Linux Kernel Networking

Implementation and Theory

(Chapter 11~14)

Rami Rosen

Apress®

Contents

CHAPTER 11

Layer 4 Protocols

Chapter 10 discussed the Linux IPsec subsystem and its implementation. In this chapter, I will discuss four transport layer (L4) protocols. I will start our discussion with the two most commonly used transport layer (L4) protocols, the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP), which are used for many years. Subsequently, I will discuss the newer Stream Control Transmission Protocol (SCTP) and Datagram Congestion Control Protocol (DCCP) protocols, which combine features of TCP and UDP. I will start the chapter with describing the sockets API, which is the interface between the transport layer (L4) and the userspace. I will discuss how sockets are implemented in the kernel and how data flows from the userspace to the transport layer and from the transport layer to the userspace. I will also deal with passing packets from the network layer (L3) to the transport layer (L4) when working with these protocols. I will discuss here mainly the IPv4 implementation of these four protocols, though some of the code is common to IPv4 and IPv6.

Sockets

Every operating system has to provide an entry point and an API to its networking subsystems. The Linux kernel networking subsystem provides an interface to the userspace by the standard POSIX socket API, which was specified by the IEEE (IEEE Std 1003.1g-2000, describing networking APIs, also known as POSIX.1g). This API is based on Berkeley sockets API (also known as BSD sockets), which originated from the 4.2BSD Unix operating system and is an industry standard in several operating systems. In Linux, everything above the transport layer belongs to the userspace. Conforming to the Unix paradigm that "everything is a file," sockets are associated with files, as you will see later in this chapter. Using the uniform sockets API makes porting applications easier. These are the available socket types:

- **Stream sockets (SOCK_STREAM):** Provides a reliable, byte-stream communication channel. TCP sockets are an example of stream sockets.
- **Datagram sockets (SOCK_DGRAM):** Provides for exchanging of messages (called *datagrams*). Datagram sockets provide an unreliable communication channel, because packets can be discarded, arrive out of order, or be duplicated. UDP sockets are an example of datagram sockets.
- **Raw sockets (SOCK_RAW):** Uses direct access to the IP layer, and allows sending or receiving traffic without any protocol-specific, transport-layer formatting.
- **Reliably delivered message (SOCK_RDM):** Used by the Transparent Inter-Process Communication (TIPC), which was originally developed at Ericsson from 1996–2005 and was used in cluster applications. See `http://tipc.sourceforge.net`.

- **Sequenced packet stream (SOCK_SEQPACKET):** This socket type is similar to the SOCK_STREAM type and is also connection-oriented. The only difference between these types is that record boundaries are maintained using the SOCK_SEQPACKET type. Record boundaries are visible to the receiver via the MSG_EOR (End of record) flag. The Sequenced packet stream type is not discussed in this chapter.
- **DCCP sockets (SOCK_DCCP):** The Datagram Congestion Control Protocol is a transport protocol that provides a congestion-controlled flow of unreliable datagrams. It combines features of both TCP and UDP. It is discussed in a later section of this chapter.
- **Data links sockets (SOCK_PACKET):** The SOCK_PACKET is considered obsolete in the AF_INET family. See the `__sock_create()` method in `net/socket.c`.

The following is a description of some methods that the sockets API provides (all the kernel methods that appear in the following list are implemented in `net/socket.c`):

- `socket()`: Creates a new socket; will be discussed in the subsection "Creating Sockets."
- `bind()`: Associates a socket with a local port and an IP address; implemented in the kernel by the `sys_bind()` method.
- `send()`: Sends a message; implemented in the kernel by the `sys_send()` method.
- `recv()`: Receives a message; implemented in the kernel by the `sys_recv()` method.
- `listen()`: Allows a socket to receive connections from other sockets; implemented in the kernel by the `sys_listen()` method. Not relevant to datagram sockets.
- `accept()`: Accepts a connection on a socket; implemented in the kernel by the `sys_accept()` method. Relevant only with connection-based socket types (SOCK_STREAM, SOCK_SEQPACKET).
- `connect()`: Establishes a connection to a peer socket; implemented in the kernel by the `sys_connect()` method. Relevant to connection-based socket types (SOCK_STREAM or SOCK_SEQPACKET) as well as to connectionless socket types (SOCK_DGRAM).

This book focuses on the kernel network implementation, so I will not delve into the details of the userspace socket API. If you want more information, I recommend the following books:

- *Unix Network Programming, Volume 1*: *The Sockets Networking API (3rd Edition)* by W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff (Addison-Wesley Professional, 2003).
- *The Linux Programming Interface* by Michael Kerrisk (No Starch Press, 2010).

■ **Note** All the socket API calls are handled by the `socketcall()` method, in `net/socket.c`.

Now that you have learned about some socket types, you will learn what happens in the kernel when a socket is created. In the next section, I will introduce the two structures that implement sockets: `struct socket` and `struct sock`. I will also describe the difference between them and I will describe the `msghdr struct` and its members.

Creating Sockets

There are two structures that represent a socket in the kernel: the first is `struct socket`, which provides an interface to the userspace and is created by the `sys_socket()` method. I will discuss the `sys_socket()` method later in this section. The second is `struct sock`, which provides an interface to the network layer (L3). Since the `sock` structure

resides in the network layer, it is a protocol agnostic structure. I will discuss the sock structure also later in this section. The socket structure is short:

```
struct socket {
        socket_state            state;

        kmemcheck_bitfield_begin(type);
        short                   type;
        kmemcheck_bitfield_end(type);

        unsigned long           flags;

        . . .

        struct file             *file;
        struct sock             *sk;
        const struct proto_ops  *ops;
};
```

(include/linux/net.h)

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

- state: A socket can be in one of several states, like SS_UNCONNECTED, SS_CONNECTED, and more. When an INET socket is created, its state is SS_UNCONNECTED; see the inet_create() method. After a stream socket connects successfully to another host, its state is SS_CONNECTED. See the socket_state enum in include/uapi/linux/net.h.
- type: The type of the socket, like SOCK_STREAM or SOCK_RAW; see the enum sock_type in include/linux/net.h.
- flags: The socket flags; for example, the SOCK_EXTERNALLY_ALLOCATED flag is set in the TUN device when allocating a socket, not by the socket() system call. See the tun_chr_open() method in drivers/net/tun.c. The socket flags are defined in include/linux/net.h.
- file: The file associated with the socket.
- sk: The sock object associated with the socket. The sock object represents the interface to the network layer (L3). When creating a socket, the associated sk object is created. For example, in IPv4, the inet_create() method, which is invoked when creating a socket, allocates a sock object, sk, and associates it with the specified socket object.
- ops: This object (an instance of the proto_ops object) consists mostly of callbacks for this socket, like connect(), listen(), sendmsg(), recvmsg(), and more. These callbacks are the interface to the userspace. The sendmsg() callback implements several library-level routines, such as write(), send(), sendto(), and sendmsg(). Quite similarly, the recvmsg() callback implements several library-level routines, such as read(), recv(), recvfrom(), and recvmsg(). Each protocol defines a proto_ops object of its own according to the protocol requirements. Thus, for TCP, its proto_ops object includes a listen callback, inet_listen(), and an accept callback, inet_accept(). On the other hand, the UDP protocol, which does not work in the client-server model, defines the listen() callback to be the sock_no_listen() method, and it defines the accept() callback to be the sock_no_accept() method. The only thing that both these methods do is return an error of –EOPNOTSUPP. See Table 11-1 in the "Quick Reference" section at the end of this chapter for the definitions of the TCP and UDP proto_ops objects. The proto_ops structure is defined in include/linux/net.h.

The sock structure is the network-layer representation of sockets; it is quite long, and following here are only some of its fields that are important for our discussion:

```
struct sock {

        struct sk_buff_head     sk_receive_queue;
        int                     sk_rcvbuf;

        unsigned long           sk_flags;

        int                     sk_sndbuf;
        struct sk_buff_head     sk_write_queue;
        . . .
        unsigned int            sk_shutdown  : 2,
                                sk_no_check  : 2,
                                sk_protocol  : 8,
                                sk_type      : 16;
        . . .

        void                    (*sk_data_ready)(struct sock *sk, int bytes);
        void                    (*sk_write_space)(struct sock *sk);
};
```

(include/net/sock.h)

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

- sk_receive_queue: A queue for incoming packets.
- sk_rcvbuf: The size of the receive buffer in bytes.
- sk_flags: Various flags, like SOCK_DEAD or SOCK_DBG; see the sock_flags enum definition in include/net/sock.h.
- sk_sndbuf: The size of the send buffer in bytes.
- sk_write_queue: A queue for outgoing packets.

■ **Note** You will see later, in the "TCP Socket Initialization" section, how the sk_rcvbuf and the sk_sndbuf are initialized, and how this can be changed by writing to procfs entries.

- sk_no_check: Disable checksum flag. Can be set with the SO_NO_CHECK socket option.
- sk_protocol: This is the protocol identifier, which is set according to the third parameter (protocol) of the socket() system call.
- sk_type: The type of the socket, like SOCK_STREAM or SOCK_RAW; see the enum sock_type in include/linux/net.h.
- sk_data_ready: A callback to notify the socket that new data has arrived.
- sk_write_space: A callback to indicate that there is free memory available to proceed with data transmission.

Creating sockets is done by calling the socket() system call from userspace:

```
sockfd = socket(int socket_family, int socket_type, int protocol);
```

The following is a description of the parameters of the socket() system call:

- socket_family: Can be, for example, AF_INET for IPv4, AF_INET6 for IPv6, or AF_UNIX for UNIX domain sockets, and so on. (UNIX domain sockets is a form of Inter Process Communication (IPC), which allows communication between processes that are running on the same host.)
- socket_type: Can be, for example, SOCK_STREAM for stream sockets, SOCK_DGRAM for datagram sockets, or SOCK_RAW for raw sockets, and so on.
- protocol: Can be any of the following:
  - 0 or IPPROTO_TCP for TCP sockets.
  - 0 or IPPROTO_UDP for UDP sockets.
  - A valid IP protocol identifier (like IPPROTO_TCP or IPPROTO_ICMP) for raw sockets; see RFC 1700, "Assigned Numbers."

The return value of the socket() system call (sockfd) is the file descriptor that should be passed as a parameter to subsequent calls with this socket. The socket() system call is handled in the kernel by the sys_socket() method. Let's take a look at the implementation of the socket() system call:

```
SYSCALL_DEFINE3(socket, int, family, int, type, int, protocol)
{
        int retval;
        struct socket *sock;
        int flags;

        . . .
        retval = sock_create(family, type, protocol, &sock);
        if (retval < 0)
                goto out;
        . . .
        retval = sock_map_fd(sock, flags & (O_CLOEXEC | O_NONBLOCK));
        if (retval < 0)
                goto out_release;
out:
        . . .
        return retval;

}
```

(net/socket.c)

The sock_create() method calls the address-family specific socket creation method, create(); in the case of IPv4, it is the inet_create() method. (See the inet_family_ops definition in net/ipv4/af_inet.c.) The inet_create() method creates the sock object (sk) that is associated with the socket; the sock object represents the network layer socket interface. The sock_map_fd() method returns an fd (file descriptor) that is associated with the socket; normally, the socket() system call returns this fd.

Sending data from a userspace socket, or receiving data in a userspace socket from the transport layer, is handled in the kernel by the `sendmsg()` and `recvmsg()` methods, respectively, which get a `msghdr` object as a parameter. The `msghdr` object includes the data blocks to send or to fill, as well as some other parameters.

```
struct msghdr {
        void            *msg_name;       /* Socket name                                          */
        int             msg_namelen;     /* Length of name                                       */
        struct iovec    *msg_iov;        /* Data blocks                                          */
        __kernel_size_t msg_iovlen;      /* Number of blocks                                     */
        void            *msg_control;    /* Per protocol magic (eg BSD file descriptor passing) */
        __kernel_size_t msg_controllen;  /* Length of cmsg list                                  */
        unsigned int    msg_flags;
};

(include/linux/socket.h)
```

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

- `msg_name`: The destination socket address. To get the destination socket, you usually cast the `msg_name` opaque pointer to a `struct sockaddr_in` pointer. See, for example, the `udp_sendmsg()` method.
- `msg_namelen`: The length of the address.
- `iovec`: A vector of data blocks.
- `msg_iovlen`: The number of blocks in the `iovec` vector.
- `msg_control`: Control information (also known as *ancillary data*).
- `msg_controllen`: The length of the control information.
- `msg_flags`: Flags of received messages, like MSG_MORE. (See, for example, the section "Sending Packets with UDP" later in this chapter.)

Note that the maximum control buffer length that the kernel can process is limited per socket by the value in `sysctl_optmem_max` (`/proc/sys/net/core/optmem_max`).

In this section, I described the kernel implementation of the `socket` and the `msghdr struct`, which is used when sending and receiving packets. In the next section, I will start my discussion about transport layer protocols (L4) by describing the UDP protocol, which is the simplest among the protocols to be discussed in this chapter.

UDP (User Datagram Protocol)

The UDP protocol is described in RFC 768 from 1980. The UDP protocol is a thin layer around the IP layer, adding only port, length, and checksum information. It dates back as early as 1980 and provides unreliable, message-oriented transport without congestion control. Many protocols use UDP. I will mention, for example, the RTP protocol (Real-time Transport Protocol), which is used for delivery of audio and video over IP networks. Such a type of traffic can tolerate some packet loss. The RTP is commonly used in VoIP applications, usually in conjunction with SIP (Session Initiation Protocol) based clients.(It should be mentioned here that, in fact, the RTP protocol can also use TCP, as specified in RFC 4571, but this is not used much.) I should mention here UDP-Lite, which is an extension of the UDP protocol to

support variable-length checksums (RFC 3828). Most of UDP-Lite is implemented in `net/ipv4/udplite.c`, but you will encounter it also in the main UDP module, `net/ipv4/udp.c`. The UDP header length is 8 bytes:

```
struct udphdr {
        __be16  source;
        __be16  dest;
        __be16  len;
        __sum16 check;
};
(include/uapi/linux/udp.h)
```

The following is a description of the members of the UDP header:

- `source`: The source port (16 bit), in the range 1-65535.
- `dest`: The destination port (16 bit), in the range 1-65535.
- `len`: The length in bytes (the payload length and the UDP header length).
- `checksum`: The checksum of the packet.

Figure 11-1 shows a UDP header.

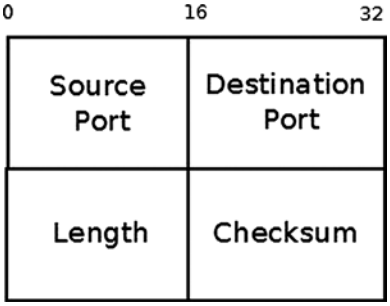

Figure 11-1. *A UDP header (IPv4)*

In this section, you learned about the UDP header and its members. To understand how the userspace applications, which use the sockets API, communicate with the kernel (sending and receiving packets), you should know about how UDP initialization is done, which is described in the next section.

UDP Initialization

We define the `udp_protocol` object (`net_protocol` object) and add it with the `inet_add_protocol()` method. This sets the `udp_protocol` object to be an element in the global protocols array (`inet_protos`).

```
static const struct net_protocol udp_protocol = {
        .handler =      udp_rcv,
        .err_handler =  udp_err,
        .no_policy =    1,
        .netns_ok =     1,
};
(net/ipv4/af_inet.c)
```

```
        /* For an inbound chunk, this tells us where it came from.
         * For an outbound chunk, it tells us where we'd like it to
         * go.  It is NULL if we have no preference.
         */
        struct sctp_transport *transport;

};
```

(include/net/sctp/structs.h)

We will now describe an SCTP association (which is the counterpart of a TCP connection).

SCTP Associations

In SCTP, we use the term *association* instead of a *connection*; a connection refers to communication between two IP addresses, whereas association refers to communication between two endpoints that might have multiple IP addresses. An SCTP association is represented by `struct sctp_association`:

```
struct sctp_association {
       ...

        sctp_assoc_t assoc_id;

        /* These are those association elements needed in the cookie.  */
        struct sctp_cookie c;

        /* This is all information about our peer.  */
        struct {
                struct list_head transport_addr_list;

                . . .
                __u16 transport_count;
                __u16 port;
                . . .

                struct sctp_transport *primary_path;
                struct sctp_transport *active_path;

        } peer;

        sctp_state_t state;
        . . .
        struct sctp_priv_assoc_stats stats;
};
```

(include/net/sctp/structs.h).

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

- `assoc_id`: The association unique id. It's set by the `sctp_assoc_set_id()` method.
- `c`: The state cookie (`sctp_cookie` object) that is attached to the association.

- `peer`: An inner structure representing the peer endpoint of the association. Adding a peer is done by the `sctp_assoc_add_peer()` method; removing a peer is done by the `sctp_assoc_rm_peer()` method. Following is a description of some of the `peer` structure important members:
  - `transport_addr_list`: Represents one or more addresses of the peer. We can add addresses to this list or remove addresses from it by using the `sctp_connectx()` method when an association is established.
  - `transport_count`: The counter of the peer addresses in the peer address list (`transport_addr_list`).
  - `primary_path`: Represents the address to which the initial connection was made (INIT <--> INIT_ACK exchange). The association will attempt to always use the primary path if it is active.
  - `active_path`: The address of the peer that is currently used when sending data.
  - `state`: The state that the association is in, like SCTP_STATE_CLOSED or SCTP_STATE_ESTABLISHED. Various SCTP states are discussed later in this section.

Adding multiple local addresses to an SCTP association or removing multiple addresses from one can be done, for example, with the `sctp_bindx()` system call, in order to support the multihoming feature mentioned earlier. Every SCTP association includes a peer object, which represents the remote endpoint; the peer object includes a list of one or more addresses of the remote endpoint (`transport_addr_list`). We can add one or more addresses to this list by calling the `sctp_connectx()` system call when establishing an association. An SCTP association is created by the `sctp_association_new()` method and initialized by the `sctp_association_init()` method. At any given moment, an SCTP association can be in one of 8 states; thus, for example, when it is created, its state is SCTP_STATE_CLOSED. Later on, these states can change; see, for example, the "Setting Up an SCTP Association" section later in this chapter. These states are represented by the `sctp_state_t enum` (`include/net/sctp/constants.h`).

To send data between two endpoints, an initialization process must be completed. In this process, an SCTP association between these two endpoints is set; a cookie mechanism is used to provide protection against synchronization attacks. This process is discussed in the following section.

Setting Up an SCTP Association

The initialization process is a 4-way handshake that consists of the following steps:

- One endpoint ("A") sends an INIT chunk to the endpoint it wants to communicate with ("Z"). This chunk will include a locally generated Tag in the Initiate Tag field of the INIT chunk, and it will also include a verification tag (`vtag` in the SCTP header) with a value of 0 (zero).
- After sending the INIT chunk, the association enters the SCTP_STATE_COOKIE_WAIT state.
- The other endpoint ("Z") sends to "A" an INIT-ACK chunk as a reply. This chunk will include a locally generated Tag in the Initiate Tag field of the INIT-ACK chunk and the remote Initiate Tag as the verification tag (`vtag` in the SCTP header). "Z" should also generate a state cookie and send it with the INIT-ACK reply.
- When "A" receives the INIT-ACK chunk, it leaves the SCTP_STATE_COOKIE_WAIT state. "A" will use the remote Initiate Tag as the verification tag (`vtag` in the SCTP header) in all transmitted packets from now on. "A" will send the state cookie it received in a COOKIE ECHO chunk. "A" will enter the SCTP_STATE_COOKIE_ECHOED state.

- When "Z" receives the COOKIE ECHO chunk, it will build a TCB (Transmission Control Block). The TCB is a data structure containing connection information on either side of an SCTP connection. "Z" will further change its state to SCTP_STATE_ESTABLISHED and reply with a COOKIE ACK chunk. This is where the association is finally established on "Z" and, at this point, this association will use the saved tags.
- When "A" receives the COOKIE ACK, it will move from the SCTP_STATE_COOKIE_ECHOED state to the SCTP_STATE_ESTABLISHED state.

■ **Note** An endpoint might respond to an INIT, INIT ACK, or COOKIE ECHO chunk with an ABORT chunk when some mandatory parameters are missing, or when receiving invalid parameter values. The cause of the ABORT chunk should be specified in the reply.

Now that you have learned about SCTP associations and how they are created, you will see how SCTP packets are received with SCTP and how SCTP packets are sent.

Receiving Packets with SCTP

The main handler for receiving SCTP packets is the `sctp_rcv()` method, which gets an SKB as a single parameter (`net/sctp/input.c`). First some sanity checks are made (size, checksum, and so on). If everything is fine, we proceed to check whether this packet is an "Out of the Blue" (OOTB) packet. A packet is an OOTB packet if it is correctly formed (that is, no checksum error), but the receiver is not able to identify the SCTP association to which this packet belongs. (See section 8.4 in RFC 4960.) The OOTB packets are handled by the `sctp_rcv_ootb()` method, which iterates over all the chunks of the packet and takes an action according to the chunk type, as specified in the RFC. Thus, for example, an ABORT chunk is discarded. If this packet is not an OOTB packet, it is put into an SCTP `inqueue` by calling the `sctp_inq_push()` method and proceeds on its journey with the `sctp_assoc_bh_rcv()` method or with the `sctp_endpoint_bh_rcv()` method.

Sending Packets with SCTP

Writing to a userspace SCTP socket reaches the `sctp_sendmsg()` method (`net/sctp/socket.c`). The packet is passed to the lower layers by calling the `sctp_primitive_SEND()` method, which in turn calls the state machine callback, `sctp_do_sm()` (`net/sctp/sm_sideeffect.c`), with SCTP_ST_PRIMITIVE_SEND. The next stage is to call `sctp_side_effects()`, and eventually call the `sctp_packet_transmit()` method.

SCTP HEARTBEAT

The HEARTBEAT mechanism tests the connectivity of a transport or path by exchanging HEARTBEAT and HEARTBEAT-ACK SCTP packets. It declares the transport IP address to be down once it reaches the threshold of a nonreturned heartbeat acknowledgment. A HEARTBEAT chunk is sent every 30 seconds by default to monitor the reachability of an idle destination transport address. This time interval is configurable by setting `/proc/sys/net/sctp/hb_interval`. The default is 30000 milliseconds (30 seconds). Sending heartbeat chunks is performed by the `sctp_sf_sendbeat_8_3()` method. The reason for the `8_3` in the method name is that it refers to section 8.3 (Path Heartbeat) in RFC 4960. When an endpoint receives a HEARTBEAT chunk, it replies with a HEARTBEAT-ECHO chunk if it is in the SCTP_STATE_COOKIE_ECHOED state or the SCTP_STATE_ESTABLISHED state.

SCTP Multistreaming

Streams are unidirectional data flows within a single association. The number of Outbound Streams and the number of Inbound Streams are declared during the association setup (by the INIT chunk), and the streams are valid during the entire association lifetime. A userspace application can set the number of streams by creating an `sctp_initmsg` object and initializing its `sinit_num_ostreams` and `sinit_max_instreams`, and then calling the `setsockopt()` method with SCTP_INITMSG. Initialization of the number of streams can also be done with the `sendmsg()` system call. This, in turn, sets the corresponding fields in the `initmsg` object of the `sctp_sock` object. One of the biggest reasons streams were added was to remove the Head-of-Line blocking (HoL Blocking) condition. Head-of-line blocking is a performance-limiting phenomenon that occurs when a line of packets is held up by the first packet—for example, in multiple requests in HTTP pipelining. When working with SCTP Multistreaming, this problem does not exist because each stream is sequenced separately and guaranteed to be delivered in order. Thus, once one of the streams is blocked due to loss/congestion, the other streams might not be blocked and data will continue to be delivered. This is due to that one stream can be blocked while the other streams are not blocked,

---

■ **Note** Regarding using sockets for SCTP, I should mention the `lksctp-tools` project (`http://lksctp.sourceforge.net/`). This project provides a Linux userspace library for SCTP (`libsctp`), including C language header files (`netinet/sctp.h`), for accessing SCTP-specific application programming interfaces not provided by the standard sockets, and also some helper utilities around SCTP. I should also mention RFC 6458, "Sockets API Extensions for Stream Control Transmission Protocol (SCTP)," which describes a mapping of the Stream Control Transmission Protocol (SCTP) into the sockets API.

SCTP Multihoming

SCTP multihoming refers to having multiple IP addresses on both endpoints. One of the really nice features of SCTP is that endpoints are multihomed by default if the local ip address was specified as a wildcard. Also, there has been a lot of confusion about the multihoming feature because people expect that simply by binding to multiple addresses, the associations will end up being multihomed. This is not true because we implement only destination multihoming. In other words, both connected endpoints have to be multihomed for it to have true failover capability. If the local association knows about only a single destination address, there will be only one path and thus no multihoming.

With describing SCTP multihoming in this section, the SCTP part of this chapter has ended. In the next section, I will describe the DCCP protocol, which is the last transport protocol to be discussed in this chapter.

DCCP: The Datagram Congestion Control Protocol

DCCP is an unreliable, congestion-controlled transport layer protocol and, as such, it borrows from both UDP and TCP while adding new features. Like UDP, it is message-oriented and unreliable. Like TCP, it is a connection-oriented protocol and it also uses a 3-way handshake to set up the connection. Development of DCCP was helped by ideas from academia, through participation of several research institutes, but it has not been tested so far in larger-scale Internet setups. The use of DCCP would make sense, for instance, in applications that require minor delays and where a small degree of data loss is permitted, like in telephony and in streaming media applications.

Congestion control in DCCP differs from that in TCP in that the congestion-control algorithm (called CCID) can be negotiated between endpoints and congestion control can be applied on both the forward and reverse paths of a connection (called half-connections in DCCP). Two classes of pluggable congestion control have been specified so far. The first type is a rate-based, smooth "TCP-friendly" algorithm (CCID-3, RFC 4342 and 5348), for which there is an experimental small-packet variation called CCID-4 (RFC 5622, RFC 4828). The second type of congestion control,

"TCP-like" (RFC 4341) applies a basic TCP congestion-control algorithm with selective acknowledgments (SACK, RFC 2018) to DCCP flows. At least one CCID needs to be implemented by endpoints in order to function. The first DCCP Linux implementation was released in Linux kernel 2.6.14 (2005). This chapter describes the implementation principles of the DCCPv4 (IPv4). Delving into the implementation details of individual DCCP congestion-control algorithms is beyond the scope of this book.

Now that I've introduced the DCCP protocol in general, I will describe the DCCP header.

DCCP Header

Every DCCP packet starts with a DCCP header. The minimum DCCP header length is 12 bytes. DCCP uses a variable-length header, which can range from 12 to 1020 bytes, depending on whether short sequence numbers are used and which TLV packet options are used. DCCP sequence numbers are incremented for each packet (not per each byte as in TCP) and can be shortened from 6 to 3 bytes.

```
struct dccp_hdr {
        __be16  dccph_sport,
                dccph_dport;
        __u8    dccph_doff;
#if defined(__LITTLE_ENDIAN_BITFIELD)
        __u8    dccph_cscov:4,
                dccph_ccval:4;
#elif defined(__BIG_ENDIAN_BITFIELD)
        __u8    dccph_ccval:4,
                dccph_cscov:4;
#else
#error  "Adjust your <asm/byteorder.h> defines"
#endif
        __sum16 dccph_checksum;
#if defined(__LITTLE_ENDIAN_BITFIELD)
        __u8    dccph_x:1,
                dccph_type:4,
                dccph_reserved:3;
#elif defined(__BIG_ENDIAN_BITFIELD)
        __u8    dccph_reserved:3,
                dccph_type:4,
                dccph_x:1;
#else
#error  "Adjust your <asm/byteorder.h> defines"
#endif
        __u8    dccph_seq2;
        __be16  dccph_seq;
};

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

The following is a description of the important members of the `dccp_hdr` structure:

- `dccph_sport`: Source port (16 bit).
- `dccph_dport`: Destination port (16 bit).
- `dccph_doff`: Data offset (8 bits). The size of the DCCP header is in multiples of 4 bytes.

- `dccph_cscov`: Determines which part of the packet is covered in the checksum. Using partial checksumming might improve performance when it is used with applications that can tolerate corruption of some low percentage.
- `dccph_ccval`: CCID-specific information from sender to receiver (not always used).
- `dccph_x`: Extended Sequence Numbers bit (1 bit). This flag is set when using 48-bit Extended Sequence and Acknowledgment Numbers.
- `dccph_type`: The DCCP header type (4 bits). This can be, for example, DCCP_PKT_DATA for a data packet or DCCP_PKT_ACK for an ACK. See Table 11-3, "DCCP packet types," in the "Quick Reference" section at the end of this chapter.
- `dccph_reserved`: Reserved for future use (1 bit).
- `dccph_checksum`: The checksum (16 bit). The Internet checksum of the DCCP header and data, computed similarly to UDP and TCP. If partial checksums are used, only the length specified by `dccph_cscov` of the application data is checksummed.
- `dccph_seq2`: Sequence number. This is used when working with Extended Sequence Numbers (8 bit).
- `dccph_seq`: Sequence number. It is incremented by 1 for each packet (16 bit).

■ **Note** DCCP sequence numbers depend on `dccph_x`. (For details, refer to the `dccp_hdr_seq()` method, `include/linux/dccp.h`).

Figure 11-4 shows a DCCP header. The `dccph_x` flag is set, so we use 48-bit Extended Sequence numbers.

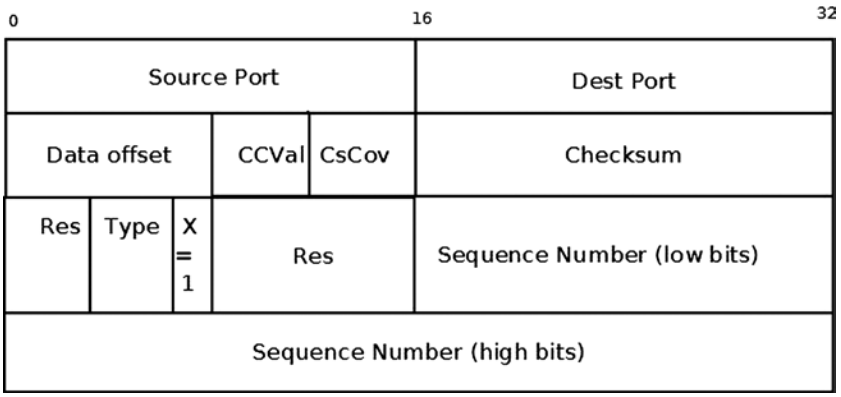

Figure 11-4. *DCCP header (the Extended Sequence Numbers bit is set, dccph_x=1)*

Figure 11-5 shows a DCCP header. The dccph_x flag is not set, so we use 24-bit Sequence numbers.

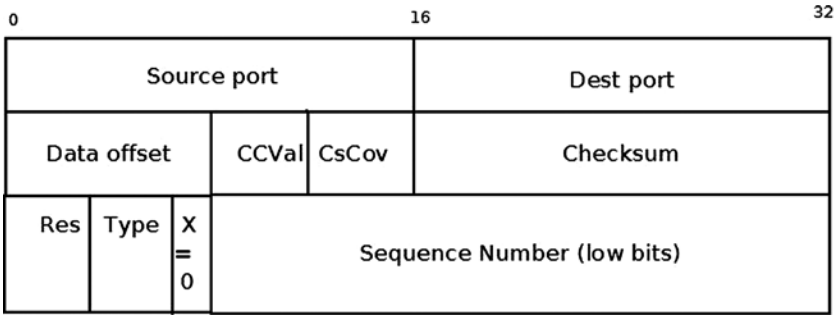

Figure 11-5. *DCCP header (the Extended Sequence Numbers bit is not set, dccph_x=0)*

DCCP Initialization

DCCP initialization happens much like in TCP and UDP. Considering the DCCPv4 case (net/dccp/ipv4.c), first a proto object is defined (dccp_v4_prot) and its DCCP specific callbacks are set; we also define a net_protocol object (dccp_v4_protocol) and initialize it:

```
static struct proto dccp_v4_prot = {
         .name                   = "DCCP",
         .owner                  = THIS_MODULE,
         .close                  = dccp_close,
         .connect                = dccp_v4_connect,
         .disconnect             = dccp_disconnect,
         .ioctl                  = dccp_ioctl,
         .init                   = dccp_v4_init_sock,
         . . .
         .sendmsg                = dccp_sendmsg,
         .recvmsg                = dccp_recvmsg,
         . . .

}

(net/dccp/ipv4.c)


static const struct net_protocol dccp_v4_protocol = {
        .handler        = dccp_v4_rcv,
        .err_handler    = dccp_v4_err,
        .no_policy      = 1,
        .netns_ok       = 1,
};

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

We register the dccp_v4_prot object and the dccp_v4_protocol object in the dccp_v4_init() method:

```
static int __init dccp_v4_init(void)
{
         int err = proto_register(&dccp_v4_prot, 1);

         if (err != 0)
                 goto out;

         err = inet_add_protocol(&dccp_v4_protocol, IPPROTO_DCCP);
         if (err != 0)
                 goto out_proto_unregister;
(net/dccp/ipv4.c)
```

DCCP Socket Initialization

Socket creation in DCCP from userspace uses the socket() system call, where the domain argument (SOCK_DCCP) indicates that a DCCP socket is to be created. Within the kernel, this causes DCCP socket initialization via the dccp_v4_init_sock() callback, which relies on the dccp_init_sock() method to perform the actual work:

```
static int dccp_v4_init_sock(struct sock *sk)
{
        static __u8 dccp_v4_ctl_sock_initialized;
        int err = dccp_init_sock(sk, dccp_v4_ctl_sock_initialized);

        if (err == 0) {
                if (unlikely(!dccp_v4_ctl_sock_initialized))
                        dccp_v4_ctl_sock_initialized = 1;
                inet_csk(sk)->icsk_af_ops = &dccp_ipv4_af_ops;
        }

        return err;
}

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

The most important tasks of the dccp_init_sock() method are these:

- Initialization of the DCCP socket fields with sane default values (for example, the socket state is set to be DCCP_CLOSED)
- Initialization of the DCCP timers (via the dccp_init_xmit_timers() method)
- Initialization of the feature-negotiation part via calling the dccp_feat_init() method. Feature negotiation is a distinguishing feature of DCCP by which endpoints can mutually agree on properties of each side of the connection. It extends TCP feature negotiation and is described further in RFC 4340, sec. 6.

Receiving Packets from the Network Layer (L3) with DCCP

The main handler for receiving DCCP packets from the network layer (L3) is the `dccp_v4_rcv ()` method:

```
static int dccp_v4_rcv(struct sk_buff *skb)
{
        const struct dccp_hdr *dh;
        const struct iphdr *iph;
        struct sock *sk;
        int min_cov;
```

First we discard invalid packets. For example, if the packet is not for this host (the packet type is not PACKET_HOST), or if the packet size is shorter than the DCCP header (which is 12 bytes):

```
        if (dccp_invalid_packet(skb))
                goto discard_it;
```

Then we perform a lookup according to the flow:

```
        sk = __inet_lookup_skb(&dccp_hashinfo, skb,
                               dh->dccph_sport, dh->dccph_dport);
```

If no socket was found, the packet is dropped:

```
        if (sk == NULL) {
               . . .
               goto no_dccp_socket;
        }
```

We make some more checks relating to Minimum Checksum Coverage, and if everything is fine, we proceed to the generic `sk_receive_skb()` method to pass the packet to the transport layer (L4). Note that the `dccp_v4_rcv()` method is very similar in structure and function to the `tcp_v4_rcv()` method. This is because the original author of DCCP in Linux, Arnaldo Carvalho de Melo, has worked quite hard to make the similarities between TCP and DCCP obvious and clear in the code.

```
         . . .
         return sk_receive_skb(sk, skb, 1);
         }
```

`(net/dccp/ipv4.c)`

Sending Packets with DCCP

Sending data from a DCCP userspace socket is eventually handled by the `dccp_sendmsg()` method in the kernel (`net/dccp/proto.c`). This parallels the TCP case, where the `tcp_sendmsg()` kernel method handles sending data from a TCP userspace socket. Let's take a look at the `dccp_sendmsg()` method:

```
int dccp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
                 size_t len)
{
         const struct dccp_sock *dp = dccp_sk(sk);
         const int flags = msg->msg_flags;
```

```
const int noblock = flags & MSG_DONTWAIT;
struct sk_buff *skb;
int rc, size;
long timeo;
```

Allocate an SKB:

```
skb = sock_alloc_send_skb(sk, size, noblock, &rc);
lock_sock(sk);
if (skb == NULL)
        goto out_release;

skb_reserve(skb, sk->sk_prot->max_header);
```

Copy the data blocks from the `msghdr` object to the SKB:

```
rc = memcpy_fromiovec(skb_put(skb, len), msg->msg_iov, len);
if (rc != 0)
        goto out_discard;

if (!timer_pending(&dp->dccps_xmit_timer))
        dccp_write_xmit(sk);
```

Depending upon the type of congestion control (window-based or rate-based) chosen for the connection, the `dccp_write_xmit()` method will cause a packet to be sent later (via `dccps_xmit_timer()` expiry) or passed on for immediate sending by the `dccp_xmit_packet()` method. This, in turn, relies on the `dccp_transmit_skb()` method to initialize the outgoing DCCP header and pass it to the L3-specific `queue_xmit` sending callback (using the `ip_queue_xmit()` method for IPv4, and the `inet6_csk_xmit()` method for IPv6). I will conclude our discussion about DCCP with a short section about DCCP and NAT.

DCCP and NAT

Some NAT devices do not let DCCP through (usually because their firmware is typically small, and hence does not support "exotic" IP protocols such as DCCP). RFC 5597 (September 2009) has suggested behavioral requirements for NATs to support NAT-ed DCCP communications. However, it is not clear to what extent the recommendations are put into consumer devices. One of the motivations for DCCP-UDP was the absence of NAT devices that would let DCCP through (RFC 6773, sec. 1). There is a detail that might be interesting in the comparison with TCP. The latter, by default, supports simultaneous open (RFC 793, section 3.4), whereas the initial specification of DCCP in RFC 4340, section 4.6 disallowed the use of simultaneous-open. To support NAPT traversal, RFC 5596 updated RFC 4340 in September 2009 with a "near simultaneous open" technique, which added one packet type (DCCP-LISTEN, RFC 5596, section 2.2.1) to the list and changed the state machine to support two more states (2.2.2) to support near-simultaneous open. The motivation was a NAT "hole punching" technique, which would require, however, that NATs with DCCP existed (same problem as above). As a result of this chicken-and-egg problem, DCCP has not seen much exposure over the Internet. Perhaps the UDP encapsulation will change that. But then it would no longer really be considered as a transport layer protocol.

Summary

This chapter discussed four transport protocols: UDP and TCP, which are the most commonly used, and SCTP and DCCP, which are newer protocols. You learned the basic differences between these protocols. You learned that TCP is a much more complex protocol than UDP, as its uses a state machine and several timers and requires acknowledgments. You learned about the header of each of these protocols and about sending and receiving packets with these protocols. I discussed some unique features of the SCTP protocol, like multihoming and multistreaming.

The next chapter will deal with the Wireless subsystem and its implementation in Linux. In the "Quick Reference" section that follows, I will cover the top methods related to the topics discussed in this chapter, ordered by their context, and also I will present the two tables that were mentioned in this chapter.

Quick Reference

I will conclude this chapter with a short list of important methods of sockets and transport-layer protocols that we discussed in this chapter. Some of them were mentioned in this chapter. Afterward, there is one macro and three tables.

Methods

Here are the methods.

int ip_cmsg_send(struct net *net, struct msghdr *msg, struct ipcm_cookie *ipc);

This method builds an `ipcm_cookie` object by parsing the specified `msghdr` object.

void sock_put(struct sock *sk);

This method decrements the reference count of the specified `sock` object.

void sock_hold(struct sock *sk);

This method increments the reference count of the specified `sock` object.

int sock_create(int family, int type, int protocol, struct socket **res);

This method performs some sanity checks, and if everything is fine, it allocates a socket by calling the `sock_alloc()` method, and then calling `net_families[family]->create`. (In the case of IPv4, it is the `inet_create()` method.)

int sock_map_fd(struct socket *sock, int flags);

This method allocates a file descriptor and fills in the file entry.

bool sock_flag(const struct sock *sk, enum sock_flags flag);

This method returns `true` if the specified `flag` is set in the specified `sock` object.

int tcp_v4_rcv(struct sk_buff *skb);

This method is the main handler to process incoming TCP packets arriving from the network layer (L3).

void tcp_init_sock(struct sock *sk);

This method performs address-family independent socket initializations.

struct tcphdr *tcp_hdr(const struct sk_buff *skb);

This method returns the TCP header associated with the specified `skb`.

int tcp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg, size_t size);

This method handles sending TCP packets that are sent from userspace.

struct tcp_sock *tcp_sk(const struct sock *sk);

This method returns the `tcp_sock` object associated with the specified sock object (`sk`).

int udp_rcv(struct sk_buff *skb);

This method is the main handler to process incoming UDP packets arriving from the network layer (L3).

struct udphdr *udp_hdr(const struct sk_buff *skb);

This method returns the UDP header associated with the specified `skb`.

int udp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg, size_t len);

This method handles UDP packets that are sent from the userspace.

struct sctphdr *sctp_hdr(const struct sk_buff *skb);

This method returns the SCTP header associated with the specified `skb`.

struct sctp_sock *sctp_sk(const struct sock *sk);

This method returns the SCTP socket (`sctp_sock` object) associated with the specified `sock` object.

int sctp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg, size_t msg_len);

This method handles SCTP packets that are sent from userspace.

struct sctp_association *sctp_association_new(const struct sctp_endpoint *ep, const struct sock *sk, sctp_scope_t scope, gfp_t gfp);

This method allocates and initializes a new SCTP association.

void sctp_association_free(struct sctp_association *asoc);

This method frees the resources of an SCTP association.

void sctp_chunk_hold(struct sctp_chunk *ch);

This method increments the reference count of the specified SCTP chunk.

void sctp_chunk_put(struct sctp_chunk *ch);

This method decrements the reference count of the specified SCTP chunk. If the reference count reaches 0, it frees it by calling the `sctp_chunk_destroy()` method.

int sctp_rcv(struct sk_buff *skb);

This method is the main input handler for input SCTP packets.

static int dccp_v4_rcv(struct sk_buff *skb);

This method is the main Rx handler for processing incoming DCCP packets that arrive from the network layer (L3).

int dccp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg, size_t len);

This method handles DCCP packets that are sent from the userspace.

Macros

And here is the macro.

sctp_chunk_is_data()

This macro returns 1 if the specified chunk is a data chunk; otherwise, it returns 0.

Tables

Take a look at the tables used in this chapter.

Table 11-1. *TCP and UDP prot_ops objects*

prot_ops callback	TCP	UDP
release	inet_release	inet_release
bind	inet_bind	inet_bind
connect	inet_stream_connect	inet_dgram_connect
socketpair	sock_no_socketpair	sock_no_socketpair
accept	inet_accept	sock_no_accept
getname	inet_getname	inet_getname
poll	tcp_poll	udp_poll
ioctl	inet_ioctl	inet_ioctl
listen	inet_listen	sock_no_listen
shutdown	inet_shutdown	inet_shutdown
setsockopt	sock_common_setsockopt	sock_common_setsockopt
getsockopt	sock_common_getsockopt	sock_common_getsockopt
sendmsg	inet_sendmsg	inet_sendmsg
recvmsg	inet_recvmsg	inet_recvmsg
mmap	sock_no_mmap	sock_no_mmap
sendpage	inet_sendpage	inet_sendpage
splice_read	tcp_splice_read	-
compat_setsockopt	compat_sock_common_setsockopt	compat_sock_common_setsockopt
compat_getsockopt	compat_sock_common_getsockopt	compat_sock_common_getsockopt
compat_ioctl	inet_compat_ioctl	inet_compat_ioctl

■ **Note** See the `inet_stream_ops` and the `inet_dgram_ops` definitions in `net/ipv4/af_inet.c`.

Table 11-2. *Chunk types*

Chunk Type	Linux Symbol	Value
Payload Data	SCTP_CID_DATA	0
Initiation	SCTP_CID_INIT	1
Initiation Acknowledgment	SCTP_CID_INIT_ACK	2
Selective Acknowledgment	SCTP_CID_SACK	3
Heartbeat Request	SCTP_CID_HEARTBEAT	4

(*continued*)

Table 11-2. (*continued*)

Chunk Type	Linux Symbol	Value
Heartbeat Acknowledgment	SCTP_CID_HEARTBEAT_ACK	5
Abort	SCTP_CID_ABORT	6
Shutdown	SCTP_CID_SHUTDOWN	7
Shutdown Acknowledgment	SCTP_CID_SHUTDOWN_ACK	8
Operation Error	SCTP_CID_ERROR	9
State Cookie	SCTP_CID_COOKIE_ECHO	10
Cookie Acknowledgment	SCTP_CID_COOKIE_ACK	11
Explicit Congestion Notification Echo (ECNE)	SCTP_CID_ECN_ECNE	12
Congestion Window Reduced (CWR)	SCTP_CID_ECN_CWR	13
Shutdown Complete	SCTP_CID_SHUTDOWN_COMPLETE	14
SCTP Authentication Chunk (RFC 4895)	SCTP_CID_AUTH	0x0F
Transmission Sequence Numbers	SCTP_CID_FWD_TSN	0xC0
Address Configuration Change Chunk	SCTP_CID_ASCONF	0xC1
Address Configuration Acknowledgment Chunk	SCTP_CID_ASCONF_ACK	0x80

Table 11-3. *DCCP packet types*

Linux Symbol	Description
DCCP_PKT_REQUEST	Sent by the client to initiate a connection (the first part of the three-way initiation handshake).
DCCP_PKT_RESPONSE	Sent by the server in response to a DCCP-Request (the second part of the three-way initiation handshake).
DCCP_PKT_DATA	Used to transmit application data.
DCCP_PKT_ACK	Used to transmit pure acknowledgments.
DCCP_PKT_DATAACK	Used to transmit application data with piggybacked acknowledgment information.
DCCP_PKT_CLOSEREQ	Sent by the server to request that the client close the connection.
DCCP_PKT_CLOSE	Used by the client or the server to close the connection; elicits a DCCP-Reset packet in response.
DCCP_PKT_RESET	Used to terminate the connection, either normally or abnormally.
DCCP_PKT_SYNC	Used to resynchronize sequence numbers after large bursts of packet loss.
DCCP_PKT_SYNCACK	Acknowledge a DCCP_PKT_SYNC.

CHAPTER 12

Wireless in Linux

Chapter 11 deals with Layer 4 protocols, which enable us to communicate with userspace. This chapter deals with the wireless stack in the Linux kernel. I describe the Linux wireless stack (mac80211 subsystem) and discuss some implementation details of important mechanisms in it, such as packet aggregation and block acknowledgement, used in IEEE 802.11n, and power save mode. Becoming familiar with the 802.11 MAC header is essential in order to understand the wireless subsystem implementation. The 802.11 MAC header, its members, and their usage are described in depth in this chapter. I also discuss some common wireless topologies, like infrastructure BSS, independent BSS, and Mesh networking.

Mac80211 Subsystem

At the end of the 1990s, there were discussions in IEEE regarding a protocol for wireless local area networks (WLANS). The original version of the IEEE 802.11 spec for WLANS was released in 1997 and revised in 1999. In the following years, some extensions were added, formally termed 802.11 amendments. These extensions can be divided into PHY (Physical) layer extensions, MAC (Medium Access Control) layer extensions, Regulatory extensions, and others. PHY layer extensions are, for example, 802.11b from 1999, 802.11a (also from 1999), and 802.11g from 2003. MAC layer extensions are, for example, 802.11e for QoS and 802.11s for Mesh networking. The "Mesh Networking" section of this chapter deals with the Linux kernel implementation of the IEEE802.11s amendment. The IEEE802.11 spec was revised, and in 2007 a second version of 1,232 pages was released. In 2012, a spec of 2,793 pages was released, available from http://standards.ieee.org/findstds/standard/802.11-2012.html. I refer to this spec as IEEE 802.11-2012 in this chapter. Following is a partial list of important 802.11 amendments:

- *IEEE 802.11d:* International (country-to-country) roaming extensions (2001).
- *IEEE 802.11e:* Enhancements: QoS, including packet bursting (2005).
- *IEEE 802.11h:* Spectrum Managed 802.11a for European compatibility (2004).
- *IEEE 802.11i:* Enhanced security (2004).
- *IEEE 802.11j:* Extensions for Japan (2004).
- *IEEE 802.11k:* Radio resource measurement enhancements (2008).
- *IEEE 802.11n:* Higher throughput improvements using MIMO (multiple input, multiple output antennas) (2009).
- *IEEE 802.11p:* WAVE: Wireless Access for the Vehicular Environment (such as ambulances and passenger cars). It has some peculiarities such as not using the BSS concept and narrower (5/10 MHz) channels. Note that IEEE 802.11p isn't supported in Linux as of this writing.
- *IEEE 802.11v:* Wireless network management.

- *IEEE 802.11w:* Protected Management Frames.
- *IEEE 802.11y:* 3650–3700 MHz operation in the U.S. (2008)
- *IEEE 802.11z:* Extensions to Direct Link Setup (DLS) (Aug 2007–Dec 2011).

It was only in about 2001, about four years after the IEEE 802.11 first spec was approved, that laptops became very popular; many of these laptops were sold with wireless network interfaces. Today every laptop includes WiFi as standard equipment. It was important to the Linux community at that time to provide Linux drivers to these wireless network interfaces and to provide a Linux network wireless stack, in order to stay competitive with other OSes (such as Windows, Mac OS, and others). Less effort has been done regarding architecture and design. "They just want their hardware to work," as Jeff Garzik, the Linux Kernel Wireless maintainer at that time, put it. When the first wireless drivers for Linux were developed, there was no general wireless API. As a result, there were many cases of duplication of code between drivers, when developers implemented their drivers from scratch. Some drivers were based on FullMAC, which means that most of the management layer (MLME) is managed in hardware. In the years since, a new 802.11 wireless stack called mac80211 was developed. It was integrated into the Linux kernel in July 2007, for the 2.6.22 Linux kernel. The mac80211 stack is based on the d80211 stack, which is an open source, GPL-licensed stack by a company named Devicescape.

I cannot delve into the details of the PHY layer, because that subject is very wide and deserves a book of its own. However, I must note that there are many differences between 802.11 and 802.3 wired Ethernet. Here are two major differences:

- Ethernet works with CSMA/CD, whereas 802.11 works with CSMA/CA. CSMA/CA stands for carrier sense multiple access/collision avoidance, and CSMA/CD stands for carrier sense multiple access/collision detection. The difference, as you might guess, is the collision detection. With Ethernet, a station starts to transmit when the medium is idle; if a collision is detected during transmission, it stops, and a random backoff period starts. Wireless stations cannot detect collisions while transmitting, whereas wired stations can. With CSMA/CA, the wireless station waits for a free medium and only then transmits the frame. In case of a collision, the station will not notice it, but because no acknowledgment frame should be sent for this packet, it is retransmitted after a timeout has elapsed if an acknowledgment is not received.
- Wireless traffic is sensitive to interferences. As a result, the 802.11 spec requires that every frame, except for broadcast and multicast, be acknowledged when it is received. Packets that are not acknowledged in time should be retransmitted. Note that since IEEE 802.11e, there is a mode which does not require acknowledgement—the QoSNoAck mode—but it's rarely used in practice.

The 802.11 MAC Header

Each MAC frame consists of a MAC header, a frame body of variable length, and an FCS (Frame Check Sequence) of 32 bit CRC. Figure 12-1 shows the 802.11 header.

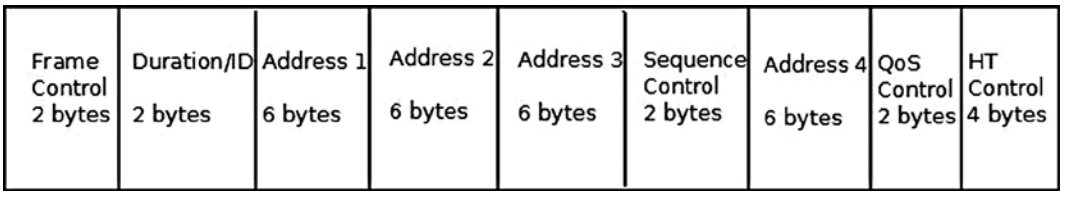

Figure 12-1. *IEEE 802.11 header. Note that all members are not always used, as this section will shortly explain*

The 802.11 header is represented in mac80211 by the `ieee80211_hdr` structure:

