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.

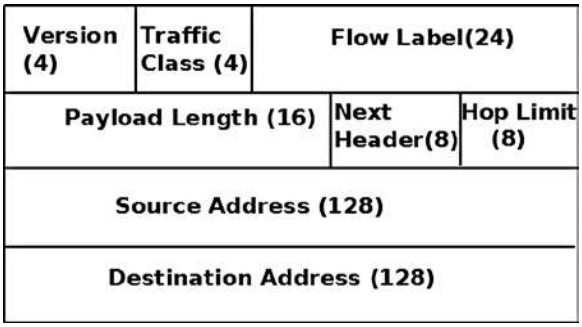

Figure 8-1. *IPv6 header*

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.

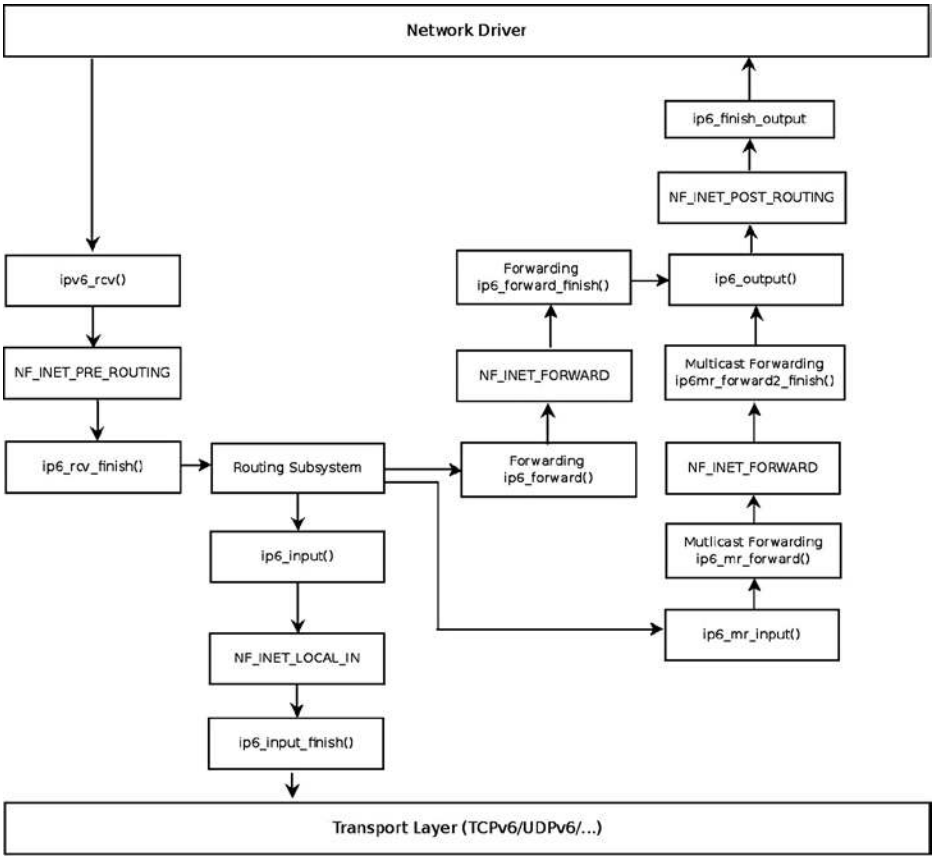

Figure 8-2. *Receiving IPv6 packets*

■ **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.

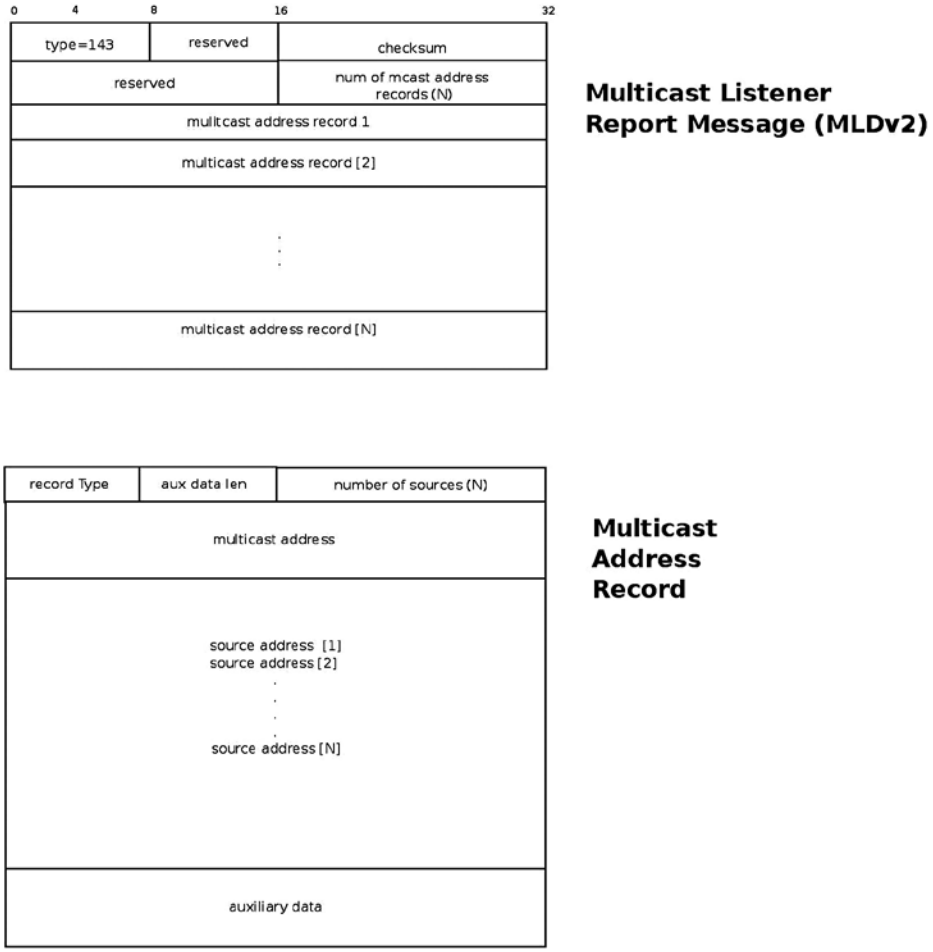

Figure 8-3. *MLDv2 Multicast Listener Report*

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.