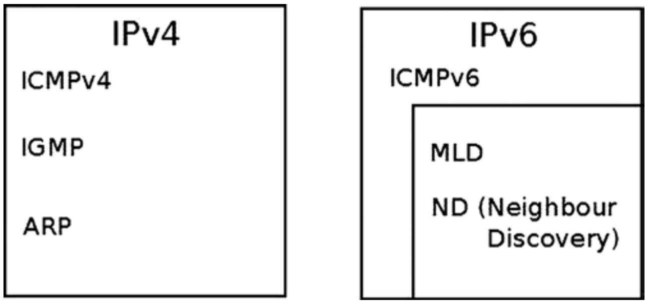


Figure 3-2. *ICMP in IPv4 and IPv6. The counterpart of the IGMP protocol in IPv6 is the MLD protocol, and the counterpart of the ARP protocol in IPv6 is the ND protocol*

This section covers the ICMPv6 implementation. As you will see, it has many things in common with the ICMPv4 implementation in the way messages are handled and sent. There are even cases when the same methods are called in ICMPv4 and in ICMPv6 (for example, `ping_rcv()` and `inet_peer_xrlim_allow()`). There are some differences, and some topics are unique to ICMPv6. The `ping6` and `traceroute6` utilities are based on ICMPv6 and are the counterparts of `ping` and `traceroute` utilities of IPv4 (mentioned in the ICMPv4 section in the beginning of this chapter). ICMPv6 is implemented in `net/ipv6/icmp.c` and in `net/ipv6/ip6_icmp.c`. As with ICMPv4, ICMPv6 cannot be built as a kernel module.

ICMPv6 Initialization

ICMPv6 initialization is done by the `icmpv6_init()` method and by the `icmpv6_sk_init()` method. Registration of the ICMPv6 protocol is done by `icmpv6_init()` (`net/ipv6/icmp.c`):

```
static const struct inet6_protocol icmpv6_protocol = {
         .handler        =       icmpv6_rcv,
         .err_handler    =       icmpv6_err,
         .flags          =       INET6_PROTO_NOPOLICY|INET6_PROTO_FINAL,
 };
```

The `handler` callback is `icmpv6_rcv()`; this means that for incoming packets whose protocol field equals IPPROTO_ICMPV6 (58), `icmpv6_rcv()` will be invoked.

When the INET6_PROTO_NOPOLICY flag is set, this implies that IPsec policy checks should not be performed; for example, the `xfrm6_policy_check()` method is not called in `ip6_input_finish()` because the INET6_PROTO_NOPOLICY flag is set:

```
int __init icmpv6_init(void)
 {
         int err;
         ...
         if (inet6_add_protocol(&icmpv6_protocol, IPPROTO_ICMPV6) < 0)
                 goto fail;
         return 0;
 }
```

```
static int __net_init icmpv6_sk_init(struct net *net)
{
    struct sock *sk;
        ...
    for_each_possible_cpu(i) {
        err = inet_ctl_sock_create(&sk, PF_INET6,
                       SOCK_RAW, IPPROTO_ICMPV6, net);
        ...
        net->ipv6.icmp_sk[i] = sk;
        ...

}
```

As in ICMPv4, a raw ICMPv6 socket is created for each CPU and is kept in an array. The current sk can be accessed by the icmpv6_sk() method.

ICMPv6 Header

The ICMPv6 header consists of type (8 bits), code (8 bits), and checksum (16 bits), as you can see in Figure 3-3.

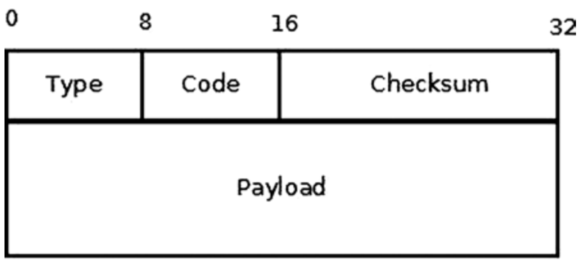


Figure 3-3. *ICMPv6 header*

The ICMPv6 header is represented by struct icmp6hdr:

```
struct icmp6hdr {
    __u8        icmp6_type;
    __u8        icmp6_code;
    __sum16    icmp6_cksum;
    ...
}
```

There is not enough room to show all the fields of struct icmp6hdr because it is too large (it is defined in include/uapi/linux/icmpv6.h). When the high-order bit of the type field is 0 (values in the range from 0 to 127), it indicates an error message; when the high-order bit is 1 (values in the range from 128 to 255), it indicates an information message. Table 3-1 shows the ICMPv6 message types by their number and kernel symbol.

Table 3-1. *ICMPv6 Messages*

Type	Kernel symbol	Error/Info	Description
1	ICMPV6_DEST_UNREACH	Error	Destination Unreachable
2	ICMPV6_PKT_TOOBIG	Error	Packet too big
3	ICMPV6_TIME_EXCEED	Error	Time Exceeded
4	ICMPV6_PARAMPROB	Error	Parameter problem
128	ICMPV6_ECHO_REQUEST	Info	Echo Request
129	ICMPV6_ECHO_REPLY	Info	Echo Reply
130	ICMPV6_MGM_QUERY	Info	Multicast group membership management query
131	ICMPV6_MGM_REPORT	Info	Multicast group membership management report
132	ICMPV6_MGM_REDUCTION	Info	Multicast group membership management reduction
133	NDISC_ROUTER_SOLICITATION	Info	Router solicitation
134	NDISC_ROUTER_ADVERTISEMENT	Info	Router advertisement
135	NDISC_NEIGHBOUR_SOLICITATION	Info	Neighbour solicitation
136	NDISC_NEIGHBOUR_ADVERTISEMENT	Info	Neighbour advertisement
137	NDISC_REDIRECT	Info	Neighbour redirect

The current complete list of assigned ICMPv6 types and codes can be found at `www.iana.org/assignments/icmpv6-parameters/icmpv6-parameters.xml`.

ICMPv6 performs some tasks that are not performed by ICMPv4. For example, Neighbour Discovery is done by ICMPv6, whereas in IPv4 it is done by the ARP/RARP protocols. Multicast group memberships are handled by ICMPv6 in conjunction with the MLD (Multicast Listener Discovery) protocol, whereas in IPv4 this is performed by IGMP (Internet Group Management Protocol). Some ICMPv6 messages are similar in meaning to ICMPv4 messages; for example, ICMPv6 has these messages: “Destination Unreachable,” (ICMPV6_DEST_UNREACH), “Time Exceeded” (ICMPV6_TIME_EXCEED), “Parameter Problem” (ICMPV6_PARAMPROB), “Echo Request” (ICMPV6_ECHO_REQUEST), and more. On the other hand, some ICMPv6 messages are unique to IPv6, such as the NDISC_NEIGHBOUR_SOLICITATION message.

Receiving ICMPv6 Messages

When getting an ICMPv6 packet, it is delivered to the `icmpv6_rcv()` method, which gets only an SKB as a parameter. Figure 3-4 shows the `Rx` path of a received ICMPv6 message.

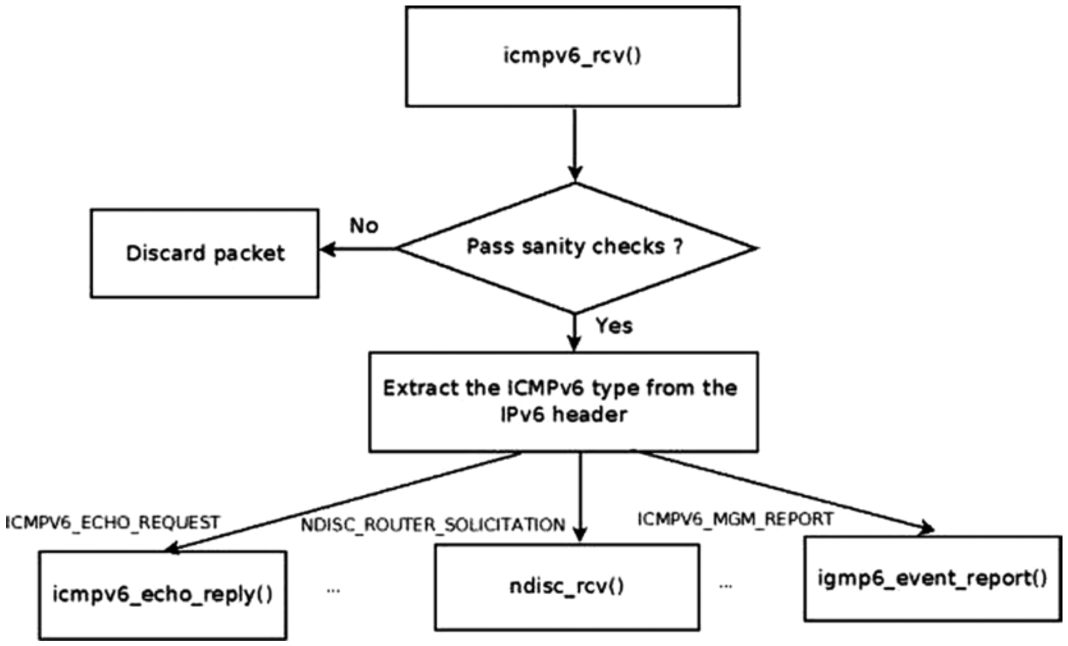


Figure 3-4. *Receive path of ICMPv6 message*

In the `icmpv6_rcv()` method, after some sanity checks, the `InMsgs` SNMP counter (ICMP6_MIB_INMSGS) is incremented. Subsequently, the checksum correctness is verified. If the checksum is not correct, the `InErrors` SNMP counter (ICMP6_MIB_INERRORS) is incremented, and the SKB is freed. The `icmpv6_rcv()` method does not return an error in this case (in fact it always returns 0, much like its IPv4 counterpart, `icmp_rcv()`).Then the ICMPv6 header is read in order to find its type; the corresponding `procfs` message type counter is incremented by the ICMP6MSGIN_INC_STATS_BH macro (each ICMPv6 message type has a `procfs` counter). For example, when receiving ICMPv6 ECHO requests ("pings"), the `/proc/net/snmp6/Icmp6InEchos` counter is incremented, and when receiving ICMPv6 Neighbour Solicitation requests, the `/proc/net/snmp6/Icmp6InNeighborSolicits` counter is incremented.

In ICMPv6, there is no dispatch table like the `icmp_pointers` table in ICMPv4. The handlers are invoked according to the ICMPv6 message type, in a long `switch(type)` command:

- "Echo Request" (ICMPV6_ECHO_REQUEST) is handled by the `icmpv6_echo_reply()` method.
- "Echo Reply" (ICMPV6_ECHO_REPLY) is handled by the `ping_rcv()` method. The `ping_rcv()` method is in the IPv4 `ping` module (`net/ipv4/ping.c`); this method is a dual-stack method (it handles both IPv4 and IPv6—discussed in the beginning of this chapter).
- Packet too big (ICMPV6_PKT_TOOBIG).
    - First a check is done to verify that the data block area (pointed to by `skb->data`) contains a block of data whose size is at least as big as an ICMP header. This is done by the `pskb_may_pull()` method. If this condition is not met, the packet is dropped.
    - Then the `icmpv6_notify()` method is invoked. This method eventually calls the `raw6_icmp_error()` method so that the registered raw sockets will handle the ICMP messages.

- “Destination Unreachable,” “Time Exceeded,” and “Parameter Problem” (ICMPV6_DEST_UNREACH, ICMPV6_TIME_EXCEED, and ICMPV6_PARAMPROB respectively) are also handled by `icmpv6_notify()`.
- Neighbour Discovery (ND) messages:
  - NDISC_ROUTER_SOLICITATION: Messages which are sent usually to the `all-routers` multicast address of FF02::2, and which are answered by router advertisements. (Special IPv6 multicast addresses are discussed in Chapter 8).
  - NDISC_ROUTER_ADVERTISEMENT: Messages which are sent periodically by routers or as an immediate response to router solicitation requests. Router advertisements contain prefixes that are used for on-link determination and/or address configuration, a suggested hop limit value, and so on.
  - NDISC_NEIGHBOUR_SOLICITATION: The counterpart of ARP request in IPv4.
  - NDISC_NEIGHBOUR_ADVERTISEMENT: The counterpart of ARP reply in IPv4.
  - NDISC_REDIRECT: Used by routers to inform hosts of a better first hop for a destination.
  - All the Neighbour Discovery (ND) messages are handled by the neighbour discovery method, `ndisc_rcv()` (`net/ipv6/ndisc.c`). The `ndisc_rcv()` method is discussed in Chapter 7.
- ICMPV6_MGM_QUERY (Multicast Listener Report) is handled by `igmp6_event_query()`.
- ICMPV6_MGM_REPORT (Multicast Listener Report) is handled by `igmp6_event_report()`. Note: Both ICMPV6_MGM_QUERY and ICMPV6_MGM_REPORT are discussed in more detail in Chapter 8.
- Messages of unknown type, and the following messages, are all handled by the `icmpv6_notify()` method:
  - ICMPV6_MGM_REDUCTION: When a host leaves a multicast group, it sends an MLDv2 ICMPV6_MGM_REDUCTION message; see the `igmp6_leave_group()` method in `net/ipv6/mcast.c`.
  - ICMPV6_MLD2_REPORT: MLDv2 Multicast Listener Report packet; usually sent with destination address of the all MLDv2-capable routers Multicast Group Address (FF02::16).
  - ICMPV6_NI_QUERY- ICMP: Node Information Query.
  - ICMPV6_NI_REPLY: ICMP Node Information Response.
  - ICMPV6_DHAAD_REQUEST: ICMP Home Agent Address Discovery Request Message; see section 6.5, RFC 6275, “Mobility Support in IPv6.”
  - ICMPV6_DHAAD_REPLY: ICMP Home Agent Address Discovery Reply Message; See section 6.6, RFC 6275.
  - ICMPV6_MOBILE_PREFIX_SOL: ICMP Mobile Prefix Solicitation Message Format; see section 6.7, RFC 6275.
  - ICMPV6_MOBILE_PREFIX_ADV: ICMP Mobile Prefix Advertisement Message Format; see section 6.8, RFC 6275.

Notice that the `switch(type)` command ends like this:

```
default:
    LIMIT_NETDEBUG(KERN_DEBUG "icmpv6: msg of unknown type\n");

    /* informational */
    if (type & ICMPV6_INFOMSG_MASK)
        break;

    /*
     * error of unknown type.
     * must pass to upper level
     */

    icmpv6_notify(skb, type, hdr->icmp6_code, hdr->icmp6_mtu);
}
```

Informational messages fulfill the condition (`type & ICMPV6_INFOMSG_MASK`), so they are discarded, whereas the other messages which do not fulfill this condition (and therefore should be error messages) are passed to the upper layer. This is done in accordance with section 2.4 ("Message Processing Rules") of RFC 4443.

Sending ICMPv6 Messages

The main method for sending ICMPv6 messages is the `icmpv6_send()` method. The method is called when the local machine initiates sending an ICMPv6 message under conditions described in this section. There is also the `icmpv6_echo_reply()` method, which is called only as a response to an ICMPV6_ECHO_REQUEST ("ping") message. The `icmp6_send()` method is invoked from many places in the IPv6 network stack. This section looks at several examples.

Example: Sending "Hop Limit Time Exceeded" ICMPv6 Messages

When forwarding a packet, every machine decrements the Hop Limit Counter by 1. The Hop Limit Counter is a member of the IPv6 header—it is the IPv6 counterpart to Time To Live in IPv4. When the value of the Hop Limit Counter header reaches 0, an ICMPV6_TIME_EXCEED message is sent with ICMPV6_EXC_HOPLIMIT code by calling the `icmpv6_send()` method, then the statistics are updated and the packet is dropped:

```
int ip6_forward(struct sk_buff *skb)
{
    ...
        if (hdr->hop_limit <= 1) {
                /* Force OUTPUT device used as source address */
                skb->dev = dst->dev;
                icmpv6_send(skb, ICMPV6_TIME_EXCEED, ICMPV6_EXC_HOPLIMIT, 0);
                IP6_INC_STATS_BH(net,
                                 ip6_dst_idev(dst), IPSTATS_MIB_INHDRERRORS);

                kfree_skb(skb);
                return -ETIMEDOUT;
         }
    ...
}
```

(net/ipv6/ip6_output.c)

Example: Sending “Fragment Reassembly Time Exceeded” ICMPv6 Messages

When a timeout of a fragment occurs, an ICMPV6_TIME_EXCEED message with ICMPV6_EXC_FRAGTIME code is sent back, by calling the `icmpv6_send()` method:

```
void ip6_expire_frag_queue(struct net *net, struct frag_queue *fq,
                            struct inet_frags *frags)
 {
        . . .
        icmpv6_send(fq->q.fragments, ICMPV6_TIME_EXCEED, ICMPV6_EXC_FRAGTIME, 0);
        . . .
 }
```

(`net/ipv6/reassembly.c`)

Example: Sending “Destination Unreachable”/“Port Unreachable” ICMPv6 Messages

When receiving UDPv6 packets, a matching UDPv6 socket is searched for. If no matching socket is found, the checksum correctness is verified. If it is wrong, the packet is dropped silently. If it is correct, the statistics (UDP_MIB_NOPORTS MIB counter, which is exported to `procfs` by `/proc/net/snmp6/Udp6NoPorts`) is updated and a “Destination Unreachable”/“Port Unreachable” ICMPv6 message is sent back with `icmpv6_send()`:

```
int __udp6_lib_rcv(struct sk_buff *skb, struct udp_table *udptable, int proto)
{
        ...
       sk = __udp6_lib_lookup_skb(skb, uh->source, uh->dest, udptable);
        if (sk != NULL) {
        ...
        }
        ...
        if (udp_lib_checksum_complete(skb))
                goto discard;

        UDP6_INC_STATS_BH(net, UDP_MIB_NOPORTS, proto == IPPROTO_UDPLITE);
        icmpv6_send(skb, ICMPV6_DEST_UNREACH, ICMPV6_PORT_UNREACH, 0);
        ...

}
```

This case is very similar to the UDPv4 example given earlier in this chapter.

Example: Sending “Fragmentation Needed” ICMPv6 Messages

When forwarding a packet, if its size is larger than the MTU of the outgoing link, and the `local_df` bit in the SKB is not set, the packet is discarded and an ICMPV6_PKT_TOOBIG message is sent back to the sender. The information in this message is used as part of the Path MTU (PMTU) discovery process.

Note that as opposed to the parallel case in IPv4, where an ICMP_DEST_UNREACH message with ICMP_FRAG_NEEDED code is sent, in this case an ICMPV6_PKT_TOOBIG message is sent back, and not a "Destination Unreachable" (ICMPV6_DEST_UNREACH) message. The ICMPV6_PKT_TOOBIG message has a message type number of its own in ICMPv6:

```
int ip6_forward(struct sk_buff *skb)
{
...
         if ((!skb->local_df && skb->len > mtu && !skb_is_gso(skb)) ||
             (IP6CB(skb)->frag_max_size && IP6CB(skb)->frag_max_size > mtu)) {
                 /* Again, force OUTPUT device used as source address */
                 skb->dev = dst->dev;
                 icmpv6_send(skb, ICMPV6_PKT_TOOBIG, 0, mtu);
                 IP6_INC_STATS_BH(net,
                                  ip6_dst_idev(dst), IPSTATS_MIB_INTOOBIGERRORS);
                 IP6_INC_STATS_BH(net,
                                  ip6_dst_idev(dst), IPSTATS_MIB_FRAGFAILS);
                 kfree_skb(skb);
                 return -EMSGSIZE;
         }
...
}

(net/ipv6/ip6_output.c)
```

Example: Sending "Parameter Problem" ICMPv6 Messages

When encountering a problem in parsing extension headers, an ICMPV6_PARAMPROB message with ICMPV6_UNK_OPTION code is sent back:

```
static bool ip6_tlvopt_unknown(struct sk_buff *skb, int optoff) {
        switch ((skb_network_header(skb)[optoff] & 0xC0) >> 6) {
        ...
        case 2: /* send ICMP PARM PROB regardless and drop packet */
                icmpv6_param_prob(skb, ICMPV6_UNK_OPTION, optoff);
                return false;
         }

(net/ipv6/exthdrs.c)
```

The `icmpv6_send()` method supports rate limiting by calling `icmpv6_xrlim_allow()`. I should mention here that, as in ICMPv4, rate limiting is not performed automatically in ICMPv6 on all types of traffic. Here are the conditions under which rate limiting check will not be performed:

- Informational messages
- PMTU discovery
- Loopback device

If all these conditions are not matched, rate limiting is performed by calling the `inet_peer_xrlim_allow()` method, which is shared between ICMPv4 and ICMPv6. Note that unlike IPv4, you can't set a rate mask in IPv6. It is not forbidden by the ICMPv6 spec, RFC 4443, but it was never implemented.

Let's look inside the `icmp6_send()` method. First, this is its prototype:

```
static void icmp6_send(struct sk_buff *skb, u8 type, u8 code, __u32 info)
```

The parameters are similar to those of the `icmp_send()` method of IPv4, so I won't repeat the explanation here. When further looking into the `icmp6_send()` code, you find some sanity checks. Checking whether the provoking message is an ICMPv6 error message is done by calling the `is_ineligible()` method; if it is, the `icmp6_send()` method terminates. The length of the message should not exceed 1280, which is IPv6 minimum MTU (IPV6_MIN_MTU, defined in `include/linux/ipv6.h`). This is done in accordance with RFC 4443, section 2.4 (c), which says that every ICMPv6 error message must include as much of the IPv6 offending (invoking) packet (the packet that caused the error) as possible without making the error message packet exceed the minimum IPv6 MTU. Then the message is passed to the IPv6 layer, by the `ip6_append_data()` method and by the `icmpv6_push_pending_frame()` method, to free the SKB.

Now I'll turn to the `icmpv6_echo_reply()` method; as a reminder, this method is called as a response to an ICMPV6_ECHO message. The `icmpv6_echo_reply()` method gets only one parameter, the SKB. It builds an `icmpv6_msg` object and sets its type to ICMPV6_ECHO_REPLY. Then it passes the message to the IPv6 layer, by the `ip6_append_data()` method and by the `icmpv6_push_pending_frame()` method. If the `ip6_append_data()` method fails, an SNMP counter (ICMP6_MIB_OUTERRORS) is incremented, and `ip6_flush_pending_frames()` is invoked to free the SKB.

Chapters 7 and 8 also discuss ICMPv6. The next section introduces ICMP sockets and the purpose they serve.

ICMP Sockets ("Ping sockets")

A new type of sockets (IPPROTO_ICMP) was added by a patch from the Openwall GNU/*/Linux distribution (Owl), which provides security enhancements over other distributions. The ICMP sockets enable a `setuid-less` "ping." For Openwall GNU/*/Linux, it was the last step on the road to a `setuid-less` distribution. With this patch, a new ICMPv4 ping socket (which is not a raw socket) is created with:

```
socket(PF_INET, SOCK_DGRAM, IPPROTO_ICMP);
```

instead of with:

```
socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);
```

There is also support for IPPROTO_ICMPV6 sockets, which was added later, in `net/ipv6/icmp.c`. A new ICMPv6 ping socket is created with:

```
socket(PF_INET6, SOCK_DGRAM, IPPROTO_ICMPV6);
```

instead of with:

```
socket(PF_INET6, SOCK_RAW, IPPROTO_ICMP6);
```

Similar functionality (non-privileged ICMP) is implemented in Mac OS X; see: `www.manpagez.com/man/4/icmp/`.

Most of the code for ICMP sockets is in `net/ipv4/ping.c`; in fact, large parts of the code in `net/ipv4/ping.c` are dual-stack (IPv4 and IPv6). In `net/ipv6/ping.c` there are only few IPv6-specific bits. Using ICMP sockets is disabled by default. You can enable ICMP sockets by setting the following `procfs` entry: `/proc/sys/net/ipv4/ping_group_range`. It is "1 0" by default, meaning that nobody (not even root) may create ping sockets. So, if you want to allow a user with `uid` and `gid` of 1000 to use the ICMP socket, you should run this from the command line (with root privileges): `echo 1000 1000 > /proc/sys/net/ipv4/ping_group_range`, and then you can `ping` from this user account using ICMP sockets. If you want to set privileges for a user in the system, you should run from the command line echo 0 2147483647 > /proc/sys/net/ipv4/ping_group_range. (2147483647 is the value of GID_T_MAX; see include/net/ping.h.) There are no separate security settings for IPv4 and IPv6; everything is controlled by /proc/sys/net/ipv4/ping_group_range. The ICMP sockets support only ICMP_ECHO for IPv4 or ICMPV6_ECHO_REQUEST for IPv6, and the code of the ICMP message must be 0 in both cases.

The ping_supported() helper method checks whether the parameters for building the ICMP message (both for IPv4 and IPv6) are valid. It is invoked from ping_sendmsg():

```
static inline int ping_supported(int family, int type, int code)
{
    return (family == AF_INET && type == ICMP_ECHO && code == 0) ||
           (family == AF_INET6 && type == ICMPV6_ECHO_REQUEST && code == 0);
}
```

(net/ipv4/ping.c)

ICMP sockets export the following entries to procfs: /proc/net/icmp for IPv4 and /proc/net/icmp6 for IPv6.

For more info about ICMP sockets see http://openwall.info/wiki/people/segoon/ping and http://lwn.net/Articles/420799/.

Summary

This chapter covered the implementation of ICMPv4 and ICMPv6. You learned about the ICMP header format of both protocols and about receiving and sending messages with both protocols. The new features of ICMPv6, which you will encounter in upcoming chapters, were also discussed. The Neighbouring Discovery protocol, which uses ICMPv6 messages, is discussed in Chapter 7, and the MLD protocol, which also uses ICMPv6 messages, is covered in Chapter 8. The next chapter, Chapter 4, talks about the implementation of the IPv4 network layer.

In the “Quick Reference” section that follows, I cover the top methods related to the topics discussed in this chapter, ordered by their context. Then two tables mentioned in the chapter, some important relevant procfs entries and a short section about ICMP messages usage in iptables reject rules are all covered.

Quick Reference

I conclude this chapter with a short list of important methods of ICMPv4 and ICMPv6, 6 tables, a section about procfs entries, and a short section about using a reject target in iptables and ip6tables to create ICMP “Destination Unreachable” messages.

Methods

The following methods were covered in this chapter.

int icmp_rcv(struct sk_buff *skb);

This method is the main handler for processing incoming ICMPv4 packets.

extern void icmp_send(struct sk_buff *skb_in, int type, int code, __be32 info);

This method sends an ICMPv4 message. The parameters are the provoking SKB, ICMPv4 message type, ICMPv4 message code, and info (which is dependent on type).

struct icmp6hdr *icmp6_hdr(const struct sk_buff *skb);

This method returns the ICMPv6 header, which the specified `skb` contains.

void icmpv6_send(struct sk_buff *skb, u8 type, u8 code, __u32 info);

This method sends an ICMPv6 message. The parameters are the provoking SKB, ICMPv6 message type, ICMPv6 message code, and `info` (which is dependent on type).

void icmpv6_param_prob(struct sk_buff *skb, u8 code, int pos);

This method is a convenient version of the `icmp6_send()` method, which all it does is call `icmp6_send()` with ICMPV6_PARAMPROB as a type, and with the other specified parameters, `skb`, `code` and `pos`, and frees the SKB afterwards.

Tables

The following tables were covered in this chapter.

Table 3-2. *ICMPv4 "Destination Unreachable" (ICMP_DEST_UNREACH) Codes*

Code	Kernel Symbol	Description
0	ICMP_NET_UNREACH	Network Unreachable
1	ICMP_HOST_UNREACH	Host Unreachable
2	ICMP_PROT_UNREACH	Protocol Unreachable
3	ICMP_PORT_UNREACH	Port Unreachable
4	ICMP_FRAG_NEEDED	Fragmentation Needed, but the DF flag is set.
5	ICMP_SR_FAILED	Source route failed
6	ICMP_NET_UNKNOWN	Destination network unknown
7	ICMP_HOST_UNKNOWN	Destination host unknown
8	ICMP_HOST_ISOLATED	Source host isolated
9	ICMP_NET_ANO	The destination network is administratively prohibited.
10	ICMP_HOST_ANO	The destination host is administratively prohibited.
11	ICMP_NET_UNR_TOS	The network is unreachable for Type Of Service.
12	ICMP_HOST_UNR_TOS	The host is unreachable for Type Of Service.
13	ICMP_PKT_FILTERED	Packet filtered
14	ICMP_PREC_VIOLATION	Precedence violation
15	ICMP_PREC_CUTOFF	Precedence cut off
16	NR_ICMP_UNREACH	Number of unreachable codes

Table 3-3. *ICMPv4 Redirect (ICMP_REDIRECT) Codes*

Code	Kernel Symbol	Description
0	ICMP_REDIR_NET	Redirect Net
1	ICMP_REDIR_HOST	Redirect Host
2	ICMP_REDIR_NETTOS	Redirect Net for TOS
3	ICMP_REDIR_HOSTTOS	Redirect Host for TOS

Table 3-4. *ICMPv4 Time Exceeded (ICMP_TIME_EXCEEDED) Codes*

Code	Kernel Symbol	Description
0	ICMP_EXC_TTL	TTL count exceeded
1	ICMP_EXC_FRAGTIME	Fragment Reassembly time exceeded

Table 3-5. *ICMPv6 "Destination Unreachable" (ICMPV6_DEST_UNREACH) Codes*

Code	Kernel Symbol	Description
0	ICMPV6_NOROUTE	No route to destination
1	ICMPV6_ADM_PROHIBITED	Communication with destination administratively prohibited
2	ICMPV6_NOT_NEIGHBOUR	Beyond scope of source address
3	ICMPV6_ADDR_UNREACH	Address Unreachable
4	ICMPV6_PORT_UNREACH	Port Unreachable

Note that ICMPV6_PKT_TOOBIG, which is the counterpart of IPv4 ICMP_DEST_UNREACH /ICMP_FRAG_NEEDED, is not a code of ICMPV6_DEST_UNREACH, but an ICMPv6 type in itself.

Table 3-6. *ICMPv6 Time Exceeded (ICMPV6_TIME_EXCEED) Codes*

Code	Kernel Symbol	Description
0	ICMPV6_EXC_HOPLIMIT	Hop limit exceeded in transit
1	ICMPV6_EXC_FRAGTIME	Fragment reassembly time exceeded

Table 3-7. *ICMPv6 Parameter Problem (ICMPV6_PARAMPROB) Codes*

Code	Kernel Symbol	Description
0	ICMPV6_HDR_FIELD	Erroneous header field encountered
1	ICMPV6_UNK_NEXTHDR	Unknown Next Header type encountered
2	ICMPV6_UNK_OPTION	Unknown IPv6 option encountered

procfs entries

The kernel provides a way of configuring various settings for various subsystems from the userspace by way of writing values to entries under `/proc`. These entries are referred to as `procfs` entries. All of the ICMPv4 `procfs` entries are represented by variables in the `netns_ipv4` structure (`include/net/netns/ipv4.h`), which is an object in the network namespace (`struct net`). Network namespaces and their implementation are discussed in Chapter 14. The following are the names of the `sysctl` variables that correspond to the ICMPv4 `netns_ipv4` elements, explanations about their usage, and the default values to which they are initialized, specifying also in which method the initialization takes place.

sysctl_icmp_echo_ignore_all

When `icmp_echo_ignore_all` is set, echo requests (ICMP_ECHO) will not be replied.
`procfs` entry: `/proc/sys/net/ipv4/icmp_echo_ignore_all`
Initialized to 0 in `icmp_sk_init()`

sysctl_icmp_echo_ignore_broadcasts

When receiving a broadcast or a multicast echo (ICMP_ECHO) message or a timestamp (ICMP_TIMESTAMP) message, you check whether broadcast/multicast requests are permitted by reading `sysctl_icmp_echo_ignore_broadcasts`. If this variable is set, you drop the packet and return 0.
`procfs` entry: `/proc/sys/net/ipv4/icmp_echo_ignore_broadcasts`
Initialized to 1 in `icmp_sk_init()`

sysctl_icmp_ignore_bogus_error_responses

Some routers violate RFC1122 by sending bogus responses to broadcast frames. In the `icmp_unreach()` method, you check this flag. If this flag is set to TRUE, the kernel will not log these warnings ("<IPv4Addr>sent an invalid ICMP type. . .").
`procfs` entry: `/proc/sys/net/ipv4/icmp_ignore_bogus_error_responses`
Initialized to 1 in `icmp_sk_init()`

sysctl_icmp_ratelimit

Limit the maximal rates for sending ICMP packets whose type matches the icmp ratemask (`icmp_ratemask`, see later in this section) to specific targets.
A value of 0 means disable any limiting; otherwise it is the minimal space between responses in milliseconds.
`procfs` entry: `/proc/sys/net/ipv4/icmp_ratelimit`
Initialized to 1 * HZ in `icmp_sk_init()`

sysctl_icmp_ratemask

Mask made of ICMP types for which rates are being limited. Each bit is an ICMPv4 type.
`procfs` entry: `/proc/sys/net/ipv4/icmp_ratemask`
Initialized to 0x1818 in `icmp_sk_init()`

sysctl_icmp_errors_use_inbound_ifaddr

The value of this variable is checked in `icmp_send()`. When it's not set, the ICMP error messages are sent with the primary address of the interface on which the packet will be sent. When it is set, the ICMP message will be sent with the primary address of the interface that received the packet that caused the icmp error.
`procfs` entry: `/proc/sys/net/ipv4/icmp_errors_use_inbound_ifaddr`
Initialized to 0 in `icmp_sk_init()`

■ **Note** See also more about the ICMP `sysctl` variables, their types and their default values in `Documentation/networking/ip-sysctl.txt`.

Creating "Destination Unreachable" Messages with iptables

The `iptables` userspace tool enables us to set rules which dictate what the kernel should do with traffic according to filters set by these rules. Handling `iptables` rules is done in the netfilter subsystem, and is discussed in Chapter 9. One of the `iptables` rules is the `reject` rule, which discards packets without further processing them. When setting an `iptables reject` target, the user can set a rule to send a "Destination Unreachable" ICMPv4 messages with various codes using the `-j REJECT` and `--reject-with` qualifiers. For example, the following `iptables` rule will discard any packet from any source with sending back an ICMP message of "ICMP Host Prohibited":

```
iptables -A INPUT -j REJECT --reject-with icmp-host-prohibited
```

These are the possible values to the `--reject-with` qualifier for setting an ICMPV4 message which will be sent in reply to the sending host:

```
icmp-net-unreachable   - ICMP_NET_UNREACH
icmp-host-unreachable  - ICMP_HOST_UNREACH
icmp-port-unreachable  - ICMP_PORT_UNREACH
icmp-proto-unreachable - ICMP_PROT_UNREACH
icmp-net-prohibited    - ICMP_NET_ANO
icmp-host-prohibited   - ICMP_HOST_ANO
icmp-admin-prohibited  - ICMP_PKT_FILTERED
```

You can also use `--reject-with tcp-reset` which will send a TCP RST packet in reply to the sending host.

(`net/ipv4/netfilter/ipt_REJECT.c`)

With `ip6tables` in IPv6, there is also a REJECT target. For example:

```
ip6tables -A INPUT -s 2001::/64 -p ICMPv6  -j REJECT --reject-with icmp6-adm-prohibited
```

These are the possible values to the `--reject-with` qualifier for setting an ICMPv6 message which will be sent in reply to the sending host:

```
no-route, icmp6-no-route              - ICMPV6_NOROUTE.
adm-prohibited, icmp6-adm-prohibited  - ICMPV6_ADM_PROHIBITED.
port-unreach, icmp6-port-unreachable  - ICMPV6_NOT_NEIGHBOUR.
addr-unreach, icmp6-addr-unreachable  - ICMPV6_ADDR_UNREACH.
```

(`net/ipv6/netfilter/ip6t_REJECT.c`)

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.

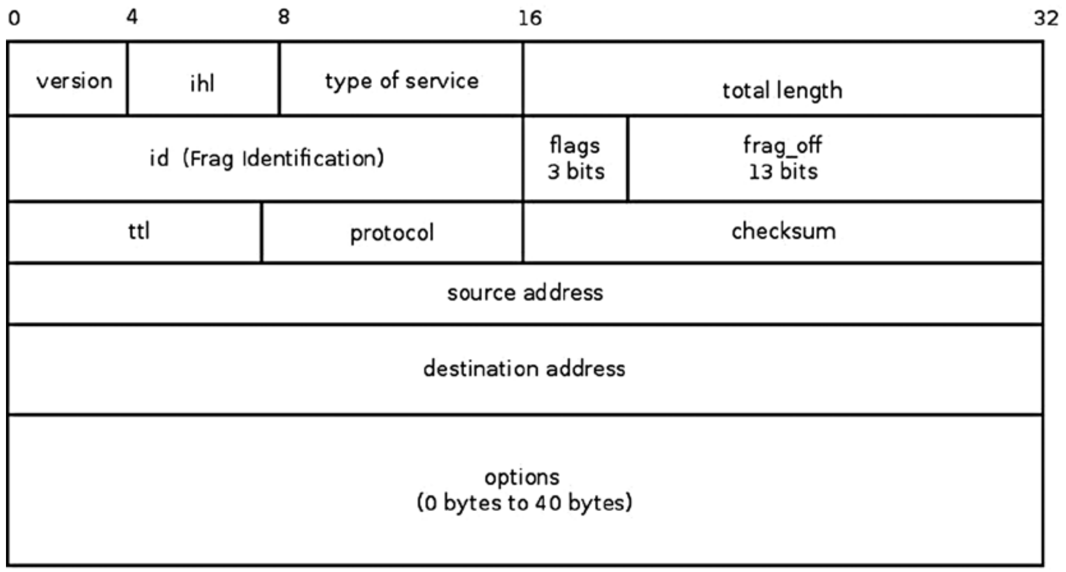


Figure 4-1. *IPv4 header*

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.

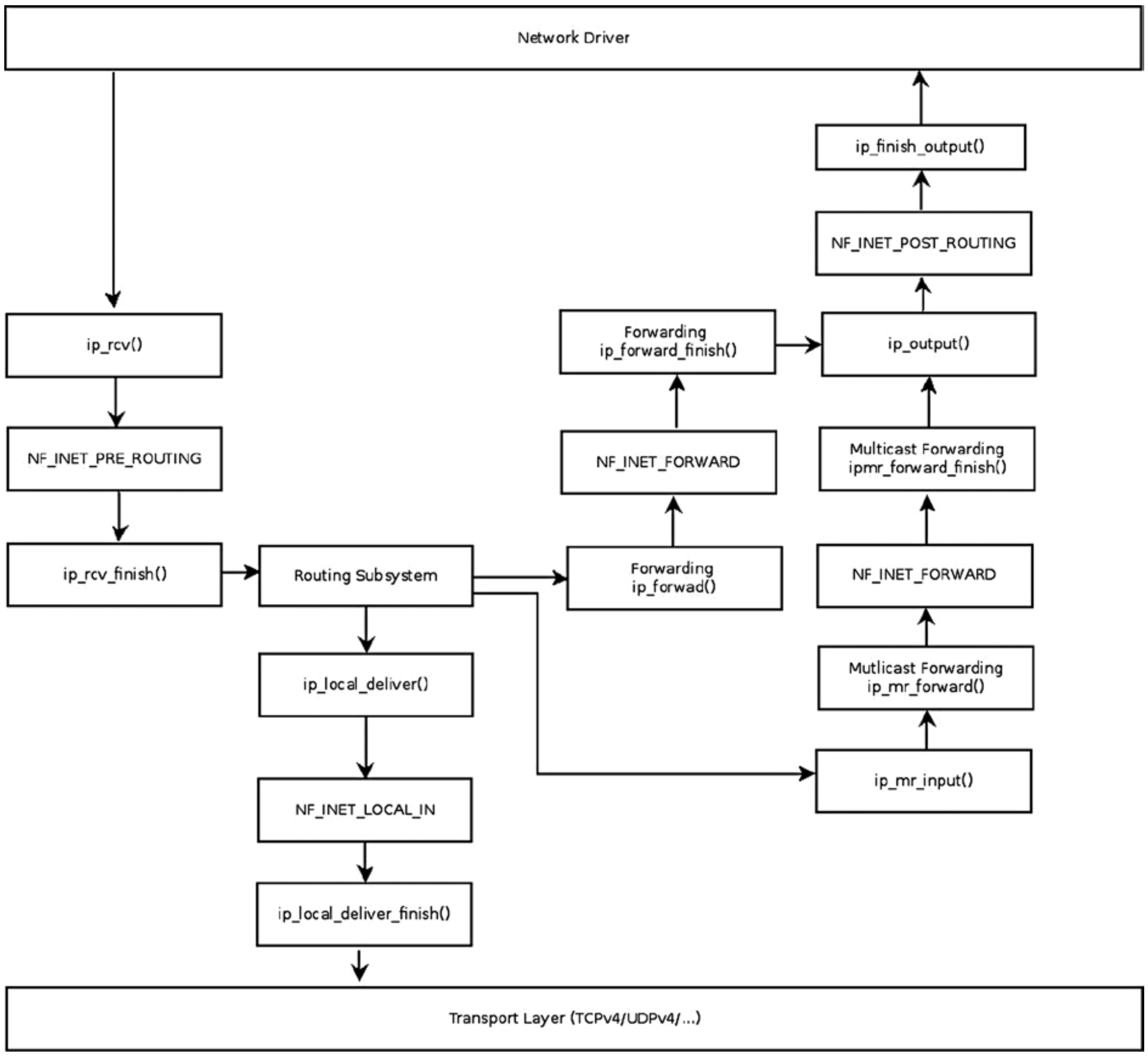


Figure 4-2. *Receiving IPv4 packets. For simplicity, the diagram does not include the fragmentation/defragmentation/options/IPsec methods*

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.

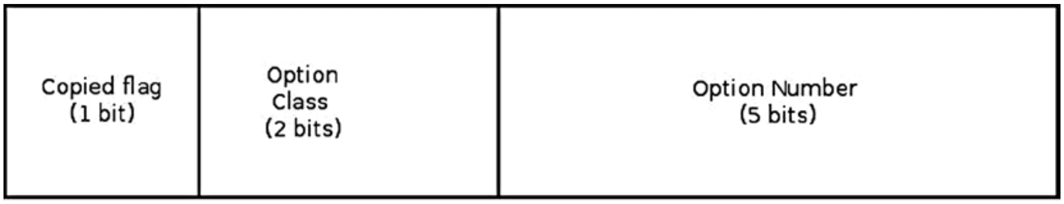


Figure 4-3. *Option type*

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.

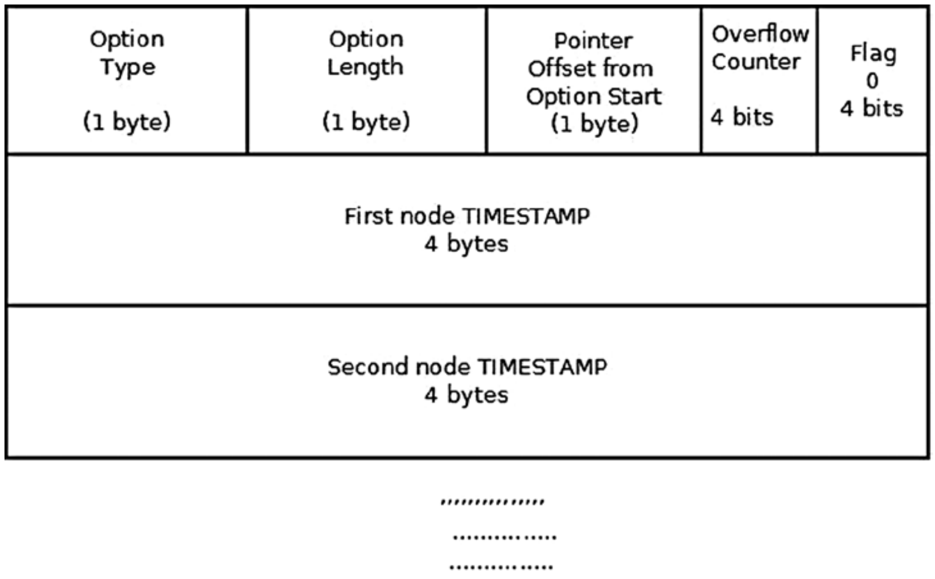

Option Type (1 byte)
Option Length (1 byte)
Pointer Offset from Option Start (1 byte)
Overflow Counter 4 bits
Flag 0 4 bits
First node TIMESTAMP 4 bytes
Second node TIMESTAMP 4 bytes

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.

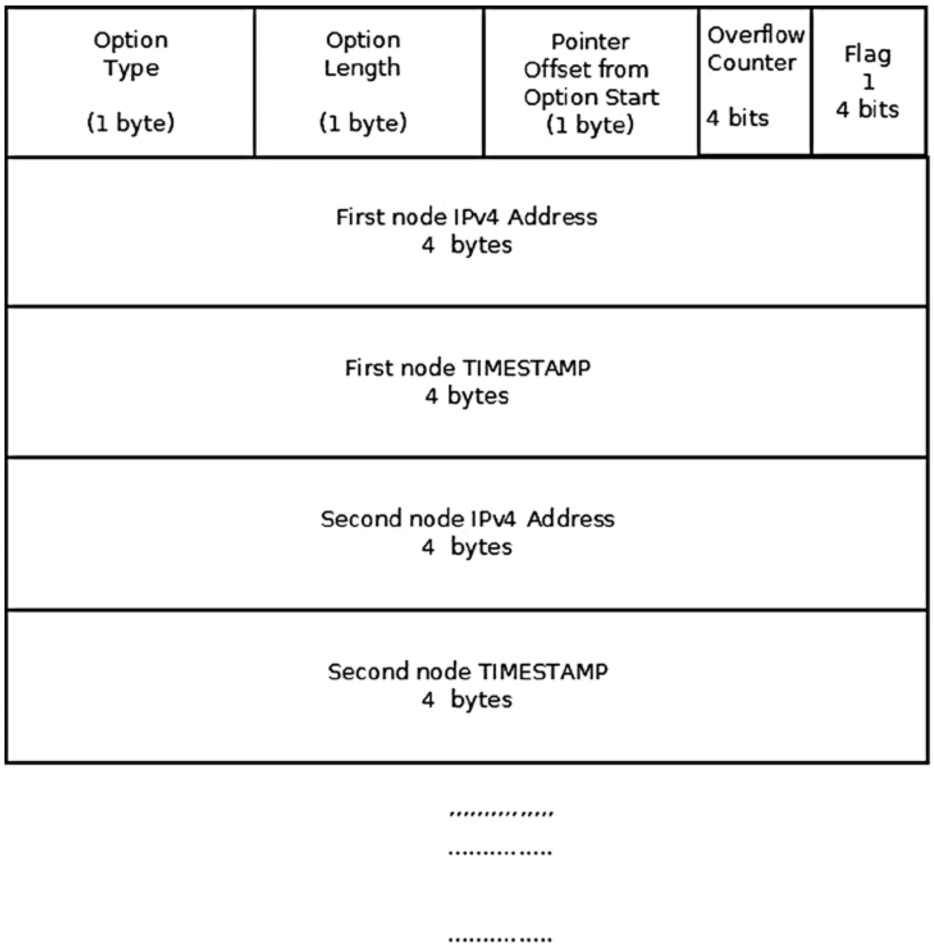

Option Type (1 byte)
Option Length (1 byte)
Pointer Offset from Option Start (1 byte)
Overflow Counter 4 bits
Flag 1 4 bits
First node IPv4 Address 4 bytes
First node TIMESTAMP 4 bytes
Second node IPv4 Address 4 bytes
Second node TIMESTAMP 4 bytes

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.

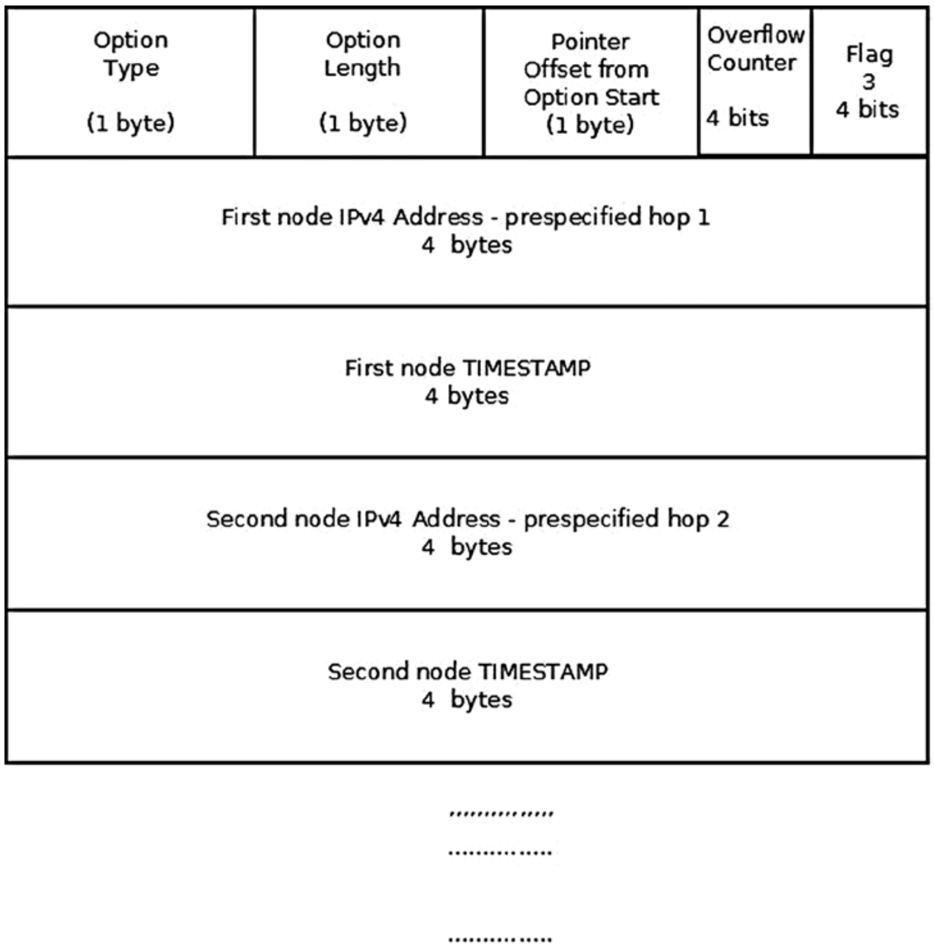

Option Type (1 byte)
Option Length (1 byte)
Pointer Offset from Option Start (1 byte)
Overflow Counter 4 bits
Flag 3 4 bits
First node IPv4 Address - prespecified hop 1 4 bytes
First node TIMESTAMP 4 bytes
Second node IPv4 Address - prespecified hop 2 4 bytes
Second node TIMESTAMP 4 bytes

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.

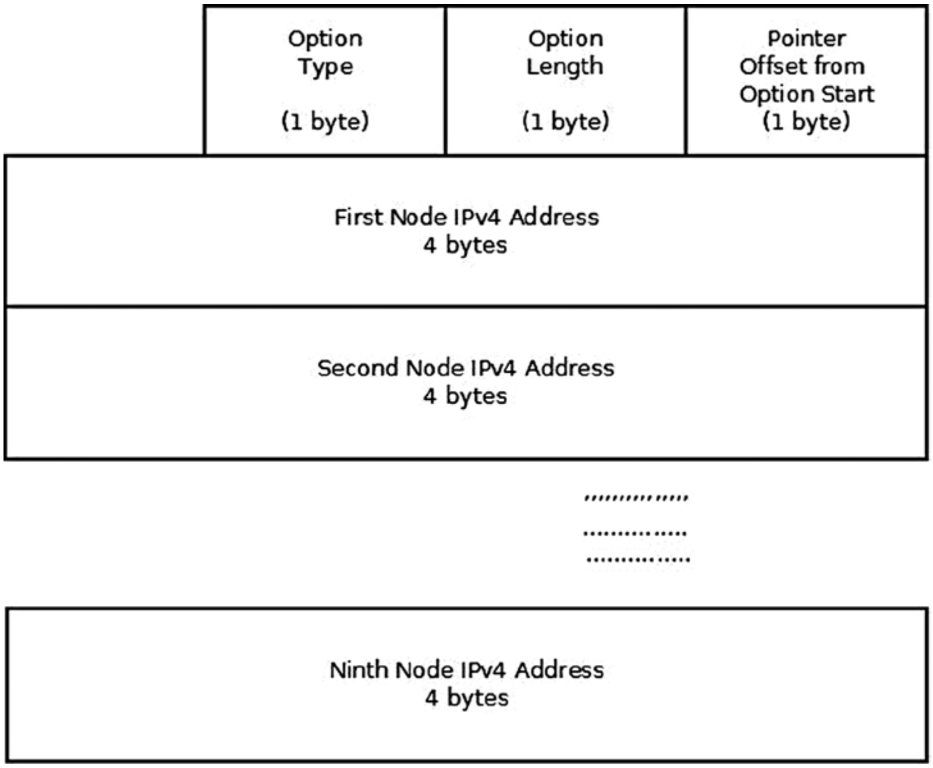

Option Type (1 byte)
Option Length (1 byte)
Pointer Offset from Option Start (1 byte)
First Node IPv4 Address 4 bytes
Second Node IPv4 Address 4 bytes
Ninth Node IPv4 Address 4 bytes

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.