```
struct ieee80211_hdr {
        __le16 frame_control;
        __le16 duration_id;
        u8 addr1[6];
        u8 addr2[6];
        u8 addr3[6];
        __le16 seq_ctrl;
        u8 addr4[6];
} __packed;
```

(`include/linux/ieee80211.h`)

In contrast to an Ethernet header (`struct ethhdr`), which contains only three fields (source MAC address, destination MAC address, and Ethertype), the 802.11 header contains up to six addresses and some other fields. For a typical data frame, though, only three addresses are used (for example, Access Point or AP/client communication). With an ACK frame, only the receiver address is used. Note that Figure 12-1 shows only four addresses, but when working with Mesh networking, a Mesh extension header with two additional addresses is used.

I now turn to a description of the 802.11 header fields, starting with the first field in the 802.11 header, called the *frame control*. This is an important field, and in many cases its contents determine the meaning of other fields of the 802.11 MAC header (especially addresses).

The Frame Control

The frame control length is 16 bits. Figure 12-2 shows its fields and the size of each field.

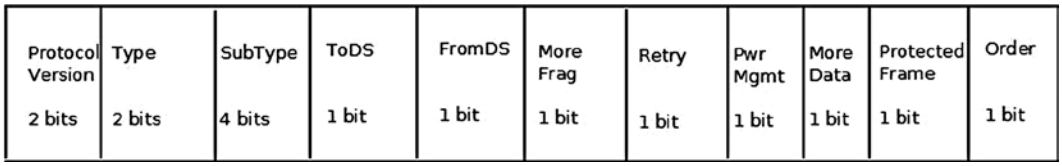

Figure 12-2. *Frame control fields*

The following is a description of the frame control members:

- `Protocol version`: The version of the MAC 802.11 we use. Currently there is only one version of MAC, so this field is always 0.
- `Type`: There are three types of packets in 802.11—management, control, and data:
  - Management packets (IEEE80211_FTYPE_MGMT) are for management actions like association, authentication, scanning, and more.
  - Control packets (IEEE80211_FTYPE_CTL) usually have some relevance to data packets; for example, a PS-Poll packet is for retrieving packets from an AP buffer. Another example: a station that wants to transmit first sends a control packet named RTS (request to send); if the medium is free, the destination station will send back a control packet named CTS (clear to send).
  - Data packets (IEEE80211_FTYPE_DATA) are the raw data packets. Null packets are a special case of raw packets, carrying no data and used mostly for power management control purposes. I discuss null packets in the "Power Save Mode" section later in this chapter.

- `Subtype`: For all the aforementioned three types of packets (management, control, and data), there is a sub-type field which identifies the character of the packet used. For example:
  - A value of 0100 for the sub-type field in a management frame denotes that the packet is a Probe Request (IEEE80211_STYPE_PROBE_REQ) management packet, which is used in a scan operation.
  - A value of 1011 for the sub-type field in a control packet denotes that this is a request to send (IEEE80211_STYPE_RTS) control packet. A value of 0100 for the sub-type field of a data packet denotes that this is a null data (IEEE80211_STYPE_NULLFUNC) packet, which is used for power management control.
  - A value of 1000 (IEEE80211_STYPE_QOS_DATA) for the sub-type of a data packet means that this is a QoS data packet; this sub-type was added by the IEEE802.11e amendment, which dealt with QoS enhancements.
- `ToDS`: When this bit is set, it means the packet is for the distribution system.
- `FromDS`: When this bit is set, it means the packet is from the distribution system.
- `More Frag`: When you use fragmentation, this bit is set to 1.
- `Retry`: When a packet is retransmitted, this bit is set to 1. A typical case of retransmission is when a packet that was sent did not receive an acknowledgment in time. The acknowledgments are usually sent by the firmware of the wireless driver.
- `Pwr Mgmt`: When the power management bit is set, it means that the station will enter power save mode. I discuss power save mode in the "Power Save Mode" section later in this chapter.
- `More Data`: When an AP sends packets that it buffered for a sleeping station, it sets the `More Data` bit to 1 when the buffer is not empty. Thus the station knows that there are more packets it should retrieve. When the buffer has been emptied, this bit is set to 0.
- `Protected Frame`: This bit is set to 1 when the frame body is encrypted; only data frames and authentication frames can be encrypted.
- `Order`: With the MAC service called strict ordering, the order of frames is important. When this service is in use, the order bit is set to 1. It is rarely used.

■ **Note** The action frame (IEEE80211_STYPE_ACTION) was introduced with the 802.11h amendment, which dealt with spectrum and transmit power management. However, because of a lack of space for management packets sub-types, action frames are used also in various newer amendments to the standard—for example, HT action frames in 802.11n.

The Other 802.11 MAC Header Members

The following describes the other members of the mac802.11 header, after the frame control:

- `Duration/ID`: The duration holds values for the Network Allocation Vector (NAV) in microseconds, and it consists of 15 bits of the `Duration/ID` field. The sixteenth field is 0. When working in power save mode, it is the AID (association id) of a station for PS-Poll frames (see 8.2.4.2 (a) in IEEE 802.11-2012). The Network Allocation Vector (NAV) is a virtual carrier sensing mechanism. I do not delve into NAV internals because that is beyond the scope of this chapter.

- `Sequence Control`: This is a 2-byte field specifying the sequence control. In 802.11, it is possible that a packet will be received more than once, most commonly when an acknowledgment is not received for some reason. The sequence control field consists of a fragment number (4 bits) and a sequence number (12 bits). The sequence number is generated by the transmitting station, in the `ieee80211_tx_h_sequence()` method. In the case of a duplicate frame in a retransmission, it is dropped, and a counter of the dropped duplicate frames (`dot11FrameDuplicateCount`) is incremented by 1; this is done in the `ieee80211_rx_h_check()` method. The `Sequence Control` field is not present in control packets.
- `Address1 – Address4`: There are four addresses, but you don't always use all of them. Address 1 is the Receive Address (RA), and is used in all packets. Address 2 is the Transmit Address (TA), and it exists in all packets except ACK and CTS packets. Address 3 is used only for management and data packets. Address 4 is used when ToDS and FromDS bits of the frame control are set; this happens when operating in a Wireless Distribution System.
- `QoS Control`: The QoS control field was added by the 802.11e amendment and is only present in QoS data packets. Because it is not part of the original 802.11 spec, it is not part of the original mac80211 implementation, so it is not a member of the IEEE802.11 header (`ieee80211_hdr struct`). In fact, it was added at the end of the IEEE802.11 header and can be accessed by the `ieee80211_get_qos_ctl()` method. The QoS control field includes the `tid` (Traffic Identification), the ACK Policy, and a field called A-MSDU present, which tells whether an A-MSDU is present. I discuss A-MSDU later in this chapter, in the "High Throughput (ieee802.11n)" section.
- HT Control Field: HT (high throughput) control field was added by the 802.11n amendment (see 7.1.3.5(a) of the 802.11n-2009 spec).

This section covered the 802.11 MAC header, with a description of its members and their use. Becoming familiar with the 802.11 MAC header is essential for understanding the mac802.11 stack.

Network Topologies

There are two popular network topologies in 802.11 wireless networks. The first topology I discuss is *Infrastructure BSS* mode, which is the most popular. You encounter Infrastructure BSS wireless networks in home wireless networks and offices. Later I discuss the IBSS (Ad Hoc) mode. Note that IBSS is *not* Infrastructure BSS; IBSS is *Independent BSS*, which is an ad hoc network, discussed later in this section.

Infrastructure BSS

When working in Infrastructure BSS mode, there is a central device, called an Access Point (AP), and some client stations. Together they form a BSS (Basic Service Set). These client stations must first perform association and authentication against the AP to be able to transmit packets via the AP. On many occasions, client stations perform scanning prior to authentication and association, in order to get details about the AP. Association is exclusive: a client can be associated with only one AP in a given moment. When a client associates with an AP successfully, it gets an AID (association id), which is a unique number (to this BSS) in the range 1–2007. An AP is in fact a wireless network device with some hardware additions (like Ethernet ports, LEDs, a button to reset to manufacturer defaults, and more). A management daemon runs on the AP device. An example of such software is the `hostapd` daemon. This software handles some of the management tasks of the MLME layer, such as authentication and association requests. It achieves this by registering itself to receive the relevant management frames via `nl80211`. The `hostapd` project is an open source project which enables several wireless network devices to operate as an AP.

Clients can communicate with other clients (or to stations in a different network which is bridged to the AP) by sending packets to the AP, which are relayed by the AP to their final destination. To cover a large area, you can deploy multiple APs and connect them by wire. This type of deployment is called Extended Service Set (ESS). Within ESS deployment, there are two or more BSSs. Multicasts and broadcasts sent in one BSS, which may arrive on a nearby BSS, are rejected in the nearby BSS stations (the `bssid` in the 802.11 header does not match). Within such a deployment, each AP usually uses a different channel to minimize interference.

IBSS, or Ad Hoc Mode

IBSS network is often formed without preplanning, for only as long as the WLAN is needed. An IBSS network is also called ad hoc network. Creating an IBSS is a simple procedure. You can set an IBSS by running from a command line this `iw` command (note that the 2412 parameter is for using channel 1):

```
iw wlan0 ibss join AdHocNetworkName 2412
```

Or when using the `iwconfig` tool, with these two commands:

```
iwconfig wlan0 mode ad-hoc
iwconfig wlan0 essid AdHocNetworkrName
```

This triggers IBSS creation by calling the `ieee80211_sta_create_ibss()` method (`net/mac80211/ibss.c`). Then the `ssid` (`AdHocNetworkName` in this case) has to be distributed manually (or otherwise) to everyone who wants to connect to the ad hoc network. When working with IBSS, you do not have an AP. The bssid of the IBSS is a random 48-bit address (based on calling the `get_random_bytes()` method). Power management in Ad Hoc mode is a bit more complex than power management in Infrastructure BSS; it uses Announcement Traffic Indication Map (ATIM) messages. ATIM is not supported by mac802.11 and is not discussed in this chapter.

The next section describes power save mode, which is one of the most important mechanisms of the mac80211 network stack.

Power Save Mode

Apart from relaying packets, there is another important function for the AP: buffering packets for client stations that enter power save mode. Clients are usually battery-powered devices. From time to time, the wireless network interface enters power save mode.

Entering Power Save Mode

When a client station enters power save mode, it informs the AP about it by sending usually a null data packet. In fact, technically speaking, it does not have to be a null data packet; it is enough that it is a packet with PM=1 (PM is the Power Management flag in the frame control). An AP that gets such a null packet starts keeping unicast packets which are destined to that station in a special buffer called `ps_tx_buf`; there is such a buffer for every station. This buffer is in fact a linked list of packets, and it can hold up to 128 packets (STA_MAX_TX_BUFFER) for each station. If the buffer is filled, it will start discarding the packets that were received first (FIFO). Apart from this, there is a single buffer called `bc_buf`, for multicast and broadcast packets (in the 802.11 stack, multicast packets should be received and processed by all the stations in the same BSS). The `bc_buf` buffer can also hold up to 128 packets (AP_MAX_BC_BUFFER). When a wireless network interface is in power save mode, it cannot receive or send packets.

Exiting Power Save Mode

From time to time, an associated station is awakened by itself (by some timer); it then checks for special management packets, called *beacons*, which the AP sends periodically. Typically, an AP sends 10 beacons in a second; on most APs, this is a configurable parameter. These beacons contain data in *information elements*, which constitute the data in the management packet. The station that awoke checks a specific information element called TIM (Traffic Indication Map), by calling the `ieee80211_check_tim()` method (`include/linux/ieee80211.h`). The TIM is an array of 2008 entries. Because the TIM size is 251 bytes (2008 bits), you are allowed to send a partial virtual bitmap, which is smaller in size. If the entry in the TIM for that station is set, it means that the AP saved unicast packets for this station, so that station should empty the buffer of packets that the AP kept for it. The station starts sending null packets (or, more rarely, special control packets, called PS-Poll packets) to retrieve these buffered packets from the AP. Usually after the buffer has been emptied, the station goes to sleep (however, this is not mandatory according to the spec).

Handling the Multicast/Broadcast Buffer

The AP buffers multicast and broadcast packets whenever at least one station is in sleeping mode. The AID for multicast/broadcast stations is 0; so, in such a case, you set TIM[0] to true. The Delivery Team (DTIM), which is a special type of TIM, is sent not in every beacon, but once for a predefined number of beacon intervals (the DTIM period). After a DTIM is sent, the AP sends its buffered broadcast and multicast packets. You retrieve packets from the multicast/broadcast buffer (`bc_buf`) by calling the `ieee80211_get_buffered_bc()` method. In Figure 12-3 you can see an AP that contains a linked list of stations (`sta_info` objects), each of them with a unicast buffer (`ps_tx_buf`) of its own, and a single `bc_buf` buffer, for storing multicast and broadcast packets.

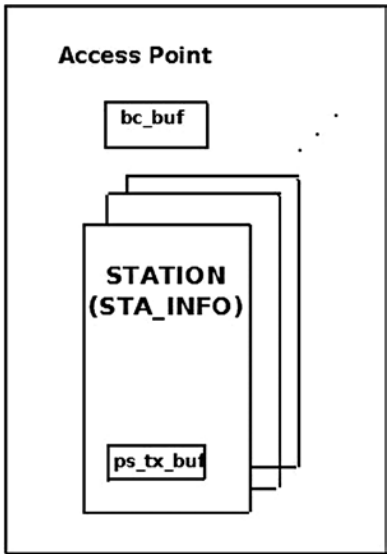

Figure 12-3. *Buffering packets in an AP*

The AP is implemented as an `ieee80211_if_ap` object in mac80211. Each such `ieee80211_if_ap` object has a member called `ps` (an instance of `ps_data`), where power save data is stored. One of the members of the `ps_data` structure is the broadcast/multicast buffer, `bc_buf`.

In Figure 12-4 you can see a flow of PS-Poll packets that a client sends in order to retrieve packets from the AP unicast buffer, `ps_tx_buf`. Note that the AP sends all the packets with the IEEE80211_FCTL_MOREDATA flag, except for the last one. Thus, the client knows that it should keep on sending PS-Poll packets until the buffer is emptied. For the sake of simplicity, the ACK traffic in this diagram is not included, but it should be mentioned here that the packets should be acknowledged.

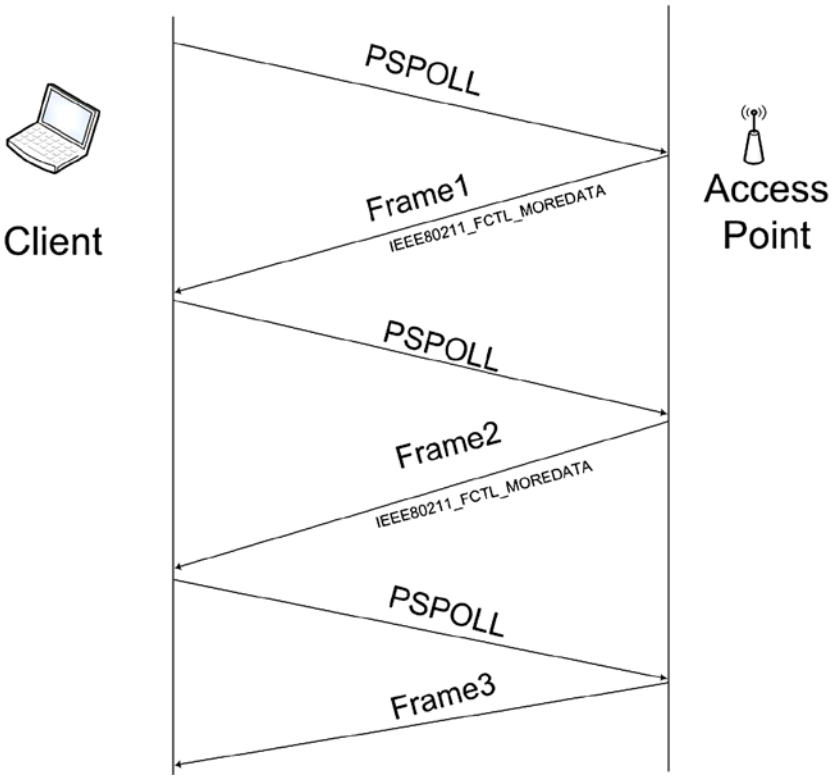

Figure 12-4. *Sending PSPOLL packets from a client to retrieve packets from the* `ps_tx_buf` *buffer within an AP*

■ **Note** *Power management* and *power save mode* are two different topics. Power management deals with handling machines that perform suspend (whether it is suspend to RAM or suspend to disk, aka hibernate, or in some cases, both suspend to RAM and suspend to disk, aka hybrid suspend), and is handled in `net/mac80211/pm.c`. In the drivers, power management is handled by the resume/suspend methods. Power save mode, on the other hand, deals with handling stations that enter sleep mode and wake up; it has nothing to do with suspend and hibernation.

This section described power save mode and the buffering mechanism. The next section discusses the management layer and the different tasks it handles.

The Management Layer (MLME)

There are three components in the 802.11 management architecture:

- The Physical Layer Management Entity (PLME).
- The System Management Entity (SME).
- The MAC Layer Management Entity (MLME).

Scanning

There are two types of scanning: passive scanning and active scanning. Passive scanning means to listen passively for beacons, without transmitting any packets for scanning. When performing passive scanning (the flags of the scan channel contain IEEE80211_CHAN_PASSIVE_SCAN), the station moves from channel to channel, trying to receive beacons. Passive scanning is needed in some higher 802.11a frequency bands, because you're not allowed to transmit anything at all until you've heard an AP beacon. With active scanning, each station sends a Probe Request packet; this is a management packet, with sub-type Probe Request (IEEE80211_STYPE_PROBE_REQ). Also with active scanning, the station moves from channel to channel, sending a Probe Request management packet on each channel (by calling the `ieee80211_send_probe_req()` method). This is done by calling the `ieee80211_request_scan()` method. Changing channels is done via a call to the `ieee80211_hw_config()` method, passing IEEE80211_CONF_CHANGE_CHANNEL as a parameter. Note that there is a one-to-one correspondence between a channel in which a station operates and the frequency in which it operates; the `ieee80211_channel_to_frequency()` method (`net/wireless/util.c`) returns the frequency in which a station operates, given its channel.

Authentication

Authentication is done by calling the `ieee80211_send_auth()` method (`net/mac80211/util.c`). It sends a management frame with authentication sub-type (IEEE80211_STYPE_AUTH). There are many authentications types; the original IEEE802.11 spec talked about only two forms: open-system authentication and shared key authentication. The only mandatory authentication method required by the IEEE802.11 spec is the open-system authentication (WLAN_AUTH_OPEN). This is a very simple authentication algorithm—in fact, it is a null authentication algorithm. Any client that requests authentication with this algorithm will become authenticated. An example of another option for an authentication algorithm is the shared key authentication (WLAN_AUTH_SHARED_KEY). In shared key authentication, the station should authenticate using a Wired Equivalent Privacy (WEP) key.

Association

In order to associate, a station sends a management frame with association sub-type (IEEE80211_STYPE_ASSOC_REQ). Association is done by calling the `ieee80211_send_assoc()` method (`net/mac80211/mlme.c`).

Reassociation

When a station moves between APs within an ESS, it is said to be *roaming*. The roaming station sends a reassociation request to a new AP by sending a management frame with reassociation sub-type (IEEE80211_STYPE_REASSOC_REQ). Reassociation is done by calling the `ieee80211_send_assoc()` method; there are many similarities between association and reassociation, so this method handles both. In addition, with reassociation, the AP returns an AID (association id) to the client in case of success.

This section talked about the management layer (MLME) and some of the operations it supports, like scanning, authentication, association, and more. In the next section I describe some mac80211 implementation details that are important in order to understand the wireless stack.

Mac80211 Implementation

Mac80211 has an API for interfacing with the low level device drivers. The implementation of mac80211 is complex and full of many small details. I cannot give an exhaustive description of the mac80211 API and implementation; I do discuss some important points that can give a good starting point to those who want to delve into the code. A fundamental structure of mac80211 API is the `ieee80211_hw` struct (`include/net/mac80211.h`); it represents hardware information. The `priv` (pointer to a private area) pointer of `ieee80211_hw` is of an opaque type (`void *`). Most wireless device drivers define a private structure for this private area, like `lbtf_private` (Marvell wireless driver) or `iwl_priv` (`iwlwifi` from Intel). Memory allocation and initialziation for the `ieee80211_hw struct` is done by the `ieee80211_alloc_hw()` method. Here are some methods related to the `ieee80211_hw` struct:

- `int ieee80211_register_hw(struct ieee80211_hw *hw)`: Called by wireless drivers for registering the specified `ieee80211_hw` object.
- `void ieee80211_unregister_hw(struct ieee80211_hw *hw)`: Unregisters the specified 802.11 hardware device.
- `struct ieee80211_hw *ieee80211_alloc_hw(size_t priv_data_len, const struct ieee80211_ops *ops)`: Allocates an `ieee80211_hw` object and initializes it.
- `ieee80211_rx_irqsafe()`: This method is for receiving a packet. It is implemented in `net/mac80211/rx.c` and called from low level wireless drivers.

The `ieee80211_ops` object, which is passed to the `ieee80211_alloc_hw()` method as you saw earlier, consists of pointers to callbacks to the driver. Not all of these callbacks must be implemented by the drivers. The following is a short description of these methods:

- `tx()`: The transmit handler called for each transmitted packet. It usually returns NETDEV_TX_OK (except for under certain limited conditions).
- `start()`: Activates the hardware device and is called before the first hardware device is enabled. It turns on frame reception.
- `stop()`: Turns off frame reception and usually turns off the hardware.
- `add_interface()`: Called when a network device attached to the hardware is enabled.
- `remove_interface()`: Informs a driver that the interface is going down.
- `config()`: Handles configuration requests, such as hardware channel configuration.
- `configure_filter()`: Configures the device's Rx filter.

Figure 12-5 shows a block diagram of the architecture of the Linux wireless subsystem. You can see that the interface between wireless device drivers layer and the mac80211 layer is the `ieee80211_ops` object and its callbacks.

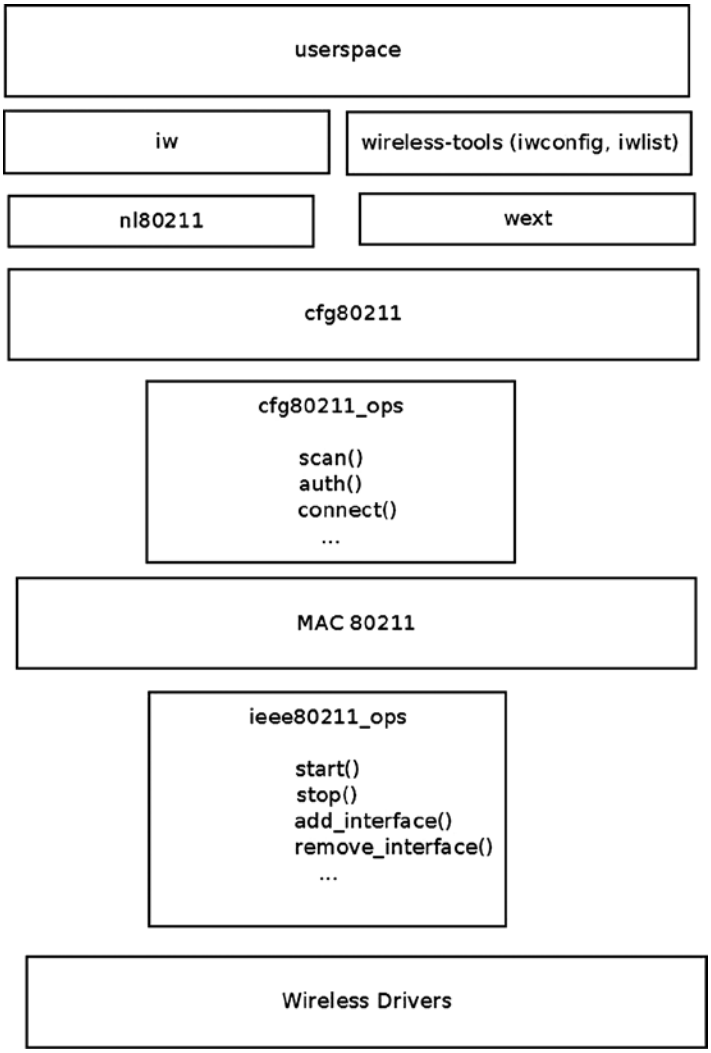

Figure 12-5. *Linux wireless architecture*

Another important structure is the `sta_info struct` (`net/mac80211/sta_info.h`), which represents a station. Among the members of this structure are various statistics counters, various flags, `debugfs` entries, the `ps_tx_buf` array for buffering unicast packets, and more. Stations are organized in a hash table (`sta_hash`) and a list (`sta_list`). The important methods related to `sta_info` are as follows:

- `int sta_info_insert(struct sta_info *sta)`: Adds a station.
- `int sta_info_destroy_addr(struct ieee80211_sub_if_data *sdata, const u8 *addr)`: Removes a station (by calling the `__sta_info_destroy()` method).
- `struct sta_info *sta_info_get(struct ieee80211_sub_if_data *sdata, const u8 *addr)`: Fetches a station; the address of the station (it's `bssid`) is passed as a parameter.

Rx Path

The `ieee80211_rx()` function (`net/mac80211/rx.c`) is the main receive handler. The status of the received packet (`ieee80211_rx_status`) is passed by the wireless driver to mac80211, embedded in the SKB control buffer (`cb`). The IEEE80211_SKB_RXCB() macro is used to fetch this status. The `flag` field of the Rx status specifies, for example, whether the FCS check failed on the packet (RX_FLAG_FAILED_FCS_CRC). The various values possible for the `flag` field are presented in Table 12-1 in the "Quick Reference" section of this chapter. In the `ieee80211_rx()` method, the `ieee80211_rx_monitor()` is invoked to remove the FCS (checksum) and remove a radiotap header (`struct ieee80211_radiotap_header`) which might have been added if the wireless interface is in monitor mode. (You use a network interface in monitor mode in case of sniffing, for example. Not all the wireless network interfaces support monitor mode, see the section "Wireless Modes" later in this chapter.)

If you work with HT (802.11n), you perform AMPDU reordering if needed by invoking the `ieee80211_rx_reorder_ampdu()` method. Then you call the `__ieee80211_rx_handle_packet()` method, which eventually calls the `ieee80211_invoke_rx_handlers()` method. Then you call, one by one, various receive handlers (using a macro named CALL_RXH). The order of calling these handlers is important. Each handler checks whether it should handle the packet or not. If it decides it should not handle the packet, then you return RX_CONTINUE and proceed to the next handler. If it decides it should handle the packet, then you return RX_QUEUED.

There are certain cases when a handler decides to drop a packet; in these cases, it returns RX_DROP_MONITOR or RX_DROP_UNUSABLE. For example, if you get a PS-Poll packet, and the type of the receiver shows that it is not an AP, you return RX_DROP_UNUSABLE. Another example: for a management frame, if the length of the SKB is less than the minimum (24), the packet is discarded and RX_DROP_MONITOR is returned. Or if the packet is not a management packet, then also the packet is discarded and RX_DROP_MONITOR is returned. Here is the code snippet from the `ieee80211_rx_h_mgmt_check()` method that implements this:

```
ieee80211_rx_h_mgmt_check(struct ieee80211_rx_data *rx)
{
        struct ieee80211_mgmt *mgmt = (struct ieee80211_mgmt *) rx->skb->data;
        struct ieee80211_rx_status *status = IEEE80211_SKB_RXCB(rx->skb);

        . . .
        if (rx->skb->len < 24)
                return RX_DROP_MONITOR;

        if (!ieee80211_is_mgmt(mgmt->frame_control))
                return RX_DROP_MONITOR;
               .  .  .
}
```

(net/mac80211/rx.c)

Tx Path

The `ieee80211_tx()` method is the main handler for transmission (`net/mac80211/tx.c`). First it invokes the `__ieee80211_tx_prepare()` method, which performs some checks and sets certain flags. Then it calls the `invoke_tx_handlers()` method, which calls, one by one, various transmit handlers (using a macro named CALL_TXH). If a transmit handler finds that it should do nothing with the packet, it returns TX_CONTINUE and you proceed to the next handler. If it decides it should handle a certain packet, it returns TX_QUEUED, and if it decides it should drop the

packet, it returns TX_DROP. The invoke_tx_handlers() method returns 0 upon success. Let's take a short look in the implementation of the ieee80211_tx() method:

```
static bool ieee80211_tx(struct ieee80211_sub_if_data *sdata,
                         struct sk_buff *skb, bool txpending,
                         enum ieee80211_band band)
{
        struct ieee80211_local *local = sdata->local;
        struct ieee80211_tx_data tx;
        ieee80211_tx_result res_prepare;
        struct ieee80211_tx_info *info = IEEE80211_SKB_CB(skb);
        bool result = true;
        int led_len;
```

Perform a sanity check, drop the SKB if its length is less than 10:

```
if (unlikely(skb->len < 10)) {
        dev_kfree_skb(skb);
        return true;
}

/* initialises tx */
led_len = skb->len;

res_prepare = ieee80211_tx_prepare(sdata, &tx, skb);

if (unlikely(res_prepare == TX_DROP)) {
        ieee80211_free_txskb(&local->hw, skb);
        return true;
} else if (unlikely(res_prepare == TX_QUEUED)) {
        return true;
}
```

Invoke the Tx handlers; if everything is fine, continue with invoking the __ieee80211_tx() method:

```
        . . .
        if (!invoke_tx_handlers(&tx))
                result = __ieee80211_tx(local, &tx.skbs, led_len,
                                        tx.sta, txpending);

        return result;
}
```

(net/mac80211/tx.c)

Fragmentation

Fragmentation in 802.11 is done only for unicast packets. Each station is assigned a fragmentation threshold size (in bytes). Packets that are bigger than this threshold should be fragmented. You can lower the number of collisions by reducing the fragmentation threshold size, making the packets smaller. You can inspect the fragmentation threshold of a station by running iwconfig or by inspecting the corresponding debugfs entry (see the "Mac80211 debugfs"

section later in this chapter). You can set the fragmentation threshold with the `iwconfig` command; thus, for example, you can set the fragmentation threshold to 512 bytes by:

```
iwconfig wlan0 frag 512
```

Each fragment is acknowledged. The more fragment field in the fragment header is set to 1 if there are more fragments. Each fragment has a fragment number (a subfield in the sequence control field of the frame control). Reassembling of the fragments on the receiver is done according to the fragments numbers. Fragmentation in the transmitter side is done by the `ieee80211_tx_h_fragment()` method (`net/mac80211/tx.c`). Reassembly on the receiver side is done by the `ieee80211_rx_h_defragment()` method (`net/mac80211/rx.c`). Fragmentation is incompatible with aggregation (used for higher throughput), and given the high rates and thus short (in time) packets it is very rarely used nowadays.

Mac80211 debugfs

`debugfs` is a technique that enables exporting debugging information to userspace. It creates entries under the `sysfs` filesystem. `debugfs` is a virtual filesystem devoted to debugging information. For mac80211, handling mac80211 `debugfs` is mostly in `net/mac80211/debugfs.c`. After mounting `debugfs`, various mac802.11 statistics and information entries can be inspected. Mounting `debugfs` is performed like this:

```
mount -t debugfs none_debugs /sys/kernel/debug
```

■ **Note** CONFIG_DEBUG_FS must be set when building the kernel to be able to mount and work with `debugfs`.

For example, let's say your phy is `phy0`; the following is a discussion about some of the entries under `/sys/kernel/debug/ieee80211/phy0`:

- `total_ps_buffered`: This is the total number of packets (unicast and multicasts/broadcasts) which the AP buffered for the station. The `total_ps_buffered` counter is incremented by `ieee80211_tx_h_unicast_ps_buf()` for unicasts, and by `ieee80211_tx_h_multicast_ps_buf()` for multicasts or broadcasts.
- Under `/sys/kernel/debug/ieee80211/phy0/statistics`, you have various statistical information—for example:
  - `frame_duplicate_count` denotes the number of duplicate frames. This `debugfs` entry represents the duplicate frames counter, `dot11FrameDuplicateCount`, which is incremented by the `ieee80211_rx_h_check()` method.
  - `transmitted_frame_count` denotes the number of transmitted packets. This `debugfs` entry represents `dot11TransmittedFrameCount`; it is incremented by the `ieee80211_tx_status()` method.
  - `retry_count` denotes number of retransmissions. This `debugfs` entry represents `dot11RetryCount`; it is incremented also by the `ieee80211_tx_status()` method.
  - `fragmentation_threshold`: The size of the fragmentation threshold, in bytes. See the "Fragmentation" section earlier.

- Under `/sys/kernel/debug/ieee80211/phy0/netdev:wlan0`, you have some entries that give information about the interface; for example, if the interface is in station mode, you will have `aid` for the association id of the station, `assoc_tries` for the number of times the stations tried to perform association, `bssid` is for the bssid of the station, and so on.
- Every station uses a rate control algorithm. Its name is exported by the following `debugfs` entry: `/sys/kernel/debug/ieee80211/phy1/rc/name`.

Wireless Modes

You can set a wireless network interface to operate in several modes, depending on its intended use and the topology of the network in which it is deployed. In some cases, you can set the mode with the `iwconfig` command, and in some cases you must use a tool like `hostapd` for this. Note that not all devices support all modes. See `www.linuxwireless.org/en/users/Drivers` for a list of Linux drivers that support different modes. Alternatively, you can also check to which values the `interface_modes` field of the `wiphy` member (in the `ieee80211_hw` object) is initialized in the driver code. The `interface_modes` are initialized to one or more modes of the `nl80211_iftype enum`, like NL80211_IFTYPE_STATION or NL80211_IFTYPE_ADHOC (see: `include/uapi/linux/nl80211.h`). The following is a detailed description of these wireless modes:

- *AP mode:* In this mode, the device acts as an AP (NL80211_IFTYPE_AP). The AP maintains and manages a list of associated stations. The network (BSS) name is the MAC address of the AP (`bssid`). There is also a human-readable name for the BSS, called the SSID.
- *Station infrastructure mode:* A managed station in an infrastructure mode (NL80211_IFTYPE_STATION).
- *Monitor mode:* All incoming packets are handed unfiltered in monitor mode (NL80211_IFTYPE_MONITOR). This is useful for sniffing. It is usually possible to transmit packets in monitor mode. This is termed *packet injection*; these packets are marked with a special flag (IEEE80211_TX_CTL_INJECTED).
- *Ad Hoc (IBSS) mode:* A station in an ad hoc (IBSS) network (NL80211_IFTYPE_ADHOC). With Ad Hoc mode, there is no AP device in the network.
- *Wireless Distribution System (WDS) mode:* A station in a WDS network (NL80211_IFTYPE_WDS).
- *Mesh mode:* A station in a Mesh network (NL80211_IFTYPE_MESH_POINT), discussed in the "Mesh Networking (802.11s)" section later in this chapter.

The next section discusses the ieee802.11n technology, which provides higher performance, and how it is implemented in the Linux wireless stack. You will learn also about block acknowledgment and packet aggregation in 802.11n and how these techniques are used to improve performance.

High Throughput (ieee802.11n)

A little after 802.11g was approved, a new task group was created in IEEE, called High Throughput Task Group (TGn). IEEE 802.11n became a final spec at the end of 2009. The IEEE 802.11n protocol allows coexistence with legacy devices. There were some vendors who already sold 802.11n pre-standard devices based on the 802.11n draft before the official approval. Broadcom set a precedent for releasing wireless interfaces based on a draft. In 2003, it released a chipset of a wireless device based on a draft of 802.11g. Following this precedent, as early as 2005 some vendors released products based on the 802.11n draft. For example, Intel Santa Rose processor has Intel Next-Gen Wireless-N (Intel WiFI Link 5000 series), supports 802.11n. Other Intel wireless network interfaces, like 4965AGN, also supported 802.11n. Other vendors, including Atheros and Ralink, also released 802.11n draft-based wireless devices. The WiFi

alliance started certification of 802.11n draft devices in June 2007. A long list of vendors released products which comply with Wi-Fi CERTIFIED 802.11n draft 2.0.

802.11n can operate on the 2.4 GHz and/or 5 GHz bands, whereas 802.11g and 802.11b operate only in the 2.4 GHz radio frequency band, and 802.11a operates only in the 5 GHz radio frequency band. The 802.11n MIMO (Multiple Input, Multiple Output) technology increases the range and reliability of traffic over the wireless coverage area. MIMO technology uses multiple transmitter and receiver antennas on both APs and clients, to allow for simultaneous data streams. The result is increased range and increased throughput. With 802.11n you can achieve a theoretical PHY rate of up to 600 Mbps (actual throughput will be much lower due to medium access rules, and so on).

802.11n added many improvements for the 802.11 MAC layer. The most well known is packet aggregation, which concatenates multiple packets of application data into a single transmission frame. A block acknowledgment (BA) mechanism was added (discussed in the next section). BA permits multiple packets to be acknowledged by a single packet instead of sending an ACK for each received packet. The wait time between two consecutive packets is cut. This enables sending multiple data packets with a fixed overhead cost of a single packet. The BA protocol was introduced in the 802.11e amendment from 2005.

Packet Aggregation

There are two types of packet aggregation:

- *AMSDU:* Aggregated Mac Service Data Unit
- *AMPDU:* Aggregated Mac Protocol Data Unit

Note that the AMSDU is only supported on Rx, and not on Tx, and is wholly independent from the Block Ack mechanism described in this section; so the discussion in this section only pertains to AMPDU.

There are two sides to a Block Ack session: *originator* and *recipient*. Each block session has a different Traffic Identifier (TID). The originator starts the block acknowledgement session by calling the `ieee80211_start_tx_ba_session()` method. This is done typically from a rate control algorithm method in the driver. For example, with the ath9k wireless driver, the `ath_tx_status()` function (`drivers/net/wireless/ath/ath9k/rc.c`), which is a rate control callback, invokes the `ieee80211_start_tx_ba_session()` method. The `ieee80211_start_tx_ba_session()` method sets the state to HT_ADDBA_REQUESTED_MSK and sends an ADDBA request packet, by invoking the `ieee80211_send_addba_request()` method. The call to `ieee80211_send_addba_request()` passes parameters for the session, such as the wanted reorder buffer size and the TID of the session.

The reorder buffer size is limited to 64K (see the definition of `ieee80211_max_ampdu_length_exp` in `include/linux/ieee80211.h`). These parameters are part of the capability member (`capab`) in the struct `addba_req`. The response to the ADDBA request should be received within 1 Hz, which is one second in x86_64 machines (ADDBA_RESP_INTERVAL). If you do not get a response in time, the `sta_addba_resp_timer_expired()` method will stop the BA session by calling the `___ieee80211_stop_tx_ba_session()` method. When the other side (the recipient) receives the ADDBA request, it first sends an ACK (every packet in ieee802.11 should be acknowledged, as mentioned before). Then it processes the ADDBA request by calling the `ieee80211_process_addba_request()` method; if everything is okay, it sets the aggregation state of this machine to operational (HT_AGG_STATE_OPERATIONAL) and sends an ADDBA response by calling the `ieee80211_send_addba_resp()` method. It also stops the response timer (the timer which has as its callback the `sta_addba_resp_timer_expired()` method) by calling `del_timer_sync()` on this timer. After a session is started, a data block containing multiple MPDU packets is sent. Consequently, the originator sends a Block Ack Request (BAR) packet by calling the `ieee80211_send_bar()` method.

Block Ack Request (BAR)

The BAR is a control packet with Block Ack Request sub-type (IEEE80211_STYPE_BACK_REQ). The BAR packet includes the SSN (start sequence number), which is the sequence number of the oldest MSDU in the block that should be acknowledged. The recipient receives the BAR and reorders the `ampdu` buffer accordingly, if needed. Figure 12-6 shows a BAR request.

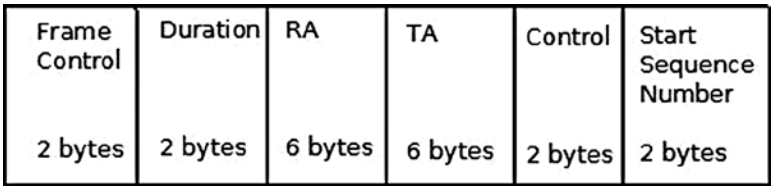

Figure 12-6. *BAR request*

When sending a BAR, the `type` subfield in the frame control is control (IEEE80211_FTYPE_CTL), and the `subtype` subfield is Block Ack request (IEEE80211_STYPE_BACK_REQ). The BAR is represented by the `ieee80211_bar` struct:

