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.

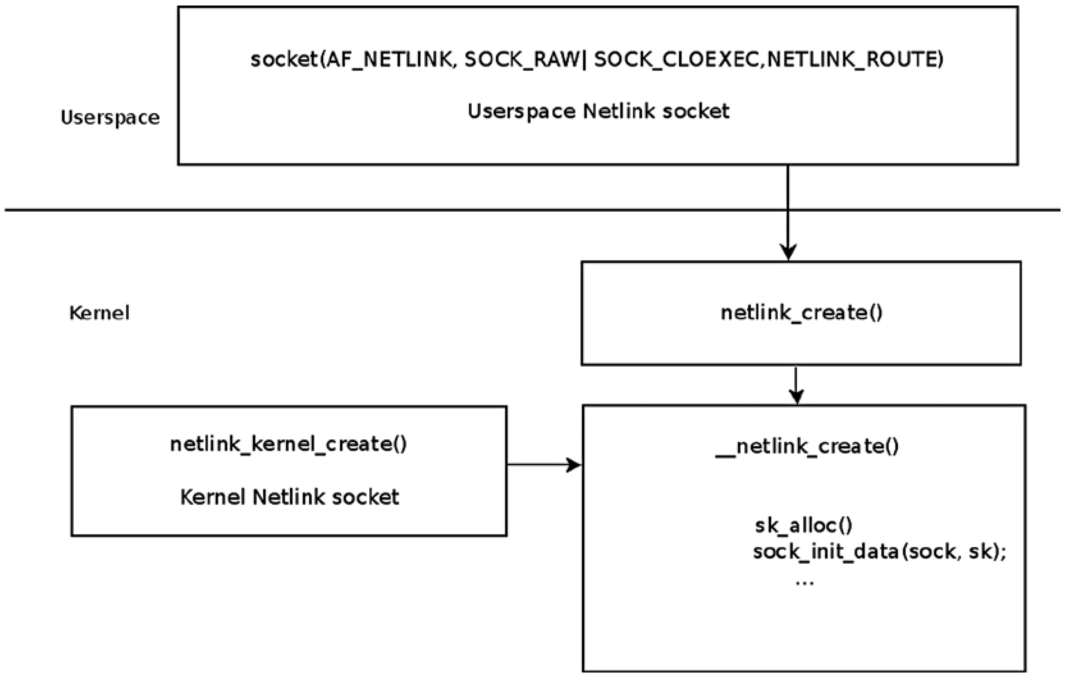

Figure 2-1. *Creating a netlink socket in the kernel and in userspace*

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.

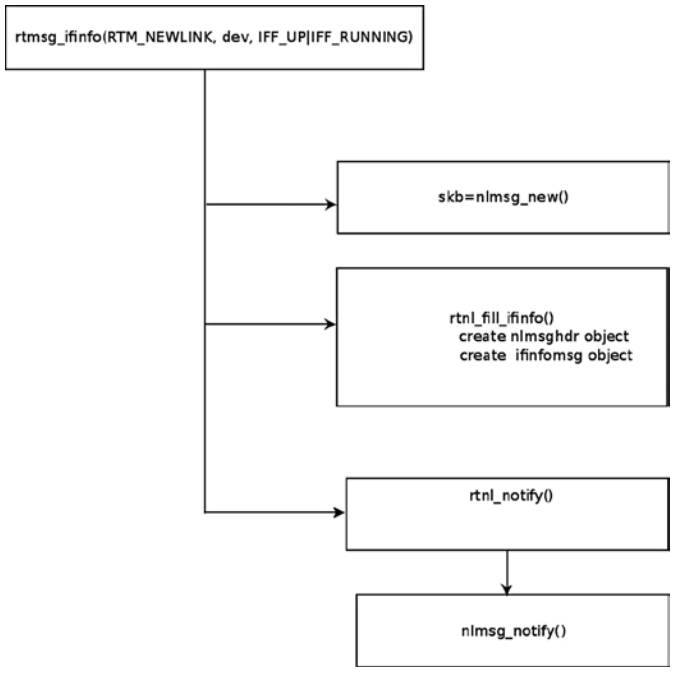

Figure 2-2. *Sending of rtnelink messages with the `rtmsg_ifinfo()` method*

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.

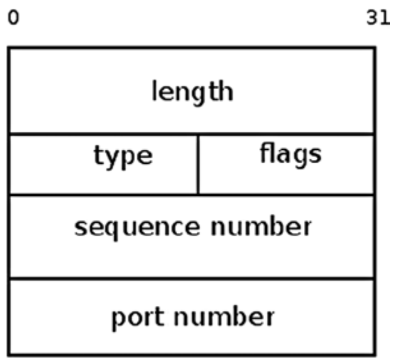

Figure 2-3. *nlmsg header*

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.

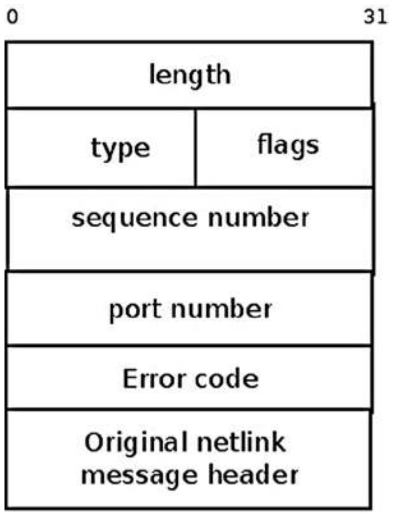

Figure 2-4. *Netlink error message*

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.

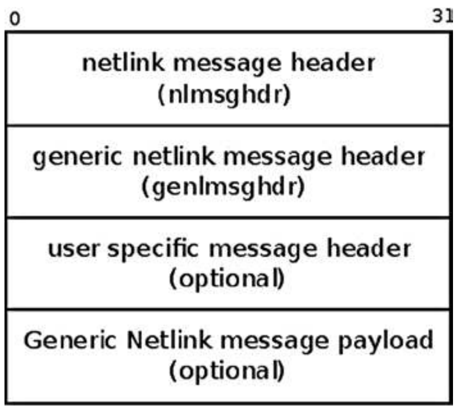
0
31
netlink message header
(nlmsghdr)
generic netlink message header
(genlmsghdr)
user specific message header
(optional)
Generic Netlink message payload
(optional)

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`.