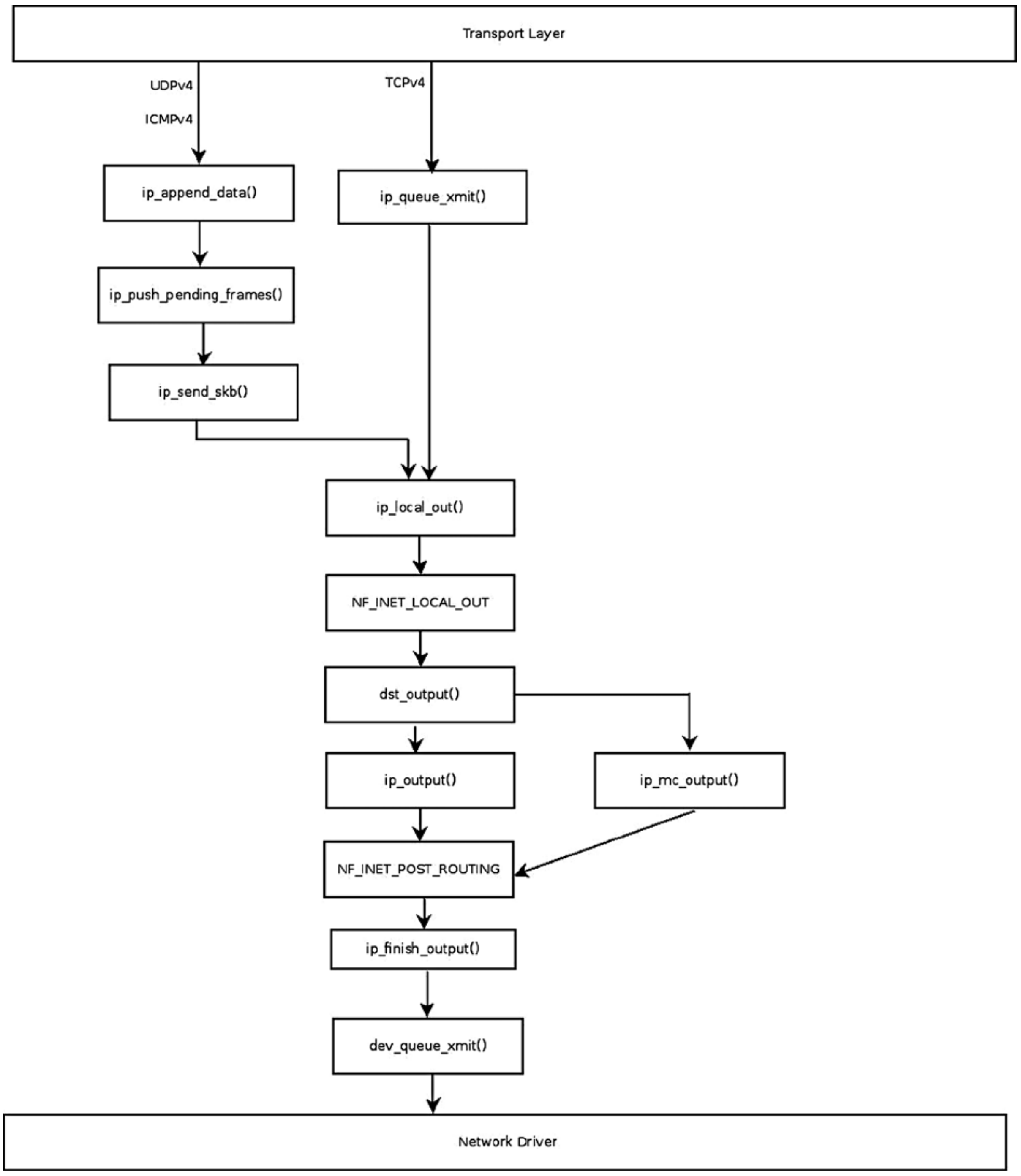


Figure 4-8. *Sending IPv4 packets*

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.

CHAPTER 5

The IPv4 Routing Subsystem

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

Forwarding and the FIB

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

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

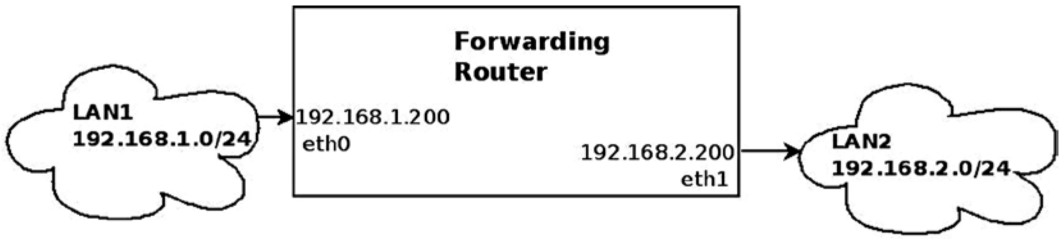


Figure 5-1. *Forwarding packets between two LANs*

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

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

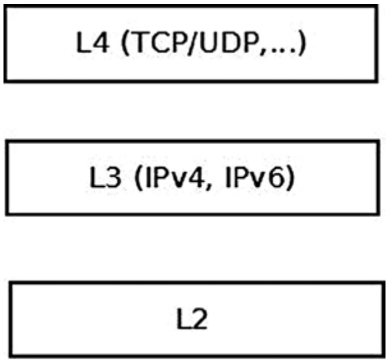


Figure 5-2. *The three layers that are handled by the networking kernel stack*

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

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

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

```
route add default gateway 192.168.2.1
```

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

Performing a Lookup in the Routing Subsystem

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

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

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

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

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

```
struct rtable {
    struct dst_entry  dst;

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

    int               rt_iif;

    /* Info on neighbour */
    __be32            rt_gateway;
    /* Miscellaneous cached information */
    u32             rt_pmtu;

    struct list_head  rt_uncached;
};
```

(include/net/route.h)

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

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

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

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

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

(include/net/ip_fib.h)

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

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

FIB Tables

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

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

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

(`include/net/ip_fib.h`)

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

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

FIB Info

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

```
struct fib_info {
    struct hlist_node    fib_hash;
    struct hlist_node    fib_lhash;
    struct net          *fib_net;
    int                 fib_treeref;
    atomic_t            fib_clntref;
    unsigned int        fib_flags;
    unsigned char       fib_dead;
    unsigned char       fib_protocol;
    unsigned char       fib_scope;
    unsigned char       fib_type;
    __be32              fib_prefsrc;
    u32                 fib_priority;
    u32                 *fib_metrics;
#define fib_mtu fib_metrics[RTAX_MTU-1]
#define fib_window fib_metrics[RTAX_WINDOW-1]
#define fib_rtt fib_metrics[RTAX_RTT-1]
#define fib_advmss fib_metrics[RTAX_ADVMSS-1]
    int                 fib_nhs;
#ifdef CONFIG_IP_ROUTE_MULTIPATH
    int                 fib_power;
#endif
    struct rcu_head     rcu;
    struct fib_nh       fib_nh[0];
#define fib_dev         fib_nh[0].nh_dev
};
```

(`include/net/ip_fib.h`)

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

- `fib_protocol`: The routing protocol identifier of this route. When adding a routing rule from userspace without specifying the routing protocol ID, the `fib_protocol` is assigned to be RTPROT_BOOT. The administrator may add a route with the "proto static" modifier, which indicates that the route was added by an administrator; this can be done, for example, like this: `ip route add proto static 192.168.5.3 via 192.168.2.1`. The `fib_protocol` can be assigned one of these flags:
  - RTPROT_UNSPEC: An error value.
  - RTPROT_REDIRECT: When set, the routing entry was created as a result of receiving an ICMP Redirect message. The RTPROT_REDIRECT protocol identifier is used only in IPv6.
  - RTPROT_KERNEL: When set, the routing entry was created by the kernel (for example, when creating the local IPv4 routing table, explained shortly).
  - RTPROT_BOOT: When set, the admin added a route without specifying the "proto static" modifier.
  - RTPROT_STATIC: Route installed by system administrator.
  - RTPROT_RA: Don't misread this— this protocol identifier is not for Router Alert; it is for RDISC/ND Router Advertisements, and it is used in the kernel by the IPv6 subsystem only; see: `net/ipv6/route.c`. I discuss it in Chapter 8.

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

- `fib_scope`: The scope of the destination address. In short, scopes are assigned to addresses and routes. Scope indicates the distance of the host from other nodes. The `ip address show` command shows the scopes of all configured IP addresses on a host. The `ip route show` command displays the scopes of all the route entries of the main table. A scope can be one of these:
  - host (RT_SCOPE_HOST): The node cannot communicate with the other network nodes. The loopback address has scope host.
  - global (RT_SCOPE_UNIVERSE): The address can be used anywhere. This is the most common case.
  - link (RT_SCOPE_LINK): This address can be accessed only from directly attached hosts.
  - site (RT_SCOPE_SITE): This is used in IPv6 only (I discuss it in Chapter 8).
  - nowhere (RT_SCOPE_NOWHERE): Destination doesn't exist.

  When a route is added by an administrator without specifying a scope, the `fib_scope` field is assigned a value according to these rules:
  - global scope (RT_SCOPE_UNIVERSE): For all gatewayed unicast routes.
  - scope link (RT_SCOPE_LINK): For direct unicast and broadcast routes.
  - scope host (RT_SCOPE_HOST): For local routes.

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

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

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

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

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

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

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

  or like this:

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

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

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

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

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

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

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

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

(`net/ipv4/fib_lookup.h`)

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

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

. . .
```

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

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

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

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

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

Caching

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

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

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

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

Nexthop (fib_nh)

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

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

(include/net/ip_fib.h)

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

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

FIB Nexthop Exceptions

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

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

(include/net/ip_fib.h)

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

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

}
```

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

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

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

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

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

Policy Routing

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

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

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

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

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

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

FIB Alias (fib_alias)

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

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

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

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

(`net/ipv4/fib_lookup.h`)

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

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

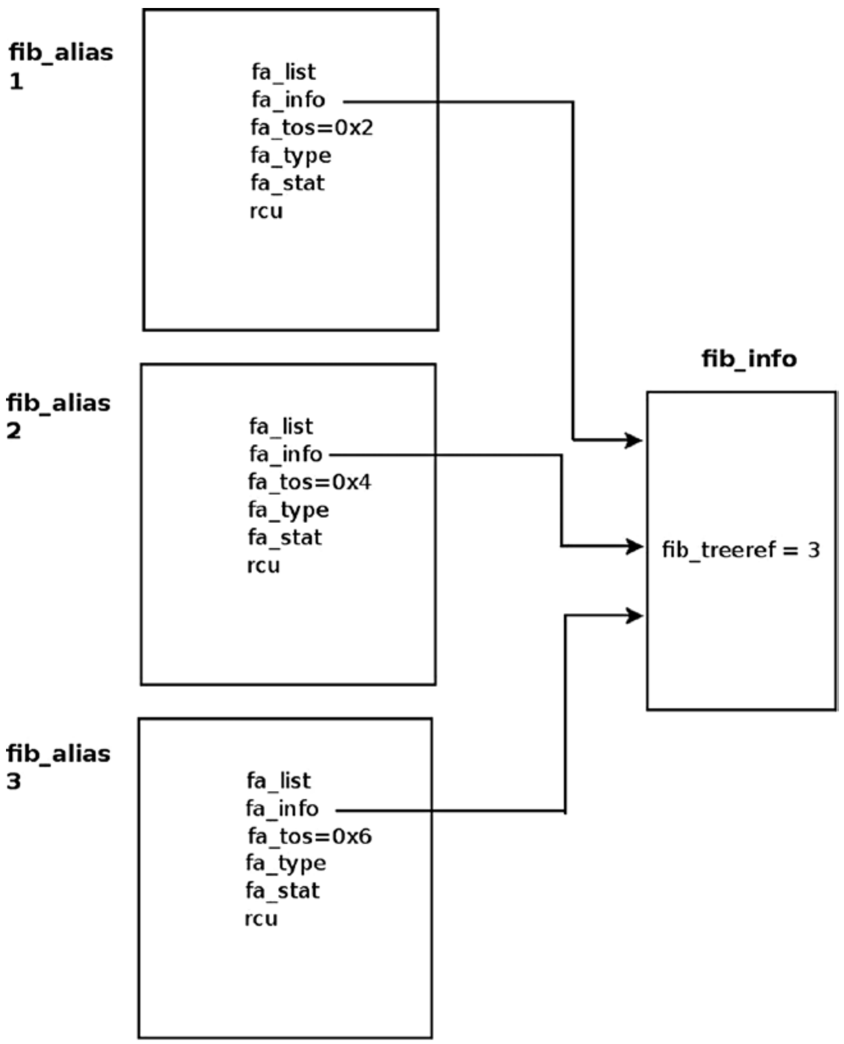


Figure 5-3. *A fib_info which is shared by three fib_alias objects. Each fib_alias object has a different fa_tos value*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ICMPv4 Redirect Message

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

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

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

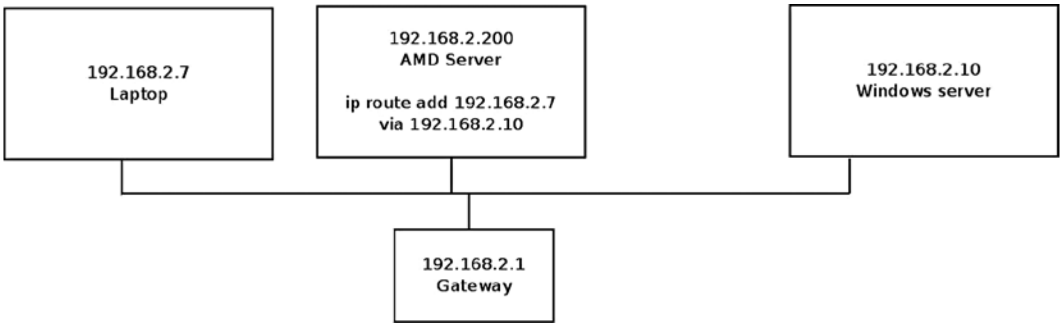


Figure 5-4. *Redirect to Host (ICMP_REDIR_HOST), a simple setup*

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

Generating an ICMPv4 Redirect Message

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

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

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

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

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

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

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

Setting the `rtable` object flags is done by:

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

}
```

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

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

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

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

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

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

(`net/ipv4/route.c`)

Receiving an ICMPv4 Redirect Message

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

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

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

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

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

net = dev_net(dev);
if (new_gw == old_gw || !IN_DEV_RX_REDIRECTS(in_dev) ||
    ipv4_is_multicast(new_gw) || ipv4_is_lbcast(new_gw) ||
    ipv4_is_zeronet(new_gw))
    goto reject_redirect;

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

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

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

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

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

reject_redirect:
      . . .

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

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

IPv4 Routing Cache

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

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

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

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

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

Rx Path

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

Tx Path

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

Summary

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

Quick Reference

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

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

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

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

CONFIG_IP_ROUTE_MULTIPATH should be set for Multipath Routing Support.

Methods

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

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

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

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

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

struct fib_info *fib_create_info(struct fib_config *cfg);

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

void free_fib_info(struct fib_info *fi);

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

void fib_alias_accessed(struct fib_alias *fa);

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

void ip_rt_send_redirect(struct sk_buff *skb);

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

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

This method handles receiving an ICMPv4 Redirect message.

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

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

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

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

struct fib_table *fib_trie_table(u32 id);

This method allocates and initializes a FIB TRIE table.

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

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

Macros

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

FIB_RES_GW()

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

FIB_RES_DEV()

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

FIB_RES_OIF()

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

FIB_RES_NH()

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

(`include/net/ip_fib.h`)

IN_DEV_FORWARD()

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

IN_DEV_RX_REDIRECTS()

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

IN_DEV_TX_REDIRECTS()

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

IS_LEAF()

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

IS_TNODE()

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

change_nexthops()

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

Tables

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

Table 5-1. *Route Metrics*

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

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

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

Table 5-2. *Route Types*

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

Route Flags

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

U (Route is up)

H (Target is a host)

G (Use gateway)

R (Reinstate route for dynamic routing)

D (Dynamically installed by daemon or redirect)

M (Modified from routing daemon or redirect)

A (Installed by addrconf)

! (Reject route)

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

Table 5-3. *Kernel IP Routing Table*

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

CHAPTER 6

Advanced Routing

Chapter 5 dealt with the IPv4 routing subsystem. This chapter continues with the routing subsystem and discusses advanced IPv4 routing topics such as Multicast Routing, Multipath Routing, Policy Routing, and more. This book deals with the Linux Kernel Networking implementation—it does not delve into the internals of userspace Multicast Routing daemons implementation, which are quite complex and beyond the scope of the book. I do, however, discuss to some extent the interaction between a userspace multicast routing daemon and the multicast layer in the kernel. I also briefly discuss the Internet Group Management Protocol (IGMP) protocol, which is the basis of multicast group membership management; adding and deleting multicast group members is done by the IGMP protocol. Some basic knowledge of IGMP is needed to understand the interaction between a multicast host and a multicast router.

Multipath Routing is the ability to add more than one nexthop to a route. Policy Routing enables configuring routing policies that are not based solely on the destination address. I start with describing Multicast Routing.

Multicast Routing

Chapter 4 briefly mentions Multicast Routing, in the "Receiving IPv4 Multicast Packets" section. I will now discuss it in more depth. Sending multicast traffic means sending the same packet to multiple recipients. This feature can be useful in streaming media, audio/video conferencing, and more. It has a clear advantage over unicast traffic in terms of saving network bandwidth. Multicast addresses are defined as Class D addresses. The Classless Inter-Domain Routing (CIDR) prefix of this group is 224.0.0.0/4. The range of IPv4 multicast addresses is from 224.0.0.0 to 239.255.255.255. Handling Multicast Routing must be done in conjunction with a userspace routing daemon which interacts with the kernel. According to the Linux implementation, Multicast Routing cannot be handled solely by the kernel code without this userspace Routing daemon, as opposed to Unicast Routing. There are various multicast daemons: for example: `mrouted`, which is based on an implementation of the Distance Vector Multicast Routing Protocol (DVMRP), or `pimd`, which is based on the Protocol-Independent Multicast protocol (PIM). The DVMRP protocol is defined in RFC 1075, and it was the first multicast routing protocol. It is based on the Routing Information Protocol (RIP) protocol.

The PIM protocol has two versions, and the kernel supports both of them (CONFIG_IP_PIMSM_V1 and CONFIG_IP_PIMSM_V2). PIM has four different modes: PIM-SM (PIM Sparse Mode), PIM-DM (PIM Dense Mode), PIM Source-Specific Multicast (PIM-SSM) and Bidirectional PIM. The protocol is called *protocol-independent* because it is not dependent on any particular routing protocol for topology discovery. This section discusses the interaction between the userspace daemon and the kernel multicast routing layer. Delving into the internals of the PIM protocol or the DVMRP protocol (or any other Multicast Routing protocol) is beyond the scope of this book. Normally, the Multicast Routing lookup is based on the source and destination addresses. There is a "Multicast Policy Routing" kernel feature, which is the parallel to the unicast policy routing kernel feature that was mentioned in Chapter 5 and which is also discussed in the course of this chapter. The multicast policy routing protocol is implemented using the Policy Routing API (for example, it calls the `fib_rules_lookup()` method to perform a lookup, creates a `fib_rules_ops` object, and registers it with the `fib_rules_register()` method, and so on). With Multicast Policy Routing, the routing can be based on additional criteria, like the ingress network interfaces. Moreover, you can work with more than one multicast routing table. In order to work with Multicast Policy Routing, IP_MROUTE_MULTIPLE_TABLES must be set.

Figure 6-1 shows a simple IPv4 Multicast Routing setup. The topology is very simple: the laptop, on the left, joins a multicast group (224.225.0.1) by sending an IGMP packet (IP_ADD_MEMBERSHIP). The IGMP protocol is discussed in the next section, "The IGMP Protocol." The AMD server, in the middle, is configured as a multicast router, and a userspace multicast routing daemon (like `pimd` or `mrouted`) runs on it. The Windows server, on the right, which has an IP address of 192.168.2.10, sends multicast traffic to 224.225.0.1; this traffic is forwarded to the laptop via the multicast router. Note that the Windows server itself did not join the 224.225.0.1 multicast group. Running `ip route add 224.0.0.0/4 dev <networkDeviceName>` tells the kernel to send all multicast traffic via the specified network device.

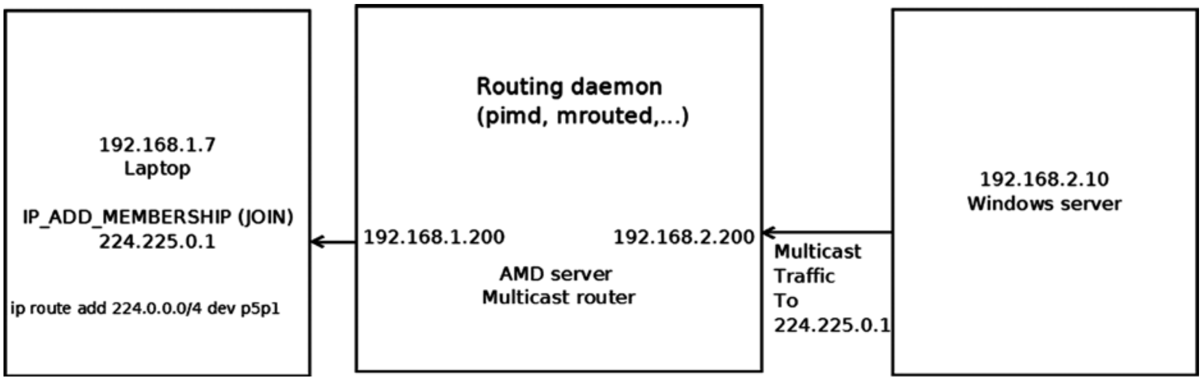


Figure 6-1. *Simple Multicast Routing setup*

The next section discusses the IGMP protocol, which is used for the management of multicast group membership.

The IGMP Protocol

The IGMP protocol is an integral part of IPv4 multicast. It must be implemented on each node that supports IPv4 multicast. In IPv6, multicast management is handled by the MLD (Multicast Listener Discovery) protocol, which uses ICMPv6 messages, discussed in Chapter 8. With the IGMP protocol, multicast group memberships are established and managed. There are three versions of IGMP:

1. *IGMPv1 (RFC 1112):* Has two types of messages—host membership report and host membership query. When a host wants to join a multicast group, it sends a membership report message. Multicast routers send membership queries to discover which host multicast groups have members on their attached local networks. Queries are addressed to the all-hosts group address (224.0.0.1, IGMP_ALL_HOSTS) and carry a TTL of 1 so that the membership query will not travel outside of the LAN.

2. *IGMPv2 (RFC 2236):* This is an extension of IGMPv1. The IGMPv2 protocol adds three new messages:
    a. Membership Query (0x11): There are two sub-types of Membership Query messages: General Query, used to learn which groups have members on an attached network, and Group-Specific Query, used to learn whether a particular group has any members on an attached network.
    b. Version 2 Membership Report (0x16).
    c. Leave Group (0x17).

■ **Note** IGMPv2 also supports Version 1 Membership Report message, for backward compatibility with IGMPv1. See RFC 2236, section 2.1.

---

3. *IGMPv3 (RFC 3376, updated by RFC 4604):* This major revision of the protocol adds a feature called source filtering. This means that when a host joins a multicast group, it can specify a set of source addresses from which it will receive multicast traffic. The source filters can also exclude source addresses. To support the source filtering feature, the socket API was extended; see RFC 3678, "Socket Interface Extensions for Multicast Source Filters." I should also mention that the multicast router periodically (about every two minutes) sends a membership query to 224.0.0.1, the all-hosts multicast group address. A host that receives a membership query responds with a membership report. This is implemented in the kernel by the `igmp_rcv()` method: getting an IGMP_HOST_MEMBERSHIP_QUERY message is handled by the `igmp_heard_query()` method.

■ **Note** The kernel implementation of IPv4 IGMP is in `net/core/igmp.c`, `include/linux/igmp.h` and `include/uapi/linux/igmp.h`.

The next section examines the fundamental data structure of IPv4 Multicast Routing, the multicast routing table, and its Linux implementation.

The Multicast Routing Table

The multicast routing table is represented by a structure named `mr_table`. Let's take a look at it:

```
struct mr_table {
    struct list_head     list;
#ifdef CONFIG_NET_NS
    struct net           *net;
#endif
    u32                  id;
    struct sock __rcu    *mroute_sk;
    struct timer_list    ipmr_expire_timer;
    struct list_head     mfc_unres_queue;
    struct list_head     mfc_cache_array[MFC_LINES];
    struct vif_device    vif_table[MAXVIFS];
    . . .
};
```

(net/ipv4/ipmr.c)

The following is a description of some members of the mr_table structure:

- net: The network namespace associated with the multicast routing table; by default it is the initial network namespace, init_net. Network namespaces are discussed in Chapter 14.
- id: The multicast routing table id; it is RT_TABLE_DEFAULT (253) when working with a single table.
- mroute_sk: This pointer represents a reference to the userspace socket that the kernel keeps. The mroute_sk pointer is initialized by calling setsockopt() from the userspace with the MRT_INIT socket option and is nullified by calling setsockopt() with the MRT_DONE socket option. The interaction between the userspace and the kernel is based on calling the setsockopt() method, on sending IOCTLs from userspace, and on building IGMP packets and passing them to the Multicast Routing daemon by calling the sock_queue_rcv_skb() method from the kernel.
- ipmr_expire_timer: Timer of cleaning unresolved multicast routing entries. This timer is initialized when creating a multicast routing table, in the ipmr_new_table() method, and removed when removing a multicast routing table, by the ipmr_free_table() method.
- mfc_unres_queue: A queue of unresolved routing entries.
- mfc_cache_array: A cache of the routing entries, with 64 (MFC_LINES) entries, discussed shortly in the next section.
- vif_table[MAXVIFS]: An array of 32 (MAXVIFS) vif_device objects. Entries are added by the vif_add() method and deleted by the vif_delete() method. The vif_device structure represents a virtual multicast routing network interface; it can be based on a physical device or on an IPIP (IP over IP) tunnel. The vif_device structure is discussed later in "The Vif Device" section.

I have covered the multicast routing table and mentioned its important members, such as the Multicast Forwarding Cache (MFC) and the queue of unresolved routing entries. Next I will look at the MFC, which is embedded in the multicast routing table object and plays an important role in Multicast Routing.

The Multicast Forwarding Cache (MFC)

The most important data structure in the multicast routing table is the MFC, which is in fact an array of cache entries (mfc_cache objects). This array, named mfc_cache_array, is embedded in the multicast routing table (mr_table) object. It has 64 (MFC_LINES) elements. The index of this array is the hash value (the hash function takes two parameters—the multicast group address and the source IP address; see the description of the MFC_HASH macro in the "Quick Reference" section at the end of this chapter).

Usually there is only one multicast routing table, which is an instance of the mr_table structure, and a reference to it is kept in the IPv4 network namespace (net->ipv4.mrt). The table is created by the ipmr_rules_init() method, which also assigns net->ipv4.mrt to point to the multicast routing table that was created. When working with the multicast policy routing feature mentioned earlier, there can be multiple multicast policy routing tables. In both cases, you get the routing table using the same method, ipmr_fib_lookup(). The ipmr_fib_lookup() method gets three parameters as an input: the network namespace, the flow, and a pointer to the mr_table object which it should fill. Normally, it simply sets the specified `mr_table` pointer to be `net->ipv4.mrt`; when working with multiple tables (IP_MROUTE_MULTIPLE_TABLES is set), the implementation is more complex. Let's take a look at the `mfc_cache` structure:

```
struct mfc_cache {
    struct list_head list;
    __be32 mfc_mcastgrp;
    __be32 mfc_origin;
    vifi_t mfc_parent;
    int mfc_flags;
    union {
            struct {
                    unsigned long expires;
                    struct sk_buff_head unresolved; /* Unresolved buffers */
            } unres;
            struct {
                    unsigned long last_assert;
                    int minvif;
                    int maxvif;
                    unsigned long bytes;
                    unsigned long pkt;
                    unsigned long wrong_if;
                    unsigned char ttls[MAXVIFS];    /* TTL thresholds */
            } res;
    } mfc_un;
    struct rcu_head rcu;
 };
```

(`include/linux/mroute.h`)

The following is a description of some members of the `mfc_cache` structure:

- `mfc_mcastgrp`: the address of the multicast group that the entry belongs to.
- `mfc_origin`: The source address of the route.
- `mfc_parent`: The source interface.
- `mfc_flags`: The flags of the entry. Can have one of these values:
    - MFC_STATIC: When the route was added statically and not by a multicast routing daemon.
    - MFC_NOTIFY: When the RTM_F_NOTIFY flag of the routing entry was set. See the `rt_fill_info()` method and the `ipmr_get_route()` method for more details.
- The `mfc_un` union consists of two elements:
    - `unres`: Unresolved cache entries.
    - `res`: Resolved cache entries.

The first time an SKB of a certain flow reaches the kernel, it is added to the queue of unresolved entries (mfc_un.unres.unresolved), where up to three SKBs can be saved. If there are three SKBs in the queue, the packet is not appended to the queue but is freed, and the ipmr_cache_unresolved() method returns -ENOBUFS ("No buffer space available"):

```
static int ipmr_cache_unresolved(struct mr_table *mrt, vifi_t vifi, struct sk_buff *skb)
{
         . . .
         if (c->mfc_un.unres.unresolved.qlen > 3) {
                 kfree_skb(skb);
                 err = -ENOBUFS;
        } else {
            . . .

}
```

(net/ipv4/ipmr.c)

This section described the MFC and its important members, including the queue of resolved entries and the queue of unresolved entries. The next section briefly describes what a multicast router is and how it is configured in Linux.

Multicast Router

In order to configure a machine as a multicast router, you should set the CONFIG_IP_MROUTE kernel configuration option. You should also run some routing daemon such as pimd or mrouted, as mentioned earlier. These routing daemons create a socket to communicate with the kernel. In pimd, for example, you create a raw IGMP socket by calling socket(AF_INET, SOCK_RAW, IPPROTO_IGMP). Calling setsockopt() on this socket triggers sending commands to the kernel, which are handled by the ip_mroute_setsockopt() method. When calling setsockopt() on this socket from the routing daemon with MRT_INIT, the kernel is set to keep a reference to the userspace socket in the mroute_sk field of the mr_table object that is used, and the mc_forwarding procfs entry (/proc/sys/net/ipv4/conf/all/mc_forwarding) is set by calling IPV4_DEVCONF_ALL(net, MC_FORWARDING)++. Note that the mc_forwarding procfs entry is a read-only entry and can't be set from userspace. You can't create another instance of a multicast routing daemon: when handling the MRT_INIT option, the ip_mroute_setsockopt() method checks whether the mroute_sk field of the mr_table object is initialized and returns -EADDRINUSE if so. Adding a network interface is done by calling setsockopt() on this socket with MRT_ADD_VIF, and deleting a network interface is done by calling setsockopt() on this socket with MRT_DEL_VIF. You can pass the parameters of the network interface to these setsockopt() calls by passing a vifctl object as the optval parameter of the setsockopt() system call. Let's take a look at the vifctl structure:

```
struct vifctl {
    vifi_t    vifc_vifi;                /* Index of VIF */
    unsigned char vifc_flags;            /* VIFF_ flags */
    unsigned char vifc_threshold;        /* ttl limit */
    unsigned int vifc_rate_limit;        /* Rate limiter values (NI) */
    union {
        struct in_addr vifc_lcl_addr;     /* Local interface address */
        int            vifc_lcl_ifindex;  /* Local interface index   */
    };
    struct in_addr vifc_rmt_addr;    /* IPIP tunnel addr */
};
```

(include/uapi/linux/mroute.h)

The following is a description of some members of the vifctl structure:

- vifc_flags can be:
    - VIFF_TUNNEL: When you want to use an IPIP tunnel.
    - VIFF_REGISTER: When you want to register the interface.
    - VIFF_USE_IFINDEX: When you want to use the local interface index and not the local interface IP address; in such a case, you will set the vifc_lcl_ifindex to be the local interface index. The VIFF_USE_IFINDEX flag is available for 2.6.33 kernel and above.
- vifc_lcl_addr: The local interface IP address. (This is the default—no flag should be set for using it).
- vifc_lcl_ifindex: The local interface index. It should be set when the VIFF_USE_IFINDEX flag is set in vifc_flags.
- vifc_rmt_addr: The address of the remote node of a tunnel.

When the multicast routing daemon is closed, the setsockopt() method is called with an MRT_DONE option. This triggers calling the mrtsock_destruct() method to nullify the mroute_sk field of the mr_table object that is used and to perform various cleanups.

This section covered what a multicast router is and how it is configured in Linux. I also examined the vifctl structure. Next, I look at the Vif device, which represents a multicast network interface.

The Vif Device

Multicast Routing supports two modes: direct multicast and multicast encapsulated in a unicast packet over a tunnel. In both cases, the same object is used (an instance of the vif_device structure) to represent the network interface. When working over a tunnel, the VIFF_TUNNEL flag will be set. Adding and deleting a multicast interface is done by the vif_add() method and by the vif_delete() method, respectively. The vif_add() method also sets the device to support multicast by calling the dev_set_allmulti(dev, 1) method, which increments the allmulti counter of the specified network device (net_device object). The vif_delete() method calls dev_set_allmulti(dev, -1) to decrement the allmulti counter of the specified network device (net_device object). For more details about the dev_set_allmulti() method, see appendix A. Let's take a look at the vif_device structure; its members are quite self-explanatory:

```
struct vif_device {
        struct net_device       *dev;       /* Device we are using */
        unsigned long   bytes_in,bytes_out;
        unsigned long   pkt_in,pkt_out;     /* Statistics                   */
        unsigned long   rate_limit;         /* Traffic shaping (NI)         */
        unsigned char   threshold;          /* TTL threshold                */
        unsigned short  flags;              /* Control flags                */
        __be32          local,remote;       /* Addresses(remote for tunnels)*/
        int             link;               /* Physical interface index     */
};
```

(include/linux/mroute.h)

In order to receive multicast traffic, a host must join a multicast group. This is done by creating a socket in userspace and calling setsockopt() with IPPROTO_IP and with the IP_ADD_MEMBERSHIP socket option. The userspace application also creates an ip_mreq object where it initializes the request parameters, like the desired group multicast address and the source IP address of the host (see the netinet/in.h userspace header). The setsockopt() call is handled in the kernel by the ip_mc_join_group() method, in net/ipv4/igmp.c. Eventually, the multicast address is added by the ip_mc_join_group() method to a list of multicast addresses (mc_list), which is a member of the in_device object. A host can leave a multicast group by calling setsockopt() with IPPROTO_IP and with the IP_DROP_MEMBERSHIP socket option. This is handled in the kernel by the ip_mc_leave_group() method, in net/ipv4/igmp.c. A single socket can join up to 20 multicast groups (sysctl_igmp_max_memberships). Trying to join more than 20 multicast groups by the same socket will fail with the -ENOBUFS error ("No buffer space available.") See the ip_mc_join_group() method implementation in net/ipv4/igmp.c.

IPv4 Multicast Rx Path

Chapter 4's "Receiving IPv4 Multicast Packets" section briefly discusses how multicast packets are handled. I will now describe this in more depth. My discussion assumes that our machine is configured as a multicast router; this means, as was mentioned earlier, that CONFIG_IP_MROUTE is set and a routing daemon like pimd or mrouted runs on this host. Multicast packets are handled by the ip_route_input_mc() method, in which a routing table entry (an rtable object) is allocated and initialized, and in which the input callback of the dst object is set to be ip_mr_input(), in case CONFIG_IP_MROUTE is set. Let's take a look at the ip_mr_input() method:

```
int ip_mr_input(struct sk_buff *skb)
{
        struct mfc_cache *cache;
        struct net *net = dev_net(skb->dev);
```

First the local flag is set to true if the packet is intended for local delivery, as the ip_mr_input() method also handles local multicast packets.

```
int local = skb_rtable(skb)->rt_flags & RTCF_LOCAL;
struct mr_table *mrt;

/* Packet is looped back after forward, it should not be
* forwarded second time, but still can be delivered locally.
*/
if (IPCB(skb)->flags & IPSKB_FORWARDED)
        goto dont_forward;
```

Normally, when working with a single multicast routing table, the ipmr_rt_fib_lookup() method simply returns the net->ipv4.mrt object:

```
mrt = ipmr_rt_fib_lookup(net, skb);
if (IS_ERR(mrt)) {
        kfree_skb(skb);
        return PTR_ERR(mrt);
}
if (!local) {
```

IGMPv3 and some IGMPv2 implementations set the router alert option (IPOPT_RA) in the IPv4 header when sending JOIN or LEAVE packets. See the igmpv3_newpack() method in net/ipv4/igmp.c:

```
if (IPCB(skb)->opt.router_alert) {
```

The ip_call_ra_chain() method (net/ipv4/ip_input.c) calls the raw_rcv() method to pass the packet to the userspace raw socket, which listens. The ip_ra_chain object contains a reference to the multicast routing socket, which is passed as a parameter to the raw_rcv() method. For more details, look at the ip_call_ra_chain() method implementation, in net/ipv4/ip_input.c:

```
if (ip_call_ra_chain(skb))
        return 0;
```

There are implementations where the router alert option is not set, as explained in the following comment; these cases must be handled as well, by calling the raw_rcv() method directly:

```
} else if (ip_hdr(skb)->protocol == IPPROTO_IGMP) {
        /* IGMPv1 (and broken IGMPv2 implementations sort of
        * Cisco IOS <= 11.2(8)) do not put router alert
        * option to IGMP packets destined to routable
        * groups. It is very bad, because it means
        * that we can forward NO IGMP messages.
        */
        struct sock *mroute_sk;
```

The mrt->mroute_sk socket is a copy in the kernel of the socket that the multicast routing userspace application created:

```
mroute_sk = rcu_dereference(mrt->mroute_sk);
        if (mroute_sk) {
        nf_reset(skb);
        raw_rcv(mroute_sk, skb);
        return 0;
        }
     }
}
```

First a lookup in the multicast routing cache, mfc_cache_array, is performed by calling the ipmr_cache_find() method. The hash key is the destination multicast group address and the source IP address of the packet, taken from the IPv4 header:

```
cache = ipmr_cache_find(mrt, ip_hdr(skb)->saddr, ip_hdr(skb)->daddr);
if (cache == NULL) {
```

A lookup in the virtual devices array (vif_table) is performed to see whether there is a corresponding entry which matches the incoming network device (skb->dev):

```
int vif = ipmr_find_vif(mrt, skb->dev);
```

The ipmr_cache_find_any() method handles the advanced feature of multicast proxy support (which is not discussed in this book):

```
        if (vif >= 0)
                cache = ipmr_cache_find_any(mrt, ip_hdr(skb)->daddr,
                                            vif);
}
```

```
/*
*      No usable cache entry
*/
if (cache == NULL) {
        int vif;
```

If the packet is destined to the local host, deliver it:

```
if (local) {
        struct sk_buff *skb2 = skb_clone(skb, GFP_ATOMIC);
        ip_local_deliver(skb);
        if (skb2 == NULL)
                return -ENOBUFS;
        skb = skb2;
}

read_lock(&mrt_lock);
vif = ipmr_find_vif(mrt, skb->dev);
if (vif >= 0) {
```

The ipmr_cache_unresolved() method creates a multicast routing entry (mfc_cache object) by calling the ipmr_cache_alloc_unres() method. This method creates a cache entry (mfc_cache object) and initializes its expiration time interval (by setting mfc_un.unres.expires). Let's take a look at this very short method, ipmr_cache_alloc_unres():

```
static struct mfc_cache *ipmr_cache_alloc_unres(void)
{
    struct mfc_cache *c = kmem_cache_zalloc(mrt_cachep, GFP_ATOMIC);

    if (c) {
        skb_queue_head_init(&c->mfc_un.unres.unresolved);
```

Setting the expiration time interval:

```
        c->mfc_un.unres.expires = jiffies + 10*HZ;
    }
    return c;
}
```

If the routing daemon does not resolve the routing entry within its expiration interval, the entry is removed from the queue of the unresolved entries. When creating a multicast routing table (by the ipmr_new_table() method), its timer (ipmr_expire_timer) is set. This timer invokes the ipmr_expire_process() method periodically. The ipmr_expire_process() method iterates over all the unresolved cache entries in the queue of unresolved entries (mfc_unres_queue of the mrtable object) and removes the expired unresolved cache entries.

After creating the unresolved cache entry, the ipmr_cache_unresolved() method adds it to the queue of unresolved entries (mfc_unres_queue of the multicast table, mrtable) and increments by 1 the unresolved queue length (cache_resolve_queue_len of the multicast table, mrtable). It also calls the ipmr_cache_report() method, which builds an IGMP message (IGMPMSG_NOCACHE) and delivers it to the userspace multicast routing daemon by calling eventually the sock_queue_rcv_skb() method.

I mentioned that the userspace routing daemon should resolve the routing within some time interval. I will not delve into how this is implemented in userspace. Note, however, that once the routing daemon decides it should resolve an unresolved entry, it builds the cache entry parameters (in an `mfcctl` object) and calls `setsockopt()` with the MRT_ADD_MFC socket option, then it passes the `mfcctl` object embedded in the `optval` parameter of the `setsockopt()` system call; this is handled in the kernel by the `ipmr_mfc_add()` method:

```
                int err2 = ipmr_cache_unresolved(mrt, vif, skb);
                read_unlock(&mrt_lock);

                return err2;
        }
        read_unlock(&mrt_lock);
        kfree_skb(skb);
        return -ENODEV;
}

read_lock(&mrt_lock);
```

If a cache entry was found in the MFC, call the `ip_mr_forward()` method to continue the packet traversal:

```
        ip_mr_forward(net, mrt, skb, cache, local);
        read_unlock(&mrt_lock);

        if (local)
                return ip_local_deliver(skb);

        return 0;

dont_forward:
        if (local)
                return ip_local_deliver(skb);
        kfree_skb(skb);
        return 0;
}
```

This section detailed the IPv4 Multicast Rx path and the interaction with the routing daemon in this path. The next section describes the multicast routing forwarding method, `ip_mr_forward()`.

The ip_mr_forward() Method

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

```
static int ip_mr_forward(struct net *net, struct mr_table *mrt,
             struct sk_buff *skb, struct mfc_cache *cache,
             int local)
{
    int psend = -1;
    int vif, ct;
    int true_vifi = ipmr_find_vif(mrt, skb->dev);

    vif = cache->mfc_parent;
```

Here you can see update statistics of the resolved cache object (`mfc_un.res`):

```
cache->mfc_un.res.pkt++;
cache->mfc_un.res.bytes += skb->len;

if (cache->mfc_origin == htonl(INADDR_ANY) && true_vifi >= 0) {
    struct mfc_cache *cache_proxy;
```

The expression (`*, G`) means traffic from any source sending to the group G:

```
    /* For an (*,G) entry, we only check that the incomming
    * interface is part of the static tree.
    */
    cache_proxy = ipmr_cache_find_any_parent(mrt, vif);
    if (cache_proxy &&
        cache_proxy->mfc_un.res.ttls[true_vifi] < 255)
        goto forward;
}
/*
 * Wrong interface: drop packet and (maybe) send PIM assert.
 */
if (mrt->vif_table[vif].dev != skb->dev) {
    if (rt_is_output_route(skb_rtable(skb))) {
        /* It is our own packet, looped back.
         * Very complicated situation...
         *
         * The best workaround until routing daemons will be
         * fixed is not to redistribute packet, if it was
         * send through wrong interface. It means, that
         * multicast applications WILL NOT work for
         * (S,G), which have default multicast route pointing
         * to wrong oif. In any case, it is not a good
         * idea to use multicasting applications on router.
         */
        goto dont_forward;
    }

    cache->mfc_un.res.wrong_if++;

    if (true_vifi >= 0 && mrt->mroute_do_assert &&
        /* pimsm uses asserts, when switching from RPT to SPT,
         * so that we cannot check that packet arrived on an oif.
         * It is bad, but otherwise we would need to move pretty
         * large chunk of pimd to kernel. Ough... --ANK
         */
        (mrt->mroute_do_pim ||
        cache->mfc_un.res.ttls[true_vifi] < 255) &&
        time_after(jiffies,
               cache->mfc_un.res.last_assert + MFC_ASSERT_THRESH)) {
        cache->mfc_un.res.last_assert = jiffies;
```

Call the ipmr_cache_report() method to build an IGMP message (IGMPMSG_WRONGVIF) and to deliver it to the userspace multicast routing daemon by calling the sock_queue_rcv_skb() method:

```
        ipmr_cache_report(mrt, skb, true_vifi, IGMPMSG_WRONGVIF);
    }
    goto dont_forward;
}
```

The frame is now ready to be forwarded:

```
forward:
    mrt->vif_table[vif].pkt_in++;
    mrt->vif_table[vif].bytes_in += skb->len;

    /*
     *    Forward the frame
     */
    if (cache->mfc_origin == htonl(INADDR_ANY) &&
        cache->mfc_mcastgrp == htonl(INADDR_ANY)) {
        if (true_vifi >= 0 &&
            true_vifi != cache->mfc_parent &&
            ip_hdr(skb)->ttl >
                cache->mfc_un.res.ttls[cache->mfc_parent]) {
            /* It's an (*,*) entry and the packet is not coming from
             * the upstream: forward the packet to the upstream
             * only.
             */
            psend = cache->mfc_parent;
            goto last_forward;
        }
        goto dont_forward;
    }
    for (ct = cache->mfc_un.res.maxvif - 1;
         ct >= cache->mfc_un.res.minvif; ct--) {
        /* For (*,G) entry, don't forward to the incoming interface */
        if ((cache->mfc_origin != htonl(INADDR_ANY) ||
             ct != true_vifi) &&
            ip_hdr(skb)->ttl > cache->mfc_un.res.ttls[ct]) {
            if (psend != -1) {
                struct sk_buff *skb2 = skb_clone(skb, GFP_ATOMIC);
```

Call the ipmr_queue_xmit() method to continue with the packet forwarding:

```
                if (skb2)
                    ipmr_queue_xmit(net, mrt, skb2, cache,
                            psend);
            }
            psend = ct;
        }
    }
last_forward:
    if (psend != -1) {
        if (local) {
            struct sk_buff *skb2 = skb_clone(skb, GFP_ATOMIC);

            if (skb2)
                ipmr_queue_xmit(net, mrt, skb2, cache, psend);
        } else {
            ipmr_queue_xmit(net, mrt, skb, cache, psend);
            return 0;
        }
    }

dont_forward:
    if (!local)
        kfree_skb(skb);
    return 0;
}
```

Now that I have covered the multicast routing forwarding method, `ip_mr_forward()`, it is time to examine the `ipmr_queue_xmit()` method.

The ipmr_queue_xmit() Method

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

```
static void ipmr_queue_xmit(struct net *net, struct mr_table *mrt,
                            struct sk_buff *skb, struct mfc_cache *c, int vifi)
{
        const struct iphdr *iph = ip_hdr(skb);
        struct vif_device *vif = &mrt->vif_table[vifi];
        struct net_device *dev;
        struct rtable *rt;
        struct flowi4 fl4;
```

The encap field is used when working with a tunnel:

```
        int encap = 0;

        if (vif->dev == NULL)
                goto out_free;

#ifdef CONFIG_IP_PIMSM
        if (vif->flags & VIFF_REGISTER) {
                vif->pkt_out++;
                vif->bytes_out += skb->len;
                vif->dev->stats.tx_bytes += skb->len;
                vif->dev->stats.tx_packets++;
                ipmr_cache_report(mrt, skb, vifi, IGMPMSG_WHOLEPKT);
                goto out_free;
        }
#endif
```

When working with a tunnel, a routing lookup is performed with the `vif->remote` and `vif->local`, which represent the destination and local addresses, respectively. These addresses are the end points of the tunnel. When working with a `vif_device` object which represents a physical device, a routing lookup is performed with the destination of the IPv4 header and 0 as a source address:

```
if (vif->flags & VIFF_TUNNEL) {
        rt = ip_route_output_ports(net, &fl4, NULL,
                                   vif->remote, vif->local,
                                   0, 0,
                                   IPPROTO_IPIP,
                                   RT_TOS(iph->tos), vif->link);
        if (IS_ERR(rt))
                goto out_free;
        encap = sizeof(struct iphdr);
} else {
       rt = ip_route_output_ports(net, &fl4, NULL, iph->daddr, 0,
                                  0, 0,
                                  IPPROTO_IPIP,
                                  RT_TOS(iph->tos), vif->link);
       if (IS_ERR(rt))
               goto out_free;
}

dev = rt->dst.dev;
```

Note that if the packet size is higher than the MTU, an ICMPv4 message is not sent (as is done in such a case under unicast forwarding); only the statistics are updated, and the packet is discarded:

```
if (skb->len+encap > dst_mtu(&rt->dst) && (ntohs(iph->frag_off) & IP_DF)) {
        /* Do not fragment multicasts. Alas, IPv4 does not
         * allow to send ICMP, so that packets will disappear
         * to blackhole.
         */

        IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_FRAGFAILS);
        ip_rt_put(rt);
        goto out_free;
}

encap += LL_RESERVED_SPACE(dev) + rt->dst.header_len;

if (skb_cow(skb, encap)) {
        ip_rt_put(rt);
        goto out_free;
}

vif->pkt_out++;
vif->bytes_out += skb->len;

skb_dst_drop(skb);
skb_dst_set(skb, &rt->dst);
```

The TTL is decreased, and the IPv4 header checksum is recalculated (because the TTL is one of the IPv4 fields) when forwarding the packet; the same is done in the `ip_forward()` method for unicast packets:

```
ip_decrease_ttl(ip_hdr(skb));

/* FIXME: forward and output firewalls used to be called here.
 * What do we do with netfilter? -- RR
 */
if (vif->flags & VIFF_TUNNEL) {
        ip_encap(skb, vif->local, vif->remote);
        /* FIXME: extra output firewall step used to be here. --RR */
        vif->dev->stats.tx_packets++;
        vif->dev->stats.tx_bytes += skb->len;
}

IPCB(skb)->flags |= IPSKB_FORWARDED;

/*
 * RFC1584 teaches, that DVMRP/PIM router must deliver packets locally
 * not only before forwarding, but after forwarding on all output
 * interfaces. It is clear, if mrouter runs a multicasting
 * program, it should receive packets not depending to what interface
 * program is joined.
 * If we will not make it, the program will have to join on all
 * interfaces. On the other hand, multihoming host (or router, but
 * not mrouter) cannot join to more than one interface - it will
 * result in receiving multiple packets.
 */
```

Invoke the NF_INET_FORWARD hook:

```
        NF_HOOK(NFPROTO_IPV4, NF_INET_FORWARD, skb, skb->dev, dev,
                ipmr_forward_finish);
        return;

out_free:
        kfree_skb(skb);
}
```

The ipmr_forward_finish() Method

Let's take a look at the `ipmr_forward_finish()` method, which is a very short method—it is in fact identical to the `ip_forward()` 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);
```

Handle IPv4 options, if set (see Chapter 4):

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

        return dst_output(skb);
}
```

Eventually, `dst_output()` sends the packet via the `ip_mc_output()` method, which calls the `ip_finish_output()` method (both methods are in `net/ipv4/route.c`).

Now that I have covered these multicast methods, let's get a better understanding of how the value of the TTL field is used in multicast traffic.

The TTL in Multicast Traffic

The TTL field of the IPv4 header has a double meaning when discussing multicast traffic. The first is the same as in unicast IPV4 traffic: the TTL represents a hop counter which is decreased by 1 on every device that is forwarding the packet. When it reaches 0, the packet is discarded. This is done to avoid endless travelling of packets due to some error. The second meaning of the TTL, which is unique to multicast traffic, is a threshold. The TTL values are divided into scopes. Routers have a TTL threshold assigned to each of their interfaces, and only packets with a TTL greater than the interface's threshold are forwarded. Here are the values of these thresholds:

- *0:* Restricted to the same host (cannot be sent out by any interface)
- *1:* Restricted to the same subnet (will not be forwarded by a router)
- *32:* Restricted to the same site
- *64:* Restricted to the same region
- *128:* Restricted to the same continent
- *255:* Unrestricted in scope (global)

See: "IP Multicast Extensions for 4.3BSD UNIX and related systems," by Steve Deering, available at `www.kohala.com/start/mcast.api.txt`.

■ **Note** IPv4 Multicast Routing is implemented in `net/ipv4/ipmr.c`, `include/linux/mroute.h`, and `include/uapi/linux/mroute.h`.

This completes my discussion of Multicast Routing. The chapter now moves on to Policy Routing, which enables you to configure routing policies that are not based solely on the destination address.

Policy Routing

With Policy Routing, a system administrator can define up to 255 routing tables. This section discusses IPv4 Policy Routing; IPv6 Policy Routing is discussed in Chapter 8. In this section, I use the terms *policy* or *rule* for entries that are created by Policy Routing, in order to avoid confusing the ordinary routing entries (discussed in Chapter 5) with policy rules.

Policy Routing Management

Policy Routing management is done with the `ip rule` command of the `iproute2` package (there is no parallel for Policy Routing management with the `route` command). Let's see how to add, delete, and dump all Policy Routing rules:

- You add a rule with the `ip rule add` command; for example: `ip rule add tos 0x04 table 252`. After this rule is inserted, every packet which has an IPv4 TOS field matching 0x04 will be handled according to the routing rules of table 252. You can add routing entries to this table by specifying the table number when adding a route; for example: `ip route add default via 192.168.2.10 table 252`. This command is handled in the kernel by the `fib_nl_newrule()` method, in `net/core/fib_rules.c`. The `tos` modifier in the `ip rule` command earlier is one of the available SELECTOR modifiers of the `ip rule` command; see `man 8 ip rule`, and also Table 6-1 in the "Quick Reference" section at the end of this chapter.
- You delete a rule with the `ip rule del` command; for example: `ip rule del tos 0x04 table 252`. This command is handled in the kernel by the `fib_nl_delrule()` method in `net/core/fib_rules.c`.
- You dump all the rules with the `ip rule list` command or the `ip rule show` command. Both these commands are handled in the kernel by the `fib_nl_dumprule()` method in `net/core/fib_rules.c`.

You now have a good idea about the basics of Policy Routing management, so let's examine the Linux implementation of Policy Routing.

Policy Routing Implementation

The core infrastructure of Policy Routing is the `fib_rules` module, `net/core/fib_rules.c`. It is used by three protocols of the kernel networking stack: IPv4 (including the multicast module, which has a multicast policy routing feature, as mentioned in the "Multicast Routing" section earlier in this chapter), IPv6, and DECnet. The IPv4 Policy Routing is implemented also in a file named `fib_rules.c`. Don't be confused by the identical name (`net/ipv4/fib_rules.c`). In IPv6, policy routing is implemented in `net/ipv6/fib6_rules.c`. The header file, `include/net/fib_rules.h`, contains the data structures and methods of the Policy Routing core. Here is the definition of the `fib4_rule` structure, which is the basis for IPv4 Policy Routing:

```
struct fib4_rule {
    struct fib_rule    common;
    u8             dst_len;
    u8             src_len;
    u8             tos;
    __be32             src;
    __be32             srcmask;
    __be32             dst;
    __be32             dstmask;
#ifdef CONFIG_IP_ROUTE_CLASSID
    u32            tclassid;
#endif
};

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

Three policies are created by default at boot time, by calling the `fib_default_rules_init()` method: the local (RT_TABLE_LOCAL) table, the main (RT_TABLE_MAIN) table, and the default (RT_TABLE_DEFAULT) table. Lookup is done by the `fib_lookup()` method. Note that there are two different implementations of the `fib_lookup()` method in `include/net/ip_fib.h`. The first one, which is wrapped in the `#ifndef` CONFIG_IP_MULTIPLE_TABLES block, is for non-Policy Routing, and the second is for Policy Routing. When working with Policy Routing, the lookup is performed like this: if there were no changes to the initial policy routing rules (`net->ipv4.fib_has_custom_rules` is not set), that means the rule must be in one of the three initial routing tables. So, first a lookup is done in the local table, then in the main table, and then the default table. If there is no corresponding entry, a network unreachable (-ENETUNREACH) error is returned. If there was some change in the initial policy routing rules (`net->ipv4.fib_has_custom_rules` is set), the`_fib_lookup()` method is invoked, which is a heavier method, because it iterates over the list of rules and calls `fib_rule_match()` for each rule in order to decide whether it matches or not. See the implementation of the `fib_rules_lookup()` method in `net/core/fib_rules.c`. (The `fib_rules_lookup()` method is invoked from the `__fib_lookup()` method). I should mention here that the `net->ipv4.fib_has_custom_rules` variable is set to `false` in the initialization phase, by the `fib4_rules_init()` method, and to `true` in the `fib4_rule_configure()` method and the `fib4_rule_delete()` method. Note that CONFIG_IP_MULTIPLE_TABLES should be set for working with Policy Routing.