```
struct ieee80211_bar {
        __le16 frame_control;
        __le16 duration;
        __u8 ra[6];
        __u8 ta[6];
        __le16 control;
        __le16 start_seq_num;
} __packed;
```

(`include/linux/ieee80211.h`)

The RA is the recipient address, and the TA is the transmitter (originator) address. The control field of the BAR request includes the TID.

Block Ack

There are two types of Block Ack: Immediate Block Ack and Delayed Block Ack. Figure 12-7 shows Immediate Block Ack.

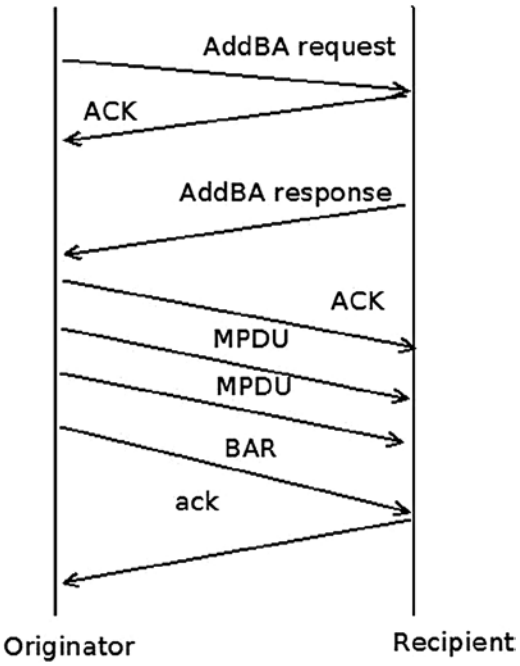

Figure 12-7. *Immediate Block Ack*

The difference between Immediate Block Ack and Delayed Block Ack is that with Delayed Block Ack, the BAR request itself is answered first with an ACK, and then after some delay, with a BA (Block Ack). When using Delayed Block Ack, there is more time to process the BAR, and this is sometime needed when working with software based processing. Using Immediate Block Ack is better in terms of performance. The BA itself is also acknowledged. When the originator has no more data to send, it can terminate the Block Ack session by calling the `ieee80211_send_delba()` method; this function sends a DELBA request packet to the other side. The DELBA request is handled by the `ieee80211_process_delba()` method. The DELBA message, which causes a Block Ack session tear down, can be sent either from the originator or recipient of the Block Ack session. The AMPDU maximum length is 65535 octets. Note that packet aggregation is only implemented for APs and managed stations; packet aggregation for IBSS is not supported by the spec.

Mesh Networking (802.11s)

The IEEE 802.11s protocol started as a Study Group of IEEE in September 2003, and became a Task Group named TGs in 2004. In 2006, 2 proposals, out of 15 (the "SEEMesh" and "Wi-Mesh" proposals) were merged into one, which resulted in draft D0.01. 802.11s was ratified in July 2011 and is now part of IEEE 802.11-2012. Mesh networks allow the creation of an 802.11 Basic Service Set over fully and partially connected Mesh topology. This can be seen as an improvement over 802.11 ad hoc network, which requires a fully-connected Mesh topology. Figures 12-8 and 12-9 illustrate the difference between the two types of Mesh topologies.

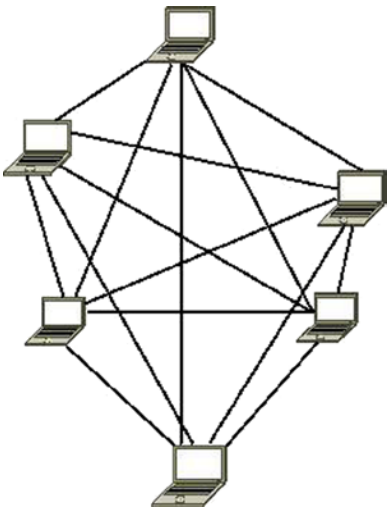

Figure 12-8. *Full Mesh*

In a partially-connected Mesh, nodes are connected to only some of the other nodes, but not to all of them. This topology is much more common in wireless Mesh networks. Figure 12-9 shows an example of a partial mesh.

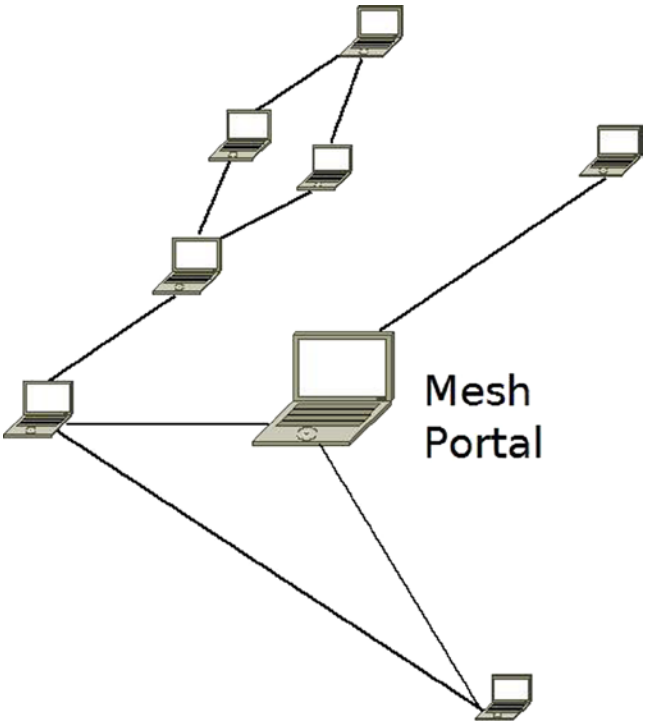

Figure 12-9. *Partial Mesh*

Wireless mesh networks forward data packets over multiple wireless hops. Each mesh node acts as a relay point/router for the other mesh nodes. In kernel 2.6.26 (2008), support for the draft of wireless mesh networking (802.11s) was added to the network wireless stack, thanks to the open80211s project. The open80211s project goal was to create the first open implementation of 802.11s. The project got some sponsorship from the OLPC project and from some commercial companies. Luis Carlos Cobo and Javier Cardona and other developers from Cozybit developed the Linux mac80211 Mesh code.

Now that you have learned a bit about Mesh networking and Mesh network topologies, you are ready for the next section, which covers the HWMP routing protocol for Mesh networks.

HWMP Protocol

The 802.11s protocol defines a default routing protocol called HWMP (Hybrid Wireless Mesh Protocol). The HWMP protocol works with Layer 2 and deals with MAC addresses, as opposed to the IPV4 routing protocol, for example, which works with Layer 3 and deals with IP addresses. HWMP routing is based on two types of routing (hence it is called *hybrid*). The first is *on-demand* routing, and the second is *proactive routing*. The main difference between the two mechanisms has to do with the time in which path establishment is initiated (*path* is the name used for route in Layer 2). In on-demand routing, a path to a destination is established by the protocol only after the protocol stack has received frames for such a destination. This minimizes the amount of management traffic required to maintain the Mesh network at the expense of introducing additional latency in data traffic. Proactive routing can be used if a Mesh node is known to be the recipient of a lot of mesh traffic. In that case, the node will periodically announce itself over the Mesh network and trigger path establishments to itself from all the Mesh nodes in the network. Both on-demand and proactive routing are implemented in the Linux kernel. There are four types of routing messages:

- *PREQ (Path Request):* This type of message is sent as a broadcast when you look for some destination that you still do not have a route to. This PREQ message is propagated in the Mesh network until it gets to its destination. A lookup is performed on each station until the final destination is reached (by calling the `mesh_path_lookup()` method). If the lookup fails, the PREQ is forwarded (as a broadcast) to the other stations. The PREQ message is sent in a management packet; its sub-type is action (IEEE80211_STYPE_ACTION). It is handled by the `hwmp_preq_frame_process()` method.
- *PREP (Path Reply):* This type is a unicast packet that is sent as a reply to a PREQ message. This packet is sent in the reverse path. The PREP message is also sent in a management packet and its `subtype` is also the action sub-type (IEEE80211_STYPE_ACTION). It is handled by the `hwmp_prep_frame_process()` method. Both the PREQ and the PREP messages are sent by the `mesh_path_sel_frame_tx()` method.
- *PERR (Path Error):* If there is some failure on the way, a PERR is sent. A PERR message is handled by the `mesh_path_error_tx()` method.
- *RANN (Root Announcement):* The Root Mesh point periodically broadcasts this frame. Mesh points that receive it send a unicast RREQ to the root via the MP from which it received the RANN. In response, the Root Mesh will send a PREP response to each PREQ.

■ **Note** The route takes into consideration a radio-aware metric (airtime metric). The airtime metric is calculated by the `airtime_link_metric_get()` method (based on rate and other hardware parameters). Mesh points continuously monitor their links and update metric values with neighbours.

The station that sent the PREQ may try to send packets to the final destination while still not knowing the route to that destination; these packets are kept in a buffer of SKBs named `frame_queue`, which is a member of the `mesh_path` object (`net/mac80211/mesh.h`). In such a case, when a PREP finally arrives, the pending packets of this buffer are sent to the final destination (by calling the `mesh_path_tx_pending()` method). The maximum number of frames buffered per destination for unresolved destinations is 10 (MESH_FRAME_QUEUE_LEN). The advantages of Mesh networking are as follows:

- Rapid deployment
- Minimal configuration, inexpensive
- Easy to deploy in hard-to-wire environments
- Connectivity while nodes are in motion
- Higher reliability: no single point of failure and the ability to heal itself

The disadvantages are as follows:

- Many broadcasts limit network performance.
- Not all wireless drivers support Mesh mode at the moment.

Setting Up a Mesh Network

There are two sets of userspace tools for managing wireless devices and networks in Linux: one is the older Wireless Tools for Linux, an open source project based on IOCTLs. Examples of command line utilities of the wireless tools are `iwconfig`, `iwlist`, `ifrename`, and more. The newer tool is `iw`, based on generic netlink sockets (described in Chapter 2). However, there are some tasks that only the newer tool, `iw`, can perform. You can set a wireless device to work in Mesh mode only with the `iw` command.

Example: setting a wireless network interface (`wlan0`) to work in Mesh mode can be done like this:

```
iw wlan0 set type mesh
```

■ **Note** Setting a wireless network interface (wlan0) to work in mesh mode can be done also like this:

```
iw wlan0 set type mp
```

`mp` stands for Mesh Point. See "Adding interfaces with iw" in http://wireless.kernel.org/en/users/Documentation/iw

---

Joining the mesh is done by: `iw wlan0 mesh join "my-mesh-ID"`

You can display statistics about a station by the following:

- `iw wlan0 station dump`
- `iw wlan0 mpath dump`

I should mention here also the `authsae` and the `wpa_supplicant` tools, which can be used to create secure Mesh networks and do not depend upon `iw`.

Linux Wireless Development Process

Most development is done using the `git` distributed version control system, as with many other Linux subsystems. There are three main `git` trees; the bleeding edge is the wireless-testing tree. There are also the regular wireless tree and the wireless-next tree. The following are the links to the `git` repositories for the development trees:

- wireless-testing development tree:

  `git://git.kernel.org/pub/scm/linux/kernel/git/linville/wireless-testing.git`

- wireless development tree:

  `git://git.kernel.org/pub/scm/linux/kernel/git/linville/wireless-2.6.git`

- wireless-next development tree:

  `git://git.kernel.org/pub/scm/linux/kernel/git/linville/wireless-next-2.6.git`

Patches are sent and discussed in the wireless mailing list: `linux-wireless@vger.kernel.org`. From time to time a pull request is sent to the kernel networking mailing list, `netdev`, mentioned in Chapter 1.

As mentioned in the "Mac80211 subsystem" section, which dealt with the mac80211 subsystem, some wireless network interface vendors maintain their own development trees for their Linux drivers on their own sites. In some cases, the code they are using does not use the mac80211 API; for example, some Ralink and Realtek wireless device drivers. Since January 2006, the maintainer of the Linux wireless subsystem is John W. Linville, who replaced Jeff Garzik. The maintainer of mac80211 is Johannes Berg, from October 2007. There were some annual Linux wireless summits; the first took place in 2006 in Beaverton (OR). A very detailed wiki page is here: `http://wireless.kernel.org/`. This web site includes a lot of important documentation. For example, a table specifies the modes each wireless network interface supports. There is a lot of information in this wiki page regarding many wireless device drivers, hardware, and various tools (such as CRDA, the central regulatory domain agent, `hostapd`, `iw`, and more).

Summary

A lot of development has been done in Linux wireless stack in recent years. The most significant change is the integration of the mac80211 stack and porting wireless drivers to use the mac80211 API, making the code much more organized. The situation is much better than before; many more wireless devices are supported in Linux. Mesh networking got a boost recently thanks to the open802.11s project. It was integrated in the Linux 2.6.26 kernel. The future will probably see more drivers that support the new standard, IEEE802.11ac, a 5 GHz-only technology that can reach maximum throughputs well above a gigabit per second, and more drivers that support P2P.

Chapter 13 discusses InfiniBand and RDMA in the Linux kernel. The "Quick Reference" section covers the top methods that are related to the topics discussed in this chapter, ordered by their context.

Quick Reference

I conclude this chapter with a short list of important methods of the Linux wireless subsystem, some of which are mentioned in this chapter. Table 12-1 shows the various possible values for the `flag` member of the `ieee80211_rx_status` object.

Methods

This section discusses the methods.

void ieee80211_send_bar(struct ieee80211_vif *vif, u8 *ra, u16 tid, u16 ssn);

This method sends a block acknowledgment request.

int ieee80211_start_tx_ba_session(struct ieee80211_sta *pubsta, u16 tid, u16 timeout);

This method starts a Block Ack session by calling the wireless driver `ampdu_action()` callback, passing IEEE80211_AMPDU_TX_START. As a result, the driver will later call the `ieee80211_start_tx_ba_cb()` callback or the `ieee80211_start_tx_ba_cb_irqsafe()` callback, which will start the aggregation session.

int ieee80211_stop_tx_ba_session(struct ieee80211_sta *pubsta, u16 tid);

This method stops a Block Ack session by calling the wireless driver `ampdu_action()` function, passing IEEE80211_AMPDU_TX_STOP. The driver must later call the `ieee80211_stop_tx_ba_cb()` callback or the `ieee80211_stop_tx_ba_cb_irqsafe()` callback.

static void ieee80211_send_addba_request(struct ieee80211_sub_if_data *sdata, const u8 *da, u16 tid, u8 dialog_token, u16 start_seq_num, u16 agg_size, u16 timeout);

This method sends an ADDBA message. An ADDBA message is a management action message.

void ieee80211_process_addba_request(struct ieee80211_local *local, struct sta_info *sta, struct ieee80211_mgmt *mgmt, size_t len);

This method handles an ADDBA message.

static void ieee80211_send_addba_resp(struct ieee80211_sub_if_data *sdata, u8 *da, u16 tid, u8 dialog_token, u16 status, u16 policy, u16 buf_size, u16 timeout);

This method sends an ADDBA response. An ADDBA response is a management packet, with `subtype` of action (IEEE80211_STYPE_ACTION).

static ieee80211_rx_result debug_noinline ieee80211_rx_h_amsdu(struct ieee80211_rx_data *rx);

This method handles AMSDU aggregation (Rx path).

void ieee80211_process_delba(struct ieee80211_sub_if_data *sdata, struct sta_info *sta, struct ieee80211_mgmt *mgmt, size_t len);

This method handles a DELBA message.

void ieee80211_send_delba(struct ieee80211_sub_if_data *sdata, const u8 *da, u16 tid, u16 initiator, u16 reason_code);

This method sends a DELBA message.

void ieee80211_rx_irqsafe(struct ieee80211_hw *hw, struct sk_buff *skb);

This method receives a packet. The `ieee80211_rx_irqsafe()` method can be called in hardware interrupt context.

static void ieee80211_rx_reorder_ampdu(struct ieee80211_rx_data *rx, struct sk_buff_head *frames);

This method handles the A-MPDU reorder buffer.

static bool ieee80211_sta_manage_reorder_buf(struct ieee80211_sub_if_data *sdata, struct tid_ampdu_rx *tid_agg_rx, struct sk_buff_head *frames);

This method handles the A-MPDU reorder buffer.

static ieee80211_rx_result debug_noinline ieee80211_rx_h_check(struct ieee80211_rx_data *rx);

This method drops duplicate frames of a retransmission and increment `dot11FrameDuplicateCount` and the station `num_duplicates` counter.

void ieee80211_send_nullfunc(struct ieee80211_local *local, struct ieee80211_sub_if_data *sdata, int powersave);

This method sends a special NULL data frame.

void ieee80211_send_pspoll(struct ieee80211_local *local, struct ieee80211_sub_if_data *sdata);

This method sends a PS-Poll control packet to an AP.

static void ieee80211_send_assoc(struct ieee80211_sub_if_data *sdata);

This method performs association or reassociation by sending a management packet with association sub-type of IEEE80211_STYPE_ASSOC_REQ or IEEE80211_STYPE_REASSOC_REQ, respectively. The `ieee80211_send_assoc()` method is invoked from the `ieee80211_do_assoc()` method.

void ieee80211_send_auth(struct ieee80211_sub_if_data *sdata, u16 transaction, u16 auth_alg, u16 status, const u8 *extra, size_t extra_len, const u8 *da, const u8 *bssid, const u8 *key, u8 key_len, u8 key_idx, u32 tx_flags);

This method performs authentication by sending a management packet with authentication sub-type (IEEE80211_STYPE_AUTH).

static inline bool ieee80211_check_tim(const struct ieee80211_tim_ie *tim, u8 tim_len, u16 aid);

This method checks whether the `tim[aid]` is set; the aid is passed as a parameter, and it represents the association id of the station.

int ieee80211_request_scan(struct ieee80211_sub_if_data *sdata, struct cfg80211_scan_request *req);

This method starts active scanning.

void mesh_path_tx_pending(struct mesh_path *mpath);

This method send packets from the `frame_queue`.

struct mesh_path *mesh_path_lookup(struct ieee80211_sub_if_data *sdata, const u8 *dst);

This method performs a lookup in a Mesh path table (routing table) of a Mesh point. The second parameter to the `mesh_path_lookup()` method is the hardware address of the destination. It returns NULL if there is no entry in the table, otherwise it returns a pointer to the mesh path structure which was found.

static void ieee80211_sta_create_ibss(struct ieee80211_sub_if_data *sdata);

This method creates an IBSS.

int ieee80211_hw_config(struct ieee80211_local *local, u32 changed);

This method is called for various configurations by the driver; in most cases, it delegates the call to the `driver config()` method, if implemented. The second parameter specifies which action to take (for instance, IEEE80211_CONF_CHANGE_CHANNEL to change channel, or IEEE80211_CONF_CHANGE_PS to change the power save mode of the driver).

struct ieee80211_hw *ieee80211_alloc_hw(size_t priv_data_len, const struct ieee80211_ops *ops);

This method allocates a new 802.11 hardware device.

int ieee80211_register_hw(struct ieee80211_hw *hw);

This method registers a 802.11 hardware device.

void ieee80211_unregister_hw(struct ieee80211_hw *hw);

This method unregisters a 802.11 hardware device and frees its allocated resources.

int sta_info_insert(struct sta_info *sta);

This method adds a station to the hash table of stations and to the list of stations.

int sta_info_destroy_addr(struct ieee80211_sub_if_data *sdata, const u8 *addr);

This method removes a station and frees its resources.

struct sta_info *sta_info_get(struct ieee80211_sub_if_data *sdata, const u8 *addr);

This method returns a pointer to a station by performing a lookup in the hash table of stations.

void ieee80211_send_probe_req(struct ieee80211_sub_if_data *sdata, u8 *dst, const u8 *ssid, size_t ssid_len, const u8 *ie, size_t ie_len, u32 ratemask, bool directed, u32 tx_flags, struct ieee80211_channel *channel, bool scan);

This method sends a probe request management packet.

static inline void ieee80211_tx_skb(struct ieee80211_sub_if_data *sdata, struct sk_buff *skb);

This method transmits an SKB.

int ieee80211_channel_to_frequency(int chan, enum ieee80211_band band);

This method returns the frequency in which a station operates, given its channel. There is a one-to-one correspondence between a channel and a frequency.

static int mesh_path_sel_frame_tx(enum mpath_frame_type action, u8 flags, const u8 *orig_addr, __le32 orig_sn, u8 target_flags, const u8 *target, __le32 target_sn, const u8 *da, u8 hop_count, u8 ttl, __le32 lifetime, __le32 metric, __le32 preq_id, struct ieee80211_sub_if_data *sdata);

This method sends a PREQ or PREP management packet.

static void hwmp_preq_frame_process(struct ieee80211_sub_if_data *sdata, struct ieee80211_mgmt *mgmt, const u8 *preq_elem, u32 metric);

This method handles a PREQ message.

struct ieee80211_rx_status *IEEE80211_SKB_RXCB(struct sk_buff *skb);

This method returns the `ieee80211_rx_status` object associated with the control buffer (`cb`), which is associated with the specified SKB.

static bool ieee80211_tx(struct ieee80211_sub_if_data *sdata, struct sk_buff *skb, bool txpending, enum ieee80211_band band);

This method is the main handler for transmission.

Table

Table 12-1 shows the bits of the flag member (a 32-bit field) of the `ieee80211_rx_status` structure and the corresponding Linux symbol.

Table 12-1. *Rx Flags: Various Possible Values for the Flag Field of the `ieee80211_rx_status` Object*

Linux Symbol	Bit	Description
RX_FLAG_MMIC_ERROR	0	Michael MIC error was reported on this frame.
RX_FLAG_DECRYPTED	1	This frame was decrypted in hardware.
RX_FLAG_MMIC_STRIPPED	3	The Michael MIC is stripped off this frame, verification has been done by the hardware.
RX_FLAG_IV_STRIPPED	4	The IV/ICV are stripped from this frame.
RX_FLAG_FAILED_FCS_CRC	5	The FCS check failed on the frame.
RX_FLAG_FAILED_PLCP_CRC	6	The PCLP check failed on the frame.
RX_FLAG_MACTIME_START	7	The timestamp passed in the RX status is valid and contains the time the first symbol of the MPDU was received.

(*continued*)

Table 12-1. (*continued*)

Linux Symbol	Bit	Description
RX_FLAG_SHORTPRE	8	Short preamble was used for this frame.
RX_FLAG_HT	9	HT MCS was used and `rate_idx` is MCS index
RX_FLAG_40MHZ	10	HT40 (40 MHz) was used.
RX_FLAG_SHORT_GI	11	Short guard interval was used.
RX_FLAG_NO_SIGNAL_VAL	12	The signal strength value is not present.
RX_FLAG_HT_GF	13	This frame was received in a HT-greenfield transmission
RX_FLAG_AMPDU_DETAILS	14	A-MPDU details are known, in particular the reference number must be populated and be a distinct number for each A-MPDU.
RX_FLAG_AMPDU_REPORT_ZEROLEN	15	Driver reports 0-length subframes.
RX_FLAG_AMPDU_IS_ZEROLEN	16	This is a zero-length subframe, for monitoring purposes only.
RX_FLAG_AMPDU_LAST_KNOWN	17	Last subframe is known, should be set on all subframes of a single A-MPDU.
RX_FLAG_AMPDU_IS_LAST	18	This subframe is the last subframe of the A-MPDU.
RX_FLAG_AMPDU_DELIM_CRC_ERROR	19	A delimiter CRC error has been detected on this subframe.
RX_FLAG_AMPDU_DELIM_CRC_KNOWN	20	The delimiter CRC field is known (the CRC is stored in the `ampdu_delimiter_crc` field of the `ieee80211_rx_status`)
RX_FLAG_MACTIME_END	21	The timestamp passed in the RX status is valid and contains the time the last symbol of the MPDU (including FCS) was received.
RX_FLAG_VHT	22	VHT MCS was used and `rate_index` is MCS index
RX_FLAG_80MHZ	23	80 MHz was used
RX_FLAG_80P80MHZ	24	80+80 MHz was used
RX_FLAG_160MHZ	25	160 MHz was used

CHAPTER 13

InfiniBand

This chapter was written by Dotan Barak, an InfiniBand Expert. Dotan is a Senior Software Manager at Mellanox Technologies working on RDMA Technologies. Dotan has been working at Mellanox for more than 10 years in various roles, both as a developer and a manager. Additionally, Dotan maintains a blog about the RDMA technology: http://www.rdmamojo.com.

Chapter 12 dealt with the wireless subsystem and its implementation in Linux. In this chapter, I will discuss the InfiniBand subsystem and its implementation in Linux. Though the InfiniBand technology might be perceived as a very complex technology for those who are unfamiliar with it, the concepts behind it are surprisingly straightforward, as you will see in this chapter. I will start our discussion with Remote Direct Memory Access (RDMA), and discuss its main data structures and its API. I will give some examples illustrating how to work with RDMA, and conclude this chapter with a short discussion about using RDMA API from the kernel level and userspace.

RDMA and InfiniBand—General

Remote Direct Memory Access (RDMA) is the ability for one machine to access—that is, to read or write to—memory on a remote machine. There are several main network protocols that support RDMA: InfiniBand, RDMA over Converged Ethernet (RoCE) and internet Wide Area RDMA Protocol (iWARP), and all of them share the same API. InfiniBand is a completely new networking protocol, and its specifications can be found in the document "InfiniBand Architecture specifications," which is maintained by the InfiniBand Trade Association (IBTA). RoCE allows you to have RDMA over an Ethernet network, and its specification can be found as an Annex to the InfiniBand specifications. iWARP is a protocol that allows using RDMA over TCP/IP, and its specifications can be found in the document, "An RDMA Protocol Specification," which is being maintained by the RDMA Consortium. **Verbs** is the description of the API to use RDMA from a client code. The RDMA API implementation was introduced to the Linux kernel in version 2.6.11. At the beginning, it supported only InfiniBand, and after several kernel versions, iWARP and RoCE support were added to it as well. When describing the API, I mention only one of them, but the following text refers to all. All of the definitions to this API can be found in `include/rdma/ib_verbs.h`. Here are some notes about the API and the implementation of the RDMA stack:

- Some of the functions are inline functions, and some of them aren't. Future implementation might change this behavior.
- Most of the APIs have the prefix "ib"; however, this API supports InfiniBand, iWARP and RoCE.
- The header `ib_verbs.h` contains functions and structures to be used by:
  - The RDMA stack itself
  - Low-level drivers for RDMA devices
  - Kernel modules that use the stack as consumers

I will concentrate on functions and structures that are relevant only for kernel modules that use the stack as consumers (the third case). The following section discusses the RDMA stack organization in the kernel tree.

The RDMA Stack Organization

Almost all of the kernel RDMA stack code is under `drivers/infiniband` in the kernel tree. The following are some of its important modules (this is not an exhaustive list, as I do not cover the entire RDMA stack in this chapter):

- **CM:** Communication manager (`drivers/infiniband/core/cm.c`)
- **IPoIB:** IP over InfiniBand (`drivers/infiniband/ulp/ipoib/`)
- **iSER:** iSCSI extension for RDMA (`drivers/infiniband/ulp/iser/`)
- **RDS:** Reliable Datagram Socket (`net/rds/`)
- **SRP:** SCSI RDMA protocol (`drivers/infiniband/ulp/srp/`)
- Hardware low-level drivers of different vendors (`drivers/infiniband/hw`)
- **verbs:** Kernel verbs (`drivers/infiniband/core/verbs.c`)
- **uverbs:** User verbs (`drivers/infiniband/core/uverbs_*.c`)
- **MAD:** Management datagram (`drivers/infiniband/core/mad.c`)

Figure 13-1 shows the Linux InfiniBand stack architecture.

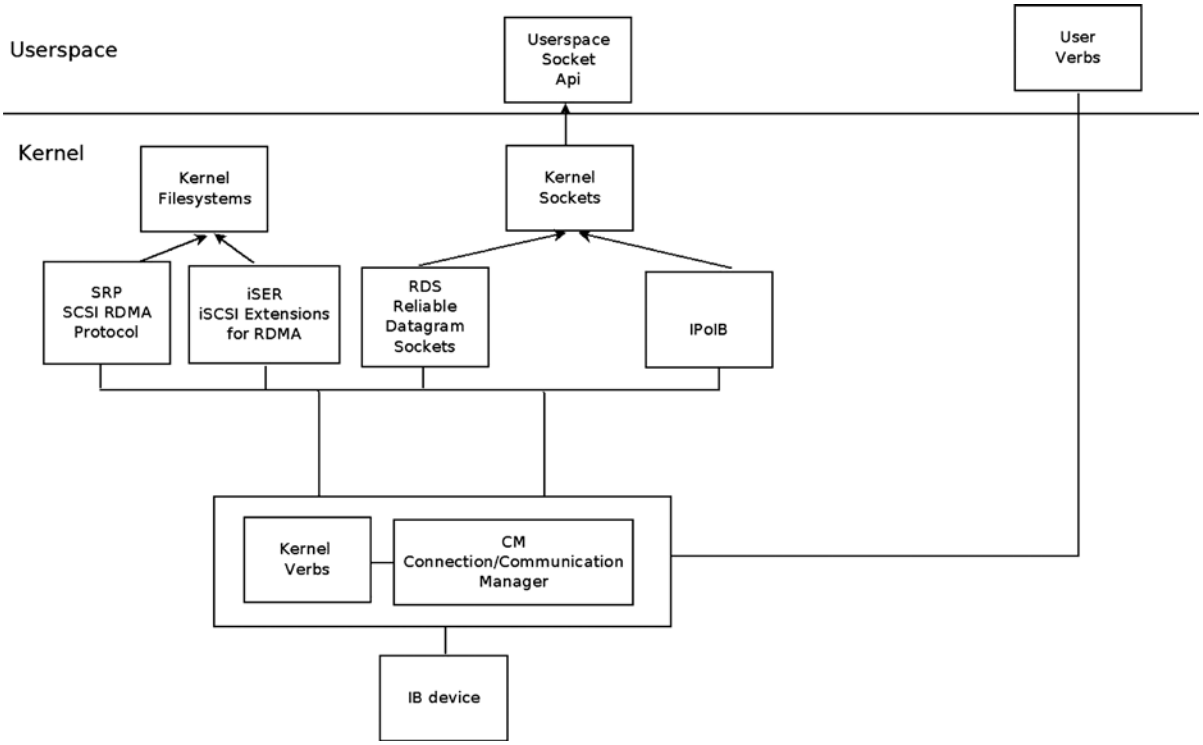

Figure 13-1. *Linux Infiniband stack architecture*

In this section, I covered the RDMA stack organization and the kernel modules that are part of it in the Linux kernel.

RDMA Technology Advantages

Here I will cover the advantages of the RDMA technology and explain the features that make it popular in many markets:

- **Zero copy:** The ability to directly write data to and read data from remote memory allows you to access remote buffers directly without the need to copy it between different software layers.
- **Kernel bypass:** Sending and receiving data from the same context of the code (that is, userspace or kernel level) saves the context switches time.
- **CPU offload:** The ability to send or receive data using dedicated hardware without any CPU intervention allows for decreasing the usage of the CPU on the remote side, because it doesn't perform any active operations.
- **Low latency:** RDMA technologies allow you to reach a very low latency for short messages. (In current hardware and on current servers, the latency for sending up to tens of bytes can be a couple of hundred nanoseconds.)
- **High Bandwidth:** In an Ethernet device, the maximum bandwidth is limited by the technology (that is, 10 or 40 Gbits/sec). In InfiniBand, the same protocol and equipment can be used from 2.5 Gbits/sec up to 120 Gbits/sec. (In current hardware and on current servers, the BW can be upto 56 Gbits/sec.)

InfiniBand Hardware Components

Like in any other interconnect technologies, in InfiniBand several hardware components are described in the spec, some of them are endpoints to the packets (generating packets and the target of the packet), and some of them forward packets in the same subnet or between different subnets. Here I will cover the most common ones:

- **Host Channel Adapter (HCA):** The network adapter that can be placed at a host or at any other system (for example, storage device). This component initiates or is the target of packets.
- **Switch:** A component that knows how to receive a packet from one port and send it to another port. If needed, it can duplicate multicast messages. (Broadcast isn't supported in InfiniBand.) Unlike other technologies, every switch is a very simple device with forwarding tables that are configured by the Subnet Manager (SM), which is an entity that configures and manages the subnet (later on in this section, I will discuss its role in more detail). The switch doesn't learn anything by itself or parse and anlyze packets; it forwards packets only within the same subnet.
- **Router:** A component that connects several different InfiniBand subnets.

A subnet is a set of HCAs, switches, and router ports that are connected together. In this section, I described the various hardware components in InfiniBand, and now I will discuss the addressing of the devices, system, and ports in InfiniBand.

Addressing in InfiniBand

Here are some rules about InfiniBand addressing and an example:

- In InfiniBand, the unique identifier of components is the Globally Unique Identifier (GUID), which is a 64-bit value that is unique in the world.
- Every node in the subnet has a Node GUID. This is the identifier of the node and a constant attribute of it.
- Every port in the subnet, including in HCAs and in switches, has a port GUID. This is the identifier of the port and a constant attribute of it.
- In systems that are made from several components, there can be a system GUID. All of the components in that system have the same system GUID.

Here is an example that demonstrates all the aforementioned GUIDs: a big switch system that is combined from several switch chips. Every switch chip has a unique Node GUID. Every port in every switch has a unique port GUID. All of the chips in that system have the same system GUID.

- Global IDentifier (GID) is used to identify an end port or a multicast group. Every port has at least one valid GID at the GID table in index 0. It is based on the port GUID plus the subnet identifier that this port is part of.
- Local IDentifier (LID) is a 16-bit value that is assigned to every subnet port by the Subnet Manager. A switch is an exception, and the switch management port has the LID assignment, and not all of its ports. Every port can be assigned only one LID, or a contiguous range of LIDs, in order to have several paths to this port. Each LID is unique at a specific point of time in the same subnet, and it is used by the switches when forwarding the packets to know which egress port to use. The unicast LID's range is 0x001 to 0xbfff. The multicast LIDs range is 0xc000 to 0xfffe.

InfiniBand Features

Here we will cover some of the InfiniBand protocol features:

- InfiniBand allows you to configure partitions of ports of HCAs, switches, and routers and allows you to provide virtual isolation over the same physical subnet. Every Partition Key (P_Key) is a 16-bit value that is combined from the following: 15 lsbs are the key value, and the msb is the membership level; 0 is limited membership; and 1 is full membership. Every port has a P_Key table that is being configured by the SM, and every Queue Pair (QP), the actual object in InfiniBand that sends and receives data, is associated with one P_Key index in this table. One QP can send or receive packets from a remote QP only if, in the P_Keys that each of them is associated with, the following is true:
  - The key value is equal.
  - At least one of them has full membership.
- **Queue Key (Q_Key):** An Unreliable Datagram (UD) QP will get unicast or multicast messages from a remote UD QP only if the Q_Key of the message is equal to the Q_Key value of this UD QP.
- **Virtual Lanes (VL):** This is a mechanism for creating multiple virtual links over a single physical link. Every virtual lane represents an autonomic set of buffers for send and receive packets in each port. The number of supported VLs is an attribute of a port.
- **Service Level (SL):** InfiniBand supports up to 16 service levels. The protocol doesn't specify the policy of each level. In InfiniBand, the QoS is implemented using the SL-to-VL mapping and the resources for each VL.
- **Failover:** Connected QPs are QPs that can send packets to or receive packets from only one remote QP. InfiniBand allows defining a primary path and an alternate path for connected QPs. If there is a problem with the primary path, instead of reporting an error, the alternate path will be used automatically.

In the next section, we will look at what packets in InfiniBand look like. This is very useful when you debug problems in InfiniBand.

InfiniBand Packets

Every packet in InfiniBand is a combination of several headers and, in many cases, a payload, which is the data of the messages that the clients want to send. Messages that contain only an ACK or messages with zero bytes (for example,

if only immediate data is being sent) won't contain a payload. Those headers describe from where the packet was sent, what the target of the packet is, the used operation, the information needed to separate the packets into messages, and enough information to detect packet loss errors.

Figure 13-2 presents the InfiniBand packet headers.

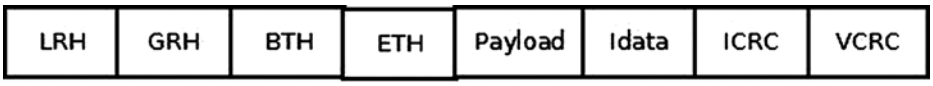

Figure 13-2. *InfiniBand packet headers*

Here are the headers in InfiniBand:

- **Local Routing Header (LRH):** 8 bytes. Always present. It identifies the local source and destination ports of the packet. It also specifies the requested QoS attributes (SL and VL) of the message.
- **Global Routing Header (GRH):** 40 bytes. Optional. Present for multicast packets or packets that travel in multiple subnets. It describes the source and destination ports using GIDs. Its format is identical to the IPv6 header.
- **Base Transport Header (BTH):** 12 bytes. Always present. It specifies the source and destination QPs, the operation, packet sequence number, and partition.
- **Extended Transport Header (ETH):** from 4 to 28 bytes. Optional. Extra family of headers that might be present, depending on the class of the service and the operation used.
- **Payload:** Optional. The data that the client wants to send.
- **Immediate data:** 4 bytes. Optional. Out-of-band, 32-bit value that can be added to Send and RDMA Write operations.
- **Invariant CRC (ICRC):** 4 bytes. Always present. It covers all fields that should not be changed as the packet travels in the subnet.
- **Variant CRC (VCRC):** 2 bytes. Always present. It covers all of the fields of the packet.

Management Entities

The SM is the entity in the subnet that is responsible for analyzing the subnet and configuring it. These are some of its missions:

- Discover the physical topology of the subnet.
- Assign the LIDs and other attributes—such as active MTU, active speeds, and more—to each port in the subnet.
- Configure the forwarding table in the subnet switches.
- Detect any changes in the topology (for example, if new nodes were added or removed from the subnet).
- Handle various errors in the subnet.

Subnet Manager is usually a software entity that can be running in a switch (which is called a *managed switch*) or in any node in the subnet.

Several SMs can be running in a subnet, but only one of them will be active and the rest of them will be in standby mode. There is an internal protocol that performs master election and decides which SM will be active. If the active SM is going down, one of the standby SMs will become the active SM. Every port in the subnet has a Subnet Management Agent (SMA), which is an agent that knows how to receive management messages sent by the SM, handle them, and return a response. Subnet Administrator (SA) is a service that is part of the SM. These are some of its missions:

- Provide information about the subnet—for example, information about how to get from one port to another (that is, a path query).
- Allow you to register to get notifications about events.
- Provide services for management of the subnet, such as joining or leaving a multicast. Those services might cause the SM to (re)configure the subnet.

Communication Manager (CM) is an entity that is capable of running on each port, if the port supports it, to establish, maintain, and tear down QP connections.

RDMA Resources

In the RDMA API, a lot of resources need to be created and handled before any data can be sent or received. All of the resources are in the scope of a specific RDMA device, those resources cannot be shared or used across more than one local device, even if there are multiple devices in the same machine. Figure 13-3 presents the RDMA resource creation hierarchy.

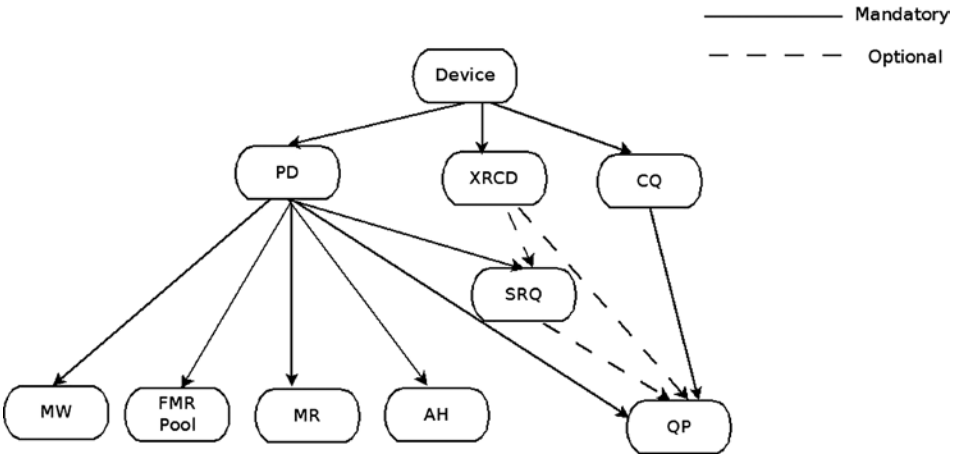

Figure 13-3. *RDMA resource creation hierarchy*

RDMA Device

The client needs to register with the RDMA stack in order to be notified about any RDMA device that is being added to the system or removed from it. After the initial registration, the client is notified for all existing RDMA devices. A callback will be invoked for every RDMA device, and the client can start working with these devices in the following ways:

- Query the device for various attributes
- Modify the device attributes
- Create, work with and destroy resources

The ib_register_client() method registers a kernel client that wants to use the RDMA stack. The specified callbacks will be invoked for every new InfiniBand device that currently exists in the system and that will be added to or removed from (using hot-plug functionality) the system. The ib_unregister_client() method unregisters a kernel client that wants to stop using the RDMA stack. Usually, it is called when the driver is being unloaded. Here is an sample code that shows how to register the RDMA stack in a kernel client:

```
static void my_add_one(struct ib_device *device)
{
...
}

static void my_remove_one(struct ib_device *device)
{
...
}

static struct ib_client my_client = {
    .name   = "my RDMA module",
    .add    = my_add_one,
    .remove = my_remove_one
};

static int __init my_init_module(void)
{
    int ret;

    ret = ib_register_client(&my_client);
    if (ret) {
        printk(KERN_ERR "Failed to register IB client\n");
        return ret;
    }

    return 0;
}

static void __exit my_cleanup_module(void)
{
    ib_unregister_client(&my_client);
}

module_init(my_init_module);
module_exit(my_cleanup_module);
```

Following here is a description of several more methods for handling an InfiniBand device.

- The ib_set_client_data() method sets a client context to be associated with an InfiniBand device.
- The ib_get_client_data() method returns the client context that was associated with an InfiniBand device using the ib_set_client_data() method.

- The `ib_register_event_handler()` method registers a callback to be called for every asynchronous event that will occur to the InfiniBand device. The callback structure must be initialized with the INIT_IB_EVENT_HANDLER macro.
- The `ib_unregister_event_handler()` method unregisters the event handler.
- The `ib_query_device()` method queries the InfiniBand device for its attributes. Those attributes are constant and won't be changed in subsequent calls of this method.
- The `ib_query_port()` method queries the InfiniBand device port for its attributes. Some of those attributes are constant, and some of them might be changed in subsequent calls of this method—for example, the port LID, state, and some other attributes.
- The `rdma_port_get_link_layer()` method returns the link layer of the device port.
- The `ib_query_gid()` method queries the InfiniBand device port's GID table in a specific index. The `ib_find_gid()` method returns the index of a specific GID value in a port's GID table.
- The `ib_query_pkey()` method queries the InfiniBand device port's P_Key table in a specific index. The `ib_find_pkey()` method returns the index of a specific P_Key value in a port's P_Key table.

Protection Domain (PD)

A PD allows associating itself with several other RDMA resources—such as SRQ, QP, AH, or MR—in order to provide a means of protection among them. RDMA resources that are associated with PDx cannot work with RDMA resources that were associated with PDy. Trying to mix those resources will end with an error. Typically, every module will have one PD. However, if a specific module wants to increase its security, it will use one PD for each remote QP or service that it uses. Allocation and deallocation of a PD is done like this:

- The `ib_alloc_pd()` method allocates a PD. It takes as an argument the pointer of the device object that was returned when the driver callback was called after its registration.
- The `ib_dealloc_pd()` method deallocates a PD. It is usually called when the driver is being unloaded or when the resources that are associated with the PD are being destroyed.

Address Handle (AH)

An AH is used in the Send Request of a UD QP to describe the path of the message from the local port to the remote port. The same AH can be used for several QPs if all of them send messages to the same remote port using the same attributes. Following is a description of four methods related to the AH:

- The `ib_create_ah()` method creates an AH. It takes as an argument a PD and attributes for the AH. The AH attributes of the AH can be filled directly or by calling the `ib_init_ah_from_wc()` method, which gets as a parameter a received Work Completion (`ib_wc` object) that includes the attributes of a successfully completed incoming message, and the port it was received from. Instead of calling the `ib_init_ah_from_wc()` method and then the `ib_create_ah()` method, one can call the `ib_create_ah_from_wc()` method.
- The `ib_modify_ah()` method modifies the attributes of an existing AH.
- The `ib_query_ah()` method queries for the attributes of an existing AH.
- The `ib_destroy_ah()` method destroys an AH. It is called when there isn't a need to send any further messages to the node that the AH describes the path to.

Memory Region (MR)

Every memory buffer that is accessed by the RDMA device needs to be registered. During the registration process, the following tasks are performed on the memory buffer:

- Separate the contiguous memory buffer to memory pages.
- The mapping of the virtual-to-physical translation will be done.
- The memory pages permission is checked to ensure that the requested permissions for the MR is supported by them.
- The memory pages are pinned, to prevent them from being swapped out. This keeps the virtual-to-physical mapping unchanged.

After a successful memory registration is completed, it has two keys:

- **Local key (lkey):** A key for accessing this memory by local Work Requests.
- **Remote key (rkey):** A key for accessing this memory by a remote machine using RDMA operations.

Those keys will be used in Work Requests when referring to those memory buffers. The same memory buffers can be registered several times, even with different permissions. The following is a description of some methods related to the MR:

- The `ib_get_dma_mr()` method returns a Memory Region for system memory that is usable for DMA. It takes a PD and the requested access permission for the MR as arguments.
- The `ib_dma_map_single()` method maps a kernel virtual address, that was allocated by the `kmalloc()` method family, to a DMA address. This DMA address will be used to access local and remote memory. The `ib_dma_mapping_error()` method should be used to check whether the mapping was successful.
- The `ib_dma_unmap_single()` method unmaps a DMA mapping that was done using `ib_dma_map_single()`. It should be called when this memory isn't needed anymore.

■ **Note** There are some more flavors of `ib_dma_map_single()` that allow the mapping of pages, mapping according to DMA attributes, mapping using a scatter/gather list, or mapping using a scatter/gather list with DMA attributes: `ib_dma_map_page()`, `ib_dma_map_single_attrs()`, `ib_dma_map_sg()`, and `ib_dma_map_sg_attrs()`. All of them have corresponding unmap functions.

Before accessing a DMA mapped memory, the following methods should be called:

- `ib_dma_sync_single_for_cpu()` if the DMA region is going to be accessed by the CPU, or `ib_dma_sync_single_for_device()` if the DMA region is going to be accessed by the InfiniBand device.
- The `ib_dma_alloc_coherent()` method allocates a memory block that can be accessed by the CPU and maps it for DMA.
- The `ib_dma_free_coherent()` method frees a memory block that was allocated using `ib_dma_alloc_coherent()`.

- The `ib_reg_phys_mr()` method takes a set of physical pages, registers them, and prepares a virtual address that can be accessed by an RDMA device. If you want to change it after it was created, you should call the `ib_rereg_phys_mr()` method.
- The `ib_query_mr()` method retrieves the attributes of a specific MR. Note that most low-level drivers do not implement this method.
- The `ib_dereg_mr()` method deregisters an MR.

Fast Memory Region (FMR) Pool

Registration of a Memory Region is a "heavy" procedure that might take some time to complete, and the context that performs it even might sleep if required resources aren't available when it is called. This behavior might be problematic when performed in certain contexts—for example, in the interrupt handler. Working with an FMR pool allows you to work with FMRs with registrations that are "lightweight" and can be registered in any context. The API of the FMR pool can be found in `include/rdma/ib_fmr_pool.h`.

Memory Window (MW)

Enabling a remote access to a memory can be done in two ways:

- Register a memory buffer with remote permissions enabled.
- Register a Memory Region and then bind a Memory Window to it.

Both of those ways will create a remote key (`rkey`) that can be used to access this memory with the specified permissions. However, if you wish to invalidate the `rkey` to prevent remote access to this memory, performing Memory Region deregistration might be a heavy procedure. Working with Memory Window on this Memory Region and binding or unbinding it when needed might provide a "lightweight" procedure for enabling and disabling remote access to memory. Following is a description of three methods related to the MW:

- The `ib_alloc_mw()` method allocates a Memory Window. It takes a PD and the MW type as arguments.
- The `ib_bind_mw()` method binds a Memory Window to a specified Memory Region with a specific address, size, and remote permissions by posting a special Work Request to a QP. It is called when you want to allow temporary remote access to its memory. A Work Completion in the Send Queue of the QP will be generated to describe the status of this operation. If `ib_bind_mw()` was called to a Memory Windows that is already bounded, to the same Memory Region or a different one, the previous binding will be invalidated.
- The `ib_dealloc_mw()` method deallocates the specified MW object.

Completion Queue (CQ)

Every posted Work Request, to either Send or Receive Queue, is considered outstanding until there is a corresponding Work Completion for it or for any Work Request that was posted after it. While the Work Request is outstanding, the content of the memory buffers that it points to is undetermined:

- If the RDMA device reads this memory and sends its content over the wire, the client cannot know if this buffer can be (re)used or released. If this is a reliable QP, a successful Work Completion means that the message was received by the remote side. If this is an unreliable QP, a successful Work Completion means that the message was sent.
- If the RDMA device writes a message to this memory, the client cannot know if this buffer contains the incoming message.

A Work Completion specifies that the corresponding Work Request was completed and provides some information about it: its status, the used opcode, its size, and so on. A CQ is an object that contains the Work Completions. The client needs to poll the CQ in order to read the Work Completions that it has. The CQ works on a first-in, first-out (FIFO) basis: the order of Work Completions that will be de-queued from it by the client will be according to the order that they were enqueued to the CQ by the RDMA device. The client can read the Work Completions in polling mode or request to get a notification when a new Work Completion is added to the CQ. A CQ cannot hold more Work Completions than its size. If more Work Completions than its capacity are added to it, a Work Completion with an error will be added, a CQ error asynchronous event will be generated, and all the Work Queues associated with it will get an error. Here are some methods related to the CQ:

- The `ib_create_cq()` method creates a CQ. It takes the following as its arguments: the pointer of the device object that was returned when the driver callback was called after its registration and the attributes for the CQ, including its size and the callbacks that will be called when there is an asynchronous event on this CQ or a Work Completion is added to it.
- The `ib_resize_cq()` method changes the size of a CQ. The new number of entries cannot be less than the number of the Work Completions that currently populate the CQ.
- The `ib_modify_cq()` method changes the moderation parameter for a CQ. A Completion event will be generated if at least a specific number of Work Completions enter the CQ or a timeout will expire. Using it might help reduce the number of interrupts that happen in an RDMA device.
- The `ib_peek_cq()` method returns the number of available Work Completions in a CQ.
- The `ib_req_notify_cq()` method requests that a Completion event notification be generated when the next Work Completion, or Work Completion that includes a solicited event indication, is added to the CQ. If no Work Completion is added to the CQ after the `ib_req_notify_cq()` method was called, no Completion event notification will occur.
- The `ib_req_ncomp_notif()` method requests that a Completion event notification be created when a specific number of Work Completions exists in the CQ. Unlike the `ib_req_notify_cq()` method, when calling the `ib_req_ncomp_notif()` method, a Completion event notification will be generated even if the CQ currently holds this number of Work Completions.
- The `ib_poll_cq()` method polls for Work Completions from a CQ. It reads the Work Completions from the CQ in the order they were added to it and removes them from it.

Here is an example of a code that empties a CQ—that is, reads all the Work Completions from a CQ, and checks their status:

```
struct ib_wc wc;
int num_comp = 0;

while (ib_poll_cq(cq, 1, &wc) > 0) {
    if (wc.status != IB_WC_SUCCESS) {
        printk(KERN_ERR "The Work Completion[%d] has a bad status %d\n",
                        num_comp, wc.status);
        return -EINVAL;
    }
    num_comp ++;
}
```

eXtended Reliable Connected (XRC) Domain

An XRC Domain is an object that is used to limit the XRC SRQs an incoming message can target. That XRC domain can be associated with several other RDMA resources that work with XRC, such as SRQ and QP.

Shared Receive Queue (SRQ)

An SRQ is a way for the RDMA architecture to be more scalable on the receive side. Instead of having a separate Receive Queue for every Queue Pair, there is a shared Receive Queue that all of the QPs are connected to. When they need to consume a Receive Request, they fetch it from the SRQ. Figure 13-4 presents QPs that are associated with an SRQ.

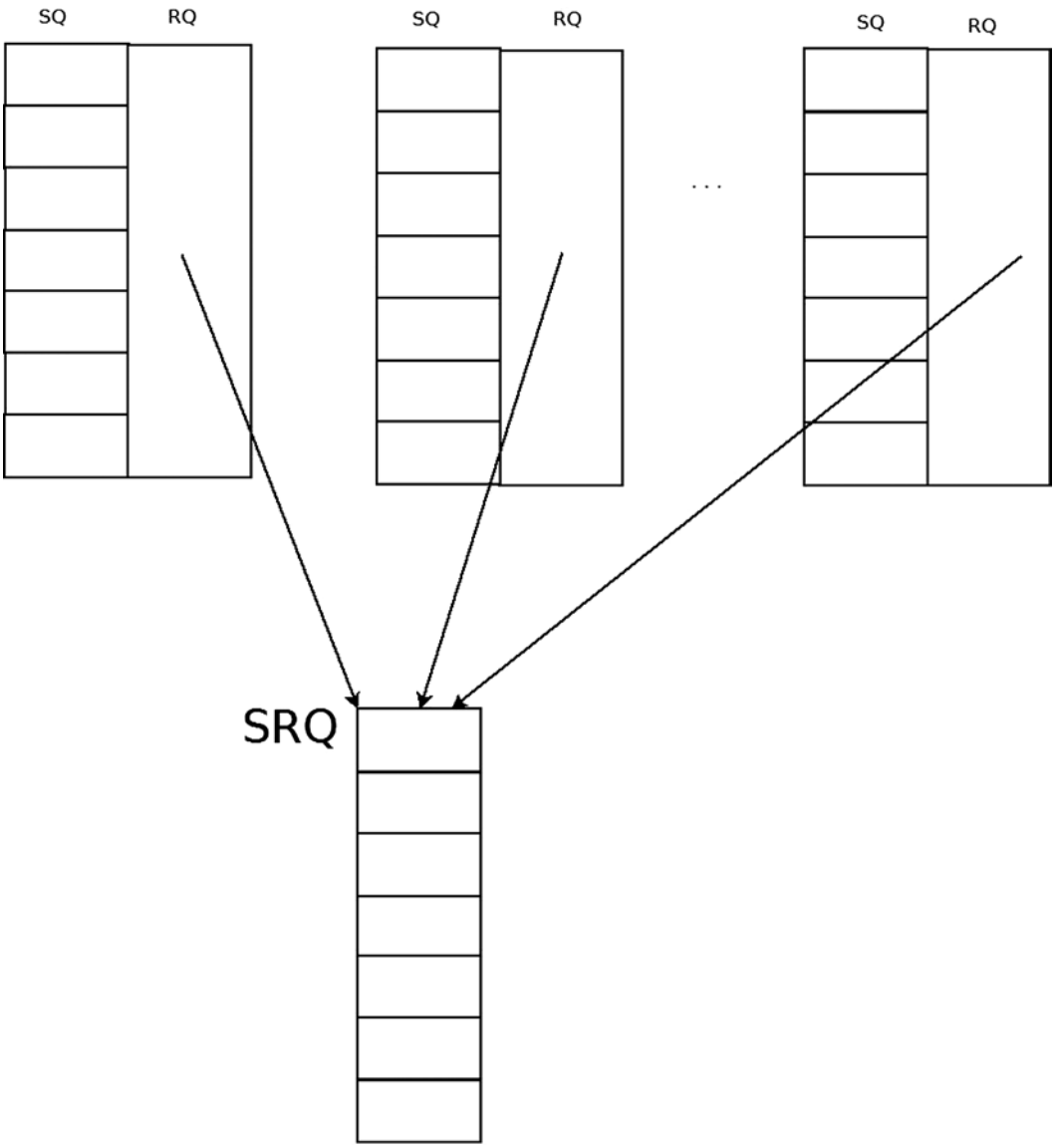

Figure 13-4. *QPs that are associated with an SRQ*

Here's what you do if you have *N* QPs, and each of them might receive a burst of *M* messages at a random time:

- Without using an SRQ, you post N*M Receive Requests.
- With SRQs, you post K*M (where K << N) Receive Requests.

Unlike a QP, which doesn't have any mechanism to determine the number of outstanding Work Requests in it, with an SRQ you can set a watermark limit. When the number of Receive Requests drops below this limit, an SRQ limit asynchronous event will be created for this SRQ. The downside of using an SRQ is that you cannot predict which QP will consume every posted Receive Request from the SRQ, so the message size that each posted Receive Request will be able to hold must be the maximum incoming message size that any of the QPs might get. This limitation can be handled by creating several SRQs, one for each different maximum message size, and associating them with the relevant QPs according to their expected message sizes.

Here is a description of some methods related to the SRQ and an example:

- The `ib_create_srq()` method creates an SRQ. It takes a PD and attributes for the SRQ.
- The `ib_modify_srq()` method modifies the attributes of the SRQ. It is used to set a new watermark value for the SRQ's limit event or to resize the SRQ for devices that support it.

Here is an example for setting the value of the watermark to get an asynchronous event when the number of RRs in the SRQ drops below 5:

```
struct ib_srq_attr srq_attr;
int ret;

memset(&srq_attr, 0, sizeof(srq_attr));
srq_attr.srq_limit = 5;

ret = ib_modify_srq(srq, &srq_attr, IB_SRQ_LIMIT);
if (ret) {
    printk(KERN_ERR "Failed to set the SRQ's limit value\n");
    return ret;
}
```

Following here is a description of several more methods for handling an SRQ.

- The `ib_query_srq()` method queries for the current SRQ attributes. This method is usually used to check the content of the SRQ's limit value. The value 0 in the `srq_limit` member in the `ib_srq_attr` object means that there isn't any SRQ limit watermark set.
- The `ib_destroy_srq()` method destroys an SRQ.
- The `ib_post_srq_recv()` method takes a linked list of Receive Requests as an argument and adds them to a specified Shared Receive Queue for future processing.

Here is an example for posting a single Receive Request to an SRQ. It saves an incoming message in a memory buffer, using its registered DMA address in a single gather entry:

```
struct ib_recv_wr wr, *bad_wr;
struct ib_sge sg;
int ret;

memset(&sg, 0, sizeof(sg));
sg.addr   = dma_addr;
sg.length = len;
sg.lkey   = mr->lkey;
```

```
memset(&wr, 0, sizeof(wr));
wr.next     = NULL;
wr.wr_id    = (uintptr_t)dma_addr;
wr.sg_list  = &sg;
wr.num_sge  = 1;

ret = ib_post_srq_recv(srq, &wr, &bad_wr);
if (ret) {
    printk(KERN_ERR "Failed to post Receive Request to an SRQ\n");
    return ret;
}
```

Queue Pair (QP)

Queue Pair is the actual object used to send and receive data in InfiniBand. It has two separate Work Queues: Send and Receive Queues. Every Work Queue has a specific number of Work Requests (WR) that can be posted to it, a number of scatter/gather elements that are supported for each WR, and a CQ to which the Work Requests whose processing has ended will add Work Completion. Those Work Queues can be created with similar or different attributes—for example, the number of WRs that can be posted to each Work Queue. The order in each Work Queue is guaranteed—that is, the processing of a Work Request in the Send Queue will start according to the order of the Send Requests submission. And the same behavior applies to the Receive Queue. However, there isn't any relation between them—that is, an outstanding Send Request can be processed even if it was posted after posting a Receive Request to the Receive Queue. Figure 13-5 presents a QP.

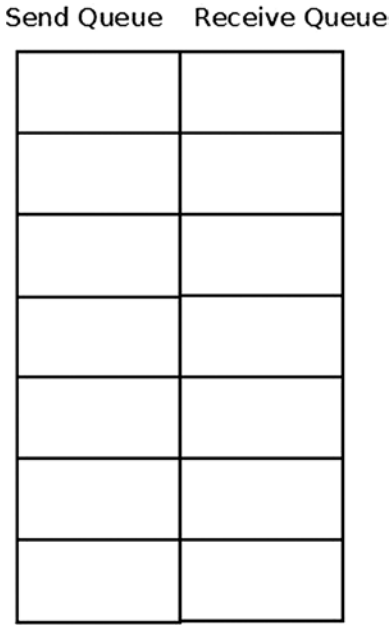

Figure 13-5. *QP (Queue Pair)*

Upon creation, every QP has a unique number across the RDMA device at a specific point in time.

QP Transport Types

There are several QP transport types supported in InfiniBand:

- **Reliable Connected (RC):** One RC QP is connected to a single remote RC QP, and reliability is guaranteed—that is, the arrival of all packets according to their order with the same content that they were sent with is guaranteed. Every message is fragmented to packets with the size of the path MTU at the sender side and defragmented at the receiver side. This QP supports Send, RDMA Write, RDMA Read, and Atomic operations.
- **Unreliable Connected (UC):** One UC QP is connected to a single remote UC QP, and reliability isn't guaranteed. Also, if a packet in a message is lost, the whole message is lost. Every message is fragmented to packets with the size of the path MTU at the sender side and defragmented at the receiver side. This QP supports Send and RDMA Write operations.
- **Unreliable Datagram (UD):** One UD QP can send a unicast message to any UD QP in the subnet. Multicast messages are supported. Reliability isn't guaranteed. Every message is limited to one packet message, with its size limited to the path MTU size. This QP supports only Send operations.
- **eXtended Reliable Connected (XRC):** Several QPs from the same node can send messages to a remote SRQ in a specific node. This is useful for decreasing the number of QPs between two nodes from the order of the number of CPU cores—that is, QP in a process per core, to one QP. This QP supports all operations that are supported by RC QP. This type is relevant only for userspace applications.
- **Raw packet:** Allows the client to build a complete packet, including the L2 headers, and send it as is. At the receiver side, no header will be stripped by the RDMA device.
- **Raw IPv6/Raw Ethertype:** QPs that allow sending raw packets that aren't interpreted by the IB device. Currently, both of these types aren't supported by any RDMA device.

There are special QP transport types that are used for subnet management and special services:

- **SMI/QP0:** QP used for subnet managements packets.
- **GSI/QP1:** QP used for general services packets.

The `ib_create_qp()` method creates a QP. It takes a PD and the requested attributes that this QP will be created with as arguments. Here is an example for creating an RC QP using a PD that was created, with two different CQs: one for the Send Queue and one for the Receive Queue.

```
struct ib_qp_init_attr init_attr;
struct ib_qp *qp;

memset(&init_attr, 0, sizeof(init_attr));
init_attr.event_handler       = my_qp_event;
init_attr.cap.max_send_wr     = 2;
init_attr.cap.max_recv_wr     = 2;
init_attr.cap.max_recv_sge    = 1;
init_attr.cap.max_send_sge    = 1;
init_attr.sq_sig_type         = IB_SIGNAL_ALL_WR;
init_attr.qp_type             = IB_QPT_RC;
init_attr.send_cq             = send_cq;
init_attr.recv_cq             = recv_cq;
```

```
qp = ib_create_qp(pd, &init_attr);
if (IS_ERR(qp)) {
    printk(KERN_ERR "Failed to create a QP\n");
    return PTR_ERR(qp);
}
```

QP State Machine

A QP has a state machine that defines what the QP is capable of doing at each state:

- **Reset state:** Each QP is generated at this state. At this state, no Send Requests or Receive Requests can be posted to it. All incoming messages are silently dropped.
- **Initialized state:** At this state, no Send Requests can be posted to it. However, Receive Requests can be posted, but they won't be processed. All incoming messages are silently dropped. It is a good practice to post a Receive Request to a QP at this state before moving it to RTR (Ready To Receive). Doing this prevents a case where remote QP sends messages need to consume a Receive Request but such were not posted yet.
- **Ready To Receive (RTR) state:** At this state, no Send Requests can be posted to it, but Receive Requests can be posted and processed. All incoming messages will be handled. The first incoming message that is received at this state will generate the communication-established asynchronous event. A QP that only receives messages can stay at this state.
- **Ready To Send (RTS) state**: At this state, both Send Requests and Receive Requests can be posted and processed. All incoming messages will be handled. This is the common state for QPs.
- **Send Queue Drained (SQD) state:** At this state, the QP completes the processing of all the Send Requests that their processing has started. Only when there aren't any messages that can be sent, you can change some of the QP attributes. This state is separated into two internal states:
  - **Draining:** Messages are still being sent.
  - **Drained:** The sending of the messages was completed.
- **Send Queue Error (SQE) state:** The RDMA device automatically moves a QP to this state when there is an error in the Send Queue for unreliable transport types. The Send Request that caused the error will be completed with the error reason, and all of the consecutive Send Requests will be flushed. The Receive Queue will still work—that is, Receive Requests can be posted, and incoming messages will be handled. The client can recover from this state and modify the QP state back to RTS.
- **Error state:** At this state, all of the outstanding Work Requests will be flushed. The RDMA device can move the QP to this state if this is a reliable transport type and there was an error with a Send Request, or if there was an error in the Receive Queue regardless of which transport type was used. All incoming messages are silently dropped.

A QP can be transitioned by `ib_modify_qp()` from any state to the Reset state and to the Error state. Moving the QP to the Error state will flush all of the outstanding Work Requests. Moving the QP to the Reset state will clear all previously configured attributes and remove all of the outstanding Work Request and Work Completions that were ended on this QP in the Completion Queues that this QP is working with. Figure 13-6 presents a QP state machine diagram.

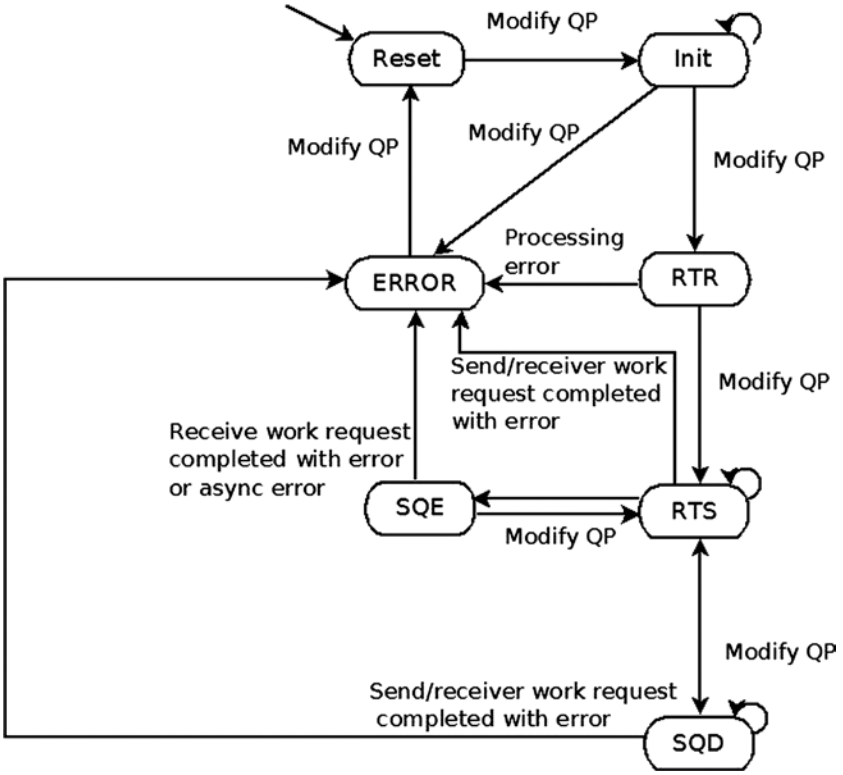

Figure 13-6. *QP state machine*

The `ib_modify_qp()` method modifies the attributes of a QP. It takes as an argument the QP to modify and the attributes of the QP that will be modified. The state machine of the QP can be changed according to the diagram shown in Figure 13-6. Every QP transport type requires different attributes to be set in each QP state transition.

Here is an example for modifying a newly created RC QP to the RTS state, in which it can send and receive packets. The local attributes are the outgoing port, the used SL, and the starting Packet Serial Number for the Send Queue. The remote attributes needed are the Receive PSN, the QP number, and the LID of the port that it uses.

```
struct ib_qp_attr attr = {
    .qp_state        = IB_QPS_INIT,
    .pkey_index      = 0,
    .port_num        = port,
    .qp_access_flags = 0
};

ret = ib_modify_qp(qp, &attr,
          IB_QP_STATE         |
          IB_QP_PKEY_INDEX    |
          IB_QP_PORT          |
          IB_QP_ACCESS_FLAGS);
```

```
if (ret) {
          printk(KERN_ERR "Failed to modify QP to INIT state\n");
          return ret;
}

attr.qp_state              = IB_QPS_RTR;
attr.path_mtu              = mtu;
attr.dest_qp_num           = remote->qpn;
attr.rq_psn                = remote->psn;
attr.max_dest_rd_atomic    = 1;
attr.min_rnr_timer         = 12;
attr.ah_attr.is_global     = 0;
attr.ah_attr.dlid          = remote->lid;
attr.ah_attr.sl            = sl;
attr.ah_attr.src_path_bits = 0,
attr.ah_attr.port_num      = port

ret = ib_modify_qp(ctx->qp, &attr,
          IB_QP_STATE                |
          IB_QP_AV                   |
          IB_QP_PATH_MTU             |
          IB_QP_DEST_QPN             |
          IB_QP_RQ_PSN               |
          IB_QP_MAX_DEST_RD_ATOMIC   |
          IB_QP_MIN_RNR_TIMER);
if (ret) {
  printk(KERN_ERR "Failed to modify QP to RTR state\n");
  return ret;
}

attr.qp_state       = IB_QPS_RTS;
attr.timeout        = 14;
attr.retry_cnt      = 7;
attr.rnr_retry      = 6;
attr.sq_psn         = my_psn;
attr.max_rd_atomic  = 1;
ret = ib_modify_qp(ctx->qp, &attr,
          IB_QP_STATE             |
          IB_QP_TIMEOUT           |
          IB_QP_RETRY_CNT         |
          IB_QP_RNR_RETRY         |
          IB_QP_SQ_PSN            |
          IB_QP_MAX_QP_RD_ATOMIC);
if (ret) {
  printk(KERN_ERR "Failed to modify QP to RTS state\n");
  return ret;
}
```

Following here is a description of several more methods for handling a QP:

- The `ib_query_qp()` method queries for the current QP attributes. Some of the attributes are constant (the values that the client specifies), and some of them can be changed (for example, the state).
- The `ib_destroy_qp()` method destroys a QP. It is called when the QP isn't needed anymore.

Work Request Processing

Every posted Work Request, to either the Send or Receive Queue, is considered outstanding until there is a Work Completion, which was polled from the CQ which is associated with this Work Queue for this Work Request or for Work Requests in the same Work Queue that were posted after it. Every outstanding Work Request in the Receive Queue will end with a Work Completion. A Work Request processing flow in a Work Queue is according to the diagram shown in Figure 13-7.

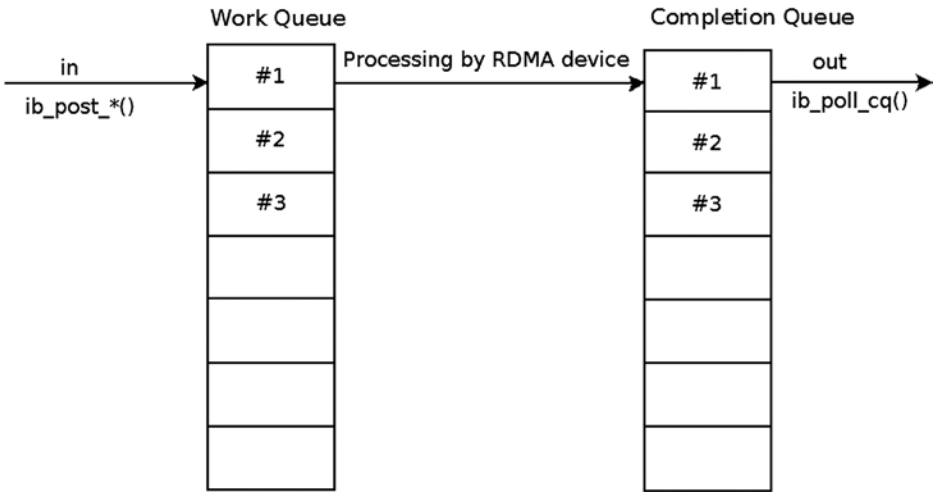

Figure 13-7. *Work Request processing flow*

In the Send Queue, you can choose (when creating a QP) whether you want every Send Request to end with a Work Completion or whether you want to select the Send Requests that will end with Work Completions—that is, selective signaling. You might encounter an error for an unsignaled Send Request; nevertheless, a Work Completion with bad status will be generated for it.

When a Work Request is outstanding one cannot (re)use or free the resources that were specified in it when posting this Work Request. For example:

- When posting a Send Request for a UD QP, the AH cannot be freed.
- When posting a Receive Request, the memory buffers that were referred to in a scatter/gather (s/g) list cannot be read, because it is unknown if the RDMA device already wrote the data in them.

"Fencing" is the ability to prevent the processing of a specific Send Request until the processing of the previous RDMA Read and Atomic operations ends. Adding the Fence indication to a Send Request can be useful, for example, when using RDMA Read from a remote address and sending the data, or part of it, in the same Send Queue. Without fencing, the send operation might start before the data is retrieved and available in local memory. When posting a Send Request to a UC or RC QP, the path to the target is known, because it was provided when moving the QP to the RTR state. However, when posting a Send Request to a UD QP, you need to add an AH to describe the path to the target(s) of this message. If there is an error related to the Send Queue, and if this is an Unreliable transport type, the Send Queue will move to the Error state (that is, the SQE state) but the Receive Queue will still be fully functional. The client can recover from this state and change the QP state back to RTS. If there is an error related to the Receive Queue, the QP will be moved to the Error state because this is an unrecoverable error. When a Work Queue is moved to the Error state, the Work Request that caused the error is ended with a status that indicates the nature of the error and the rest of the Work Requests in this Queue are flushed with error.

Supported Operations in the RDMA Architecture

There are several operation types supported in InfiniBand:

- **Send:** Send a message over the wire. The remote side needs to have a Receive Request available, and the message will be written in its buffers.
- **Send with Immediate:** Send a message over the wire with an extra 32 bits of out-of-band data. The remote side needs to have a Receive Request available, and the message will be written in its buffers. This immediate data will be available in the Work Completion of the receiver.
- **RDMA Write:** Send a message over the wire to a remote address.
- **RDMA Write with Immediate:** Send a message over the wire, and write it to a remote address. The remote side needs to have a Receive Request available. This immediate data will be available in the Work Completion of the receiver. This operation can be seen as RDMA Write + Send with immediate with a zero-byte message.
- **RDMA Read:** Read a remote address, and fill the local buffer with its content.
- **Compare and Swap:** Compare the content of a remote address with valueX; if they are equal, replace its content with the valueY. All of this is performed in an atomic way. The original remote memory content is sent and saved locally.
- **Fetch and Add:** Add a value to the content of a remote address in an atomic way. The original remote memory content is sent and saved locally.
- **Masked Compare and Swap:** Compare the part of the content using maskX of a remote address with valueX; if they are equal, replace part of its content using the bits in maskY with valueY. All of this is performed in an atomic way. The original remote memory content is sent and saved locally.
- **Masked Fetch and Add:** Add a value to the content of a remote address in an atomic way, and change only the bits that are specified in the mask. The original remote memory content is sent and saved locally.
- **Bind Memory Window:** Binds a Memory Windows to a specific Memory Region.
- **Fast registration:** Registers a Fast Memory Region using a Work Request.
- **Local invalidate:** Invalidates a Fast Memory Region using a Work Request. If someone uses its old `lkey`/`rkey`, it will be considered an error. It can be combined with send/RDMA read; in such a case, first the send/read will be performed, and only then this Fast Memory Region will be invalidated.

The Receive Request specifies where the incoming message will be saved for operations that consume a Receive Request. The total size of the memory buffers specified in the scatter list must be equal to or greater than the size of the incoming message.

For UD QP, because the origin of the message is unknown in advance (same subnet or another subnet, unicast or multicast message), an extra 40 bytes, which is the GRH header size, must be added to the Receive Request buffers. The first 40 bytes will be filled with the GRH of the message, if such is available. This GRH information describes how to send a message back to the sender. The message itself will start at offset 40 in the memory buffers that were described in the scatter list.

The `ib_post_recv()` method takes a linked list of Receive Requests and adds them to the Receive Queue of a specific QP for future processing. Here is an example for posting a single Receive Request for a QP. It saves an incoming

message in a memory buffer using its registered DMA address in a single gather entry. qp is a pointer to a QP that was created using ib_create_qp(). The memory buffer is a block that was allocated using kmalloc() and mapped for DMA using ib_dma_map_single(). The used lkey is from the MR that was registered using ib_get_dma_mr().

```
struct ib_recv_wr wr, *bad_wr;
struct ib_sge sg;
int ret;

memset(&sg, 0, sizeof(sg));
sg.addr   = dma_addr;
sg.length = len;
sg.lkey   = mr->lkey;

memset(&wr, 0, sizeof(wr));
wr.next     = NULL;
wr.wr_id    = (uintptr_t)dma_addr;
wr.sg_list  = &sg;
wr.num_sge  = 1;

ret = ib_post_recv(qp, &wr, &bad_wr);

if (ret) {
    printk(KERN_ERR "Failed to post Receive Request to a QP\n");
    return ret;
}
```

The ib_post_send() method takes as an argument a linked list of Send Requests and adds them to the Send Queue of a specific QP for future processing. Here is an example for posting a single Send Request of a Send operation for a QP. It sends the content of a memory buffer using its registered DMA address in a single gather entry.

```
struct ib_sge sg;
struct ib_send_wr wr, *bad_wr;
int ret;

memset(&sg, 0, sizeof(sg));
sg.addr   = dma_addr;
sg.length = len;
sg.lkey   = mr->lkey;

memset(&wr, 0, sizeof(wr));
wr.next       = NULL;
wr.wr_id      = (uintptr_t)dma_addr;
wr.sg_list    = &sg;
wr.num_sge    = 1;
wr.opcode     = IB_WR_SEND;
wr.send_flags = IB_SEND_SIGNALED;

ret = ib_post_send(qp, &wr, &bad_wr);
```

```
if (ret) {
    printk(KERN_ERR "Failed to post Send Request to a QP\n");
    return ret;
}
```

Work Completion Status

Every Work Completion can be ended successfully or with an error. If it ends successfully, the operation was finished and the data was sent according to the transport type reliability level. If this Work Completion contains an error, the content of the memory buffers is unknown. There can be many reasons that the Work Request status indicates that there is an error: protection violation, bad address, and so on. The violation errors won't perform any retransmission. However, there are two special retry flows that are worth mentioning. Both of them are done automatically by the RDMA device, which retransmit packets, until the problem is solved or it exceeds the number of retransmissions. If the issue was solved, the client code won't be aware that this even happened, besides a temporary performance hiccup. This is relevant only for Reliable transport types.

Retry Flow

If the receiver side didn't return any ACK or NACK to the sender side within the expected timeout, the sender might send the message again, according to the timeout and the retry count attributes that were configured in the QP attributes. There might be several reasons for having such a problem:

- The attributes of the remote QP or the path to it aren't correct.
- The remote QP state didn't get to (at least) the RTR state.
- The remote QP state moved to the Error state.
- The message itself was dropped on the way from the sender to the receiver (for example, a CRC error).
- The ACK or NACK of messages was dropped on the way from the receiver to the sender (for example, a CRC error).

Figure 13-8 presents the retry flow becasue of a packet loss that overcame a packet drop.

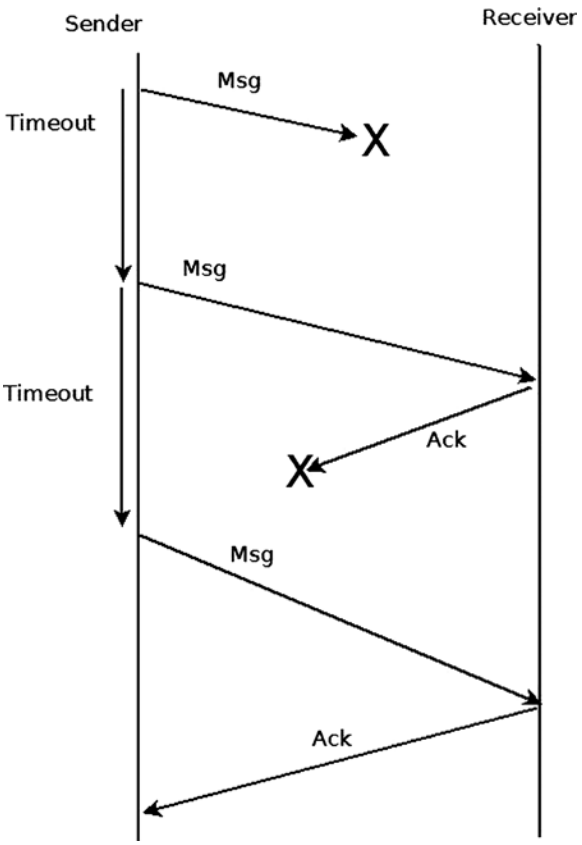

Figure 13-8. *A retry flow (on reliable transport types)*

If eventually the ACK/NACK is received by the sender QP successfully, it will continue to send the rest of the messages. If any message in the future has this problem too, the retry flow will be done again for this message as well, without any history that this was done before. If even after retrying several times the receiver side still doesn't respond, there will be a Work Completion with Retry Error on the sender side.

Receiver Not Ready (RNR) Flow

If the receiver side got a message that needs to consume a Receive Request from the Receiver Queue, but there isn't any outstanding Receive Request, the receiver will send back to the sender an RNR NACK. After a while, according to the time that was specified in the RNR NACK, the sender will try to send the message again.

If eventually the receiver side posts a Receiver Request in time, and the incoming message consumes it, an ACK will be sent to the sender side to indicate that the message was saved successfully. If any message in the future has this problem too, the RNR retry flow will be done again for this message as well, without any history that this was done before. If even after retrying several times the receiver side still didn't post a Receiver Request and an RNR NACK was sent to the sender for each sent message, a Work Completion with RNR Retry Error will be generated on the sender side. Figure 13-9 presents the RNR retry flow of retry that overcome a missing Receive Request in he receiver side.

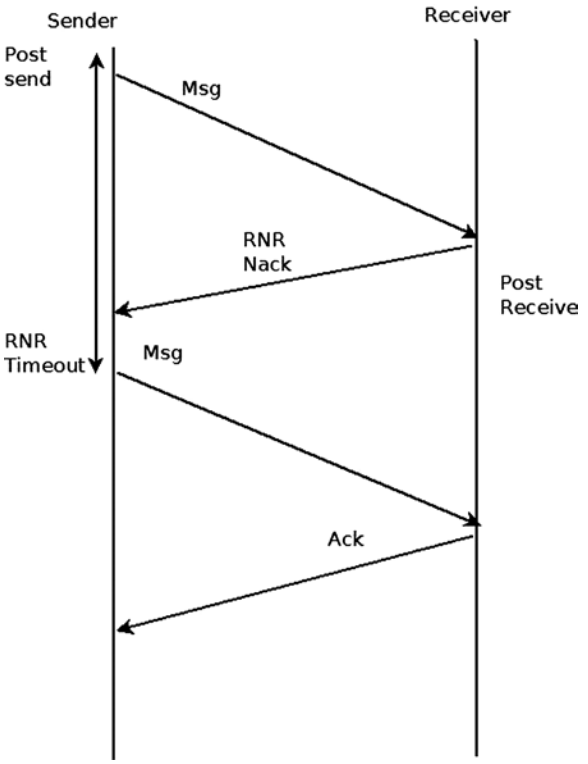

Figure 13-9. *RNR retry flow (on reliable transport types)*

In this section, I covered the Work Request status and some of the bad flows that can happen to a message. In the next section, I will discuss the multicast groups.

Multicast Groups

Multicast groups are a means to send a message from one UD QP to many UD QPs. Every UD QP that wants to get this message needs to be attached to the multicast group. When a device gets a multicast packet, it duplicates it to all of the QPs that are attached to that group. Following is a description of two methods related to multicast groups:

- The `ib_attach_mcast()` method attaches a UD QP to a multicast group within an InfiniBand device. It accepts the QP to be attached and the multicast group attributes.
- The `ib_detach_mcast()` method detaches a UD QP from a multicast group.

Difference Between the Userspace and the Kernel-Level RDMA API

The userspace and the kernel level of the RDMA stack API are quite similar, because they cover the same technology and need to be able to provide the same functionality. When the userspace is calling a method of the control path from the RDMA API, it performs a context switch to the kernel level to protect privileged resources and to synchronize objects that need to be synchronized (for example, the same QP number cannot be assigned to more than one QP at the same time).

However, there are some differences between the userspace and the kernel-level RDMA API and functionality:

- The prefix of all the APIs in the kernel level is “ib_”, while in the userspace the prefix is “ibv_”.
- There are enumerations and macros that exist only in the RDMA API in the kernel level.
- There are QP types that are available only in the kernel (for example, the SMI and GSI QPs).
- There are privileged operations that can be performed only in the kernel level—for example, registration of a physical memory, registration of an MR using a WR, and FMRs.
- Some functionality isn’t available in the RDMA API in the userspace—for example, Request for N notification.
- The kernel API is asynchronous. There are callbacks that are called when there is an asynchronous event or Completion event. In the userspace, everything is synchronous and the user needs to explicitly check if there is an asynchronous event or Completion event in its running context (that is, thread).
- XRC isn’t relevant for kernel-level clients.
- There are new features that were introduced to the kernel level, but they are not available (yet) in the userspace.

The userspace API is supplied by the userspace library “libibverbs.” And although some of the RDMA functionality in the user level is less than the kernel-level one, it is enough to enjoy the benefits of the InfiniBand technology.

Summary

You have learned in this chapter about the advantages of the InfiniBand technology. I reviewed the RDMA stack organization. I discussed the resource-creation hierarchy and all of the important objects and their API, which is needed in order to write client code that uses InfiniBand. You also saw some examples that use this API. The next chapter will deal with advanced topics like network namespaces and the Bluetooth subsystem.

Quick Reference

I will conclude this chapter with a short list of important methods of the RDMA API. Some of them were mentioned in this chapter.

Methods

Here are the methods.

int ib_register_client(struct ib_client *client);

Register a kernel client that wants to use the RDMA stack.

void ib_unregister_client(struct ib_client *client);

Unregister a kernel client that wants to stop using the RDMA stack.

void ib_set_client_data(struct ib_device *device, struct ib_client *client, void *data);

Set a client context to be associated with an InfiniBand device.

void *ib_get_client_data(struct ib_device *device, struct ib_client *client);

Read the client context that was associated with an InfiniBand device.

int ib_register_event_handler(struct ib_event_handler *event_handler);

Register a callback to be called for every asynchronous event that occurs to the InfiniBand device.

int ib_unregister_event_handler(struct ib_event_handler *event_handler);

Unregister a callback to be called for every asynchronous event that occurs to the InfiniBand device.

int ib_query_device(struct ib_device *device, struct ib_device_attr *device_attr);

Query an InfiniBand device for its attributes.

int ib_query_port(struct ib_device *device, u8 port_num, struct ib_port_attr *port_attr);

Query an InfiniBand device port for its attributes.

enum rdma_link_layer rdma_port_get_link_layer(struct ib_device *device, u8 port_num);

Query for the link layer of the InfiniBand device's port.

int ib_query_gid(struct ib_device *device, u8 port_num, int index, union ib_gid *gid);

Query for the GID in a specific index in the InfiniBand device's port GID table.

int ib_query_pkey(struct ib_device *device, u8 port_num, u16 index, u16 *pkey);

Query for the P_Key-specific index in the InfiniBand device's port P_Key table.

int ib_find_gid(struct ib_device *device, union ib_gid *gid, u8 *port_num, u16 *index);

Find the index of a specific GID value in the InfiniBand device's port GID table.

int ib_find_pkey(struct ib_device *device, u8 port_num, u16 pkey, u16 *index);

Find the index of a specific P_Key value in the InfiniBand device's port P_Key table.

struct ib_pd *ib_alloc_pd(struct ib_device *device);

Allocate a PD to be used later to create other InfiniBand resources.

int ib_dealloc_pd(struct ib_pd *pd);

Deallocate a PD.

struct ib_ah *ib_create_ah(struct ib_pd *pd, struct ib_ah_attr *ah_attr);

Create an AH that will be used when posting a Send Request in a UD QP.

int ib_init_ah_from_wc(struct ib_device *device, u8 port_num, struct ib_wc *wc, struct ib_grh *grh, struct ib_ah_attr *ah_attr);

Initializes an AH attribute from a Work Completion of a received message and a GRH buffer. Those AH attributes can be used when calling the `ib_create_ah()` method.

struct ib_ah *ib_create_ah_from_wc(struct ib_pd *pd, struct ib_wc *wc, struct ib_grh *grh, u8 port_num);

Create an AH from a Work Completion of a received message and a GRH buffer.

int ib_modify_ah(struct ib_ah *ah, struct ib_ah_attr *ah_attr);

Modify the attributes of an existing AH.

int ib_query_ah(struct ib_ah *ah, struct ib_ah_attr *ah_attr);

Query the attributes of an existing AH.

int ib_destroy_ah(struct ib_ah *ah);

Destroy an AH.

struct ib_mr *ib_get_dma_mr(struct ib_pd *pd, int mr_access_flags);

Return an MR system memory that is usable for DMA.

static inline int ib_dma_mapping_error(struct ib_device *dev, u64 dma_addr);

Check if the DMA memory points to an invalid address—that is, check whether the DMA mapping operation failed.

static inline u64 ib_dma_map_single(struct ib_device *dev, void *cpu_addr, size_t size, enum dma_data_direction direction);

Map a kernel virtual address to a DMA address.

static inline void ib_dma_unmap_single(struct ib_device *dev, u64 addr, size_t size, enum dma_data_direction direction);

Unmap a DMA mapping of a virtual address.

static inline u64 ib_dma_map_single_attrs(struct ib_device *dev, void *cpu_addr, size_t size, enum dma_data_direction direction, struct dma_attrs *attrs)

Map a kernel virtual memory to a DMA address according to DMA attributes.

static inline void ib_dma_unmap_single_attrs(struct ib_device *dev, u64 addr, size_t size, enum dma_data_direction direction, struct dma_attrs *attrs);

Unmap a DMA mapping of a virtual address that was mapped according to DMA attributes.

static inline u64 ib_dma_map_page(struct ib_device *dev, struct page *page, unsigned long offset, size_t size, enum dma_data_direction direction);

Maps a physical page to a DMA address.

static inline void ib_dma_unmap_page(struct ib_device *dev, u64 addr, size_t size, enum dma_data_direction direction);

Unmap a DMA mapping of a physical page.

static inline int ib_dma_map_sg(struct ib_device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction);

Map a scatter/gather list to a DMA address.

static inline void ib_dma_unmap_sg(struct ib_device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction);

Unmap a DMA mapping of a scatter/gather list.

static inline int ib_dma_map_sg_attrs(struct ib_device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction, struct dma_attrs *attrs);

Map a scatter/gather list to a DMA address according to DMA attributes.

static inline void ib_dma_unmap_sg_attrs(struct ib_device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction, struct dma_attrs *attrs);

Unmap a DMA mapping of a scatter/gather list according to DMA attributes.

static inline u64 ib_sg_dma_address(struct ib_device *dev, struct scatterlist *sg);

Return the address attribute of a scatter/gather entry.

static inline unsigned int ib_sg_dma_len(struct ib_device *dev, struct scatterlist *sg);

Return the length attribute of a scatter/gather entry.

static inline void ib_dma_sync_single_for_cpu(struct ib_device *dev, u64 addr, size_t size, enum dma_data_direction dir);

Transfer a DMA region ownership to the CPU. It should be called before the CPU accesses a DMA mapped region whose ownership was previously transferred to the device.

static inline void ib_dma_sync_single_for_device(struct ib_device *dev, u64 addr, size_t size, enum dma_data_direction dir);

Transfer a DMA region ownership to the device. It should be called before the device accesses a DMA mapped region whose ownership was previously transferred to the CPU.

static inline void *ib_dma_alloc_coherent(struct ib_device *dev, size_t size, u64 *dma_handle, gfp_t flag);

Allocate a memory block that can be accessed by the CPU, and map it for DMA.

static inline void ib_dma_free_coherent(struct ib_device *dev, size_t size, void *cpu_addr, u64 dma_handle);

Free a memory block that was allocated using `ib_dma_alloc_coherent()`.

struct ib_mr *ib_reg_phys_mr(struct ib_pd *pd, struct ib_phys_buf *phys_buf_array, int num_phys_buf, int mr_access_flags, u64 *iova_start);

Take a physical page list, and prepare it for being accessed by the InfiniBand device.

int ib_rereg_phys_mr(struct ib_mr *mr, int mr_rereg_mask, struct ib_pd *pd, struct ib_phys_buf *phys_buf_array, int num_phys_buf, int mr_access_flags, u64 *iova_start);

Change the attributes of an MR.

int ib_query_mr(struct ib_mr *mr, struct ib_mr_attr *mr_attr);

Query for the attributes of an MR.

int ib_dereg_mr(struct ib_mr *mr);

Deregister an MR.

struct ib_mw *ib_alloc_mw(struct ib_pd *pd, enum ib_mw_type type);

Allocate an MW. This MW will be used to allow remote access to an MR.

static inline int ib_bind_mw(struct ib_qp *qp, struct ib_mw *mw, struct ib_mw_bind *mw_bind);

Bind an MW to an MR to allow a remote access to local memory with specific permissions.

int ib_dealloc_mw(struct ib_mw *mw);

Deallocates an MW.

struct ib_cq *ib_create_cq(struct ib_device *device, ib_comp_handler comp_handler, void (*event_handler)(struct ib_event *, void *), void *cq_context, int cqe, int comp_vector);

Create a CQ. This CQ will be used to indicate the status of ended Work Requests for Send or Receive Queues.

int ib_resize_cq(struct ib_cq *cq, int cqe);

Change the number of entries in a CQ.

int ib_modify_cq(structib_cq *cq, u16 cq_count, u16 cq_period);

Modify the moderation attributes of a CQ. This method is used to decrease the number of interrupts of an InfiniBand device.

int ib_peek_cq(structib_cq *cq, intwc_cnt);

Return the number of available Work Completions in a CQ.

static inline int ib_req_notify_cq(struct ib_cq *cq, enum ib_cq_notify_flags flags);

Request a Completion notification event to be generated when the next Work Completion is added to the CQ.

static inline int ib_req_ncomp_notif(struct ib_cq *cq, int wc_cnt);

Request a Completion notification event to be generated when there is a specific number of Work Completions in a CQ.

static inline int ib_poll_cq(struct ib_cq *cq, int num_entries, struct ib_wc *wc);

Read and remove one or more Work Completions from a CQ. They are read in the order that they were added to the CQ.

struct ib_srq *ib_create_srq(struct ib_pd *pd, struct ib_srq_init_attr *srq_init_attr);

Create an SRQ that will be used as a shared Receive Queue for several QPs.

int ib_modify_srq(struct ib_srq *srq, struct ib_srq_attr *srq_attr, enum ib_srq_attr_mask srq_attr_mask);

Modify the attributes of an SRQ.

int ib_query_srq(struct ib_srq *srq, struct ib_srq_attr *srq_attr);

Query for the attributes of an SRQ. The SRQ limit value might be changed in subsequent calls to this method.

int ib_destroy_srq(struct ib_srq *srq);

Destroy an SRQ.

struct ib_qp *ib_create_qp(struct ib_pd *pd, struct ib_qp_init_attr *qp_init_attr);

Create a QP. Every new QP is assigned with a QP number that isn't in use by other QPs at the same time.

int ib_modify_qp(struct ib_qp *qp, struct ib_qp_attr *qp_attr, int qp_attr_mask);

Modify the attributes of a QP, which includes Send and Receive Queue attributes and the QP state.

int ib_query_qp(struct ib_qp *qp, struct ib_qp_attr *qp_attr, int qp_attr_mask, struct ib_qp_init_attr *qp_init_attr);

Query for the attributes of a QP. Some of the attributes might be changed in subsequent calls to this method.

int ib_destroy_qp(struct ib_qp *qp);

Destroy a QP.

static inline int ib_post_srq_recv(struct ib_srq *srq, struct ib_recv_wr *recv_wr, struct ib_recv_wr **bad_recv_wr);

Adds a linked list of Receive Requests to an SRQ.

static inline int ib_post_recv(struct ib_qp *qp, struct ib_recv_wr *recv_wr, struct ib_recv_wr **bad_recv_wr);

Adds a linked list of Receive Requests to the Receive Queue of a QP.

static inline int ib_post_send(struct ib_qp *qp, struct ib_send_wr *send_wr, struct ib_send_wr **bad_send_wr);

Adds a linked list of Send Requests to the Send Queue of a QP.

int ib_attach_mcast(struct ib_qp *qp, union ib_gid *gid, u16 lid);

Attaches a UD QP to a multicast group.

int ib_detach_mcast(struct ib_qp *qp, union ib_gid *gid, u16 lid);

Detaches a UD QP from a multicast group.

CHAPTER 14

Advanced Topics

Chapter 13 dealt with the InfiniBand subsystem and its implementation in Linux. This chapter deals with several advanced topics and some topics that didn't fit logically into other chapters. The chapter starts with a discussion about network namespaces, a type of lightweight process virtualization mechanism that was added to Linux in recent years. I will discuss the namespaces implementation in general and network namespaces in particular. You will learn that only two new system calls are needed in order to implement namespaces. You will also see several examples of how simple it is to create and manage network namespaces with the `ip` command of `iproute2`, and how simple it is to move one network device from one network namespace to another and to attach a specified process to a specified network namespace. The cgroups subsystem also provides resource management solution, which is different from namespaces. I will describe the cgroups subsystem and its two network modules, `net_prio` and `cls_cgroup`, and give two examples of using these cgroup network modules.

Later on in this chapter, you will learn about Busy Poll Sockets and how to tune them. The Busy Poll Sockets feature provides an interesting performance optimization technique for sockets that need low latency and are willing to pay a cost of higher CPU utilization. The Busy Poll Sockets feature is available from kernel 3.11. I will also cover the Bluetooth subsystem, the IEEE 802.15.4 subsystem and the Near Field Communication (NFC) subsystem; these three subsystems typically work in short range networks, and the development of new features for these subsystem is progressing at a rapid pace. I will also discuss Notification Chains, which is an important mechanism that you may encounter while developing or debugging kernel networking code and the PCI subsystem, as many network devices are PCI devices. I will not delve deep into the PCI subsystem details, as this book is not about device drivers. I will conclude the chapter with three short sections, one about the teaming network driver (which is the new kernel link aggregation solution), one about the Point-to-Point over Ethernet (PPPoE) Protocol, and finally one about Android.

Network Namespaces

This section covers Linux namespaces, what they are for and how they are implemented. It includes an in-depth discussion of network namespaces, giving some examples that will demonstrate their usage. Linux namespaces are essentially a virtualization solution. Operating system virtualization was implemented in mainframes many years before solutions like Xen or KVM hit the market. Also with Linux namespaces, which are a form of process virtualization, the idea is not new at all. It was tried in the Plan 9 operating system (see this article from 1992: "The Use of Name Spaces in Plan 9", `www.cs.bell-labs.com/sys/doc/names.html`).

Namespaces is a form of lightweight process virtualization, and it provides resource isolation. As opposed to virtualization solutions like KVM or Xen, with namespaces you do not create additional instances of the operating system on the same host, but use only a single operating system instance. I should mention in this context that the Solaris operating system has a virtualization solution named Solaris Zones, which also uses a single operating system instance, but the scheme of resource partitioning is somewhat different than that of Linux namespaces (for example, in Solaris Zones there is a global zone which is the primary zone, and which has more capabilities). In the FreeBSD operating system there is a mechanism called `jails`, which also provides resource partitioning without running more than one instance of the kernel.

The main idea of Linux namespaces is to partition resources among groups of processes to enable a process (or several processes) to have a different view of the system than processes in other groups of processes. This feature is used, for example, to provide resource isolation in the Linux containers project (http://lxc.sourceforge.net/). The Linux containers project also uses another resource management mechanism that is provided by the cgroups subsystem, which will be described later in this chapter. With containers, you can run different Linux distributions on the same host using one instance of the operating systems. Namespaces are also needed for the checkpoint/restore feature, which is used in high performance computing (HPC). For example, it is used in CRIU (http://criu.org/Main_Page), a software tool of OpenVZ (http://openvz.org/Main_Page), which implements checkpoint/restore functionality for Linux processes mostly in userspace, though there are very few places when CRIU kernel patches were merged. I should mention that there were some projects to implement checkpoint/restore in the kernel, but these projects were not accepted in mainline because they were too complex. For example, take the CKPT project: https://ckpt.wiki.kernel.org/index.php/Main_Page. The checkpoint/restore feature (sometimes referred to as checkpoint/restart) enables stopping and saving several processes on a filesystem, and at a later time restores those processes (possibly on a different host) from the filesystem and resumes its execution from where it was stopped. Without namespaces, checkpoint/restore has very limited use cases, in particular live migration is only possible with them. Another use case for network namespaces is when you need to set up an environment that needs to simulate different network stacks for testing, debugging, etc. For readers who want to learn more about checkpoint/restart, I suggest reading the article "Virtual Servers and Checkpoint/Restart in Mainstream Linux," by Sukadev Bhattiprolu, Eric W. Biederman, Serge Hallyn, and Daniel Lezcano.

Mount namespaces were the first type of Linux namespaces to be merged in 2002, for kernel 2.4.19. User namespaces were the last to be implemented, in kernel 3.8, for almost all filesystems types. It could be that additional namespaces will be developed, as is discussed later in this section. For creating a namespace you should have the CAP_SYS_ADMIN capability for all namespaces, except for the user namespace. Trying to create a namespace without the CAP_SYS_ADMIN capability for all namespaces, except for the user namespace, will result with an –EPRM error ("Operation not permitted"). Many developers took part in the development of namespaces, among them are Eric W. Biederman, Pavel Emelyanov, Al Viro, Cyrill Gorcunov, Andrew Vagin, and more.

After getting some background about process virtualization and Linux namespaces, and how they are used, you are now ready to dive in into the gory implementation details.

Namespaces Implementation

As of this writing, six namespaces are implemented in the Linux kernel. Here is a description of the main additions and changes that were needed in order to implement namespaces in the Linux kernel and to support namespaces in userspace packages:

- A structure called `nsproxy` (namespace proxy) was added. This structure contains pointers to five namespaces out of the six namespaces that are implemented. There is no pointer to the user namespace in the `nsproxy` structure; however, all the other five namespace objects contain a pointer to the user namespace object that owns them, and in each of these five namespaces, the user namespace pointer is called `user_ns`. The user namespace is a special case; it is a member of the credentials structure (`cred`), called `user_ns`. The `cred` structure represents the security context of a process. Each process descriptor (`task_struct`) contains two `cred` objects, for effective and objective process descriptor credentials. I will not delve into all the details and nuances of user namespaces implementation, since this is not in the scope of this book. An `nsproxy` object is created by the `create_nsproxy()` method and it is released by the `free_nsproxy()` method. A pointer to `nsproxy` object, which is also called `nsproxy`,

was added to the process descriptor (a process descriptor is represented by the `task_struct` structure, `include/linux/sched.h`.) Let's take a look at the `nsproxy` structure, as it's quite short and should be quite self-explanatory:

```
struct nsproxy {
      atomic_t count;
       struct uts_namespace *uts_ns;
       struct ipc_namespace *ipc_ns;
       struct mnt_namespace *mnt_ns;
       struct pid_namespace *pid_ns;
       struct net           *net_ns;
};
(include/linux/nsproxy.h)
```

- You can see in the `nsproxy` structure five pointers of namespaces (there is no user namespace pointer). Using the `nsproxy` object in the process descriptor (`task_struct` object) instead of five namespace objects is an optimization. When performing `fork()`, a new child is likely to live in the same set of namespaces as its parent. So instead of five reference counter increments (one per each namespace), only one reference counter increment would happen (of the `nsproxy` object). The `nsproxy count` member is a reference counter, which is initialized to 1 when the `nsproxy` object is created by the `create_nsproxy()` method, and which is decremented by the `put_nsproxy()` method and incremented by the `get_nsproxy()` method. Note that the `pid_ns` member of the `nsproxy` object was renamed to `pid_ns_for_children` in kernel 3.11.
- A new system call, `unshare()`, was added. This system call gets a single parameter that is a bitmask of CLONE* flags. When the flags argument consists of one or more namespace CLONE_NEW* flags, the `unshare()` system call performs the following steps:
  - First, it creates a new namespace (or several namespaces) according to the specified flag. This is done by calling the `unshare_nsproxy_namespaces()` method, which in turn creates a new `nsproxy` object and one or more namespaces by calling the `create_new_namespaces()` method. The type of the new namespace (or namespaces) is determined according to the specified CLONE_NEW* flag. The `create_new_namespaces()` method returns a new `nsproxy` object that contains the new created namespace (or namespaces).
  - Then it attaches the calling process to that newly created `nsproxy` object by calling the `switch_task_namespaces()` method.

  When CLONE_NEWPID is the flag of the `unshare()` system call, it works differently than with the other flags; it's an implicit argument to `fork()`; only the child task will happen in a new PID namespace, not the one calling the `unshare()` system call. Other CLONE_NEW* flags immediately put the calling process into a new namespace.

  The six CLONE_NEW* flags, which were added to support the creation of namespaces, are described later in this section. The implementation of the `unshare()` system call is in `kernel/fork.c`.

- A new system call, `setns()`, was added. It attaches the calling thread to an existing namespace. Its prototype is `int setns(int fd, int nstype)`; the parameters are:
  - `fd`: A file descriptor which refers to a namespace. These are obtained by opening links from the `/proc/<pid>/ns/` directory.
  - `nstype`: An optional parameter. When it is one of the new CLONE_NEW* namespaces flags, the specified file descriptor must refer to a namespace which matches the type of the specified CLONE_NEW* flag. When the `nstype` is not set (its value is 0) the `fd` argument can refer to a namespace of any type. If the `nstype` does not correspond to the namespace type associated with the specified `fd`, a value of –EINVAL is returned.

You can find the implementation of the `setns()` system call in `kernel/nsproxy.c`.

- The following six new clone flags were added in order to support namespaces:
  - CLONE_NEWNS (for mount namespaces)
  - CLONE_NEWUTS (for UTS namespaces)
  - CLONE_NEWIPC (for IPC namespaces)
  - CLONE_NEWPID (for PID namespaces)
  - CLONE_NEWNET (for network namespaces)
  - CLONE_NEWUSER (for user namespaces)

  The `clone()` system call is used traditionally to create a new process. It was adjusted to support these new flags so that it will create a new process attached to a new namespace (or namespaces). Note that you will encounter usage of the CLONE_NEWNET flag, for creating a new network namespace, in some of the examples later in this chapter.
- Each subsystem, from the six for which there is a namespace support, had implemented a unique namespace of its own. For example, the mount namespace is represented by a structure called `mnt_namespace`, and the network namespace is represented by a structure called `net`, which is discussed later in this section. I will mention the other namespaces later in this chapter.
- For namespaces creation, a method named `create_new_namespaces()` was added (`kernel/nsproxy.c`). This method gets as a first parameter a CLONE_NEW* flag or a bitmap of CLONE_NEW* flags. It first creates an `nsproxy` object by calling the `create_nsproxy()` method, and then it associates a namespace according to the specified flag; since the flag can be a bitmask of flags, the `create_new_namespaces()` method can associate more than one namespace. Let's take a look at the `create_new_namespaces()` method:

```
static struct nsproxy *create_new_namespaces(unsigned long flags,
        struct task_struct *tsk, struct user_namespace *user_ns,
        struct fs_struct *new_fs)
{
        struct nsproxy *new_nsp;
        int err;
```

Allocate an nsproxy object and initialize its reference counter to 1:

```
    new_nsp = create_nsproxy();
    if (!new_nsp)
            return ERR_PTR(-ENOMEM);
    . . .
```

After creating successfully an nsproxy object, we should create namespaces according to the specified flags, or associate an existing namespace to the new nsproxy object we created. We start by calling copy_mnt_ns(), for the mount namespaces, and then we call copy_utsname(), for the UTS namespace. I will describe here shortly the copy_utsname() method, because the UTS namespace is discussed in the "UTS Namespaces Implementation" section later in this chapter. If the CLONE_NEWUTS is not set in the specified flags of the copy_utsname() method, the copy_utsname() method does not create a new UTS namespace; it returns the UTS namespace that was passed by tsk->nsproxy->uts_ns as the last parameter to the copy_utsname() method. In case the CLONE_NEWUTS is set, the copy_utsname() method clones the specified UTS namespace by calling the clone_uts_ns() method. The clone_uts_ns() method, in turn, allocates a new UTS namespace object, copies the new_utsname object of the specified UTS namespace (tsk->nsproxy->uts_ns) into the new_utsname object of the newly created UTS namespace object, and returns the newly created UTS namespace. You will learn more about the new_utsname structure in the "UTS Namespaces Implementation" section later in this chapter:

```
     new_nsp->uts_ns = copy_utsname(flags, user_ns, tsk->nsproxy->uts_ns);
     if (IS_ERR(new_nsp->uts_ns)) {
             err = PTR_ERR(new_nsp->uts_ns);
             goto out_uts;
     }
    . . .
```

After handling the UTS namespace, we continue with calling the copy_ipcs() method to handle the IPC namespace, copy_pid_ns() to handle the PID namespace, and copy_net_ns() to handle the network namespace. Note that there is no call to the copy_user_ns() method, as the nsproxy does not contain a pointer to user namespace, as was mentioned earlier. I will describe here shortly the copy_net_ns() method. If the CLONE_NEWNET is not set in the specified flags of the create_new_namespaces() method, the copy_net_ns() method returns the network namespace that was passed as the third parameter to the copy_net_ns() method, tsk->nsproxy->net_ns, much like the copy_utsname() did, as you saw earlier in this section. If the CLONE_NEWNET is set, the copy_net_ns() method allocates a new network namespace by calling the net_alloc() method, initializes it by calling the setup_net() method, and adds it to the global list of all network namespaces, net_namespace_list:

```
    new_nsp->net_ns = copy_net_ns(flags, user_ns, tsk->nsproxy->net_ns);
    if (IS_ERR(new_nsp->net_ns)) {
            err = PTR_ERR(new_nsp->net_ns);
            goto out_net;
    }
    return new_nsp;
}
```

> Note that the setns() system call, which does not create a new namespace but only attaches the calling thread to a specified namespace, also calls create_new_namespaces(), but it passes 0 as a first parameter; this implies that only an nsproxy is created by calling the create_nsproxy() method, but no new namespace is created, but the calling thread is associated with an existing network namespace which is identified by the specified fd argument of the setns() system call. Later in the setns() system call implementation, the switch_task_namespaces() method is invoked, and it assigns the new nsproxy which was just created to the calling thread (see kernel/nsproxy.c).

- A method named `exit_task_namespaces()` was added in `kernel/nsproxy.c`. It is called when a process is terminated, by the `do_exit()` method (`kernel/exit.c`). The `exit_task_namespaces()` method gets the process descriptor (`task_struct` object) as a single parameter. In fact the only thing it does is call the `switch_task_namespaces()` method, passing the specified process descriptor and a NULL `nsproxy` object as arguments. The `switch_task_namespaces()` method, in turn, nullifies the `nsproxy` object of the process descriptor of the process which is being terminated. If there are no other processes that use that `nsproxy`, it is freed.

- A method named `get_net_ns_by_fd()` was added. This method gets a file descriptor as its single parameter, and returns the network namespace associated with the inode that corresponds to the specified file descriptor. For readers who are not familiar with filesystems and with inode semantics, I suggest reading the "Inode Objects" section of Chapter 12, "The Virtual Filesystem," in *Understanding the Linux Kernel* by Daniel P. Bovet and Marco Cesati (O'Reilly, 2005).

- A method named `get_net_ns_by_pid()` was added. This method gets a PID number as a single argument, and it returns the network namespace object to which this process is attached.

- Six entries were added under `/proc/<pid>/ns`, one for each namespace. These files, when opened, should be fed into the `setns()` system call. You can use `ls -al` or `readlink` to display the unique proc inode number which is associated with a namespace. This unique proc inode is created by the `proc_alloc_inum()` method when the namespace is created, and is freed by the `proc_free_inum()` method when the namespace is released. See, for example, in the `create_pid_namespace()` method in `kernel/pid_namespace.c`. In the following example, the number in square brackets on the right is the unique proc inode number of each namespace:

```
ls -al /proc/1/ns/
total 0
dr-x--x--x 2 root root 0 Nov  3 13:32 .
dr-xr-xr-x 8 root root 0 Nov  3 12:17 ..
lrwxrwxrwx 1 root root 0 Nov  3 13:32 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 Nov  3 13:32 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root root 0 Nov  3 13:32 net -> net:[4026531956]
lrwxrwxrwx 1 root root 0 Nov  3 13:32 pid -> pid:[4026531836]
lrwxrwxrwx 1 root root 0 Nov  3 13:32 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 Nov  3 13:32 uts -> uts:[4026531838]
```

- A namespace can stay alive if either one of the following conditions is met:
  - The namespace file under `/proc/<pid>/ns/` descriptor is held.
  - bind mounting the namespace proc file somewhere else, for example, for PID namespace, by: `mount --bind /proc/self/ns/pid /some/filesystem/path`

- For each of the six namespaces, a proc namespace operations object (an instance of `proc_ns_operations` structure) is defined. This object consists of callbacks, such as `inum`, to return the unique proc inode number associated with the namespace or `install`, for namespace installation (in the `install` callback, namespace specific actions are performed,

such as attaching the specific namespace object to the nsproxy object, and more; the install callback is invoked by the setns system call). The proc_ns_operations structure in defined in include/linux/proc_fs.h. Following is the list of the six proc_ns_operations objects:

  - utsns_operations for UTS namespace (kernel/utsname.c)
  - ipcns_operations for IPC namespace (ipc/namespace.c)
  - mntns_operations for mount namespaces (fs/namespace.c)
  - pidns_operations for PID namespaces (kernel/pid_namespace.c)
  - userns_operations for user namespace (kernel/user_namespace.c)
  - netns_operations for network namespace (net/core/net_namespace.c)

- For each namespace, except the mount namespace, there is an **initial namespace**:
  - init_uts_ns: For UTS namespace (init/version.c).
  - init_ipc_ns: For IPC namespace (ipc/msgutil.c).
  - init_pid_ns: For PID namespace (kernel/pid.c).
  - init_net: For network namespace (net/core/net_namespace.c).
  - init_user_ns: For user namespace (kernel/user.c).
- An initial, default nsproxy object is defined: it is called init_nsproxy and it contains pointers to five initial namespaces; they are all initialized to be the corresponding specific initial namespace except for the mount namespace, which is initialized to be NULL:

```
struct nsproxy init_nsproxy = {
        .count  = ATOMIC_INIT(1),
        .uts_ns = &init_uts_ns,
#if defined(CONFIG_POSIX_MQUEUE) || defined(CONFIG_SYSVIPC)
        .ipc_ns = &init_ipc_ns,
#endif
        .mnt_ns = NULL,
        .pid_ns = &init_pid_ns,
#ifdef CONFIG_NET
        .net_ns = &init_net,
#endif
};
(kernel/nsproxy.c)
```

- A method named task_nsproxy() was added; it gets as a single parameter a process descriptor (task_struct object), and it returns the nsproxy associated with the specified task_struct object. See include/linux/nsproxy.h.

These are the six namespaces available in the Linux kernel as of this writing:

- **Mount namespaces:** The mount namespaces allows a process to see its own view of the filesystem and of its mount points. Mounting a filesystem in one mount namespace does not propagate to the other mount namespaces. Mount namespaces are created by setting the CLONE_NEWNS flag when calling the clone() or unshare() system calls. In order to implement mount namespaces, a structure called mnt_namespace was added (fs/mount.h),

and nsproxy holds a pointer to an mnt_namespace object called mnt_ns. Mount namespaces are available from kernel 2.4.19. Mount namespaces are implemented primarily in fs/namespace.c. When creating a new mount namespace, the following rules apply:

- All previous mounts will be visible in the new mount namespace.
- Mounts/unmounts in the new mount namespace are invisible to the rest of the system.
- Mounts/unmounts in the global mount namespace are visible in the new mount namespace.

Mount namespaces use a VFS enhancement called *shared subtrees*, which was introduced in the Linux 2.6.15 kernel; the shared subtrees feature introduced new flags: MS_PRIVATE, MS_SHARED, MS_SLAVE and MS_UNBINDABLE . (See http://lwn.net/Articles/159077/ and Documentation/filesystems/sharedsubtree.txt.) I will not discuss the internals of mount namespaces implementation. For readers who want to learn more about mount namespaces usage, I suggest reading the following article: "Applying Mount Namespaces," by Serge E. Hallyn and Ram Pai (http://www.ibm.com/developerworks/linux/library/l-mount-namespaces/index.html).

- **PID namespaces:** The PID namespaces provides the ability for different processes in different PID namespaces to have the same PID. This feature is a building block for Linux containers. It is important for checkpoint/restore of a process, because a process checkpointed on one host can be restored on a different host even if there is a process with the same PID on that host. When creating the first process in a new PID namespace, its PID is 1. The behavior of this process is somewhat like the behavior of the init process. This means that when a process dies, all its orphaned children will now have the process with PID 1 as their parent (child reaping). Sending SIGKILL signal to a process with PID 1 does not kill the process, regardless of in which namespace the SIGKILL signal was sent, in the initial PID namespace or in any other PID namespace. But killing init of one PID namespace from another (parent one) will work. In this case, all of the tasks living in the former namespace will be killed and the PID namespace will be stopped. PID namespaces are created by setting the CLONE_NEWPID flag when calling the clone() or unshare() system calls. In order to implement PID namespaces, a structure called pid_namespace was added (include/linux/pid_namespace.h), and nsproxy holds a pointer to a pid_namespace object called pid_ns. In order to have PID namespaces support, CONFIG_PID_NS should be set. PID namespaces are available from kernel 2.6.24. PID namespaces are implemented primarily in kernel/pid_namespace.c.
- **Network namespaces:** The network namespace allows creating what appears to be multiple instances of the kernel network stack. Network namespaces are created by setting the CLONE_NEWNET flag when calling the clone() or unshare() system calls. In order to implement network namespaces, a structure called net was added (include/net/net_namespace.h), and nsproxy holds a pointer to a net object called net_ns. In order to have network namespaces support, CONFIG_NET_NS should be set. I will discuss network namespaces later in this section. Network namespaces are available from kernel 2.6.29. Network namespaces are implemented primarily in net/core/net_namespace.c.
- **IPC namespaces:** The IPC namespace allows a process to have its own System V IPC resources and POSIX message queues resources. IPC namespaces are created by setting the CLONE_NEWIPC flag when calling the clone() or unshare() system calls. In order to implement IPC namespaces, a structure called ipc_namespace was added (include/linux/ipc_namespace.h), and nsproxy holds a pointer to an ipc_namespace object called ipc_ns.

In order to have IPC namespaces support, CONFIG_IPC_NS should be set. Support for System V IPC resources is available in IPC namespaces from kernel 2.6.19. Support for POSIX message queues resources in IPC namespaces was added later, in kernel 2.6.30. IPC namespaces are implemented primarily in `ipc/namespace.c`.

- **UTS namespaces:** The UTS namespace provides the ability for different UTS namespaces to have different host name or domain name (or other information returned by the `uname()` system call). UTS namespaces are created by setting the CLONE_NEWUTS flag when calling the `clone()` or `unshare()` system calls. UTS namespace implementation is the simplest among the six namespaces that were implemented. In order to implement the UTS namespace, a structure called `uts_namespace` was added (`include/linux/utsname.h`), and `nsproxy` holds a pointer to a `uts_namespace` object called `uts_ns`. In order to have UTS namespaces support, CONFIG_UTS_NS should be set. UTS namespaces are available from kernel 2.6.19. UTS namespaces are implemented primarily in `kernel/utsname.c`.

- **User namespaces:** The user namespace allows mapping of user and group IDs. This mapping is done by writing to two `procfs` entries that were added for supporting user namespaces: `/proc/sys/kernel/overflowuid` and `/proc/sys/kernel/overflowgid`. A process attached to a user namespace can have a different set of capabilities then the host. User namespaces are created by setting the CLONE_NEWUSER flag when calling the `clone()` or `unshare()` system calls. In order to implement user namespaces, a structure called `user_namespace` was added (`include/linux/user_namespace.h`). The `user_namespace` object contains a pointer to the user namespace object that created it (`parent`). As opposed to the other five namespaces, `nsproxy` does not hold a pointer to a `user_namespace` object. I will not delve into more implementation details of user namespaces, as it is probably the most complex namespace and as it is beyond the scope of the book. In order to have user namespaces support, CONFIG_USER_NS should be set. User namespaces are available from kernel 3.8 for almost all filesystem types. User namespaces are implemented primarily in `kernel/user_namespace.c`.

Support to namespaces was added in four userspace packages:

- In `util-linux`:
  - The `unshare` utility can create any of the six namespaces, available since version 2.17.
  - The `nsenter` utility (which is in fact a light wrapper around the `setns` system call), available since version 2.23.
- In `iproute2`, management of network namespaces is done with the `ip netns` command, and you will see several examples for this later in this chapter. Moreover, you can move a network interface to a different network namespace with the `ip link` command as you will see in the "Moving a Network Interface to a different Network Namespace" section later in this chapter.
- In `ethtool`, support was added to enable to find out whether the NETIF_F_NETNS_LOCAL feature is set for a specified network interface. When the NETIF_F_NETNS_LOCAL feature is set, this indicates that the network interface is local to that network namespace, and you cannot move it to a different network namespace. The NETIF_F_NETNS_LOCAL feature will be discussed later in this section.
- In the wireless `iw` package, an option was added to enable moving a wireless interface to a different namespace.

■ **Note** In a presentation in Ottawa Linux Symposium (OLS) in 2006, "Multiple Instances of the Global Linux Namespaces," Eric W. Biederman (one of the main developers of Linux namespaces) mentioned ten namespaces; the other four namespaces that he mentioned in this presentation and that are not implemented yet are: device namespace, security namespace, security keys namespace, and time namespace. (See https://www.kernel.org/doc/ols/2006/ols2006v1-pages-101-112.pdf.) For more information about namespaces, I suggest reading a series of six articles about it by Michael Kerrisk (https://lwn.net/Articles/531114/). Mobile OS virtualization projects triggered a development effort to support device namespaces; for more information about device namespaces, which are not yet part of the kernel, see "Device Namespaces" By Jake Edge (http://lwn.net/Articles/564854/) and also (http://lwn.net/Articles/564977/). There was also some work for implementing a new syslog namespace (see the article "Stepping Closer to Practical Containers: "syslog" namespaces", http://lwn.net/Articles/527342/).

The following three system calls can be used with namespaces:

- `clone()`: Creates a new process attached to a new namespace (or namespaces). The type of the namespace is specified by a CLONE_NEW* flag which is passed as a parameter. Note that you can also use a bitmask of these CLONE_NEW* flags. The implementation of the `clone()` system call is in `kernel/fork.c`.
- `unshare()`: Discussed earlier in this section.
- `setns()`: Discussed earlier in this section.

■ **Note** Namespaces do not have names inside the kernel that userspace processes can use to talk with them. If namespaces would have names, this would require keeping them globally, in yet another special namespace. This would complicate the implementation and can raise problems in checkpoint/restore for example. Instead, userspace processes should open namespace files under `/proc/<pid>/ns/` and their file descriptors can be used to talk to a specific namespace, in order to keep that namespace alive. Namespaces are identified by a unique proc inode number generated when they are created and freed when they are released. Each of the six namespace structures contains an integer member called `proc_inum`, which is the namespace unique proc inode number and is assigned by calling the `proc_alloc_inum()` method. Each of the six namespaces has also a `proc_ns_operations` object, which includes namespace-specific callbacks; one of these callbacks, called `inum`, returns the `proc_inum` of the associated namespace (for the definition of `proc_ns_operations` structure, refer to `include/linux/proc_fs.h`).

Before discussing network namespaces, let's describe how the simplest namespace, the UTS namespace, is implemented. This is a good starting point to understand the other, more complex namespaces.

UTS Namespaces Implementation

In order to implement UTS namespaces, a struct called `uts_namespace` was added:

```
struct uts_namespace {
        struct kref kref;
        struct new_utsname name;
```

```
        struct user_namespace *user_ns;
        unsigned int proc_inum;
};
(include/linux/utsname.h)
```

Here is a short description of the members of the `uts_namespace` structure:

- `kref`: A reference counter. It is a generic kernel reference counter, incremented by the `kref_get()` method and decremented by the `kref_put()` method. Besides the UTS namespace, also the PID namespace has a `kref` object as a reference counter; all the other four namespaces use an atomic counter for reference counting. For more info about the `kref` API look in `Documentation/kref.txt`.
- `name`: A `new_utsname` object, contains fields like `domainname` and `nodename` (will be discussed shortly).
- `user_ns`: The user namespace associated with the UTS namespace.
- `proc_inum`: The unique proc inode number of the UTS namespace.

The `nsproxy` structure contains a pointer to the `uts_namespace`:

```
struct nsproxy {
        . . .
        struct uts_namespace *uts_ns;
        . . .
};
(include/linux/nsproxy.h)
```

As you saw earlier, the `uts_namespace` object contains an instance of the `new_utsname` structure. Let's take a look at the `new_utsname` structure, which is the essence of the UTS namespace:

```
struct new_utsname {
        char sysname[__NEW_UTS_LEN + 1];
        char nodename[__NEW_UTS_LEN + 1];
        char release[__NEW_UTS_LEN + 1];
        char version[__NEW_UTS_LEN + 1];
        char machine[__NEW_UTS_LEN + 1];
        char domainname[__NEW_UTS_LEN + 1];
};
(include/uapi/linux/utsname.h)
```

The `nodename` member of the `new_utsname` is the host name, and `domainname` is the domain name. A method named `utsname()` was added; this method simply returns the `new_utsname` object which is associated with the process that currently runs (`current`):

```
static inline struct new_utsname *utsname(void)
{
         return &current->nsproxy->uts_ns->name;
}
(include/linux/utsname.h)
```

Now, the new gethostname() system call implementation is the following:

```
SYSCALL_DEFINE2(gethostname, char __user *, name, int, len)
{
        int i, errno;
        struct new_utsname *u;

        if (len < 0)
                return -EINVAL;
        down_read(&uts_sem);
```

Invoke the utsname() method, which accesses the new_utsname object of the UTS namespace associated with the current process:

```
        u = utsname();
        i = 1 + strlen(u->nodename);
        if (i > len)
                i = len;
        errno = 0;
```

Copy to userspace the nodename of the new_utsname object that the utsname() method returned:

```
        if (copy_to_user(name, u->nodename, i))
                errno = -EFAULT;
        up_read(&uts_sem);
        return errno;
}
(kernel/sys.c)
```

You can find a similar approach in the sethostbyname() and in the uname() system calls, which are also defined in kernel/sys.c. I should note that UTS namespaces implementation also handles UTS procfs entries. There are only two UTS procfs entries, /proc/sys/kernel/domainname and /proc/sys/kernel/hostname, which are writable (this means that you can change them from userspace). There are other UTS procfs entries which are not writable, like /proc/sys/kernel/ostype and /proc/sys/kernel/osrelease. If you will look at the table of the UTS procfs entries, uts_kern_table (kernel/utsname_sysctl.c), you will see that some entries, like ostype and osrelease, have mode of "0444", which means they are not writable, and only two of them, hostname and domainname, have mode of "0644", which means they are writable. Reading and writing the UTS procfs entries is handled by the proc_do_uts_string() method. Readers who want to learn more about how UTS procfs entries are handled should look into the proc_do_uts_string() method and into the get_uts() method; both are in kernel/utsname_sysctl.c.

Now that you learned about how the simplest namespace, the UTS namespace, is implemented, it is time to learn about network namespaces and their implementation.

Network Namespaces Implementation

A network namespace is logically another copy of the network stack, with its own network devices, routing tables, neighbouring tables, netfilter tables, network sockets, network procfs entries, network sysfs entries, and other network resources. A practical feature of network namespaces is that network applications running in a given namespace (let's say ns1) will first look for configuration files under /etc/netns/ns1, and only afterward under /etc. So, for example, if you created a namespace called ns1 and you have created /etc/netns/ns1/hosts, every userspace application that tries to access the hosts file will first access /etc/netns/ns1/hosts and only then (if the entry being looked for does not exist) will it read /etc/hosts. This feature is implemented using bind mounts and is available only for network namespaces created with the ip netns add command.

The Network Namespace Object (struct net)

Let's turn now to the definition of the net structure, which is the fundamental data structure that represents a network namespace:

```
struct net {
        . . .
        struct user_namespace   *user_ns;       /* Owning user namespace */
        unsigned int            proc_inum;
        struct proc_dir_entry   *proc_net;
        struct proc_dir_entry   *proc_net_stat;
        . . .
        struct list_head        dev_base_head;
        struct hlist_head       *dev_name_head;
        struct hlist_head       *dev_index_head;
        . . .
        int                     ifindex;
        . . .
        struct net_device       *loopback_dev;  /* The loopback */
        . . .
        atomic_t                count;          /* To decided when the network
                                                *  namespace should be shut down.
                                                */

        struct netns_ipv4       ipv4;
#if IS_ENABLED(CONFIG_IPV6)
        struct netns_ipv6       ipv6;
#endif
#if defined(CONFIG_IP_SCTP) || defined(CONFIG_IP_SCTP_MODULE)
        struct netns_sctp       sctp;
#endif
       . . .

#if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
        struct netns_ct         ct;
#endif
#if IS_ENABLED(CONFIG_NF_DEFRAG_IPV6)
        struct netns_nf_frag    nf_frag;
#endif
        . . .
        struct net_generic __rcu  *gen;
#ifdef CONFIG_XFRM
        struct netns_xfrm       xfrm;
#endif
        . . .
};
(include/net/net_namespace.h)
```

Here is a short description of several members of the net structure:

- user_ns represents the user namespace that created the network namespace; it owns the network namespace and all its resources. It is assigned in the setup_net() method. For the initial network namespace object (init_net), the user namespace that created it is the initial user namespace, init_user_ns.
- proc_inum is the unique proc inode number associated to the network namespace. This unique proc inode is created by the proc_alloc_inum() method, which also assigns proc_inum to be the proc inode number. The proc_alloc_inum() method is invoked by the network namespace initialization method, net_ns_net_init(), and it is freed by calling the proc_free_inum() method in the network namespace cleanup method, net_ns_net_exit().
- proc_net represents the network namespace procfs entry (/proc/net) as each network namespace maintains its own procfs entry.
- proc_net_stat represents the network namespace procfs statistics entry (/proc/net/stat) as each network namespace maintains its own procfs statistics entry.
- dev_base_head points to a linked list of all network devices.
- dev_name_head points to a hashtable of network devices, where the key is the network device name.
- dev_index_head points to a hashtable of network devices, where the key is the network device index.
- ifindex is the last device index assigned inside a network namespace. Indices are virtualized in network namespaces; this means that loopback devices would always have index of 1 in all network namespaces, and other network devices may have coinciding indices when living in different network namespaces.
- loopback_dev is the loopback device. Every new network namespace is created with only one network device, the loopback device. The loopback_dev object of a network namespace is assigned in the loopback_net_init() method, drivers/net/loopback.c. You cannot move the loopback device from one network namespace to another.
- count is the network namespace reference counter. It is initialized to 1 when the network namespace is created by the by the setup_net() method. It is incremented by the get_net() method and decremented by the put_net() method. If the count reference counter reaches 0 in the put_net() method, the __put_net() method is called. The __put_net() method, in turn, adds the network namespace to a global list of network namespaces to be removed, cleanup_list, and later removes it.
- ipv4 (an instance of the netns_ipv4 structure) for the IPv4 subsystem. The netns_ipv4 structure contains IPv4 specific fields which are different for different namespaces. For example, in chapter 6 you saw that the multicast routing table of a specified network namespace called net is stored in net->ipv4.mrt. I will discuss the netns_ipv4 later in this section.
- ipv6 (an instance of the netns_ipv6 structure) for the IPv6 subsystem.
- sctp (an instance of the netns_sctp structure) for SCTP sockets.
- ct (an instance of the netns_ct structure, which is discussed in chapter 9) for the netfilter connection tracking subsystem.

- gen (an instance of the net_generic structure, defined in include/net/netns/generic.h) is a set of generic pointers on structures describing a network namespace context of optional subsystems. For example, the sit module (Simple Internet Transition, an IPv6 tunnel, implemented in net/ipv6/sit.c) puts its private data on struct net using this engine. This was introduced in order not to flood the struct net with pointers for every single network subsystem that is willing to have per network namespace context.
- xfrm (an instance of the netns_xfrm structure, which is mentioned several times in chapter 10) for the IPsec subsystem.

Let's take a look at the IPv4 specific namespace, the netns_ipv4 structure:

```
struct netns_ipv4 {
    . . .
#ifdef CONFIG_IP_MULTIPLE_TABLES
        struct fib_rules_ops    *rules_ops;
        bool                    fib_has_custom_rules;
        struct fib_table        *fib_local;
        struct fib_table        *fib_main;
        struct fib_table        *fib_default;
#endif
   . . .
        struct hlist_head       *fib_table_hash;
        struct sock             *fibnl;

        struct sock             **icmp_sk;
   . . .
#ifdef CONFIG_NETFILTER
        struct xt_table         *iptable_filter;
        struct xt_table         *iptable_mangle;
        struct xt_table         *iptable_raw;
        struct xt_table         *arptable_filter;
#ifdef CONFIG_SECURITY
        struct xt_table         *iptable_security;
#endif
        struct xt_table         *nat_table;
#endif

        int sysctl_icmp_echo_ignore_all;
        int sysctl_icmp_echo_ignore_broadcasts;
        int sysctl_icmp_ignore_bogus_error_responses;
        int sysctl_icmp_ratelimit;
        int sysctl_icmp_ratemask;
        int sysctl_icmp_errors_use_inbound_ifaddr;

        int sysctl_tcp_ecn;

        kgid_t sysctl_ping_group_range[2];
        long sysctl_tcp_mem[3];

        atomic_t dev_addr_genid;
```

```
#ifdef CONFIG_IP_MROUTE
#ifndef CONFIG_IP_MROUTE_MULTIPLE_TABLES
        struct mr_table         *mrt;
#else
        struct list_head        mr_tables;
        struct fib_rules_ops    *mr_rules_ops;
#endif
#endif
};
(net/netns/ipv4.h)
```

You can see in the `netns_ipv4` structure many IPv4-specific tables and variables, like the routing tables, the netfilter tables, the multicast routing tables, and more.

Network Namespaces Implementation: Other Data Structures

In order to support network namespaces, a member called `nd_net`, which is a pointer to a network namespace, was added to the network device object (`struct net_device`). Setting the network namespace for a network device is done by calling the `dev_net_set()` method, and getting the network namespace associated to a network device is done by calling the `dev_net()` method. Note that a network device can belong to only a single network namespace at a given moment. The `nd_net` is set typically when a network device is registered or when a network device is moved to a different network namespace. For example, when registering a VLAN device, both these methods just mentioned are used:

```
static int register_vlan_device(struct net_device *real_dev, u16 vlan_id)
{
    struct net_device *new_dev;
```

The network namespace to be assigned to the new VLAN device is the network namespace associated with the real device, which is passed as a parameter to the `register_vlan_device()` method; we get this namespace by calling `dev_net(real_dev)`:

```
    struct net *net = dev_net(real_dev);
    . . .
    new_dev = alloc_netdev(sizeof(struct vlan_dev_priv), name, vlan_setup);

    if (new_dev == NULL)
        return -ENOBUFS;
```

Switch the network namespace by calling the `dev_net_set()` method:

```
    dev_net_set(new_dev, net);

    . . .
}
```

A member called `sk_net`, a pointer to a network namespace, was added to `struct sock`, which represents a socket. Setting the network namespace for a `sock` object is done by calling the `sock_net_set()` method, and getting the network namespace associated to a `sock` object is done by calling the `sock_net()` method. Like in the case of the `nd_net` object, also a `sock` object can belong to only a single network namespace at a given moment.

When the system boots, a default network namespace, `init_net`, is created. After the boot, all physical network devices and all sockets belong to that initial namespace, as well as the network loopback device.

Some network devices and some network subsystems should have network namespaces specific data. In order to enable this, a structure named `pernet_operations` was added; this structure includes an `init` and `exit` callbacks:

```
struct pernet_operations {
        . . .
        int (*init)(struct net *net);
        void (*exit)(struct net *net);
        . . .
        int *id;
        size_t size;
};
(include/net/net_namespace.h)
```

Network devices that need network namespaces specific data should define a `pernet_operations` object, and define its `init()` and `exit()` callbacks for device specific initialization and cleanup, respectively, and call the `register_pernet_device()` method in their module initialization and the `unregister_pernet_device()` method when the module is removed, passing the `pernet_operations` object as a single parameter in both cases. For example, the PPPoE module exports information about PPPoE session by a `procfs` entry, `/proc/net/pppoe`. The information exported by this `procfs` entry depends on the network namespace to which this PPPoE device belongs (since different PPPoE devices can belong to different network namespaces). So the PPPoE module defines a `pernet_operations` object called `pppoe_net_ops`:

```
static struct pernet_operations pppoe_net_ops = {
        .init = pppoe_init_net,
        .exit = pppoe_exit_net,
        .id   = &pppoe_net_id,
        .size = sizeof(struct pppoe_net),
}
(net/ppp/pppoe.c)
```

In the `init` callback, `pppoe_init_net()`, it only creates the PPPoE `procfs` entry, `/proc/net/pppoe`, by calling the `proc_create()` method:

```
static __net_init int pppoe_init_net(struct net *net)
{
        struct pppoe_net *pn = pppoe_pernet(net);
        struct proc_dir_entry *pde;

        rwlock_init(&pn->hash_lock);

        pde = proc_create("pppoe", S_IRUGO, net->proc_net, &pppoe_seq_fops);
#ifdef CONFIG_PROC_FS
        if (!pde)
                return -ENOMEM;
#endif

        return 0;
}
(net/ppp/pppoe.c)
```

And in the exit callback, pppoe_exit_net(), it only removes the PPPoE procfs entry, /proc/net/pppoe, by calling the remove_proc_entry() method:

```
static __net_exit void pppoe_exit_net(struct net *net)
{
        remove_proc_entry("pppoe", net->proc_net);
}
(net/ppp/pppoe.c)
```

Network subsystems that need network-namespace-specific data should call register_pernet_subsys() when the subsystem is initialized and unregister_pernet_subsys() when the subsystem is removed. You can look for examples in net/ipv4/route.c, and there are many other examples of reviewing these methods. The network namespace module itself also defines a net_ns_ops object and registers it in the boot phase:

```
static struct pernet_operations __net_initdata net_ns_ops = {
        .init = net_ns_net_init,
        .exit = net_ns_net_exit,
};

static int __init net_ns_init(void)
{
    . . .
    register_pernet_subsys(&net_ns_ops);
    . . .
}
(net/core/net_namespace.c)
```

Each time a new network namespace is created, the init callback (net_ns_net_init) is called, and each time a network namespace is removed, the exit callback (net_ns_net_exit) is called. The only thing that the net_ns_net_init() does is to allocate a unique proc inode for the newly created namespace by calling the proc_alloc_inum() method; the newly created unique proc inode number is assigned to net->proc_inum:

```
static __net_init int net_ns_net_init(struct net *net)
{
        return proc_alloc_inum(&net->proc_inum);
}
```

And the only thing that the net_ns_net_exit() method does is to remove that unique proc inode by calling the proc_free_inum() method:

```
static __net_exit void net_ns_net_exit(struct net *net)
{
        proc_free_inum(net->proc_inum);
}
```

When you create a new network namespace, it has only the network loopback device. The most common ways to create a network namespace are:

- By a userspace application which will create a network namespace with the clone() system call or with the unshare() system call, setting the CLONE_NEWNET flag in both cases.
- Using ip netns command of iproute2 (you will shortly see an example).
- Using the unshare utility of util-linux, with the --net flag.

Network Namespaces Management

Next you will see some examples of using the `ip netns` command of the `iproute2` package to perform actions such as creating a network namespace, deleting a network namespace, showing all the network namespaces, and more.

- Creating a network namespace named `ns1` is done by:

  ```
  ip netns add ns1
  ```

  Running this command triggers first the creation of a file called `/var/run/netns/ns1`, and then the creation of the network namespace by the `unshare()` system call, passing it a CLONE_NEWNET flag. Then `/var/run/netns/ns1` is attached to the network namespace (`/proc/self/ns/net`) by a `bind` mount (calling the `mount()` system call with MS_BIND). Note that network namespaces can be nested, which means that from within `ns1` you can also create a new network namespace, and so on.

- Deleting a network namespace named `ns1` is done by:

  ```
  ip netns del ns1
  ```

  Note that this will not delete a network namespace if there is one or more processes attached to it. In case there are no such processes, the `/var/run/netns/ns1` file is deleted. Note also that when deleting a namespace, all its network devices are moved to the initial, default network namespace, `init_net`, except for network namespace local devices, which are network devices whose NETIF_F_NETNS_LOCAL feature is set; such network devices are deleted. See more in the "Moving a Network Interface to a Network Namespace" section later in this chapter and in Appendix A.

- Showing all the network namespaces in the system that were added by `ip netns add` is done by:

  ```
  ip netns list
  ```

  In fact, running `ip netns list` simply shows the names of files under `/var/run/netns`. Note that network namespaces not added by `ip netns add` will not be displayed by `ip netns list`, because creating such network namespaces did not trigger creation of any file under `/var/run/netns`. So, for example, a network namespace created by `unshare --net bash` will not appear when running `ip netns list`.

- Monitoring creation and removal of a network namespace is done by:

  ```
  ip netns monitor
  ```

  After running `ip netns monitor`, when you add a new namespace by `ip netns add ns2` you will see on screen the following message: "add ns2", and after you delete that namespace by `ip netns delete ns2` you will see on screen the following message: "delete ns2". Note that adding and removing network namespaces not by running `ip netns add` and `ip netns delete`, respectively, does not trigger displaying any messages on screen by `ip netns monitor`. The `ip netns monitor` command is implemented by setting an `inotify` watch on `/var/run/netns`. Note that in case you will run `ip netns monitor` before adding at least one network namespace with `ip netns add` you will get the following error: `inotify_add_watch failed: No such file or directory`. The reason is that trying to set a watch on `/var/run/netns`, which does not exist yet, fails. See `man inotify_init()` and `man inotify_add_watch()`.

- Start a shell in a specified namespace (ns1 in this example) is done by:

  ```
  ip netns exec ns1 bash
  ```

  Note that with ip netns exec you can run **any** command in a specified network namespace. For example, the following command will display all network interfaces in the network namespace called ns1:

  ```
  ip netns exec ns1 ifconfig -a
  ```

In recent versions of iproute2 (since version 3.8), you have these two additional helpful commands:

- Show the network namespace associated with the specified pid:

  ```
  ip netns identify #pid
  ```

  This is implemented by reading /proc/<pid>/ns/net and iterating over the files under /var/run/netns to find a match (using the stat() system call).

- Show the PID of a process (or list of processes) attached to a network namespace called ns1 by:

  ```
  ip netns pids ns1
  ```

  This is implemented by reading /var/run/netns/ns1, and then iterating over /proc/<pid> entries to find a matching /proc/pid/ns/net entry (using the stat() system call).

■ **Note** For more information about the various ip netns command options see man ip netns.

Moving a Network Interface to a Different Network Namespace

Moving a network interface to a network namespace named ns1 can be done with the ip command. For example, by: ip link set eth0 netns ns1. As part of implementing network namespaces, a new feature named NETIF_F_NETNS_LOCAL was added to the features of the net_device object (The net_device structure represents a network interface. For more information about the net_device structure and its features see Appendix A). You can find out whether the NETIF_F_NETNS_LOCAL feature is set for a specified network device by looking at the netns-local flag in the output of ethtool -k eth0 or in the output of ethtool --show-features eth0 (both commands are equivalent.) Note that you cannot set the NETIF_F_NETNS_LOCAL feature with ethtool. This feature, when set, denotes that the network device is a network namespace local device. For example, the loopback, the bridge, the VXLAN and the PPP devices are network namespace local devices. Trying to move a network device whose NETIF_F_NETNS_LOCAL feature is set to a different namespace will fail with an error of –EINVAL, as you will shortly see in the following code snippet. The dev_change_net_namespace() method is invoked when trying to move a network interface to a different network namespace, for example by: ip link set eth0 netns ns1. Let's take a look at the dev_change_net_namespace() method:

```
int dev_change_net_namespace(struct net_device *dev, struct net *net, const char *pat)
{
        int err;
```

```
        ASSERT_RTNL();