This concludes my Multicast Routing discussion. The next section talks about Multipath Routing, which is the ability to add more than one nexthop to a route.

Multipath Routing

Multipath Routing provides the ability to add more than one nexthop to a route. Defining two nexthop nodes can be done like this, for example: `ip route add default scope global nexthop dev eth0 nexthop dev eth1`. A system administrator can also assign weights for each nexthop—like this, for example: `ip route add 192.168.1.10 nexthop via 192.168.2.1 weight 3 nexthop via 192.168.2.10 weight 5`. The `fib_info` structure represents an IPv4 routing entry that can have more than one FIB nexthop. The `fib_nhs` member of the `fib_info` object represents the number of FIB nexthop objects; the `fib_info` object contains an array of FIB nexthop objects named `fib_nh`. So in this case, a single `fib_info` object is created, with an array of two FIB nexthop objects. The kernel keeps the weight of each next hop in the `nh_weight` field of the FIB nexthop object (`fib_nh`). If weight was not specified when adding a multipath route, it is set by default to 1, in the `fib_create_info()` method. The `fib_select_multipath()` method is called to determine the nexthop when working with Multipath Routing. This method is invoked from two places: from the `__ip_route_output_key()` method, in the Tx path, and from the `ip_mkroute_input()` method, in the Rx path. Note that when the output device is set in the flow, the `fib_select_multipath()` method is not invoked, because the output device is known:

```
struct rtable *__ip_route_output_key(struct net *net, struct flowi4 *fl4) {
. . .
#ifdef CONFIG_IP_ROUTE_MULTIPATH
    if (res.fi->fib_nhs > 1 && fl4->flowi4_oif == 0)
        fib_select_multipath(&res);
    else
#endif
. . .

}
```

In the Rx path there is no need for checking whether `fl4->flowi4_oif` is 0, because it is set to 0 in the beginning of this method. I won't delve into the details of the `fib_select_multipath()` method. I will only mention that there is an element of randomness in the method, using `jiffies`, for helping in creating a fair weighted route distribution, and that the weight of each next hop is taken in account. The FIB nexthop to use is assigned by setting the FIB nexthop selector (`nh_sel`) of the specified `fib_result` object. In contrast to Multicast Routing, which is handled by a dedicated module (`net/ipv4/ipmr.c`), the code of Multipath Routing appears scattered in the existing routing code, enclosed in `#ifdef` CONFIG_IP_ROUTE_MULTIPATH conditionals, and no separate module was added in the source code for supporting it. As mentioned in Chapter 5, there was support for IPv4 multipath routing cache, but it was removed in 2007 in kernel 2.6.23; in fact, it never did work very well, and never got out of the experimental state. Do not confuse the removal of the multipath routing cache with the removal of the routing cache; these are two different caches. The removal of the routing cache took place five years later, in kernel 3.6 (2012).

■ **NOTE** CONFIG_IP_ROUTE_MULTIPATH should be set for Multipath Routing Support.

Summary

This chapter covered advanced IPv4 routing topics, like Multicast Routing, the IGMP protocol, Policy Routing, and Multipath Routing. You learned about the fundamental structures of Multicast Routing, such as the multicast table (`mr_table`), the multicast forwarding cache (MFC), the Vif device, and more. You also learned what should be done to set a host to be a multicast router, and all about the use of the `ttl` field in Multicast Routing. Chapter 7 deals with the Linux neighbouring subsystem. The "Quick Reference" section that follows covers the top methods related to the topics discussed in this chapter, ordered by their context.

Quick Reference

I conclude this chapter with a short list of important routing subsystem methods (some of which were mentioned in this chapter), a list of macros, and `procfs` multicast entries and tables.

Methods

Let's start with the methods:

int ip_mroute_setsockopt(struct sock *sk, int optname, char __user *optval, unsigned int optlen);

This method handles `setsockopt()` calls from the multicast routing daemon. The supported socket options are: MRT_INIT, MRT_DONE, MRT_ADD_VIF, MRT_DEL_VIF, MRT_ADD_MFC, MRT_DEL_MFC, MRT_ADD_MFC_PROXY, MRT_DEL_MFC_PROXY, MRT_ASSERT, MRT_PIM (when PIM support is set), and MRT_TABLE (when Multicast Policy Routing is set).

int ip_mroute_getsockopt(struct sock *sk, int optname, char __user *optval, int __user *optlen);

This method handles `getsockopt()` calls from the multicast routing daemon. The supported socket options are MRT_VERSION, MRT_ASSERT and MRT_PIM.

struct mr_table *ipmr_new_table(struct net *net, u32 id);

This method creates a new multicast routing table. The `id` of the table will be the specified `id.`

void ipmr_free_table(struct mr_table *mrt);

This method frees the specified multicast routing table and the resources attached to it.

int ip_mc_join_group(struct sock *sk , struct ip_mreqn *imr);

This method is for joining a multicast group. The address of the multicast group to be joined is specified in the given `ip_mreqn` object. The method returns 0 on success.

static struct mfc_cache *ipmr_cache_find(struct mr_table *mrt, __be32 origin, __be32 mcastgrp);

This method performs a lookup in the IPv4 multicast routing cache. It returns NULL when no entry is found.

bool ipv4_is_multicast(__be32 addr);

This method returns `true` if the address is a multicast address.

int ip_mr_input(struct sk_buff *skb);

This method is the main IPv4 multicast Rx method (`net/ipv4/ipmr.c`).

struct mfc_cache *ipmr_cache_alloc(void);

This method allocates a multicast forwarding cache (`mfc_cache`) entry.

static struct mfc_cache *ipmr_cache_alloc_unres(void);

This method allocates a multicast routing cache (`mfc_cache`) entry for the unresolved cache and sets the `expires` field of the queue of unresolved entries.

void fib_select_multipath(struct fib_result *res);

This method is called to determine the nexthop when working with Multipath Routing.

int dev_set_allmulti(struct net_device *dev, int inc);

This method increments/decrements the `allmulti` counter of the specified network device according to the specified increment (the increment can be a positive number or a negative number).

int igmp_rcv(struct sk_buff *skb);

This method is the receive handler for IGMP packets.

static int ipmr_mfc_add(struct net *net, struct mr_table *mrt, struct mfcctl *mfc, int mrtsock, int parent);

This method adds a multicast cache entry; it is invoked by calling `setsockopt()` from userspace with MRT_ADD_MFC.

static int ipmr_mfc_delete(struct mr_table *mrt, struct mfcctl *mfc, int parent);

This method deletes a multicast cache entry; it is invoked by calling `setsockopt()` from userspace with MRT_DEL_MFC.

static int vif_add(struct net *net, struct mr_table *mrt, struct vifctl *vifc, int mrtsock);

This method adds a multicast virtual interface; it is invoked by calling `setsockopt()` from userspace with MRT_ADD_VIF.

static int vif_delete(struct mr_table *mrt, int vifi, int notify, struct list_head *head);

This method deletes a multicast virtual interface; it is invoked by calling `setsockopt()` from userspace with MRT_DEL_VIF.

static void ipmr_expire_process(unsigned long arg);

This method removes expired entries from the queue of unresolved entries.

static int ipmr_cache_report(struct mr_table *mrt, struct sk_buff *pkt, vifi_t vifi, int assert);

This method builds an IGMP packet, setting the type in the IGMP header to be the specified `assert` value and the code to be `0`. This IGMP packet is delivered to the userspace multicast routing daemon by calling the `sock_queue_rcv_skb()` method. The `assert` parameter can be assigned one of these values: IGMPMSG_NOCACHE, when an unresolved cache entry is added to the queue of unresolved entries and wants to notify the userspace routing daemon that it should resolve it, IGMPMSG_WRONGVIF, and IGMPMSG_WHOLEPKT.

static int ipmr_device_event(struct notifier_block *this, unsigned long event, void *ptr);

This method is a notifier callback which is registered by the `register_netdevice_notifier()` method; when some network device is unregistered, a NETDEV_UNREGISTER event is generated; this callback receives this event and deletes the `vif_device` objects in the `vif_table`, whose device is the one that was unregistered.

static void mrtsock_destruct(struct sock *sk);

This method is called when the userspace routing daemon calls `setsockopt()` with MRT_DONE. This method nullifies the multicast routing socket (`mroute_sk` of the multicast routing table), decrements the `mc_forwarding` `procfs` entry, and calls the `mroute_clean_tables()` method to free resources.

Macros

This section describes our macros.

MFC_HASH(a,b)

This macro calculates the hash value for adding entries to the MFC cache. It takes the group multicast address and the source IPv4 address as parameters.

VIF_EXISTS(_mrt, _idx)

This macro checks the existence of an entry in the `vif_table`; it returns `true` if the array of multicast virtual devices (`vif_table`) of the specified multicast routing table (`mrt`) has an entry with the specified index (`_idx`).

Procfs Multicast Entries

The following is a description of two important `procfs` multicast entries:

/proc/net/ip_mr_vif

Lists all the multicast virtual interfaces; it displays all the `vif_device` objects in the multicast virtual device table (`vif_table`). Displaying the `/proc/net/ip_mr_vif entry` is handled by the `ipmr_vif_seq_show()` method.

/proc/net/ip_mr_cache

The state of the Multicast Forwarding Cache (MFC). This entry shows the following fields of all the cache entries: group multicast address (`mfc_mcastgrp`), source IP address (`mfc_origin`), input interface index (`mfc_parent`), forwarded packets (`mfc_un.res.pkt`), forwarded bytes (`mfc_un.res.bytes`), wrong interface index (`mfc_un.res.wrong_if`), the index of the forwarding interface (an index in the `vif_table`), and the entry in the `mfc_un.res.ttls` array corresponding to this index. Displaying the `/proc/net/ip_mr_cache` entry is handled by the `ipmr_mfc_seq_show()` method.

Table

And finally, here in Table 6-1, is the table of rule selectors.

Table 6-1. *IP Rule Selectors*

Linux Symbol	Selector	Member of fib_rule	fib4_rule
FRA_SRC	from	src	(fib4_rule)
FRA_DST	to	dst	(fib4_rule)
FRA_IIFNAME	iif	iifname	(fib_rule)
FRA_OIFNAME	oif	oifname	(fib_rule)
FRA_FWMARK	fwmark	mark	(fib_rule)
FRA_FWMASK	fwmark/fwmask	mark_mask	(fib_rule)
FRA_PRIORITY	preference,order,priority	pref	(fib_rule)
-	tos, dsfield	tos	(fib4_rule)

CHAPTER 7

Linux Neighbouring Subsystem

This chapter discusses the Linux neighbouring subsystem and its implementation in Linux. The neighbouring subsystem is responsible for the discovery of the presence of nodes on the same link and for translation of L3 (network layer) addresses to L2 (link layer) addresses. L2 addresses are needed to build the L2 header for outgoing packets, as described in the next section. The protocol that implements this translation is called the Address Resolution Protocol (ARP) in IPv4 and Neighbour Discovery protocol (NDISC or ND) in IPv6. The neighbouring subsystem provides a protocol-independent infrastructure for performing L3-to-L2 mappings. The discussion in this chapter, however, is restricted to the most common cases—namely, the neighbouring subsystem usage in IPv4 and in IPv6. Keep in mind that the ARP protocol, like the ICMP protocol discussed in Chapter 3, is subject to security threats—such as ARP poisoning attacks and ARP spoofing attacks (security aspects of the ARP protocol are beyond the scope of this book).

I first discuss the common neighbouring data structures in this chapter and some important API methods, which are used both in IPv4 and in IPv6. Then I discuss the particular implementations of the ARP protocol and NDISC protocol. You will see how a neighbour is created and how it is freed, and you will learn about the interaction between userspace and the neighbouring subsystem. You will also learn about ARP requests and ARP replies, about NDISC neighbour solicitation and NDISC neighbour advertisements, and about a mechanism called Duplicate Address Detection (DAD), which is used by the NDISC protocol to avoid duplicate IPv6 addresses.

The Neighbouring Subsystem Core

What is the neighbouring subsystem needed for? When a packet is sent over the L2 layer, the L2 destination address is needed to build an L2 header. Using the neighbouring subsystem solicitation requests and solicitation replies, the L2 address of a host can be found out given its L3 address (or the fact that such L3 address does not exist). In Ethernet, which is the most commonly used link layer (L2), the L2 address of a host is its MAC address. In IPv4, ARP is the neighbouring protocol, and solicitation requests and solicitation replies are called ARP requests and ARP replies, respectively. In IPv6, the neighbouring protocol is NDISC, and solicitation requests and solicitation replies are called neighbour solicitations and neighbour advertisements, respectively.

There are cases where the destination address can be found without any help from the neighbouring subsystem—for example, when a broadcast is sent. In this case, the destination L2 address is fixed (for example, it is FF:FF:FF:FF:FF:FF in Ethernet). Or when the destination address is a multicast address, there is a fixed mapping between the L3 multicast address to its L2 address. I discuss such cases in the course of this chapter.

The basic data structure of the Linux neighbouring subsystem is the neighbour. A *neighbour* represents a network node that is attached to the same link (L2). It is represented by the `neighbour` structure. This representation is not unique for a particular protocol. However, as mentioned, the discussion of the `neighbour` structure will be restricted to its use in the IPv4 and in the IPv6 protocols. Let's take a look in the `neighbour` structure:

```
struct neighbour {
        struct neighbour __rcu  *next;
        struct neigh_table      *tbl;
        struct neigh_parms      *parms;
        unsigned long           confirmed;
        unsigned long           updated;
        rwlock_t                lock;
        atomic_t                refcnt;
        struct sk_buff_head     arp_queue;
        unsigned int            arp_queue_len_bytes;
        struct timer_list       timer;
        unsigned long           used;
        atomic_t                probes;
        __u8                    flags;
        __u8                    nud_state;
        __u8                    type;
        __u8                    dead;
        seqlock_t               ha_lock;
        unsigned char           ha[ALIGN(MAX_ADDR_LEN, sizeof(unsigned long))];
        struct hh_cache         hh;
        int                     (*output)(struct neighbour *, struct sk_buff *);
        const struct neigh_ops  *ops;
        struct rcu_head         rcu;
        struct net_device       *dev;
        u8                      primary_key[0];
};
```

(include/net/neighbour.h)

The following is a description of some of the important members of the neighbour structure:

- next: A pointer to the next neighbour on the same bucket in the hash table.
- tbl: The neighbouring table associated to this neighbour.
- parms: The neigh_parms object associated to this neighbour. It is initialized by the constructor method of the associated neighbouring table. For example, in IPv4 the arp_constructor() method initializes parms to be the arp_parms of the associated network device. Do not confuse it with the neigh_parms object of the neighbouring table.
- confirmed: Confirmation timestamp (discussed later in this chapter).
- refcnt: Reference counter. Incremented by the neigh_hold() macro and decremented by the neigh_release() method. The neigh_release() method frees the neighbour object by calling the neigh_destroy() method only if after decrementing the reference counter its value is 0.
- arp_queue: A queue of unresolved SKBs. Despite the name, this member is not unique to ARP and is used by other protocols, such as the NDISC protocol.
- timer: Every neighbour object has a timer; the timer callback is the neigh_timer_handler() method. The neigh_timer_handler() method can change the Network Unreachability Detection (NUD) state of the neighbour. When sending solicitation requests, and the state of the neighbour is NUD_INCOMPLETE or NUD_PROBE, and the number of solicitation requests probes is higher or equal to neigh_max_probes(), then the state of the neighbour is set to be NUD_FAILED, and the neigh_invalidate() method is invoked.
- ha_lock: Provides access protection to the neighbour hardware address (ha).

- ha: The hardware address of the neighbour object; in the case of Ethernet, it is the MAC address of the neighbour.
- hh: A hardware header cache of the L2 header (An hh_cache object).
- output: A pointer to a transmit method, like the neigh_resolve_output() method or the neigh_direct_output() method. It is dependent on the NUD state and as a result can be assigned to different methods during a neighbour lifetime. When initializing the neighbour object in the neigh_alloc() method, it is set to be the neigh_blackhole() method, which discards the packet and returns -ENETDOWN.

  And here are the helper methods (methods which set the output callback):

  - void neigh_connect(struct neighbour *neigh)

    Sets the output() method of the specified neighbour to be neigh->ops->connected_output.

  - void neigh_suspect(struct neighbour *neigh)

    Sets the output() method of the specified neighbour to be neigh->ops->output.

- nud_state: The NUD state of the neighbour. The nud_state value can be changed dynamically during the lifetime of a neighbour object. Table 7-1 in the "Quick Reference" section at the end of this chapter describes the basic NUD states and their Linux symbols. The NUD state machine is very complex; I do not delve into all of its nuances in this book.
- dead: A flag that is set when the neighbour object is alive. It is initialized to 0 when creating a neighbour object, at the end of the __neigh_create() method. The neigh_destroy() method will fail for neighbour objects whose dead flag is not set. The neigh_flush_dev() method sets the dead flag to 1 but does not yet remove the neighbour entry. The removal of neighbours marked as dead (their dead flag is set) is done later, by the garbage collectors.
- primary_key: The IP address (L3) of the neighbour. A lookup in the neighbouring tables is done with the primary_key. The primary_key length is based on which protocol is used. For IPv4, for example, it should be 4 bytes. For IPv6 it should be sizeof(struct in6_addr), as the in6_addr structure represents an IPv6 address. Therefore, the primary_key is defined as an array of 0 bytes, and when allocating a neighbour it should be taken into account which protocol is used. See the explanation about entry_size and key_len later in this chapter, in the description of the neigh_table structure members.

To avoid sending solicitation requests for each new packet that is transmitted, the kernel keeps the mapping between L3 addresses and L2 addresses in a data structure called a neighbouring table; in the case of IPv4, it is the ARP table (sometimes also called the ARP cache, though they are the same)—in contrast to what you saw in the IPv4 routing subsystem in Chapter 5: the routing cache, before it was removed, and the routing table, were two different entities, which were represented by two different data structures. In the case of IPv6, the neighbouring table is the NDISC table (also known as the NDISC cache). Both the ARP table (arp_tbl) and the NDISC table (nd_tbl) are instances of the neigh_table structure. Let's take a look at the neigh_table structure:

```
struct neigh_table {
        struct neigh_table      *next;
        int                     family;
        int                     entry_size;
        int                     key_len;
        __u32                   (*hash)(const void *pkey,
                                        const struct net_device *dev,
                                        __u32 *hash_rnd);
        int                     (*constructor)(struct neighbour *);
        int                     (*pconstructor)(struct pneigh_entry *);
        void                    (*pdestructor)(struct pneigh_entry *);
        void                    (*proxy_redo)(struct sk_buff *skb);
        char                    *id;
        struct neigh_parms      parms;
        /* HACK. gc_* should follow parms without a gap! */
        int                     gc_interval;
        int                     gc_thresh1;
        int                     gc_thresh2;
        int                     gc_thresh3;
        unsigned long           last_flush;
        struct delayed_work     gc_work;
        struct timer_list       proxy_timer;
        struct sk_buff_head     proxy_queue;
        atomic_t                entries;
        rwlock_t                lock;
        unsigned long           last_rand;
        struct neigh_statistics __percpu *stats;
        struct neigh_hash_table __rcu *nht;
        struct pneigh_entry     **phash_buckets;
};
```

(include/net/neighbour.h)

Here are some important members of the neigh_table structure:

- next: Each protocol creates its own neigh_table instance. There is a linked list of all the neighbouring tables in the system. The neigh_tables global variable is a pointer to the beginning of the list. The next variable points to the next item in this list.
- family: The protocol family: AF_INET for the IPv4 neighbouring table (arp_tbl), and AF_INET6 for the IPv6 neighbouring table (nd_tbl).
- entry_size: When allocating a neighbour entry by the neigh_alloc() method, the size for allocation is tbl->entry_size + dev->neigh_priv_len. Usually the neigh_priv_len value is 0. Before kernel 3.3, the entry_size was explicitly initialized to be sizeof(struct neighbour) + 4 for ARP, and sizeof(struct neighbour) + sizeof(struct in6_addr) for NDISC. The reason for this initialization was that when allocating a neighbour, you want to allocate space also for the primary_key[0] member. From kernel 3.3, the enrty_size was removed from the static initialization of arp_tbl and ndisc_tbl, and the entry_size initialization is done based on the key_len in the core neighbouring layer, by the neigh_table_init_no_netlink() method.
- key_len: The size of the lookup key; it is 4 bytes for IPv4, because the length of IPv4 address is 4 bytes, and it is sizeof(struct in6_addr) for IPv6. The in6_addr structure represents an IPv6 address.
- hash: The hash function for mapping a key (L3 address) to a specific hash value; for ARP it is the arp_hash() method. For NDISC it is the ndisc_hash() method.
- constructor: This method performs protocol-specific initialization when creating a neighbour object. For example, arp_constructor() for ARP in IPv4 and ndisc_constructor() for NDISC in IPv6. The constructor callback is invoked by the __neigh_create() method. It returns 0 on success.

- pconstructor: A method for creation of a neighbour proxy entry; it is not used by ARP, and it is pndisc_constructor for NDISC. This method should return 0 upon success. The pconstructor method is invoked from the pneigh_lookup() method if the lookup fails, on the condition that the pneigh_lookup() was invoked with creat = 1.
- pdestructor: A method for destroying a neighbour proxy entry. Like the pconstructor callback, the pdestructor is not used by ARP, and it is pndisc_destructor for NDISC. The pdestructor method is invoked from the pneigh_delete() method and from the pneigh_ifdown() method.
- id: The name of the table; it is arp_cache for IPv4 and ndisc_cache for IPv6.
- parms: A neigh_parms object: each neighbouring table has an associated neigh_parms object, which consists of various configuration settings, like reachability information, various timeouts, and more. The neigh_parms initialization is different in the ARP table and in the NDISC table.
- gc_interval: Not used directly by the neighbouring core.
- gc_thresh1, gc_thresh2, gc_thresh3: Thresholds of the number of neighbouring table entries. Used as criteria to activation of the synchronous garbage collector (neigh_forced_gc) and in the neigh_periodic_work() asynchronous garbage collector handler. See the explanation about allocating a neighbour object in the "Creating and Freeing a Neighbour" section later in this chapter. In the ARP table, the default values are: gc_thresh1 is 128, gc_thresh2 is 512, and gc_thresh3 is 1024. These values can be set by procfs. The same default values are also used in the NDISC table in IPv6. The IPv4 procfs entries are:
  - /proc/sys/net/ipv4/neigh/default/gc_thresh1
  - /proc/sys/net/ipv4/neigh/default/gc_thresh2
  - /proc/sys/net/ipv4/neigh/default/gc_thresh3

  and for IPv6, these are the procfs entries:
  - /proc/sys/net/ipv6/neigh/default/gc_thresh1
  - /proc/sys/net/ipv6/neigh/default/gc_thresh2
  - /proc/sys/net/ipv6/neigh/default/gc_thresh3
- last_flush: The most recent time when the neigh_forced_gc() method ran. It is initialized to be the current time (jiffies) in the neigh_table_init_no_netlink () method.
- gc_work: Asynchronous garbage collector handler. Set to be the neigh_periodic_work() timer by the neigh_table_init_no_netlink() method. The delayed_work struct is a type of a work queue. Before kernel 2.6.32, the neigh_periodic_timer() method was the asynchronous garbage collector handler; it processed only one bucket and not the entire neighbouring hash table. The neigh_periodic_work() method first checks whether the number of the entries in the table is less than gc_thresh1, and if so, it exits without doing anything; then it recomputes the reachable time (the reachable_time field of parms, which is the neigh_parms object associated with the neighbouring table). Then it scans the neighbouring hash table and removes entries which their state is not NUD_PERMANENT or NUD_IN_TIMER, and which their reference count is 1, and if one of these conditions is met: either they are in the NUD_FAILED state or the current time is after their used timestamp + gc_staletime (gc_staletime is a member of the neighbour parms object). Removal of the neighbour entry is done by setting the dead flag to 1 and calling the neigh_cleanup_and_release() method.

- proxy_timer: When a host is configured as an ARP proxy, it is possible to avoid immediate processing of solicitation requests and to process them with some delay. This is due to the fact that for an ARP proxy host, there can be a large number of solicitation requests (as opposed to the case when the host is not an ARP proxy, when you usually have a small amount of ARP requests). Sometimes you may prefer to delay the reply to such broadcasts so that you can give priority to hosts that own such IP addresses to be the first to get the request. This delay is a random value up to the proxy_delay parameter. The ARP proxy timer handler is the neigh_proxy_process() method. The proxy_timer is initialized by the neigh_table_init_no_netlink() method.
- proxy_queue: Proxy ARP queue of SKBs. SKBs are added with the pneigh_enqueue() method.
- stats: The neighbour statistics (neigh_statistics) object; consists of per CPU counters like allocs, which is the number of neighbour objects allocated by the neigh_alloc() method, or destroys, which is the number of neighbour objects which were freed by the neigh_destroy() method, and more. The neighbour statistics counters are incremented by the NEIGH_CACHE_STAT_INC macro. Note that because the statistics are per CPU counters, the macro this_cpu_inc() is used by this macro. You can display the ARP statistics and the NDISC statistics with cat /proc/net/stat/arp_cache and cat/proc/net/stat/ndisc_cache, respectively. In the "Quick Reference" section at the end of this chapter, there is a description of the neigh_statistics structure, specifying in which method each counter is incremented.
- nht: The neighbour hash table (neigh_hash_table object).
- phash_buckets: The neighbouring proxy hash table; allocated in the neigh_table_init_no_netlink() method.

The initialization of the neighbouring table is done with the neigh_table_init() method:

- In IPv4, the ARP module defines the ARP table (an instance of the neigh_table structure named arp_tbl) and passes it as an argument to the neigh_table_init() method (see the arp_init() method in net/ipv4/arp.c).
- In IPv6, the NDISC module defines the NDSIC table (which is also an instance of the neigh_table structure named nd_tbl) and passes it as an argument to the neigh_table_init() method (see the ndisc_init() method in net/ipv6/ndisc.c).

The neigh_table_init() method also creates the neighbouring hash table (the nht object) by calling the neigh_hash_alloc() method in the neigh_table_init_no_netlink() method, allocating space for eight hash entries:

```
static void neigh_table_init_no_netlink(struct neigh_table *tbl)
{
    . . .
    RCU_INIT_POINTER(tbl->nht, neigh_hash_alloc(3));
    . . .
}

static struct neigh_hash_table *neigh_hash_alloc(unsigned int shift)
{
```

The size of the hash table is 1<< shift (when size <= PAGE_SIZE):

```
    size_t size = (1 << shift) * sizeof(struct neighbour *);
    struct neigh_hash_table *ret;
    struct neighbour __rcu **buckets;
    int i;
    ret = kmalloc(sizeof(*ret), GFP_ATOMIC);
    if (!ret)
        return NULL;
    if (size <= PAGE_SIZE)
        buckets = kzalloc(size, GFP_ATOMIC);
    else
        buckets = (struct neighbour __rcu **)
              __get_free_pages(GFP_ATOMIC | __GFP_ZERO,
                       get_order(size));
    . . .

}
```

You may wonder why you need the neigh_table_init_no_netlink() method—why not perform all of the initialization in the neigh_table_init() method? The neigh_table_init_no_netlink() method performs all of the initializations of the neighbouring tables, except for linking it to the global linked list of neighbouring tables, neigh_tables. Originally such initialization, without linking to the neigh_tables linked list, was needed for ATM, and as a result the neigh_table_init() method was split, and the ATM clip module called the neigh_table_init_no_netlink() method instead of calling the neigh_table_init() method; however, over time, a different solution was found in ATM. Though the ATM clip module does not invoke the neigh_table_init_no_netlink() method anymore, the split of these methods remained, perhaps in case it is needed in the future.

I should mention that each L3 protocol that uses the neighbouring subsystem also registers a protocol handler: for IPv4, the handler for ARP packets (packets whose type in their Ethernet header is 0x0806) is the arp_rcv() method:

```
static struct packet_type arp_packet_type __read_mostly = {
         .type = cpu_to_be16(ETH_P_ARP),
         .func = arp_rcv,
 };

 void __init arp_init(void)
 {
     . . .
         dev_add_pack(&arp_packet_type);
     . . .
}
```

(net/ipv4/arp.c)

For IPv6, the neighbouring messages are ICMPv6 messages, so they are handled by the icmpv6_rcv() method, which is the ICMPv6 handler. There are five ICMPv6 neighbouring messages; when each of them is received (by the icmpv6_rcv() method), the ndisc_rcv() method is invoked to handle them (see net/ipv6/icmp.c). The ndisc_rcv() method is discussed in a later section in this chapter. Each neighbour object defines a set of methods by the neigh_ops structure. This is done by its constructor method. The neigh_ops structure contains a protocol family member and four function pointers:

```
struct neigh_ops {
        int      family;
        void     (*solicit)(struct neighbour *, struct sk_buff *);
        void     (*error_report)(struct neighbour *, struct sk_buff *);
        int      (*output)(struct neighbour *, struct sk_buff *);
        int      (*connected_output)(struct neighbour *, struct sk_buff *);
};
```

(include/net/neighbour.h)

- family: AF_INET for IPv4 and AF_INET6 for IPv6.
- solicit: This method is responsible for sending the neighbour solicitation requests: in ARP it is the arp_solicit() method, and in NDISC it is the ndisc_solicit() method.
- error_report: This method is called from the neigh_invalidate() method when the neighbour state is NUD_FAILED. This happens, for example, after some timeout when a solicitation request is not replied.
- output: When the L3 address of the next hop is known, but the L2 address is not resolved, the output callback should be neigh_resolve_output().
- connected_output: The output method of the neighbour is set to be connected_output() when the neighbour state is NUD_REACHABLE or NUD_CONNECTED. See the invocations of neigh_connect() in the neigh_update() method and in the neigh_timer_handler() method.

Creating and Freeing a Neighbour

A neighbour is created by the __neigh_create() method:

```
struct neighbour *__neigh_create(struct neigh_table *tbl, const void *pkey, struct
net_device *dev, bool want_ref)
```

First, the __neigh_create() method allocates a neighbour object by calling the neigh_alloc() method, which also performs various initializations. There are cases when the neigh_alloc() method calls the synchronous garbage collector (which is the neigh_forced_gc() method):

```
static struct neighbour *neigh_alloc(struct neigh_table *tbl, struct net_device *dev)
{
        struct neighbour *n = NULL;
        unsigned long now = jiffies;
        int entries;

        entries = atomic_inc_return(&tbl->entries) - 1;
```

If the number of table entries is greater than gc_thresh3 (1024 by default) or if the number of table entries is greater than gc_thresh2 (512 by default), and the time passed since the last flush is more than 5 Hz, the synchronous garbage collector method is invoked (the neigh_forced_gc() method). If after running the neigh_forced_gc() method, the number of table entries is greater than gc_thresh3 (1024), you do not allocate a neighbour object and return NULL:

```
        if (entries >= tbl->gc_thresh3 ||
            (entries >= tbl->gc_thresh2 &&
            time_after(now, tbl->last_flush + 5 * HZ))) {
               if (!neigh_forced_gc(tbl) &&
                   entries >= tbl->gc_thresh3)
                       goto out_entries;
        }
```

Then the __neigh_create() method performs the protocol-specific setup by calling the constructor method of the specified neighbouring table (arp_constructor() for ARP, ndisc_constructor() for NDISC). In the constructor method, special cases like multicast or loopback addresses are handled. In the `arp_constructor()` method, for example, you call the `arp_mc_map()` method to set the hardware address of the neighbour (`ha`) according to the neighbour IPv4 `primary_key` address, and you set the `nud_state` to be NUD_NOARP, because multicast addresses don't need ARP. In the `ndisc_constructor()` method, for example, you do something quite similar when handling multicast addresses: you call the `ndisc_mc_map()` to set the hardware address of the neighbour (`ha`) according to the neighbour IPv6 `primary_key` address, and you again set the `nud_state` to be NUD_NOARP. There's also special treatment for broadcast addresses: in the `arp_constructor()` method, for example, when the neighbour type is RTN_BROADCAST, you set the neighbour hardware address (`ha`) to be the network device broadcast address (the `broadcast` field of the `net_device` object), and you set the `nud_state` to be NUD_NOARP. Note that the IPv6 protocol does not implement traditional IP broadcast, so the notion of a broadcast address is irrelevant (there is a link-local all nodes multicast group at address `ff02::1`, though). There are two special cases when additional setup needs to be done:

- When the `ndo_neigh_construct()` callback of the `netdev_ops` is defined, it is invoked. In fact, this is done only in the classical IP over ATM code (`clip`); see `net/atm/clip.c`.
- When the `neigh_setup()` callback of the `neigh_parms` object is defined, it is invoked. This is used, for example, in the bonding driver; see `drivers/net/bonding/bond_main.c`.

When trying to create a `neighbour` object by the `__neigh_create()` method, and the number of the neighbour entries exceeds the hash table size, it must be enlarged. This is done by calling the `neigh_hash_grow()` method, like this:

```
struct neighbour *__neigh_create(struct neigh_table *tbl, const void *pkey,
                 struct net_device *dev, bool want_ref)
{
     . . .
```

The hash table size is `1 << nht->hash_shift`; the hash table must be enlarged if it is exceeded:

```
     if (atomic_read(&tbl->entries) > (1 << nht->hash_shift))
        nht = neigh_hash_grow(tbl, nht->hash_shift + 1);
     . . .
}
```

When the `want_ref` parameter is true, you will increment the neighbour reference count within this method. You also initialize the `confirmed` field of the `neighbour` object:

```
n->confirmed = jiffies - (n->parms->base_reachable_time << 1);
```

It is initialized to be a little less than the current time, `jiffies` (for the simple reason that you want reachability confirmation to be required sooner). At the end of the `__neigh_create()` method, the `dead` flag is initialized to be 0, and the `neighbour` object is added to the neighbour hash table.

The `neigh_release()` method decrements the reference counter of the neighbour and frees it when it reaches zero by calling the `neigh_destroy()` method. The `neigh_destroy()` method will verify that the neighbour is marked as `dead`: neighbours whose `dead` flag is 0 will not be removed.

In this section, you learned about the kernel methods to create and free a neighbour. Next you will learn how adding and deleting a neighbour entry can be triggered from userspace, as well as how to display the neighbouring table, with the `arp` command for IPv4 and the `ip` command for IPv4/IPv6.

Interaction Between Userspace and the Neighbouring Subsystem

Management of the ARP table is done with the `ip neigh` command of the `iproute2` package or with the `arp` command of the `net-tools` package. Thus, you can display the ARP table by running, from the command line, one of the following commands:

- `arp`: Handled by the `arp_seq_show()` method in `net/ipv4/arp.c`.
- `ip neigh show` (or `ip neighbour show`): Handled by the `neigh_dump_info()` method in `net/core/neighbour.c`.

Note that the `ip neigh show` command shows the NUD states of the neighbouring table entries (like NUD_REACHABLE or NUD_STALE). Note also that the `arp` command can display only the IPv4 neighbouring table (the ARP table), whereas with the `ip` command you can display both the IPv4 ARP table and the IPv6 neighbouring table. If you want to display only the IPv6 neighbouring table, you should run `ip -6 neigh show`.

The ARP and NDISC modules also export data via `procfs`. That means you can display the ARP table by running `cat /proc/net/arp` (this `procfs` entry is handled by the `arp_seq_show()` method, which is the same method that handles the `arp` command, as mentioned earlier). Or you can display ARP statistics by `cat /proc/net/stat/arp_cache`, and you can display the NDISC statistics by `cat /proc/net/stat/ndisc_cache` (both are handled by the `neigh_stat_seq_show()` method).

You can add an entry with `ip neigh add`, which is handled by the `neigh_add()` method. When running `ip neigh add`, you can specify the state of the entry which you are adding (like NUD_PERMANENT, NUD_STALE, NUD_REACHABLE and so on). For example:

```
ip neigh add 192.168.0.121 dev eth0 lladdr 00:30:48:5b:cc:45 nud permanent
```

Deleting an entry can be done by `ip neigh del`, and is handled by the `neigh_delete()` method. For example:

```
ip neigh del 192.168.0.121 dev eth0
```

Adding an entry to the proxy ARP table can be done with `ip neigh add proxy`. For example:

```
ip neigh add proxy 192.168.2.11 dev eth0
```

The addition is handled again by the `neigh_add()` method. In this case, the NTF_PROXY flag is set in the data passed from userspace (see the `ndm_flags` field of the `ndm` object), and therefore the `pneigh_lookup()` method is called to perform a lookup in the proxy neighbouring hash table (`phash_buckets`). In case the lookup failed, the `pneigh_lookup()` method adds an entry to the proxy neighbouring hash table.

Deleting an entry from the proxy ARP table can be done with `ip neigh del proxy`. For example:

```
ip neigh del proxy 192.168.2.11 dev eth0
```

The deletion is handled by the `neigh_delete()` method. Again, in this case the NTF_PROXY flag is set in the data passed from userspace (see the `ndm_flags` field of the `ndm` object), and therefore the `pneigh_delete()` method is called to delete the entry from the proxy neighbouring table.

With the `ip ntable` command, you can control the parameters for the neighbouring tables. For example:

- `ip ntable show`: Shows the parameters for all the neighbouring tables.
- `ip ntable change`: Change a value of a parameter of a neighbouring table. Handled by the `neightbl_set()` method. For example: `ip ntable change name arp_cache queue 20 dev eth0`.

You can also add entries to the ARP table by `arp add`. And it is possible to add static entries manually to the ARP table, like this: `arp -s <IPAddress> <MacAddress>`. The static ARP entries are not deleted by the neigbouring subsystem garbage collector, but they are not persistent over reboot.

The next section briefly describes how network events are handled in the neighbouring subsystem.

Handling Network Events

The neighbouring core does not register any events with the `register_netdevice_notifier()` method. On the other hand, the ARP module and the NDISC module do register network events. In ARP, the `arp_netdev_event()` method is registered as the callback for `netdev` events. It handles changes of MAC address events by calling the generic `neigh_changeaddr()` method and by calling the `rt_cache_flush()` method. From kernel 3.11, you handle a NETDEV_CHANGE event when there was a change of the IFF_NOARP flag by calling the `neigh_changeaddr()` method. A NETDEV_CHANGE event is triggered when a device changes its flags, by the `__dev_notify_flags()` method, or when a device changes its state, by the `netdev_state_change()` method. In NDISC, the `ndisc_netdev_event()` method is registered as the callback for netdev events; it handles the NETDEV_CHANGEADDR, NETDEV_DOWN, and NETDEV_NOTIFY_PEERS events.

After describing the fundamental data structures common to IPv4 and IPv6, like the neighbouring table (`neigh_table`) and the `neighbour` structure, and after discussing how a `neighbour` object is created and freed, it is time to describe the implementation of the first neighbouring protocol, the ARP protocol.

The ARP protocol (IPv4)

The ARP protocol is defined in RFC 826. When working with Ethernet, the addresses are called MAC addresses and are 48-bit values. MAC addresses should be unique, but you must take into account that you may encounter a non-unique MAC address. A common reason for this is that on most network interfaces, a system administrator can configure MAC addresses with userspace tools like `ifconfig` or `ip`.

When sending an IPv4 packet, you know the destination IPv4 address. You should build an Ethernet header, which should include a destination MAC address. Finding the MAC address based on a given IPv4 address is done by the ARP protocol as you will see shortly. If the MAC address is unknown, you send an ARP request as a broadcast. This ARP request contains the IPv4 address you are seeking. If there is a host with such an IPv4 address, this host sends a unicast ARP response as a reply. The ARP table (`arp_tbl`) is an instance of the `neigh_table` structure. The ARP header is represented by the `arphdr` structure:

```
struct arphdr {
    __be16          ar_hrd;         /* format of hardware address   */
    __be16          ar_pro;         /* format of protocol address   */
    unsigned char   ar_hln;         /* length of hardware address   */
    unsigned char   ar_pln;         /* length of protocol address   */
    __be16          ar_op;          /* ARP opcode (command)         */
#if 0
    *
    *      Ethernet looks like this : This bit is variable sized however...
    */
    unsigned char           ar_sha[ETH_ALEN];       /* sender hardware address      */
    unsigned char           ar_sip[4];              /* sender IP address            */
    unsigned char           ar_tha[ETH_ALEN];       /* target hardware address      */
    unsigned char           ar_tip[4];              /* target IP address            */
#endif
};
```

(include/uapi/linux/if_arp.h)

The following is a description of some of the important members of the arphdr structure:

- ar_hrd is the hardware type; for Ethernet it is 0x01. For the full list of available ARP header hardware identifiers, see ARPHRD_XXX definitions in include/uapi/linux/if_arp.h.
- ar_pro is the protocol ID; for IPv4 it is 0x80. For the full list of available protocols IDs, see ETH_P_XXX in include/uapi/linux/if_ether.h.
- ar_hln is the hardware address length in bytes, which is 6 bytes for Ethernet addresses.
- ar_pln is the length of the protocol address in bytes, which is 4 bytes for IPv4 addresses.
- ar_op is the opcode, ARPOP_REQUEST for an ARP request, and ARPOP_REPLY for an ARP reply. For the full list of available ARP header opcodes look in include/uapi/linux/if_arp.h.

Immediately after the ar_op are the sender hardware (MAC) address and IPv4 address, and the target hardware (MAC) address and IPv4 address. These addresses are not part of the ARP header (arphdr) structure. In the arp_process() method, they are extracted by reading the corresponding offsets of the ARP header, as you can see in the explanation about the arp_process() method in the section "ARP: Receiving Solicitation Requests and Replies" later in this chapter. Figure 7-1 shows an ARP header for an ARP Ethernet packet.

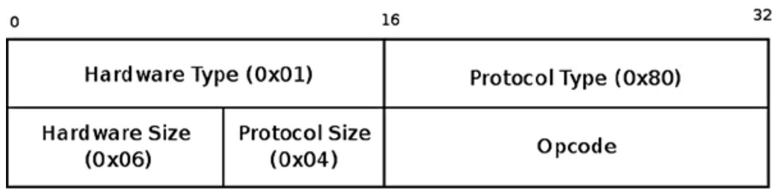


Figure 7-1. *ARP header (for Ethernet)*

In ARP, four neigh_ops objects are defined: arp_direct_ops, arp_generic_ops, arp_hh_ops, and arp_broken_ops. The initialization of the ARP table neigh_ops object is done by the arp_constructor() method, based on the network device features:

- If the header_ops of the net_device object is NULL, the neigh_ops object will be set to be arp_direct_ops. In this case, sending the packet will be done with the neigh_direct_output() method, which is in fact a wrapper around dev_queue_xmit(). In most Ethernet network devices, however, the header_ops of the net_device object is initialized to be eth_header_ops by the generic ether_setup() method; see net/ethernet/eth.c.
- If the header_ops of the net_device object contains a NULL cache() callback, then the neigh_ops object will be set to be arp_generic_ops.
- If the header_ops of the net_device object contains a non-NULL cache() callback, then the neigh_ops object will be set to be arp_hh_ops. In the case of using the generic eth_header_ops object, the cache() callback is the eth_header_cache() callback.
- For three types of devices, the neigh_ops object will be set to be arp_broken_ops (when the type of the net_device object is ARPHRD_ROSE, ARPHRD_AX25, or ARPHRD_NETROM).

Now that I've covered the ARP protocol and the ARP header (arphdr) object, let's look at how ARP solicitation requests are sent.

ARP: Sending Solicitation Requests

Where are solicitation requests being sent? The most common case is in the Tx path, before actually leaving the network layer (L3) and moving to the link layer (L2). In the `ip_finish_output2()` method, you first perform a lookup for the next hop IPv4 address in the ARP table by calling the `__ipv4_neigh_lookup_noref()` method, and if you don't find any matching neighbour entry, you create one by calling the `__neigh_create()` method:

```
static inline int ip_finish_output2(struct sk_buff *skb)
{
        struct dst_entry *dst = skb_dst(skb);
        struct rtable *rt = (struct rtable *)dst;
        struct net_device *dev = dst->dev;
        unsigned int hh_len = LL_RESERVED_SPACE(dev);
        struct neighbour *neigh;
        u32 nexthop;
        . . .
        . . .
        nexthop = (__force u32) rt_nexthop(rt, ip_hdr(skb)->daddr);
        neigh = __ipv4_neigh_lookup_noref(dev, nexthop);
        if (unlikely(!neigh))
                neigh = __neigh_create(&arp_tbl, &nexthop, dev, false);
        if (!IS_ERR(neigh)) {
                int res = dst_neigh_output(dst, neigh, skb);
     . . .
}
```

Let's take a look in the `dst_neigh_output()` method:

```
static inline int dst_neigh_output(struct dst_entry *dst, struct neighbour *n,
                                   struct sk_buff *skb)
{
        const struct hh_cache *hh;

        if (dst->pending_confirm) {
                unsigned long now = jiffies;

                dst->pending_confirm = 0;
                /* avoid dirtying neighbour */
                if (n->confirmed != now)
                        n->confirmed = now;
        }
```

When you reach this method for the first time with this flow, `nud_state` is not NUD_CONNECTED, and the output callback is the `neigh_resolve_output()` method:

```
        hh = &n->hh;
        if ((n->nud_state & NUD_CONNECTED) && hh->hh_len)
                return neigh_hh_output(hh, skb);
        else
                return n->output(n, skb);
}
```

(include/net/dst.h)

In the neigh_resolve_output() method, you call the neigh_event_send() method, which eventually puts the SKB in the arp_queue of the neighbour by __skb_queue_tail(&neigh->arp_queue, skb); later, the neigh_probe() method, invoked from the neighbour timer handler, neigh_timer_handler(), will send the packet by invoking the solicit() method (neigh->ops->solicit is the arp_solicit() method in our case):

```
static void neigh_probe(struct neighbour *neigh)
        __releases(neigh->lock)
{
        struct sk_buff *skb = skb_peek(&neigh->arp_queue);
        . . .
        neigh->ops->solicit(neigh, skb);
        atomic_inc(&neigh->probes);
        kfree_skb(skb);
}
```

Let's take a look at the arp_solicit() method, which actually sends the ARP request:

```
static void arp_solicit(struct neighbour *neigh, struct sk_buff *skb)
{
        __be32 saddr = 0;
        u8 dst_ha[MAX_ADDR_LEN], *dst_hw = NULL;
        struct net_device *dev = neigh->dev;
        __be32 target = *(__be32 *)neigh->primary_key;
        int probes = atomic_read(&neigh->probes);
        struct in_device *in_dev;

        rcu_read_lock();
        in_dev = __in_dev_get_rcu(dev);
        if (!in_dev) {
                rcu_read_unlock();
                return;
        }
```

With the arp_announce procfs entry, you can set restrictions for which local source IP address to use for the ARP packet you want to send:

- *0:* Use any local address, configured on any interface. This is the default value.
- *1:* First try to use addresses that are on the target subnet. If there are no such addresses, use level 2.
- *2:* Use primary IP address.

Note that the max value of these two entries is used:

```
/proc/sys/net/ipv4/conf/all/arp_announce
/proc/sys/net/ipv4/conf/<netdeviceName>/arp_announce
```

See also the description of the IN_DEV_ARP_ANNOUNCE macro in the “Quick Reference” section at the end of this chapter.

```
        switch (IN_DEV_ARP_ANNOUNCE(in_dev)) {
        default:
        case 0:         /* By default announce any local IP */
                if (skb && inet_addr_type(dev_net(dev),
                                          ip_hdr(skb)->saddr) == RTN_LOCAL)
                        saddr = ip_hdr(skb)->saddr;
                break;
        case 1:         /* Restrict announcements of saddr in same subnet */
                if (!skb)
                break;
                saddr = ip_hdr(skb)->saddr;
                if (inet_addr_type(dev_net(dev), saddr) == RTN_LOCAL) {
```

The `inet_addr_onlink()` method checks whether the specified target address and the specified source address are on the same subnet:

```
                /* saddr should be known to target */
                if (inet_addr_onlink(in_dev, target, saddr))
                        break;
         }
         saddr = 0;
         break;
case 2:         /* Avoid secondary IPs, get a primary/preferred one */
         break;
}
rcu_read_unlock();

if (!saddr)
```

The `inet_select_addr()` method returns the address of the first primary interface of the specified device whose scope is smaller than the specified scope (RT_SCOPE_LINK in this case), and which is in the same subnet as the target:

```
        saddr = inet_select_addr(dev, target, RT_SCOPE_LINK);

        probes -= neigh->parms->ucast_probes;
        if (probes < 0) {
                if (!(neigh->nud_state & NUD_VALID))
                        pr_debug("trying to ucast probe in NUD_INVALID\n");
                neigh_ha_snapshot(dst_ha, neigh, dev);
                dst_hw = dst_ha;
        } else {
                probes -= neigh->parms->app_probes;
                if (probes < 0) {
```

CONFIG_ARPD is set when working with the userspace ARP daemon; there are projects like OpenNHRP, which are based on ARPD. Next Hop Resolution Protocol (NHRP) is used to improve the efficiency of routing computer network traffic over Non-Broadcast, Multiple Access (NBMA) networks (I don't discuss the ARPD userspace daemon in this book):

```
#ifdef CONFIG_ARPD
                        neigh_app_ns(neigh);
#endif
                        return;
                }
        }
```

Now you call the `arp_send()` method to send an ARP request. Note that the last parameter, `target_hw`, is NULL. You do not yet know the target hardware (MAC) address. When calling `arp_send()` with `target_hw` as NULL, a broadcast ARP request is sent:

```
        arp_send(ARPOP_REQUEST, ETH_P_ARP, target, dev, saddr,
                 dst_hw, dev->dev_addr, NULL);
}
```

Let's take a look at the `arp_send()` method, which is quite short:

```
void arp_send(int type, int ptype, __be32 dest_ip,
              struct net_device *dev, __be32 src_ip,
              const unsigned char *dest_hw, const unsigned char *src_hw,
              const unsigned char *target_hw)
{
        struct sk_buff *skb;

        /*
         *      No arp on this interface.
         */
```

You must check whether the IFF_NOARP is supported on this network device. There are cases in which ARP is disabled: an administrator can disable ARP, for example, by `ifconfig eth1 -arp` or by `ip link set eth1 arp off`. Some network devices set the IFF_NOARP flag upon creation—for example, IPv4 tunnel devices, or PPP devices, which do not need ARP. See the `ipip_tunnel_setup()` method in `net/ipv4/ipip.c` or the `ppp_setup()` method in `drivers/net/ppp_generic.c`.

```
        if (dev->flags&IFF_NOARP)
                return;
```

The `arp_create()` method creates an SKB with an ARP header and initializes it according to the specified parameters:

```
        skb = arp_create(type, ptype, dest_ip, dev, src_ip,
                         dest_hw, src_hw, target_hw);
        if (skb == NULL)
                return;
```

The only thing the `arp_xmit()` method does is call `dev_queue_xmit()` by the NF_HOOK() macro:

```
        arp_xmit(skb);
}
```

Now it is time to learn how these ARP requests are processed and how ARP replies are processed.

ARP: Receiving Solicitation Requests and Replies

In IPv4, the `arp_rcv()` method is responsible for handling ARP packets, as mentioned earlier. Let's take a look at the `arp_rcv()` method:

```
static int arp_rcv(struct sk_buff *skb, struct net_device *dev,
                   struct packet_type *pt, struct net_device *orig_dev)
{
        const struct arphdr *arp;
```

If the network device on which the ARP packet was received has the IFF_NOARP flag set, or if the packet is not destined for the local machine, or if it is for a loopback device, then the packet should be dropped. You continue and make some more sanity checks, and if everything is okay, you proceed to the `arp_process()` method, which performs the real work of processing an ARP packet:

```
        if (dev->flags & IFF_NOARP ||
            skb->pkt_type == PACKET_OTHERHOST ||
            skb->pkt_type == PACKET_LOOPBACK)
                goto freeskb;
```

If the SKB is shared, you must clone it because it might be changed by someone else while being processed by the `arp_rcv()` method. The `skb_share_check()` method creates a clone of the SKB if it is shared (see Appendix A).

```
        skb = skb_share_check(skb, GFP_ATOMIC);
        if (!skb)
                goto out_of_mem;

        /* ARP header, plus 2 device addresses, plus 2 IP addresses.  */
        if (!pskb_may_pull(skb, arp_hdr_len(dev)))
                goto freeskb;

        arp = arp_hdr(skb);
```

The `ar_hln` of the ARP header represents the length of a hardware address, which should be 6 bytes for Ethernet header, and should be equal to the `addr_len` of the `net_device` object. The `ar_pln` of the ARP header represents the length of the protocol address and should be equal to the length of an IPv4 address, which is 4 bytes:

```
        if (arp->ar_hln != dev->addr_len || arp->ar_pln != 4)
                goto freeskb;

        memset(NEIGH_CB(skb), 0, sizeof(struct neighbour_cb));
        return NF_HOOK(NFPROTO_ARP, NF_ARP_IN, skb, dev, NULL, arp_process);
```

```
freeskb:
        kfree_skb(skb);
out_of_mem:
        return 0;
}
```

Handling ARP requests is not restricted to packets that have the local host as their destination. When the local host is configured as a proxy ARP, or as a private VLAN proxy ARP (see RFC 3069), you also handle packets which have a destination that is not the local host. Support for private VLAN proxy ARP was added in kernel 2.6.34.

In the `arp_process()` method, you handle only ARP requests or ARP responses. For ARP requests you perform a lookup in the routing subsystem by the `ip_route_input_noref()` method. If the ARP packet is for the local host (the `rt_type` of the routing entry is RTN_LOCAL), you proceed to check some conditions (described shortly). If all these checks pass, an ARP reply is sent back with the `arp_send()` method. If the ARP packet is not for the local host but should be forwarded (the `rt_type` of the routing entry is RTN_UNICAST), then you check some conditions (also described shortly), and if they are fulfilled you perform a lookup in the proxy ARP table by calling the `pneigh_lookup()` method.

You will now see the implementation details of the main ARP method which handles ARP requests, the `arp_process()` method.

The arp_process() Method

Let's take a look at the `arp_process()` method, where the real work is done:

```
static int arp_process(struct sk_buff *skb)
{
        struct net_device *dev = skb->dev;
        struct in_device *in_dev = __in_dev_get_rcu(dev);
        struct arphdr *arp;
        unsigned char *arp_ptr;
        struct rtable *rt;
        unsigned char *sha;
        __be32 sip, tip;
        u16 dev_type = dev->type;
        int addr_type;
        struct neighbour *n;
        struct net *net = dev_net(dev);

        /* arp_rcv below verifies the ARP header and verifies the device
         * is ARP'able.
         */

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

Fetch the ARP header from the SKB (it is the network header, see the `arp_hdr()` method):

```
        arp = arp_hdr(skb);

        switch (dev_type) {
        default:
                if (arp->ar_pro != htons(ETH_P_IP) ||
                htons(dev_type) != arp->ar_hrd)
                    goto out;
            break;
    case ARPHRD_ETHER:
            . . .
            if ((arp->ar_hrd != htons(ARPHRD_ETHER) &&
                 arp->ar_hrd != htons(ARPHRD_IEEE802)) ||
                arp->ar_pro != htons(ETH_P_IP))
                    goto out;
            break;
            . . .
```

You want to handle only ARP requests or ARP responses in the `arp_process()` method, and discard all other packets:

```
    /* Understand only these message types */

    if (arp->ar_op != htons(ARPOP_REPLY) &&
        arp->ar_op != htons(ARPOP_REQUEST))
            goto out;

/*
 *      Extract fields
 */
        arp_ptr = (unsigned char *)(arp + 1);
```

The arp_process() Method—Extracting Headers:

Immediately after the ARP header, there are the following fields (see the ARP header definition above):

- `sha`: The source hardware address (the MAC address, which is 6 bytes).
- `sip`: The source IPv4 address (4 bytes).
- `tha`: The target hardware address (the MAC address, which is 6 bytes).
- `tip`: The target IPv4 address (4 bytes).

Extract the `sip` and `tip` addresses:

```
    sha     = arp_ptr;
    arp_ptr += dev->addr_len;
```

Set `sip` to be the source IPv4 address after advancing `arp_ptr` with the corresponding offset:

```
    memcpy(&sip, arp_ptr, 4);
    arp_ptr += 4;
    switch (dev_type) {
    . . .
    default:
            arp_ptr += dev->addr_len;
    }
```

Set tip to be the target IPv4 address after advancing arp_ptr with the corresponding offset:

```
        memcpy(&tip, arp_ptr, 4);
```

Discard these two types of packets:

- Multicast packets
- Packets for the loopback device if the use of local routing with loopback addresses is disabled; see also the description of the IN_DEV_ROUTE_LOCALNET macro in the "Quick Reference" section at the end of this chapter.

```
/*
 *      Check for bad requests for 127.x.x.x and requests for multicast
 *      addresses.  If this is one such, delete it.
 */
        if (ipv4_is_multicast(tip) ||
            (!IN_DEV_ROUTE_LOCALNET(in_dev) && ipv4_is_loopback(tip)))
                goto out;

        . . .
```

The source IP (sip) is 0 when you use Duplicate Address Detection (DAD). DAD lets you detect the existence of double L3 addresses on different hosts on a LAN. DAD is implemented in IPv6 as an integral part of the address configuration process, but not in IPv4. However, there is support for correctly handling DAD requests in IPv4, as you will soon see. The arping utility of the iputils package is an example for using DAD in IPv4. When sending ARP request with arping -D, you send an ARP request where the sip of the ARP header is 0. (The -D modifier tells arping to be in DAD mode); the tip is usually the sender IPv4 address (because you want to check whether there is another host on the same LAN with the same IPv4 address as yours); if there is a host with the same IP address as the tip of the DAD ARP request, it will send back an ARP reply (without adding the sender to its neighbouring table):

```
        /* Special case: IPv4 duplicate address detection packet (RFC2131) */
        if (sip == 0) {
                if (arp->ar_op == htons(ARPOP_REQUEST) &&
```

The arp_process() Method—arp_ignore() and arp_filter() Methods

The arp_ignore procfs entry provides support for different modes for sending ARP replies as a response for an ARP request. The value used is the max value of /proc/sys/net/ipv4/conf/all/arp_ignore and /proc/sys/net/ipv4/conf/<netDeviceName>/arp_ignore. By default, the value of the arp_ignore procfs entry is 0, and in such a case, the arp_ignore() method returns 0. You reply to the ARP request with arp_send(), as you can see in the next code snippet (assuming that inet_addr_type(net, tip) returned RTN_LOCAL). The arp_ignore() method checks the value of IN_DEV_ARP_IGNORE(in_dev); for more details, see the arp_ignore() implementation in net/ipv4/arp.c and the description of the IN_DEV_ARP_IGNORE macro in the "Quick Reference" section at the end of this chapter:

```
                 inet_addr_type(net, tip) == RTN_LOCAL &&
                !arp_ignore(in_dev, sip, tip))
                arp_send(ARPOP_REPLY, ETH_P_ARP, sip, dev, tip, sha,
                         dev->dev_addr, sha);
        goto out;
}
```

```
if (arp->ar_op == htons(ARPOP_REQUEST) &&
    ip_route_input_noref(skb, tip, sip, 0, dev) == 0) {

        rt = skb_rtable(skb);
        addr_type = rt->rt_type;
```

When addr_type equals RTN_LOCAL, the packet is for local delivery:

```
        if (addr_type == RTN_LOCAL) {
                int dont_send;

                dont_send = arp_ignore(in_dev, sip, tip);
```

The arp_filter() method fails (returns 1) in two cases:

- When the lookup in the routing tables with the ip_route_output() method fails.
- When the outgoing network device of the routing entry is different than the network device on which the ARP request was received.

In case of success, the arp_filter() method returns 0 (see also the description of the IN_DEV_ARPFILTER macro in the “Quick Reference” section at the end of this chapter):

```
                     if (!dont_send && IN_DEV_ARPFILTER(in_dev))
                        dont_send = arp_filter(sip, tip, dev);
                if (!dont_send) {
```

Before sending the ARP reply, you want to add the sender to your neighbouring table or update it; this is done with the neigh_event_ns() method. The neigh_event_ns() method creates a new neighbouring table entry and sets its state to be NUD_STALE. If there is already such an entry, it updates its state to be NUD_STALE, with the neigh_update() method. Adding entries this way is termed *passive learning:*

```
                        n = neigh_event_ns(&arp_tbl, sha, &sip, dev);
                        if (n) {
                                arp_send(ARPOP_REPLY, ETH_P_ARP, sip,
                                         dev, tip, sha, dev->dev_addr,
                                         sha);
                                neigh_release(n);
                        }
                }
                goto out;
        } else if (IN_DEV_FORWARD(in_dev)) {
```

The arp_fwd_proxy() method returns 1 when the device can be used as an ARP proxy; the arp_fwd_pvlan() method returns 1 when the device can be used as an ARP VLAN proxy:

```
               if (addr_type == RTN_UNICAST  &&
                    (arp_fwd_proxy(in_dev, dev, rt) ||
                     arp_fwd_pvlan(in_dev, dev, rt, sip, tip) ||
                     (rt->dst.dev != dev &&
                      pneigh_lookup(&arp_tbl, net, &tip, dev, 0)))) {
```

Again, call the `neigh_event_ns()` method to create a neighbour entry of the sender with NUD_STALE, or if such an entry exists, update that entry state to be NUD_STALE:

```
                        n = neigh_event_ns(&arp_tbl, sha, &sip, dev);
                        if (n)
                                neigh_release(n);

                       if (NEIGH_CB(skb)->flags & LOCALLY_ENQUEUED ||
                            skb->pkt_type == PACKET_HOST ||
                            in_dev->arp_parms->proxy_delay == 0) {
                                arp_send(ARPOP_REPLY, ETH_P_ARP, sip,
                                         dev, tip, sha, dev->dev_addr,
                                         sha);
                        } else {
```

Delay sending an ARP reply by putting the SKB at the tail of the `proxy_queue`, by calling the `pneigh_enqueue()` method. Note that the delay is random and is a number between 0 and `in_dev->arp_parms->proxy_delay`:

```
                                pneigh_enqueue(&arp_tbl,
                                               in_dev->arp_parms, skb);
                                return 0;
                        }
                        goto out;
                }
        }
}

        /* Update our ARP tables */
```

Note that the last parameter of calling the `__neigh_lookup()` method is 0, which means that you only perform a lookup in the neighbouring table (and do not create a new neighbour if the lookup failed):

```
        n = __neigh_lookup(&arp_tbl, &sip, dev, 0);
```

The IN_DEV_ARP_ACCEPT macro tells you whether the network device is set to accept ARP requests (see also the description of the IN_DEV_ARP_ACCEPT macro in the “Quick Reference” section at the end of this of this chapter):

```
        if (IN_DEV_ARP_ACCEPT(in_dev)) {
                /* Unsolicited ARP is not accepted by default.
                   It is possible, that this option should be enabled for some
                   devices (strip is candidate)
                */
```

Unsolicited ARP requests are sent only to update the neighbouring table. In such requests, `tip` is equal to `sip` (the `arping` utility supports sending unsolicited ARP requests by `arping -U`):

```
        if (n == NULL &&
            (arp->ar_op == htons(ARPOP_REPLY) ||
             (arp->ar_op == htons(ARPOP_REQUEST) && tip == sip)) &&
           inet_addr_type(net, sip) == RTN_UNICAST)
               n = __neigh_lookup(&arp_tbl, &sip, dev, 1);
}
```

```
if (n) {
        int state = NUD_REACHABLE;
        int override;

        /* If several different ARP replies follows back-to-back,
           use the FIRST one. It is possible, if several proxy
           agents are active. Taking the first reply prevents
           arp trashing and chooses the fastest router.
        */
        override = time_after(jiffies, n->updated + n->parms->locktime);

        /* Broadcast replies and request packets
           do not assert neighbour reachability.
         */
         if (arp->ar_op != htons(ARPOP_REPLY) ||
             skb->pkt_type != PACKET_HOST)
                 state = NUD_STALE;
```

Call `neigh_update()` to update the neighbouring table:

```
                neigh_update(n, sha, state,
                             override ? NEIGH_UPDATE_F_OVERRIDE : 0);
                neigh_release(n);
        }

out:
        consume_skb(skb);
        return 0;
}
```

Now that you know about the IPv4 ARP protocol implementation, it is time to move on to IPv6 NDISC protocol implementation. You will soon notice some of the differences between the neighbouring subsystem implementation in IPv4 and in IPv6.

The NDISC Protocol (IPv6)

The Neighbour Discovery (NDISC) protocol is based on RFC 2461, "Neighbour Discovery for IP Version 6 (IPv6)," which was later obsoleted by RFC 4861 from 2007. IPv6 nodes (hosts or routers) on the same link use the Neighbour Discovery protocol to discover each other's presence, to discover routers, to determine each other's L2 addresses, and to maintain neighbour reachability information. Duplicate Address Detection (DAD) was added to avoid double L3 addresses on the same LAN. I discuss DAD and handling NDISC neighbour solicitation and neighbour advertisements shortly.

Next you learn how IPv6 neighbour discovery protocols avoid creating duplicate IPv6 addresses.

Duplicate Address Detection (DAD)

How can you be sure there is no other same IPv6 address on a LAN? The chances are low, but if such address does exist, it may cause trouble. DAD is a solution. When a host tries to configure an address, it first creates a Link Local address (a Link Local address starts with FE80). This address is tentative (IFA_F_TENTATIVE), which means that the host can communicate only with ND messages. Then the host starts the DAD process by calling the addrconf_dad_start() method (net/ipv6/addrconf.c). The host sends a Neighbour Solicitation DAD message. The target is its tentative address, the source is all zeros (the unspecified address). If there is no answer in a specified time interval, the state is changed to permanent (IFA_F_PERMANENT). When Optimistic DAD (CONFIG_IPV6_OPTIMISTIC_DAD) is set, you don't wait until DAD is completed, but allow hosts to communicate with peers before DAD has finished successfully. See RFC 4429, "Optimistic Duplicate Address Detection (DAD) for IPv6," from 2006.

The neighbouring table for IPv6 is called nd_tbl:

```
struct neigh_table nd_tbl = {
        .family =       AF_INET6,
        .key_len =      sizeof(struct in6_addr),
        .hash =         ndisc_hash,
        .constructor =  ndisc_constructor,
        .pconstructor = pndisc_constructor,
        .pdestructor =  pndisc_destructor,
        .proxy_redo =   pndisc_redo,
        .id =           "ndisc_cache",
        .parms = {
                .tbl                    = &nd_tbl,
                .base_reachable_time    = ND_REACHABLE_TIME,
                .retrans_time           = ND_RETRANS_TIMER,
                .gc_staletime           = 60 * HZ,
                .reachable_time         = ND_REACHABLE_TIME,
                .delay_probe_time       = 5 * HZ,
                .queue_len_bytes        = 64*1024,
                .ucast_probes           = 3,
                .mcast_probes           = 3,
                .anycast_delay          = 1 * HZ,
                .proxy_delay            = (8 * HZ) / 10,
                .proxy_qlen             = 64,
        },
        .gc_interval =    30 * HZ,
        .gc_thresh1 =    128,
        .gc_thresh2 =    512,
        .gc_thresh3 =   1024,
};
(net/ipv6/ndisc.c)
```

Note that some of the members of the NDISC table are equal to the parallel members in the ARP table—for example, the values of the garbage collector thresholds (gc_thresh1, gc_thresh2 and gc_thresh3).

The Linux IPv6 Neighbour Discovery implementation is based on ICMPv6 messages to manage the interaction between neighbouring nodes. The Neighbour Discovery protocol defines the following five ICMPv6 message types:

```
#define NDISC_ROUTER_SOLICITATION       133
#define NDISC_ROUTER_ADVERTISEMENT      134
#define NDISC_NEIGHBOUR_SOLICITATION    135
#define NDISC_NEIGHBOUR_ADVERTISEMENT   136
#define NDISC_REDIRECT                  137

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

Note that these five ICMPv6 message types are informational messages. ICMPv6 message types whose values are in the range from 0 to 127 are error messages, and ICMPv6 message types whose values are from 128 to 255 are informational messages. For more on that, see Chapter 3, which discusses the ICMP protocol. This chapter discusses only the Neighbour Solicitation and the Neighbour Discovery messages.

As mentioned in the beginning of this chapter, because neighbouring discovery messages are ICMPv6 messages, they are handled by the `icmpv6_rcv()` method, which in turn invokes the `ndisc_rcv()` method for ICMPv6 packets whose message type is one of the five types mentioned earlier (see `net/ipv6/icmp.c`).

In NDISC, there are three `neigh_ops` objects: `ndisc_generic_ops`, `ndisc_hh_ops`, and `ndisc_direct_ops`:

- If the `header_ops` of the `net_device` object is NULL, the `neigh_ops` object will be set to be `ndisc_direct_ops`. As in the case of `arp_direct_ops`, sending the packet is done with the `neigh_direct_output()` method, which is in fact a wrapper around `dev_queue_xmit()`. Note that, as mentioned in the ARP section earlier, in most Ethernet network devices, the `header_ops` of the `net_device` object is not NULL.
- If the `header_ops` of the `net_device` object contains a NULL `cache()` callback, then the `neigh_ops` object is set to be `ndisc_generic_ops`.
- If the `header_ops` of the `net_device` object contains a non-NULL `cache()` callback, then the `neigh_ops` object is set to be `ndisc_hh_ops`.

This section discussed the DAD mechanism and how it helps to avoid duplicate addresses. The next section describes how solicitation requests are sent.

NIDSC: Sending Solicitation Requests

Similarly to what you saw in IPv6, you also perform a lookup and create an entry if you did not find any match:

```
static int ip6_finish_output2(struct sk_buff *skb)
{
        struct dst_entry *dst = skb_dst(skb);
        struct net_device *dev = dst->dev;
        struct neighbour *neigh;
        struct in6_addr *nexthop;
        int ret;
               . . .

               . . .

        nexthop = rt6_nexthop((struct rt6_info *)dst, &ipv6_hdr(skb)->daddr);
        neigh = __ipv6_neigh_lookup_noref(dst->dev, nexthop);
        if (unlikely(!neigh))
                neigh = __neigh_create(&nd_tbl, nexthop, dst->dev, false);
        if (!IS_ERR(neigh)) {
                ret = dst_neigh_output(dst, neigh, skb);
               . . .
```

Eventually, much like in the IPv4 Tx path, you call the solicit method `neigh->ops->solicit(neigh, skb)` from the `neigh_probe()` method. The `neigh->ops->solicit` in this case is the `ndisc_solicit()` method. The `ndisc_solicit()` is a very short method; it is in fact a wrapper around the `ndisc_send_ns()` method:

```
static void ndisc_solicit(struct neighbour *neigh, struct sk_buff *skb)
{
        struct in6_addr *saddr = NULL;
        struct in6_addr mcaddr;
        struct net_device *dev = neigh->dev;
        struct in6_addr *target = (struct in6_addr *)&neigh->primary_key;
        int probes = atomic_read(&neigh->probes);

        if (skb && ipv6_chk_addr(dev_net(dev), &ipv6_hdr(skb)->saddr, dev, 1))
                saddr = &ipv6_hdr(skb)->saddr;

        if ((probes -= neigh->parms->ucast_probes) < 0) {
                if (!(neigh->nud_state & NUD_VALID)) {
                        ND_PRINTK(1, dbg,
                                  "%s: trying to ucast probe in NUD_INVALID: %pI6\n",
                                  __func__, target);
                }
                ndisc_send_ns(dev, neigh, target, target, saddr);
        } else if ((probes -= neigh->parms->app_probes) < 0) {
#ifdef CONFIG_ARPD
                neigh_app_ns(neigh);
#endif
        } else {
                addrconf_addr_solict_mult(target, &mcaddr);
                ndisc_send_ns(dev, NULL, target, &mcaddr, saddr);
        }
}
```

In order to send the solicitation request, we need to build an `nd_msg` object:

```
struct nd_msg {
        struct icmp6hdr icmph;
        struct in6_addr target;
        __u8            opt[0];
};
```

(include/net/ndisc.h)

For a solicitation request, the ICMPv6 header type should be set to NDISC_NEIGHBOUR_SOLICITATION, and for solicitation reply, the ICMPv6 header type should be set to NDISC_NEIGHBOUR_ADVERTISEMENT. Note that with Neighbour Advertisement messages, there are cases when you need to set flags in the ICMPv6 header. The ICMPv6 header includes a structure named `icmpv6_nd_advt`, which includes the override, solicited, and router flags:

```
struct icmp6hdr {
        __u8            icmp6_type;
        __u8            icmp6_code;
        __sum16         icmp6_cksum;
        union {
                . . .
                . . .
                struct icmpv6_nd_advt {
#if defined(__LITTLE_ENDIAN_BITFIELD)
                        __u32           reserved:5,
                                        override:1,
                                        solicited:1,
                                        router:1,
                                        reserved2:24;
. . .
#endif
                } u_nd_advt;
        } icmp6_dataun;
. . .
#define icmp6_router            icmp6_dataun.u_nd_advt.router
#define icmp6_solicited         icmp6_dataun.u_nd_advt.solicited
#define icmp6_override          icmp6_dataun.u_nd_advt.override
. . .
```

(include/uapi/linux/icmpv6.h)

- When a message is sent in response to a Neighbour Solicitation, you set the `solicited` flag (`icmp6_solicited`).
- When you want to override a neighbouring cache entry (update the L2 address), you set the `override` flag (`icmp6_override`).
- When the host sending the Neighbour Advertisement message is a router, you set the `router` flag (`icmp6_router`).

You can see the use of these three flags in the `ndisc_send_na()` method that follows. Let's take a look at the `ndisc_send_ns()` method:

```
void ndisc_send_ns(struct net_device *dev, struct neighbour *neigh,
                   const struct in6_addr *solicit,
                   const struct in6_addr *daddr, const struct in6_addr *saddr)
{
        struct sk_buff *skb;
        struct in6_addr addr_buf;
        int inc_opt = dev->addr_len;
        int optlen = 0;
        struct nd_msg *msg;

        if (saddr == NULL) {
                if (ipv6_get_lladdr(dev, &addr_buf,
                                   (IFA_F_TENTATIVE|IFA_F_OPTIMISTIC)))
                        return;
                saddr = &addr_buf;
        }

        if (ipv6_addr_any(saddr))
                inc_opt = 0;
        if (inc_opt)
                optlen += ndisc_opt_addr_space(dev);

        skb = ndisc_alloc_skb(dev, sizeof(*msg) + optlen);
        if (!skb)
                return;
```

Build the ICMPv6 header, which is embedded in the nd_msg object:

```
        msg = (struct nd_msg *)skb_put(skb, sizeof(*msg));
        *msg = (struct nd_msg) {
                .icmph = {
                        .icmp6_type = NDISC_NEIGHBOUR_SOLICITATION,
                },
                .target = *solicit,
        };

        if (inc_opt)
                ndisc_fill_addr_option(skb, ND_OPT_SOURCE_LL_ADDR,
                                       dev->dev_addr);

        ndisc_send_skb(skb, daddr, saddr);
}
```

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

```
static void ndisc_send_na(struct net_device *dev, struct neighbour *neigh,
                          const struct in6_addr *daddr,
                          const struct in6_addr *solicited_addr,
                          bool router, bool solicited, bool override, bool inc_opt)
{
        struct sk_buff *skb;
        struct in6_addr tmpaddr;
        struct inet6_ifaddr *ifp;
        const struct in6_addr *src_addr;
        struct nd_msg *msg;
        int optlen = 0;

        . . .

        skb = ndisc_alloc_skb(dev, sizeof(*msg) + optlen);
        if (!skb)
                return;
```

Build the ICMPv6 header, which is embedded in the nd_msg object:

```
        msg = (struct nd_msg *)skb_put(skb, sizeof(*msg));
        *msg = (struct nd_msg) {
                .icmph = {
                        .icmp6_type = NDISC_NEIGHBOUR_ADVERTISEMENT,
                        .icmp6_router = router,
                        .icmp6_solicited = solicited,
                        .icmp6_override = override,
                },
                .target = *solicited_addr,
        };
        if (inc_opt)
                ndisc_fill_addr_option(skb, ND_OPT_TARGET_LL_ADDR,
                                       dev->dev_addr);

        ndisc_send_skb(skb, daddr, src_addr);
}
```

This section described how solicitation requests are sent. The next section talks about how Neighbour Solicitations and Advertisements are handled.

NDISC: Receiving Neighbour Solicitations and Advertisements

As mentioned, the `ndisc_rcv()` method handles all five neighbour discovery message types; let's take a look at this method:

```
int ndisc_rcv(struct sk_buff *skb)
{
        struct nd_msg *msg;

        if (skb_linearize(skb))
                return 0;

        msg = (struct nd_msg *)skb_transport_header(skb);

        __skb_push(skb, skb->data - skb_transport_header(skb));
```

According to RFC 4861, the hop limit of neighbour messages should be 255; the hop limit length is 8 bits, so the maximum hop limit is 255. A value of 255 assures that the packet was not forwarded, and this assures you that you are not exposed to some security attack. Packets that do not fulfill this requirement are discarded:

```
        if (ipv6_hdr(skb)->hop_limit != 255) {
                ND_PRINTK(2, warn, "NDISC: invalid hop-limit: %d\n",
                          ipv6_hdr(skb)->hop_limit);
                return 0;
        }
```

According to RFC 4861, the ICMPv6 code of neighbour messages should be 0, so drop packets that do not fulfill this requirement:

```
        if (msg->icmph.icmp6_code != 0) {
                ND_PRINTK(2, warn, "NDISC: invalid ICMPv6 code: %d\n",
                          msg->icmph.icmp6_code);
        return 0;
        }

        memset(NEIGH_CB(skb), 0, sizeof(struct neighbour_cb));

        switch (msg->icmph.icmp6_type) {
        case NDISC_NEIGHBOUR_SOLICITATION:
                ndisc_recv_ns(skb);
                break;
        case NDISC_NEIGHBOUR_ADVERTISEMENT:
                ndisc_recv_na(skb);
                break;

        case NDISC_ROUTER_SOLICITATION:
                ndisc_recv_rs(skb);
                break;

        case NDISC_ROUTER_ADVERTISEMENT:
                ndisc_router_discovery(skb);
                break;

        case NDISC_REDIRECT:
                ndisc_redirect_rcv(skb);
                break;
        }

        return 0;
}
```

I do not discuss router solicitations and router advertisements in this chapter, since they are discussed in Chapter 8. Let's take a look at the `ndisc_recv_ns()` method:

```
static void ndisc_recv_ns(struct sk_buff *skb)
{
        struct nd_msg *msg = (struct nd_msg *)skb_transport_header(skb);
        const struct in6_addr *saddr = &ipv6_hdr(skb)->saddr;
        const struct in6_addr *daddr = &ipv6_hdr(skb)->daddr;
        u8 *lladdr = NULL;
        u32 ndoptlen = skb->tail - (skb->transport_header +
                            offsetof(struct nd_msg, opt));
        struct ndisc_options ndopts;
        struct net_device *dev = skb->dev;
        struct inet6_ifaddr *ifp;
        struct inet6_dev *idev = NULL;
        struct neighbour *neigh;
```

The `ipv6_addr_any()` method returns 1 when `saddr` is the unspecified address of all zeroes (IPV6_ADDR_ANY). When the source address is the unspecified address (all zeroes), this means that the request is DAD:

```
        int dad = ipv6_addr_any(saddr);
        bool inc;
        int is_router = -1;
```

Perform some validity checks:

```
        if (skb->len < sizeof(struct nd_msg)) {
                ND_PRINTK(2, warn, "NS: packet too short\n");
                return;
        }
```

```
if (ipv6_addr_is_multicast(&msg->target)) {
        ND_PRINTK(2, warn, "NS: multicast target address\n");
        return;
}

/*
 * RFC2461 7.1.1:
 * DAD has to be destined for solicited node multicast address.
 */
if (dad && !ipv6_addr_is_solict_mult(daddr)) {
        ND_PRINTK(2, warn, "NS: bad DAD packet (wrong destination)\n");
        return;
}

if (!ndisc_parse_options(msg->opt, ndoptlen, &ndopts)) {
        ND_PRINTK(2, warn, "NS: invalid ND options\n");
        return;
}

if (ndopts.nd_opts_src_lladdr) {
        lladdr = ndisc_opt_addr_data(ndopts.nd_opts_src_lladdr, dev);
        if (!lladdr) {
                ND_PRINTK(2, warn,
                          "NS: invalid link-layer address length\n");
                return;
        }

        /* RFC2461 7.1.1:
         *      If the IP source address is the unspecified address,
         *      there MUST NOT be source link-layer address option
         *      in the message.
         */
        if (dad) {
                ND_PRINTK(2, warn,
                          "NS: bad DAD packet (link-layer address option)\n");
                return;
        }
}

inc = ipv6_addr_is_multicast(daddr);

ifp = ipv6_get_ifaddr(dev_net(dev), &msg->target, dev, 1);
if (ifp) {

        if (ifp->flags & (IFA_F_TENTATIVE|IFA_F_OPTIMISTIC)) {
                if (dad) {
                        /*
                         * We are colliding with another node
                         * who is doing DAD
                         * so fail our DAD process
                        */
```

```
                                addrconf_dad_failure(ifp);
                                return;
                        } else {
                                /*
                                 * This is not a dad solicitation.
                                 * If we are an optimistic node,
                                 * we should respond.
                                 * Otherwise, we should ignore it.
                                */
                                if (!(ifp->flags & IFA_F_OPTIMISTIC))
                                        goto out;
                        }
                }

                idev = ifp->idev;
        } else {
                struct net *net = dev_net(dev);

                idev = in6_dev_get(dev);
                if (!idev) {
                        /* XXX: count this drop? */
                        return;
                }

                if (ipv6_chk_acast_addr(net, dev, &msg->target) ||
                    (idev->cnf.forwarding &&
                     (net->ipv6.devconf_all->proxy_ndp || idev->cnf.proxy_ndp) &&
                     (is_router = pndisc_is_router(&msg->target, dev)) >= 0)) {
                        if (!(NEIGH_CB(skb)->flags & LOCALLY_ENQUEUED) &&
                            skb->pkt_type != PACKET_HOST &&
                            inc != 0 &&
                            idev->nd_parms->proxy_delay != 0) {
                                /*
                                 * for anycast or proxy,
                                 * sender should delay its response
                                 * by a random time between 0 and
                                 * MAX_ANYCAST_DELAY_TIME seconds.
                                 * (RFC2461) -- yoshfuji
                                */
                                struct sk_buff *n = skb_clone(skb, GFP_ATOMIC);
                                if (n)
                                        pneigh_enqueue(&nd_tbl, idev->nd_parms, n);
                                goto out;
                        }
                } else
                        goto out;
        }
```

```
if (is_router < 0)
        is_router = idev->cnf.forwarding;

if (dad) {
```

Send a neighbour advertisement message:

```
        ndisc_send_na(dev, NULL, &in6addr_linklocal_allnodes, &msg->target,
                      !!is_router, false, (ifp != NULL), true);
        goto out;
}

if (inc)
        NEIGH_CACHE_STAT_INC(&nd_tbl, rcv_probes_mcast);
else
        NEIGH_CACHE_STAT_INC(&nd_tbl, rcv_probes_ucast);

/*
 *      update / create cache entry
 *      for the source address
*/
neigh = __neigh_lookup(&nd_tbl, saddr, dev,
                       !inc || lladdr || !dev->addr_len);
if (neigh)
```

Update your neighbouring table with the sender's L2 address; the `nud_state` will be set to be NUD_STALE:

```
        neigh_update(neigh, lladdr, NUD_STALE,
                     NEIGH_UPDATE_F_WEAK_OVERRIDE|
                     NEIGH_UPDATE_F_OVERRIDE);
if (neigh || !dev->header_ops) {
```

Send a Neighbour Advertisement message:

```
                ndisc_send_na(dev, neigh, saddr, &msg->target,
                              !!is_router,
                              true, (ifp != NULL && inc), inc);
                if (neigh)
                        neigh_release(neigh);
        }

out:
        if (ifp)
                in6_ifa_put(ifp);
        else
                in6_dev_put(idev);
}
```

Let's take a look at the method that handles Neighbour Advertisements, ndisc_recv_na():

```
static void ndisc_recv_na(struct sk_buff *skb)
{
        struct nd_msg *msg = (struct nd_msg *)skb_transport_header(skb);
        const struct in6_addr *saddr = &ipv6_hdr(skb)->saddr;
        const struct in6_addr *daddr = &ipv6_hdr(skb)->daddr;
        u8 *lladdr = NULL;
        u32 ndoptlen = skb->tail - (skb->transport_header +
                                    offsetof(struct nd_msg, opt));
        struct ndisc_options ndopts;
        struct net_device *dev = skb->dev;
        struct inet6_ifaddr *ifp;
        struct neighbour *neigh;

        if (skb->len < sizeof(struct nd_msg)) {
                ND_PRINTK(2, warn, "NA: packet too short\n");
                return;
        }

        if (ipv6_addr_is_multicast(&msg->target)) {
                ND_PRINTK(2, warn, "NA: target address is multicast\n");
                return;
        }

        if (ipv6_addr_is_multicast(daddr) &&
            msg->icmph.icmp6_solicited) {
                ND_PRINTK(2, warn, "NA: solicited NA is multicasted\n");
                return;
        }

        if (!ndisc_parse_options(msg->opt, ndoptlen, &ndopts)) {
                ND_PRINTK(2, warn, "NS: invalid ND option\n");
                return;
        }
        if (ndopts.nd_opts_tgt_lladdr) {
                lladdr = ndisc_opt_addr_data(ndopts.nd_opts_tgt_lladdr, dev);
                if (!lladdr) {
                        ND_PRINTK(2, warn,
                                  "NA: invalid link-layer address length\n");
                        return;
                }
        }
        ifp = ipv6_get_ifaddr(dev_net(dev), &msg->target, dev, 1);
        if (ifp) {
                if (skb->pkt_type != PACKET_LOOPBACK
                    && (ifp->flags & IFA_F_TENTATIVE)) {
                                addrconf_dad_failure(ifp);
                                return;
                }
                /* What should we make now? The advertisement
                   is invalid, but ndisc specs say nothing
                   about it. It could be misconfiguration, or
                   an smart proxy agent tries to help us :-)

                   We should not print the error if NA has been
                   received from loopback - it is just our own
                   unsolicited advertisement.
                 */
                if (skb->pkt_type != PACKET_LOOPBACK)
                        ND_PRINTK(1, warn,
                                  "NA: someone advertises our address %pI6 on %s!\n",
                                  &ifp->addr, ifp->idev->dev->name);
                in6_ifa_put(ifp);
                return;
        }
        neigh = neigh_lookup(&nd_tbl, &msg->target, dev);

        if (neigh) {
                u8 old_flags = neigh->flags;
                struct net *net = dev_net(dev);

                if (neigh->nud_state & NUD_FAILED)
                        goto out;

                /*
                 * Don't update the neighbour cache entry on a proxy NA from
                 * ourselves because either the proxied node is off link or it
                 * has already sent a NA to us.
                 */
                if (lladdr && !memcmp(lladdr, dev->dev_addr, dev->addr_len) &&
                    net->ipv6.devconf_all->forwarding &&
                    net->ipv6.devconf_all->proxy_ndp &&
                    pneigh_lookup(&nd_tbl, net, &msg->target, dev, 0)) {
                        /* XXX: idev->cnf.proxy_ndp */
                        goto out;
                }
```

Update the neighbouring table. When the received message is a Neighbour Solicitation, the `icmp6_solicited` is set, so you want to set the state to be NUD_REACHABLE. When the `icmp6_override` flag is set, you want the override flag to be set (this mean update the L2 address with the specified `lladdr`, if it is different):

```
                neigh_update(neigh, lladdr,
                             msg->icmph.icmp6_solicited ? NUD_REACHABLE : NUD_STALE,
                             NEIGH_UPDATE_F_WEAK_OVERRIDE|
                             (msg->icmph.icmp6_override ? NEIGH_UPDATE_F_OVERRIDE : 0)|
                             NEIGH_UPDATE_F_OVERRIDE_ISROUTER|
                             (msg->icmph.icmp6_router ? NEIGH_UPDATE_F_ISROUTER : 0));

                if ((old_flags & ~neigh->flags) & NTF_ROUTER) {
                        /*
                         * Change: router to host
                         */
                        struct rt6_info *rt;
                        rt = rt6_get_dflt_router(saddr, dev);
                        if (rt)
                                ip6_del_rt(rt);
                }

out:
                neigh_release(neigh);
        }
}
```

Summary

This chapter described the neighbouring subsystem in IPv4 and in IPv6. First you learned about the goals of the neighbouring subsystem. Then you learned about ARP requests and ARP replies in IPv4, and about NDISC Neighbour Solicitation and NDISC Neighbour Advertisements in IPv6. You also found out about how DAD implementation avoids duplicate IPv6 addresses, and you saw various methods for handling the neighbouring subsystem requests and replies. Chapter 8 discusses the IPv6 subsystem implementation. The “Quick Reference” section that follows covers the top methods and macros related to the topics discussed in this chapter, ordered by their context. I also show the `neigh_statistics` structure, which represents statistics collected by the neighbouring subsystem.

Quick Reference

The following are some important methods and macros of the neighbouring subsystem, and a description of the `neigh_statistics` structure.

■ **Note** The core neighbouring code is in `net/core/neighbour.c`, `include/net/neighbour.h` and `include/uapi/linux/neighbour.h`.

The ARP code (IPv4) is in `net/ipv4/arp.c`, `include/net/arp.h` and in `include/uapi/linux/if_arp.h`.

The NDISC code (IPv6) is in `net/ipv6/ndisc.c` and `include/net/ndisc.h`.

Methods

Let’s start by covering the methods.

void neigh_table_init(struct neigh_table *tbl)

This method invokes the `neigh_table_init_no_netlink()` method to perform the initialization of the neighbouring table, and links the table to the global neighbouring tables linked list (`neigh_tables`).

void neigh_table_init_no_netlink(struct neigh_table *tbl)

This method performs all the neighbour initialization apart from linking it to the global neighbouring table linked list, which is done by the `neigh_table_init()`, as mentioned earlier.

int neigh_table_clear(struct neigh_table *tbl)

This method frees the resources of the specified neighbouring table.

struct neighbour *neigh_alloc(struct neigh_table *tbl, struct net_device *dev)

This method allocates a neighbour object.

struct neigh_hash_table *neigh_hash_alloc(unsigned int shift)

This method allocates a neighbouring hash table.

struct neighbour *__neigh_create(struct neigh_table *tbl, const void *pkey, struct net_device *dev, bool want_ref)

This method creates a neighbour object.

int neigh_add(struct sk_buff *skb, struct nlmsghdr *nlh, void *arg)

This method adds a neighbour entry; it is the handler for netlink RTM_NEWNEIGH message.

int neigh_delete(struct sk_buff *skb, struct nlmsghdr *nlh, void *arg)

This method deletes a neighbour entry; it is the handler for netlink RTM_DELNEIGH message.

void neigh_probe(struct neighbour *neigh)

This method fetches an SKB from the neighbour `arp_queue` and calls the corresponding `solicit()` method to send it. In case of ARP, it will be `arp_solicit()`. It increments the neighbour `probes` counter and frees the packet.

int neigh_forced_gc(struct neigh_table *tbl)

This method is a synchronous garbage collection method. It removes neighbour entries that are not in the permanent state (NUD_PERMANENT) and whose reference count equals 1. The removal and cleanup of a neighbour is done by first setting the dead flag of the neighbour to be 1 and then calling the `neigh_cleanup_and_release()` method, which gets a neighbour object as a parameter. The `neigh_forced_gc()` method is invoked from the `neigh_alloc()` method under some conditions, as described in the "Creating and Freeing a Neighbour" section earlier in this chapter. The `neigh_forced_gc()` method returns 1 if at least one neighbour object was removed, and 0 otherwise.

void neigh_periodic_work(struct work_struct *work)

This method is the asynchronous garbage collector handler.

static void neigh_timer_handler(unsigned long arg)

This method is the per-neighbour periodic timer garbage collector handler.

struct neighbour *__neigh_lookup(struct neigh_table *tbl, const void *pkey, struct net_device *dev, int creat)

This method performs a lookup in the specified neighbouring table by the given key. If the `creat` parameter is 1, and the lookup fails, call the `neigh_create()` method to create a neighbour entry in the specified neighbouring table and return it.

neigh_hh_init(struct neighbour *n, struct dst_entry *dst)

This method initializes the L2 cache (`hh_cache` object) of the specified neighbour based on the specified routing cache entry.

void __init arp_init(void)

This method performs the setup for the ARP protocol: initialize the ARP table, register the `arp_rcv()` as a handler for receiving ARP packets, initialize `procfs` entries, register `sysctl` entries, and register the ARP `netdev` notifier callback, `arp_netdev_event()`.

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

This method is the Rx handler for ARP packets (Ethernet packets with type 0x0806).

int arp_constructor(struct neighbour *neigh)

This method performs ARP neighbour initialization.

int arp_process(struct sk_buff *skb)

This method, invoked by the `arp_rcv()` method, handles the main processing of ARP requests and ARP responses.

void arp_solicit(struct neighbour *neigh, struct sk_buff *skb)

This method sends the solicitation request (ARPOP_REQUEST) after some checks and initializations, by calling the `arp_send()` method.

void arp_send(int type, int ptype, __be32 dest_ip, struct net_device *dev, __be32 src_ip, const unsigned char *dest_hw, const unsigned char *src_hw, const unsigned char *target_hw)

This method creates an ARP packet and initializes it with the specified parameters, by calling the `arp_create()` method, and sends it by calling the `arp_xmit()` method.

void arp_xmit(struct sk_buff *skb)

This method actually sends the packet by calling the NF_HOOK macro with `dev_queue_xmit()`.

struct arphdr *arp_hdr(const struct sk_buff *skb)

This method fetches the ARP header of the specified SKB.

int arp_mc_map(__be32 addr, u8 *haddr, struct net_device *dev, int dir)

This method translates an IPv4 address to L2 (link layer) address according to the network device type. When the device is an Ethernet device, for example, this is done with the `ip_eth_mc_map()` method; when the device is an Infiniband device, this is done with the `ip_ib_mc_map()` method.

static inline int arp_fwd_proxy(struct in_device *in_dev, struct net_device *dev, struct rtable *rt)

This method returns 1 if the specified device can use proxy ARP for the specified routing entry.

static inline int arp_fwd_pvlan(struct in_device *in_dev, struct net_device *dev, struct rtable *rt, __be32 sip, __be32 tip)

This method returns 1 if the specified device can use proxy ARP VLAN for the specified routing entry and specified IPv4 source and destination addresses.

int arp_netdev_event(struct notifier_block *this, unsigned long event, void *ptr)

This method is the ARP handler for `netdev` notification events.

int ndisc_netdev_event(struct notifier_block *this, unsigned long event, void *ptr)

This method is the NDISC handler for `netdev` notification events.

int ndisc_rcv(struct sk_buff *skb)

This method is the main NDISC handler for receiving one of the five types of solicitation packets.

static int neigh_blackhole(struct neighbour *neigh, struct sk_buff *skb)

This method discards the packet and returns –ENETDOWN error (network is down).

static void ndisc_recv_ns(struct sk_buff *skb) and static void ndisc_recv_na(struct sk_buff *skb)

These methods handle receiving Neighbour Solicitation and Neighbour Advertisement, respectively.

static void ndisc_recv_rs(struct sk_buff *skb) and static void ndisc_router_discovery(struct sk_buff *skb)

These methods handle receiving router solicitation and router advertisement, respectively.

int ndisc_mc_map(const struct in6_addr *addr, char *buf, struct net_device *dev, int dir)

This method translates an IPv4 address to a L2 (link layer) address according to the network device type. In Ethernet under IPv6, this is done by the `ipv6_eth_mc_map()` method.

int ndisc_constructor(struct neighbour *neigh)

This method performs NDISC neighbour initialization.

void ndisc_solicit(struct neighbour *neigh, struct sk_buff *skb)

This method sends the solicitation request after some checks and initializations, by calling the `ndisc_send_ns()` method.

int icmpv6_rcv(struct sk_buff *skb)

This method is a handler for receiving ICMPv6 messages.

bool ipv6_addr_any(const struct in6_addr *a)

This method returns 1 when the given IPv6 address is the unspecified address of all zeroes (IPV6_ADDR_ANY).

int inet_addr_onlink(struct in_device *in_dev, __be32 a, __be32 b)

This method checks whether the two specified addresses are on the same subnet.

Macros

Now, let's look at the macros.

IN_DEV_PROXY_ARP(in_dev)

This macro returns `true` if `/proc/sys/net/ipv4/conf/<netDevice>/proxy_arp` is set or if `/proc/sys/net/ipv4/conf/all/proxy_arp` is set, where `netDevice` is the network device associated with the specified `in_dev`.

IN_DEV_PROXY_ARP_PVLAN(in_dev)

This macro returns `true` if `/proc/sys/net/ipv4/conf/<netDevice>/proxy_arp_pvlan` is set, where `netDevice` is the network device associated with the specified `in_dev`.

IN_DEV_ARPFILTER(in_dev)

This macro returns `true` if `/proc/sys/net/ipv4/conf/<netDevice>/arp_filter` is set or if `/proc/sys/net/ipv4/conf/all/arp_filter` is set, where `netDevice` is the network device associated with the specified `in_dev`.

IN_DEV_ARP_ACCEPT(in_dev)

This macro returns `true` if `/proc/sys/net/ipv4/conf/<netDevice>/arp_accept` is set or if `/proc/sys/net/ipv4/conf/all/arp_accept` is set, where `netDevice` is the network device associated with the specified `in_dev`.

IN_DEV_ARP_ANNOUNCE(in_dev)

This macro returns the max value of `/proc/sys/net/ipv4/conf/<netDevice>/arp_announce` and `/proc/sys/net/ipv4/conf/all/arp_announce`, where `netDevice` is the network device associated with the specified `in_dev`.

IN_DEV_ARP_IGNORE(in_dev)

This macro returns the max value of `/proc/sys/net/ipv4/conf/<netDevice>/arp_ignore` and `/proc/sys/net/ipv4/conf/all/arp_ignore`, where `netDevice` is the network device associated with the specified `in_dev`.

IN_DEV_ARP_NOTIFY(in_dev)

This macro returns the max value of `/proc/sys/net/ipv4/conf/<netDevice>/arp_notify` and `/proc/sys/net/ipv4/conf/all/arp_notify`, where `netDevice` is the network device associated with the specified `in_dev`.

IN_DEV_SHARED_MEDIA(in_dev)

This macro returns `true` if `/proc/sys/net/ipv4/conf/<netDevice>/shared_media` is set or if `/proc/sys/net/ipv4/conf/all/shared_media` is set, where `netDevice` is the network device associated with the specified `in_dev`.

IN_DEV_ROUTE_LOCALNET(in_dev)

This macro returns `true` if `/proc/sys/net/ipv4/conf/<netDevice>/route_localnet` is set or if `/proc/sys/net/ipv4/conf/all/route_localnet` is set, where `netDevice` is the network device associated with the specified `in_dev`.

neigh_hold()

This macro increments the reference count of the specified neighbour.

The neigh_statistics Structure

The `neigh_statistics` structure is important for monitoring the neighbouring subsystem; as mentioned in the beginning of the chapter, both ARP and NDISC export this structure members via `procfs` (`/proc/net/stat/arp_cache` and `/proc/net/stat/ndisc_cache`, respectively). Following is a description of its members and pointing out where they are incremented:

```
struct neigh_statistics {
        unsigned long allocs;           /* number of allocated neighs     */
        unsigned long destroys;         /* number of destroyed neighs     */
        unsigned long hash_grows;       /* number of hash resizes         */
        unsigned long res_failed;       /* number of failed resolutions   */
        unsigned long lookups;          /* number of lookups              */
        unsigned long hits;             /* number of hits (among lookups) */
        unsigned long rcv_probes_mcast; /* number of received mcast ipv6  */
        unsigned long rcv_probes_ucast; /* number of received ucast ipv6  */
        unsigned long periodic_gc_runs; /* number of periodic GC runs     */
        unsigned long forced_gc_runs;   /* number of forced GC runs       */
        unsigned long unres_discards;   /* number of unresolved drops     */
};
```

Here is a description of the members of the `neigh_statistics` structure:

- `allocs`: The number of the allocated neighbours; incremented by the `neigh_alloc()` method.
- `destroys`: The number of the destroyed neighbours; incremented by the `neigh_destroy()` method.
- `hash_grows`: The number of times that hash resize was done; incremented by the `neigh_hash_grow()` method.
- `res_failed`: The number of failed resolutions; incremented by the `neigh_invalidate()` method.
- `lookups`: The number of neighbour lookups that were done; incremented by the `neigh_lookup()` method and by the `neigh_lookup_nodev()` method.
- `hits`: The number of hits when performing a neighbour lookup ; incremented by the `neigh_lookup()` method and by the `neigh_lookup_nodev()` method, when you have a hit.
- `rcv_probes_mcast`: The number of received multicast probes (IPv6 only); incremented by the `ndisc_recv_ns()` method.
- `rcv_probes_ucast`: The number of received unicast probes (IPv6 only); incremented by the `ndisc_recv_ns()` method.
- `periodic_gc_runs`: The number of periodic GC invocations; incremented by the `neigh_periodic_work()` method.

- `forced_gc_runs`: The number of forced GC invocations; incremented by the `neigh_forced_gc()` method.
- `unres_discards`: The number of unresolved drops; incremented by the `__neigh_event_send()` method when an unresolved packet is discarded.

Table

Here is the table that was covered.

Table 7-1. *Network Unreachability Detection States*

Linux	Symbol
NUD_INCOMPLETE	Address resolution is in progress and the link-layer address of the neighbour has not yet been determined. This means that a solicitation request was sent, and you are waiting for a solicitation reply or a timeout.
NUD_REACHABLE	The neighbour is known to have been reachable recently.
NUD_STALE	More than ReachableTime milliseconds have elapsed since the last positive confirmation that the forward path was functioning properly was received.
NUD_DELAY	The neighbour is no longer known to be reachable. Delay sending probes for a short while in order to give upper layer protocols a chance to provide reachability confirmation.
NUD_PROBE	The neighbour is no longer known to be reachable, and unicast Neighbour Solicitation probes are being sent to verify reachability.
NUD_FAILED	Set the neighbour to be unreachable. When you delete a neighbour, you set it to be in the NUD_FAILED state.

CHAPTER 8

IPv6

In Chapter 7, I dealt with the Linux Neighbouring Subsystem and its implementation. In this chapter, I will discuss the IPv6 protocol and its implementation in Linux. IPv6 is the next-generation network layer protocol of the TCP/IP protocol stack. It was developed by the Internet Engineering Task Force (IETF), and it is intended to replace IPv4, which still carries the vast majority of Internet traffic.

In the early '90s, the IETF started an effort to develop the next generation of the IP protocol, due to the anticipated Internet growth. The first IPv6 RFC is from 1995: RFC 1883, "Internet Protocol, Version 6 (IPv6) Specification." Later, in 1998, RFC 2460 replaced it. The main problem IPv6 solves is the shortage of addresses: the length of an IPv6 address is 128 bits. IPv6 sets a much larger address space. Instead of 2^32 addresses in IPv4, we have 2^128 addresses in IPv6. This indeed enlarges the address space significantly, probably far more than will be needed in the next few decades. But extended address space is not the only advantage of IPv6, as some might think. Based on the experience gained with IPv4, many changes were made in IPv6 to improve the IP protocol. We will discuss many of these changes in this chapter.

The IPv6 protocol is now gaining momentum as an improved network layer protocol. The growing popularity of the Internet all over the globe, and the growing markets for smart mobile devices and tablets, surely make the exhaustion of IPv4 addresses a more evident problem. This gives rise to the need for transitioning to the IPv4 successor, the IPv6 protocol.

IPv6 – Short Introduction

The IPv6 subsystem is undoubtedly a very broad subject, which is growing steadily. Exciting features were added during the last decade. Some of these new features are based on IPv4, like ICMPv6 sockets, IPv6 Multicast Routing, and IPv6 NAT. IPsec is mandatory in IPv6 and optional in IPv4, though most operating systems implemented IPsec also in IPv4. When we delve into the IPv6 kernel internals, we find many similarities. Sometime the names of the methods and even the names of some of the variables are similar, except for the addition of "v6" or "6." There are, however, some changes in the implementation in some places.

We chose to discuss in this chapter the important new features of IPv6, show some places where it differs from IPv4, and explain why a change was made. The extension headers, the Multicast Listener Discovery (MLD) protocol, and the Autoconfiguration process are some of the new features that we discuss and demonstrate with some userspace examples. We also discuss how receiving IPv6 packets works, how IPv6 forwarding works, and some points of difference when comparing them to IPv4. On the whole, it seems that the developers of IPv6 made a lot of improvements based on the past experience with IPv4, and the IPv6 implementation brings a lot of benefits not found in IPv4 and a lot of advantages over IPv4. We will discuss IPv6 addresses in the following section, including multicast addresses and special addresses.

IPv6 Addresses

The first step in learning IPv6 is to become familiar with the IPv6 Addressing Architecture, which is defined in RFC 4291. There are three types of IPv6 addresses:

- **Unicast:** This address uniquely identifies an interface. A packet sent to a unicast address is delivered to the interface identified by that address.
- **Anycast:** This address can be assigned for a set of interfaces (usually on different nodes). This type of address does not exist in IPv4. It is, in fact, a mixture of a unicast address and a multicast address. A packet sent to an anycast address is delivered to one of the interfaces identified by that address (the "nearest" one, according to the routing protocols).
- **Multicast:** This address can be assigned for a set of interfaces (usually on different nodes). A packet sent to a multicast address is delivered to all the interfaces identified by that address. An interface can belong to any number of multicast groups.

There is no broadcast address in IPv6. In IPv6, to get the same result as broadcast, you can send a packet to the group multicast address of all nodes (`ff02::1`). In IPv4, a large part of the functionality of the Address Resolution Protocol (ARP) protocol is based on broadcasts. The IPv6 subsystem uses neighbour discovery instead of ARP to map L3 addresses to L2 addresses. The IPv6 neighbour discovery protocol is based on ICMPv6, and it uses multicast addresses instead of broadcasts, as you saw in the previous chapter. You will see more examples of using multicast traffic later in this chapter.

An IPv6 address comprises of 8 blocks of 16 bits, which is 128 bits in total. An IPv6 address looks like this: `xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx` (where x is a hexadecimal digit.) Sometimes you will encounter "`::`" inside an IPv6 address; this is a shortcut for leading zeroes.

In IPv6, address prefixes are used. Prefixes are, in fact, the parallel of IPv4 subnet masks. IPv6 prefixes are described in RFC 4291, "IP Version 6 Addressing Architecture." An IPv6 address prefix is represented by the following notation: `ipv6-address/prefix-length`.

The prefix-length is a decimal value specifying how many of the leftmost contiguous bits of the address comprise the prefix. We use "`/n`" to denote a prefix *n* bits long. For example, for all IPv6 addresses that begin with the 32 bits `2001:0da7`, the following prefix is used: `2001:da7::/32`.

Now that you have learned about the types of IPv6 addresses, you will learn in the following section about some special IPv6 addresses and their usage.

Special Addresses

In this section, I describe some special IPv6 addresses and their usage. It is recommended that you be familiar with these special addresses because you will encounter some of them later in this chapter (like the unspecified address of all zeroes that is used in DAD, or Duplicate Address Detection) and while browsing the code. The following list contains special IPv6 addresses and explanations about their usage:

- There should be at least one **link-local** unicast address on each interface. The link-local address allows communication with other nodes in the same physical network; it is required for neighbour discovery, automatic address configuration, and more. Routers must not forward any packets with link-local source or destination addresses. Link-local addresses are assigned with the prefix `fe80::/64`.
- The Global Unicast Address general format is as follows: the first *n* bits are the `global routing prefix`, the next *m* bits are the `subnet ID`, and the rest of the 128-*n*-*m* bits are the `interface ID`.

- `global routing prefix`: A value assigned to a site. It represents the network ID or prefix of the address.
- `subnet ID`: An identifier of a subnet within the site.
- `interface ID`: An `id`; its value must be unique within the subnet. This is defined in RFC 3513, section 2.5.1.

The Global Unicast Address is described in RFC 3587, "IPv6 Global Unicast Address Format." The assignable Global Unicast Address space is defined in RFC 4291.

- The IPv6 loopback address is `0:0:0:0:0:0:0:1`, or `::1` in short notation.
- The address of all zeroes (`0:0:0:0:0:0:0:0`) is called the **unspecified address**. It is used in DAD (Duplicate Address Detection) as you saw in the previous chapter. It should not be used as a destination address. You cannot assign the unspecified address to an interface by using userspace tools like the `ip` command or the `ifconfig` command.
- **IPv4-mapped IPv6 addresses** are addresses that start with 80 bits of zero. The next 16 bits are one, and the remaining 32 bits are the IPv4 address. For example, `::ffff:192.0.2.128` represents the IPv4 address of 192.0.2.128. For usage of these addresses, see RFC 4038, "Application Aspects of IPv6 Transition."
- **The IPv4-compatible** format is deprecated; in this format, the IPv4 address is in the lower 32 bits of the IPv6 address and all remaining bits are 0; the address mentioned earlier should be `::192.0.2.128` in this format. See RFC 4291, section 2.5.5.1.
- **Site local addresses** were originally designed to be used for addressing inside of a site without the need for a global prefix, but they were deprecated in RFC 3879, "Deprecating Site Local Addresses," in 2004.