        /* Don't allow namespace local devices to be moved. */
        err = -EINVAL;
```

Return –EINVAL in case that the device is a local device (The NETIF_F_NETNS_LOCAL flag in the features of net_device object is set)

```
        if (dev->features & NETIF_F_NETNS_LOCAL)
                goto out;
        . . .
```

Actually switch the network namespace by setting nd_net of the net_device object to the new specified namespace:

```
        dev_net_set(dev, net)
        . . .

out:
        return err;
}
(net/core/dev.c)
```

■ **Note** You can move a network interface to a network namespace named ns1 also by specifying a PID of a process that is attached to that namespace, without specifying the namespace name explicitly. For example, if you know that a process whose PID is <pidNumber> is attached to ns1, running ip link set eth1 netns <pidNumber> will move eth1 to the ns1 namespace. Implementation details: getting the network namespace object when specifying one of the PIDs of its attached processes is implemented by the get_net_ns_by_pid() method, whereas getting the network namespace object when specifying the network namespace name is implemented by the get_net_ns_by_fd() method; both methods are in net/core/net_namespace.c. In order to move a wireless network interface to a different network namespace you should use the iw command. For example, if you want to move wlan0 to a network namespace and you know that a process whose PID is <pidNumber> is attached to that namespace, you can run iw phy phy0 set netns <pidNumber> to move it to that network namespace. For the implementation details, refer to the nl80211_wiphy_netns() method in net/wireless/nl80211.c.

Communicating Between Two Network Namespaces

I will end the network namespaces section with a short example of how two network namespaces can communicate with each other. It can be done either by using Unix sockets or by using the Virtual Ethernet (VETH) network driver to create a pair of virtual network devices and moving one of them to another network namespace. For example, here are the first two namespaces, ns1 and ns2:

```
ip netns add ns1
ip netns add ns2
```

Start a shell in ns1:

```
ip netns exec ns1 bash
```

Create a virtual Ethernet device (its type is veth):

```
ip link add name if_one type veth peer name if_one_peer
```

Move if_one_peer to ns2:

```
ip link set dev if_one_peer netns ns2
```

You can now set addresses on if_one and on if_one_peer as usual, with the ifconfig command or with the ip command, and send packets from one network namespace to the other.

■ **Note** Network namespaces are not mandatory for a kernel image. By default, network namespaces are enabled (CONFIG_NET_NS is set) in most distributions. However, you can build and boot a kernel where network namespaces are disabled.

I have discussed in this section what namespaces are, and in particular what are network namespaces. I mentioned some of the major changes that were required in order to implement namespaces in general, like adding 6 new CLONE_NEW* flags, adding two new systems calls, adding an nsproxy object to the process descriptor, and more. I also described the implementation of UTS namespaces, which are the most simple among all namespaces, and the implementation of network namespaces. Several examples were given showing how simple it is to manipulate network namespaces with the ip netns command of the iproute2 package. Next I will describe the cgroups subsystem, which provides another solution of resource management, and two network modules that belong to it.

Cgroups

The cgroups subsystem is a project started by Paul Menage, Rohit Seth, and other Google developers in 2006. It was initially called "process containers," but later it was renamed to "Control Groups." It provides resource management and resource accounting for groups of processes. It has been part of the mainline kernel since kernel 2.6.24, and it's used in several projects: for example by systemd (a service manager which replaced SysV init scripts; used, for example, by Fedora and by openSUSE), by the Linux Containers project, which was mentioned earlier in this chapter, by Google containers (https://github.com/google/lmctfy/), by libvirt (http://libvirt.org/cgroups.html) and more. Cgroups kernel implementation is mostly in non-critical paths in terms of performance. The cgroups subsystem implements a new Virtual File System (VFS) type named "cgroups". All cgroups actions are done by filesystem actions, like creating cgroups directories in a cgroup filesystem, writing or reading to entries in these directories, mounting cgroup filesystems, etc. There is a library called libcgroup (a.k.a. libcg), which provides a set of userspace utilities for cgroups management: for example, cgcreate to create a new cgroup, cgdelete to delete a cgroup, cgexec to run a task in a specified control group, and more. In fact this is done by calling the cgroup filesystem operations from the libcg library. The libcg library is likely to see reduced usage in the future because it doesn't provide any coordination among multiple parties trying to use the cgroup controllers. It could be that in the future all the cgroup file operations will be performed by a library or by a daemon and not directly. The cgroups subsystem, as currently implemented, needs some form of coordination, because there is only a single controller for each resource type. When multiple actors modify it, this necessarily leads to conflicts. The cgroups controllers can be used by many projects like libvirt, systemd, lxc and more, simultaneously. When working only via cgroups filesystem operations, and when all the projects try to impose their own policy through cgroups at too low a level, without knowing about each other, they

may accidently walk over each other. When each will talk to a daemon, for example, such a clash will be avoided. For more information about `libcg` see http://libcg.sourceforge.net/.

As opposed to namespaces, no new system calls were added for implementing the cgroup subsystem. As in namespaces, several cgroups can be nested. There were code additions in the boot phase, mainly for the initialization of the cgroups subsystem, and in various subsystems, like the memory subsystem or security subsystem. Following here is a short, partial list of tasks that you can perform with cgroups:

- Assign a set of CPUs to a set of processes, with the cpusets cgroup controller. You can also control the NUMA node memory is allocated from with the cpusets cgroup controller.
- Manipulate the out of memory (`oom`) killer operation or create a process with a limited amount of memory with the memory cgroup controller (`memcg`). You will see an example later in this chapter.
- Assign permissions to devices under `/dev`, with the devices cgroup. You will see later an example of using the devices cgroup in the "Cgroup Devices – A Simple Example" section.
- Assign priority to traffic (see the section "The net_prio Module" later in this chapter).
- Freeze processes with the freezer cgroup.
- Report CPU resource usage of tasks of a cgroup with the cpuacct cgroup. Note that there is also the cpu controller, which can provision CPU cycles either by priority or by absolute bandwidth and provides the same or a superset of statistics.
- Tag network traffic with a class identifier (`classid`); see the section "The cls_cgroup Classifier" later in this chapter.

Next I will describe very briefly some changes that were done for supporting cgroups.

Cgroups Implementation

The cgroup subsystem is very complex. Here are several implementation details about the cgroup subsystem that should give you a good starting point to delve into its internals:

- A new structure called `cgroup_subsys` was added (`include/linux/cgroup.h`). It represents a cgroup subsystem (also known as a cgroup controller). The following cgroup subsystems are implemented:
  - `mem_cgroup_subsys`: `mm/memcontrol.c`
  - `blkio_subsys`: `block/blk-cgroup.c`
  - `cpuset_subsys`: `kernel/cpuset.c`
  - `devices_subsys`: `security/device_cgroup.c`
  - `freezer_subsys`: `kernel/cgroup_freezer.c`
  - `net_cls_subsys`: `net/sched/cls_cgroup.c`
  - `net_prio_subsys`: `net/core/netprio_cgroup.c`
  - `perf_subsys`: `kernel/events/core.c`
  - `cpu_cgroup_subsys`: `kernel/sched/core.c`
  - `cpuacct_subsys`: `kernel/sched/core.c`
  - `hugetlb_subsys`: `mm/hugetlb_cgroup.c`

- A new structure called cgroup was added; it represents a control group (linux/cgroup.h)
- A new virtual file system was added; this was done by defining the cgroup_fs_type object and a cgroup_ops object (instance of super_operations):

```
static struct file_system_type cgroup_fs_type = {
        .name = "cgroup",
        .mount = cgroup_mount,
        .kill_sb = cgroup_kill_sb,
};
static const struct super_operations cgroup_ops = {
        .statfs = simple_statfs,
        .drop_inode = generic_delete_inode,
        .show_options = cgroup_show_options,
        .remount_fs = cgroup_remount,
};
(kernel/cgroup.c)
```

  And registering it is done like any other filesystem with the register_filesystem() method in the cgroup_init() method; see kernel/cgroup.c.

- The following sysfs entry, /sys/fs/cgroup, is created by default when the cgroup subsystem is initialized; this is done by calling kobject_create_and_add("cgroup", fs_kobj) in the cgroup_init() method. Note that cgroup controllers can be mounted also on other directories.
- There is a global array of cgroup_subsys objects named subsys, defined in kernel/cgroup.c (note that from kernel 3.11, the array name was changed from subsys to cgroup_subsys). There are CGROUP_SUBSYS_COUNT elements in this array. A procfs entry called /proc/cgroups is exported by the cgroup subsystem. You can display the elements of the global subsys array in two ways:
  - By running cat /proc/cgroups.
  - By the lssubsys utility of libcgroup-tools.
- Creating a new cgroup entails generating these four control files always under that cgroup VFS:
  - notify_on_release: Its initial value is inherited from its parent. It's represents a boolean variable, and its usage is related to the release_agent topmost-only control file, which will be explained shortly.
  - cgroup.event_control: This file enables getting notification from a cgroup, using the eventfd() system call. See man 2 eventfd, and fs/eventfd.c.
  - tasks: A list of the PIDs which are attached to this group. Attaching a process to a cgroup is done by writing the value of its PID to the tasks control file and is handled by the cgroup_attach_task() method, kernel/cgroup.c. Displaying the cgroups to which a process is attached is done by cat /proc/<processPid>/cgroup. This is handled in the kernel by the proc_cgroup_show() method, in kernel/cgroup.c.
- cgroup.procs: A list of the thread group ids which are attached to this cgroup. The tasks entry allows attaching threads of the same process to different cgroup controllers, whereas cgroup.procs has a process-level granularity (all threads of a single process are moved together and belong to the same cgroup).

- In addition to these four control files, a control file named `release_agent` is created for the topmost cgroup root object only. The value of this file is a path of an executable that will be executed when the last process of a cgroup is terminated; the `notify_on_release` mentioned earlier should be set so that the `release_agent` feature will be enabled. The `release_agent` can be assigned as a cgroup mount option; this is the case, for example, in `systemd` in Fedora. The `release_agent` mechanism is based on a user-mode helper: the `call_usermodehelper()` method is invoked and a new userspace process is created each time that the `release_agent` is activated, which is costly in terms of performance. See: "The past, present, and future of control groups", `lwn.net/Articles/574317/`. For the `release_agent` implementation details see the `cgroup_release_agent()` method in `kernel/cgroup.c`.

- Apart from these four default control files and the `release_agent` topmost-only control file, each subsystem can create its own specific control files. This is done by defining an array of `cftype` (Control File type) objects and assigning this array to the `base_cftypes` member of the `cgroup_subsys` object. For example, for the memory cgroup controller, we have this definition for the `usage_in_bytes` control file:

```
static struct cftype mem_cgroup_files[] = {
        {
                .name = "usage_in_bytes",
                .private = MEMFILE_PRIVATE(_MEM, RES_USAGE),
                .read = mem_cgroup_read,
                .register_event = mem_cgroup_usage_register_event,
                .unregister_event = mem_cgroup_usage_unregister_event,
        },
        . . .

struct cgroup_subsys mem_cgroup_subsys = {
        .name = "memory",
        . . .
        .base_cftypes = mem_cgroup_files,
};
(mm/memcontrol.c)
```

- A member called `cgroups`, which is a pointer to a `css_set` object, was added to the process descriptor, `task_struct`. The `css_set` object contains an array of pointers to `cgroup_subsys_state` objects (one such pointer for each cgroup subsystem). The process descriptor itself (`task_struct`) does not contain a direct pointer to a cgroup subsystem it is associated to, but this could be determined from this array of `cgroup_subsys_state` pointers.

Two cgroups networking modules were added. They will be discussed later in this section:

- `net_prio` (`net/core/netprio_cgroup.c`).
- `cls_cgroup` (`net/sched/cls_cgroup.c`).

■ **Note** The cgroup subsystem is still in its early days and likely to see a fair amount of development in its features and interface.

Next you will see a short example that illustrates how the devices cgroup controller can be used to change the write permission of a device file.

Cgroup Devices Controller: A Simple Example

Let's look at a simple example of using the devices cgroup. Running the following command will create a devices cgroup:

```
mkdir   /sys/fs/cgroup/devices/0
```

Three control files will be created under `/sys/fs/cgroup/devices/0`:

- `devices.deny`: Devices for which access is denied.
- `devices.allow`: Devices for which access is allowed.
- `devices.list`: Available devices.

Each such control file consists of four fields:

- `type`: possible values are: 'a' is all, 'c' is char device and 'b' is block device.
- The device major number.
- The device minor number.
- Access permission: 'r' is permission to read, 'w' is permission to write, and 'm' is permission to perform `mknod`.

By default, when creating a new devices cgroup, it has all the permissions:

```
cat /sys/fs/cgroup/devices/0/devices.list
a *:* rwm
```

The following command adds the current shell to the devices cgroup that you created earlier:

```
echo $$ > /sys/fs/cgroup/devices/0/tasks
```

The following command will deny access from all devices:

```
echo a > /sys/fs/cgroup/devices/0/devices.deny
echo "test" > /dev/null
-bash: /dev/null: Operation not permitted
```

The following command will return the access permission for all devices:

```
echo a >  /sys/fs/cgroup/devices/0/devices.allow
```

Running the following command, which previously failed, will succeed now:

```
echo "test" > /dev/null
```

Cgroup Memory Controller: A Simple Example

You can disable the out of memory (OOM) killer thus, for example:

```
mkdir /sys/fs/cgroup/memory/0
echo $$ > /sys/fs/cgroup/memory/0/tasks
echo 1 > /sys/fs/cgroup/memory/0/memory.oom_control
```

Now if you will run some memory-hogging userspace program, the OOM killer will not be invoked. Enabling the OOM killer can be done by:

```
echo 0 > /sys/fs/cgroup/memory/0/memory.oom_control
```

You can use the `eventfd()` system call the get notifications in a userspace application about a change in the status of a cgroup. See `man 2 eventfd`.

■ **Note** You can limit the memory a process in a cgroup can have up to 20M, for example, by:

echo 20M > /sys/fs/cgroup/memory/0/memory.limit_in_bytes

The net_prio Module

The network priority control group (`net_prio`) provides an interface for setting the priority of network traffic that is generated by various userspace applications. Usually this can be done by setting the SO_PRIORITY socket option, which sets the priority of the SKB, but it is not always wanted to use this socket option. To support the `net_prio` module, an object called `priomap`, an instance of `netprio_map` structure, was added to the `net_device` object. Let's take a look at the `netprio_map` structure:

```
struct netprio_map {
        struct rcu_head rcu;
        u32 priomap_len;
        u32 priomap[];
};
(include/net/netprio_cgroup.h)
```

The `priomap` array is using the `net_prio sysfs` entries, as you will see shortly. The `net_prio` module exports two entries to cgroup `sysfs`: `net_prio.ifpriomap` and `net_prio.prioidx`. The `net_prio.ifpriomap` is used to set the `priomap` object of a specified network device, as you will see in the example immediately following. In the Tx path, the `dev_queue_xmit()` method invokes the `skb_update_prio()` method to set `skb->priority` according to the `priomap` which is associated with the outgoing network device (`skb->dev`). The `net_prio.prioidx` is a read-only entry, which shows the id of the cgroup. The `net_prio` module is a good example of how simple it is to develop a cgroup kernel module in less than 400 lines of code. The `net_prio` module was developed by Neil Horman and is available from kernel 3.3. For more information see `Documentation/cgroups/net_prio.txt`. The following is an example of how to use the network priority cgroup module (note that you must load the `netprio_cgroup.ko` kernel module in case CONFIG_NETPRIO_CGROUP is set as a module and not as a built-in):

```
mkdir /sys/fs/cgroup/net_prio
mount -t cgroup -onet_prio none /sys/fs/cgroup/net_prio
mkdir /sys/fs/cgroup/net_prio/0
echo "eth1 4" > /sys/fs/cgroup/net_prio/0/net_prio.ifpriomap
```

This sequence of commands would set any traffic originating from processes belonging to the netprio "0" group and outgoing on interface `eth1` to have the priority of four. The last command triggers writing an entry to a field in the `net_device` object called `priomap`.

■ **Note** In order to work with net_prio, CONFIG_NETPRIO_CGROUP should be set.

The cls_cgroup Classifier

The `cls_cgroup` classifier provides an interface to tag network packets with a class identifier (`classid`). You can use it in conjunction with the `tc` tool to assign different priorities to packets from different cgroups, as the example that you will soon see demonstrates. The `cls_cgroup` module exports one entry to cgroup `sysfs`, `net_cls.classid`. The control group classifier (`cls_cgroup`) was merged in kernel 2.6.29 and was developed by Thomas Graf. Like the `net_prio` module which was discussed in the previous section, also this cgroup kernel module is less than 400 lines of code, which proves again that adding a cgroup controller by a kernel module is not a heavy task. Here is an example of using the control group classifier (note that you must load the `cls_cgroup.ko` kernel module in case that CONFIG_NETPRIO_CGROUP is set as a module and not as a built-in):

```
mkdir /sys/fs/cgroup/net_cls
mount -t cgroup -onet_cls none /sys/fs/cgroup/net_cls
mkdir /sys/fs/cgroup/net_cls/0
echo 0x100001 > /sys/fs/cgroup/net_cls/0/net_cls.classid
```

The last command assigns classid 10:1 to group 0. The `iproute2` package contains a utility named `tc` for managing traffic control settings. You can use the `tc` tool with this class id, for example:

```
tc qdisc add dev eth0 root handle 10: htb
tc class add dev eth0 parent 10: classid 10:1 htb rate 40mbit
tc filter add dev eth0 parent 10: protocol ip prio 10 handle 1: cgroup
```

For more information see `Documentation/cgroups/net_cls.txt` (only from kernel 3.10.)

■ **Note** In order to work with `cls_cgroup`, CONFIG_NET_CLS_CGROUP should be set.

I will conclude the discussion about the cgroup subsystem with a short section about mounting cgroups.

Mounting cgroup Subsystems

Mounting a cgroup subsystem can be done also in other mount points than `/sys/fs/cgroup`, which is created by default. For example, you can mount the memory controller on `/mycgroup/mymemtest` by the following sequence:

```
mkdir –p /mycgroup/mymemtest
mount -t cgroup -o memory mymemtest /mycgroup/mymemtest
```

Here are some of the mount options when mounting cgroup subsystems:

- `all`: Mount all cgroup controllers.
- `none`: Do not mount any controller.
- `release_agent`: A path to an executable which will be executed when the last process of a cgroup is terminated. `Systemd` uses the `release_agent` cgroup mount option.

- noprefix: Avoid prefix in control files. Each cgroup controller has its own prefix for its own control files; for example, the cpuset controller entry mem_exclusive appears as cpuset.mem_exclusive. The noprefix mount option avoids adding the controller prefix. For example,

```
mkdir /cgroup
mount -t tmpfs xxx /cgroup/
mount -t cgroup -o noprefix,cpuset xxx /cgroup/
ls /cgroup/
cgroup.clone_children  mem_hardwall             mems
cgroup.event_control   memory_migrate           notify_on_release
cgroup.procs           memory_pressure          release_agent
cpu_exclusive          memory_pressure_enabled  sched_load_balance
cpus                   memory_spread_page       sched_relax_domain_level
mem_exclusive          memory_spread_slab       tasks
```

■ **Note** Readers who want to delve into how parsing of the cgroups mount options is implemented should look into the parse_cgroupfs_options() method, kernel/cgroup.c.

For more information about cgroups, see the following resources:

- Documentation/cgroups
- cgroups mailing list: cgroups@vger.kernel.org
- cgroups mailing list archives: http://news.gmane.org/gmane.linux.kernel.cgroups
- git repository: git://git.kernel.org/pub/scm/linux/kernel/git/tj/cgroup.git

---

■ **Note** Linux namespaces and cgroups are orthogonal and are not related technically. You can build a kernel with namespaces support and without cgroups support, and vice versa. In the past there were experiments with a cgroups namespace subsystem, called "ns", but the code was eventually removed.

You have seen what cgroups are and you learned about its two network modules, net_prio and cls_cgroup. You also saw short examples demonstrating how the devices, memory, and the networking cgroups controllers can be used. The Busy Poll Sockets feature, which was added in kernel 3.11 and above, provides lower latency for sockets. Let's take a look at how it is implemented and how it is configured and used.

Busy Poll Sockets

The traditional way the networking stack operates when the socket queue runs dry, is that it will sleep waiting for the driver to put more data on the socket queue, or returns if it is a non-blocking operation. This causes additional latency due to interrupts and context switches. For sockets applications that need the lowest possible latency and are willing to pay a cost of higher CPU utilization, Linux has added a capability for Busy Polling on Sockets from kernel 3.11 and above (in the beginning this technique was called Low Latency Sockets Poll, but it was changed to Busy Poll Sockets according to Linus suggestion). Busy Polling takes a more aggressive approach toward moving data to the application. When the application asks for more data and there is none in the socket queue, the networking stack actively calls into

the device driver. The driver checks for newly arrived data and pushes it through the network layer (L3) to the socket. The driver may find data for other sockets and will push that data as well. When the poll call returns to the networking stack, the socket code checks whether new data is pending on the socket receive queue.

In order that a network driver will support busy polling, it should supply its busy polling method and add it as the `ndo_busy_poll` callback of the `net_device_ops` object. This driver `ndo_busy_poll` callback should move the packets into the network stack; see for example, the `ixgbe_low_latency_recv()` method, `drivers/net/ethernet/intel/ixgbe/ixgbe_main.c`. This `ndo_busy_poll` callback should return the number of packets that were moved to the stack or 0 if there were no such packets, and LL_FLUSH_FAILED or LL_FLUSH_BUSY in case of some problem. An unmodified driver that does not fill in the `ndo_busy_poll` callback will continue to work as usual and will not be busy polled.

An important component to providing low latency is busy polling. Sometimes when the driver polling routine returns with no data, more data is arriving and just misses being returned to the networking stack. This is where busy polling comes in to play. The networking stack polls the driver for a configurable period of time so new packets can be picked up as soon as they arrive.

The active and busy polling of the device driver can provide reduced latency very close to that of the hardware. Busy polling can be used for large numbers of sockets at the same time but will not yield the best results, since busy polling on some sockets will slow down other sockets when using the same CPU core. Figure 14-1 contrasts the traditional receive flow with that of a socket that has been enabled for Busy Polling.

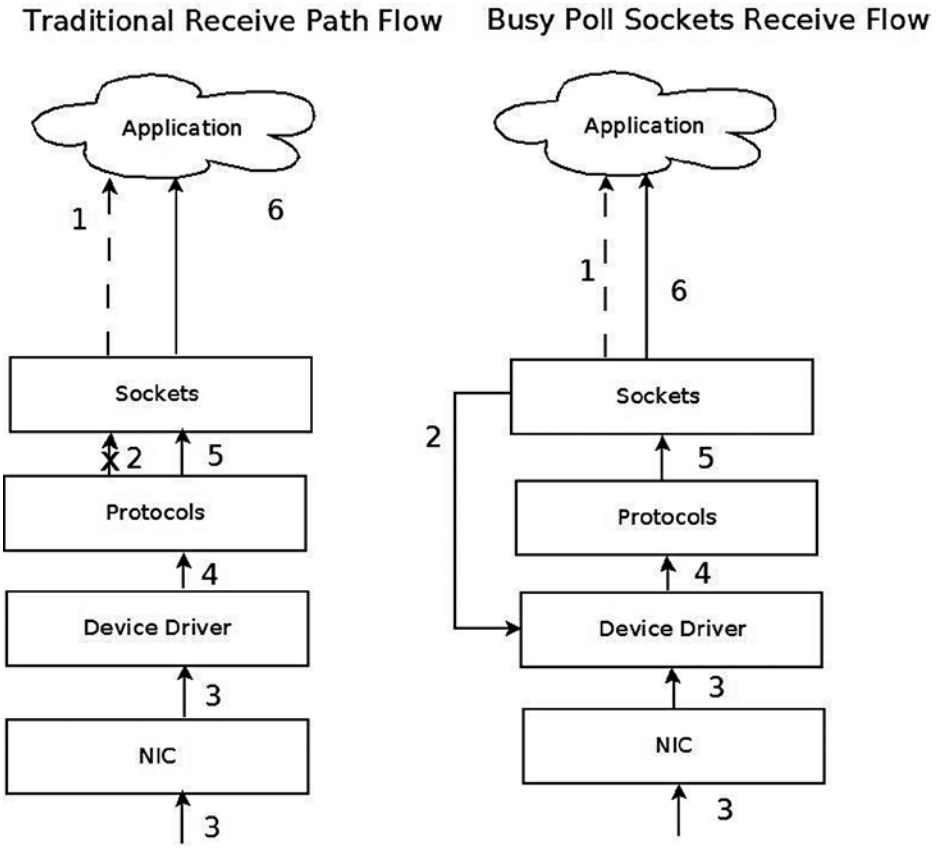

Figure 14-1. *Traditional receive flow versus Busy Poll Sockets receive flow*

```
1. Application checks for receive.
2. No immediate receive - thus block.
3. Packet Received.
4. Driver passes packet to the protocol layer.
5. Protocol/socket wakes application.
   - Bypass context switch and interrupt.
6. Application receives data through sockets.
   Repeat.
```

```
1. Application checks for receive
2. Check device driver for pending packet (poll starts).
3. Meanwhile, packet received to NIC.
4. Driver processes pending packet
5. Driver passes to the protocol layer

6. Application receives data through sockets.
Repeat.
```

Enabling Globally

Busy Polling on Sockets can be turned on globally for all sockets via `procfs` parameters or it can be turned on for individual sockets by setting the SO_BUSY_POLL socket option. For global enabling, there are two parameters: `net.core.busy_poll` and `net.core.busy_read`, which are exported to `procfs` by `/proc/sys/net/core/busy_poll` and `/proc/sys/net/core/busy_read`, respectively. Both are zero by default, which means that Busy Polling is off. Setting these values will enable Busy Polling globally. A value of 50 will usually yield good results, but some experimentation might help find a better value for some applications.

- `busy_read` controls the time limit when busy polling on blocking read operations. For a non-blocking read, if busy polling is enabled for the socket, the stack code polls just once before returning control to the user.
- `busy_poll` controls how long select and poll will busy poll waiting for new events on any of the sockets that are enabled for Busy Polling. Only sockets with the busy read socket operation enabled are busy polled.

For more information, see: `Documentation/sysctl/net.txt`.

Enabling Per Socket

A better way to enable Busy Polling is to modify the application to use the SO_BUSY_POLL socket option, which sets the `sk_ll_usec` of the socket object (an instance of the `sock` structure). By using this socket option, an application can specify which sockets are Busy Polled so CPU utilization is increased only for those sockets. Sockets from other applications and services will continue to use the traditional receive path. The recommended starting value for SO_BUSY_POLL is 50. The `sysctl.net.busy_read` value must be set to 0 and the `sysctl.net.busy_poll` value should be set as described in `Documentation/sysctl/net.txt`.

Tuning and Configuration

Here are several ways in which you can tune and configure Busy Poll sockets:

- The interrupt coalescing (`ethtool -C` setting for `rx-usecs`) on the network device should be on the order of 100 to lower the interrupt rate. This limits the number of context switches caused by interrupts.
- Disabling GRO and LRO by using `ethtool -K` on the network device may avoid out of order packets on the receive queue. This should only be an issue when mixed bulk and low latency traffic arrive on the same queue. Generally, keeping GRO and LRO enabled usually gives best results.

- Application threads and the network device IRQs should be bound to separate CPU cores. Both sets of cores should be on the same CPU NUMA node as the network device. When the application and the IRQ run on the same core, there is a small penalty. If interrupt coalescing is set to a low value this penalty can be very large.
- For lowest latency, it may help to turn off the I/O Memory Management Unit (IOMMU) support. This may already be disabled by default on some systems.

Performance

Many applications that use Busy Polling Sockets should show reduced latency and jitter as well as improved transactions per second. However, overloading the system with too many sockets that are busy polling can hurt performance as CPU contention increases. The parameters `net.core.busy_poll`, `net.core.busy_read` and the SO_BUSY_POLL socket option are all tunable. Experimenting with these values may give better results for various applications.

I will now start a discussion of three wireless subsystems, which typically serve short range and low power devices: the Bluetooth subsystem, IEEE 802.15.4 and NFC. There is a growing interest in these three subsystems as new exciting features are added quite steadily. I will start the discussion with the Bluetooth subsystem.

The Linux Bluetooth Subsystem

The Bluetooth protocol is one of the major transport protocols mainly for small and embedded devices. Bluetooth network interfaces are included nowadays in almost every new laptop or tablet and in every mobile phone, and in many electronic gadgets. The Bluetooth protocol was created by the mobile vendor Ericsson in 1994. In the beginning, it was intended to be a cable-replacement for point-to-point connections. Later, it evolved to enable wireless Personal Area Networks (PANs). Bluetooth operates in the 2.4 GHz Industrial, Scientific and Medical (ISM) radio-frequency band, which is license-free for low-power transmissions. The Bluetooth specifications are formalized by the Bluetooth Special Interest Group (SIG), which was founded in 1998; see `https://www.bluetooth.org`. The SIG is responsible for development of Bluetooth specification and for the qualification process that helps to ensure interoperability between Bluetooth devices from different vendors. The Bluetooth core specification is freely available. There were several specifications for Bluetooth over the years, I will mention the most recent:

- Bluetooth v2.0 + Enhanced Data Rate (EDR) from 2004.
- Bluetooth v2.1 + EDR 2007; included improvement of the pairing process by secure simple pairing (SSP).
- Bluetooth v3.0 + HS (High Speed) from 2009; the main new feature is AMP (Alternate MAC/PHY), the addition of 802.11 as a high-speed transport.
- Bluetooth v4.0 + BLE (Bluetooth Low Energy, which was formerly known as WiBree) from 2010.

There is a variety of uses for the Bluetooth protocol, like file transfer, audio streaming, health-care devices, networking, and more. Bluetooth is designed for short distance data exchange, in a range that typically extends up to 10 meters. There are three classes of Bluetooth devices, with the following ranges:

- Class 1 – about 100 m.
- Class 2 – about 10 m.
- Class 3 – about 1 m.

The Linux Bluetooth protocol stack is called BlueZ. Originally it was a project started by Qualcomm. It was officially integrated in kernel 2.4.6 (2001). Figure 14-2 shows the Bluetooth stack.

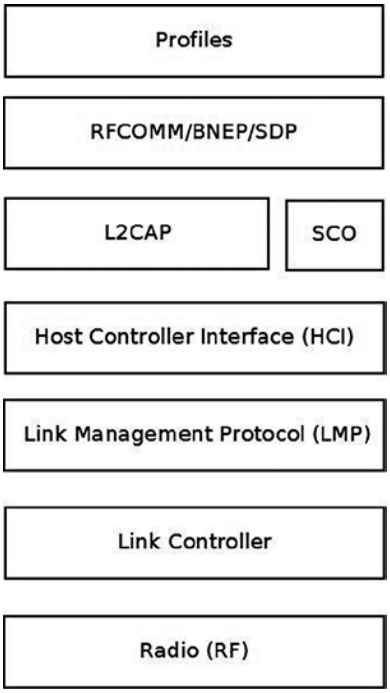

Figure 14-2. *Bluetooth stack. Note: In the layer above L2CAP there can be other Bluetooth protocols that are not discussed in this chapter, like AVDTP (Audio/Video Distribution Transport Protocol), HFP (Hands-Free Profile), Audio/video control transport protocol (AVCTP), and more*

- The lower three layers (The RADIO layer, Link controller and Link Management Protocol) are implemented in hardware or firmware.
- The Host Controller Interface (HCI) specifies how the host interacts and communicates with a local Bluetooth device (the controller). I will discuss it in the "HCI Layer" section, later in this chapter.
- The L2CAP (Logical link control and adaptation protocol) provides the ability to transmit and to receive packets from other Bluetooth devices. An application can use the L2CAP protocol as a message-based, unreliable data-delivery transport protocol similarly to the UDP protocol. Access to the L2CAP protocol from userspace is done by BSD sockets API, which was discussed in Chapter 11. Note that in L2CAP, packets are always delivered in the order they were sent, as opposed to UDP. In Figure 14-2, I showed three protocols that are located on top of L2CAP (there are other protocols on top of L2CAP that are not discussed in this chapter, as mentioned earlier).
    - BNEP: Bluetooth Network Encapsulation Protocol. I will present an example of using the BNEP protocol later in this chapter.
    - RFCOMM: The Radio Frequency Communications (RFCOMM) protocol is a reliable streams-based protocol. RFCOMM allows operation over only 30 ports. RFCOMM is used for emulating communication over a serial port and for sending unframed data.

    - SDP: Service Discovery Protocol. Enables an application to register a description and a port number in an SDP server it runs. Clients can perform a lookup in the SDP server providing the description.
- The SCO (Synchronous Connection-Oriented) Layer: for sending audio; I do not delve into its details in this chapter as it falls outside the scope of this book.
- Bluetooth profiles are definitions of possible applications and specify general behaviors that Bluetooth- enabled devices use to communicate with other Bluetooth devices. There are many Bluetooth profiles, and I will mention some of the most commonly used ones:
    - File Transfer Profile (FTP): Manipulates and transfers objects (files and folders) in an object store (file system) of another system.
    - Health Device Profile (HDP): Handles medical data.
    - Human Interface Device Profile (HID): A wrapper of USB HID (Human Interface Device) that provides support for devices like mice and keyboards.
    - Object Push Profile (OPP) – Push objects profile.
    - Personal Area Networking Profile (PAN): Provides networking over a Bluetooth link; you will see an example of it in the BNEP section later in this chapter.
    - Headset Profile (HSP): Provides support for Bluetooth headsets, which are used with mobile phones.

The seven layers in this diagram are roughly parallel to the seven layers of the OS model. The Radio (RF) layer is parallel to the Physical layer, the Link Controller is parallel to the Data Link Layer, the Link Management Protocol is parallel to the Network Protocol, and so on. The Linux Bluetooth subsystem consists of several ingredients:

- Bluetooth Core
    - HCI device and connection manager, scheduler; files: `net/bluetooth/hci*.c`, `net/bluetooth/mgmt.c`.
    - Bluetooth Address Family sockets; file: `net/bluetooth/af_bluetooth.c`.
    - SCO audio links; file: `net/bluetooth/sco.c`.
    - L2CAP (Logical Link Control and Adaptation Protocol); files: `net/bluetooth/l2cap*.c`.
    - SMP (Security Manager Protocol) on LE (Low Energy) links; file: `net/bluetooth/smp.c`
    - AMP manager - Alternate MAC/PHY management; file: `net/bluetooth/a2mp.c`.
- HCI Device drivers (Interface to the hardware); files: `drivers/bluetooth/*`. Includes vendor specific drivers as well as generic drivers, like the Bluetooth USB generic driver, `btusb`.
- RFCOMM Module (RFCOMM Protocol); files: `net/bluetooth/rfcomm/*`.
- BNEP Module (Bluetooth Network Encapsulation Protocol); files: `net/bluetooth/bnep/*`.
- CMTP Module (CAPI Message Transport Protocol), used by the ISDN protocol. CMTP is in fact obsolete; files: `net/bluetooth/cmtp/*`.
- HIDP Module (Human Interface Device Protocol); files: `net/bluetooth/hidp/*`.

I discussed briefly the Bluetooth protocol, the architecture of the Bluetooth stack and the Linux Bluetooth subsystem tree, and Bluetooth profiles. In the next section I will describe the HCI layer, which is the first layer above the LMP (see Figure 14-2 earlier in this section).

HCI Layer

I will start the discussion of the HCI layer with describing the HCI device, which represents a Bluetooth controller. Later in this section I will describe the interface between the HCI layer and the layer below it, the Link Controller layer, and the interface between the HCI and the layers above it, L2CAP and SCO.

HCI Device

A Bluetooth device is represented by `struct hci_dev`. This structure is quite big (over 100 members), and will partially be shown here:

```
struct hci_dev {
         char            name[8];
         unsigned long   flags;
         __u8            bus;
         bdaddr_t        bdaddr;
         __u8            dev_type;
         . . .
         struct work_struct      rx_work;
         struct work_struct      cmd_work;
         . . .
         struct sk_buff_head     rx_q;
         struct sk_buff_head     raw_q;
         struct sk_buff_head     cmd_q;
         . . .
         int (*open)(struct hci_dev *hdev);
         int (*close)(struct hci_dev *hdev);
         int (*flush)(struct hci_dev *hdev);
         int (*send)(struct sk_buff *skb);
         void (*notify)(struct hci_dev *hdev, unsigned int evt);
         int (*ioctl)(struct hci_dev *hdev, unsigned int cmd, unsigned long arg);
}
(include/net/bluetooth/hci_core.h)
```

Here is a description of some of the important members of the `hci_dev` structure:

- `flags`: Represents the state of a device, like HCI_UP or HCI_INIT.
- `bus`: The bus associated with the device, like USB (HCI_USB), UART (HCI_UART), PCI (HCI_PCI), etc. (see `include/net/bluetooth/hci.h`).
- `bdaddr`: Each HCI device has a unique address of 48 bits. It is exported to `sysfs` by: `/sys/class/bluetooth/<hciDeviceName>/address`
- `dev_type`: There are two types of Bluetooth devices:
  - Basic Rate devices (HCI_BREDR).
  - Alternate MAC and PHY devices (HCI_AMP).
- `rx_work`: Handles receiving packets that are kept in the `rx_q` queue of the HCI device, by the `hci_rx_work()` callback.
- `cmd_work`: Handles sending command packets which are kept in the `cmd_q` queue of the HCI device, by the `hci_cmd_work()` callback.

- `rx_q`: Receive queue of SKBs. SKBs are added to the `rx_q` by calling the `skb_queue_tail()` method when receiving an SKB, in the `hci_recv_frame()` method.
- `raw_q`: SKBs are added to the `raw_q` by calling the `skb_queue_tail()` method in the `hci_sock_sendmsg()` method.
- `cmd_q`: Command queue. SKBs are added to the `cmd_q` by calling the `skb_queue_tail()` method in the `hci_sock_sendmsg()` method.

The `hci_dev` callbacks (like `open()`, `close()`, `send()`, etc) are typically assigned in the `probe()` method of a Bluetooth device driver (for example, refer to the generic USB Bluetooth driver, `drivers/bluetooth/btusb.c`).

The HCI layer exports methods for registering/unregistering an HCI device (by the `hci_register_dev()` and the `hci_unregister_dev()` methods, respectively). Both methods get an `hci_dev` object as a single parameter. The registration will fail if the `open()` or `close()` callbacks of the specified `hci_dev` object are not defined.

There are five types of HCI packets:

- HCI_COMMAND_PKT: Commands sent from the host to the Bluetooth device.
- HCI_ACLDATA_PKT: Asynchronous data which is sent or received from a Bluetooth device. ACL stands for Asynchronous Connection-oriented Link (ACL) protocol.
- HCI_SCODATA_PKT: Synchronous data which is sent or received from a Bluetooth device (usually audio). SCO stands for Synchronous Connection-Oriented (SCO).
- HCI_EVENT_PKT: Sent when an event (such as connection establishment) occurs.
- HCI_VENDOR_PKT: Used in some Bluetooth device drivers for vendor specific needs.

HCI and the Layer Below It (Link Controller)

The HCI communicates with the layer below it, the Link Controller, by:

- Sending data packets (HCI_ACLDATA_PKT or HCI_SCODATA_PKT) by calling the `hci_send_frame()` method, which delegates the call to the `send()` callback of the `hci_dev` object. The `hci_send_frame()` method gets an SKB as a single parameter.
- Sending command packets (HCI_COMMAND_PKT), by calling the `hci_send_cmd()` method. For example, sending a scan command.
- Receiving data packets, by calling the `hci_acldata_packet()` method or by calling the `hci_scodata_packet()` method.
- Receiving event packets, by calling the `hci_event_packet()` method. Handling HCI commands is asynchronous; so some time after sending a command packet (HCI_COMMAND_PKT), a single event or several events are received as a response by the HCI `rx_work work_queue` (the `hci_rx_work()` method). There are more than 45 different events (see HCI_EV_* in `include/net/bluetooth/hci.h`). For example, when performing a scan for nearby Bluetooth devices using the command-line `hcitool`, by `hcitool scan`, a command packet (HCI_OP_INQUIRY) is sent. As a result, three event packets are returned asynchronously to be handled by the `hci_event_packet()` method: HCI_EV_CMD_STATUS, HCI_EV_EXTENDED_INQUIRY_RESULT, and HCI_EV_INQUIRY_COMPLETE.

HCI and the Layers Above It (L2CAP/SCO)

Let's take a look at the methods by which the HCI layer communicates with the layers above it, the L2CAP layer and the SCO layer:

- HCI communicates with the L2CAP layer above it when receiving data packets by calling the `hci_acldata_packet()` method, which invokes the `l2cap_recv_acldata()` method of the L2CAP protocol.
- HCI communicates with the SCO layer above it when receiving SCO packets by calling the `hci_scodata_packet()` method, which invokes the `sco_recv_scodata()` method of the SCO protocol.

HCI Connection

The HCI connection is represented by the `hci_conn` structure:

```
struct hci_conn {
        struct list_head list;
        atomic_t         refcnt;
        bdaddr_t         dst;
        . . .
        __u8              type;

}
(include/net/bluetooth/hci_core.h)
```

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

- `refcnt`: A reference counter.
- `dst`: The Bluetooth destination address.
- `type`: Represents the type of the connection:
  - SCO_LINK for SCO connection.
  - ACL_LINK for ACL connection.
  - ESCO_LINK for Extended Synchronous connection.
  - LE_LINK – represents LE (Low Energy) connection; was added in kernel v2.6.39 to support Bluetooth V4.0, which added the LE feature.
  - AMP_LINK – Added in v3.6 to support Bluetooth AMP controllers.

An HCI connection is created by calling the `hci_connect()` method. There are three types of connections: SCO, ACL, and LE connection.

L2CAP

In order to provide several data streams, L2CAP uses channels, which are represented by the `l2cap_chan` structure (`include/net/bluetooth/l2cap.h`). There is a global linked list of channels, named `chan_list`. Access to this list is serialized by a global read-write lock, `chan_list_lock`.

The l2cap_recv_acldata() method, which I described in the section "HCI and the layers above it (L2CAP/SCO)" earlier in this chapter, is called when HCI passes data packets to the L2CAP layer. The l2cap_recv_acldata() method first performs some sanity checks and drops the packet if something is wrong, then it invokes the l2cap_recv_frame() method in case a complete packet was received. Each received packet starts with an L2CAP header:

```
struct l2cap_hdr {
        __le16     len;
        __le16     cid;
} __attribute__ ((packed));
(include/net/bluetooth/l2cap.h)
```

The l2cap_recv_frame() method checks the channel id of the received packet by inspecting the cid of the l2cap_hdr object. In case it is an L2CAP command (the cid is 0x0001) the l2cap_sig_channel() method is invoked to handle it. For example, when another Bluetooth device wants to connect to our device, an L2CAP_CONN_REQ request is received on the L2CAP signal channel, which will be handled by the l2cap_connect_req() method, net/bluetooth/l2cap_core.c. In the l2cap_connect_req() method, an L2CAP channel is created by calling the l2cap_chan_create() method, via pchan->ops->new_connection(). The L2CAP channel state is set to be BT_OPEN, and the configuration state is set to be CONF_NOT_COMPLETE. This means that the channel should be configured in order to work with it.

BNEP

The BNEP protocol enables IP over Bluetooth, which means in practical terms running TCP/IP applications on top of L2CAP Bluetooth channels. You can also run TCP/IP applications with PPP over Bluetooth RFCOMM, but networking over serial PPP link is less efficient. The BNEP protocol uses a PAN profile. I will show a short example of using the BNEP protocol to setup Bluetooth over IP, and subsequently I will describe the kernel methods which implement such communication. Delving into the details of BNEP is beyond the scope of this book. If you want to learn more, see the BNEP spec, which can be found in: http://grouper.ieee.org/groups/802/15/Bluetooth/BNEP.pdf. A very simple way to create a PAN is by running:

- On the server side:
  - pand --listen --role=NAP
  - Note: NAP stands for: Network Access Point (NAP)
- On the client side
  - pand --connect btAddressOfTheServer

On both endpoints, a virtual interface (bnep0) is created. Afterward, you can assign an IP addresses on bnep0 for both endpoints with the ifconfig command (or with the ip command), just like with Ethernet devices, and you will have a network connection over Bluetooth between these endpoints. See more in http://bluez.sourceforge.net/contrib/HOWTO-PAN.

The pand --listen command creates an L2CAP server socket, and calls the accept() system call, whereas the pand --connect btAddressOfTheServer creates an L2CAP client socket and calls the connect() system call. When the connect request is received in the server side, it sends an IOCTL of BNEPCONNADD, which is handled in the kernel by the bnep_add_connection() method (net/bluetooth/bnep/core.c), which performs the following tasks:

- Creates a BNEP session (bnep_session object).
- Adds the BNEP session object to the BNEP session list (bnep_session_list) by calling the __bnep_link_session() method.

- Creates a network device named `bnepX` (for the first BNEP device X is 0, for the second X is 1, and so on).
- Registers the network device by calling the `register_netdev()` method.
- Creates a kernel thread named "`kbnepd btDeviceName`". This kernel thread runs the `bnep_session()` method which contains an endless loop, to receive or transmit packets. This endless loop terminates only when a userspace application sends an IOCTL of BNEPCONNDEL, which calls the method `bnep_del_connection()` to set the terminate flag of the BNEP session, or when the state of the socket is changed and it is not connected anymore.
- The `bnep_session()` method invokes the `bnep_rx_frame()` method to receive incoming packets and to pass them to the network stack, and it invokes the `bnep_tx_frame()` method to send outgoing packets.

Receiving Bluetooth Packets: Diagram

Figure 14-3 shows the path of a received Bluetooth ACL packet (as opposed to SCO, which is for handling audio and is handled differently). The first layer where the packet is handled is the HCI layer, by the `hci_acldata_packet()` method. It then proceeds to the higher L2CAP layer by calling the `l2cap_recv_acldata()` method.

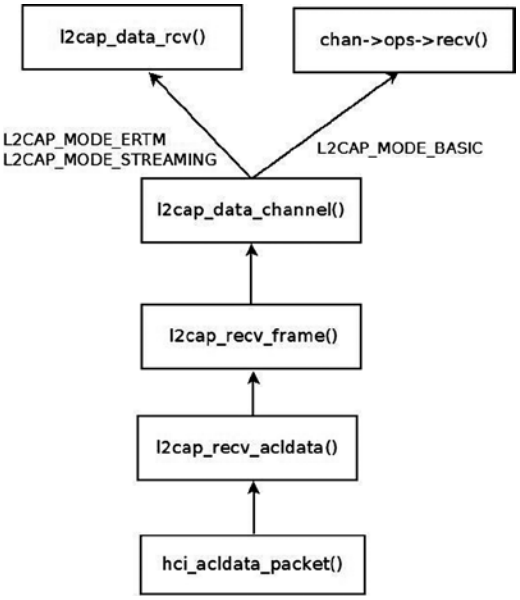

Figure 14-3. *Receiving an ACL packet*

The `l2cap_recv_acldata()` method calls the `l2cap_recv_frame()` method, which fetches the L2CAP header (the `l2cap_hdr` object was described earlier) from the SKB.

An action is being taken according to the channel ID of the L2CAP header.

L2CAP Extended Features

Support for L2CAP Extended Features (also called eL2CAP) was added in kernel 2.6.36. These extended features include:

- Enhanced Retransmission Mode (ERTM), a reliable protocol with error and flow control.
- Streaming Mode (SM), an unreliable protocol for streaming purposes.
- Frame Check Sequence (FCS), a checksum for each received packet.
- Segmentation and Reassembly (SAR) of L2CAP packets that make retransmission easier.

Some of these extensions were required for new profiles, like the Bluetooth Health Device Profile (HDP). Note that these features were available also before, but they were considered experimental and were disabled by default, and you should have set CONFIG_BT_L2CAP_EXT_FEATURES to enable them.

Bluetooth Tools

Accessing the kernel from userspace is done with sockets with minor changes: instead of using AF_INET sockets, we use AF_BLUTOOTH sockets. Here is a short description of some important and useful Bluetooth tools:

- `hciconfig`: A tool for configuring Bluetooth devices. Displays information such as the interface type (BR/EDR or AMP), its Bluetooth address, its flags, and more. The `hciconfig` tool works by opening a raw HCI socket (BTPROTO_HCI) and sending IOCTLs; for example, in order to bring up or bring down the HCI device, an HCIDEVUP or HCIDEVDOWN is sent, respectively. These IOCTLs are handled in the kernel by the `hci_sock_ioctl()` method, `net/bluetooth/hci_sock.c`.
- `hcitool`: A tool for configuring Bluetooth connections and sending some special command to Bluetooth devices. For example `hcitool scan` will scan for nearby Bluetooth devices.
- `hcidump`: Dump raw HCI data coming from and going to a Bluetooth device.
- `l2ping`: Send an L2CAP echo request and receive answer.
- `btmon`: A friendlier version of `hcidump`.
- `bluetoothctl`: A friendlier version of `hciconfig/hcitool`.

You can find more information about the Linux Bluetooth subsystem in:

- Linux BlueZ, the official Linux Bluetooth website: http://www.bluez.org.
- Linux Bluetooth mailing list: `linux-bluetooth@vger.kernel.org`.
- Linux Bluetooth mailing list archives: http://www.spinics.net/lists/linux-bluetooth/.
  - Note that this mailing list is for Bluetooth kernel patches as well as Bluetooth userspace patches.
- IRC channels on `freenode.net`:
  - `#bluez` (development related topics)
  - `#bluez-users` (non-development related topics)

In this section I described the Linux Bluetooth subsystem, focusing on the networking aspects of this subsystem. You learned about the layers of the Bluetooth stack and how they are implemented in the Linux kernel. You also learned about the important Bluetooth kernel structures like HCI device and HCI connection. Next, I will describe the second wireless subsystem, the IEEEE 802.15.4 subsystem, and its implementation.

IEEE 802.15.4 and 6LoWPAN

The IEEE 802.15.4 standard (IEEE Std 802.15.4-2011) specifies the Medium Access Control (MAC) layer and Physical (PHY) layer for Low-Rate Wireless Personal Area Networks (LR-WPANs). It is intended for low-cost and low-power consumption devices in a short-range network. Several bands are supported, among which the most common are the 2.4 GHz ISM band, 915 MHz, and 868 MHz. IEEE 802.15.4 devices can be used for example in wireless sensor networks (WSNs), security systems, industry automation systems, and more. It was designed to organize networks of sensors, switches, automation devices, etc. The maximum allowed bit rate is 250 kb/s. The standard also supports a 1000 kb/s bit rate for the 2.4 GHz band, but it is less common. Typical personal operating space is around 10m. The IEEE 802.15.4 standard is maintained by the IEEE 802.15 working group (http://www.ieee802.org/15/). There are several protocols which sit on top of IEEE 802.15.4; the most known are ZigBee and 6LoWPAN.

The ZigBee Alliance (ZA) has published non GPL specifications for IEEE802.15.4, but also the ZigBee IP (Z-IP) open standard (http://www.zigbee.org/Specifications/ZigBeeIP/Overview.aspx). It is based on Internet protocols such as IPv6, TCP, UDP, 6LoWPAN, and more. Using the IPv6 protocol for IEEE 802.15.4 is a good option because there is a huge address space of IPv6 addresses, which makes it possible to assign a unique routable address to each IPv6 node. The IPv6 header is simpler than the IPv4 header, and processing its extension headers is simpler than processing IPv4 header options. Using IPv6 with LR-WPANs is termed IPv6 over Low-power Wireless Personal Area Networks (6LoWPAN). IPv6 is not adapted for its use on an LR-WPAN and therefore requires an adaptation layer, as will be explained later in this section. There are five RFCs related to 6LoWPAN:

- RFC 4944: “Transmission of IPv6 Packets over IEEE 802.15.4 Networks.”
- RFC 4919: “IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals.”
- RFC 6282: “Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks.” This RFC introduced a new encoding format, the LOWPAN_IPHC Encoding Format, instead of LOWPAN_HC1 and LOWPAN_HC2.
- RFC 6775: “Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs).”
- RFC 6550: “RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks.”

The main challenges for implementing 6LoWPAN are:

- Different packet sizes: IPv6 has MTU of 1280 whereas IEEE802.15.4 has an MTU of 127 (IEEE802154_MTU). In order to support packets larger than 127 bytes, an adaptation layer between IPv6 and IEEE 802.15.4 should be defined. This adaptation layer is responsible for the transparent fragmentation/defragmentation of IPv6 packets.
- Different addresses: IPv6 address is 128 bit whereas IEEE802.15.4 are IEEE 64-bit extended (IEEE802154_ADDR_LONG) or, after association and after a PAN id is assigned, a 16 bit short addresses (IEEE802154_ADDR_SHORT) which are unique in that PAN. The main challenge is that we need compression mechanisms to reduce the size of a 6LoWPAN packet, largely made up of the IPv6 addresses. 6LoWPAN can for example leverage the fact that IEEE802.15.4 supports 16 bits short addresses to avoid the need of a 64-bit IID.
- Multicast is not supported natively in IEEE 802.15.4 whereas IPv6 uses multicast for ICMPv6 and for protocols that rely on ICMPv6 like the Neighbour Discovery protocol.

IEEE 802.15.4 defines four types of frames:

- Beacon frames (IEEE802154_FC_TYPE_BEACON)
- MAC command frames (IEEE802154_FC_TYPE_MAC_CMD)

- Acknowledgement frames (IEEE802154_FC_TYPE_ACK)
- Data frames (IEEE802154_FC_TYPE_DATA)

IPv6 packets must be carried on the fourth type, data frames. Acknowledgment for data packets is not mandatory, although it is recommended. As with 802.11, there are device drivers that implement most parts of the protocol by themselves (HardMAC device drivers), and device drivers that handle most of the protocol in software (SoftMAC device drivers). There are three types of nodes in 6LoWPAN:

- 6LoWPAN Node (6LN): Either a host or a router.
- 6LoWPAN Router (6LR): can send and receive Router Advertisements (RA) and Router Solicitations (RS) messages as well as forward and route IPv6 packets. These nodes are more complex than simple 6LoWPAN nodes and may need more memory and processing capacity.
- 6LoWPAN Border Router (6LBR): A border router located at the junction of separate 6LoWPAN networks or between a 6LoWPAN network and another IP network. The 6LBR is responsible for Forwarding between the IP network and the 6LoWPAN network and for the IPv6 configuration of the 6LoWPAN nodes. A 6LBR requires much more memory and processing capacity than a 6LN. They share context for the nodes in the LoWPAN, keep track of registered nodes with 6LoWPAN-ND and RPL. Generally 6LBR is always-on in contrast to 6LN who sleep most of their times. Figure 14-4 shows a simple setup with 6LBR, which connects between an IP network and a Wireless Sensor Network based on 6LoWPAN.

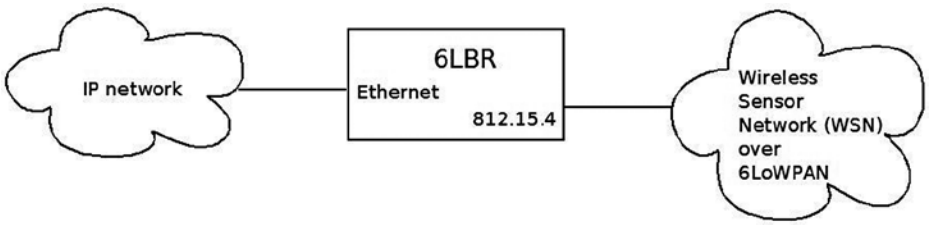

Figure 14-4. *6LBR connecting an IP network to WSN which runs over 6LoWPAN*

Neighbor Discovery Optimization

There are two reasons we should have optimizations and extensions for the IPv6 Neighboring protocol:

- IEEE 802.15.4 link layer does not have multicast support, although it supports broadcast (it uses 0xFFFF short address for message broadcasting).
- The Neighbor Discovery protocol is designed for sufficiently powered devices, and IEEE 802.15.4 devices can sleep in order to preserve energy; moreover, they operate in a lossy network environment, as the RFC puts it.

RFC 6775, which deals with Neighbor Discovery Optimization, added new optimizations such as:

- Host-initiated refresh of Router Advertisement information. In IPv6, routers usually send periodically Router Advertisements. This feature removes the need for periodic or unsolicited Router Advertisements sent from routers to hosts.
- EUI-64-based IPv6 addresses are considered to be globally unique. When such addresses are used, DAD (Duplicate Address Detection) is not needed.

- Three options were added:
  - Address Registration Option (ARO): The ARO option (33) can be a part of unicast NS message that a host sends as part of NUD (Neighbor Unreachability Detection) to determine that it can still reach a default router. When a host has a non-link-local address, it sends periodically NS messages to its default routers with the ARO options in order to register its address. Unregistration is done by sending an NS with an ARO containing a lifetime of 0.
  - 6LoWPAN Context Option (6CO): The 6CO option (34) carries prefix information for LoWPAN header compression, and is similar to Prefix Information option (PIO) which is specified in RFC 4861.
  - Authoritative Border Router Option (ABRO): The ABRO option (35) enables disseminating prefixes and context information across a route-over topology.
- Two new DAD messages were added:
  - Duplicate Address Request (DAR). New ICMPv6 type of 157.
  - Duplicate Address Confirmation (DAC). New ICMPv6 type of 158.

Linux Kernel 6LoWPAN

The 6LoWPAN basic implementation was integrated into v3.2 Linux. It was contributed by the Embedded Systems Open Platform Group, from Siemens Corporate Technology. It has three layers:

- Network layer - `net/ieee802154` (includes the 6lowpan module, Raw IEEE 802.15.4 sockets, the netlink interface, and more).
- MAC layer - `net/mac802154`. Implements a partial MAC layer for SoftMAC device drivers.
- PHY layer - `drivers/net/ieee802154` – the IEEE802154 device drivers.
- There are currently two 802.15.4 devices which are supported:
  - AT86RF230/231 transceiver driver
  - Microchip MRF24J40
- There is the Fakelb driver (IEEE 802.15.4 loopback interface).
- These two devices, as well as many other 802.15.4 transceivers, are connected via SPI. There is also a serial driver, although it is not included in the mainline kernel and still experimental. There are devices like `atusb`, which are based on an AT86RF231 BN but are not in mainline as of this writing.

6LoWPAN Initialization

In the `lowpan_init_module()` method, initialization of 6LoWPAN netlink sockets is done by calling the `lowpan_netlink_init()` method, and a protocol handler is registered for 6LoWPAN packets by calling the `dev_add_pack()` method:

```
. . .
static struct packet_type lowpan_packet_type = {
        .type = __constant_htons(ETH_P_IEEE802154),
        .func = lowpan_rcv,
};
```

```
. . .
static int __init lowpan_init_module(void)
{
        . . .
        dev_add_pack(&lowpan_packet_type);
        . . .
}
(net/ieee802154/6lowpan.c)
```

The `lowpan_rcv()` method is the main Rx handler for 6LoWPAN packets, which has an ethertype of 0x00F6 (ETH_P_IEEE802154). It handles two cases:

- Reception of uncompressed packets (dispatch type is IPv6.)
- Reception of compressed packets.

You use a virtual link to ensure the translation between 6LoWPAN and IPv6 packets. One endpoint of this virtual link speaks IPv6 and has an MTU of 1280, this is the 6LoWPAN interface. The other one speaks 6LoWPAN and has an MTU of 127, this is the WPAN interface. Compressed 6LoWPAN packets are processed by the `lowpan_process_data()` method, which calls the `lowpan_uncompress_addr()` to uncompress addresses and the `lowpan_uncompress_udp_header()` to uncompress the UDP header accordingly to the IPHC header. The uncompressed IPv6 packet is then delivered to the 6LoWPAN interface with the `lowpan_skb_deliver()` method (`net/ieee802154/6lowpan.c`).

Figure 14-5 shows the 6LoWPAN Adaptation layer.

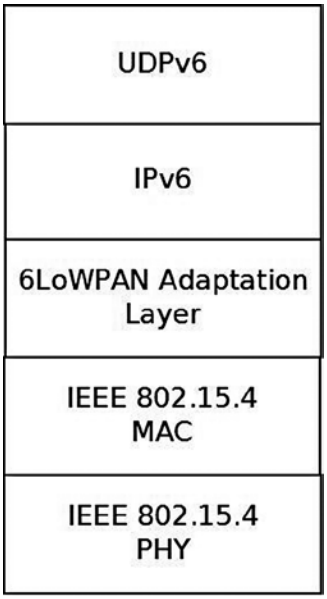

Figure 14-5. *6LoWPAN Adaptation layer*

Figure 14-6 shows the path of a packet from the PHY layer (the driver) via the MAC layer to the 6LoWPAN adaptation layer.

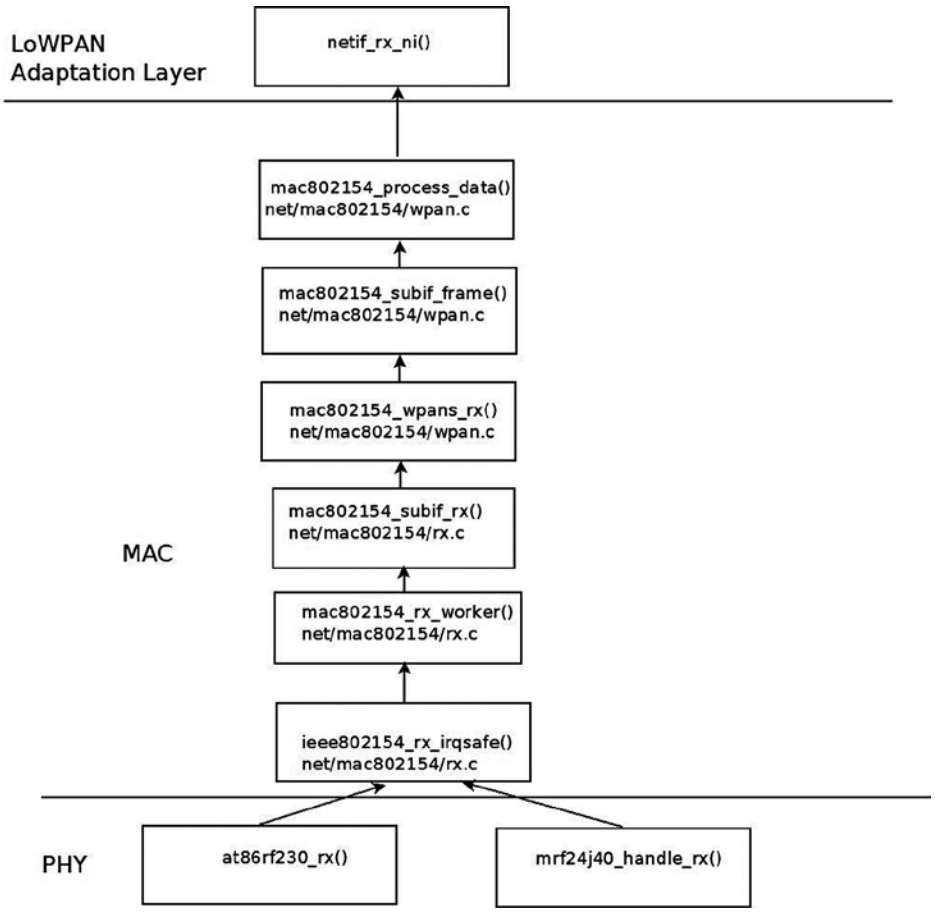

Figure 14-6. *Receiving a packet*

I will not delve into the details of the device drivers implementation, as this is out of our scope. I will mention that each device driver should create an `ieee802154_dev` object by calling the `ieee802154_alloc_device()` method, passing as a parameter an `ieee802154_ops` object. Every driver should define some `ieee802154_ops` object callbacks, like `xmit`, `start`, `stop`, and more. This applies for SoftMAC drivers only.

I will mention here that an Internet-Draft was submitted for applying 6LoWPAN technology over Bluetooth Low-Energy devices (these devices are part of the Bluetooth 4.0 specification, as was mentioned in the previous chapter). See "Transmission of IPv6 Packets over Bluetooth Low Energy," http://tools.ietf.org/html/draft-ietf-6lowpan-btle-12.

■ **Note** `Contiki` is an open source Operating System implementing the Internet of Things (IoT) concept; some patches of the Linux IEEE802.15.4 6LoWPAN are derived from it, like the UDP header compression and decompression. It implements 6LoWPAN, and RPL. It was developed by Adam Dunkels. See http://www.contiki-os.org/

For additional resources about 6LoWPAN and 802.15.4:

- Books:
  - "6LoWPAN: The Wireless Embedded Internet", by Zach Shelby and Carsten Bormann, Wiley, 2009.
  - "Interconnecting Smart Objects with IP: The Next Internet," by Jean-Philippe Vasseur and Adam Dunkels (the Contiki developer), Morgan Kaufmann, 2010.
- An article about IPv6 Neighbor Discovery Optimization: http://www.internetsociety.org/articles/ipv6-neighbor-discovery-optimization.

The `lowpan-tools` is a set of utilities to manage the Linux LoWPAN stack. See: http://sourceforge.net/projects/linux-zigbee/files/linux-zigbee-sources/0.3/

■ **Note** The IEEE802.15.4 does not maintain a `git` repository of its own (though in the past there was one). Patches are sent to the `netdev` mailing list; some of the developers send the patches first to the linux zigbee developer mailing list to get some feedback: https://lists.sourceforge.net/lists/listinfo/linux-zigbee-devel

---

I described the IEEE 802.15.4 and the 6LoWPAN protocol in this section and the challenges it poses for integration in the Linux kernel, like adding Neighboring Discovery messages. In the next section I will describe the third wireless subsystem, which is intended for the most shortest ranges among the three wireless subsystems described in this chapter: the Near Field Communication (NFC) subsystem.

Near Field Communication (NFC)

Near Field Communication is a very short range wireless technology (less than two inches) designed to transfer small amount of data over a very low latency link at up to 424 kb/s. NFC payloads range from very simple URLs or raw texts to more complex out of band data to trigger connection handover. Through its very short range and latency, NFC implements a tap and share concept by linking proximity to an immediate action triggered by the NFC data payload. Touch an NFC tag with your NFC enabled mobile phone and this will, for example, immediately fire up a web browser.

NFC runs on the 13.65MHz band and is based on the Radio Frequency ID (RFID) ISO14443 and FeliCa standards. The NFC Forum (http://www.nfc-forum.org/) is a consortium responsible for standardizing the technology through a set of specifications, ranging from the NFC Digital layer up to high-level services definitions like the NFC Connection Handover or the Personal Health Device Communication (PHDC) ones. All adopted NFC Forum specifications are available free of charge. See http://www.nfc-forum.org/specs/.

At the heart of the NFC Forum specification is the NFC Data Exchange Format (NDEF) definition. It defines the NFC data structure used to exchange NFC payloads from NFC tags or between NFC peers. All NDEFs contain one or more NDEF Records that embed the actual payload. NDEF record header contains metadata that allow applications to build the semantic link between the NFC payload and an action to trigger on the reader side.

NFC Tags

NFC tags are cheap, mostly static and battery less data containers. They're typically made of an inductive antenna connected to a very small amount of flash memory, packaged in many different form factors (labels, key rings, stickers, etc.). As per the NFC Forum definitions, NFC tags are passive devices, i.e., they're unable to generate any

radio field. Instead they're powered by NFC active devices initiated RF fields. The NFC Forum defines four different tag types, each of them carrying a strong RFID and smart card legacy:

- Type 1 specifications derive from Innovision/Broadcom Topaz and Jewel card specifications. They can expose from 96 up to 2 KBytes of data at 106 kb/s.
- Type 2 tags are based on NXP Mifare Ultralight specifications. They're very similar to Type 1 tags.
- Type 3 tags are built on top of the non-secure parts of Sony FeliCa tags. They're more expensive than Type 1 and 2 tags, but can carry up to 1 MBytes at 212 or 424 kb/s.
- Type 4 specifications are based on NXP DESFire cards, support up to 32 KBytes and three transmission speeds: 106, 212, or 424 kb/s.

NFC Devices

As opposed to NFC tags, NFC devices can generate their own magnetic field to initiate NFC communications. NFC-enabled mobile phones and NFC readers are the most common kinds of NFC devices. They support a larger feature set than NFC tags. They can read from or write to NFC tags, but they can also pretend to be a card and be seen as simple NFC tags from any reader. But one of the key advantages of the NFC technology over RFID is the possibility to have two NFC devices talking to each other in an NFC specific peer-to-peer mode. The link between two NFC devices is kept alive as long as the two devices are in magnetic range. In practice this means two NFC devices can maintain a peer-to-peer link while they physically touch each other. This introduces a whole new range of mobile use cases where one can exchange data, context, or credentials by touching someone else NFC device.

Communication and Operation Modes

The NFC Forum defines two communication and three operation modes. An active NFC communication is established when two NFC devices can talk to one another by alternatively generating the magnetic field. This implies that both devices have their own power supply as they don't rely on any inductively generated power. Active communications can only be established in NFC peer-to-peer mode. On the other hand, only one NFC device generates the radio field on a passive NFC communication, and the other device replies by using that field.

There are three NFC operation modes:

- Reader/Writer: An NFC device (e.g., an NFC-enabled mobile phone) read from or write to an NFC tag.
- Peer-to-peer: Two NFC devices establish a Logical Link Control Protocol (LLCP) over which several NFC services can be multiplexed: Simple NDEF Exchange Protocol (SNEP) for exchanging NDEF formatted data, Connection Handover for initiating a carrier (Bluetooth or WiFi) handover, or any proprietary protocol.
- Card Emulation: An NFC device replies to a reader poll by pretending to be an NFC tag. Payment and transaction issuers rely on this mode to implement contactless payments on top of NFC. In card emulation mode, payment applets running on a trusted execution environment (also known as "secure elements") take control of the NFC radio and expose themselves as a legacy payment card that can be read from an NFC-enabled point-of-sale terminal.

Host-Controller Interfaces

Communication between hardware controllers and host stacks must follow a precisely defined interface: the host-controller one (HCI). The NFC hardware ecosystem is quite fragmented in that regard, as most of the initial NFC controllers implement an ETSI specified HCI originally designed for communication between SIM cards and

contactless front-ends. (See http://www.etsi.org/deliver/etsi_ts/102600_102699/102622/07.00.00_60/ts_102622v070000p.pdf). This HCI was not tailored for NFC specific use cases, and so each and every manufacturer defined a large number of proprietary extensions to support their features. The NFC Forum tries to address that situation by defining its own interface, much more NFC oriented, the NFC Controller Interface (NCI). The industry trend is clearly showing that manufacturers abandon ETSI HCI in favor of NCI, building a more standardized hardware ecosystem.

Linux NFC support

Unlike the Android operating system NFC stack, which is described later in this section, the standard Linux one is partly implemented by the kernel itself. Since the 3.1 Linux kernel release, Linux based application will find an NFC specific socket domain, along with a generic netlink family for NFC. (See http://git.kernel.org/?p=linux/kernel/git/sameo/nfc-next.git;a=shortlog;h=refs/heads/master.) The NFC generic netlink family is intended to be an NFC out of band channel for controlling and monitoring NFC adapters. The NFC socket domain supports two families:

- Raw sockets for sending NFC frames that will arrive unmodified to the drivers
- LLCP sockets for implementing NFC peer-to-peer services

The hardware abstraction is implemented in NFC kernel drivers that register against various parts of the stack, mostly depending on the host-controller interface used by the controllers they support. As a consequence, Linux applications can work on top of a hardware agnostic and fully POSIX compatible NFC kernel APIs. The Linux NFC stack is split between kernel and userspace. The kernel NFC sockets allow userspace applications to implement NFC tags support by sending tag types specific commands through the raw protocol. NFC peer-to-peer protocols (SNEP, Connection Handover, PHDC, etc.) can be implemented by transmitting their specific payloads through NFC sockets as well. Finally, card emulation mode is built on top of the secure element parts of the kernel NFC netlink API. The Linux NFC daemon, `neard`, sits on top of the kernel and implements all three NFC modes, regardless of the NFC controller physically wired to the host platform. (See https://01.org/linux-nfc/.)

Figure 14-7 shows an overview of the NFC system.

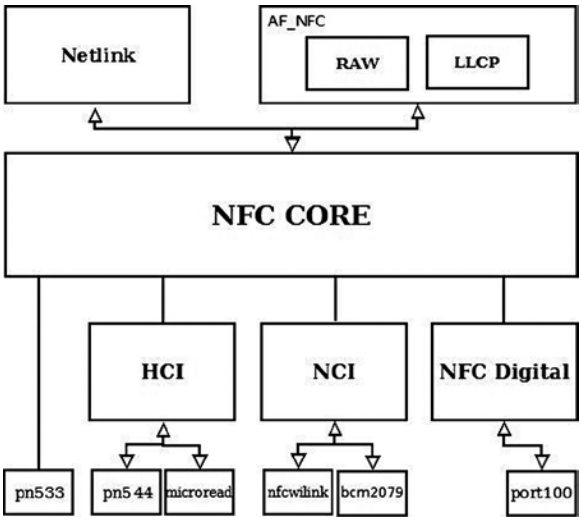

Figure 14-7. *NFC overview*

NFC Sockets

NFC sockets are of two kinds: raw and LLCP. Raw NFC sockets were designed with reader mode support in mind, as they provide a way to transmit tag specific commands and receive the tag replies back. The `neard` daemon uses NFC Raw sockets to implement all four tag types support, in both reader and writer modes. LLCP sockets implement the NFC peer-to-peer logical link control protocol on top of which `neard` implements all NFC Forum specified peer-to-peer services (SNEP, Connection Handover, and PHDC).

Depending on the selected protocol, NFC socket semantics differ.

Raw Sockets

- `connect:` Select and enable a detected NFC tag
- `bind:` Not supported
- `send/recv:` Send and receive raw NFC payloads. The NFC core implementation does not modify those payloads.

LLCP Sockets

- `connect:` Connect to a specific LLCP service on a detected peer device, like the SNEP or Connection Handover services.
- `bind:` Link a device to a specific LLCP service. The service will be exported through the LLCP service name lookup (SNL) protocol for any NFC peer device to attempt a connection to it.
- `send/recv:` Transmit LLCP service payloads to and from an NFC peer device. The kernel will handle the LLCP specific link layer encapsulation and fragmentation.
- LLCP transport can be connected or connectionless, and this is handled through the UNIX standard SOCK_STREAM and SOCK_DGRAM socket types. NFC LLCP sockets also support the SOCK_RAW type for monitoring and sniffing purposes.

NFC Netlink API

The NFC generic netlink API is designed to implement out of band NFC specific operations. It also handles any discoverable secure element from an NFC controller. Through NFC netlink commands, you can:

- List all available NFC controllers.
- Power NFC controllers up and down.
- Start (and stop) NFC polls for discovering NFC tags and devices.
- Enable NFC peer-to-peer (a.k.a. LLCP) links between the local controller and remote NFC peers.
- Send LLCP service name lookup requests, in order to discover the available LLCP services on a remote peer.
- Enable and disable NFC discoverable secure elements (typically SIM card based or embedded secure elements).
- Send ISO7816 frames to enabled secure elements.
- Trigger NFC controller firmware downloads.

The netlink API is not only about sending synchronous commands from NFC applications, but also about receiving asynchronous NFC-related events. Applications listening for broadcast NFC events on an NFC netlink socket will get notified about:

- Detected NFC tags and devices
- Discovered secure elements
- Secure element transaction status
- LLCP service name lookup replies

The entire netlink API (both commands and events) along with the socket one are exported through the kernel headers, and installed at `/usr/include/linux/nfc.h` on standard Linux distributions.

NFC Initialization

NFC initialization is done by the `nfc_init()` method:

```
static int __init nfc_init(void)
{
        int rc;
        . . .
```

Register the generic netlink NFC family and the NFC notifier callback, the `nfc_genl_rcv_nl_event()` method:

```
        rc = nfc_genl_init();
        if (rc)
                goto err_genl;

        /* the first generation must not be 0 */
        nfc_devlist_generation = 1;
```

Initialize NFC Raw sockets:

```
        rc = rawsock_init();
        if (rc)
                goto err_rawsock;
```

Initialize NFC LLCP sockets:

```
        rc = nfc_llcp_init();
        if (rc)
                goto err_llcp_sock;
```

Initialize the AF_NFC protocol:

```
        rc = af_nfc_init();
        if (rc)
                goto err_af_nfc;
```

```
        return 0;
        . . .
}
(net/nfc/core.c)
```

Drivers API

As explained earlier, most NFC controllers nowadays either use HCI or NCI as their host-controller interface. Others define their proprietary interface over USB, like most PC-compatible NFC readers, for example. There are also some "Soft" NFC controllers that expect the host platform to implement the NFC Forum Digital layer and talk to an analog-only capable firmware. In order to support this variety of hardware controllers, the NFC kernel implements NFC NCI, HCI, and Digital layers. Depending on the NFC hardware they intend to support, device driver developers will need to register at module probing time against one of these stacks, or directly against the NFC core implementation for purely proprietary protocols. When registering, they typically provide a stack operands implementation, which is the actual hardware abstraction layer between NFC kernel drivers and the core parts of the NFC stack. The NFC driver registration APIs and operand prototypes are defined in the kernel `include/net/nfc/` directory.

Figure 14-8 shows a block diagram of the NFC Linux Architecture.

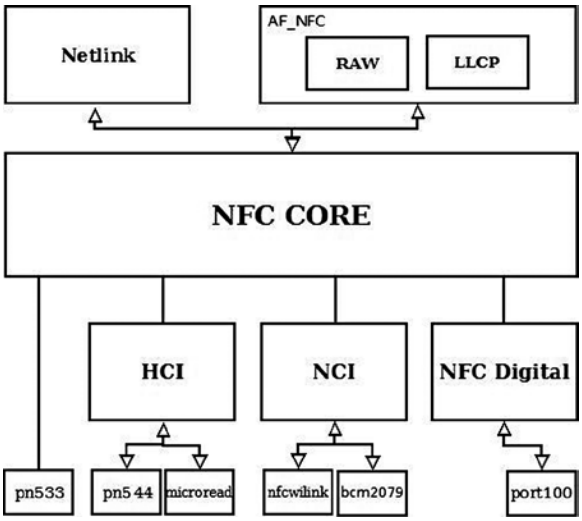

Figure 14-8. *NFC Linux Kernel Architecture. (Note that the NFC Digital layer is not in kernel 3.9. It is to be integrated into kernel 3.13.)*

The hierarchy shown in this figure can be understood better by looking into the implementation details of the registration of NFC device drivers directly to the NFC core and against the HCI and the NCI layer:

- Registration directly against the NFC core is done typically in the driver `probe()` callback. The registration is done using these steps:
  - Create an `nfc_dev` object by calling the `nfc_allocate_device()` method.
  - Call the `nfc_register_device()` method, passing the `nfc_dev` object which was created in the previous step as a single parameter.
  - See: `drivers/nfc/pn533.c`.

- Registration against the HCI layer is done typically also in the `probe()` callback of the driver; in the case of the `pn544` and `microread` NFC device drivers, which are the only HCI drivers in kernel 3.9, this `probe()` method is invoked by the I2C subsystem. The registration is done using these steps:
  - Create an `nfc_hci_dev` object by calling the `nfc_hci_allocate_device()` method.
  - The `nfc_hci_dev` structure is defined in `include/net/nfc/hci.h`.
  - Call the `nfc_hci_register_device()` method, passing the `nfc_hci_dev` object which was created in the previous step as a single parameter. The `nfc_hci_register_device()` method in turn performs a registration against the NFC core by calling the `nfc_register_device()` method.
  - See `drivers/nfc/pn544/pn544.c` and `drivers/nfc/microread/microread.c`.
- Registration against the NCI layer is done typically also in the `probe()` callback of the driver, for example in the `nfcwilink` driver. The registration is done using these steps:
  - Create an `nci_dev` object by calling the `nci_allocate_device()` method.
  - The `nci_dev` structure is defined in `include/net/nfc/nci_core.h.`
  - Call the `nci_register_device()` method, passing the `nci_dev` object that was created in the previous step as a single parameter. The `nci_register_device()` method in turn performs a registration against the NFC core by calling the `nfc_register_device()` method, similarly to what you saw earlier in this section with registration against the HCI layer.
  - See `drivers/nfc/nfcwilink.c`.

When working directly against the NFC core, the driver must define five callbacks in the `nfs_ops` object (this object is passed as a first parameter of the `nfc_allocate_device()` method):

- `start_poll`: Set the driver to work in polling mode.
- `stop_poll`: Stop polling.
- `activate_target`: Activate a chosen target.
- `deactivate_target`: Deactivate a chosen target.
- `im_transceive`: Transceive operation.

When working with HCI , the `hci_nfc_ops` object, which is an instance of `nfs_ops`, defines these five callbacks, and when allocating an HCI object with the `nfc_hci_allocate_device()` method, the `nfc_allocate_device()` method is invoked with this `hci_nfc_ops` object as a first parameter.

With NCI, there is something quite similar, with the `nci_nfc_ops` object; see: `net/nfc/nci/core.c`.

Userspace Architecture

`neard` (`http://git.kernel.org/?p=network/nfc/neard.git;a=summary`) is the Linux NFC daemon that runs on top of the kernel NFC APIs. It is a single threaded, GLib based process that implements the higher layers of the NFC peer-to-peer stack along with the four tag types specific commands for reading from and writing to NFC tags. The NDEF Push Protocol (NPP), SNEP, PHDC, and Connection Handover specifications are implemented through `neard` plugins. One of neard's main design goals is to provide a small, simple, and uniform NFC API for Linux based applications willing to provide high-level NFC services. This is achieved through a small D-Bus API that abstracts

tags and devices interfaces and methods, hiding the NFC complexity away from application developers. This API is compatible with the freedesktop D-Bus ObjectManager one and provides the following interfaces:

- `org.neard.Adapter`: For detecting new NFC controllers, turning them on and off, and starting NFC polls.
- `org.neard.Device, org.neard.Tag`: For representing detected NFC tags and devices. Calling the Device.Push method will send NDEFs to the peer device while Tag.Write will write them to the selected tag.
- `org.neard.Record`: Represents human readable and understandable NDEF record payload and properties. Registering agents against the `org.neard.NDEF`Agent interface will give application access to the NDEF raw payloads.

You can find more information about the `neard` userspace daemon here: http://git.kernel.org/cgit/network/nfc/neard.git/tree/doc.

NFC on Android

The initial NFC support was added to the Android operating system on December 2010, with the official 2.3 (Gingerbread) release. Android 2.3 only supported the reader/writer mode, but things have improved significantly since then, and the latest Android releases (Jelly Bean 4.3) come with a fully featured NFC support. For more information, see the Android NFC page: http://developer.android.com/guide/topics/connectivity/nfc/index.html. Following the classic Android architecture, a Java specific NFC API is available for applications to provide NFC services and operations. It is left to integrators to implement these APIs through native hardware abstraction layers (HAL). Google ships a Broadcom NFC HAL that currently only supports Broadcom NFC hardware. Here again, it is left to Android OEMs and integrators to either adapt the Broadcom NFC HAL to their selected NFC chipset or to implement their own HAL. It is important to note that since the Broadcom stack implements the NFC Controller Interface (NCI) specification, it is relatively easy to adapt it to support any NCI compatible NFC controller. The Android NFC architecture is what one could call a userspace NFC stack. In fact the entire NFC implementation is done in userspace through the HAL. NFC frames are then pushed down to the NFC controller through a kernel driver stub. The driver simply encapsulates those frames into buffers that are ready to be sent to the physical link (e.g., I2C, SPI, UART) between the host platform and the NFC controller.

■ **Note** Pull requests of the `nfc-next git` tree are sent to the `wireless-next` tree (Apart from the NFC subsystem, also the Bluetooth subsystem and the mac802.11 subsystem pull requests are handled by the wireless maintainer). From the `wireless-next` tree, pull requests are sent to `net-next` tree, and from there to Linus `linux-next` tree. The `nfc-next` tree is available in: `git://git.kernel.org/pub/scm/linux/kernel/git/sameo/nfc-next.git`

There is also an `nfc-fixes git` repository, which contains urgent and critical fixes for the current release(-rc*). The `git` tree of `nfc-fixes` is available in: git://git.kernel.org/pub/scm/linux/kernel/git/sameo/nfc-fixes.git/

NFC mailing list: linux-nfc@lists.01.org.

NFC mailing list archives: https://lists.01.org/pipermail/linux-nfc/.

In this section you learned about what NFC is in general, and about the Linux NFC subsystem implementation and about the Android NFC subsystem implementation. In the next section I will discuss the notification chains mechanism, which is an important mechanism to inform network devices about various events.

Notifications Chains

Network devices state can change dynamically; from time to time, the user/administrator can register/unregister network devices, change their MAC address, change their MTU, etc. The network stack and other subsystems and modules should be able to be notified about these events and handle them properly. The network notifications chains provide a mechanism for handling such events, and I will describe its API and the possible network events it handles in this section. For a full list of the events, see Table 14-1 later in this section. Every subsystem and every module can register itself to notification chains. This is done by defining a `notifier_block` and registering it. The core methods of notification chain registration and unregistration is the `notifier_chain_register()` and the `notifier_chain_unregister()` method, respectively. Generation of notification events is done by calling the `notifier_call_chain()` method. These three methods are not used directly (they are not exported; see `kernel/notifier.c`), and they do not use any locking mechanism. The following methods are wrappers around `notifier_chain_register()`, all of them implemented in `kernel/notifier.c`:

- `atomic_notifier_chain_register()`
- `blocking_notifier_chain_register()`
- `raw_notifier_chain_register()`
- `srcu_notifier_chain_register()`
- `register_die_notifier()`

Table 14-1. *Network Device Events:*

Event	Meaning
NETDEV_UP	device up event
NETDEV_DOWN	device down event
NETDEV_REBOOT	detected a hardware crash and restarted the device
NETDEV_CHANGE	device state change
NETDEV_REGISTER	device registration event
NETDEV_UNREGISTER	device unregistration event
NETDEV_CHANGEMTU	device MTU changed
NETDEV_CHANGEADDR	device MAC address changed
NETDEV_GOING_DOWN	device is going down
NETDEV_CHANGENAME	device has changed its name
NETDEV_FEAT_CHANGE	device features changed
NETDEV_BONDING_FAILOVER	bonding failover event
NETDEV_PRE_UP	this event enables to veto changing the device state to UP; for example, in cfg80211, denying interfaces to be set UP if the device is known to be rfkill'ed. see `cfg80211_netdev_notifier_call()`
NETDEV_PRE_TYPE_CHANGE	The device is about to change its type. This is a generalization of the NETDEV_BONDING_OLDTYPE flag, which was replaced by NETDEV_PRE_TYPE_CHANGE

(*continued*)

Table 14-1. (*continued*)

Event	Meaning
NETDEV_POST_TYPE_CHANGE	device changed its type. This is a generalization of the NETDEV_BONDING_NEWTYPE flag, which was replaced by NETDEV_POST_TYPE_CHANGE
NETDEV_POST_INIT	This event is generated in device registration (`register_netdevice()`), before creating the network device kobjects by `netdev_register_kobject()`; used in cfg80211 (`net/wireless/core.c`)
NETDEV_UNREGISTER_FINAL	An event which is generated to finalize the device unregistration.
NETDEV_RELEASE	the last slave of a bond is released (when working with netconsole over bonding) (This flag was also once used for bridges, in `br_if.c`).
NETDEV_NOTIFY_PEERS	notify network peers event (i.e., a device wants to inform the rest of the network about some sort of reconfiguration such as a failover event or a virtual machine migration)
NETDEV_JOIN	The device added a slave. Used for example in the bonding driver, in the `bond_enslave()` method, where we add a slave; see `drivers/net/bonding/bond_main.c`

There are also corresponding wrapper methods for unregistering notification chains and for generating notification events for each of these wrappers. For example, for the notification chain registered with the `atomic_notifier_chain_register()` method, the `atomic_notifier_chain_unregister()` is for unregistering the notification chain, and the `__atomic_notifier_call_chain()` method is for generating notification events. Each of these wrappers has also a corresponding macro to define a notification chain; for the `atomic_notifier_chain_register()` wrapper it is the `ATOMIC_NOTIFIER_HEAD` macro (`include/linux/notifier.h`).

After registering a `notifier_block` object, when every one of the events shown in Table 14-1 occurs, the callback specified in a `notifier_block` is invoked. The fundamental data structure of notification chains is the `notifier_block` structure; let's take a look:

```
struct notifier_block {
        int (*notifier_call)(struct notifier_block *, unsigned long, void *);
        struct notifier_block __rcu *next;
        int priority;
};
(include/linux/notifier.h)
```

- `notifier_call`: The callback to be invoked.
- `priority`: callbacks of `notifier_block` objects with higher priority are performed first.

There are many chains in the networking subsystem and in other subsystems. Let's mention some of the important ones:

- `netdev_chain`: Registered by the `register_netdevice_notifier()` method and unregistered by the `unregister_netdevice_notifier()` method (`net/core/dev.c`).
- `inet6addr_chain`: Registered by the `register_inet6addr_notifier()` method and unregistered by the `unregister_inet6addr_notifier ()` method. Notifications are generated by the `inet6addr_notifier_call_chain ()` method (`net/ipv6/addrconf_core.c`).

- netevent_notif_chain: Registered by the register_netevent_notifier() method and unregistered by the unregister_netevent_notifier() method. Notifications are generated by the call_netevent_notifiers() method (net/core/netevent.c).
- inetaddr_chain: Registered by the register_inetaddr_notifier() method and unregistered by the unregister_inetaddr_notifier() method. Notifications are generated by calling the blocking_notifier_call_chain() method.

Let's take a look at an example of using the netdev_chain; you saw earlier that with netdev_chain, registration is done with the register_netdevice_notifier() method, which is a wrapper around the raw_notifier_chain_register() method. Following is an example of registering a callback named br_device_event; First, a notifier_block object is defined, and then it is registered by calling the register_netdevice_notifier() method:

```
struct notifier_block br_device_notifier = {
        .notifier_call = br_device_event
};
(net/bridge/br_notify.c)
static int __init br_init(void)
{
        ...
        register_netdevice_notifier(&br_device_notifier);
        ...
}
(net/bridge/br.c)
```

Notifications of the netdev_chain are generated by invoking the call_netdevice_notifiers() method. The first parameter of this method is the event. The call_netdevice_notifiers() method :is in fact a wrapper around raw_notifier_call_chain().

So, when a network notification is generated, all callbacks which were registered are invoked; in this example, the br_device_event() callback will be called, regardless of which network event occurred; the callback will decide how to handle the notification, or maybe it will ignore it. Let's take a look at the callback method, br_device_event():

```
static int br_device_event(struct notifier_block *unused, unsigned long event, void *ptr)
{
        struct net_device *dev = ptr;
        struct net_bridge_port *p;
        struct net_bridge *br;
        bool changed_addr;
        int err;
        . . .
```

The second parameter for the br_device_event() method is the event (all the events are defined in include/linux/netdevice.h):

```
        switch (event) {
        case NETDEV_CHANGEMTU:
                dev_set_mtu(br->dev, br_min_mtu(br));
                break;
        . . .
}
```

■ **Note** Registration of notification chains is not limited only to the networking subsystem. Thus, for example, the `clockevents` subsystem defines a chain called `clockevents_chain` and registers it by calling the `raw_notifier_chain_register()` method, and the `hung_task` module defines a chain named `panic_notifier_list` and registers it by calling the `atomic_notifier_chain_register()` method.

Beside the notifications that are discussed in this section, there is another type or notifications, named RTNetlink notifications; these notifications are sent with the `rtmsg_ifinfo()` method. :This type of notifications was discussed in Chapter 2, which dealt with Netlink Sockets.

These are the event types supported for networking (Note: the event types mentioned in the following table are defined in `include/linux/netdevice.h`):

We have now covered notification events, a mechanism that enables network devices to get notifications about events such as change of MTU, change of MAC address and more. The next section will discuss shortly the PCI subsystem, describing some of its main data structures.

The PCI Subsystem

Many network interfaces cards are Peripheral Component Interconnect (PCI) devices and should work in conjunction with the Linux PCI subsystem. Not all network interfaces are PCI devices; there are many embedded devices where the network interface is not on a PCI bus; the initialization and handling of these devices is done in a different way, and the following discussion is not relevant for these non-PCI devices. The new PCI devices are PCI Express (PCIe or PCIE) devices; the standard was created in 2004. They have a serial interface instead of a parallel interface, and as a result they have higher maximum system bus throughput. Each PCI device has a read-only configuration space; it is at least 256 bytes. The extended configuration space, available in PCI-X 2.0 and PCI Express buses, is 4096 bytes. You can read the PCI configuration space and the extended PCI configuration space by `lspci` (the `lspci` utility belongs to the `pciutils` package):

- `lspci -xxx`: Shows a hexadecimal dump of the PCI configuration space.
- `lspci -xxxx`: Shows a hexadecimal dump of the extended PCI configuration space.

The Linux PCI API provides three methods for reading the configuration space, for handling 8-, 16-, and 32-bit granularity:

- `static inline int pci_read_config_byte(const struct pci_dev *dev, int where, u8 *val)`
- `static inline int pci_read_config_word(const struct pci_dev *dev, int where, u16 *val)`
- `static inline int pci_read_config_dword(const struct pci_dev *dev, int where, u32 *val)`

There are also three methods for writing the configuration space; likewise, 8-, 16-, and 32-bit granularities are handled:

- `static inline int pci_write_config_byte(const struct pci_dev *dev, int where, u8 val)`
- `static inline int pci_write_config_word(const struct pci_dev *dev, int where, u16 val)`
- `static inline int pci_write_config_dword(const struct pci_dev *dev, int where, u32 val)`

Every PCI manufacturer assigns values to at least the vendor, device, and class fields in the configuration space of the PCI device. A PCI device is identified by the Linux PCI subsystem by a `pci_device_id` object. The `pci_device_id` `struct` is defined in `include/linux/mod_devicetable.h`:

```
struct pci_device_id {
        __u32 vendor, device;           /* Vendor and device ID or PCI_ANY_ID*/
        __u32 subvendor, subdevice;     /* Subsystem ID's or PCI_ANY_ID */
        __u32 class, class_mask;        /* (class,subclass,prog-if) triplet */
        kernel_ulong_t driver_data;     /* Data private to the driver */
};
(include/linux/mod_devicetable.h)
```

The vendor, device, and class fields in `pci_device_id` identify a PCI device; most drivers do not need to specify the class as vendor/device is normally sufficient.

Each PCI device driver declares a `pci_driver` object. Let's take a look at the `pci_driver` structure:

```
struct pci_driver {
    . . .
    const char *name;
    const struct pci_device_id *id_table;   /* must be non-NULL for probe to be called */
    int  (*probe)  (struct pci_dev *dev, const struct pci_device_id *id);   /* New device inserted */
    void (*remove) (struct pci_dev *dev);   /* Device removed (NULL if not a hot-plug capable driver) */
    int  (*suspend) (struct pci_dev *dev, pm_message_t state);      /* Device suspended */
    . . .
    int  (*resume) (struct pci_dev *dev);                   /* Device woken up */
    . . .
};
(include/linux/pci.h)
```

Here are short descriptions of the members of the `pci_driver` structure:

- `name`: Name of the PCI device.
- `id_table`: An array of `pci_device_id` objects which it supports. Initializing `id_table` is done usually with the DEFINE_PCI_DEVICE_TABLE macro.
- `probe`: A method for device initialization.
- `remove`: A method for freeing the device. The `remove()` method usually frees all the resources that were assigned in the `probe()` method.
- `suspend`: A power management callback which puts the device to be in low power state, for devices that support power management.
- `resume`: A power management callback that wakes the device from low power state, for devices that support power management.

A PCI device is represented by `struct pci_dev`. It is a large structure; let's take a look at some of its members (they are self-explanatory):