An IPv6 address is represented in Linux by the `in6_addr` structure; using a union with three arrays (with 8, 16, and 32 bit elements) in the `in6_addr` structure helps in bit-manipulation operations:

```
struct in6_addr {
        union {
                __u8            u6_addr8[16];
                __be16          u6_addr16[8];
                __be32          u6_addr32[4];
        } in6_u;
#define s6_addr                 in6_u.u6_addr8
#define s6_addr16               in6_u.u6_addr16
#define s6_addr32               in6_u.u6_addr32
};
```

(`include/uapi/linux/in6.h`)

Multicast plays an important role in IPv6, especially for ICMPv6-based protocols like NDISC (which I discussed in Chapter 7, which dealt with the Linux Neighbouring Subsystem) and MLD (which is discussed later in this chapter). I will now discuss multicast addresses in IPv6 in the next section.

Multicast Addresses

Multicast addresses provide a way to define a multicast group; a node can belong to one or more multicast groups. Packets whose destination is a multicast address should be delivered to every node that belongs to that multicast group. In IPv6, all multicast addresses start with FF (8 first bits). Following are 4 bits for flags and 4 bits for scope. Finally, the last 112 bits are the group ID. The 4 bits of the flags field have this meaning:

- **Bit 0:** Reserved for future use.
- **Bit 1:** A value of 1 indicates that a Rendezvous Point is embedded in the address. Discussion of Rendezvous Points is more related to userspace daemons and is not within the scope of this book. For more details, see RFC 3956, "Embedding the Rendezvous Point (RP) Address in an IPv6 Multicast Address." This bit is sometimes referred to as the R-flag (R for Rendezvous Point.)
- **Bit 2:** A value of 1 indicates a multicast address that is assigned based on the network prefix. (See RFC 3306.) This bit is sometimes referred to as the P-flag (P for Prefix information.)
- **Bit 3:** A value of 0 indicates a permanently-assigned ("well-known") multicast address, assigned by the Internet Assigned Numbers Authority (IANA). A value of 1 indicates a non-permanently-assigned ("transient") multicast address. This bit is sometimes referred to as the T-flag (T for Temporary.)

The scope can be one of the entries in Table 8-1, which shows the various IPv6 scopes by their Linux symbol and by their value.

Table 8-1. *IPv6 scopes*

Hex value	Description	Linux Symbol
0x01	node local	IPV6_ADDR_SCOPE_NODELOCAL
0x02	link local	IPV6_ADDR_SCOPE_LINKLOCAL
0x05	site local	IPV6_ADDR_SCOPE_SITELOCAL
0x08	organization	IPV6_ADDR_SCOPE_ORGLOCAL
0x0e	global	IPV6_ADDR_SCOPE_GLOBAL

Now that you've learned about IPv6 multicast addresses, you will learn about some special multicast addresses in the next section.

Special Multicast Addresses

There are some special multicast addresses that I will mention in this chapter. Section 2.7.1 of RFC 4291 defines these special multicast addresses:

- All Nodes Multicast Address group: `ff01::1`, `ff02::1`
- All Routers Multicast Address group: `ff01::2`, `ff02::2`, `ff05::2`

According to RFC 3810, there is this special address: All MLDv2-capable routers Multicast Group, which is `ff02::16`. Version 2 Multicast Listener Reports will be sent to this special address; I will discuss it in the "Multicast Listener Discovery (MLD)" section later in this chapter.

A node is required to compute and join (on the appropriate interface) the associated Solicited-Node multicast addresses for all unicast and anycast addresses that have been configured for the node's interfaces (manually or automatically). Solicited-Node multicast addresses are computed based on the node's unicast and anycast addresses. A Solicited-Node multicast address is formed by taking the low-order 24 bits of an address (unicast or anycast) and appending those bits to the prefix `ff02:0:0:0:0:1:ff00::/104`, resulting in a multicast address in the range `ff02:0:0:0:0:1:ff00:0000` to `ff02:0:0:0:0:1:ffff:ffff`. See RFC 4291.

The method `addrconf_addr_solict_mult()` computes a link-local, solicited-node multicast address (`include/net/addrconf.h`). The method `addrconf_join_solict()` joins to a solicited address multicast group (`net/ipv6/addrconf.c`).

In the previous chapter, you saw that a neighbour advertisement message is sent by the `ndisc_send_na()` method to the link-local, all nodes address (`ff02::1`). You will see more examples of using special addresses like the all nodes multicast group address or all routers multicast group address in later subsections of this chapter. In this section, you have seen some multicast addresses, which you will encounter later in this chapter and while browsing the IPv6 source code. I will now discuss the IPv6 header in the following section.

IPv6 Header

Each IPv6 packet starts with an IPv6 header, and it is important to learn about its structure to understand fully the IPv6 Linux implementation. The IPv6 header has a fixed length of 40 bytes; for this reason, there is no field specifying the IPv6 header length (as opposed to IPv4, where the `ihl` member of the IPv4 header represents the header length). Note that there is also no checksum field in the IPv6 header, and this will be explained later in this chapter. In IPv6, there is no IP options mechanism as in IPv4. The IP options processing mechanism in IPv4 has a performance cost. Instead, IPV6 has a much more efficient mechanism of extension headers, which will be discussed in the next section, "extension headers." Figure 8-1 shows the IPv6 header and its fields.

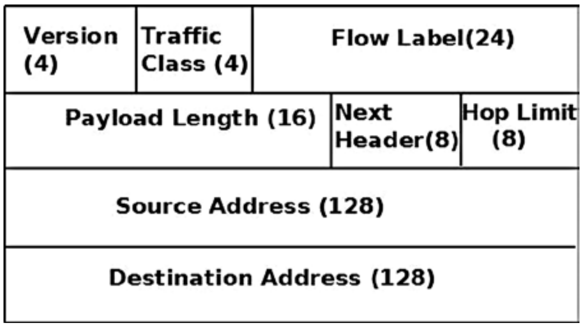


Figure 8-1. *IPv6 header*

Note that in the original IPv6 standard, RFC 2460, the priority (Traffic Class) is 8 bits and the flow label is 20 bits. In the definition of the `ipv6hdr` structure, the `priority` (Traffic Class) field size is 4 bits. In fact, in the Linux IPv6 implementation, the first 4 bits of `flow_lbl` are glued to the `priority` (Traffic Class) field in order to form a "class." Figure 8-1 reflects the Linux definition of the `ipv6hdr` structure, which is shown here:

```
struct ipv6hdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
        __u8                    priority:4,
                                version:4;
#elif defined(__BIG_ENDIAN_BITFIELD)
        __u8                    version:4,
                                priority:4;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
        __u8                    flow_lbl[3];

        __be16                  payload_len;
        __u8                    nexthdr;
        __u8                    hop_limit;

        struct  in6_addr        saddr;
        struct  in6_addr        daddr;
};
```

(include/uapi/linux/ipv6.h)

The following is a description of the members of the ipv6hdr structure:

- version: A 4-bit field. It should be set to 6.
- priority: Indicates the traffic class or priority of the IPv6 packet. RFC 2460, the base of IPv6, does not define specific traffic class or priority values.
- flow_lbl: The flow labeling field was regarded as experimental when the base IPv6 standard was written (RFC 2460). It provides a way to label sequences of packets of a particular flow; this labeling can be used by upper layers for various purposes. RFC 6437, "IPv6 Flow Label Specification," from 2011, suggests using flow labeling to detect address spoofing.
- payload_len: A 16-bit field. The size of the packet, without the IPv6 header, can be up to 65,535 bytes. I will discuss larger packets ("jumbo frames") in the next section, when presenting the Hop-by-Hop Options header.
- nexthdr: When there are no extension headers, this will be the upper layer protocol number, like IPPROTO_UDP (17) for UDP or IPPROTO_TCP (6) for TCP. The list of available protocols is in include/uapi/linux/in.h. When using extension headers, this will be the type of the next header immediately following the IPv6 header. I will discuss extension headers in the next section.
- hop_limit: One byte field. Every forwarding device decrements the hop_limit counter by one. When it reaches zero, an ICMPv6 message is sent back and the packet is discarded. This parallels the TTL member in the IPv4 header. See the ip6_forward() method in net/ipv6/ip6_output.c.
- saddr: IPv6 source address (128 bit).
- daddr: IPv6 destination address (128 bit). This is possibly not the final packet destination if a Routing Header is used.

Note that, as opposed to the IPv4 header, there is no checksum in the IPv6 header. Checksumming is assumed to be assured by both Layer 2 and Layer 4. UDP in IPv4 permits having a checksum of 0, indicating no checksum; UDP in IPV6 requires having its own checksum normally. There are some special cases in IPv6 where zero UDP checksum is allowed for IPv6 UDP tunnels; see RFC 6935, "IPv6 and UDP Checksums for Tunneled Packets." In Chapter 4, which deals with the IPv4 subsystem, you saw that when forwarding a packet the ip_decrease_ttl() method is invoked. This method recomputes the checksum of the IPv4 header because the value of the ttl was changed. In IPv6, there is no such a need for recomputation of the checksum when forwarding a packet, because there is no checksum at all in the IPv6 header. This results in a performance improvement in software-based routers.

In this section, you have seen how the IPv6 header is built. You saw some differences between the IPv4 header and the IPv6 header—for example, in the IPv6 header there is no checksum and no header length. The next section discusses the IPv6 extension headers, which are the counterpart of IPv4 options.

Extension Headers

The IPv4 header can include IP options, which can extend the IPv4 header from a minimum size of 20 bytes to 60 bytes. In IPv6, we have optional extension headers instead. With one exception (Hop-by-Hop Options header), extension headers are not processed by any node along a packet's delivery path until the packet reaches its final destination; this improves the performance of the forwarding process significantly. The base IPv6 standard defines extension headers. An IPv6 packet can include 0, 1 or more extension headers. These headers can be placed between the IPv6 header and the upper-layer header in a packet. The `nexthdr` field of the IPv6 header is the number of the next header immediately after the IPv6 header. These extension headers are chained; every extension header has a Next Header field. In the last extension header, the Next Header indicates the upper-layer protocol (such as TCP, UDP, or ICMPv6). Another advantage of extension headers is that adding new extension headers in the future is easy and does not require any changes in the IPv6 header.

Extension headers must be processed strictly in the order they appear in the packet. Each extension header should occur at most once, except for the Destination Options header, which should occur at most twice. (See more detail later in this section in the description of the Destination Options header.) The Hop-by-Hop Options header must appear immediately after the IPv6 header; all other options can appear in any order. Section 4.1 of RFC 2460 ("Extension Header Order") states a recommended order in which extension headers should appear, but this is not mandatory. When an unknown Next Header number is encountered while processing a packet, an ICMPv6 "Parameter Problem" message with a code of "unknown Next Header" (ICMPV6_UNK_NEXTHDR) will be sent back to the sender by calling the `icmpv6_param_prob()` method. A description of the available ICMPv6 "Parameter Problem Codes" appears in Table 8-4 in the "Quick Reference" section at the end of this chapter.

Each extension header must be aligned on an 8-byte boundary. For extension headers of variable size, there is a Header Extension Length field, and they use padding if needed to ensure that they are aligned on an 8-byte boundary. The numbers of all Linux IPv6 extension headers and their Linux Kernel symbol representation are displayed in Table 8-2, "IPv6 extension headers," in the "Quick Reference" section at the end of this chapter.

A protocol handler is registered for each of the extension headers (except the Hop-by-Hop Options header) with the `inet6_add_protocol()` method. The reason for not registering a protocol handler for the Hop-by-Hop Options header is that there is a special method for parsing the Hop-by-Hop Options header, the `ipv6_parse_hopopts()` method. This method is invoked before calling the protocol handlers. (See the `ipv6_rcv()` method, `net/ipv6/ip6_input.c`). As mentioned before, the Hop-by-Hop Options header must be the first one, immediately following the IPv6 header. In this way, for example, the protocol handler for the Fragment extension header is registered:

```
static const struct inet6_protocol frag_protocol =
{
    .handler    =    ipv6_frag_rcv,
    .flags      =    INET6_PROTO_NOPOLICY,
};


int __init ipv6_frag_init(void)
{
    int ret;

    ret = inet6_add_protocol(&frag_protocol, IPPROTO_FRAGMENT);

(net/ipv6/reassembly.c)
```

Here is a description of all IPv6 Extension headers:

- **Hop-by-Hop Options header:** The Hop-by-Hop Options header must be processed on each node. It is parsed by the `ipv6_parse_hopopts()` method (`net/ipv6/exthdrs.c`).
- The Hop-by-Hop Options header must be immediately after the IPv6 header. It is used, for example, by the Multicast Listener Discovery protocol, as you will see in the "Multicast Listener Discovery (MLD)" section later in this chapter. The Hop-by-Hop Options header includes a variable-length option field. Its first byte is its type, which can be one of the following:
    - Router Alert (Linux Kernel symbol: IPV6_TLV_ROUTERALERT, value: 5). See RFC 6398, "IP Router Alert Considerations and Usage."
    - Jumbo (Linux Kernel symbol: IPV6_TLV_JUMBO, value: 194). The IPv6 packet payload normally can be up to 65,535 bytes long. With the jumbo option, it can be up to 2^32 bytes. See RFC 2675, "IPv6 Jumbograms."
    - Pad1 (Linux Kernel symbol: IPV6_TLV_PAD1, value: 0). The Pad1 option is used to insert one byte of padding. When more than one padding byte is needed, the PadN option (see next) should be used (and not multiple Pad1 options). See section 4.2 of RFC 2460.
    - PadN (Linux Kernel symbol: IPV6_TLV_PADN, value: 1). The PadN option is used to insert two or more octets of padding into the Options area of a header.
- **Routing Options header:** This parallels the IPv4 Loose Source Record Route (IPOPT_LSRR), which is discussed in the "IP Options" section in Chapter 4. It provides the ability to specify one or more routers that should be visited along the packet's traversal route to its final destination.
- **Fragment Options header:** As opposed to IPv4, fragmentation in IPv6 can occur only on the host that sends the packet, not on any of the intermediate nodes. Fragmentation is implemented by the `ip6_fragment()` method, which is invoked from the `ip6_finish_output()` method. In the `ip6_fragment()` method, there is a slow path and a fast path, much the same as in IPv4 fragmentation. The implementation of IPv6 fragmentation is in `net/ipv6/ip6_output.c`, and the implementation of IPv6 defragmentation is in `net/ipv6/reassembly.c`.
- **Authentication Header:** The Authentication header (AH) provides data authentication, data integrity, and anti-replay protection. It is described in RFC 4302, "IP Authentication Header," which makes RFC 2402 obsolete.
- **Encapsulating Security Payload Options header:** It is described in RFC 4303, "IP Encapsulating Security Payload (ESP)," which makes RFC 2406 obsolete. Note: The Encapsulating Security Payload (ESP) protocol is discussed in Chapter 10, which discusses the IPsec subsystem.
- **Destination Options header:** The Destination Options header can appear twice in a packet; before a Routing Options header, and after it. When it is before the Routing Options header, it includes information that should be processed by the routers that are specified by the Router Options header. When it is after the Router Options header, it includes information that should be processed by the final destination.

In the next section, you will see how the IPv6 protocol handler, which is the `ipv6_rcv()` method, is associated with IPv6 packets.

IPv6 Initialization

The inet6_init() method performs various IPv6 initializations (like procfs initializations, registration of protocol handlers for TCPv6, UDPv6 and other protocols), initialization of IPv6 subsystems (like IPv6 neighbour discovery, IPv6 Multicast Routing, and IPv6 routing subsystem) and more. For more details, look in net/ipv6/af_inet6.c. The ipv6_rcv() method is registered as a protocol handler for IPv6 packets by defining a packet_type object for IPv6 and registering it with the dev_add_pack() method, quite similarly to what is done in IPv4:

```
static struct packet_type ipv6_packet_type __read_mostly = {
        .type = cpu_to_be16(ETH_P_IPV6),
        .func = ipv6_rcv,
};

static int __init ipv6_packet_init(void)
{
        dev_add_pack(&ipv6_packet_type);
        return 0;
}
```

(net/ipv6/af_inet6.c)

As a result of the registration just shown, each Ethernet packet whose ethertype is ETH_P_IPV6 (0x86DD) will be handled by the ipv6_rcv() method. Next, I will discuss the IPv6 Autoconfiguration mechanism for setting IPv6 addresses.

Autoconfiguration

Autoconfiguration is a mechanism that allows a host to obtain or create a unique address for each of its interfaces. The IPv6 autoconfiguration process is initiated at system startup; nodes (both hosts and routers) generate a link-local address for their interfaces. This address is regarded as “tentative” (the interface flag IFA_F_TENTATIVE is set); this means that it can communicate only with neighbour discovery messages. It should be verified that this address is not already in use by another node on the link. This is done with the DAD (Duplicate Address Detection) mechanism, which was described in the previous chapter which deals with the Linux Neighbouring Subsystem. If the node is not unique, the autoconfiguration process will stop and manual configuration will be needed. In cases where the address is unique, the autoconfiguration process will continue. The next phase of autoconfiguration of hosts involves sending one or more Router Solicitations to the all routers multicast group address (ff02::2). This is done by calling the ndisc_send_rs() method from the addrconf_dad_completed() method. Routers reply with a Router Advertisement message, which is sent to the all hosts address, ff02::1. Both the Router Solicitation and the Router Advertisement use the Neighbour Discovery Protocol via ICMPv6 messages. The router solicitation ICMPv6 type is NDISC_ROUTER_SOLICITATION (133), and the router advertisement ICMPv6 type is NDISC_ROUTER_ADVERTISEMENT (134).

The radvd daemon is an example of an open source Router Advertisement daemon that is used for stateless autoconfiguration (http://www.litech.org/radvd/). You can set a prefix in the radvd configuration file, which will be sent in Router Advertisement messages. The radvd daemon sends Router Advertisements periodically. Apart from that, it also listens to Router Solicitations (RS) requests and answers with Router Advertisement (RA) reply messages. These Router Advertisement (RA) messages include a prefix field, which plays an important role in the autoconfiguration process, as you will immediately see. The prefix must be 64 bits long. When a host receives the Router Advertisement (RA) message, it configures its IP address based on this prefix and its own MAC address. If the Privacy Extensions feature (CONFIG_IPV6_PRIVACY) was set, there is also an element of randomness added in the IPv6 address creation. The Privacy Extensions mechanism avoids getting details about the identity of a machine from its IPv6 address, which is generated normally using its MAC address and a prefix, by adding randomness as was mentioned earlier. For more details on Privacy Extensions, see RFC 4941, “Privacy Extensions for Stateless Address Autoconfiguration in IPv6.”

When a host receives a Router Advertisement message, it can automatically configure its address and some other parameters. It can also choose a default router based on these advertisements. It is also possible to set a **preferred lifetime** and a **valid lifetime** for the addresses that are configured automatically on the hosts. The preferred lifetime value specifies the length of time in seconds that the address, which was generated from the prefix via stateless address autoconfiguration, remains in a preferred state. When the preferred time is over, this address will stop communicating (will not answer `ping6`, etc.). The valid lifetime value specifies the length of time in seconds that the address is valid (i.e., that applications already using it can keep using it); when this time is over, the address is removed. The preferred lifetime and the valid lifetime are represented in the kernel by the `prefered_lft` and the `valid_lft` fields of the `inet6_ifaddr` object, respectively (`include/net/if_inet6.h`).

Renumbering is the process of replacing an old prefix with a new prefix, and changing the IPv6 addresses of hosts according to a new prefix. Renumbering can also be done quite easily with `radvd`, by adding a new prefix to its configuration settings, setting a preferred lifetime and a valid lifetime, and restarting the `radvd` daemon. See also RFC 4192, "Procedures for Renumbering an IPv6 Network without a Flag Day," and RFCs 5887, 6866, and 6879.

The Dynamic Host Configuration Protocol version 6 (DHCPv6) is an example of stateful address configuration; in the stateful autoconfiguration model, hosts obtain interface addresses and/or configuration information and parameters from a server. Servers maintain a database that keeps track of which addresses have been assigned to which hosts. I will not delve into the details of the DHCPv6 protocol in this book. The DHCPv6 protocol is specified by RFC 3315, "Dynamic Host Configuration Protocol for IPv6 (DHCPv6)." The IPv6 Stateless Autoconfiguration standard is described in RFC 4862, "IPv6 Stateless Address Autoconfiguration."

You have learned in this section about the Autoconfiguration process, and you saw how easy it is to replace an old prefix with a new prefix by configuring and restarting `radvd`. The next section discusses how the `ipv6_rcv()` method, which is the IPv6 protocol handler, handles the reception of IPv6 packets in a somewhat similar way to what you saw in IPv4.

Receiving IPv6 Packets

The main IPv6 receive method is the `ipv6_rcv()` method, which is the handler for all IPv6 packets (including multicasts; there are no broadcasts in IPv6 as mentioned before). There are many similarities between the Rx path in IPv4 and in IPv6. As in IPv4, we first make some sanity checks, like checking that the version of the IPv6 header is 6 and that the source address is not a multicast address. (According to section 2.7 of RFC 4291, this is forbidden.) If there is a Hop-by-Hop Options header, it must be the first one. If the value of the `nexthdr` of the IPV6 header is `0`, this indicates a Hop-by-Hop Options header, and it is parsed by calling the `ipv6_parse_hopopts()` method. The real work is done by the `ip6_rcv_finish()` method, which is invoked by calling the NF_HOOK() macro. If there is a netfilter callback that is registered at this point (NF_INET_PRE_ROUTING), it will be invoked. I will discuss netfilter hooks in the next chapter. Let's take a look at the `ipv6_rcv()` method:

```
int ipv6_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt,
             struct net_device *orig_dev)
{
        const struct ipv6hdr *hdr;
        u32             pkt_len;
        struct inet6_dev *idev;
```

Fetch the network namespace from the network device that is associated with the Socket Buffer (SKB):

```
struct net *net = dev_net(skb->dev);

        . . .
```

Fetch the IPv6 header from the SKB:

```
hdr = ipv6_hdr(skb);
```

Perform some sanity checks, and discard the SKB if necessary:

```
        if (hdr->version != 6)
                goto err;

        /*
         * RFC4291 2.5.3
         * A packet received on an interface with a destination address
         * of loopback must be dropped.
         */
        if (!(dev->flags & IFF_LOOPBACK) &&
            ipv6_addr_loopback(&hdr->daddr))
                goto err;

        . . .

        /*
         * RFC4291 2.7
         * Multicast addresses must not be used as source addresses in IPv6
         * packets or appear in any Routing header.
         */
        if (ipv6_addr_is_multicast(&hdr->saddr))
                goto err;

        . . .
        if (hdr->nexthdr == NEXTHDR_HOP) {
                if (ipv6_parse_hopopts(skb) < 0) {
                        IP6_INC_STATS_BH(net, idev, IPSTATS_MIB_INHDRERRORS);
                        rcu_read_unlock();
                        return NET_RX_DROP;
                }
        }
        . . .

        return NF_HOOK(NFPROTO_IPV6, NF_INET_PRE_ROUTING, skb, dev, NULL,
                       ip6_rcv_finish);
err:
        IP6_INC_STATS_BH(net, idev, IPSTATS_MIB_INHDRERRORS);
drop:
        rcu_read_unlock();
        kfree_skb(skb);
        return NET_RX_DROP;
}
```

(net/ipv6/ip6_input.c)

The ip6_rcv_finish() method first performs a lookup in the routing subsystem by calling the ip6_route_input() method, in case there is no dst attached to the SKB. The ip6_route_input() method eventually invokes the fib6_rule_lookup().

```
int ip6_rcv_finish(struct sk_buff *skb)
 {
      . . .
      if (!skb_dst(skb))
                 ip6_route_input(skb);
```

Invoke the input callback of the dst attached to the SKB:

```
         return dst_input(skb);
 }
```

(net/ipv6/ip6_input.c)

■ **Note** There are two different implementations of the fib6_rule_lookup() method: one when Policy Routing (CONFIG_IPV6_MULTIPLE_TABLES) is set, in net/ipv6/fib6_rules.c, and one when Policy Routing is not set, in net/ipv6/ip6_fib.c.

As you saw in Chapter 5, which dealt with advanced topics of the IPv4 Routing Subsystem, the lookup in the routing subsystem builds a dst object and sets its input and output callbacks; in IPv6, similar tasks are performed. After the ip6_rcv_finish() method performs the lookup in the routing subsystem, it calls the dst_input() method, which in fact invokes the input callback of the dst object that is associated with the packet.

Figure 8-2 shows the receive path (Rx) of a packet that is received by the network driver. This packet can either be delivered to the local machine or be forwarded to another host. It is the result of the lookup in the routing tables that determines which of these two options will take place.

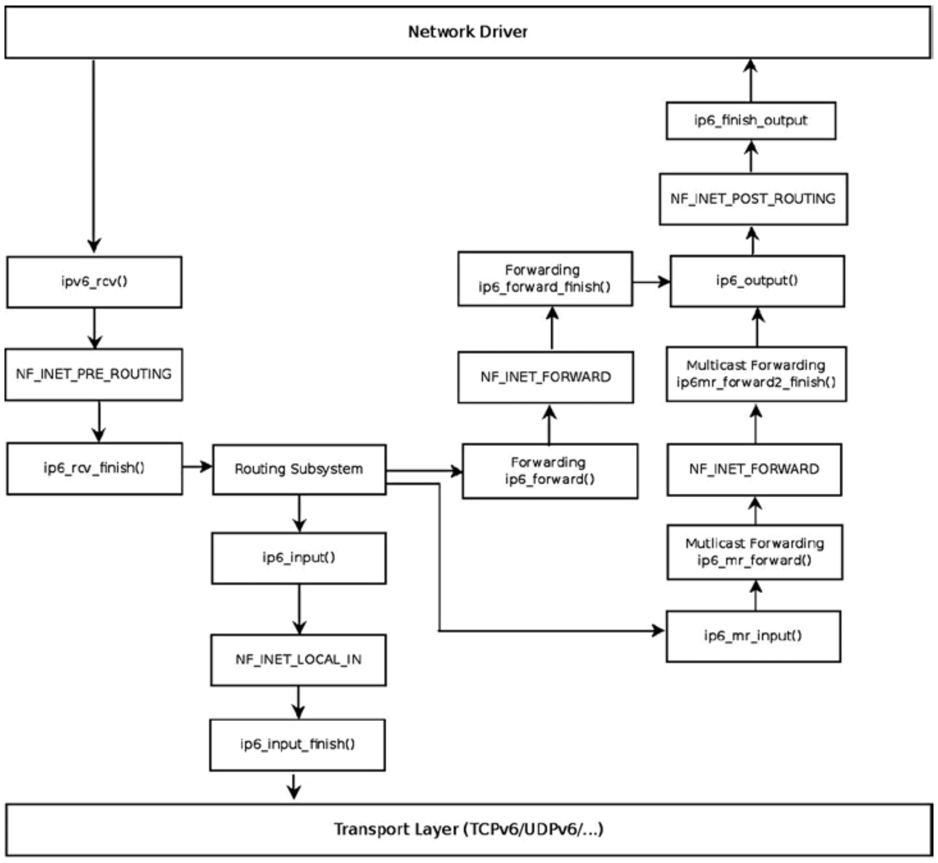


Figure 8-2. *Receiving IPv6 packets*

■ **Note** For simplicity, the diagram does not include the fragmentation/defragmentation/ parsing of extension headers /IPsec methods.

The lookup in the IPv6 routing subsystem will set the `input` callback of the destination cache (`dst`) to be:

- `ip6_input()` when the packet is destined to the local machine.
- `ip6_forward()` when the packet is to be forwarded.
- `ip6_mc_input()` when the packet is destined to a multicast address.
- `ip6_pkt_discard()` when the packet is to be discarded. The `ip6_pkt_discard()` method drops the packet and replies to the sender with a destination unreachable (ICMPV6_DEST_UNREACH) ICMPv6 message.

Incoming IPv6 packets can be locally delivered or forwarded; in the next section, you will learn about local delivery of IPv6 packets.

Local Delivery

Let's look first at the local delivery case: the `ip6_input()` method is a very short method:

```
int ip6_input(struct sk_buff *skb)
{
        return NF_HOOK(NFPROTO_IPV6, NF_INET_LOCAL_IN, skb, skb->dev, NULL,
                       ip6_input_finish);
}
```

(`net/ipv6/ip6_input.c`)

If there is a netfilter hook registered in this point (NF_INET_LOCAL_IN) it will be invoked. Otherwise, we will proceed to the `ip6_input_finish()` method:

```
static int ip6_input_finish(struct sk_buff *skb)
{
        struct net *net = dev_net(skb_dst(skb)->dev);
        const struct inet6_protocol *ipprot;
```

The `inet6_dev` structure (`include/net/if_inet6.h`) is the IPv6 parallel of the IPv4 `in_device` structure. It contains IPv6-related configuration such as the network interface unicast address list (`addr_list`) and the network interface multicast address list (`mc_list`). This IPv6-related configuration can be set by the user with the `ip` command or with the `ifconfig` command.

```
        struct inet6_dev *idev;
        unsigned int nhoff;
        int nexthdr;
        bool raw;

        /*
         *      Parse extension headers
         */

        rcu_read_lock();
resubmit:
        idev = ip6_dst_idev(skb_dst(skb));
        if (!pskb_pull(skb, skb_transport_offset(skb)))
                goto discard;
        nhoff = IP6CB(skb)->nhoff;
```

Fetch the next header number from the SKB:

```
nexthdr = skb_network_header(skb)[nhoff];
```

First in case of a raw socket packet, we try to deliver it to a raw socket:

```
raw = raw6_local_deliver(skb, nexthdr);
```

Every extension header (except the Hop by Hop extension header) has a protocol handler which was registered by the `inet6_add_protocol()` method; this method in fact adds an entry to the global `inet6_protos` array (see `net/ipv6/protocol.c`).

```
if ((ipprot = rcu_dereference(inet6_protos[nexthdr])) != NULL) {
        int ret;

        if (ipprot->flags & INET6_PROTO_FINAL) {
                const struct ipv6hdr *hdr;

                /* Free reference early: we don't need it any more,
                   and it may hold ip_conntrack module loaded
                   indefinitely. */
                nf_reset(skb);

                skb_postpull_rcsum(skb, skb_network_header(skb),
                                   skb_network_header_len(skb));
                hdr = ipv6_hdr(skb);
```

RFC 3810, which is the MLDv2 specification, says: "Note that MLDv2 messages are not subject to source filtering and must always be processed by hosts and routers." We do not want to discard MLD multicast packets due to source filtering, since these MLD packets should be always processed according to the RFC. Therefore, before discarding the packet we make sure that if the destination address of the packet is a multicast address, the packet is not an MLD packet. This is done by calling the `ipv6_is_mld()` method before discarding it. If this method indicates that the packet is an MLD packet, it is not discarded. You can also see more about this in the "Multicast Listener Discovery (MLD)" section later in this chapter.

```
        if (ipv6_addr_is_multicast(&hdr->daddr) &&
            !ipv6_chk_mcast_addr(skb->dev, &hdr->daddr,
            &hdr->saddr) &&
            !ipv6_is_mld(skb, nexthdr, skb_network_header_len(skb)))
                goto discard;
}
```

When the INET6_PROTO_NOPOLICY flag is set, this indicates that there is no need to perform IPsec policy checks for this protocol:

```
        if (!(ipprot->flags & INET6_PROTO_NOPOLICY) &&
            !xfrm6_policy_check(NULL, XFRM_POLICY_IN, skb))
                goto discard;
        ret = ipprot->handler(skb);
        if (ret > 0)
                goto resubmit;
        else if (ret == 0)
                IP6_INC_STATS_BH(net, idev, IPSTATS_MIB_INDELIVERS);
} else {
        if (!raw) {
                if (xfrm6_policy_check(NULL, XFRM_POLICY_IN, skb)) {
                        IP6_INC_STATS_BH(net, idev,
                                         IPSTATS_MIB_INUNKNOWNPROTOS);
                        icmpv6_send(skb, ICMPV6_PARAMPROB,
                                    ICMPV6_UNK_NEXTHDR, nhoff);
                }
                kfree_skb(skb);
        } else {
```

Everything went fine, so increment the INDELIVERS SNMP MIB counter (/proc/net/snmp6/Ip6InDelivers) and free the packet with the consume_skb() method:

```
                        IP6_INC_STATS_BH(net, idev, IPSTATS_MIB_INDELIVERS);
                        consume_skb(skb);
                }
        }
        rcu_read_unlock();
        return 0;

discard:
        IP6_INC_STATS_BH(net, idev, IPSTATS_MIB_INDISCARDS);
        rcu_read_unlock();
        kfree_skb(skb);
        return 0;
}
```

(net/ipv6/ip6_input.c)

You have seen the implementation details of local delivery, which is performed by the ip6_input() and ip6_input_finish() methods. Now is the time to turn to the implementation details of forwarding in IPv6. Also here, there are many similarities between forwarding in IPv4 and forwarding in IPv6.

Forwarding

Forwarding in IPv6 is very similar to forwarding in IPv4. There are some slight changes, though. For example, in IPv6, a checksum is not calculated when forwarding a packet. (There is no checksum field at all in an IPv6 header, as was mentioned before.) Let's take a look at the ip6_forward() method:

```
int ip6_forward(struct sk_buff *skb)
{
        struct dst_entry *dst = skb_dst(skb);
        struct ipv6hdr *hdr = ipv6_hdr(skb);
        struct inet6_skb_parm *opt = IP6CB(skb);
        struct net *net = dev_net(dst->dev);
        u32 mtu;
```

The IPv6 procfs forwarding entry (/proc/sys/net/ipv6/conf/all/forwarding) should be set:

```
if (net->ipv6.devconf_all->forwarding == 0)
        goto error;
```

When working with Large Receive Offload (LRO), the packet length will exceed the Maximum transmission unit (MTU). As in IPv4, when LRO is enabled, the SKB is freed and an error of –EINVAL is returned:

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

if (!xfrm6_policy_check(NULL, XFRM_POLICY_FWD, skb)) {
        IP6_INC_STATS(net, ip6_dst_idev(dst), IPSTATS_MIB_INDISCARDS);
        goto drop;
}
```

Drop packets that are not destined to go to the local host. The pkt_type associated with an SKB is determined according to the destination MAC address in the Ethernet header of an incoming packet. This is done by the eth_type_trans() method, which is typically called in the network device driver when handling an incoming packet. See the eth_type_trans() method, net/ethernet/eth.c.

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

skb_forward_csum(skb);

/*
 *      We DO NOT make any processing on
 *      RA packets, pushing them to user level AS IS
 *      without any WARRANTY that application will be able
 *      to interpret them. The reason is that we
 *      cannot make anything clever here.
 *
 *      We are not end-node, so that if packet contains
 *      AH/ESP, we cannot make anything.
 *      Defragmentation also would be mistake, RA packets
 *      cannot be fragmented, because there is no warranty
 *      that different fragments will go along one path. --ANK
 */
if (opt->ra) {
        u8 *ptr = skb_network_header(skb) + opt->ra;
```

We should try to deliver the packet to sockets that had the IPV6_ROUTER_ALERT socket option set by setsockopt(). This is done by calling the ip6_call_ra_chain() method; if the delivery in ip6_call_ra_chain() succeeded, the ip6_forward() method returns 0 and the packet is not forwarded. See the implementation of the ip6_call_ra_chain() method in net/ipv6/ip6_output.c.

```
        if (ip6_call_ra_chain(skb, (ptr[2]<<8) + ptr[3]))
                return 0;
}

/*
 *      check and decrement ttl
 */
if (hdr->hop_limit <= 1) {
        /* Force OUTPUT device used as source address */
        skb->dev = dst->dev;
```

Send back an ICMP error message when the Hop Limit is 1 (or less), much like what we have in IPv4 when forwarding a packet and the TTL reaches 0. In this case, the packet is discarded:

```
        icmpv6_send(skb, ICMPV6_TIME_EXCEED, ICMPV6_EXC_HOPLIMIT, 0);
        IP6_INC_STATS_BH(net,
                         ip6_dst_idev(dst), IPSTATS_MIB_INHDRERRORS);

        kfree_skb(skb);
        return -ETIMEDOUT;
}
```

```
/* XXX: idev->cnf.proxy_ndp? */
if (net->ipv6.devconf_all->proxy_ndp &&
    pneigh_lookup(&nd_tbl, net, &hdr->daddr, skb->dev, 0)) {
        int proxied = ip6_forward_proxy_check(skb);
        if (proxied > 0)
                return ip6_input(skb);
        else if (proxied < 0) {
                IP6_INC_STATS(net, ip6_dst_idev(dst),
                              IPSTATS_MIB_INDISCARDS);
                goto drop;
        }
}

if (!xfrm6_route_forward(skb)) {
        IP6_INC_STATS(net, ip6_dst_idev(dst), IPSTATS_MIB_INDISCARDS);
        goto drop;
}
dst = skb_dst(skb);

/* IPv6 specs say nothing about it, but it is clear that we cannot
   send redirects to source routed frames.
   We don't send redirects to frames decapsulated from IPsec.
 */
if (skb->dev == dst->dev && opt->srcrt == 0 && !skb_sec_path(skb)) {
        struct in6_addr *target = NULL;
        struct inet_peer *peer;
        struct rt6_info *rt;

        /*
         *      incoming and outgoing devices are the same
         *      send a redirect.
         */

        rt = (struct rt6_info *) dst;
        if (rt->rt6i_flags & RTF_GATEWAY)
                target = &rt->rt6i_gateway;
        else
                target = &hdr->daddr;

        peer = inet_getpeer_v6(net->ipv6.peers, &rt->rt6i_dst.addr, 1);

        /* Limit redirects both by destination (here)
           and by source (inside ndisc_send_redirect)
         */
        if (inet_peer_xrlim_allow(peer, 1*HZ))
        ndisc_send_redirect(skb, target);
        if (peer)
        inet_putpeer(peer);
} else {
        int addrtype = ipv6_addr_type(&hdr->saddr);
        /* This check is security critical. */
        if (addrtype == IPV6_ADDR_ANY ||
            addrtype & (IPV6_ADDR_MULTICAST | IPV6_ADDR_LOOPBACK))
        goto error;
        if (addrtype & IPV6_ADDR_LINKLOCAL) {
                icmpv6_send(skb, ICMPV6_DEST_UNREACH,
                            ICMPV6_NOT_NEIGHBOUR, 0);
                goto error;
        }
}
```

Note that the IPv6 IPV6_MIN_MTU is 1280 bytes, according to section 5, "Packet Size Issues," of the base IPv6 standard, RFC 2460.

```
mtu = dst_mtu(dst);
if (mtu < IPV6_MIN_MTU)
        mtu = IPV6_MIN_MTU;

if ((!skb->local_df && skb->len > mtu && !skb_is_gso(skb)) ||
    (IP6CB(skb)->frag_max_size && IP6CB(skb)->frag_max_size > mtu)) {
        /* Again, force OUTPUT device used as source address */
        skb->dev = dst->dev;
```

Reply back to the sender with an ICMPv6 message of "Packet Too Big," and free the SKB; the `ip6_forward()` method returns –EMSGSIZ in this case:

```
        icmpv6_send(skb, ICMPV6_PKT_TOOBIG, 0, mtu);
        IP6_INC_STATS_BH(net,
                         ip6_dst_idev(dst), IPSTATS_MIB_INTOOBIGERRORS);
        IP6_INC_STATS_BH(net,
                         ip6_dst_idev(dst), IPSTATS_MIB_FRAGFAILS);
        kfree_skb(skb);
        return -EMSGSIZE;
}
if (skb_cow(skb, dst->dev->hard_header_len)) {
        IP6_INC_STATS(net, ip6_dst_idev(dst), IPSTATS_MIB_OUTDISCARDS);
        goto drop;
}

hdr = ipv6_hdr(skb);
```

The packet is to be forwarded, so decrement the `hop_limit` of the IPv6 header.

```
/* Mangling hops number delayed to point after skb COW */
hdr->hop_limit--;

IP6_INC_STATS_BH(net, ip6_dst_idev(dst), IPSTATS_MIB_OUTFORWDATAGRAMS);
IP6_ADD_STATS_BH(net, ip6_dst_idev(dst), IPSTATS_MIB_OUTOCTETS, skb->len);
return NF_HOOK(NFPROTO_IPV6, NF_INET_FORWARD, skb, skb->dev, dst->dev,
               ip6_forward_finish);
```

```
error:
       IP6_INC_STATS_BH(net, ip6_dst_idev(dst), IPSTATS_MIB_INADDRERRORS);
drop:
       kfree_skb(skb);
       return -EINVAL;
}
```

(net/ipv6/ip6_output.c)

The ip6_forward_finish() method is a one-line method, which simply invokes the destination cache (dst) output callback:

```
static inline int ip6_forward_finish(struct sk_buff *skb)
{
return dst_output(skb);
}
```

(net/ipv6/ip6_output.c)

You have seen in this section how the reception of IPv6 packets is handled, either by local delivery or by forwarding. You have also seen some differences between receiving IPv6 packets and receiving IPv4 packets. In the next section, I will discuss the Rx path for multicast traffic.

Receiving IPv6 Multicast Packets

The ipv6_rcv() method is the IPv6 handler for both unicast packets and multicast packets. As mentioned above, after some sanity checks, it invokes the ip6_rcv_finish() method, which performs a lookup in the routing subsystem by calling the ip6_route_input() method. In the ip6_route_input() method, the input callback is set to be the ip6_mc_input method in cases of receiving a multicast packet. Let's take a look at the ip6_mc_input() method:

```
int ip6_mc_input(struct sk_buff *skb)
{
        const struct ipv6hdr *hdr;
        bool deliver;

        IP6_UPD_PO_STATS_BH(dev_net(skb_dst(skb)->dev),
                         ip6_dst_idev(skb_dst(skb)), IPSTATS_MIB_INMCAST,
                         skb->len);

        hdr = ipv6_hdr(skb);
```

The ipv6_chk_mcast_addr() method (net/ipv6/mcast.c) checks whether the multicast address list (mc_list) of the specified network device contains the specified multicast address (which is the destination address in the IPv6 header in this case, hdr->daddr). Note that because the third parameter is NULL, we do not check in this invocation whether there are any source filters for the source address; handling source filtering is discussed later in this chapter.

```
deliver = ipv6_chk_mcast_addr(skb->dev, &hdr->daddr, NULL);
```

If the local machine is a multicast router (that is, CONFIG_IPV6_MROUTE is set), we continue after some checks to the ip6_mr_input() method. The IPv6 multicast routing implementation is very similar to the IPv4 multicast routing implementation, which was discussed in Chapter 6, so I will not discuss it in this book. The IPv6 multicast routing implementation is in net/ipv6/ip6mr.c. Support for IPv6 Multicast Routing was added in kernel 2.6.26 (2008), based on a patch by Mickael Hoerdt.

```
#ifdef CONFIG_IPV6_MROUTE
. . .
         if (dev_net(skb->dev)->ipv6.devconf_all->mc_forwarding &&
             !(ipv6_addr_type(&hdr->daddr) &
               (IPV6_ADDR_LOOPBACK|IPV6_ADDR_LINKLOCAL)) &&
             likely(!(IP6CB(skb)->flags & IP6SKB_FORWARDED))) {
                 /*
                  * Okay, we try to forward - split and duplicate
                  * packets.
                  */
                 struct sk_buff *skb2;

                 if (deliver)
                         skb2 = skb_clone(skb, GFP_ATOMIC);
                 else {
                         skb2 = skb;
                         skb = NULL;
                 }

                 if (skb2) {
```

Continue to the IPv6 Multicast Routing code, via the `ip6_mr_input()` method (`net/ipv6/ip6mr.c`):

```
                         ip6_mr_input(skb2);
                 }

            }
#endif
        if (likely(deliver))
                ip6_input(skb);
        else {
                /* discard */
                kfree_skb(skb);
        }

        return 0;
}
```

`(net/ipv6/ip6_input.c)`

When the multicast packet is not destined to be forwarded by multicast routing (for example, when CONFIG_IPV6_MROUTE is not set), we will continue to the `ip6_input()` method, which is in fact a wrapper around the `ip6_input_finish()` method as you already saw. In the `ip6_input_finish()` method, we again call the `ipv6_chk_mcast_addr()` method, but this time the third parameter is not NULL, it is the source address from the IPv6 header. This time we do check in the `ipv6_chk_mcast_addr()` method whether source filtering is set, and we handle the packet accordingly. Source filtering is discussed in the "Multicast Source Filtering (MSF)" section later in this chapter. Next, I will describe the Multicast Listener Discovery protocol, which parallels the IPv4 IGMPv3 protocol.

Multicast Listener Discovery (MLD)

The MLD protocol is used to exchange group information between multicast hosts and routers. The MLD protocol is an asymmetric protocol; it specifies different behavior to Multicast Routers and to Multicast Listeners. In IPv4, multicast group management is handled by the Internet Group Management Protocol (IGMP) protocol, as you saw in Chapter 6. In IPv6, multicast group management is handled by the MLDv2 protocol, which is specified in RFC 3810, from 2004. The MLDv2 protocol is derived from the IGMPv3 protocol, which is used by IPv4. However, as opposed to the IGMPv3 protocol, MLDv2 is part of the ICMPv6 protocol, while IGMPv3 is a standalone protocol that does not use any of the ICMPv4 services; this is the main reason why the IGMPv3 protocol is not used in IPv6. Note that you might encounter the term GMP (Group Management Protocol), which is used to refer to both IGMP and MLD.

The former version of the Multicast Listener Discovery protocol is MLDv1, and it is specified in RFC 2710; it is derived from IGMPv2. MLDv1 is based on the Any-Source Multicast (ASM) model; this means that you do not specify interest in receiving multicast traffic from a single source address or from a set of addresses. MLDv2 extends MLDv1 by adding support for Source Specific Multicast (SSM); this means the ability of a node to specify interest in including or excluding listening to packets from specific unicast source addresses. This feature is referred to as **source filtering**. Later in this section, I will show a short, detailed userspace example of how to use source filtering. See more in RFC 4604, "Using Internet Group Management Protocol Version 3 (IGMPv3) and Multicast Listener Discovery Protocol Version 2 (MLDv2) for Source-Specific Multicast."

The MLDv2 protocol is based on Multicast Listener Reports and Multicast Listener Queries. An MLDv2 Router (which is also sometimes termed "Querier") sends periodically Multicast Listener Queries in order to learn about the state of multicast groups of nodes. If there are several MLDv2 Routers on the same link, only one of them is selected to be the Querier, and all the other routers are set to be in a Non-Querier state. This is done by a Querier Election mechanism, as described in section 7.6.2 of RFC 3810. Nodes respond to these queries with Multicast Listener Reports, in which they provide information about multicast groups to which they belong. When a listener wants to stop listening on some multicast group, it informs the Querier about it, and the Querier must query for other listeners of that multicast group address before deleting it from its Multicast Address Listener state. An MLDv2 router can provide state information about listeners to multicast routing protocols.

Now that you have learned generally what the MLD protocol is, I will turn your attention in the following section to how joining and leaving a multicast group is handled.

Joining and Leaving a Multicast Group

There are two ways to join or leave a multicast group in IPv6. The first one is from within the kernel, by calling the `ipv6_dev_mc_inc()` method, which gets as a parameter a network device object and a multicast group address. For example, when registering a network device, the `ipv6_add_dev()` method is invoked; each device should join the interface-local all nodes multicast group (`ff01::1`) and the link-local all nodes multicast group (`ff02::1`).

```
static struct inet6_dev *ipv6_add_dev(struct net_device *dev) {

. . .
        /* Join interface-local all-node multicast group */
        ipv6_dev_mc_inc(dev, &in6addr_interfacelocal_allnodes);

        /* Join all-node multicast group */
        ipv6_dev_mc_inc(dev, &in6addr_linklocal_allnodes);

. . .
}
```

(net/ipv6/addrconf.c)

Routers are devices that have their procfs forwarding entry, /proc/sys/net/ipv6/conf/all/forwarding, set. Routers join three multicast address groups, in addition to the two multicast group that each host joins and that were mentioned earlier. These are the link-local all-routers multicast group (ff02::2), interface-local all routers multicast group (ff01::2), and site-local all routers multicast group (ff05::2).

Note that setting the IPv6 procfs forwarding entry value is handled by the addrconf_fixup_forwarding() method, which eventually calls the dev_forward_change() method, which causes the specified network interface to join or leave these three multicast address groups according to the value of the procfs entry (which is represented by idev->cnf.forwarding, as you can see in the following code snippet):

```
static void dev_forward_change(struct inet6_dev *idev)
{
        struct net_device *dev;
        struct inet6_ifaddr *ifa;
     . . .
        dev = idev->dev;
     . . .
        if (dev->flags & IFF_MULTICAST) {
                if (idev->cnf.forwarding) {
                        ipv6_dev_mc_inc(dev, &in6addr_linklocal_allrouters);
                        ipv6_dev_mc_inc(dev, &in6addr_interfacelocal_allrouters);
                        ipv6_dev_mc_inc(dev, &in6addr_sitelocal_allrouters);
                } else {
                        ipv6_dev_mc_dec(dev, &in6addr_linklocal_allrouters);
                        ipv6_dev_mc_dec(dev, &in6addr_interfacelocal_allrouters);
                        ipv6_dev_mc_dec(dev, &in6addr_sitelocal_allrouters);
                }
        }
. . .
}
```

(net/ipv6/addrconf.c)

To leave a multicast group from within the kernel, you should call the ipv6_dev_mc_dec() method. The second way of joining a multicast group is by opening an IPv6 socket in userspace, creating a multicast request (ipv6_mreq object) and setting the ipv6mr_multiaddr of the request to be the multicast group address to which this host wants to join, and setting the ipv6mr_interface to the ifindex of the network interface it wants to set. Then it should call setsockopt() with the IPV6_JOIN_GROUP socket option:

```
int                sockd;
struct ipv6_mreq   mcgroup;
struct addrinfo    *results;
. . .

/* read an IPv6 multicast group address to which we want to join */
/* into the address info object (results) */
. . .
```

Set the network interface that we want to use (by its `ifindex` value):

```
mcgroup.ipv6mr_interface=3;
```

Set the multicast group address for the group that we want to join in the request (`ipv6mr_multiaddr`):

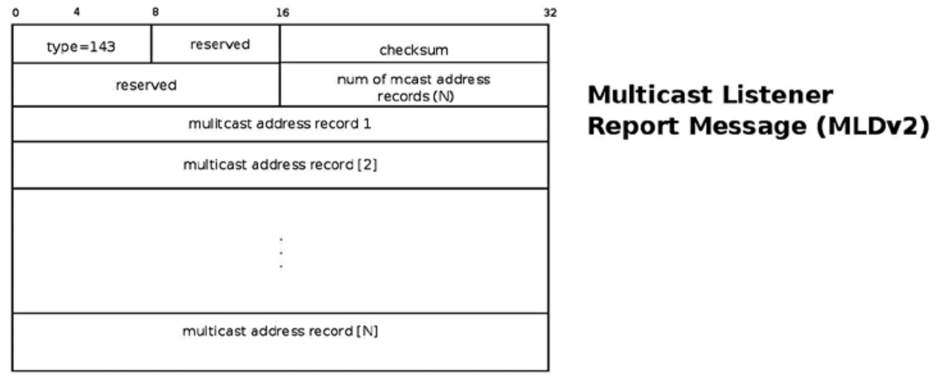
```
memcpy( &(mcgroup.ipv6mr_multiaddr),
     &(((struct sockaddr_in6 *) results->ai_addr)->sin6_addr),
     sizeof(struct in6_addr));

sockd  = socket(AF_INET6, SOCK_DGRAM,0);
```

Call `setsockopt()` with IPV6_JOIN_GROUP to join the multicast group; this call is handled in the kernel by the `ipv6_sock_mc_join()` method (`net/ipv6/mcast.c`).

```
status = setsockopt(sockd, IPPROTO_IPV6, IPV6_JOIN_GROUP,
                    &mcgroup, sizeof(mcgroup));
. . .
```

The IPV6_ADD_MEMBERSHIP socket option can be used instead of IPV6_JOIN_GROUP. (They are equivalent.) Note that we can set the same multicast group address on more than one network device by setting different values of network interfaces to `mcgroup.ipv6mr_interface`. The value of `mcgroup.ipv6mr_interface` is passed as the `ifindex` parameter to the `ipv6_sock_mc_join()` method. In such a case, the kernel builds and sends an MLDv2 Multicast Listener Report packet (ICMPV6_MLD2_REPORT), where the destination address is `ff02::16` (the all MLDv2-capable routers Multicast Group Address). According to section 5.2.14 in RFC 3810, all MLDv2-capable multicast routers should listen to this multicast address. The number of Multicast Address Records in the MLDv2 header (shown in Figure 8-3) will be 1, because only one Multicast Address Record is used, containing the address of the multicast group that we want to join. The multicast group address that a host wants to join is part of the ICMPv6 header. The Hop-by-Hop Options header with Router Alert is set in this packet. MLD packets contain a Hop-by-Hop Options header, which in turn contains a Router Alert options header; the next header of the Hop-by-Hop extension header is IPPROTO_ICMPV6 (58), because following the Hop-by-Hop header is the ICMPv6 packet, which contains the MLDv2 message.


0
4
8
16
32
type=143
reserved
checksum
reserved
num of mcast address records (N)
multcast address record 1
multicast address record [2]
multicast address record [N]
Multicast Listener Report Message (MLDv2)

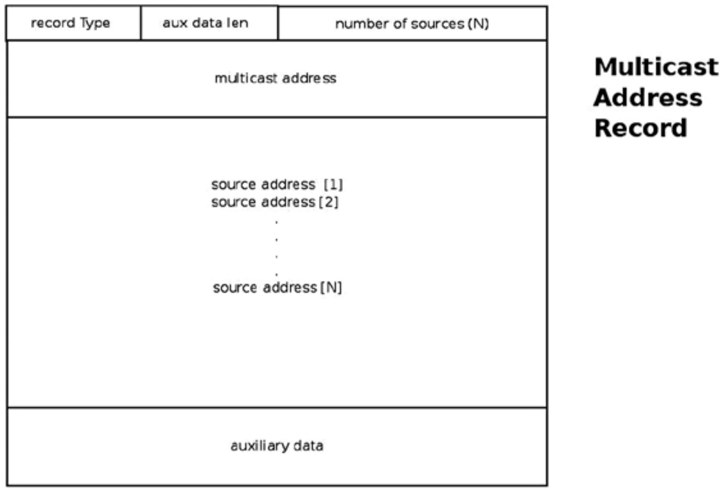


Figure 8-3. *MLDv2 Multicast Listener Report*

A host can leave a multicast group by calling `setsockopt()` with the IPV6_DROP_MEMBERSHIP socket option, which is handled in the kernel by calling the `ipv6_sock_mc_drop()` method or by closing the socket. Note that IPV6_LEAVE_GROUP is equivalent to IPV6_DROP_MEMBERSHIP.

After talking about how joining and leaving a multicast group is handled, it is time to see what an MLDv2 Multicast Listener Report is.

MLDv2 Multicast Listener Report

The MLDv2 Multicast Listener Report is represented in the kernel by the `mld2_report` structure:

```
struct mld2_report {
        struct icmp6hdr         mld2r_hdr;
        struct mld2_grec        mld2r_grec[0];
};
```

(include/net/mld.h)

The first member of the `mld2_report` structure is the `mld2r_hdr`, which is an ICMPv6 header; its `icmp6_type` should be set to ICMPV6_MLD2_REPORT (143). The second member of the `mld2_report` structure is the `mld2r_grec[0]`, an instance of the `mld2_grec` structure, which represents the MLDv2 group record. (This is the Multicast Address Record in Figure 8-3.) Following is the definition of the `mld2_grec` structure:

```
struct mld2_grec {
        __u8            grec_type;
        __u8            grec_auxwords;
        __be16          grec_nsrcs;
        struct in6_addr grec_mca;
        struct in6_addr grec_src[0];
};
```

(include/net/mld.h)

The following is a description of the members of the `mld2_grec` structure:

- `grec_type`: Specifies the type of the Multicast Address Record. See Table 8-3, "Multicast Address Record (record types)" in the "Quick Reference" section at the end of this chapter.
- `grec_auxwords`: The length of the Auxiliary Data (*aux data len* in Figure 8-3). The Auxiliary Data field, if present, contains additional information that pertains to this Multicast Address Record. Usually it is 0. See also section 5.2.10 in RFC 3810.
- `grec_nsrcs`: The number of source addresses.
- `grec_mca`: The multicast address to which this Multicast Address Record pertains.
- `grec_src[0]`: A unicast source address (or an array of unicast source addresses). These are addresses that we want to filter (block or allow).

In the next section, I will discuss the Multicast Source Filtering (MSF) feature. You will find in it detailed examples of how a Multicast Address Record is used in source filtering.

Multicast Source Filtering (MSF)

With Multicast Source Filtering, the kernel will drop the multicast traffic from sources other than the expected ones. This feature, which is also known as Source-Specific Multicast (SSM) was not part of MLDv1. It was introduced in MLDv2; see RFC 3810. It is the opposite of Any-Source Multicast (ASM), where a receiver expresses interest in a destination multicast address. To understand better what Multicast Source Filtering is all about, I will show here an example of a userspace application demonstrating how to join and leave a multicast group with source filtering.

Joining and Leaving a Multicast Group with Source Filtering

A host can join a multicast group with source filtering by opening an IPv6 socket in userspace, creating a multicast group source request (`group_source_req` object), and setting three parameters in the request:

- `gsr_group`: The multicast group address that this host wants to join
- `gsr_source`: The multicast group source address that it wants to allow
- `ipv6mr_interface`: The *ifindex* of the network interface it wants to set

Then it should call `setsockopt()` with the MCAST_JOIN_SOURCE_GROUP socket option. Following here is a code snippet of a userspace application demonstrating this (checking the success of the system calls was removed, for brevity):

```
int                         sockd;
struct group_source_req     mreq;
struct addrinfo             *results1;
struct addrinfo             *results2;

/* read an IPv6 multicast group address that we want to join into results1 */
/* read an IPv6 multicast group address which we want to allow into results2 */
memcpy(&(mreq.gsr_group),  results1->ai_addr,  sizeof(struct sockaddr_in6));
memcpy(&(mreq.gsr_source), results2->ai_addr,  sizeof(struct sockaddr_in6));

mreq.gsr_interface = 3;

sockd = socket(AF_INET6, SOCK_DGRAM, 0);
setsockopt(sockd, IPPROTO_IPV6, MCAST_JOIN_SOURCE_GROUP, &mreq, sizeof(mreq));
```

This request is handled in the kernel first by the `ipv6_sock_mc_join()` method, and then by the `ip6_mc_source()` method. To leave the group, you should call `setsockopt()` with the MCAST_LEAVE_SOURCE_GROUP socket option or close the socket that you opened.

You can set another address that you want to allow and again call `setsockopt()` with this socket with the MCAST_UNBLOCK_SOURCE socket option. This will add additional addresses to the source filter list. Each such call to `setsockopt()` will trigger sending an MLDv2 Multicast Listener Report message with one Multicast Address Record; the Record Type will be 5 ("Allow new sources"), and the number of sources will be 1 (the unicast address that you want to unblock). I will show now an example of using the MCAST_MSFILTER socket option for source filtering.

Example: Using MCAST_MSFILTER for Source Filtering

You can also block or permit multicast traffic from several multicast addresses in one `setsockopt()` call using MCAST_MSFILTER and a `group_filter` object. First, let's take a look at the definition of the `group_filter` structure definition in userspace, which is quite self-explanatory:

```
struct group_filter
  {
    /* Interface index.  */
    uint32_t gf_interface;

    /* Group address.  */
    struct sockaddr_storage gf_group;

    /* Filter mode.  */
    uint32_t gf_fmode;

    /* Number of source addresses.  */
    uint32_t gf_numsrc;
    /* Source addresses.  */
    struct sockaddr_storage gf_slist[1];
};
```

(include/netinet/in.h)

The Filter mode (gf_fmode) can be MCAST_INCLUDE (when you want to allow multicast traffic from some unicast address) or MCAST_EXCLUDE (when you want to disallow multicast traffic from some unicast address). Following are two examples for this; the first will allow multicast traffic from three resources, and the second will disallow multicast traffic from two resources:

```
struct ipv6_mreq        mcgroup;
struct group_filter     filter;
struct sockaddr_in6     *psin6;

int                     sockd[2];
```

Set the multicast group address that we want to join, `ffff::9`.

```
inet_pton(AF_INET6,"ffff::9", &mcgroup.ipv6mr_multiaddr);
```

Set the network interface that we want to use by its `ifindex` (here, we use `eth0`, which has an `ifindex` value of 2):

```
mcgroup.ipv6mr_interface=2;
```

Set the filter parameters: use the same `ifindex` (2), use MCAST_INCLUDE to set the filter to allow traffic from the sources that are specified by the filter, and set `gf_numsrc` to 3, because we want to prepare a filter of 3 unicast addresses:

```
filter.gf_interface = 2;
```

We want to prepare two filters: the first one will allow traffic from a set of three multicast addresses, and the second one will permit traffic from a set of two multicast addresses. First set the filter mode to MCAST_INCLUDE, which means to allow traffic from this filter:

```
filter.gf_fmode = MCAST_INCLUDE;
```

Set the number of source addresses of the filter (`gf_numsrc`) to be 3:

```
filter.gf_numsrc = 3;
```

Set the group address of the filter (`gf_group`) to be the same one that we use for the `mcgrouop` earlier, `ffff::9`:

```
psin6 = (struct sockaddr_in6 *)&filter.gf_group;
psin6->sin6_family = AF_INET6;
inet_pton(PF_INET6, "ffff::9", &psin6->sin6_addr);
```

The three unicast addresses that we want to allow are `2000::1`, `2000::2`, and `2000::3`.

Set `filter.gf_slist[0]`, `filter.gf_slist[1]`, and `filter.gf_slist[2]` accordingly:

```
psin6 = (struct sockaddr_in6 *)&filter.gf_slist[0];
psin6->sin6_family = AF_INET6;
inet_pton(PF_INET6, "2000::1", &psin6->sin6_addr);
```

```
psin6 = (struct sockaddr_in6 *)&filter.gf_slist[1];
psin6->sin6_family = AF_INET6;
inet_pton(PF_INET6, "2000::2", &psin6->sin6_addr);

psin6 = (struct sockaddr_in6 *)&filter.gf_slist[2];
psin6->sin6_family = AF_INET6;
inet_pton(PF_INET6, "2000::3",&psin6->sin6_addr);
```

Create a socket, and join a multicast group:

```
sockd[0] = socket(AF_INET6, SOCK_DGRAM,0);
status = setsockopt(sockd[0], IPPROTO_IPV6, IPV6_JOIN_GROUP,
        &mcgroup, sizeof(mcgroup));
```

Activate the filter we created:

```
status=setsockopt(sockd[0], IPPROTO_IPV6, MCAST_MSFILTER, &filter,
   GROUP_FILTER_SIZE(filter.gf_numsrc));
```

This will trigger sending of an MLDv2 Multicast Listener Report (ICMPV6_MLD2_REPORT) to all MLDv2 routers (`ff02::16`) with a Multicast Address Record object (`mld2_grec`) embedded in it. (See the description of the `mld2_report` structure and Figure 8-3 earlier.) The values of the fields of `mld2_grec` will be as follows:

- `grec_type` will be MLD2_CHANGE_TO_INCLUDE (3).
- `grec_auxwords` will be 0. (We do not use Auxiliary Data.)
- `grec_nsrcs` is 3 (because we want to use a filter with 3 source addresses and we set `gf_numsrc` to 3).
- `grec_mca` will be `ffff::9`; this is the multicast group address that the Multicast Address Record pertains to.

The following three unicast source addresses:

- `grec_src[0]` is `2000::1`
- `grec_src[1]` is `2000::2`
- `grec_src[2]` is `2000::3`

Now we want to create a filter of 2 unicast source addresses that we want to exclude. So first create a new userspace socket:

```
sockd[1] = socket(AF_INET6, SOCK_DGRAM,0);
```

Set the filter mode to EXCLUDE, and set the number of sources of the filter to be 2:

```
filter.gf_fmode = MCAST_EXCLUDE;
filter.gf_numsrc = 2;
```

Set the two addresses we want to exclude, 2001::1 and 2001::2:

```
psin6 = (struct sockaddr_in6 *)&filter.gf_slist[0];
psin6->sin6_family = AF_INET6;
inet_pton(PF_INET6, "2001::1", &psin6->sin6_addr);

psin6 = (struct sockaddr_in6 *)&filter.gf_slist[1];
psin6->sin6_family = AF_INET6;
inet_pton(PF_INET6, "2001::2", &psin6->sin6_addr);
```

Create a socket, and join a multicast group:

```
status = setsockopt(sockd[1], IPPROTO_IPV6, IPV6_JOIN_GROUP,
        &mcgroup, sizeof(mcgroup));
```

Activate the filter:

```
status=setsockopt(sockd[1], IPPROTO_IPV6, MCAST_MSFILTER, &filter,
       GROUP_FILTER_SIZE(filter.gf_numsrc));
```

This again will trigger the sending of an MLDv2 Multicast Listener Report (ICMPV6_MLD2_REPORT) to all MLDv2 routers (`ff02::16`). This time the content of the Multicast Address Record object (`mld2_grec`) will be different:

- `grec_type` will be MLD2_CHANGE_TO_EXCLUDE (4).
- `grec_auxwords` will be 0. (We do not use Auxiliary Data.)
- `grec_nsrcs` is 2 (because we want to use 2 source addresses and we set `gf_numsrc` to 2).
- `grec_mca` will be `ffff::9`, as before; this is the multicast group address that the Multicast Address Record pertains to.
- The following two unicast source addresses:
  - `grec_src[0]` is `2001::1`
  - `grec_src[1]` is `2002::2`

■ **Note** We can display the source filtering mapping that we created by `cat/proc/net/mcfilter6`; this is handled in the kernel by the `igmp6_mcf_seq_show()` method.

For example, the first three entries in this mapping will show that for the `ffff::9` multicast address, we permit (INCLUDE) multicast traffic from `2000::1`, `2000::2`, and `2000::3`. Note that for the first three entries the value in the INC (Include) column is 1. For the fourth and fifth entries, we disallow traffic from `2001::1` and `2001::2`. Note that the value in the EX (Exclude) column is 1 for the fourth and fifth entries.

```
cat  /proc/net/mcfilter6
Idx Device Multicast Address                Source Address                   INC    EXC
  2   eth0 ffff0000000000000000000000000009 20000000000000000000000000000001 1       0
  2   eth0 ffff0000000000000000000000000009 20000000000000000000000000000002 1       0
  2   eth0 ffff0000000000000000000000000009 20000000000000000000000000000003 1       0
  2   eth0 ffff0000000000000000000000000009 20010000000000000000000000000001 0       1
  2   eth0 ffff0000000000000000000000000009 20010000000000000000000000000002 0       1
```

■ **Note** Creating filters by calling the `setsockopt()` method with MCAST_MSFILTER is handled in the kernel by the `ip6_mc_msfilter()` method, in `net/ipv6/mcast.c`.

An MLD router (which is also sometimes known as the "Querier") joins the all MLDv2-capable routers Multicast Group (`ff02::16`) when it is started. It periodically sends Multicast Listener Query packets in order to know which hosts belong to a Multicast group, and to which Multicast group they belong. These are ICMPv6 packets whose type is ICMPV6_MGM_QUERY. The destination address of these query packets is the all-hosts multicast group (`ff02::1`). When a host receives an ICMPv6 Multicast Listener Query packet, the ICMPv6 Rx handler (the `icmpv6_rcv()` method) calls the `igmp6_event_query()` method to handle that query. Note that the `igmp6_event_query()` method handles both MLDv2 queries and MLDv1 queries (because both use ICMPV6_MGM_QUERY as the ICMPv6 type). The `igmp6_event_query()` method finds out whether the message is MLDv1 or MLDv2 by checking its length; in MLDv1 the length is 24 bytes, and in MLDv2 it is 28 bytes at least. Handling MLDv1 and MLDv2 messages is different; for MLDv2, we should support source filtering, as was mentioned before in this section, while this feature is not available in MLDv1. The host sends back a Multicast Listener Report by calling the `igmp6_send()` method. The Multicast Listener Report packet is an ICMPv6 packet.

An example of an IPv6 MLD router is the `mld6igmp` daemon of the open source XORP project: `http://www.xorp.org`. The MLD router keeps information about the multicast address groups of network nodes (MLD listeners) and updates this information dynamically. This information can be provided to Multicast Routing daemons. Delving into the implementation of MLDv2 routing daemons like the `mld6igmp` daemon, or into the implementation of other Multicast Routing daemons, is beyond the scope of this book because it is implemented in userspace.

According to RFC 3810, MLDv2 should be interoperable with nodes that implement MLDv1; an implementation of MLDv2 must support the following two MLDv1 message types:

- MLDv1 Multicast Listener Report (ICMPV6_MGM_REPORT, decimal 131)
- MLDv1 Multicast Listener Done (ICMPV6_MGM_REDUCTION, decimal 132)

We can use the MLDv1 protocol for Multicast Listener messages instead of MLDv2; this can be done by using the following:

```
echo 1 > /proc/sys/net/ipv6/conf/all/force_mld_version
```

In such a case, when a host joins a multicast group, a Multicast Listener Report message will be sent by the `igmp6_send()` method. This message will use ICMPV6_MGM_REPORT (131) of MLDv1 as the ICMPv6 type, not ICMPV6_MLD2_REPORT(143) as in MLDv2. Note that in this case you cannot use source filtering request for this message, as MLDv1 does not support it. We will join the multicast group by calling the `igmp6_join_group()` method. When you leave the multicast group, a Multicast Listener Done message will be sent. In this message, the ICMPv6 type is ICMPV6_MGM_REDUCTION (132).

In the next section, I will very briefly talk about the IPv6 Tx path, which is quite similar to the IPv4 Tx path, and which I do not cover in depth in this chapter.

Sending IPv6 Packets

The IPv6 Tx path is very similar to the IPv4 Tx path; even the names of the methods are very similar. Also in IPv6, there are two main methods for sending IPv6 packets from Layer 4, the transport layer: the first is the `ip6_xmit()` method, which is used by the TCP, Stream Control Transmission Protocol (SCTP), and Datagram Congestion Control Protocol (DCCP) protocols. The second method is the `ip6_append_data()` method, which is used, for example, by UDP and Raw sockets. Packets that are created on the local host are sent out by the `ip6_local_out()` method. The `ip6_output()` method is set to be the output callback of the protocol-independent `dst_entry`; it first calls the NF_HOOK() macro for the NF_INET_POST_ROUTING hook, and then it calls the `ip6_finish_output()` method. If fragmentation is needed, the ip6_finish_output() method calls the ip6_fragment() method to handle it; otherwise, it calls the ip6_finish_output2() method, which eventually sends the packet. For implementation details, look in the IPv6 Tx path code; it is mostly in net/ipv6/ip6_output.c.

In the next section, I will very briefly talk about IPv6 routing, which is, again, quite similar to the IPv4 routing, and which I do not cover in depth in this chapter.

IPv6 Routing

The implementation of IPv6 routing is very similar to the IPv4 routing implementation that was discussed in Chapter 5, which dealt with the IPv4 routing subsystem. Like in the IPv4 routing subsystem, Policy routing is also supported in IPv6 (when CONFIG_IPV6_MULTIPLE_TABLES is set). A routing entry is represented in IPv6 by the rt6_info structure (include/net/ip6_fib.h). The rt6_info object parallels the IPv4 rtable structure, and the flowi6 structure (include/net/flow.h) parallels the IPv4 flowi4 structure. (In fact, they both have as their first member the same flowi_common object.) For implementation details, look in the IPv6 routing modules: net/ipv6/route.c, net/ipv6/ip6_fib.c, and the policy routing module, net/ipv6/fib6_rules.c.

Summary

I dealt with the IPv6 subsystem and its implementation in this chapter. I discussed various IPv6 topics, like IPv6 addresses (including Special Addresses and Multicast Addresses), how the IPv6 header is built, what the IPv6 extension headers are, the autoconfiguration process, the Rx path in IPv6, and the MLD protocol. In the next chapter, we will continue our journey into the kernel networking internals and discuss the netfilter subsystem and its implementation. In the "Quick Reference" section that follows, we will cover the top methods related to the topics we discussed in this chapter, ordered by their context.

Quick Reference

I conclude this chapter with a short list of important methods of the IPv6 subsystem. Some of them were mentioned in this chapter. Subsequently, there are three tables and two short sections about IPv6 Special Addresses and about the management of routing tables in IPv6.

Methods

Let's start with the methods.

bool ipv6_addr_any(const struct in6_addr *a);

This method returns true if the specified address is the all-zeroes address ("unspecified address").

bool ipv6_addr_equal(const struct in6_addr *a1, const struct in6_addr *a2);

This method returns true if the two specified IPv6 addresses are equal.

static inline void ipv6_addr_set(struct in6_addr *addr, __be32 w1, __be32 w2, __be32 w3, __be32 w4);

This method sets the IPv6 address according to the four 32-bit input parameters.

bool ipv6_addr_is_multicast(const struct in6_addr *addr);

This method returns `true` if the specified address is a multicast address.

bool ipv6_ext_hdr(u8 nexthdr);

This method returns `true` if the specified `nexthdr` is a well-known extension header.

struct ipv6hdr *ipv6_hdr(const struct sk_buff *skb);

This method returns the IPv6 header (`ipv6hdr`) of the specified `skb`.

struct inet6_dev *in6_dev_get(const struct net_device *dev);

This method returns the `inet6_dev` object associated with the specified device.

bool ipv6_is_mld(struct sk_buff *skb, int nexthdr, int offset);

This method returns `true` if the specified `nexthdr` is ICMPv6 (IPPROTO_ICMPV6) and the type of the ICMPv6 header located at the specified offset is an MLD type. It should be one of the following:

- ICMPV6_MGM_QUERY
- ICMPV6_MGM_REPORT
- ICMPV6_MGM_REDUCTION
- ICMPV6_MLD2_REPORT

bool raw6_local_deliver(struct sk_buff *, int);

This method tries to deliver the packet to a raw socket. It returns `true` on success.

int ipv6_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 IPv6 packets.

bool ipv6_accept_ra(struct inet6_dev *idev);

This method returns `true` if a host is configured to accept Router Advertisements, in these cases:

- If forwarding is enabled, the special hybrid mode should be set, which means that `/proc/sys/net/ipv6/conf/<deviceName>/accept_ra` is 2.
- If forwarding is not enabled, `/proc/sys/net/ipv6/conf/<deviceName>/accept_ra` should be 1.

void ip6_route_input(struct sk_buff *skb);

This method is the main IPv6 routing subsystem lookup method in the Rx path. It sets the `dst entry` of the specified `skb` according to the results of the lookup in the routing subsystem.

int ip6_forward(struct sk_buff *skb);

This method is the main forwarding method.

struct dst_entry *ip6_route_output(struct net *net, const struct sock *sk, struct flowi6 *fl6);

This method is the main IPv6 routing subsystem lookup method in the Tx path. The return value is the destination cache entry (`dst`).

■ **Note** Both the `ip6_route_input()` method and the `ip6_route_output()` method eventually perform the lookup by calling the `fib6_lookup()` method.

---

void in6_dev_hold(struct inet6_dev *idev); and void __in6_dev_put(struct inet6_dev *idev);

This method increments and decrements the reference counter of the specified `idev` object, respectively.

int ip6_mc_msfilter(struct sock *sk, struct group_filter *gsf);

This method handles a `setsockopt()` call with MCAST_MSFILTER.

int ip6_mc_input(struct sk_buff *skb);

This method is the main Rx handler for multicast packets.

int ip6_mr_input(struct sk_buff *skb);

This method is the main Rx handler for multicast packets that are to be forwarded.

int ipv6_dev_mc_inc(struct net_device *dev, const struct in6_addr *addr);

This method adds the specified device to a multicast group specified by `addr`, or creates such a group if not found.

int __ipv6_dev_mc_dec(struct inet6_dev *idev, const struct in6_addr *addr);

This method removes the specified device from the specified address group.

bool ipv6_chk_mcast_addr(struct net_device *dev, const struct in6_addr *group, const struct in6_addr *src_addr);

This method checks if the specified network device belongs to the specified multicast address group. If the third parameter is not NULL, it will also check whether source filtering permits receiving multicast traffic from the specified address (`src_addr`) that is destined to the specified multicast address group.

inline void addrconf_addr_solict_mult(const struct in6_addr *addr, struct in6_addr *solicited)

This method computes link-local solicited-node multicast addresses.

void addrconf_join_solict(struct net_device *dev, const struct in6_addr *addr);

This method joins to a solicited address multicast group.

int ipv6_sock_mc_join(struct sock *sk, int ifindex, const struct in6_addr *addr);

This method handles socket join on a multicast group.

int ipv6_sock_mc_drop(struct sock *sk, int ifindex, const struct in6_addr *addr);

This method handles socket leave on a multicast group.

int inet6_add_protocol(const struct inet6_protocol *prot, unsigned char protocol);

This method registers an IPv6 protocol handler. It’s used with L4 protocol registration (UDPv6, TCPv6, and more) and also with extension headers (like the Fragment Extension Header).

int ipv6_parse_hopopts(struct sk_buff *skb);

This method parses the Hop-by-Hop Options header, which must be the first extension header immediately after the IPv6 header.

int ip6_local_out(struct sk_buff *skb);

This method sends out packets that were generated on the local host.

int ip6_fragment(struct sk_buff *skb, int (*output)(struct sk_buff *));

This method handles IPv6 fragmentation. It is called from the `ip6_finish_output()` method.

void icmpv6_param_prob(struct sk_buff *skb, u8 code, int pos);

This method sends an ICMPv6 parameter problem (ICMPV6_PARAMPROB) error. It is called when there is some problem in parsing extension headers or in the defragmentation process.

int do_ipv6_setsockopt(struct sock *sk, int level, int optname, char __user *optval, unsigned int optlen); static int do_ipv6_getsockopt(struct sock *sk, int level, int optname, char __user *optval, int __user *optlen, unsigned int flags);

These methods are the generic IPv6 handlers for calling the `setsockopt()` and `getsockopt()` methods on IPv6 sockets, respectively (`net/ipv6/ipv6_sockglue.c`).

int igmp6_event_query(struct sk_buff *skb);

This method handles MLDv2 and MLDv1 queries.

void ip6_route_input(struct sk_buff *skb);

This method performs a routing lookup by building a `flow6` object, based on the specified `skb` and invoking the `ip6_route_input_lookup()` method.

Macros

And here are the macros.

IPV6_ADDR_MC_SCOPE()

This macro returns the scope of the specified IPv6 Multicast address, which is located in bits 11-14 of the multicast address.

IPV6_ADDR_MC_FLAG_TRANSIENT()

This macro returns 1 if the T bit of the flags of the specified multicast address is set.

IPV6_ADDR_MC_FLAG_PREFIX()

This macro returns 1 if the P bit of the flags of the specified multicast address is set.

IPV6_ADDR_MC_FLAG_RENDEZVOUS()

This macro returns 1 if the R bit of the flags of the specified multicast address is set.

Tables

Here are the tables.

Table 8-2 shows the IPv6 extension headers by their Linux symbol, value and description. You can find more details in the "extension headers" section of this chapter.

Table 8-2. *IPv6 extension headers*

Linux Symbol	Value	Description
NEXTHDR_HOP	0	Hop-by-Hop Options header.
NEXTHDR_TCP	6	TCP segment.
NEXTHDR_UDP	17	UDP message.
NEXTHDR_IPV6	41	IPv6 in IPv6.
NEXTHDR_ROUTING	43	Routing header.
NEXTHDR_FRAGMENT	44	Fragmentation/reassembly header.
NEXTHDR_GRE	47	GRE header.
NEXTHDR_ESP	50	Encapsulating security payload.
NEXTHDR_AUTH	51	Authentication header.
NEXTHDR_ICMP	58	ICMP for IPv6.
NEXTHDR_NONE	59	No next header.
NEXTHDR_DEST	60	Destination options header.
NEXTHDR_MOBILITY	135	Mobility header.

Table 8-3 shows the Multicast Address Record types by their Linux symbol and value. For more details see the "MLDv2 Multicast Listener Report" section in this chapter.

Table 8-3. *Multicast Address Record (record types)*

Linux Symbol	Value
MLD2_MODE_IS_INCLUDE	1
MLD2_MODE_IS_EXCLUDE	2
MLD2_CHANGE_TO_INCLUDE	3
MLD2_CHANGE_TO_EXCLUDE	4
MLD2_ALLOW_NEW_SOURCES	5
MLD2_BLOCK_OLD_SOURCES	6

(`include/uapi/linux/icmpv6.h`)

Table 8-4 shows the codes of ICMPv6 "Parameter Problem" message by their Linux symbol and value. These codes gives more information about the type of problem which occurred.

Table 8-4. *ICMPv6 Parameter Problem codes*

Linux Symbol	Value
ICMPV6_HDR_FIELD	0 Erroneous header field encountered
ICMPV6_UNK_NEXTHDR	1 Unknown header field encountered
ICMPV6_UNK_OPTION	2 Unknown IPv6 option encountered

Special Addresses

All of the following variables are instances of the `in6_addr` structure:

- `in6addr_any`: Represents the unspecified device of all zeroes (`::`).
- `in6addr_loopback`: Represents the loopback device (`::1`).
- `in6addr_linklocal_allnodes`: Represents the link-local all nodes multicast address (`ff02::1`).
- `in6addr_linklocal_allrouters`: Represents the link-local all routers multicast address (`ff02::2`).
- `in6addr_interfacelocal_allnodes`: Represents the interface-local all nodes (`ff01::1`).
- `in6addr_interfacelocal_allrouters`: Represents the interface-local all routers (`ff01::2`).
- `in6addr_sitelocal_allrouters`: Represents the site-local all routers address (`ff05::2`).

`(include/linux/in6.h)`

Routing Tables Management in IPv6

Like in IPv4, we can manage adding and deleting routing entries and displaying the routing tables with the `ip route` command of `iproute2` and with the `route` command of `net-tools`:

- Adding a route by `ip -6 route add` is handled by the `inet6_rtm_newroute()` method by invoking the `ip6_route_add()` method.
- Deleting a route by `ip -6 route del` is handled by the `inet6_rtm_delroute()` method by invoking the `ip6_route_del()` method.
- Displaying the routing table by `ip -6 route show` is handled by the `inet6_dump_fib()` method.
- Adding a route by `route -A inet6 add` is implemented by sending SIOCADDRT IOCTL, which is handled by the `ipv6_route_ioctl()` method, by invoking the `ip6_route_add()` method.
- Deleting a route by `route -A inet6 del` is implemented by sending SIOCDELRT IOCTL, which is handled by the `ipv6_route_ioctl()` method by invoking the `ip6_route_del()` method.

CHAPTER 9

Netfilter

Chapter 8 discusses the IPv6 subsystem implementation. This chapter discusses the netfilter subsystem. The netfilter framework was started in 1998 by Rusty Russell, one of the most widely known Linux kernel developers, as an improvement of the older implementations of `ipchains` (Linux 2.2.x) and `ipfwadm` (Linux 2.0.x). The netfilter subsystem provides a framework that enables registering callbacks in various points (netfilter hooks) in the packet traversal in the network stack and performing various operations on packets, such as changing addresses or ports, dropping packets, logging, and more. These netfilter hooks provide the infrastructure to netfilter kernel modules that register callbacks in order to perform various tasks of the netfilter subsystem.

Netfilter Frameworks

The netfilter subsystem provides the following functionalities, discussed in this chapter:

- Packet selection (iptables)
- Packet filtering
- Network Address Translation (NAT)
- Packet mangling (modifying the contents of packet headers before or after routing)
- Connection tracking
- Gathering network statistics

Here are some common frameworks that are based on the Linux kernel netfilter subsystem:

- **IPVS (IP Virtual Server):** A transport layer load-balancing solution (`net/netfilter/ipvs`). There is support for IPv4 IPVS from very early kernels, and support for IPVS in IPv6 is included since kernel 2.6.28. The IPv6 kernel support for IPVS was developed by Julius Volz and Vince Busam from Google. For more details, see the IPVS official website, `www.linuxvirtualserver.org`.
- **IP sets:** A framework which consists of a userspace tool called `ipset` and a kernel part (`net/netfilter/ipset`). An IP set is basically a set of IP addresses. The IP sets framework was developed by Jozsef Kadlecsik. For more details, see `http://ipset.netfilter.org`.
- **iptables:** Probably the most popular Linux firewall, **iptables** is the front end of netfilter, and it provides a management layer for netfilter: for example, adding and deleting netfilter rules, displaying statistics, adding a table, zeroing the counters of a table, and more.

There are different iptables implementations in the kernel, according to the protocol:

- **iptables** for IPv4: (net/ipv4/netfilter/ip_tables.c)
- **ip6tables** for IPv6: (net/ipv6/netfilter/ip6_tables.c)
- **arptables** for ARP: (net/ipv4/netfilter/arp_tables.c)
- **ebtables** for Ethernet: (net/bridge/netfilter/ebtables.c)

In userspace, you have the iptables and the ip6tables command-line tools, which are used to set up, maintain, and inspect the IPv4 and IPv6 tables, respectively. See man 8 iptables and man 8 ip6tables. Both iptables and ip6tables use the setsockopt()/getsockopt() system calls to communicate with the kernel from userspace.
I should mention here two interesting ongoing netfilter projects. The xtables2 project—being developed primarily by Jan Engelhardt, a work in progress as of this writing—uses a netlink-based interface to communicate with the kernel netfilter subsystem. See more details on the project website, http://xtables.de. The second project, the nftables project, is a new packet filtering engine that is a candidate to replace iptables. The nftables solution is based on using a virtual machine and a single unified implementation instead of the four iptables objects mentioned earlier (iptables, ip6tables, arptables, and ebtables). The nftables project was first presented in a netfilter workshop in 2008, by Patrick McHardy. The kernel infrastructure and userspace utility have been developed by Patrick McHardy and Pablo Neira Ayuso. For more details, see http://netfilter.org/projects/nftables, and “Nftables: a new packet filtering engine” at http://lwn.net/Articles/324989/.

There are a lot of netfilter modules that extend the core functionality of the core netfilter subsystem; apart from some examples, I do not describe these modules here in depth. There are a lot of information resources about these netfilter extensions from the administration perspective on the web and in various administration guides. See also the official netfilter project website: www.netfilter.org.

Netfilter Hooks

There are five points in the network stack where you have netfilter hooks: you have encountered these points in previous chapters’ discussions of the Rx and Tx paths in IPv4 and in IPv6. Note that the names of the hooks are common to IPv4 and IPv6:

- NF_INET_PRE_ROUTING: This hook is in the ip_rcv() method in IPv4, and in the ipv6_rcv() method in IPv6. The ip_rcv() method is the protocol handler of IPv4, and the ipv6_rcv() method is the protocol handler of IPv6. It is the first hook point that all incoming packets reach, before performing a lookup in the routing subsystem.
- NF_INET_LOCAL_IN: This hook is in the ip_local_deliver() method in IPv4, and in the ip6_input() method in IPv6. All incoming packets addressed to the local host reach this hook point after first passing via the NF_INET_PRE_ROUTING hook point and after performing a lookup in the routing subsystem.
- NF_INET_FORWARD: This hook is in the ip_forward() method in IPv4, and in the ip6_forward() method in IPv6. All forwarded packets reach this hook point after first passing via the NF_INET_PRE_ROUTING hook point and after performing a lookup in the routing subsystem.
- NF_INET_POST_ROUTING: This hook is in the ip_output() method in IPv4, and in the ip6_finish_output2() method in IPv6. Packets that are forwarded reach this hook point after passing the NF_INET_FORWARD hook point. Also packets that are created in the local machine and sent out arrive to NF_INET_POST_ROUTING after passing the NF_INET_LOCAL_OUT hook point.

- NF_INET_LOCAL_OUT: This hook is in the `__ip_local_out()` method in IPv4, and in the `__ip6_local_out()` method in IPv6. All outgoing packets that were created on the local host reach this point before reaching the NF_INET_POST_ROUTING hook point.

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

The NF_HOOK macro, mentioned in previous chapters, is called in some distinct points along the packet traversal in the kernel network stack; it is defined in `include/linux/netfilter.h`:

```
static inline 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 *))
{
    return NF_HOOK_THRESH(pf, hook, skb, in, out, okfn, INT_MIN);
}
```

The parameters of the NF_HOOK() are as follows:

- `pf`: Protocol family. NFPROTO_IPV4 for IPv4 and NFPROTO_IPV6 for IPv6.
- `hook`: One of the five netfilter hooks mentioned earlier (for example, NF_INET_PRE_ROUTING or NF_INET_LOCAL_OUT).
- `skb`: The SKB object represents the packet that is being processed.
- `in`: The input network device (`net_device` object).
- `out`: The output network device (`net_device` object). There are cases when the output device is NULL, as it is yet unknown; for example, in the `ip_rcv()` method, `net/ipv4/ip_input.c`, which is called before a routing lookup is performed, and you don't know yet which is the output device; the NF_HOOK() macro is invoked in this method with a NULL output device.
- `okfn:` A pointer to a continuation function which will be called when the hook will terminate. It gets one argument, the SKB.

The return value from a netfilter hook must be one of the following values (which are also termed *netfilter verdicts*):

- NF_DROP (0): Discard the packet silently.
- NF_ACCEPT (1): The packet continues its traversal in the kernel network stack as usual.
- NF_STOLEN (2): Do not continue traversal. The packet is processed by the hook method.
- NF_QUEUE (3): Queue the packet for user space.
- NF_REPEAT (4): The hook function should be called again.

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

Now that you know about the various netfilter hooks, the next section covers how netfilter hooks are registered.

Registration of Netfilter Hooks

To register a hook callback at one of the five hook points mentioned earlier, you first define an `nf_hook_ops` object (or an array of `nf_hook_ops` objects) and then register it; the `nf_hook_ops` structure is defined in `include/linux/netfilter.h`:

```
struct nf_hook_ops {
    struct list_head list;
    /* User fills in from here down. */
    nf_hookfn     *hook;
    struct module *owner;
    u_int8_t      pf;
    unsigned int  hooknum;
    /* Hooks are ordered in ascending priority. */
    int           priority;
};
```

The following introduces some of the important members of the `nf_hook_ops` structure:

- `hook`: The hook callback you want to register. Its prototype is:

```
unsigned int nf_hookfn(unsigned int hooknum,
                       struct sk_buff *skb,
                       const struct net_device *in,
                       const struct net_device *out,
                       int (*okfn)(struct sk_buff *));
```

- `pf`: The protocol family (NFPROTO_IPV4 for IPv4 and NFPROTO_IPV6 for IPv6).
- `hooknum`: One of the five netfilter hooks mentioned earlier.
- `priority`: More than one hook callback can be registered on the same hook. Hook callbacks with lower priorities are called first. The `nf_ip_hook_priorities enum` defines possible values for IPv4 hook priorities (`include/uapi/linux/netfilter_ipv4.h`). See also Table 9-4 in the "Quick Reference" section at the end of this chapter.

There are two methods to register netfilter hooks:

- `int nf_register_hook(struct nf_hook_ops *reg)`: Registers a single `nf_hook_ops` object.
- `int nf_register_hooks(struct nf_hook_ops *reg, unsigned int n)`: Registers an array of *n* `nf_hook_ops` objects; the second parameter is the number of the elements in the array.

You will see two examples of registration of an array of `nf_hook_ops` objects in the next two sections. Figure 9-1 in the next section illustrates the use of priorities when registering more than one hook callback on the same hook point.

Connection Tracking

It is not enough to filter traffic only according to the L4 and L3 headers in modern networks. You should also take into account cases when the traffic is based on sessions, such as an FTP session or a SIP session. By FTP session, I mean this sequence of events, for example: the client first creates a TCP control connection on TCP port 21, which is the default FTP port. Commands sent from the FTP client (such as listing the contents of a directory) to the server are sent on this control port. The FTP server opens a data socket on port 20, where the destination port on the client side is dynamically allocated. Traffic should be filtered according to other parameters, such as the state of a connection or timeout. This is one of the main reasons for using the Connection Tracking layer.

Connection Tracking allows the kernel to keep track of sessions. The Connection Tracking layer's primary goal is to serve as the basis of NAT. The IPv4 NAT module (`net/ipv4/netfilter/iptable_nat.c`) cannot be built if CONFIG_NF_CONNTRACK_IPV4 is not set. Similarly, the IPv6 NAT module (`net/ipv6/netfilter/ip6table_nat.c`) cannot be built if the CONFIG_NF_CONNTRACK_IPV6 is not set. However, Connection Tracking does not depend on NAT; you can run the Connection Tracking module without activating any NAT rule. The IPv4 and IPv6 NAT modules are discussed later in this chapter.

■ **Note** There are some userspace tools (`conntrack-tools`) for Connection Tracking administration mentioned in the "Quick Reference" section at the end of this chapter. These tools may help you to better understand the Connection Tracking layer.

Connection Tracking Initialization

An array of `nf_hook_ops` objects, called `ipv4_conntrack_ops`, is defined as follows:

```
static struct nf_hook_ops ipv4_conntrack_ops[] __read_mostly = {
        {
                .hook           = ipv4_conntrack_in,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_PRE_ROUTING,
                .priority       = NF_IP_PRI_CONNTRACK,
        },
        {
                .hook           = ipv4_conntrack_local,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_LOCAL_OUT,
                .priority       = NF_IP_PRI_CONNTRACK,
        },
        {
                .hook           = ipv4_helper,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_POST_ROUTING,
                .priority       = NF_IP_PRI_CONNTRACK_HELPER,
        },
        {
                .hook           = ipv4_confirm,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_POST_ROUTING,
                .priority       = NF_IP_PRI_CONNTRACK_CONFIRM,
        },
        {
                .hook           = ipv4_helper,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_LOCAL_IN,
                .priority       = NF_IP_PRI_CONNTRACK_HELPER,
        },
        {
                .hook           = ipv4_confirm,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_LOCAL_IN,
                .priority       = NF_IP_PRI_CONNTRACK_CONFIRM,
        },
};
```

(net/ipv4/netfilter/nf_conntrack_l3proto_ipv4.c)

The two most important Connection Tracking hooks you register are the NF_INET_PRE_ROUTING hook, handled by the `ipv4_conntrack_in()` method, and the NF_INET_LOCAL_OUT hook, handled by the `ipv4_conntrack_local()` method. These two hooks have a priority of NF_IP_PRI_CONNTRACK (-200). The other hooks in the `ipv4_conntrack_ops` array have an NF_IP_PRI_CONNTRACK_HELPER (300) priority and an NF_IP_PRI_CONNTRACK_CONFIRM (INT_MAX, which is 2^31-1) priority. In netfilter hooks, a callback with a lower-priority value is executed first. (The `enum nf_ip_hook_priorities` in `include/uapi/linux/netfilter_ipv4.h` represents the possible priority values for IPv4 hooks). Both the `ipv4_conntrack_local()` method and the `ipv4_conntrack_in()` method invoke the `nf_conntrack_in()` method, passing the corresponding `hooknum` as a parameter. The `nf_conntrack_in()` method belongs to the protocol-independent NAT core, and is used both in IPv4 Connection Tracking and in IPv6 Connection Tracking; its second parameter is the protocol family, specifying whether it is IPv4 (PF_INET) or IPv6 (PF_INET6). I start the discussion with the `nf_conntrack_in()` callback. The other hook callbacks, `ipv4_confirm()` and `ipv4_help()`, are discussed later in this section.

■ **Note** When the kernel is built with Connection Tracking support (CONFIG_NF_CONNTRACK is set), the Connection Tracking hook callbacks are called even if there are no iptables rules that are activated. Naturally, this has some performance cost. If the performance is very important, and you know beforehand that the device will not use the netfilter subsystem, consider building the kernel without Connection Tracking support or building Connection Tracking as a kernel module and not loading it.

Registration of IPv4 Connection Tracking hooks is done by calling the `nf_register_hooks()` method in the `nf_conntrack_l3proto_ipv4_init()` method (`net/ipv4/netfilter/nf_conntrack_l3proto_ipv4.c`):

```
in nf_conntrack_l3proto_ipv4_init(void) {
       . . .
       ret = nf_register_hooks(ipv4_conntrack_ops,
                               ARRAY_SIZE(ipv4_conntrack_ops))
       . . .
}
```

In Figure 9-1, you can see the Connection Tracking callbacks (`ipv4_conntrack_in()`, `ipv4_conntrack_local()`, `ipv4_helper()` and `ipv4_confirm()`), according to the hook points where they are registered.

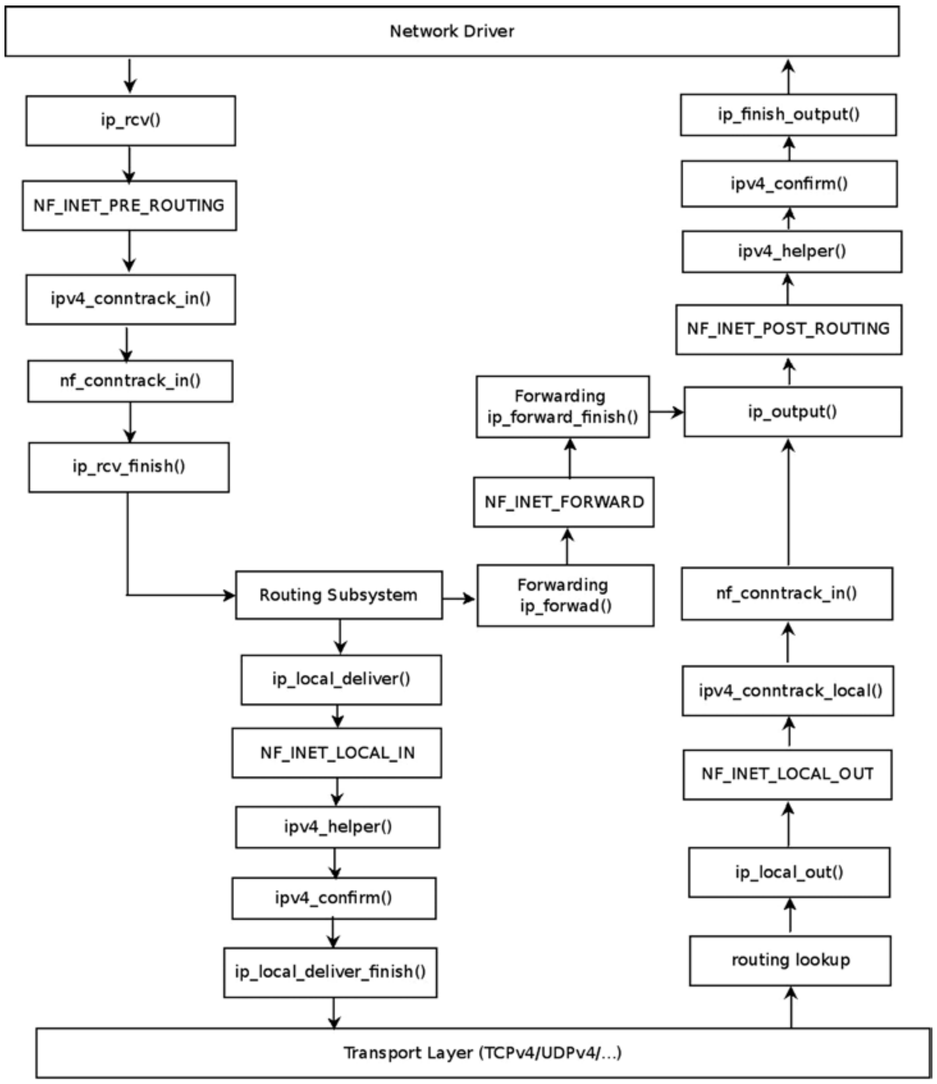


Figure 9-1. *Connection Tracking hooks (IPv4)*

■ **Note** For the sake of simplicity, Figure 9-1 does not include more complex scenarios, such as when using IPsec or fragmentation or multicasting. It also omits the functions that are called for packets generated on the local host and sent out (like the `ip_queue_xmit()` method or the `ip_build_and_send_pkt()` method) for the sake of simplicity.

The basic element of Connection Tracking is the nf_conntrack_tuple structure:

```
struct nf_conntrack_tuple {
        struct nf_conntrack_man src;

        /* These are the parts of the tuple which are fixed. */
        struct {
                union nf_inet_addr u3;
                union {
                        /* Add other protocols here. */
                        __be16 all;

                        struct {
                                __be16 port;
                        } tcp;
                        struct {
                                __be16 port;
                        } udp;
                        struct {
                                u_int8_t type, code;
                        } icmp;
                        struct {
                                __be16 port;
                        } dccp;
                        struct {
                                __be16 port;
                        } sctp;
                        struct {
                                __be16 key;
                        } gre;
                } u;

                /* The protocol. */
                u_int8_t protonum;

                /* The direction (for tuplehash) */
                u_int8_t dir;
        } dst;
};
```

(include/net/netfilter/nf_conntrack_tuple.h)

The nf_conntrack_tuple structure represents a flow in one direction. The union inside the dst structure includes various protocol objects (like TCP, UDP, ICMP, and more). For each transport layer (L4) protocol, there is a Connection Tracking module, which implements the protocol-specific part. Thus, for example, you have net/netfilter/nf_conntrack_proto_tcp.c for the TCP protocol, net/netfilter/nf_conntrack_proto_udp.c for the UDP protocol, net/netfilter/nf_conntrack_ftp.c for the FTP protocol, and more; these modules support both IPv4 and IPv6. You will see examples of how protocol-specific implementations of Connection Tracking modules differ later in this section.

Connection Tracking Entries

The nf_conn structure represents the Connection Tracking entry:

```
struct nf_conn {
        /* Usage count in here is 1 for hash table/destruct timer, 1 per skb,
           plus 1 for any connection(s) we are `master' for */
        struct nf_conntrack ct_general;

        spinlock_t lock;

        /* XXX should I move this to the tail ? - Y.K */
        /* These are my tuples; original and reply */
        struct nf_conntrack_tuple_hash tuplehash[IP_CT_DIR_MAX];

        /* Have we seen traffic both ways yet? (bitset) */
        unsigned long status;

        /* If we were expected by an expectation, this will be it */
        struct nf_conn *master;

        /* Timer function; drops refcnt when it goes off. */
        struct timer_list timeout;

      . . .

        /* Extensions */
        struct nf_ct_ext *ext;
#ifdef CONFIG_NET_NS
        struct net *ct_net;
#endif

        /* Storage reserved for other modules, must be the last member */
        union nf_conntrack_proto proto;
};
```

(include/net/netfilter/nf_conntrack.h)

The following is a description of some of the important members of the nf_conn structure :

- ct_general: A reference count.
- tuplehash: There are two tuplehash objects: tuplehash[0] is the original direction, and tuplehash[1] is the reply. They are usually referred to as tuplehash[IP_CT_DIR_ORIGINAL] and tuplehash[IP_CT_DIR_REPLY], respectively.
- status: The status of the entry. When you start to track a connection entry, it is IP_CT_NEW; later on, when the connection is established, it becomes IP_CT_ESTABLISHED. See the ip_conntrack_info enum in include/uapi/linux/netfilter/nf_conntrack_common.h.

- master: An expected connection. Set by the init_conntrack() method, when an expected packet arrives (this means that the nf_ct_find_expectation() method, which is invoked by the init_conntrack() method, finds an expectation). See also the “Connection Tracking Helpers and Expectations” section later in this chapter.
- timeout: Timer of the connection entry. Each connection entry is expired after some time interval when there is no traffic. The time interval is determined according to the protocol. When allocating an nf_conn object with the __nf_conntrack_alloc() method, the timeout timer is set to be the death_by_timeout() method.

Now that you know about the nf_conn struct and some of its members, let’s take a look at the nf_conntrack_in() method:

```
unsigned int nf_conntrack_in(struct net *net, u_int8_t pf, unsigned int hooknum,
                          struct sk_buff *skb)
{
        struct nf_conn *ct, *tmpl = NULL;
        enum ip_conntrack_info ctinfo;
        struct nf_conntrack_l3proto *l3proto;
        struct nf_conntrack_l4proto *l4proto;
        unsigned int *timeouts;
        unsigned int dataoff;
        u_int8_t protonum;
        int set_reply = 0;
        int ret;

        if (skb->nfct) {
                /* Previously seen (loopback or untracked)?  Ignore. */
                tmpl = (struct nf_conn *)skb->nfct;
                if (!nf_ct_is_template(tmpl)) {
                        NF_CT_STAT_INC_ATOMIC(net, ignore);
                        return NF_ACCEPT;
                }
                skb->nfct = NULL;
        }
```

First you try to find whether the network layer (L3) protocol can be tracked:

```
        l3proto = __nf_ct_l3proto_find(pf);
```

Now you try to find if the transport layer (L4) protocol can be tracked. For IPv4, it is done by the ipv4_get_l4proto() method (net/ipv4/netfilter/nf_conntrack_l3proto_ipv4):

```
        ret = l3proto->get_l4proto(skb, skb_network_offset(skb),
                                   &dataoff, &protonum);
        if (ret <= 0) {
           . . .
                ret = -ret;
                goto out;
        }
```

```
l4proto = __nf_ct_l4proto_find(pf, protonum);

/* It may be an special packet, error, unclean...
 * inverse of the return code tells to the netfilter
 * core what to do with the packet. */
```

Now you check protocol-specific error conditions (see, for example, the `udp_error()` method in `net/netfilter/nf_conntrack_proto_udp.c`, which checks for malformed packets, packets with invalid checksum, and more, or the `tcp_error()` method, in `net/netfilter/nf_conntrack_proto_tcp.c`):

```
if (l4proto->error != NULL) {
        ret = l4proto->error(net, tmpl, skb, dataoff, &ctinfo,
                             pf, hooknum);
        if (ret <= 0) {
                NF_CT_STAT_INC_ATOMIC(net, error);
                NF_CT_STAT_INC_ATOMIC(net, invalid);
                ret = -ret;
                goto out;
        }
        /* ICMP[v6] protocol trackers may assign one conntrack. */
        if (skb->nfct)
                goto out;
}
```

The `resolve_normal_ct()` method, which is invoked hereafter immediately, performs the following:

- Calculates the hash of the tuple by calling the `hash_conntrack_raw()` method.
- Performs a lookup for a tuple match by calling the `__nf_conntrack_find_get()` method, passing the hash as a parameter.
- If no match is found, it creates a new `nf_conntrack_tuple_hash` object by calling the `init_conntrack()` method. This `nf_conntrack_tuple_hash` object is added to the list of unconfirmed tuplehash objects. This list is embedded in the network namespace object; the `net` structure contains a `netns_ct` object, which consists of network namespace specific Connection Tracking information. One of its members is `unconfirmed`, which is a list of unconfirmed `tuplehash` objects (see `include/net/netns/conntrack.h`). Later on, in the `__nf_conntrack_confirm()` method, it will be removed from the unconfirmed list. I discuss the `__nf_conntrack_confirm()` method later in this section.
- Each SKB has a member called `nfctinfo`, which represents the connection state (for example, it is IP_CT_NEW for new connections), and also a member called `nfct` (an instance of the `nf_conntrack struct`) which is in fact a reference counter. The `resolve_normal_ct()` method initializes both of them.

  ```
  ct = resolve_normal_ct(net, tmpl, skb, dataoff, pf, protonum,
                         l3proto, l4proto, &set_reply, &ctinfo);
  if (!ct) {
          /* Not valid part of a connection */
          NF_CT_STAT_INC_ATOMIC(net, invalid);
          ret = NF_ACCEPT;
          goto out;
  }
  ```

```
        if (IS_ERR(ct)) {
                /* Too stressed to deal. */
                NF_CT_STAT_INC_ATOMIC(net, drop);
                ret = NF_DROP;
                goto out;
        }

        NF_CT_ASSERT(skb->nfct);
```

You now call the nf_ct_timeout_lookup() method to decide what timeout policy you want to apply to this flow. For example, for UDP, the timeout is 30 seconds for unidirectional connections and 180 seconds for bidirectional connections; see the definition of the udp_timeouts array in net/netfilter/nf_conntrack_proto_udp.c. For TCP, which is a much more complex protocol, there are 11 entries in tcp_timeouts array (net/netfilter/nf_conntrack_proto_tcp.c):

```
        /* Decide what timeout policy we want to apply to this flow. */
        timeouts = nf_ct_timeout_lookup(net, ct, l4proto);
```

You now call the protocol-specific packet() method (for example, the udp_packet() for UDP or the tcp_packet() method for TCP). The udp_packet() method extends the timeout according to the status of the connection by calling the nf_ct_refresh_acct() method. For unreplied connections (where the IPS_SEEN_REPLY_BIT flag is not set), it will be set to 30 seconds, and for replied connections, it will be set to 180. Again, in the case of TCP, the tcp_packet() method is much more complex, due to the TCP advanced state machine. Moreover, the udp_packet() method always returns a verdict of NF_ACCEPT, whereas the tcp_packet() method may sometimes fail:

```
        ret = l4proto->packet(ct, skb, dataoff, ctinfo, pf, hooknum, timeouts);
        if (ret <= 0) {
                 /* Invalid: inverse of the return code tells
                  * the netfilter core what to do */
                 pr_debug("nf_conntrack_in: Can't track with proto module\n");
                 nf_conntrack_put(skb->nfct);
                 skb->nfct = NULL;
                 NF_CT_STAT_INC_ATOMIC(net, invalid);
                 if (ret == -NF_DROP)
                         NF_CT_STAT_INC_ATOMIC(net, drop);
                 ret = -ret;
                 goto out;
        }

        if (set_reply && !test_and_set_bit(IPS_SEEN_REPLY_BIT, &ct->status))
                 nf_conntrack_event_cache(IPCT_REPLY, ct);
        out:
        if (tmpl) {
                 /* Special case: we have to repeat this hook, assign the
                  * template again to this packet. We assume that this packet
                  * has no conntrack assigned. This is used by nf_ct_tcp. */
                 if (ret == NF_REPEAT)
                         skb->nfct = (struct nf_conntrack *)tmpl;
                 else
                         nf_ct_put(tmpl);
        }

        return ret;
}
```

The `ipv4_confirm()` method, which is called in the NF_INET_POST_ROUTING hook and in the NF_INET_LOCAL_IN hook, will normally call the `__nf_conntrack_confirm()` method, which will remove the tuple from the unconfirmed list.

Connection Tracking Helpers and Expectations

Some protocols have different flows for data and for control—for example, FTP, the File Transfer Protocol, and SIP, the Session Initiation Protocol, which is a VoIP protocol. Usually in these protocols, the control channel negotiates some configuration setup with the other side and agrees with it on which parameters to use for the data flow. These protocols are more difficult to handle by the netfilter subsystem, because the netfilter subsystem needs to be aware that flows are related to each other. In order to support these types of protocols, the netfilter subsystem provides the Connection Tracking Helpers, which extend the Connection Tracking basic functionality. These modules create expectations (`nf_conntrack_expect` objects), and these expectations tell the kernel that it should expect some traffic on a specified connection and that two connections are related. Knowing that two connections are related lets you define rules on the master connection that pertain also to the related connections. You can use a simple iptables rule based on the Connection Tracking state to accept packets whose Connection Tracking state is RELATED:

```
iptables -A INPUT -m conntrack --ctstate RELATED -j ACCEPT
```

■ **Note** Connections can be related not only as a result of expectation. For example, an ICMPv4 error packet such as "ICMP fragmentation needed" will be related if netfilter finds a `conntrack` entry that matches the tuple in the ICMP-embedded L3/L4 header. See the `icmp_error_message()` method for more details, `net/ipv4/netfilter/nf_conntrack_proto_icmp.c`.

The Connection Tracking Helpers are represented by the `nf_conntrack_helper` structure (`include/net/netfilter/nf_conntrack_helper.h`). They are registered and unregistered by the `nf_conntrack_helper_register()` method and the `nf_conntrack_helper_unregister()` method, respectively. Thus, for example, the `nf_conntrack_helper_register()` method is invoked by `nf_conntrack_ftp_init()` (`net/netfilter/nf_conntrack_ftp.c`) in order to register the FTP Connection Tracking Helpers. The Connection Tracking Helpers are kept in a hash table (`nf_ct_helper_hash`). The `ipv4_helper()` hook callback is registered in two hook points, NF_INET_POST_ROUTING and NF_INET_LOCAL_IN (see the definition of `ipv4_conntrack_ops` array in the "Connection Tracking Initialization" section earlier). Because of this, when the FTP packet reaches the NF_INET_POST_ROUTING callback, `ip_output()`, or the NF_INET_LOCAL_IN callback, `ip_local_deliver()`, the `ipv4_helper()` method is invoked, and this method eventually calls the callbacks of the registered Connection Tracking Helpers. In the case of FTP, the registered helper method is the `help()` method, `net/netfilter/nf_conntrack_ftp.c`. This method looks for FTP-specific patterns, like the "PORT" FTP command; see the invocation of the `find_pattern()` method in the `help()` method, in the following code snippet (`net/netfilter/nf_conntrack_ftp.c`). If there is a match, an `nf_conntrack_expect` object is created by calling the `nf_ct_expect_init()` method:

```
static int help(struct sk_buff *skb,
         unsigned int protoff,
         struct nf_conn *ct,
         enum ip_conntrack_info ctinfo)
{
       struct nf_conntrack_expect *exp;
    . . .
```

```
        for (i = 0; i < ARRAY_SIZE(search[dir]); i++) {
                found = find_pattern(fb_ptr, datalen,
                                search[dir][i].pattern,
                                search[dir][i].plen,
                                search[dir][i].skip,
                                search[dir][i].term,
                                &matchoff, &matchlen,
                                &cmd,
                                search[dir][i].getnum);
                if (found) break;
        }

        if (found == -1) {
               /* We don't usually drop packets.  After all, this is
                  connection tracking, not packet filtering.
                  However, it is necessary for accurate tracking in
                  this case. */
               nf_ct_helper_log(skb, ct, "partial matching of `%s'",
                                search[dir][i].pattern);
```

■ **Note** Normally, Connection Tracking does not drop packets. There are some cases when, due to some error or abnormal situation, packets are dropped. The following is an example of such a case: the invocation of `find_pattern()` earlier returned –1, which means that there is only a partial match; and the packet is dropped due to not finding a full pattern match.

```
                ret = NF_DROP;
                goto out;
        } else if (found == 0) { /* No match */
                ret = NF_ACCEPT;
                goto out_update_nl;
        }

        pr_debug("conntrack_ftp: match `%.*s' (%u bytes at %u)\n",
                 matchlen, fb_ptr + matchoff,
                 matchlen, ntohl(th->seq) + matchoff);

        exp = nf_ct_expect_alloc(ct);
    . . .
        nf_ct_expect_init(exp, NF_CT_EXPECT_CLASS_DEFAULT, cmd.l3num,
                          &ct->tuplehash[!dir].tuple.src.u3, daddr,
                          IPPROTO_TCP, NULL, &cmd.u.tcp.port);
    . . .
}
```

(net/netfilter/nf_conntrack_ftp.c)

Later on, when a new connection is created by the `init_conntrack()` method, you check whether it has expectations, and if it does, you set the IPS_EXPECTED_BIT flag and set the master of the connection (`ct->master`) to refer to the connection that created the expectation:

```
static struct nf_conntrack_tuple_hash *
init_conntrack(struct net *net, struct nf_conn *tmpl,
               const struct nf_conntrack_tuple *tuple,
               struct nf_conntrack_l3proto *l3proto,
               struct nf_conntrack_l4proto *l4proto,
               struct sk_buff *skb,
               unsigned int dataoff, u32 hash)
{
        struct nf_conn *ct;
        struct nf_conn_help *help;
        struct nf_conntrack_tuple repl_tuple;
        struct nf_conntrack_ecache *ecache;
        struct nf_conntrack_expect *exp;
        u16 zone = tmpl ? nf_ct_zone(tmpl) : NF_CT_DEFAULT_ZONE;
        struct nf_conn_timeout *timeout_ext;
        unsigned int *timeouts;

        . . .
        ct = __nf_conntrack_alloc(net, zone, tuple, &repl_tuple, GFP_ATOMIC,
                                  hash);
    . . .

        exp = nf_ct_find_expectation(net, zone, tuple);
        if (exp) {
                pr_debug("conntrack: expectation arrives ct=%p exp=%p\n",
                         ct, exp);
                /* Welcome, Mr. Bond.  We've been expecting you... */
                __set_bit(IPS_EXPECTED_BIT, &ct->status);
                ct->master = exp->master;
                if (exp->helper) {
                        help = nf_ct_helper_ext_add(ct, exp->helper,
                                                    GFP_ATOMIC);
                        if (help)
                                rcu_assign_pointer(help->helper, exp->helper);
                }
        . . .
```

Note that helpers listen on a predefined port. For example, the FTP Connection Tracking Helper listens on port 21 (see FTP_PORT definition in `include/linux/netfilter/nf_conntrack_ftp.h`). You can set a different port (or ports) in one of two ways: the first way is by a module parameter—you can override the default port value by supplying a single port or a comma-separated list of ports to the `modprobe` command:

```
modprobe nf_conntrack_ftp ports=2121
modprobe nf_conntrack_ftp ports=2022,2023,2024
```

The second way is by using the CT target:

```
iptables -A PREROUTING -t raw -p tcp --dport 8888 -j CT --helper ftp
```

Note that the CT target (net/netfilter/xt_CT.c) was added in kernel 2.6.34.

■ **Note** Xtables target extensions are represented by the xt_target structure and are registered by the xt_register_target() method for a single target, or by the xt_register_targets() method for an array of targets. Xtables match extensions are represented by the xt_match structure and are registered by the xt_register_match() method, or by the xt_register_matches() for an array of matches. The match extensions inspect a packet according to some criterion defined by the match extension module; thus, for example, the xt_length match module (net/netfilter/xt_length.c) inspects packets according to their length (the tot_len of the SKB in case of IPv4 packet), and the xt_connlimit module (net/netfilter/xt_connlimit.c) limits the number of parallel TCP connections per IP address.

This section detailed the Connection Tracking initialization. The next section deals with iptables, which is probably the most known part of the netfilter framework.

IPTables

There are two parts to iptables. The kernel part—the core is in net/ipv4/netfilter/ip_tables.c for IPv4, and in net/ipv6/netfilter/ip6_tables.c for IPv6. And there is the userspace part, which provides a front end for accessing the kernel iptables layer (for example, adding and deleting rules with the iptables command). Each table is represented by the xt_table structure (defined in include/linux/netfilter/x_tables.h). Registration and unregistration of a table is done by the ipt_register_table() and the ipt_unregister_table() methods, respectively. These methods are implemented in net/ipv4/netfilter/ip_tables.c. In IPv6, you also use the xt_table structure for creating tables, but registration and unregistration of a table is done by the ip6t_register_table() method and the ip6t_unregister_table() method, respectively.

The network namespace object contains IPv4- and IPv6-specific objects (netns_ipv4 and netns_ipv6, respectively). The netns_ipv4 and netns_ipv6 objects, in turn, contain pointers to xt_table objects. For IPv4, in struct netns_ipv4 you have, for example, iptable_filter, iptable_mangle, nat_table, and more (include/net/netns/ipv4.h). In struct netns_ipv6 you have, for example, ip6table_filter, ip6table_mangle, ip6table_nat, and more (include/net/netns/ipv6.h). For a full list of the IPv4 and of the IPv6 network namespace netfilter tables and the corresponding kernel modules, see Tables 9-2 and 9-3 in the "Quick Reference" section at the end of this chapter.

To understand how iptables work, let's take a look at a real example with the filter table. For the sake of simplicity, let's assume that the filter table is the only one that is built, and also that the LOG target is supported; the only rule I am using is for logging, as you will shortly see. First, let's take a look at the definition of the filter table:

```
#define FILTER_VALID_HOOKS ((1 << NF_INET_LOCAL_IN) | \
                            (1 << NF_INET_FORWARD) | \
                            (1 << NF_INET_LOCAL_OUT))

static const struct xt_table packet_filter = {
        .name           = "filter",
        .valid_hooks    = FILTER_VALID_HOOKS,
        .me             = THIS_MODULE,
        .af             = NFPROTO_IPV4,
        .priority       = NF_IP_PRI_FILTER,
};
```

(net/ipv4/netfilter/iptable_filter.c)

Initialization of the table is done first by calling the `xt_hook_link()` method, which sets the `iptable_filter_hook()` method as the hook callback of the `nf_hook_ops` object of the `packet_filter` table:

```
static struct nf_hook_ops *filter_ops __read_mostly;
static int __init iptable_filter_init(void)
{
     . . .
        filter_ops = xt_hook_link(&packet_filter, iptable_filter_hook);
     . . .
}
```

Then you call the `ipt_register_table()` method (note that the IPv4 `netns` object, `net->ipv4`, keeps a pointer to the filter table, `iptable_filter`):

```
static int __net_init iptable_filter_net_init(struct net *net)
{
    . . .
       net->ipv4.iptable_filter =
                ipt_register_table(net, &packet_filter, repl);
    . . .

       return PTR_RET(net->ipv4.iptable_filter);
}
```

(`net/ipv4/netfilter/iptable_filter.c`)

Note that there are three hooks in the filter table:

- NF_INET_LOCAL_IN
- NF_INET_FORWARD
- NF_INET_LOCAL_OUT

For this example, you set the following rule, using the `iptable` command line:

```
iptables -A INPUT -p udp --dport=5001 -j LOG --log-level 1
```

The meaning of this rule is that you will dump into the syslog incoming UDP packets with destination port 5001. The `log-level` modifier is the standard syslog level in the range 0 through 7; 0 is emergency and 7 is debug. Note that when running an `iptables` command, you should specify the table you want to use with the `-t` modifier; for example, `iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE` will add a rule to the NAT table. When not specifying a table name with the `-t` modifier, you use the filter table by default. So by running `iptables -A INPUT -p udp --dport=5001 -j LOG --log-level 1`, you add a rule to the filter table.

■ **Note** You can set targets to iptables rules; usually these can be targets from the Linux netfilter subsystems (see the earlier example for using the LOG target). You can also write your own targets and extend the iptables userspace code to support them. See "Writing Netfilter modules," by Jan Engelhardt and Nicolas Bouliane: `http://inai.de/documents/Netfilter_Modules.pdf`.

Note that CONFIG_NETFILTER_XT_TARGET_LOG must be set in order to use the LOG target in an iptables rule, as shown in the earlier example. You can refer to the code of `net/netfilter/xt_LOG.c` as an example of an `iptables` target module.

When a UDP packet with destination port 5001 reaches the network driver and goes up to the network layer (L3), the first hook it encounters is the NF_INET_PRE_ROUTING hook; the filter table callback does not register a hook in NF_INET_PRE_ROUTING. It has only three hooks: NF_INET_LOCAL_IN, NF_INET_FORWARD, and NF_INET_LOCAL_OUT, as mentioned earlier. So you continue to the `ip_rcv_finish()` method and perform a lookup in the routing subsystem. Now there are two cases: the packet is intended to be delivered to the local host or intended to be forwarded (let's ignore cases when the packet is to be discarded). In Figure 9-2, you can see the packet traversal in both cases.

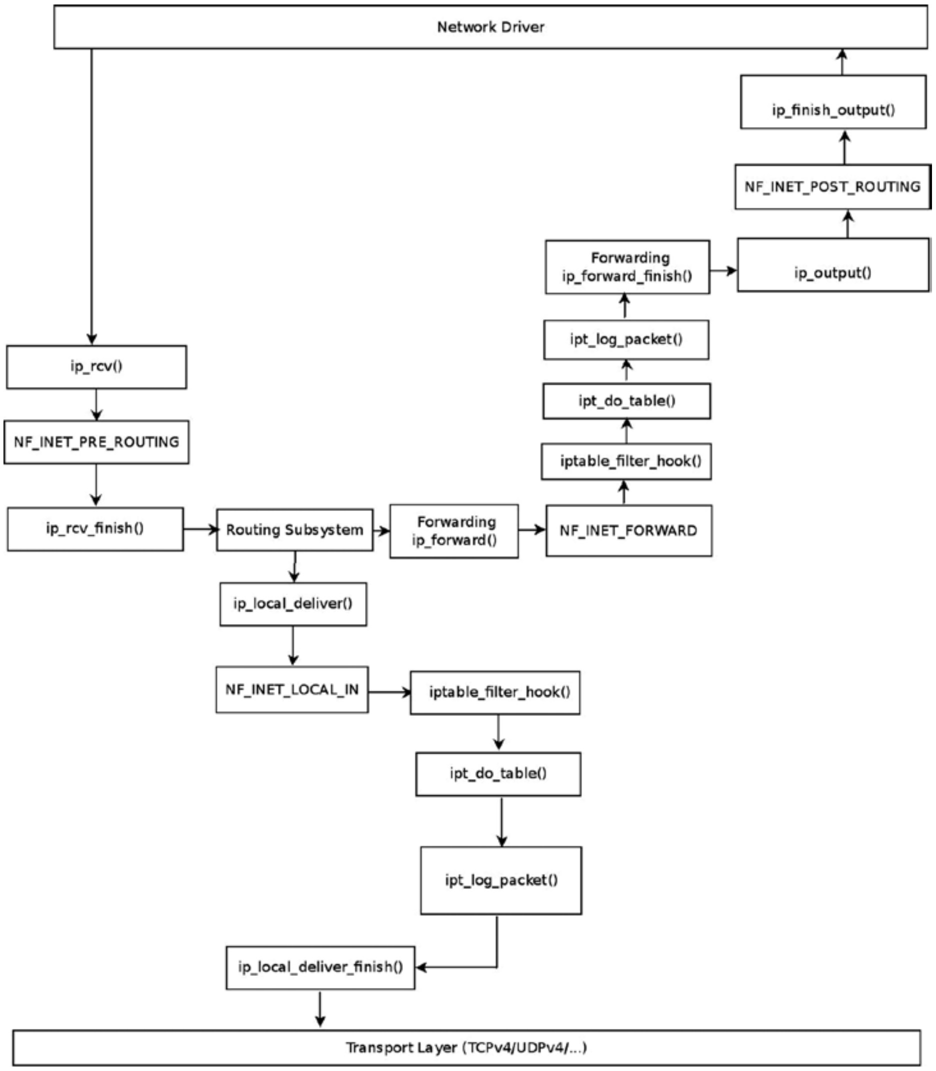


Figure 9-2. *Traffic for me and Forwarded Traffic with a Filter table rule*

Delivery to the Local Host

First you reach the `ip_local_deliver()` method; take a short look at this method:

```
int ip_local_deliver(struct sk_buff *skb)
{
    . . .
       return NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_IN, skb, skb->dev, NULL,
                       ip_local_deliver_finish);
}
```

As you can see, you have the NF_INET_LOCAL_IN hook in this method, and as mentioned earlier, NF_INET_LOCAL_IN is one of the filter table hooks; so the NF_HOOK() macro will invoke the `iptable_filter_hook()` method. Now take a look in the `iptable_filter_hook()` method:

```
static unsigned int iptable_filter_hook(unsigned int hook, struct sk_buff *skb,
                                    const struct net_device *in,
                        const struct net_device *out,
                                 int (*okfn)(struct sk_buff *))
{
        const struct net *net;
        . . .
        net = dev_net((in != NULL) ? in : out);
        . . .

        return ipt_do_table(skb, hook, in, out, net->ipv4.iptable_filter);
}
```

(net/ipv4/netfilter/iptable_filter.c)

The `ipt_do_table()` method, in fact, invokes the LOG target callback, `ipt_log_packet()`, which writes the packet headers into the syslog. If there were more rules, they would have been called at this point. Because there are no more rules, you continue to the `ip_local_deliver_finish()` method, and the packet continues its traversal to the transport layer (L4) to be handled by a corresponding socket.

Forwarding the Packet

The second case is that after a lookup in the routing subsystem, you found that the packet is to be forwarded, so the `ip_forward()` method is called:

```
int ip_forward(struct sk_buff *skb)
  {
  . . .
   return NF_HOOK(NFPROTO_IPV4, NF_INET_FORWARD, skb, skb->dev,
                      rt->dst.dev, ip_forward_finish);
   . . .
```

Because the filter table has a registered hook callback in NF_INET_FORWARD, as mentioned, you again invoke the `iptable_filter_hook()` method. And consequently, as before, you again call the `ipt_do_table()` method, which will in turn again call the `ipt_log_packet()` method. You will continue to the `ip_forward_finish()` method (note that `ip_forward_finish` is the last argument of the NF_HOOK macro above, which represents the continuation method). Then call the `ip_output()` method, and because the filter table has no NF_INET_POST_ROUTING hook, you continue to the `ip_finish_output()` method.

■ **Note** You can filter packets according to their Connection Tracking state. The next rule will dump into syslog packets whose Connection Tracking state is ESTABLISHED:

```
iptables -A INPUT -p tcp -m conntrack --ctstate ESTABLISHED -j LOG --log-level 1
```

Network Address Translation (NAT)

The Network Address Translation (NAT) module deals mostly with IP address translation, as the name implies, or port manipulation. One of the most common uses of NAT is to enable a group of hosts with a private IP address on a Local Area Network to access the Internet via some residential gateway. You can do that, for example, by setting a NAT rule. The NAT, which is installed on the gateway, can use such a rule and provide the hosts the ability to access the Web. The netfilter subsystem has NAT implementation for IPv4 and for IPv6. The IPv6 NAT implementation is mainly based on the IPv4 implementation and provides, from a user perspective, an interface similar to IPv4. IPv6 NAT support was merged in kernel 3.7. It provides some features like an easy solution to load balancing (by setting a DNAT on incoming traffic) and more. The IPv6 NAT module is in `net/ipv6/netfilter/ip6table_nat.c`. There are many types of NAT setups, and there is a lot of documentation on the Web about NAT administration. I talk about two common configurations: SNAT is source NAT, where the source IP address is changed, and DNAT is a destination NAT, where the destination IP address is changed. You can use the `-j` flag to select SNAT or DNAT. The implementation of both DNAT and SNAT is in `net/netfilter/xt_nat.c`. The next section discusses NAT initialization.

NAT initialization

The NAT table, like the filter table in the previous section, is also an `xt_table` object. It is registered on all hook points, except for the NF_INET_FORWARD hook:

```
static const struct xt_table nf_nat_ipv4_table = {
        .name           = "nat",
        .valid_hooks    = (1 << NF_INET_PRE_ROUTING) |
                          (1 << NF_INET_POST_ROUTING) |
                          (1 << NF_INET_LOCAL_OUT) |
                          (1 << NF_INET_LOCAL_IN),
        .me             = THIS_MODULE,
        .af             = NFPROTO_IPV4,
};
```

(`net/ipv4/netfilter/iptable_nat.c`)

Registration and unregistration of the NAT table is done by calling the `ipt_register_table()` and the `ipt_unregister_table()`, respectively (`net/ipv4/netfilter/iptable_nat.c`). The network namespace (`struct net`) includes an IPv4 specific object (`netns_ipv4`), which includes a pointer to the IPv4 NAT table (`nat_table`), as mentioned in the earlier "IP tables" section. This xt_table object, which is created by the ipt_register_table() method, is assigned to this nat_table pointer. You also define an array of nf_hook_ops objects and register it:

```
 static struct nf_hook_ops nf_nat_ipv4_ops[] __read_mostly = {
        /* Before packet filtering, change destination */
        {
                .hook           = nf_nat_ipv4_in,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_PRE_ROUTING,
                .priority       = NF_IP_PRI_NAT_DST,
        },
        /* After packet filtering, change source */
        {
                .hook           = nf_nat_ipv4_out,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_POST_ROUTING,
                .priority       = NF_IP_PRI_NAT_SRC,
        },
        /* Before packet filtering, change destination */
        {
                .hook           = nf_nat_ipv4_local_fn,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_LOCAL_OUT,
                .priority       = NF_IP_PRI_NAT_DST,
        },
        /* After packet filtering, change source */
        {
                .hook           = nf_nat_ipv4_fn,
                .owner          = THIS_MODULE,
                .pf             = NFPROTO_IPV4,
                .hooknum        = NF_INET_LOCAL_IN,
                .priority       = NF_IP_PRI_NAT_SRC,
        },
};
```

Registration of the nf_nat_ipv4_ops array is done in the iptable_nat_init() method:

```
static int __init iptable_nat_init(void)
{
        int err;
        . . .
        err = nf_register_hooks(nf_nat_ipv4_ops, ARRAY_SIZE(nf_nat_ipv4_ops));
        if (err < 0)
                goto err2;
        return 0;
        . . .
}
```

(net/ipv4/netfilter/iptable_nat.c)

NAT Hook Callbacks and Connection Tracking Hook Callbacks

There are some hooks on which both NAT callbacks and Connection Tracking callbacks are registered. For example, on the NF_INET_PRE_ROUTING hook (the first hook an incoming packet arrives at), there are two registered callbacks: the Connection Tracking callback, `ipv4_conntrack_in()`, and the NAT callback, `nf_nat_ipv4_in()`. The priority of the Connection Tracking callback, `ipv4_conntrack_in()`, is NF_IP_PRI_CONNTRACK (-200), and the priority of the NAT callback, `nf_nat_ipv4_in()`, is NF_IP_PRI_NAT_DST (-100). Because callbacks of the same hook with lower priorities are invoked first, the Connection Tracking `ipv4_conntrack_in()` callback, which has a priority of –200, will be invoked before the NAT `nf_nat_ipv4_in()` callback, which has a priority of –100. See Figure 9-1 for the location of the `ipv4_conntrack_in()` method and Figure 9-4 for the location of the `nf_nat_ipv4_in()`; both are in the same place, in the NF_INET_PRE_ROUTING point. The reason behind this is that NAT performs a lookup in the Connection Tracking layer, and if it does not find an entry, NAT does not perform any address translation action:

```
static unsigned int nf_nat_ipv4_fn(unsigned int hooknum,
                           struct sk_buff *skb,
                           const struct net_device *in,
                           const struct net_device *out,
                           int (*okfn)(struct sk_buff *))
{
        struct nf_conn *ct;
        . . .
        /* Don't try to NAT if this packet is not conntracked */
        if (nf_ct_is_untracked(ct))
                return NF_ACCEPT;
    . . .
}
```

(net/ipv4/netfilter/iptable_nat.c)

■ **Note** The `nf_nat_ipv4_fn ()` method is called from the NAT PRE_ROUTING callback, `nf_nat_ipv4_in()`.

On the NF_INET_POST_ROUTING hook, you have two registered Connection Tracking callbacks: the `ipv4_helper()` callback (with priority of NF_IP_PRI_CONNTRACK_HELPER, which is 300) and the `ipv4_confirm()` callback with priority of NF_IP_PRI_CONNTRACK_CONFIRM (INT_MAX, which is the highest integer value for a priority). You also have a registered NAT hook callback, `nf_nat_ipv4_out()`, with a priority of NF_IP_PRI_NAT_SRC, which is 100. As a result, when reaching the NF_INET_POST_ROUTING hook, first the NAT callback, `nf_nat_ipv4_out()`, will be called, and then the `ipv4_helper()` method will be called, and the `ipv4_confirm()` will be the last to be called. See Figure 9-4.

Let's take a look in a simple DNAT rule and see the traversal of a forwarded packet and the order in which the Connection Tracking callbacks and the NAT callbacks are called (for the sake of simplicity, assume that the filter table is not built in this kernel image). In the setup shown in Figure 9-3, the middle host (the AMD server) runs this DNAT rule:

```
iptables -t nat -A PREROUTING -j DNAT -p udp --dport 9999 --to-destination 192.168.1.8
```

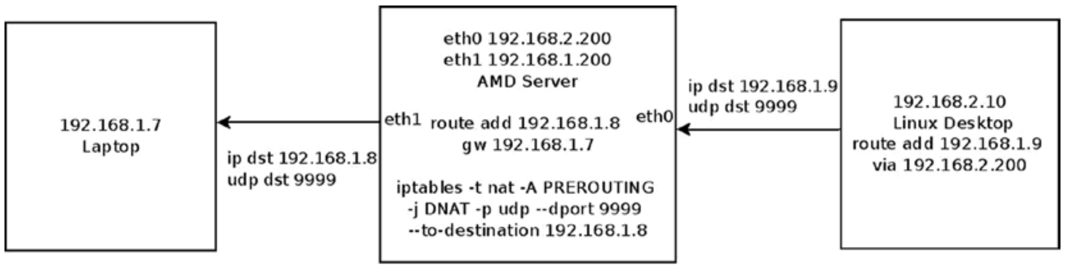


Figure 9-3. *A simple setup with a DNAT rule*

The meaning of this DNAT rule is that incoming UDP packets that are sent on UDP destination port 9999 will change their destination IP address to 192.168.1.8. The right side machine (the Linux desktop) sends UDP packets to 192.168.1.9 with UDP destination port of 9999. In the AMD server, the destination IPv4 address is changed to 192.168.1.8 by the DNAT rule, and the packets are sent to the laptop on the left.

In Figure 9-4, you can see the traversal of a first UDP packet, which is sent according to the setup mentioned earlier.

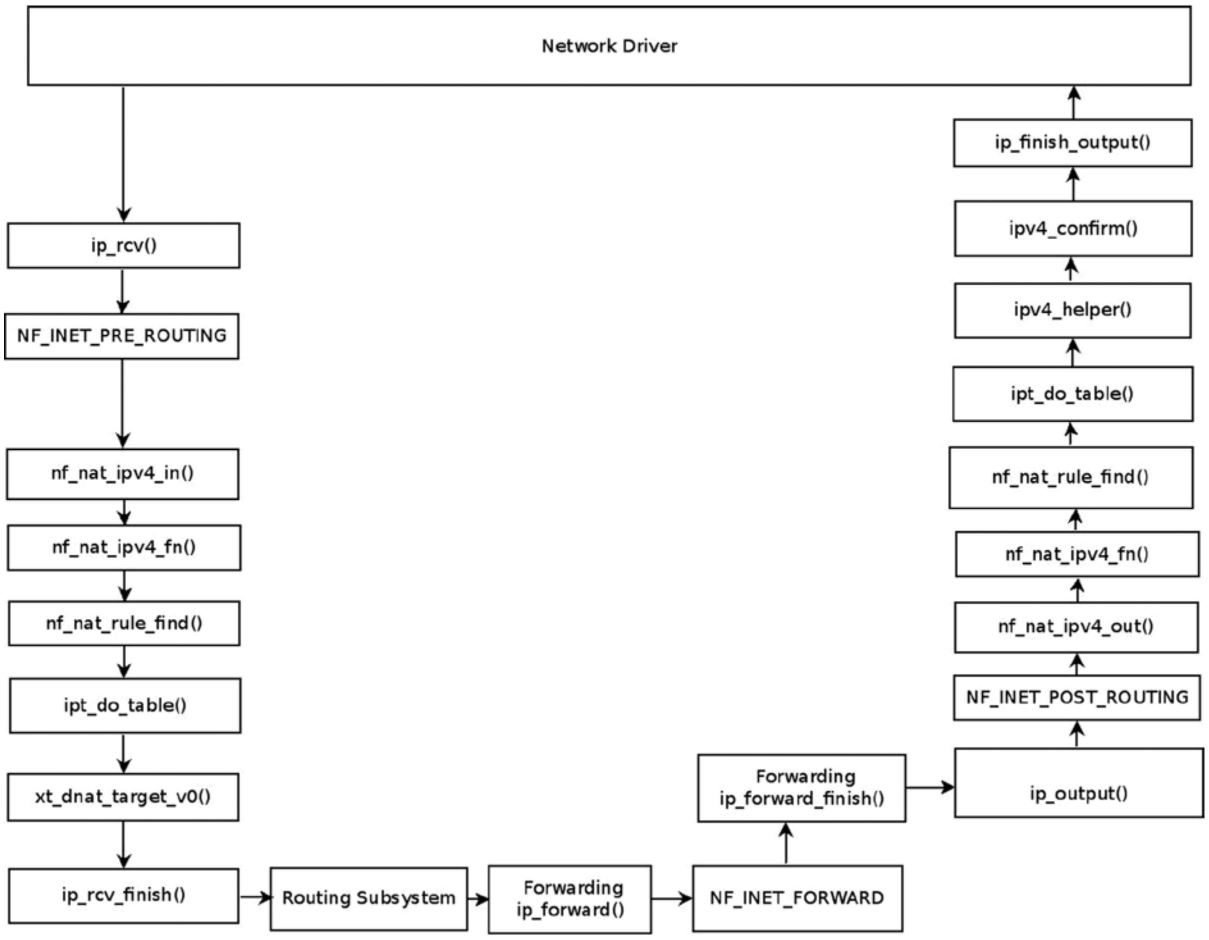


Figure 9-4. *NAT and netfilter hooks*

The generic NAT module is net/netfilter/nf_nat_core.c. The basic elements of the NAT implementation are the nf_nat_l4proto structure (include/net/netfilter/nf_nat_l4proto.h) and the nf_nat_l3proto structure. In kernels prior to 3.7, you will encounter the nf_nat_protocol structure instead of these two structures, which replaced them as part of adding IPv6 NAT support. These two structures provide a protocol-independent NAT core support.

Both of these structures contain a manip_pkt() function pointer that changes the packet headers. Let's look at an example of the manip_pkt() implementation for the TCP protocol, in net/netfilter/nf_nat_proto_tcp.c:

```
static bool tcp_manip_pkt(struct sk_buff *skb,
              const struct nf_nat_l3proto *l3proto,
              unsigned int iphdroff, unsigned int hdroff,
              const struct nf_conntrack_tuple *tuple,
              enum nf_nat_manip_type maniptype)
{
        struct tcphdr *hdr;
        __be16 *portptr, newport, oldport;
        int hdrsize = 8; /* TCP connection tracking guarantees this much */

        /* this could be an inner header returned in icmp packet; in such
           cases we cannot update the checksum field since it is outside of
           the 8 bytes of transport layer headers we are guaranteed */
        if (skb->len >= hdroff + sizeof(struct tcphdr))
                hdrsize = sizeof(struct tcphdr);

        if (!skb_make_writable(skb, hdroff + hdrsize))
                return false;

        hdr = (struct tcphdr *)(skb->data + hdroff);
```

Set newport according to maniptype:

- If you need to change the source port, maniptype is NF_NAT_MANIP_SRC. So you extract the port from the tuple->src.
- If you need to change the destination port, maniptype is NF_NAT_MANIP_DST. So you extract the port from the tuple->dst:

```
        if (maniptype == NF_NAT_MANIP_SRC) {
                /* Get rid of src port */
                newport = tuple->src.u.tcp.port;
                portptr = &hdr->source;
        } else {
                /* Get rid of dst port */
                newport = tuple->dst.u.tcp.port;
                portptr = &hdr->dest;
        }
```

You are going to change the source port (when maniptype is NF_NAT_MANIP_SRC) or the destination port (when maniptype is NF_NAT_MANIP_DST) of the TCP header, so you need to recalculate the checksum. You must keep the old port for the checksum recalculation, which will be immediately done by calling the csum_update() method and the inet_proto_csum_replace2() method:

```
        oldport = *portptr;
        *portptr = newport;

        if (hdrsize < sizeof(*hdr))
                return true;
```

Recalculate the checksum:

```
        l3proto->csum_update(skb, iphdroff, &hdr->check, tuple, maniptype);
        inet_proto_csum_replace2(&hdr->check, skb, oldport, newport, 0);
        return true;
}
```

NAT Hook Callbacks

The protocol-specific NAT module is net/ipv4/netfilter/iptable_nat.c for the IPv4 protocol, and net/ipv6/netfilter/ip6table_nat.c for the IPv6 protocol. These two NAT modules have four hooks callbacks each, shown in Table 9-1.

Table 9-1. *IPv4 and IPv6 NAT Callbacks*

Hook	Hook Callback (IPv4)	Hook Callback (IPv6)
NF_INET_PRE_ROUTING	nf_nat_ipv4_in	nf_nat_ipv6_in
NF_INET_POST_ROUTING	nf_nat_ipv4_out	nf_nat_ipv6_out
NF_INET_LOCAL_OUT	nf_nat_ipv4_local_fn	nf_nat_ipv6_local_fn
NF_INET_LOCAL_IN	nf_nat_ipv4_fn	nf_nat_ipv6_fn

The nf_nat_ipv4_fn() is the most important of these methods (for IPv4). The other three methods, nf_nat_ipv4_in(), nf_nat_ipv4_out(), and nf_nat_ipv4_local_fn(), all invoke the nf_nat_ipv4_fn() method. Let's take a look at the nf_nat_ipv4_fn() method:

```
static unsigned int nf_nat_ipv4_fn(unsigned int hooknum,
                              struct sk_buff *skb,
                              const struct net_device *in,
                              const struct net_device *out,
                              int (*okfn)(struct sk_buff *))
{
        struct nf_conn *ct;
        enum ip_conntrack_info ctinfo;
        struct nf_conn_nat *nat;
        /* maniptype == SRC for postrouting. */
        enum nf_nat_manip_type maniptype = HOOK2MANIP(hooknum);
        /* We never see fragments: conntrack defrags on pre-routing
         * and local-out, and nf_nat_out protects post-routing.
         */
        NF_CT_ASSERT(!ip_is_fragment(ip_hdr(skb)));

        ct = nf_ct_get(skb, &ctinfo);
        /* Can't track?  It's not due to stress, or conntrack would
         * have dropped it.  Hence it's the user's responsibilty to
         * packet filter it out, or implement conntrack/NAT for that
         * protocol. 8) --RR
         */
        if (!ct)
                return NF_ACCEPT;

        /* Don't try to NAT if this packet is not conntracked */
        if (nf_ct_is_untracked(ct))
                return NF_ACCEPT;

        nat = nfct_nat(ct);
        if (!nat) {
                /* NAT module was loaded late. */
                if (nf_ct_is_confirmed(ct))
                        return NF_ACCEPT;
                nat = nf_ct_ext_add(ct, NF_CT_EXT_NAT, GFP_ATOMIC);
                if (nat == NULL) {
                        pr_debug("failed to add NAT extension\n");
                        return NF_ACCEPT;
                }
        }

        switch (ctinfo) {
        case IP_CT_RELATED:
        case IP_CT_RELATED_REPLY:
                if (ip_hdr(skb)->protocol == IPPROTO_ICMP) {
                        if (!nf_nat_icmp_reply_translation(skb, ct, ctinfo,
                                                           hooknum))
                                return NF_DROP;
                        else
                                return NF_ACCEPT;
                }
                /* Fall thru... (Only ICMPs can be IP_CT_IS_REPLY) */
        case IP_CT_NEW:
                /* Seen it before?  This can happen for loopback, retrans,
                 * or local packets.
                 */
                if (!nf_nat_initialized(ct, maniptype)) {
                        unsigned int ret;
```

The `nf_nat_rule_find()` method calls the `ipt_do_table()` method, which iterates through all the matches of an entry in a specified table, and if there is a match, calls the target callback:

```
                        ret = nf_nat_rule_find(skb, hooknum, in, out, ct);
                        if (ret != NF_ACCEPT)
                                return ret;
                } else {
                        pr_debug("Already setup manip %s for ct %p\n",
                                 maniptype == NF_NAT_MANIP_SRC ? "SRC" : "DST",
                                 ct);
                        if (nf_nat_oif_changed(hooknum, ctinfo, nat, out))
                                goto oif_changed;
                }
                break;

        default:
                /* ESTABLISHED */
                NF_CT_ASSERT(ctinfo == IP_CT_ESTABLISHED ||
                             ctinfo == IP_CT_ESTABLISHED_REPLY);
                if (nf_nat_oif_changed(hooknum, ctinfo, nat, out))
                        goto oif_changed;
        }

        return nf_nat_packet(ct, ctinfo, hooknum, skb);

oif_changed:
        nf_ct_kill_acct(ct, ctinfo, skb);
        return NF_DROP;
}
```

Connection Tracking Extensions

Connection Tracking (CT) Extensions were added in kernel 2.6.23. The main point of Connection Tracking Extensions is to allocate only what is required—for example, if the NAT module is not loaded, the extra memory needed for NAT in the Connection Tracking layer will not be allocated. Some extensions are enabled by `sysctls` or even depending on certain iptables rules (for example, `-m connlabel`). Each Connection Tracking Extension module should define an `nf_ct_ext_type` object and perform registration by the `nf_ct_extend_register()` method (unregistration is done by the `nf_ct_extend_unregister()` method). Each extension should define a method to attach its Connection Tracking Extension to a connection (`nf_conn`) object, which should be called from the `init_conntrack()` method. Thus, for example, you have the `nf_ct_tstamp_ext_add()` method for the timestamp CT Extension and `nf_ct_labels_ext_add()` for the labels CT Extension. The Connection Tracking Extensions infrastructure is implemented in `net/netfilter/nf_conntrack_extend.c`. These are the Connection Tracking Extensions modules as of this writing (all under `net/netfilter`):

- `nf_conntrack_timestamp.c`
- `nf_conntrack_timeout.c`
- `nf_conntrack_acct.c`
- `nf_conntrack_ecache.c`
- `nf_conntrack_labels.c`
- `nf_conntrack_helper.c`

Summary

This chapter described the netfilter subsystem implementation. I covered the netfilter hooks and how they are registered. I also discussed important subjects such as the Connection Tracking mechanism, iptables, and NAT. Chapter 10 deals with the IPsec subsystem and its implementation.

Quick Reference

This section covers the top methods that are related to the topics discussed in this chapter, ordered by their context, followed by three tables and a short section about tools and libraries.

Methods

The following is a short list of important methods of the netfilter subsystem. Some of them were mentioned in this chapter.

struct xt_table *ipt_register_table(struct net *net, const struct xt_table *table, const struct ipt_replace *repl);

This method registers a table in the netfilter subsystem.

void ipt_unregister_table(struct net *net, struct xt_table *table);

This method unregisters a table in the netfilter subsystem.

int nf_register_hook(struct nf_hook_ops *reg);

This method registers a single `nf_hook_ops` object.

int nf_register_hooks(struct nf_hook_ops *reg, unsigned int n);

This method registers an array of *n* `nf_hook_ops` objects; the second parameter is the number of the elements in the array.

void nf_unregister_hook(struct nf_hook_ops *reg);

This method unregisters a single `nf_hook_ops` object.

void nf_unregister_hooks(struct nf_hook_ops *reg, unsigned int n);

This method unregisters an array of *n* `nf_hook_ops` objects; the second parameter is the number of the elements in the array.

static inline void nf_conntrack_get(struct nf_conntrack *nfct);

This method increments the reference count of the associated `nf_conntrack` object.

static inline void nf_conntrack_put(struct nf_conntrack *nfct);

This method decrements the reference count of the associated `nf_conntrack` object. If it reaches 0, the `nf_conntrack_destroy()` method is called.

int nf_conntrack_helper_register(struct nf_conntrack_helper *me);

This method registers an `nf_conntrack_helper` object.

static inline struct nf_conn *resolve_normal_ct(struct net *net, struct nf_conn *tmpl, struct sk_buff *skb, unsigned int dataoff, u_int16_t l3num, u_int8_t protonum, struct nf_conntrack_l3proto *l3proto, struct nf_conntrack_l4proto *l4proto, int *set_reply, enum ip_conntrack_info *ctinfo);

This method tries to find an `nf_conntrack_tuple_hash` object according to the specified SKB by calling the `__nf_conntrack_find_get()` method, and if it does not find such an entry, it creates one by calling the `init_conntrack()` method. The `resolve_normal_ct()` method is called from the `nf_conntrack_in()` method (`net/netfilter/nf_conntrack_core.c`).

struct nf_conntrack_tuple_hash *init_conntrack(struct net *net, struct nf_conn *tmpl, const struct nf_conntrack_tuple *tuple, struct nf_conntrack_l3proto *l3proto, struct nf_conntrack_l4proto *l4proto, struct sk_buff *skb, unsigned int dataoff, u32 hash);

This method allocates a Connection Tracking `nf_conntrack_tuple_hash` object. Invoked from the `resolve_normal_ct()` method, it tries to find an expectation for this connection by calling the `nf_ct_find_expectation()` method.

static struct nf_conn *__nf_conntrack_alloc(struct net *net, u16 zone, const struct nf_conntrack_tuple *orig, const struct nf_conntrack_tuple *repl, gfp_t gfp, u32 hash);

This method allocates an `nf_conn` object. Sets the timeout timer of the `nf_conn` object to be the `death_by_timeout()` method.

int xt_register_target(struct xt_target *target);

This method registers an Xtable target extension.

void xt_unregister_target(struct xt_target *target);

This method unregisters an Xtable target extension.

int xt_register_targets(struct xt_target *target, unsigned int n);

This method registers an array of Xtable target extensions; *n* is the number of targets.

void xt_unregister_targets(struct xt_target *target, unsigned int n);

This method unregisters an array of Xtable target extensions; *n* is the number of targets.

int xt_register_match(struct xt_match *target);

This method registers an Xtable match extension.

void xt_unregister_match(struct xt_match *target);

This method unregisters an Xtable match extension.

int xt_register_matches(struct xt_match *match, unsigned int n);

This method registers an array of Xtable match extensions; *n* is the number of matches.

void xt_unregister_matches(struct xt_match *match, unsigned int n);

This method unregisters an array of Xtable match extensions; *n* is the number of matches.

int nf_ct_extend_register(struct nf_ct_ext_type *type);

This method registers a Connection Tracking Extension object.

void nf_ct_extend_unregister(struct nf_ct_ext_type *type);

This method unregisters a Connection Tracking Extension object.

int __init iptable_nat_init(void);

This method initializes the IPv4 NAT table.

int __init nf_conntrack_ftp_init(void);

This method initializes the Connection Tracking FTP Helper. Calls the `nf_conntrack_helper_register()` method to register the FTP helpers.

MACRO

Let's look at the macro used in this chapter.

NF_CT_DIRECTION(hash)

This is a macro that gets an `nf_conntrack_tuple_hash` object as a parameter and returns the direction (IP_CT_DIR_ORIGINAL, which is `0`, or IP_CT_DIR_REPLY, which is `1`) of the destination (`dst` object) of the associated tuple (`include/net/netfilter/nf_conntrack_tuple.h`).

Tables

And here are the tables, showing netfilter tables in IPv4 network namespace and in IPv6 network namespace and netfilter hook priorities.

Table 9-2. *IPv4 Network Namespace (`netns_ipv4`) Tables (`xt_table` Objects)*

Linux Symbol (netns_ipv4)	Linux Module
iptable_filter	net/ipv4/netfilter/iptable_filter.c
iptable_mangle	net/ipv4/netfilter/iptable_mangle.c
iptable_raw	net/ipv4/netfilter/iptable_raw.c
arptable_filter	net/ipv4/netfilter/arp_tables.c
nat_table	net/ipv4/netfilter/iptable_nat.c
iptable_security	net/ipv4/netfilter/iptable_security.c (Note: CONFIG_SECURITY should be set).

Table 9-3. *IPv6 Network Namespace (`netns_ipv6`) Tables (`xt_table` Objects)*

Linux Symbol (netns_ipv6)	Linux Module
ip6table_filter	net/ipv6/netfilter/ip6table_filter.c
ip6table_mangle	net/ipv6/netfilter/ip6table_mangle.c
ip6table_raw	net/ipv6/netfilter/ip6table_raw.c
ip6table_nat	net/ipv6/netfilter/ip6table_nat.c
ip6table_security	net/ipv6/netfilter/ip6table_security.c (Note: CONFIG_SECURITY should be set).

Table 9-4. *Netfilter Hook Priorities*

Linux Symbol	value
NF_IP_PRI_FIRST	INT_MIN
NF_IP_PRI_CONNTRACK_DEFRAG	-400
NF_IP_PRI_RAW	-300
NF_IP_PRI_SELINUX_FIRST	-225
NF_IP_PRI_CONNTRACK	-200
NF_IP_PRI_MANGLE	-150
NF_IP_PRI_NAT_DST	-100
NF_IP_PRI_FILTER	0
NF_IP_PRI_SECURITY	50
NF_IP_PRI_NAT_SRC	100
NF_IP_PRI_SELINUX_LAST	225
NF_IP_PRI_CONNTRACK_HELPER	300
NF_IP_PRI_CONNTRACK_CONFIRM	INT_MAX
NF_IP_PRI_LAST	INT_MAX

See the `nf_ip_hook_priorities enum` definition in `include/uapi/linux/netfilter_ipv4.h`.

Tools and Libraries

The `conntrack-tools` consist of a userspace daemon, `conntrackd`, and a command line tool, `conntrack`. It provides a tool with which system administrators can interact with the netfilter Connection Tracking layer. See: `http://conntrack-tools.netfilter.org/`.

Some libraries are developed by the netfilter project and allow you to perform various userspace tasks; these libraries are prefixed with “libnetfilter”; for example, `libnetfilter_conntrack`, `libnetfilter_log`, and `libnetfilter_queue`. For more details, see the netfilter official website, `www.netfilter.org`.

CHAPTER 10

IPsec

Chapter 9 deals with the netfilter subsystem and with its kernel implementation. This chapter discusses the Internet Protocol Security (IPsec) subsystem. IPsec is a group of protocols for securing IP traffic by authenticating and encrypting each IP packet in a communication session. Most security services are provided by two major IPsec protocols: the Authentication Header (AH) protocol and the Encapsulating Security Payload (ESP) protocol. Moreover, IPsec provides protection against trying to eavesdrop and send again packets (replay attacks). IPsec is mandatory according to IPv6 specification and optional in IPv4. Nevertheless, most modern operating systems, including Linux, have support for IPsec both in IPv4 and in IPv6. The first IPsec protocols were defined in 1995 (RFCs 1825–1829). In 1998, these RFCs were deprecated by RFCs 2401–2412. Then again in 2005, these RFCs were updated by RFCs 4301–4309.

The IPsec subsystem is very complex—perhaps the most complex part of the Linux kernel network stack. Its importance is paramount when considering the growing security requirements of organizations and of private citizens. This chapter gives you a basis for delving into this complex subsystem.

General

IPsec has become a standard for most of the IP Virtual Private Network (VPN) technology in the world. That said, there are also VPNs based on different technologies, such as Secure Sockets Layer (SSL) and `pptp` (tunneling a PPP connection over the GRE protocol). Among IPsec's several modes of operation, the most important are transport mode and tunnel mode. In transport mode, only the payload of the IP packet is encrypted, whereas in tunnel mode, the entire IP packet is encrypted and inserted into a new IP packet with a new IP header. When using a VPN with IPsec, you usually work in tunnel mode, although there are cases in which you work in transport mode (L2TP/IPsec, for example).

I start with a short discussion about the Internet Key Exchange (IKE) userspace daemon and cryptography in IPsec. These are topics that are mostly not a part of the kernel networking stack but that are related to IPsec operation and are needed to get a better understanding of the kernel IPsec subsystem. I follow that with a discussion of the XFRM framework, which is the configuration and monitoring interface between the IPsec userspace part and IPsec kernel components, and explain the traversal of IPsec packets in the Tx and Rx paths. I conclude the chapter with a short section about NAT traversal in IPsec, which is an important and interesting feature, and a "Quick Reference" section. The next section begins the discussion with the IKE protocol.

IKE (Internet Key Exchange)

The most popular open source userspace Linux IPsec solutions are Openswan (and `libreswan`, which forked from Openswan), strongSwan, and racoon (of `ipsec-tools`). Racoon is part of the Kame project, which aimed to provide a free IPv6 and IPsec protocol stack implementation for variants of BSD.

To establish an IPsec connection, you need to set up a Security Association (SA). You do that with the help of the already mentioned userspace projects. An SA is defined by two parameters: a source address and a 32-bit Security Parameter Index (SPI). Both sides (called *initiator* and *responder* in IPsec terminology) should agree on parameters such as a key (or more than one key), authentication, encryption, data integrity and key exchange algorithms, and other parameters such as key lifetime (IKEv1 only). This can be done in two different ways of key distribution: by manual key exchange, which is rarely used since it is less secure, or by the IKE protocol. Openswan and strongSwan implementations provide an IKE daemon (`pluto` in Openswan and `charon` in strongSwan) that uses UDP port 500 (both source and destination) to send and receive IKE messages. Both use the XFRM Netlink interface to communicate with the native IPsec stack of the Linux kernel. The strongSwan project is the only complete open source implementation of RFC 5996, "Internet Key Exchange Protocol Version 2 (IKEv2)," whereas the Openswan project only implements a small mandatory subset.

You can use IKEv1 Aggressive Mode in Openswan and in strongSwan 5.x (for strongSwan, it should be explicitly configured, and the name of the `charon` daemon changes to be `weakSwan` in this case); but this option is regarded unsafe. IKEv1 is still used by Apple operating systems (iOS and Mac OS X) because of the built-in `racoon` legacy client. Though many implementations use IKEv1, there are many improvements and advantages when using IKEv2. I'll mention some of them very briefly: in IKEv1, more messages are needed to establish an SA than in IKEv2. IKEv1 is very complex, whereas IKEv2 is considerably simpler and more robust, mainly because each IKEv2 request message must be acknowledged by an IKEv2 response message. In IKEv1, there are no acknowledgements, but there is a backoff algorithm which, in case of packet loss, keeps trying forever. However, in IKEv1 there can be a race when the two sides perform retransmission, whereas in IKEv2 that can't happen because the responsibility for retransmission is on the initiator only. Among the other important IKEv2 features are that IKEv2 has integrated NAT traversal support, automatic narrowing of Traffic Selectors (`left|rightsubnet` on both sides don't have to match exactly, but one proposal can be a subset of the other proposal), an IKEv2 configuration payload allowing to assign virtual IPv4/IPv6 addresses and internal DNS information (replacement for IKEv1 Mode Config), and finally IKEv2 EAP authentication (replacement for the dangerous IKEv1 XAUTH protocol), which solves the problem of potentially weak PSKs by requesting a VPN server certificate and digital signature first, before the client uses a potentially weak EAP authentication algorithm (for example, EAP-MSCHAPv2).

There are two phases in IKE: the first is called Main Mode. In this stage, each side verifies the identity of the other side, and a common session key is established using the Diffie-Hellman key exchange algorithm. This mutual authentication is based on RSA or ECDSA certificates or pre-shared secrets (pre-shared key, PSKs), which are password based and assumed to be weaker. Other parameters like the Encryption algorithm and the Authentication method to be used are also negotiated. If this phase completes successfully, the two peers are said to establish an ISAKMP SA (Internet Security Association Key Management Protocol Security Association). The second phase is called Quick Mode. In this phase, both sides agree on the cryptographic algorithms to use. The IKEv2 protocol does not differentiate between phase 1 and 2 but establishes the first CHILD_SA as part of the IKE_AUTH message exchange. THE CHILD_SA_CREATE message exchange is used only to establish additional CHILD_SAs or for the periodic rekeying of the IKE and IPsec SAs. This is why IKEv1 needs nine messages to establish a single IPsec SA, whereas IKEv2 does the same in just four messages.

The next section briefly discusses cryptography in the context of IPsec (a fuller treatment of the subject would be beyond the scope of this book).

IPsec and Cryptography

There are two widely used IPsec stacks for Linux: the native Netkey stack (developed by Alexey Kuznetsov and David S. Miller) introduced with the 2.6 kernel, and the KLIPS stack, originally written for 2.0 kernel (it predates netfilter!). Netkey uses the Linux kernel Crypto API, whereas KLIPS might support more crypto hardware through Open Cryptography Framework (OCF). OCF's advantage is that it enables using asynchronous calls to encrypt/decrypt data. In the Linux kernel, most of the Crypto API performs synchronous calls. I should mention the `acrypto` kernel code, which is the asynchronous crypto layer of the Linux kernel. There are asynchronous implementations for all algorithm types. A lot of hardware crypto accelerators use the asynchronous crypto interface for crypto request offloading. That is simply because they can't block until the crypto job is done. They have to use the asynchronous API.

It is also possible to use software-implemented algorithms with the asynchronous API. For example, the cryptd crypto template can run arbitrary algorithms in asynchronous mode. And you can use the pcrypt crypto template when working in multicore environment. This template parallelizes the crypto layer by sending incoming crypto requests to a configurable set of CPUs. It also takes care of the order of the crypto requests, so it does not introduce packet reorder when used with IPsec. The use of pcrypt can speed up IPsec by magnitudes in some situations. The crypto layer has a user management API which is used by the crconf (http://sourceforge.net/projects/crconf/) tool to configure the crypto layer, so asynchronous crypto algorithms can be configured whenever needed. With the Linux 2.6.25 kernel, released in 2008, the XFRM framework started to offer support for the very efficient AEAD (Authenticated Encryption with Associated Data) algorithms (for example, AES-GCM), especially when the Intel AES-NI instruction set is available and data integrity comes nearly for free. Delving deeply into the details of cryptography in IPsec is beyond the scope of this book. For further information, I suggest reading the relevant chapters in *Network Security Essentials,* Fifth Edition by William Stallings (Prentice Hall, 2013).

The next section discusses the XFRM framework, which is the infrastructure of IPsec.

The XFRM Framework

IPsec is implemented by the XFRM (pronounced "transform") framework, originated in the USAGI project, which aimed at providing a production quality IPv6 and IPsec protocol stack. The term *transform* refers to an incoming packet or an outgoing packet being transformed in the kernel stack according to some IPsec rule. The XFRM framework was introduced in kernel 2.5. The XFRM infrastructure is protocol-family independent, which means that there is a generic part common to both IPv4 and IPv6, located under net/xfrm. Both IPv4 and IPv6 have their own implementation of ESP, AH, and IPCOMP. For example, the IPv4 ESP module is net/ipv4/esp4.c, and the IPv6 ESP module is net/ipv6/esp6.c. Apart from it, IPv4 and IPv6 implement some protocol-specific modules for supporting the XFRM infrastructure, such as net/ipv4/xfrm4_policy.c or net/ipv6/xfrm6_policy.c.

The XFRM framework supports network namespaces, which is a form of lightweight process virtualization that enables a single process or a group of processes to have their own network stack (I discuss network namespaces in Chapter 14). Each network namespace (instance of struct net) includes a member called xfrm, which is an instance of the netns_xfrm structure. This object includes many data structures and variables that you will encounter in this chapter, such as the hash tables of XFRM policies and the hash tables of XFRM states, sysctl parameters, XFRM state garbage collector, counters, and more:

```
struct netns_xfrm {
        struct hlist_head       *state_bydst;
        struct hlist_head       *state_bysrc;
        struct hlist_head       *state_byspi;
        . . .
        unsigned int            state_num;
        . . .

        struct work_struct      state_gc_work;

     . . .

        u32                     sysctl_aevent_etime;
        u32                     sysctl_aevent_rseqth;
        int                     sysctl_larval_drop;
        u32                     sysctl_acq_expires;
};
```

(include/net/netns/xfrm.h)

XFRM Initialization

In IPv4, XFRM initialization is done by calling the `xfrm_init()` method and the `xfrm4_init()` method from the `ip_rt_init()` method in `net/ipv4/route.c`. In IPv6, the `xfrm6_init()` method is invoked from the `ip6_route_init()` method for performing XFRM initialization. Communication between the userspace and the kernel is done by creating a NETLINK_XFRM netlink socket and sending and receiving netlink messages. The netlink NETLINK_XFRM kernel socket is created in the following method:

```
static int __net_init xfrm_user_net_init(struct net *net)
{
        struct sock *nlsk;
        struct netlink_kernel_cfg cfg = {
                .groups = XFRMNLGRP_MAX,
                .input  = xfrm_netlink_rcv,
        };

        nlsk = netlink_kernel_create(net, NETLINK_XFRM, &cfg);
        . . .
        return 0;
}
```

Messages sent from userspace (like XFRM_MSG_NEWPOLICY for creating a new Security Policy or XFRM_MSG_NEWSA for creating a new Security Association) are handled by the `xfrm_netlink_rcv()` method (`net/xfrm/xfrm_user.c`), which in turn calls the `xfrm_user_rcv_msg()` method (I discuss netlink sockets in Chapter 2).

The XFRM policy and the XFRM state are the fundamental data structures of the XFRM framework. I start by describing what XFRM policy is, and subsequently I describe what XFRM state is.

XFRM Policies

A Security Policy is a rule that tells IPsec whether a certain flow should be processed or whether it can bypass IPsec processing. The `xfrm_policy` structure represents an IPsec policy. A policy includes a selector (an `xfrm_selector` object). A policy is applied when its selector matches a flow. The XFRM selector consists of fields like source and destination addresses, source and destination ports, protocol, and more, which can identify a flow:

```
struct xfrm_selector {
        xfrm_address_t  daddr;
        xfrm_address_t  saddr;
        __be16  dport;
        __be16  dport_mask;
        __be16  sport;
        __be16  sport_mask;
        __u16   family;
        __u8    prefixlen_d;
        __u8    prefixlen_s;
        __u8    proto;
        int     ifindex;
        __kernel_uid32_t        user;
};
```

(include/uapi/linux/xfrm.h)

The xfrm_selector_match() method, which gets an XFRM selector, a flow, and a family (AF_INET for IPv4 or AF_INET6 for IPv6) as parameters, returns true when the specified flow matches the specified XFRM selector. Note that the xfrm_selector structure is also used in XFRM states, as you will see hereafter in this section. A Security Policy is represented by the xfrm_policy structure:

```
struct xfrm_policy {
        . . .
        struct hlist_node             bydst;
        struct hlist_node             byidx;

        /* This lock only affects elements except for entry. */
        rwlock_t                      lock;
        atomic_t                      refcnt;
        struct timer_list             timer;

        struct flow_cache_object      flo;
        atomic_t                      genid;
        u32                           priority;
        u32                           index;
        struct xfrm_mark              mark;
        struct xfrm_selector          selector;
        struct xfrm_lifetime_cfg      lft;
        struct xfrm_lifetime_cur      curlft;
        struct xfrm_policy_walk_entry walk;
        struct xfrm_policy_queue      polq;
        u8                            type;
        u8                            action;
        u8                            flags;
        u8                            xfrm_nr;
        u16                           family;
        struct xfrm_sec_ctx           *security;
        struct xfrm_tmpl              xfrm_vec[XFRM_MAX_DEPTH];
};
```

(include/net/xfrm.h)

The following description covers the important members of the xfrm_policy structure:

- refcnt: The XFRM policy reference counter; initialized to 1 in the xfrm_policy_alloc( ) method, incremented by the xfrm_pol_hold() method, and decremented by the xfrm_pol_put() method.
- timer: Per-policy timer; the timer callback is set to be xfrm_policy_timer() in the xfrm_policy_alloc() method. The xfrm_policy_timer() method handles policy expiration: it is responsible for deleting a policy when it is expired by calling the xfrm_policy_delete() method, and sending an event (XFRM_MSG_POLEXPIRE) to all registered Key Managers by calling the km_policy_expired() method.
- lft: The XFRM policy lifetime (xfrm_lifetime_cfg object). Every XFRM policy has a lifetime, which is a time interval (expressed as a time or byte count).

You can set XFRM policy lifetime values with the `ip` command and the `limit` parameter—for example:

```
ip xfrm policy add src 172.16.2.0/24 dst 172.16.1.0/24 limit byte-soft 6000 ...
```

- sets the `soft_byte_limit` of the XFRM policy lifetime (`lft`) to be 6000; see `man 8 ip xfrm`.

You can display the lifetime (`lft`) of an XFRM policy by inspecting the lifetime configuration entry when running `ip -stat xfrm policy show`.

- `curlft`: The XFRM policy current lifetime, which reflects the current status of the policy in context of lifetime. The `curlft` is an `xfrm_lifetime_cur` object. It consists of four members (all of them are fields of 64 bits, unsigned):
    - `bytes`: The number of bytes which were processed by the IPsec subsystem, incremented in the Tx path by the `xfrm_output_one()` method and in the Rx path by the `xfrm_input()` method.
    - `packets`: The number of packets that were processed by the IPsec subsystem, incremented in the Tx path by the `xfrm_output_one()` method, and in the Rx path by the `xfrm_input()` method.
    - `add_time`: The timestamp of adding the policy, initialized when adding a policy, in the `xfrm_policy_insert()` method and in the `xfrm_sk_policy_insert()` method.
    - `use_time`: The timestamp of last access to the policy. The `use_time` timestamp is updated, for example, in the `xfrm_lookup()` method or in the `__xfrm_policy_check()` method. Initialized to 0 when adding the XFRM policy, in the `xfrm_policy_insert()` method and in the `xfrm_sk_policy_insert()` method.

■ **Note** You can display the current lifetime (`curlft`) object of an XFRM policy by inspecting the lifetime current entry when running `ip -stat xfrm policy show`.

- `polq`: A queue to hold packets that are sent while there are still no XFRM states associated with the policy. As a default, such packets are discarded by calling the `make_blackhole()` method. When setting the `xfrm_larval_drop sysctl` entry to 0 (`/proc/sys/net/core/xfrm_larval_drop`), these packets are kept in a queue (`polq.hold_queue`) of SKBs; up to 100 packets (XFRM_MAX_QUEUE_LEN) can be kept in this queue. This is done by creating a dummy XFRM bundle, by the `xfrm_create_dummy_bundle()` method (see more in the "XFRM lookup" section later in this chapter). By default, the `xfrm_larval_drop sysctl` entry is set to 1 (see the `__xfrm_sysctl_init()` method in `net/xfrm/xfrm_sysctl.c`).
- `type`: Usually the type is XFRM_POLICY_TYPE_MAIN (0). When the kernel has support for subpolicy (CONFIG_XFRM_SUB_POLICY is set), two policies can be applied to the same packet, and you can use the XFRM_POLICY_TYPE_SUB (1) type. Policy that lives a shorter time in kernel should be a subpolicy. This feature is usually needed only for developers/debugging and for mobile IPv6, because you might apply one policy for IPsec and one for mobile IPv6. The IPsec policy is usually the main policy with a longer lifetime than the mobile IPv6 (sub) policy.
- `action`: Can have one of these two values:
    - XFRM_POLICY_ALLOW (0): Permit the traffic.
    - XFRM_POLICY_BLOCK(1): Disallow the traffic (for example, when using `type=reject` or `type=drop` in `/etc/ipsec.conf`).

- xfrm_nr: Number of templates associated with the policy—can be up to six templates (XFRM_MAX_DEPTH). The xfrm_tmpl structure is an intermediate structure between the XFRM state and the XFRM policy. It is initialized in the copy_templates() method, net/xfrm/xfrm_user.c.
- family: IPv4 or IPv6.
- security: A security context (xfrm_sec_ctx object) that allows the XFRM subsystem to restrict the sockets that can send or receive packets via Security Associations (XFRM states). For more details, see http://lwn.net/Articles/156604/.
- xfrm_vec: An array of XFRM templates (xfrm_tmpl objects).

The kernel stores the IPsec Security Policies in the Security Policy Database (SPD). Management of the SPD is done by sending messages from a userspace socket. For example:

- Adding an XFRM policy (XFRM_MSG_NEWPOLICY) is handled by the xfrm_add_policy() method.
- Deleting an XFRM policy (XFRM_MSG_DELPOLICY) is handled by the xfrm_get_policy() method.
- Displaying the SPD (XFRM_MSG_GETPOLICY) is handled by the xfrm_dump_policy() method.
- Flushing the SPD (XFRM_MSG_FLUSHPOLICY) is handled by the xfrm_flush_policy() method.

The next section describes what XFRM state is.

XFRM States (Security Associations)

The xfrm_state structure represents an IPsec Security Association (SA) (include/net/xfrm.h). It represents unidirectional traffic and includes information such as cryptographic keys, flags, request id, statistics, replay parameters, and more. You add XFRM states by sending a request (XFRM_MSG_NEWSA) from a userspace socket; it is handled in the kernel by the xfrm_state_add() method (net/xfrm/xfrm_user.c). Likewise, you delete a state by sending an XFRM_MSG_DELSA message, and it is handled in the kernel by the xfrm_del_sa() method:

```
struct xfrm_state {
        . . .
        union {
                struct hlist_node       gclist;
                struct hlist_node       bydst;
        };
        struct hlist_node       bysrc;
        struct hlist_node       byspi;

        atomic_t                refcnt;
        spinlock_t              lock;

        struct xfrm_id          id;
        struct xfrm_selector    sel;
        struct xfrm_mark        mark;
        u32                     tfcpad;
        u32                     genid;

        /* Key manager bits */
        struct xfrm_state_walk  km;

        /* Parameters of this state. */
        struct {
                u32             reqid;
                u8              mode;
                u8              replay_window;
                u8              aalgo, ealgo, calgo;
                u8              flags;
                u16             family;
                xfrm_address_t  saddr;
                int             header_len;
                int             trailer_len;
        } props;

        struct xfrm_lifetime_cfg lft;

        /* Data for transformer */
        struct xfrm_algo_auth   *aalg;
        struct xfrm_algo        *ealg;
        struct xfrm_algo        *calg;
        struct xfrm_algo_aead   *aead;

        /* Data for encapsulator */
        struct xfrm_encap_tmpl  *encap;

        /* Data for care-of address */
        xfrm_address_t  *coaddr;

        /* IPComp needs an IPIP tunnel for handling uncompressed packets */
        struct xfrm_state       *tunnel;

        /* If a tunnel, number of users + 1 */
        atomic_t                tunnel_users;

        /* State for replay detection */
        struct xfrm_replay_state replay;
        struct xfrm_replay_state_esn *replay_esn;

        /* Replay detection state at the time we sent the last notification */
        struct xfrm_replay_state preplay;
        struct xfrm_replay_state_esn *preplay_esn;

        /* The functions for replay detection. */
        struct xfrm_replay      *reply;
        /* internal flag that only holds state for delayed aevent at the
         * moment
        */
        u32                     xflags;

        /* Replay detection notification settings */
        u32                     replay_maxage;
        u32                     replay_maxdiff;

        /* Replay detection notification timer */
        struct timer_list       rtimer;

        /* Statistics */
        struct xfrm_stats       stats;

        struct xfrm_lifetime_cur curlft;
        struct tasklet_hrtimer  mtimer;

        /* used to fix curlft->add_time when changing date */
        long            saved_tmo;

        /* Last used time */
        unsigned long           lastused;

        /* Reference to data common to all the instances of this
         * transformer. */
        const struct xfrm_type  *type;
        struct xfrm_mode        *inner_mode;
        struct xfrm_mode        *inner_mode_iaf;
        struct xfrm_mode        *outer_mode;

        /* Security context */
        struct xfrm_sec_ctx     *security;

        /* Private data of this transformer, format is opaque,
         * interpreted by xfrm_type methods. */
        void                    *data;
};
```

(include/net/xfrm.h)

The following description details some of the important members of the `xfrm_state` structure:

- `refcnt`: A reference counter, incremented by the `xfrm_state_hold()` method and decremented by the `__xfrm_state_put()` method or by the `xfrm_state_put()` method (the latter also releases the XFRM state by calling the `__xfrm_state_destroy()` method when the reference counter reaches 0).
- `id`: The `id` (`xfrm_id` object) consists of three fields, which uniquely define it: destination address, spi, and security protocol (AH, ESP, or IPCOMP).

- `props`: The properties of the XFRM state. For example:
    - `mode`: Can be one of five modes (for example, XFRM_MODE_TRANSPORT for transport mode or XFRM_MODE_TUNNEL for tunnel mode; see `include/uapi/linux/xfrm.h`).
    - `flag`: For example, XFRM_STATE_ICMP. These flags are available in `include/uapi/linux/xfrm.h`. These flags can be set from userspace, for example, with the `ip` command and the `flag` option: `ip xfrm add state flag icmp` ...
    - `family`: IPv4 of IPv6.
    - `saddr`: The source address of the XFRM state.
    - `lft`: The XFRM state lifetime (`xfrm_lifetime_cfg` object).
    - `stats`: An `xfrm_stats` object, representing XFRM state statistics. You can display the XFRM state statistics by `ip -stat xfrm show`.

The kernel stores the IPsec Security Associations in the Security Associations Database (SAD). The `xfrm_state` objects are stored in three hash tables in `netns_xfrm` (the XFRM namespace, discussed earlier): `state_bydst`, `state_bysrc`, `state_byspi`. The keys to these tables are computed by the `xfrm_dst_hash()`, `xfrm_src_hash()`, and `xfrm_spi_hash()` methods, respectively. When an `xfrm_state` object is added, it is inserted into these three hash tables. If the value of the spi is 0 (the value 0 is not normally to be used for spi—I will shortly mention when it is 0), the `xfrm_state` object is not added to the `state_byspi` hash table (see the `__xfrm_state_insert()` method in `net/xfrm/xfrm_state.c`).

■ **Note** An spi with value of 0 is only used for acquire states. The kernel sends an acquire message to the key manager and adds a temporary acquire state with spi 0 if traffic matches a policy, but the state is not yet resolved. The kernel does not bother to send a further acquire as long as the acquire state exists; the lifetime can be configured at `net->xfrm.sysctl_acq_expires`. If the state gets resolved, this acquire state is replaced by the actual state.

Lookup in the SAD can be done by the following:

- `xfrm_state_lookup()` method: In the `state_byspi` hash table.
- `xfrm_state_lookup_byaddr()` method: In the `state_bysrc` hash table.
- `xfrm_state_find()` method: In the `state_bydst` hash table.

The ESP protocol is the most commonly used IPsec protocol; it supports both encryption and authentication. The next section discusses the IPv4 ESP implementation.

ESP Implementation (IPv4)

The ESP protocol is specified in RFC 4303; it supports both encryption and authentication. Though it also supports encryption-only and authentication-only modes, it is usually used with both encryption and authentication because it is safer. I should also mention here the new Authenticated Encryption (AEAD) methods like AES-GCM, which can do the encryption and data integrity computations in a single pass and can be highly parallelized on multiple cores, so that with the Intel AES-NI instruction set, an IPsec throughput of several Gbit/s can be achieved. The ESP protocol supports both tunnel mode and transport mode; the protocol identifier is 50 (IPPROTO_ESP). The ESP adds a new header and a trailer to each packet. According to the ESP format, illustrated in Figure 10-1, there are the following fields:

- *SPI:* A 32-bit Security Parameter Index. Together with the source address, it identities an SA.
- *Sequence Number:* 32 bits, incremented by 1 for each transmitted packet in order to protect against replay attacks.
- *Payload Data:* A variable size encrypted data block.
- *Padding:* Padding for the encrypted data block in order to satisfy alignment requirements (0–255 bytes).
- *Pad Length:* The size of padding in bytes (1 byte).
- *Next Header:* The type of the next header (1 byte).
- *Authentication Data:* The Integrity Check Value (ICV).

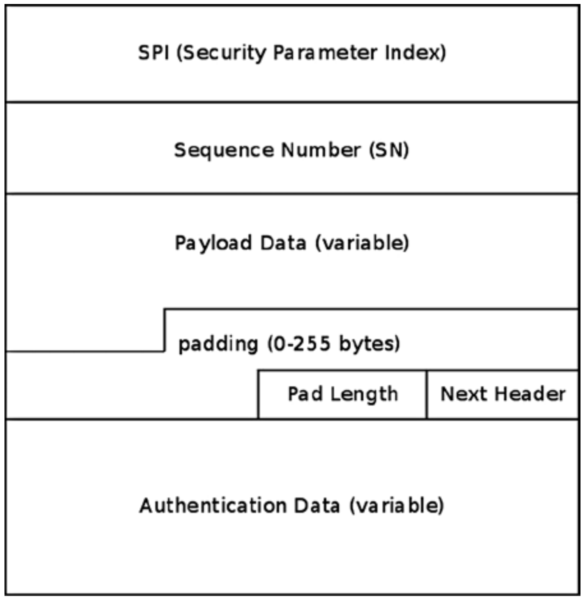


Figure 10-1. *ESP format*

The next section discusses IPv4 ESP initialization.

IPv4 ESP Initialization

We first define an esp_type (xfrm_type object) and esp4_protocol (net_protocol object) and register them thus:

```
static const struct xfrm_type esp_type =
{
        .description    = "ESP4",
        .owner          = THIS_MODULE,
        .proto          = IPPROTO_ESP,
        .flags          = XFRM_TYPE_REPLAY_PROT,
        .init_state     = esp_init_state,
        .destructor     = esp_destroy,
        .get_mtu        = esp4_get_mtu,
        .input          = esp_input,
        .output         = esp_output
};

static const struct net_protocol esp4_protocol = {
        .handler        =       xfrm4_rcv,
        .err_handler    =       esp4_err,
        .no_policy      =       1,
        .netns_ok       =       1,
};

static int __init esp4_init(void)
{
```

Each protocol family has an instance of an xfrm_state_afinfo object, which includes protocol-family specific state methods; thus there is xfrm4_state_afinfo for IPv4 (net/ipv4/xfrm4_state.c) and xfrm6_state_afinfo for IPv6. This object includes an array of xfrm_type objects called type_map. Registering XFRM type by calling the xfrm_register_type() method will set the specified xfrm_type as an element in this array:

```
        if (xfrm_register_type(&esp_type, AF_INET) < 0) {
                pr_info("%s: can't add xfrm type\n", __func__);
                return -EAGAIN;
        }
```

Registering the IPv4 ESP protocol is done like registering any other IPv4 protocol, by calling the inet_add_protocol() method. Note that the protocol handler used by IPv4 ESP, namely the xfrm4_rcv() method, is also used by the IPv4 AH protocol (net/ipv4/ah4.c) and by the IPv4 IPCOMP (IP Payload Compression Protocol) protocol (net/ipv4/ipcomp.c).

```
        if (inet_add_protocol(&esp4_protocol, IPPROTO_ESP) < 0) {
                pr_info("%s: can't add protocol\n", __func__);
                xfrm_unregister_type(&esp_type, AF_INET);
                return -EAGAIN;
        }
        return 0;
}
```

(net/ipv4/esp4.c)

Receiving an IPsec Packet (Transport Mode)

Suppose you work in transport mode in IPv4, and you receive an ESP packet that is destined to the local host. ESP in transport mode does not encrypt the IP header, only the IP payload. Figure 10-2 shows the traversal of an incoming IPv4 ESP packet, and its stages are described in this section. We will pass all the usual stages of local delivery, starting with the `ip_rcv()` method, and we will reach the `ip_local_deliver_finish()` method. Because the value of the protocol field in the IPv4 header is ESP (50), we invoke its handler, which is the `xfrm4_rcv()` method, as you saw earlier. The `xfrm4_rcv()` method further calls the generic `xfrm_input()` method, which performs a lookup in the SAD by calling the `xfrm_state_lookup()` method. If the lookup fails, the packet is dropped. In case of a lookup hit, the `input` callback method of the corresponding IPsec protocol is invoked:

```
int xfrm_input(struct sk_buff *skb, int nexthdr, __be32 spi, int encap_type)
{
        struct xfrm_state *x;
        do {
                . . .
```

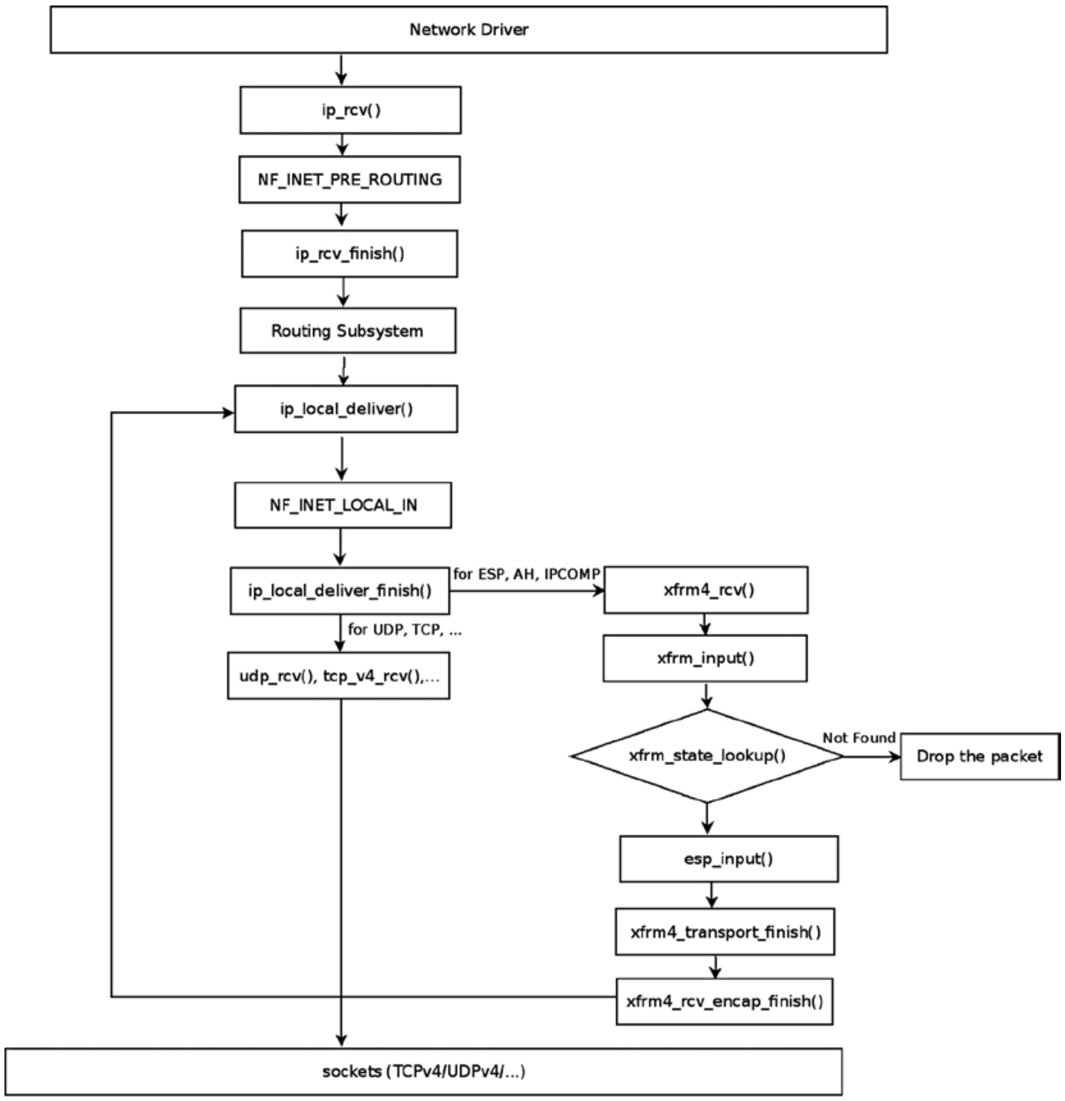


Figure 10-2. *Receiving IPv4 ESP packet, local delivery, transport mode. Note: The figure describes an IPv4 ESP packet. For IPv4 AH packets, the ah_input() method is invoked instead of the esp_input( ) method; likewise, for IPv4 IPCOMP packets, the ipcomp_input() method is invoked instead of the esp_input( ) method*

Perform a lookup in the `state_byspi` hash table:

```
x = xfrm_state_lookup(net, skb->mark, daddr, spi, nexthdr, family);
```

Drop the packet silently if the lookup failed:

```
if (x == NULL) {
        XFRM_INC_STATS(net, LINUX_MIB_XFRMINNOSTATES);
        xfrm_audit_state_notfound(skb, family, spi, seq);
        goto drop;
}
```

In this case, of IPv4 ESP incoming traffic, the XFRM type associated with the state (`x->type`) is the ESP XFRM Type (`esp_type`); its `input` callback was set to `esp_input()`, as mentioned earlier in the "IPv4 ESP initialization" section.

By calling `x->type->input()`, in the following line the `esp_input()` method is invoked; this method returns the protocol number of the original packet, before it was encrypted by ESP:

```
                    nexthdr = x->type->input(x, skb);
                    . . .
```

The original protocol number is kept in the control buffer (`cb`) of the SKB by using the XFRM_MODE_SKB_CB macro; it will be used later for modifying the IPv4 header of the packet, as you will see:

```
                    XFRM_MODE_SKB_CB(skb)->protocol = nexthdr;
```

After the `esp_input()` method terminates, the `xfrm4_transport_finish()` method is invoked. This method modifies various fields of the IPv4 header. Take a look at the `xfrm4_transport_finish()` method:

```
int xfrm4_transport_finish(struct sk_buff *skb, int async)
{
        struct iphdr *iph = ip_hdr(skb);
```

The protocol of the IPv4 header (`iph->protocol`) is 50 (ESP) at this point; you should set it to be the protocol number of the original packet (before it was encrypted by ESP) so that it will be processed by L4 sockets. The protocol number of the original packet was kept in `XFRM_MODE_SKB_CB(skb)->protocol`, as you saw earlier in this section:

```
        iph->protocol = XFRM_MODE_SKB_CB(skb)->protocol;

        . . .
        __skb_push(skb, skb->data - skb_network_header(skb));
        iph->tot_len = htons(skb->len);
```

Recalculate the checksum, since the IPv4 header was modified:

```
        ip_send_check(iph);
```

Invoke any netfilter NF_INET_PRE_ROUTING hook callback and then call the `xfrm4_rcv_encap_finish()` method:

```
        NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING, skb, skb->dev, NULL,
                xfrm4_rcv_encap_finish);
        return 0;
}
```

The `xfrm4_rcv_encap_finish()` method calls the `ip_local_deliver()` method. Now the value of the `protocol` member in the IPv4 header is the original transport protocol (UDPv4, TCPv4, and so on), so from now on you proceed in the usual packet traversal, and the packet is passed to the transport layer (L4).

Sending an IPsec Packet (Transport Mode)

Figure 10-3 shows the Tx path of an outgoing packet sent via IPv4 ESP in transport mode. The first step after performing a lookup in the routing subsystem (by calling the `ip_route_output_flow()` method), is to perform a lookup for an XFRM policy, which can be applied on this flow. You do that by calling the `xfrm_lookup()` method (I discuss the internals of this method later in this section). If there is a lookup hit, continue to the `ip_local_out()` method, and then, after calling several methods as you can see in Figure 10-3, you eventually reach the `esp_output()` method, which encrypts the packet and then sends it out by calling the `ip_output()` method.

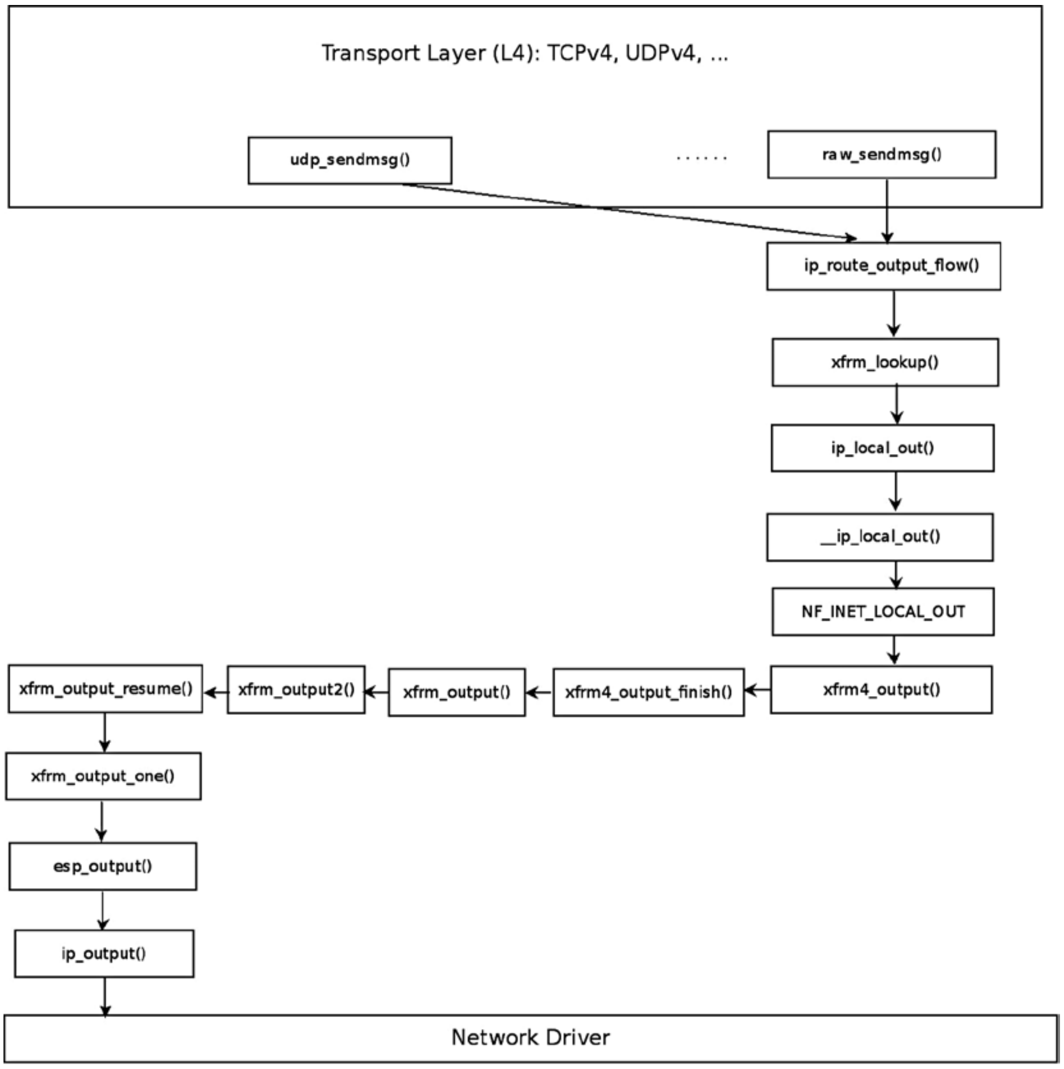


Figure 10-3. *Transmitting IPv4 ESP packet, transport mode. For the sake of simplicity, the case of creating a dummy bundle (when there are no XFRM states) and some other details are omitted*

The following section talks about how a lookup is performed in XFRM.

XFRM Lookup

The `xfrm_lookup()` method is called for each packet that is sent out of the system. You want this lookup to be as efficient as possible. To achieve this goal, bundles are used. Bundles let you cache important information such as the route, the policies, the number of policies, and more; these bundles, which are instances of the `xfrm_dst` structure, are stored by using the flow cache. When the first packet of some flow arrives, you create an entry in the generic flow cache and subsequently create a bundle (`xfrm_dst` object). The bundle creation is done after a lookup for this bundle fails, because it is the first packet of this flow. When subsequent packets of this flow arrive, you will get a hit when performing a flow cache lookup:

```
struct xfrm_dst {
        union {
                struct dst_entry        dst;
                struct rtable           rt;
                struct rt6_info         rt6;
        } u;
        struct dst_entry *route;
        struct flow_cache_object flo;
        struct xfrm_policy *pols[XFRM_POLICY_TYPE_MAX];
        int num_pols, num_xfrms;
#ifdef CONFIG_XFRM_SUB_POLICY
        struct flowi *origin;
        struct xfrm_selector *partner;
#endif
        u32 xfrm_genid;
        u32 policy_genid;
        u32 route_mtu_cached;
        u32 child_mtu_cached;
        u32 route_cookie;
        u32 path_cookie;
};
```

(include/net/xfrm.h)

The `xfrm_lookup()` method is a very complex method. I discuss its important parts but I don't delve into all its nuances. Figure *10-4* shows a block diagram of the internals of the `xfrm_lookup()` method.

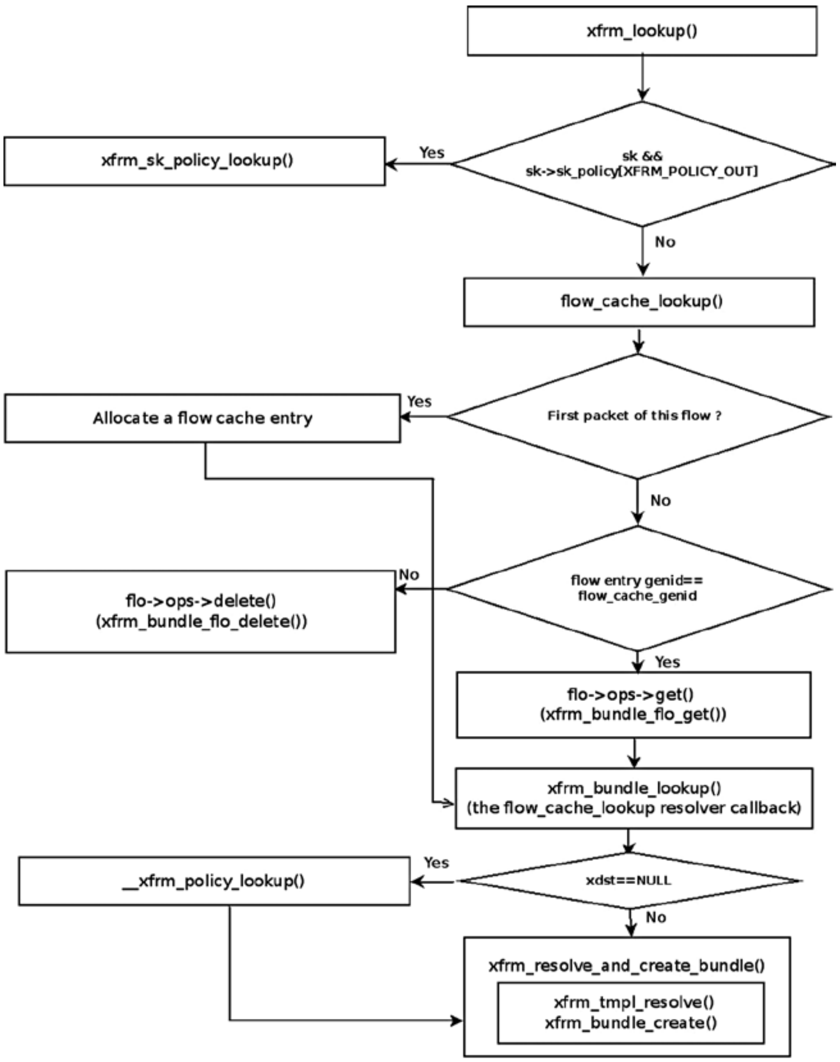


Figure 10-4. *xfrm_lookup() internals*

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

```
struct dst_entry *xfrm_lookup(struct net *net, struct dst_entry *dst_orig,
                              const struct flowi *fl, struct sock *sk, int flags)
{
```

The `xfrm_lookup()` method handles only the Tx path; so you set the flow direction (`dir`) to be FLOW_DIR_OUT by:

```
        u8 dir = policy_to_flow_dir(XFRM_POLICY_OUT);
```

If a policy is associated with this socket, you perform a lookup by the `xfrm_sk_policy_lookup()` method, which checks whether the packet flow matches the policy selector. Note that if the packet is to be forwarded, the `xfrm_lookup()` method was invoked from the `__xfrm_route_forward()` method, and there is no socket associated with the packet, because it was not generated on the local host; in this case, the specified `sk` argument is NULL:

```
        if (sk && sk->sk_policy[XFRM_POLICY_OUT]) {
                num_pols = 1;
                pols[0] = xfrm_sk_policy_lookup(sk, XFRM_POLICY_OUT, fl);



        . . .
}
```

If there is no policy associated with this socket, you perform a lookup in the generic flow cache by calling the `flow_cache_lookup()` method, passing as an argument a function pointer to the `xfrm_bundle_lookup` method (the `resolver` callback). The key to the lookup is the flow object (the specified `fl` parameter). If you don't find an entry in the flow cache, allocate a new flow cache entry. If you find an entry with the same `genid`, call the `xfrm_bundle_flo_get()` method by invoking `flo->ops->get(flo)`. Eventually, you call the `xfrm_bundle_lookup()` method by invoking the `resolver` callback, which gets the flow object as a parameter (`oldflo`). See the `flow_cache_lookup()` method implementation in `net/core/flow.c`:

```
                flo = flow_cache_lookup(net, fl, family, dir, xfrm_bundle_lookup, dst_orig);
```

Fetch the bundle (`xfrm_dst` object) that contains the flow cache object as a member:

```
                xdst = container_of(flo, struct xfrm_dst, flo);
```

Fetch cached data, like the number of policies, number of templates, the policies and the route:

```
        num_pols = xdst->num_pols;
        num_xfrms = xdst->num_xfrms;
        memcpy(pols, xdst->pols, sizeof(struct xfrm_policy*) * num_pols);
        route = xdst->route;
}

dst = &xdst->u.dst;
```

Next comes handling a dummy bundle. A *dummy bundle* is a bundle where the route member is NULL. It is created in the XFRM bundle lookup process (by the `xfrm_bundle_lookup()` method) when no XFRM states were found, by calling the `xfrm_create_dummy_bundle()` method. In such a case, either one of the two options are available, according to the value of `sysctl_larval_drop` (`/proc/sys/net/core/xfrm_larval_drop`):

- If `sysctl_larval_drop` is set (which means its value is 1—it is so by default, as mentioned earlier in this chapter), the packet should be discarded.
- If `sysctl_larval_drop` is not set (its value is 0), the packets are kept in a per-policy queue (`polq.hold_queue`), which can contain up to 100 (XFRM_MAX_QUEUE_LEN) SKBs; this is implemented by the `xdst_queue_output()` method. These packets are kept until the XFRM

states are resolved or until some timeout elapses. Once the states are resolved, the packets are sent out of the queue. If the XFRM states are not resolved after some time interval (the timeout of the xfrm_policy_queue object), the queue is flushed by the xfrm_queue_purge() method:

```
if (route == NULL && num_xfrms > 0) {
        /* The only case when xfrm_bundle_lookup() returns a
         * bundle with null route, is when the template could
         * not be resolved. It means policies are there, but
         * bundle could not be created, since we don't yet
         * have the xfrm_state's. We need to wait for KM to
         * negotiate new SA's or bail out with error.*/
         if (net->xfrm.sysctl_larval_drop) {
```

For IPv4, the make_blackhole() method calls the ipv4_blackhole_route() method. For IPv6, it calls the ip6_blackhole_route() method:

```
        return make_blackhole(net, family, dst_orig);
}
```

The next section covers one of the most important features of IPsec—NAT traversal—and explains what it is and why it is needed.

NAT Traversal in IPsec

Why don't NAT devices allow IPsec traffic to pass? NAT changes the IP addresses and sometimes also the port numbers of the packet. As a result, it recalculates the checksum of the TCP or the UDP header. The transport layer checksum calculation takes into account the source and destination of the IP addresses. So even if only the IP addresses were changed, the TCP or UDP checksum should be recalculated. However, with ESP encryption in transport mode, the NAT device can't update the checksum because the TCP or UDP headers are encrypted with ESP. There are protocols where the checksum does not cover the IP header (like SCTP), so this problem does not occur there. To solve these problems, the NAT traversal standard for IPsec was developed (or, as officially termed in RFC 3948, "UDP Encapsulation of IPsec ESP Packets"). UDP Encapsulation can be applied to IPv4 packets as well as to IPv6 packets. NAT traversal solutions are not limited to IPsec traffic; these techniques are typically required for client-to-client networking applications, especially for peer-to-peer and Voice over Internet Protocol (VoIP) applications.

There are some partial solutions for VoIP NAT-traversal, such as STUN, TURN, ICE, and more. I should mention here that strongSwan implements the IKEv2 Mediation Extension service (http://tools.ietf.org/html/draft-brunner-ikev2-mediation-00), which allows two VPN endpoints located behind a NAT router each to establish a direct peer-to-peer IPsec tunnel using a mechanism similar to TURN and ICE. STUN, for example, is used in the VoIP open source Ekiga client (formerly gnomemeeting). The problem with these solutions is NAT devices they don't cope with. Devices called SBCs (session border controllers) provide a full solution for NAT traversal in VoIP. SBCs can be implemented in hardware (Juniper Networks, for example, provides a router-integrated SBC solution) or in software. These SBC solutions perform NAT traversal of the media traffic—which is sent by Real Time Protocol (RTP)—and sometimes also for the signaling traffic—which is sent by Session Initiation Protocol (SIP). NAT traversal is optional in IKEv2. Openswan, strongSwan, and racoon support NAT traversal, but Openswan and racoon support NAT-T only with IKEv1, whereas strongSwan supports NAT traversal in both IKEv1 and IKEv2.

NAT-T Mode of Operation

How does NAT traversal work? First, keep in mind that NAT-T is a good solution only for ESP traffic and not for AH. Another restriction is that NAT-T can't be used with manual keying, but only with IKEv1 and IKEv2. This is because NAT-T is tied with exchanging IKEv1/IKEv2 messages. First, you must tell the userspace daemon (`pluto`) that you want to use the NAT traversal feature, because it is not activated by default. You do that in Openswan by adding `nat_traversal=yes` to the connection parameters in `/etc/ipsec.conf`. Clients not behind a NAT are not affected by the addition of this entry. In strongSwan, the IKEv2 `charon` daemon always supports NAT traversal, and this feature cannot be deactivated. In the first phase of IKE (Main Mode), you check whether both peers support NAT-T. In IKEv1, when a peer supports NAT-T, one of the ISAKAMP header members (vendor ID) tells whether it supports NAT-T. In IKEv2, NAT-T is part of the standard and does not have to be announced. If this condition is met, you check whether there is one or more NAT devices in the path between the two IPsec peers by sending NAT-D payload messages. If this condition is also met, NAT-T protects the original IPsec encoded packet by inserting in it a UDP header between the IP header and the ESP header. Both the source and destination ports in the UDP header are 4500. Besides, NAT-T sends keep-alive messages every 20 seconds so that the NAT retains its mapping. Keep alive messages are also sent on UDP port 4500 and are recognized by their content and value (which is one byte, 0xFF). When this packet reaches the IPsec peer, after going through the NAT, the kernel strips the UDP header and decrypts the ESP payload. See the `xfrm4_udp_encap_rcv()` method in `net/ipv4/xfrm4_input.c`.

Summary

This chapter covered IPsec and the XFRM framework, which is the infrastructure of IPsec, and XFRM policies and states, which are the fundamental data structures of the XFRM framework. I also discussed IKE, the ESP4 implementation, the Rx/Tx path of ESP4 in transport mode, and NAT traversal in IPsec. Chapter 11 deals with the following transport Layer (L4) protocols: UDP, TCP, SCTP, and DCCP. The "Quick Reference" section that follows covers the top methods 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 of IPsec. Some of them were mentioned in this chapter. Afterward, I include a table of XFRM SNMP MIB counters.

Methods

Let's start with the methods.

bool xfrm_selector_match(const struct xfrm_selector *sel, const struct flowi *fl, unsigned short family);

This method returns `true` when the specified flow matches the specified XFRM selector. Invokes the `__xfrm4_selector_match()` method for IPv4 or the `__xfrm6_selector_match()` method for IPv6.

int xfrm_policy_match(const struct xfrm_policy *pol, const struct flowi *fl, u8 type, u16 family, int dir);

This method returns 0 if the specified policy can be applied to the specified flow, otherwise it returns an `-errno`.

struct xfrm_policy *xfrm_policy_alloc(struct net *net, gfp_t gfp);

This method allocates and initializes an XFRM policy. It sets its reference counter to 1, initializes the read-write lock, assigns the policy namespace (`xp_net`) to be the specified network namespace, sets its timer callback to be `xfrm_policy_timer()`, and sets its state resolution packet queue timer (`policy->polq.hold_timer`) callback to be `xfrm_policy_queue_process()`.

void xfrm_policy_destroy(struct xfrm_policy *policy);

This method removes the timer of specified XFRM policy object and releases the specified XFRM policy memory.

void xfrm_pol_hold(struct xfrm_policy *policy);

This method increments by 1 the reference count of the specified XFRM policy.

static inline void xfrm_pol_put(struct xfrm_policy *policy);

This method decrements by 1 the reference count of the specified XFRM policy. If the reference count reaches 0, call the `xfrm_policy_destroy()` method.

struct xfrm_state_afinfo *xfrm_state_get_afinfo(unsigned int family);

This method returns the `xfrm_state_afinfo` object associated with the specified protocol family.

struct dst_entry *xfrm_bundle_create(struct xfrm_policy *policy, struct xfrm_state **xfrm, int nx, const struct flowi *fl, struct dst_entry *dst);

This method creates an XFRM bundle. Called from the `xfrm_resolve_and_create_bundle()` method.

int policy_to_flow_dir(int dir);

This method returns the flow direction according to the specified policy direction. For example, return FLOW_DIR_IN when the specified direction is XFRM_POLICY_IN, and so on.

static struct xfrm_dst *xfrm_create_dummy_bundle(struct net *net, struct dst_entry *dst, const struct flowi *fl, int num_xfrms, u16 family);

This method creates a dummy bundle. Called from the `xfrm_bundle_lookup()` method when policies were found but there are no matching states.

struct xfrm_dst *xfrm_alloc_dst(struct net *net, int family);

This method allocates an XFRM bundle object. Called from the `xfrm_bundle_create()` method and from the `xfrm_create_dummy_bundle()` method.

int xfrm_policy_insert(int dir, struct xfrm_policy *policy, int excl);

This method adds an XFRM policy to the SPD. Invoked from the `xfrm_add_policy()` method (`net/xfrm/xfrm_user.c`), or from the `pfkey_spdadd()` method (`net/key/af_key.c`).

int xfrm_policy_delete(struct xfrm_policy *pol, int dir);

This method releases the resources of the specified XFRM policy object. The direction argument (`dir`) is needed to decrement by 1 the corresponding XFRM policy counter in the `policy_count` in the per namespace `netns_xfrm` object.

int xfrm_state_add(struct xfrm_state *x);

This method adds the specified XFRM state to the SAD.

int xfrm_state_delete(struct xfrm_state *x);

This method deletes the specified XFRM state from the SAD.

void __xfrm_state_destroy(struct xfrm_state *x);

This method releases the resources of an XFRM state by adding it to the XFRM states garbage list and activating the XFRM state garbage collector.

int xfrm_state_walk(struct net *net, struct xfrm_state_walk *walk, int (*func)(struct xfrm_state *, int, void*), void *data);

This method iterates over all XFRM states (`net->xfrm.state_all`) and invokes the specified `func` callback.

struct xfrm_state *xfrm_state_alloc(struct net *net);

This method allocates and initializes an XFRM state.

void xfrm_queue_purge(struct sk_buff_head *list);

This method flushes the state resolution per-policy queue (`polq.hold_queue`).

int xfrm_input(struct sk_buff *skb, int nexthdr, __be32 spi, int encap_type);

This method is the main Rx IPsec handler.

static struct dst_entry *make_blackhole(struct net *net, u16 family, struct dst_entry *dst_orig);

This method is invoked from the `xfrm_lookup()` method when there are no resolved states and `sysctl_larval_drop` is set. For IPv4, the `make_blackhole()` method calls the `ipv4_blackhole_route()` method; for IPv6, it calls the `ip6_blackhole_route()` method.

int xdst_queue_output(struct sk_buff *skb);

This method handles adding packets to the per-policy state resolution packet queue (`pq->hold_queue`). This queue can contain up to 100 (XFRM_MAX_QUEUE_LEN) packets.

struct net *xs_net(struct xfrm_state *x);

This method returns the namespace object (`xs_net`) associated with the specified `xfrm_state` object.

struct net *xp_net(const struct xfrm_policy *xp);

This method returns the namespace object (`xp_net`) associated with the specified `xfrm_policy` object.

int xfrm_policy_id2dir(u32 index);

This method returns the direction of the policy according to the specified index.

int esp_input(struct xfrm_state *x, struct sk_buff *skb);

This method is the main IPv4 ESP protocol handler.

struct ip_esp_hdr *ip_esp_hdr(const struct sk_buff *skb);

This method returns the ESP header associated with the specified SKB.

int verify_newpolicy_info(struct xfrm_userpolicy_info *p);

This method verifies that the specified `xfrm_userpolicy_info` object contains valid values. (`xfrm_userpolicy_info` is the object which is passed from userspace). It returns 0 if it is a valid object, and -EINVAL or -EAFNOSUPPORT if not.

Table

Table 10-1 lists XFRM SNMP MIB counters.

Table 10-1. XFRM SNMP MIB counters

Linux Symbol	SNMP (procfs) Symbol	Methods in Which the Counter Might Be Incremented
LINUX_MIB_XFRMINERROR	XfrmInError	xfrm_input()
LINUX_MIB_XFRMINBUFFERERROR	XfrmInBufferError	xfrm_input(),__xfrm_policy_check()
LINUX_MIB_XFRMINHDRERROR	XfrmInHdrError	xfrm_input(),__xfrm_policy_check()
LINUX_MIB_XFRMINNOSTATES	XfrmInNoStates	xfrm_input()
LINUX_MIB_XFRMINSTATEPROTOERROR	XfrmInStateProtoError	xfrm_input()
LINUX_MIB_XFRMINSTATEMODEERROR	XfrmInStateModeError	xfrm_input()
LINUX_MIB_XFRMINSTATESEQERROR	XfrmInStateSeqError	xfrm_input()
LINUX_MIB_XFRMINSTATEEXPIRED	XfrmInStateExpired	xfrm_input()
LINUX_MIB_XFRMINSTATEMISMATCH	XfrmInStateMismatch	xfrm_input(), __xfrm_policy_check()
LINUX_MIB_XFRMINSTATEINVALID	XfrmInStateInvalid	xfrm_input()
LINUX_MIB_XFRMINTMPLMISMATCH	XfrmInTmplMismatch	__xfrm_policy_check()
LINUX_MIB_XFRMINNOPOLS	XfrmInNoPols	__xfrm_policy_check()
LINUX_MIB_XFRMINPOLBLOCK	XfrmInPolBlock	__xfrm_policy_check()
LINUX_MIB_XFRMINPOLERROR	XfrmInPolError	__xfrm_policy_check()
LINUX_MIB_XFRMOUTERROR	XfrmOutError	xfrm_output_one(),xfrm_output()
LINUX_MIB_XFRMOUTBUNDLEGENERROR	XfrmOutBundleGenError	xfrm_resolve_and_create_bundle()
LINUX_MIB_XFRMOUTBUNDLECHECKERROR	XfrmOutBundleCheckError	xfrm_resolve_and_create_bundle()
LINUX_MIB_XFRMOUTNOSTATES	XfrmOutNoStates	xfrm_lookup()
LINUX_MIB_XFRMOUTSTATEPROTOERROR	XfrmOutStateProtoError	xfrm_output_one()
LINUX_MIB_XFRMOUTSTATEMODEERROR	XfrmOutStateModeError	xfrm_output_one()
LINUX_MIB_XFRMOUTSTATESEQERROR	XfrmOutStateSeqError	xfrm_output_one()
LINUX_MIB_XFRMOUTSTATEEXPIRED	XfrmOutStateExpired	xfrm_output_one()
LINUX_MIB_XFRMOUTPOLBLOCK	XfrmOutPolBlock	xfrm_lookup()
LINUX_MIB_XFRMOUTPOLDEAD	XfrmOutPolDead	n/a
LINUX_MIB_XFRMOUTPOLERROR	XfrmOutPolError	xfrm_bundle_lookup(), xfrm_resolve_and_create_bundle()
LINUX_MIB_XFRMFWDHDRERROR	XfrmFwdHdrError	__xfrm_route_forward()
LINUX_MIB_XFRMOUTSTATEINVALID	XfrmOutStateInvalid	xfrm_output_one()

■ **Note** The IPsec git tree: `git://git.kernel.org/pub/scm/linux/kernel/git/klassert/ipsec.git.`

The ipsec git tree is for fixes for the IPsec networking subsystem; the development in this tree is done against David Miller's net git tree.

The ipsec-next git tree: `git://git.kernel.org/pub/scm/linux/kernel/git/klassert/ipsec-next.git.`

The ipsec-next tree is for changes for IPsec with linux-next as target; the development in this tree is done against David Miller's net-next git tree.

The IPsec subsystem maintainers are Steffen Klassert, Herbert Xu, and David S. Miller.