```
struct pci_dev {
        . . .
        unsigned short  vendor;
        unsigned short  device;
```

```
        unsigned short  subsystem_vendor;
        unsigned short  subsystem_device;
        . . .
        struct pci_driver *driver;      /* which driver has allocated this device */
        . . .
        pci_power_t     current_state;  /* Current operating state. In ACPI-speak,
                                           this is D0-D3, D0 being fully functional,
                                           and D3 being off. */
        struct  device  dev;            /* Generic device interface */

        int             cfg_size;       /* Size of configuration space */

        unsigned int    irq;
};
(include/linux/pci.h)
```

Registering of a PCI network device against the PCI subsystem is done by defining a `pci_driver` object and calling the `pci_register_driver()` macro, which gets as its single argument a `pci_driver` object. In order to initialize the PCI device before it's being used, a driver should call the `pci_enable_device()` method. This method wakes up the device if it was suspended, and allocates the required I/O resources and memory resources. Unregistering the PCI driver is done by the `pci_unregister_driver()` method. Usually the `pci_register_driver()` macro is called in the driver `module_init()` method and the `pci_unregister_driver()` method is called in the driver `module_exit()` method. Each driver should call the `request_irq()` method specifying the IRQ handler when the device is brought up, and call `free_irq()` when the device is brought down.

Allocation and freeing of DMA (Direct Memory Access) memory is usually done with `dma_alloc_coherent()`/ `dma_free_coherent()` when working with uncached memory buffer. With `dma_alloc_coherent()` we don't need to worry about cache coherency, as the mappings of this method are cache-coherent. See for example in `e1000_alloc_ring_dma()`, `drivers/net/ethernet/intel/e1000e/netdev.c`. The Linux DMA API is described in *Documentation/DMA-API.txt*.

■ **Note** Single Root I/O Virtualization (SR-IOV) is a PCI feature that makes one physical device appear as several virtual devices. The SR-IOV specification was created by the PCI SIG. See http://www.pcisig.com/specifications/iov/single_root/. For more information see `Documentation/PCI/pci-iov-howto.txt`.

More information about PCI can be found in the third edition of "Linux Device Drivers" by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, which is available (under Creative Commons License) in this URL: *http://lwn.net/Kernel/LDD3/*.

Wake-On-LAN (WOL)

Wake-On-LAN is a standard that allows a device that had been soft-powered-down to be powered up or awakened by a network packet. Wake-On-LAN is disabled by default. There are some network device drivers which let the sysadmin enable the Wake-On-LAN feature, usually by running from userspace the `ethtool` command. In order to support this, the network device driver should define a `set_wol()` callback in the `ethtool_ops` object. See for example, the `8139cp` driver of RealTek (`net/ethernet/realtek/8139cp.c`). Running `ethtool <networkDeviceName>` shows whether the network device supports Wake-On-LAN. The `ethtool` also lets the sysadmin define which packets should wake the device; for example, `ethtool -s eth1 wol g` will enable Wake-On-LAN for MagicPacket frames (MagicPacket is a standard of AMD). You can use the `ether-wake` utility of the `net-tools` package to send Wake-On-LAN MagicPacket frames.

Teaming Network Device

The virtual teaming network device driver is intended to be a replacement for the bonding network device (`drivers/net/bonding`). The bonding network device provides a link aggregation solution (also known as: "link bundling" or "trunking"). See `Documentation/networking/bonding.txt`. The bonding driver is implemented fully in the kernel, and is known to be very large and prone to problems. The teaming network driver is controlled by userspace, as opposed to the bonding network driver. The userspace daemon is called `teamd` and it communicates with the kernel teaming driver by a library name `libteam`. The libteam library is based on generic netlink sockets (see Chapter 2).

There are four modes for the teaming driver:

- **loadbalance:** Used in Link Aggregation Control Protocol (LACP), which is part of the 802.3ad standard.

  `net/team/team_mode_loadbalance.c`

- **activebackup:** Only one port is active at a given time. This port can transmit and receive SKBs. The other ports are backup ports. A userspace application can specify which port to use as the active port.

  `net/team/team_mode_activebackup.c`

- **broadcast:** All packets are sent by all ports.

  `net/team/team_mode_broadcast.c`

- **roundrobin:** Selection of ports is done by a round robin algorithm. No need for interaction with userspace for this mode.

  `net/team/team_mode_roundrobin.c`

■ **Note** The teaming network driver resides under `drivers/net/team` and is developed by Jiri Pirko.

For more information see `http://libteam.org/`.

`libteam` site: `https://github.com/jpirko/libteam`.

Our brief overview about the teaming driver is over. Many of the readers use PPPoE services when they are surfing the Internet. The following short section covers the PPPoE protocol.

The PPPoE Protocol

PPPoE is a specification for connecting multiple clients to a remote site. PPPoE is typically used by DSL providers to handle IP addresses and authenticate users. The PPPoE protocol provides the ability to use PPP encapsulation for Ethernet packets. The PPPoE protocol is specified in RFC 2516 from 1999, and the PPP protocol is specified in RFC 1661 from 1994. There are two stages in PPPoE:

- PPPoE discovery stage. The discovery is done in a client-server session. The server is called an Access Concentrator, and there can be more than one. These Access Concentrators are often deployed by an Internet Server Provider (ISP). These are the four steps in the Discovery stage:
  - The PPPoE Active Discovery Initiation (PADI). A broadcast packet is sent from a host. The `code` in the PPPoE header is 0x09 (PADI_CODE), and the session id (`sid`) in the PPPoE header must be 0.
  - The PPPoE Active Discovery Offer (PADO). An Access Concentrator replies to a PADI request with a PADO reply. The destination address is the address of the host that sent the PADI. The `code` in the PPPoE header is 0x07 (PADO_CODE). The session id (`sid`) in the PPPoE header must again be 0.
  - PPPoE Active Discovery Request (PADR). A host sends a PADR packet to an Access Concentrator after it receives a PADO reply. The `code` in the PPPoE header is 0x19 (PADR_CODE). The session id (`sid`) in the PPPoE header must again be 0.
  - PPPoE Active Discovery Session-confirmation (PADS). When the Access Concentrator gets a PADR request, it generates a unique session id, and sends a PADS packet as a reply. The `code` in the PPPoE header is 0x65 (PADS_CODE). The session id (`sid`) in the PPPoE header is the session id that it generated. The destination of the packet is the IP address of the host that sent the PADR request.
  - A session is terminated by sending PPPoE Active Discovery Terminate (PADT) packet. The `code` in the PPPoE header is 0xa7 (PADT_CODE). A PADT can be sent either by an Access Concentrator or a host, and it can be sent any time after the session was established. The destination address is a unicast address. The ethertype of the Ethernet header of all the five discovery packets (PADI, PADO, PADR, PADS and PADT) is 0x8863 (ETH_P_PPP_DISC).
- PPPoE Session stage. Once the PPPoE discovery stage completed successfully, packets are sent using PPP encapsulation, which means adding a PPP header of two bytes. Using PPP enables registration and authentication using PPP subprotocols like Password Authentication Protocol (PAP) or Challenge Handshake Authentication Protocol (CHAP), and also PPP subprotocol called the Link Control Protocol (LCP), which is responsible for establishing and testing the data-link connection. The ethertype of the Ethernet header is 0x8864 (ETH_P_PPP_SES).

Every PPPoE packet starts with a 6-byte of PPPoE header, and you must learn about the PPPoE header in order to understand better the PPPoE protocol.

PPPoE Header

I will start by showing the PPPoE header definition in the Linux kernel:

```
struct pppoe_hdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
        __u8 ver : 4;
        __u8 type : 4;
```

```
#elif defined(__BIG_ENDIAN_BITFIELD)
        __u8 type : 4;
        __u8 ver : 4;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
        __u8 code;
        __be16 sid;
        __be16 length;
        struct pppoe_tag tag[0];
} __packed;
(include/uapi/linux/if_pppox.h)
```

The following is a description of the members of the `pppoe_hdr` structure:

- `ver`: The `ver` field is a 4-bit field and it must be set to 0x1 according to section 4 in RFC 2516.
- `type`: The `type` field is a 4-bit field and it must also be set to 0x1 according to section 4 in RFC 2516.
- `code`: The code field is a 8-bit field and it can be one of the constants mentioned earlier: PADI_CODE, PADO_CODE, PADR_CODE, PADS_CODE and PADT_CODE.
- `sid`: Session ID (16-bit).
- `length`: The `length` is a 16-bit field, and it represents the length of the PPPoE payload, without the length of the PPPoE header or the length of the Ethernet header.
- `tag[0]`: The PPPoE payload can contains zero or more tags, in a type-length-value (TLV) format. A tag consists of 3 fields:
  - TAG_TYPE: 16-bit (for example, AC-Name, Service-Name, Generic-Error and more).
  - TAG_LENGTH: 16-bit.
  - TAG_VALUE: variable in length.
- Appendix A of RFC 2516 lists the various TAG_TYPEs and TAG_VALUEs.

Figure 14-9 shows a PPPoE header:

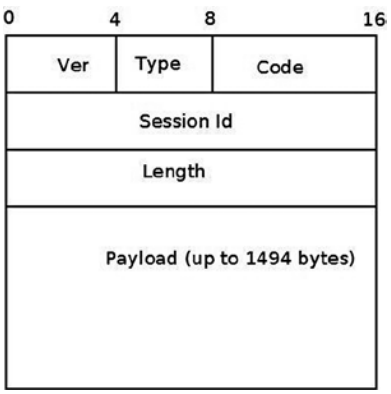

Figure 14-9. *PPPoE header*

PPPoE Initialization

PPPoE Initialization is done by the `pppoe_init()` method, `drivers/net/ppp/pppoe.c`. Two PPPoE protocol handlers are registered, one for PPPoE discovery packets, and one for PPPoE session packets. Let's take a look at the PPPoE protocol handler registration:

```
static struct packet_type pppoes_ptype __read_mostly = {
        .type   = cpu_to_be16(ETH_P_PPP_SES),
        .func   = pppoe_rcv,
};

static struct packet_type pppoed_ptype __read_mostly = {
        .type   = cpu_to_be16(ETH_P_PPP_DISC),
        .func   = pppoe_disc_rcv,
};

static int __init pppoe_init(void)
{
        int err;

        dev_add_pack(&pppoes_ptype);
        dev_add_pack(&pppoed_ptype);
        . . .

        return 0;

}
```

The `dev_add_pack()` method is the generic method for registering protocol handlers, and you encountered in previous chapters. The protocol handlers which are registered by the `pppoe_init()` method are:

- The `pppoe_disc_rcv()` method is the handler for PPPoE discovery packets.
- The `pppoe_rcv()` method is the handler for PPPoE session packets.

The PPPoE module exports an entry to `procfs`, `/proc/net/pppoe`. This entry consists of the session id, the MAC address, and the device of the current PPPoE sessions. Running `cat /proc/net/pppoe` is handled by the `pppoe_seq_show()` method. A notifier chain is registered by the `pppoe_init()` method by calling the `register_netdevice_notifier(&pppoe_notifier)`.

PPPoX Sockets

PPPoX sockets are represented by the `pppox_sock` structure (`include/linux/if_pppox.h`) and are implemented in `net/ppp/pppox.c`. These sockets implement a Generic PPP encapsulation socket family. Apart from PPPoE, they are used also by Layer 2 Tunneling Protocol (L2TP) over PPP. PPPoX sockets are registered by calling `register_pppox_proto(PX_PROTO_OE, &pppoe_proto)` in the `pppoe_init()` method. Let's take a look at the definition of the `pppox_sock` structure:

```
struct pppox_sock {
        /* struct sock must be the first member of pppox_sock */
        struct sock sk;
        struct ppp_channel chan;
        struct pppox_sock       *next;    /* for hash table */
```

```
        union {
                struct pppoe_opt pppoe;
                struct pptp_opt  pptp;
        } proto;
        __be16                  num;
};
(include/linux/if_pppox.h)
```

When the PPPoX socket is used by PPPoE, the `pppoe_opt` of the `proto` union of the `pppox_sock` object is used. The `pppoe_opt` structure includes a member called `pa`, which is an instance of the `pppoe_addr` structure. The `pppoe_addr` structure represents the parameters of the PPPoE session: session id, remote MAC address of the peer, and the name of the network device that is used:

```
struct pppoe_addr {
        sid_t         sid;                    /* Session identifier */
        unsigned char remote[ETH_ALEN];       /* Remote address */
        char          dev[IFNAMSIZ];          /* Local device to use */
};
(include/uapi/linux/if_pppox.h)
```

■ **Note** Access to the `pa` member of the `pppoe_opt` structure which is embedded in the `proto` union is done in most cases in the PPPoE module using the `pppoe_pa` macro:

```
#define pppoe_pa        proto.pppoe.pa

(include/linux/if_pppox.h)
```

Sending and Receiving Packets with PPPoE

Once the discovery stage is completed, the PPP protocol must be used in order to enable traffic between the two peers, as was mentioned earlier. When starting a PPP connection by running, for example, `pppd eth0` (see the example later in this section), the userspace `pppd` daemon creates a PPPoE socket by calling `socket(AF_PPPOX, SOCK_STREAM, PX_PROTO_OE)`; this is done in the `rp-pppoe` plugin of the `pppd` daemon, in the `PPPOEConnectDevice()` method of `pppd/plugins/rp-pppoe/plugin.c`. This `socket()` system call creates a PPPoE socket by the `pppoe_create()` method of the PPPoE kernel module. Releasing the socket after the PPPoE session completed is done by the `pppoe_release()` method of the PPPoE kernel module. Let's take a look at the `pppoe_create()` method:

```
static const struct proto_ops pppoe_ops = {
        .family         = AF_PPPOX,
        .owner          = THIS_MODULE,
        .release        = pppoe_release,
        .bind           = sock_no_bind,
        .connect        = pppoe_connect,
        . . .
        .sendmsg        = pppoe_sendmsg,
        .recvmsg        = pppoe_recvmsg,
        . . .
        .ioctl          = pppox_ioctl,
};
```

```
static int pppoe_create(struct net *net, struct socket *sock)
{
        struct sock *sk;

        sk = sk_alloc(net, PF_PPPOX, GFP_KERNEL, &pppoe_sk_proto);
        if (!sk)
                return -ENOMEM;

        sock_init_data(sock, sk);

        sock->state     = SS_UNCONNECTED;
        sock->ops       = &pppoe_ops;

        sk->sk_backlog_rcv      = pppoe_rcv_core;
        sk->sk_state            = PPPOX_NONE;
        sk->sk_type             = SOCK_STREAM;
        sk->sk_family           = PF_PPPOX;
        sk->sk_protocol         = PX_PROTO_OE;

        return 0;
}
(drivers/net/ppp/pppoe.c)
```

By defining `pppoe_ops` we set callbacks for this socket. So calling from userspace the `connect()` system call on an AF_PPPOX socket will be handled by the `pppoe_connect()` method of the PPPoE module in the kernel. After creating a PPPoE socket, the `PPPOEConnectDevice()` method calls `connect()`. Let's take a look at the `pppoe_connect()` method:

```
static int pppoe_connect(struct socket *sock, struct sockaddr *uservaddr,
                  int sockaddr_len, int flags)
{
        struct sock *sk = sock->sk;
        struct sockaddr_pppox *sp = (struct sockaddr_pppox *)uservaddr;
        struct pppox_sock *po = pppox_sk(sk);
        struct net_device *dev = NULL;
        struct pppoe_net *pn;
        struct net *net = NULL;
        int error;

        lock_sock(sk);

        error = -EINVAL;
        if (sp->sa_protocol != PX_PROTO_OE)
                goto end;

        /* Check for already bound sockets */
        error = -EBUSY;
```

The stage_session() method returns true when the session id is not 0 (as mentioned earlier, the session id is 0 in the discovery stage only). In case the socket is connected and it is in the session stage, the socket is already bound, so we exit:

```
if ((sk->sk_state & PPPOX_CONNECTED) &&
     stage_session(sp->sa_addr.pppoe.sid))
        goto end;
```

Reaching here means that the socket is not connected (it's sk_state is not PPPOX_CONNECTED) and we need to register a PPP channel:

```
. . .
/* Re-bind in session stage only */
if (stage_session(sp->sa_addr.pppoe.sid)) {
        error = -ENODEV;
        net = sock_net(sk);
        dev = dev_get_by_name(net, sp->sa_addr.pppoe.dev);
        if (!dev)
                goto err_put;

        po->pppoe_dev = dev;
        po->pppoe_ifindex = dev->ifindex;
        pn = pppoe_pernet(net);
```

The network device must be up:

```
        if (!(dev->flags & IFF_UP)) {
                goto err_put;
        }

        memcpy(&po->pppoe_pa,
               &sp->sa_addr.pppoe,
               sizeof(struct pppoe_addr));

        write_lock_bh(&pn->hash_lock);
```

The __set_item() method inserts the pppox_sock object, po, into the PPPoE socket hashtable; the hash key is generated according to the session id and the remote peer MAC address by the hash_item() method. The remote peer MAC address is po->pppoe_pa.remote. If there is an entry in the hash table with the same session id and the same remote MAC address and the same ifindex of the network device, the __set_item() method will return an error of –EALREADY:

```
        error = __set_item(pn, po);
        write_unlock_bh(&pn->hash_lock);

        if (error < 0)
                goto err_put;
```

po->chan is a ppp_channel object, see earlier in the pppox_sock structure definition. Before registering it by the ppp_register_net_channel() method, some of its members should be initialized:

```
                po->chan.hdrlen = (sizeof(struct pppoe_hdr) +
                                   dev->hard_header_len);

                po->chan.mtu = dev->mtu - sizeof(struct pppoe_hdr);
                po->chan.private = sk;
                po->chan.ops = &pppoe_chan_ops;

                error = ppp_register_net_channel(dev_net(dev), &po->chan);
                if (error) {
```

The delete_item() method deletes a pppox_sock object from the PPPoE socket hashtable.

```
                        delete_item(pn, po->pppoe_pa.sid,
                                    po->pppoe_pa.remote, po->pppoe_ifindex);
                        goto err_put;
                }
```

Set the socket state to be connected:

```
                sk->sk_state = PPPOX_CONNECTED;
        }

        po->num = sp->sa_addr.pppoe.sid;

end:
        release_sock(sk);
        return error;
err_put:
        if (po->pppoe_dev) {
                dev_put(po->pppoe_dev);
                po->pppoe_dev = NULL;
        }
        goto end;
}
```

By registration of a PPP channel we are allowed to use PPP services. We are able to process PPPoE session packets by calling the generic PPP method, ppp_input(), from the pppoe_rcv_core() method. Transmission of PPPoE session packets is done with the generic ppp_start_xmit() method.

RP-PPPoE is an open source project which provides a PPPoE client and a PPPoE server for Linux: http://www.roaringpenguin.com/products/pppoe. A simple example of running a PPPoE server is:

```
pppoe-server -I  p3p1 -R 192.168.3.101  -L 192.168.3.210 -N 200
```

The options that are used in this example are:

- -I: The interface name (p3p1)
- -L: Set local IP address (192.168.3.210)

- -R: Set the starting remote IP address (192.168.3.101)
- -N: Max number of concurrent PPPoE sessions (200 in this case)

For other options, see `man 8 pppoe-server`.

Clients on the same LAN can create a PPPoE connection to this server by a `pppd` daemon, using the `rp-pppoe` plugin.

Android popularity as a mobile Operating System for smartphones and tablets is growing steadily. I will conclude the book with a short section about Android, discussing briefly the Android development model and showing four examples about Android networking.

Android

In the recent years, the Android operating system proved to be a very reliable and successful mobile OS. The Android operating system is based on a Linux kernel, with changes by Google developers. Android runs on hundreds of types of mobile devices, which are mostly based on the ARM processor. (I should mention that there is a project of porting Android to Intel x86 processors, http://www.android-x86.org/). The first generation of Google TV devices is based on x86 processors by Intel, but the second generation of Google TV devices are based on ARM. Originally Android was developed by "Android Inc.", a company that was founded in California in 2003 by Andy Rubin and others. Google bought this company in 2005. The Open Handset Alliance (OHA), a consortium of over 80 companies, announced Android in 2007. Android is an open source operating system, and its source code is released under the Apache License. Unlike Linux, most of the development is done by Google employees behind closed doors. As opposed to Linux, there is no public mailing list where developers are sending and discussing patches. One can, however, send patches to public Gerrit (see http://source.android.com/source/submit-patches.html). But it is up to Google only to decide whether or not they will be included in the Android tree.

Google developers had contributed a lot to the Linux kernel. You had learned earlier in this chapter that the cgroup subsystem was started by Google developers. I will mention also two Linux kernel networking patches, the Receive Packet Steering (RPS) patch, and the Receive flow steering (RFS) patch by Tom Herbert from Google (see http://lwn.net/Articles/362339/ and http://lwn.net/Articles/382428/), which were integrated into kernel 2.6.35. When working with multicore platforms, RPS and RFS let you steer packets according to the hash of the payload to a specific CPU. And there are a lot of other examples of contributions from Google to the Linux kernel, and it seems that also in the future you will encounter many important contributions to the Linux kernel from Google. One can find a lot of code from Android kernel in the staging tree of the Linux kernel. However, it is difficult to say whether the Android kernel will be merged fully into the Linux kernel; probably a very large part of it will find its way into the Linux kernel. For more information about Mainlining Android see this wiki: http://elinux.org/Android_Mainlining_Project. In the past there were many obstacles in the way, as Google implemented unique mechanisms, like wakelocks, alternative power management, its own IPC (called Binder), which is based on a Lightweight Remote Procedure Call (RPC), Android shared memory driver (Ashmem), Low Memory Killer and more. In fact, the Kernel community rejected the Google power management wakelocks patches in 2010. But since then, some of these features were merged and the situation changed. (See "Autosleep and Wake Locks," https://lwn.net/Articles/479841/, and "The LPC Android microconference", https://lwn.net/Articles/570406/). Linaro (www.linaro.org/) is a non-profit organization that was established in 2010 by leading big companies such as ARM, Freescale, IBM, Samsung, ST-Ericsson, and Texas Instruments (TI). Its engineering teams develop Linux ARM kernel and also optimizations for GCC toolchain. Linaro teams are doing an amazing job of coordinating and pushing/tweaking changes upstream. Delving into the details of Android kernel implementation and mainlining is beyond the scope of this book.

Android Networking

The main networking issue with Android is, however, not due to Linux kernel but to Android userspace. Android heavily relies on HAL even for networking, as well as for system framework. Originally (i.e., up to 4.2), there's no Ethernet support at all at framework level. If drivers are compiled in the kernel, the TCP/IP stack still allows basic Ethernet connectivity for Android Debug Bridge (ADB) debugging, but that's all. Starting with 4.0, Android-x86 project

fork added an early implementation (badly designed but somehow working) of Ethernet at framework level. Starting with 4.2, official upstream sources support Ethernet, but there is no way to actually configure it (it detects Ethernet plug in/out, and if a DHCP server is there, it provides an IP address to the interface). Applications can actually make use of this interface through framework, but mostly no one does this. If you require real Ethernet support (i.e., being able to configure your interface, static/DHCP configure it, set proxy, ensure that all apps are using the interface, then a lot of hacks are still required (see www.slideshare.net/gxben/abs-2013-dive-into-android-networking-adding-ethernet-connectivity). In all cases, only one interface is being supported at a time (`eth0` only, even if you have `eth0` and `eth1`, so don't expect to act as a router of any kind). I will show here four short examples of how Android networking differs from Linux kernel networking:

- Security privileges and networking: Android added a security feature (named "paranoid network") to the Linux kernel, which restricts access to some networking features, depending on the group of the calling process. As opposed to the standard Linux kernel, where any application can open a socket and transmit/receive with it, in Android access to network resources is filtered by GID (group ID). The part of network security will be probably very difficult to merge into the mainline kernel, as it includes many features that are unique to Android. For more information about Android network security, see http://elinux.org/Android_Security#Paranoid_network-ing.
- Bluetooth: Bluedroid is a Bluetooth stack based on code that was developed by Broadcom. It replaced the BlueZ based stack in Android 4.2. Support for Bluetooth Low Energy (BLE, or Bluetooth LE) devices, also known as Bluetooth Smart and Smart Ready devices, was introduced in Android 4.3 (API Level 18), July 2013. Prior to this, Android Open Source Project (AOSP) did not have support for BLE devices, but there were some vendors who provided an API to BLE.
- Netfilter: There is an interesting project from Google that provides better network statistics on Android. This is implemented by `xt_qtaguid`, a netfilter module, which enables userspace applications to tag their sockets. This project required some changes in the Linux kernel netfilter subsystem. Patches of these changes were also sent to the Linux Kernel Mailing List (LKML); see http://lwn.net/Articles/517358/. For details, see "Android netfilter changes" http://www.linuxplumbersconf.org/2013/ocw/sessions/1491.
- NFC: As was described in the Near Field Communication (NFC) section earlier in this chapter, the Android NFC architecture is a userspace NFC stack: the implementation is done in userspace through the HAL which is supplied by Broadcom or by Android OEMs.

Android internals: Resources

Although there are many resources about developing applications for Android (whether in books, mailing list, forums, courses, etc.), there are very few resources about the internals of Android. For those readers who are interested to learn more, I suggest these resources:

- The book *Embedded Android: Porting, Extending, and Customizing,* by Karim Yaghmour (O'Reilly Media, 2013)
- Slides: Android System Development by Maxime Ripard, Alexandre Belloni (over 400 slides); http://free-electrons.com/doc/training/android/.
- Slides: Android Platform Anatomy by Benjamin Zores (59 slides); http://www.slideshare.net/gxben/droidcon-2013-france-android-platform-anatomy.
- Slides: Jelly Bean Device Porting by Benjamin Zores (127 slides); http://www.slideshare.net/gxben/as-2013-jelly-bean-device-porting-walkthrough.

- Website: http://developer.android.com/index.html.
- Android platform internals forum - archives: http://news.gmane.org/gmane.comp.handhelds.android.platform
- Once a year, an Android Builders Summit (ABS) is held. The first ABS was held in 2011 in San Francisco. It is recommended to read slides, watch videos, or attend.
- XDA Developers Conference: http://xda-devcon.com/; Slides and videos in http://xda-devcon.com/presentations/
- Slides: Android Internals, Marko Gargenta: http://www.scandevconf.se/db/Marakana-Android-Internals.pdf

---

■ **Note** Android git repositories are available in https://android.googlesource.com/

Note that Android uses a special tool based on `python` called `repo` for management of hundreds of `git` repositories, which makes working with `git` easier.

Summary

I have dealt in this chapter with namespaces in Linux, focusing on network namespaces. I also described the cgroups subsystem and its implementation; furthermore, I described its two network modules, `net_prio` and `cls_cgroup`. The Linux Bluetooth subsystem and its implementation, the IEEE 802.15.4 Linux subsystem and 6LoWPAN, and the NFC subsystem were all covered. The optimization achieved by Low Latency Sockets Poll was also discussed in this chapter, along with the Notification Chains mechanism, which is widely used in the kernel networking stack (and you will encounter it when browsing the source code). Another topic that was briefly discussed was the PCI subsystem, in order to give some background about PCI devices, as many network devices are PCI devices. The chapter was concluded with three short sections about the network teaming driver (which is intended to replace the bonding driver), the PPPoE implementation, and Android.

Although we've come to the end of the book, there is much more to learn about Linux Kernel networking, as it is a vast ocean of details, and it is progressing dynamically and at such a fast pace. New features and new patches are added constantly. I hope you enjoyed the book and that you learned a thing or two!

Quick Reference

I will conclude with a list of methods and macros that were mentioned in this chapter.

Methods

The following list contains the prototypes and descriptions of several methods covered in this chapter.

void switch_task_namespaces(struct task_struct *p, struct nsproxy *new);

This method assigns the specified `nsproxy` object to the specified process descriptor (`task_struct` object).

struct nsproxy *create_nsproxy(void);

This method allocates an `nsproxy` object and initializes its reference counter to 1.

void free_nsproxy(struct nsproxy *ns);

This method released the resources of the specified `nsproxy` object.

struct net *dev_net(const struct net_device *dev);

This method returns the network namespace object (`nd_net`) associated with the specified network device.

void dev_net_set(struct net_device *dev, struct net *net);

This method associates the specified network namespace to the specified network device by setting the `nd_net` member of the `net_device` object.

void sock_net_set(struct sock *sk, struct net *net);

This method associates the specified network namespace to the specified `sock` object.

struct net *sock_net(const struct sock *sk);

This method returns the network namespace object (`sk_net`) associated with the specified sock object.

int net_eq(const struct net *net1, const struct net *net2);

This method returns 1 if the first specified network namespace pointer equals the second specified network namespace pointer and 0 otherwise.

struct net *net_alloc(void);

This method allocates a network namespace. It is invoked from the `copy_net_ns()` method.

struct net *copy_net_ns(unsigned long flags, struct user_namespace *user_ns, struct net *old_net);

This method creates a new network namespace if the CLONE_NEWNET flag is set in its first parameter, `flags`. It creates the new network namespace by first calling the `net_alloc()` method to allocate it, then it initializes it by calling the `setup_net()` method, and finally adds it to the global list of all namespaces, `net_namespace_list`. In case the CLONE_NEWNET flag is set in its first parameter, `flags`, there is no need to create a new namespace and the specified old network namespace, `old_net`, is returned. Note that this description of the `copy_net_ns()` method refers to the case when CONFIG_NET_NS is set. When CONFIG_NET_NS is not set, there is a second implementation of `copy_net_ns()`, which the only thing it does is first verify that CLONE_NEWNET is set in the specified flags, and in case it is, returns the specified old network namespace (`old_net`); see `include/net/net_namespace.h`.

int setup_net(struct net *net, struct user_namespace *user_ns);

This method initializes the specified network namespace object. It assigns the network namespace `user_ns` member to be the specified `user_ns`, it initializes the reference counter (`count`) of the specified network namespace to be 1, and performs more initializations. It is invoked from the `copy_net_ns()` method and from the `net_ns_init()` method.

int proc_alloc_inum(unsigned int *inum);

This method allocates a proc inode and sets `*inum` to be the generated proc inode number (an integer between 0xf0000000 and 0xffffffff). It returns 0 on success.

struct nsproxy *task_nsproxy(struct task_struct *tsk);

This method returns the `nsproxy` object which is attached to the specified process descriptor (`tsk`).

struct new_utsname *utsname(void);

This method returns the `new_utsname` object which is associated with the process which currently runs (`current`).

struct uts_namespace *clone_uts_ns(struct user_namespace *user_ns, struct uts_namespace *old_ns);

This method creates a new UTS namespace object by calling the `create_uts_ns()` method, and copies the `new_utsname` object of the specified `old_ns` UTS namespace into the `new_utsname` of the newly created UTS namespace.

struct uts_namespace *copy_utsname(unsigned long flags, struct user_namespace *user_ns, struct uts_namespace *old_ns);

This method creates a new UTS namespace if the CLONE_NEWUTS flag is set in its first parameter, `flags`. It creates the new UTS namespace by calling the `clone_uts_ns()` method, and returns the newly created UTS namespace. In case the CLONE_NEWUTS flag is set in its first parameter, there is no need to create a new namespace and the specified old UTS namespace (`old_ns`) is returned.

struct net *sock_net(const struct sock *sk);

This method returns the network namespace object (`sk_net`) associated with the specified sock object.

void sock_net_set(struct sock *sk, struct net *net);

This method assigns the specified network namespace to the specified `sock` object.

int dev_change_net_namespace(struct net_device *dev, struct net *net, const char *pat);

This method changes the network namespace of the specified network device to be the specified network namespace. It returns 0 on success or –`errno` on failure. Callers must hold the `rtnl` semaphore. If the NETIF_F_NETNS_LOCAL flag is set in the features of the network device, an error of –EINVAL is returned.

void put_net(struct net *net);

This method decrements the reference counter of the specified network namespace. In case it reaches zero, it calls the `__put_net()` method to free its resources.

struct net *get_net(struct net *net);

This method returns the specified network namespace object after incrementing its reference counter.

void get_nsproxy(struct nsproxy *ns);

This method increments the reference counter of the specified `nsproxy` object.

struct net *get_net_ns_by_pid(pid_t pid);

This method gets a process id (PID) as an argument, and returns the network namespace object to which this process is attached.

struct net *get_net_ns_by_fd(int fd);

This method gets a file descriptor as an argument, and returns the network namespace associated with the inode that corresponds to the specified file descriptor.

struct pid_namespace *ns_of_pid(struct pid *pid);

This method returns the PID namespace in which the specified `pid` was created.

void put_nsproxy(struct nsproxy *ns);

This method decrements the reference counter of the specified `nsproxy` object; in case it reaches 0, the specified `nsproxy` is freed by calling the `free_nsproxy()` method.

int register_pernet_device(struct pernet_operations *ops);

This method registers a network namespace device.

void unregister_pernet_device(struct pernet_operations *ops);

This method unregisters a network namespace device.

int register_pernet_subsys(struct pernet_operations *ops);

This method registers a network namespace subsystem.

void unregister_pernet_subsys(struct pernet_operations *ops);

This method unregisters a network namespace subsystem.

static int register_vlan_device(struct net_device *real_dev, u16 vlan_id);

This method registers a VLAN device associated with the specified physical device (`real_dev`).

void cgroup_release_agent(struct work_struct *work);

This method is called when a cgroup is released. It creates a userspace process by invoking the `call_usermodehelper()` method.

int call_usermodehelper(char * path, char ** argv, char ** envp, int wait);

This method prepares and starts a userspace application.

int bacmp(bdaddr_t *ba1, bdaddr_t *ba2);

This method compares two Bluetooth addresses. It returns 0 if they are equal.

void bacpy(bdaddr_t *dst, bdaddr_t *src);

This method copies the specified source Bluetooth address (`src`) to the specified destination Bluetooth address (`dst`).

int hci_send_frame(struct sk_buff *skb);

This method is the main Bluetooth method for transmitting SKBs (commands and data).

int hci_register_dev(struct hci_dev *hdev);

This method registers the specified HCI device. It is invoked from Bluetooth device drivers. If the `open()` or `close()` callbacks of the specified `hci_dev` object are not defined, the method will fail and return –EINVAL. This method sets the HCI_SETUP flag in the `dev_flags` member of the specified HCI device; it also creates a `sysfs` entry for the device.

void hci_unregister_dev(struct hci_dev *hdev);

This method unregisters the specified HCI device. It is invoked from Bluetooth device drivers. It sets the HCI_UNREGISTER flag in the `dev_flags` member of the specified HCI device; it also removes the `sysfs` entry of the device.

void hci_event_packet(struct hci_dev *hdev, struct sk_buff *skb);

This method handles events that are received from the HCI layer by the `hci_rx_work()` method.

int lowpan_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 6LoWPAN packets. 6LoWPAN packets have an `ethertype` of 0x00F6.

void pci_unregister_driver(struct pci_driver *dev);

This method unregisters a PCI driver. It is usually called in the network driver `module_exit()` method.

int pci_enable_device(struct pci_dev *dev);

This method initializes the PCI device before it is used by driver.

int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev);

This method registers the specified `handler` as the interrupt service routine for the specified `irq`.

void free_irq(unsigned int irq, void *dev_id);

This method frees an interrupt which was allocated with the `request_irq()` method.

int nfc_init(void);

This method performs initialization of the NFC subsystem by registering the generic netlink NFC family, initializing NFC Raw sockets and NFC LLCP sockets, and initializing the AF_NFC protocol.

int nfc_register_device(struct nfc_dev *dev);

This method registers an NFC device (an `nfc_dev` object) against the NFC core.

int nfc_hci_register_device(struct nfc_hci_dev *hdev);

This method registers an NFC HCI device (an `nfc_hci_dev` object) against the NFC HCI layer.

int nci_register_device(struct nci_dev *ndev);

This method registers an NFC NCI device (an `nci_dev` object) against the NFC NCI layer.

static int __init pppoe_init(void);

This method initializes the PPPoE layer (PPPoE protocol handlers, the sockets used by PPPoE, the network notification handler, the PPPoE `procfs` entry, and more).

struct pppoe_hdr *pppoe_hdr(const struct sk_buff *skb);

This method returns the PPPoE header associated with the specified `skb`.

static int pppoe_create(struct net *net, struct socket *sock);

This method creates a PPPoE socket. Return 0 on success or –ENOMEM if allocation of a socket by the `sk_alloc()` method failed.

int __set_item(struct pppoe_net *pn, struct pppox_sock *po);

This method inserts the specified `pppox_sock` object into the PPPoE socket hashtable. The hash key is calculated according to the session id and the remote peer MAC address by the `hash_item()` method.

void delete_item(struct pppoe_net *pn, __be16 sid, char *addr, int ifindex);

This method removes the PPPoE socket hashtable entry which has the specified session id, the specified MAC address, and the specified network interface index (`ifindex`).

bool stage_session(__be16 sid);

This method returns `true` when the specified session id is not 0.

int notifier_chain_register(struct notifier_block **nl, struct notifier_block *n);

This method registers the specified `notifier_block` object (`n`) to the specified notifier chain (`nl`). Note that this method is not used directly, there are several wrappers around it.

int notifier_chain_unregister(struct notifier_block **nl, struct notifier_block *n);

This method unregistered the specified `notifier_block` object (`n`) from the specified notifier chain (`nl`). Note that also this method is not used directly, there are several wrappers around it.

int register_netdevice_notifier(struct notifier_block *nb);

This method registers the specified `notifier_block` object to `netdev_chain` by calling the `raw_notifier_chain_register()` method.

int unregister_netdevice_notifier(struct notifier_block *nb);

This method unregisters the specified `notifier_block` object from `netdev_chain` by calling the `raw_notifier_chain_unregister()` method.

int register_inet6addr_notifier(struct notifier_block *nb);

This method registers the specified `notifier_block` object to `inet6addr_chain` by calling the `atomic_notifier_chain_register()` method.

int unregister_inet6addr_notifier(struct notifier_block *nb);

This method unregisters the specified `notifier_block` object from `inet6addr_chain` by calling the `atomic_notifier_chain_unregister()` method.

int register_netevent_notifier(struct notifier_block *nb);

This method registers the specified `notifier_block` object to `netevent_notif_chain` by calling the `atomic_notifier_chain_register()` method.

int unregister_netevent_notifier(struct notifier_block *nb);

This method unregisters the specified `notifier_block` object from `netevent_notif_chain` by calling the `atomic_notifier_chain_unregister()` method.

int __kprobes notifier_call_chain(struct notifier_block **nl, unsigned long val, void *v, int nr_to_call, int *nr_calls);

This method is for generating notification events. Note that also this method is not used directly, there are several wrappers around it.

int call_netdevice_notifiers(unsigned long val, struct net_device *dev);

This method is for generating notification events on the `netdev_chain`, by calling the `raw_notifier_call_chain()` method.

int blocking_notifier_call_chain(struct blocking_notifier_head *nh, unsigned long val, void *v);

This method is for generating notification events; eventually, after using locking mechanism, it invokes the `notifier_call_chain()` method.

int __atomic_notifier_call_chain(struct atomic_notifier_head *nh,unsigned long val, void *v, int nr_to_call, int *nr_calls);

This method is for generating notification events. Eventually, after using locking mechanism, it invokes the `notifier_call_chain()` method.

Macros

Here you'll find a description of the macro that was covered in this chapter.

pci_register_driver()

This macro registers a PCI driver in the PCI subsystem. It gets a `pci_driver` object as a parameter. It is usually called in the network driver `module_init()` method.

APPENDIX A

Linux API

In this appendix I cover the two most fundamental data structures in the Linux Kernel Networking stack: the `sk_buff` and the `net_device`. This is reference material that can help when reading the rest of this book, as you will probably encounter these two structures in almost every chapter. Becoming familiar with and learning about these two data structures is essential for understanding the Linux Kernel Networking stack. Subsequently, there is a section about remote DMA (RDMA), which is further reference material for Chapter 13. It describes in detail the main methods and the main data structures that are used by RDMA. This appendix is a good place to always return to, especially when looking for definitions of the basic terms.

The sk_buff Structure

The `sk_buff` structure represents a packet. SKB stands for *socket buffer*. A packet can be generated by a local socket in the local machine, which was created by a userspace application; the packet can be sent outside or to another socket in the same machine. A packet can also be created by a kernel socket; and you can receive a physical frame from a network device (Layer 2) and attach it to an `sk_buff` and pass it on to Layer 3. When the packet destination is your local machine, it will continue to Layer 4. If the packet is not for your machine, it will be forwarded according to your routing tables rules, if your machine supports forwarding. If the packet is damaged for any reason, it will be dropped. The `sk_buff` is a very large structure; I mention most of its members in this section. The `sk_buff` structure is defined in `include/linux/skbuff.h`. Here is a description of most of its members:

- `ktime_t tstamp`

  Timestamp of the arrival of the packet. Timestamps are stored in the SKB as offsets to a base timestamp. Note: do not confuse `tstamp` of the SKB with hardware timestamping, which is implemented with the `hwtstamps` of `skb_shared_info`. I describe the `skb_shared_info` object later in this appenidx.

  Helper methods:

  - `skb_get_ktime(const struct sk_buff *skb)`: Returns the `tstamp` of the specified `skb`.
  - `skb_get_timestamp(const struct sk_buff *skb, struct timeval *stamp)`: Converts the offset back to a `struct timeval`.
  - `net_timestamp_set(struct sk_buff *skb)`: Sets the timestamp for the specified `skb`. The timestamp calculation is done with the `ktime_get_real()` method, which returns the time in `ktime_t` format.
  - `net_enable_timestamp()`: This method should be called to enable SKB timestamping.
  - `net_disable_timestamp()`: This method should be called to disable SKB timestamping.

- `struct sock *sk`

The socket that owns the SKB, for local generated traffic and for traffic that is destined for the local host. For packets that are being forwarded, `sk` is NULL. Usually when talking about sockets you deal with sockets which are created by calling the `socket()` system call from userspace. It should be mentioned that there are also kernel sockets, which are created by calling the `sock_create_kern()` method. See for example in `vxlan_init_net()` in the VXLAN driver, `drivers/net/vxlan.c`.

Helper method:

- `skb_orphan(struct sk_buff *skb)`: If the specified `skb` has a destructor, call this destructor; set the sock object (`sk`) of the specified `skb` to NULL, and set the destructor of the specified `skb` to NULL.

- `struct net_device *dev`

  The `dev` member is a `net_device` object which represents the network interface device associated to the SKB; you will sometimes encounter the term NIC (Network Interface Card) for such a network device. It can be the network device on which the packet arrives, or the network device on which the packet will be sent. The `net_device` structure will be discussed in depth in the next section.

- `char cb[48]`

  This is the control buffer. It is free to use by any layer. This is an opaque area used to store private information. For example, the TCP protocol uses it for the TCP control buffer:

  ```
  #define TCP_SKB_CB(__skb) ((struct tcp_skb_cb *)&((__skb)->cb[0]))
  (include/net/tcp.h)
  ```

  The Bluetooth protocol also uses the control block:

  ```
  #define bt_cb(skb) ((struct bt_skb_cb *)((skb)->cb))
      (include/net/bluetooth/bluetooth.h)
  ```

- `unsigned long _skb_refdst`

  The destination entry (`dst_entry`) address. The `dst_entry struct` represents the routing entry for a given destination. For each packet, incoming or outgoing, you perform a lookup in the routing tables. Sometimes this lookup is called FIB lookup. The result of this lookup determines how you should handle this packet; for example, whether it should be forwarded, and if so, on which interface it should be transmitted; or should it be thrown, should an ICMP error message be sent, and so on. The `dst_entry` object has a reference counter (the `__refcnt` field). There are cases when you use this reference count, and there are cases when you do not use it. The `dst_entry` object and the lookup in the FIB is discussed in more detail in Chapter 4.

  Helper methods:

  - `skb_dst_set(struct sk_buff *skb, struct dst_entry *dst)`: Sets the `skb dst`, assuming a reference was taken on `dst` and should be released by the `dst_release()` method (which is invoked by the `skb_dst_drop()` method).

- `skb_dst_set_noref(struct sk_buff *skb, struct dst_entry *dst)`: Sets the `skb` `dst`, assuming a reference was not taken on `dst`. In this case, the `skb_dst_drop()` method will not call the `dst_release()` method for the `dst`.

■ **Note** The SKB might have a `dst_entry` pointer attached to it; it can be reference counted or not. The low order bit of `_skb_refdst` is set if the reference counter was not taken.

- `struct sec_path *sp`

  The security path pointer. It includes an array of IPsec XFRM transformations states (`xfrm_state` objects). IPsec (IP Security) is a Layer 3 protocol which is used mostly in VPNs. It is mandatory in IPv6 and optional in IPv4. Linux, like many other operating systems, implements IPsec both for IPv4 and IPv6. The `sec_path` structure is defined in `include/net/xfrm.h`. See more in Chapter 10, which deals with the IPsec subsystem.

  Helper method:

  - `struct sec_path *skb_sec_path(struct sk_buff *skb)`: Returns the `sec_path` object (`sp`) associated with the specified `skb`.

- `unsigned int len`

  The total number of packet bytes.

- `unsigned int data_len`

  The data length. This field is used only when the packet has nonlinear data (paged data).

  Helper method:

  - `skb_is_nonlinear(const struct sk_buff *skb)`: Returns `true` when the `data_len` of the specified `skb` is larger than 0.

- `__u16 mac_len`

  The length of the MAC (Layer 2) header.

- `__wsum csum`

  The checksum.

- `__u32 priority`

  The queuing priority of the packet. In the Tx path, the priority of the SKB is set according to the socket priority (the `sk_priority` field of the socket). The socket priority in turn can be set by calling the `setsockopt()` system call with the SO_PRIORITY socket option. Using the `net_prio cgroup` kernel module, you can define a rule which will set the priority for the SKB; see in the description of the `sk_buff netprio_map` field, later in this section, and also in `Documentation/cgroup/netprio.txt`. For forwarded packets, the priority is set according to TOS (Type Of Service) field in the IP header. There is a table named `ip_tos2prio` which consists of 16 elements. The mapping from TOS to priority is done by the `rt_tos2priority()` method, according to the TOS field of the IP header; see the `ip_forward()` method in `net/ipv4/ip_forward.c` and the `ip_tos2prio` definition in `include/net/route.h`.

- `__u8 local_df:1`

  Allow local fragmentation flag. If the value of the `pmtudisc` field of the socket which sends the packet is IP_PMTUDISC_DONT or IP_PMTUDISC_WANT, `local_df` is set to 1; if the value of the `pmtudisc` field of the socket is IP_PMTUDISC_DO or IP_PMTUDISC_PROBE, `local_df` is set to 0. See the implementation of the `__ip_make_skb()` method in `net/ipv4/ip_output.c`. Only when the packet `local_df` is 0 do you set the IP header don't fragment flag, IP_DF; see the `ip_queue_xmit()` method in `net/ipv4/ip_output.c`:

```
. . .
 if (ip_dont_fragment(sk, &rt->dst) && !skb->local_df)
        iph->frag_off = htons(IP_DF);
    else
      iph->frag_off = 0;
. . .
```

  The `frag_off` field in the IP header is a 16-bit field, which represents the offset and the flags of the fragment. The 13 leftmost (MSB) bits are the offset (the offset unit is 8-bytes) and the 3 rightmost (LSB) bits are the flags. The flags can be IP_MF (there are more fragments), IP_DF (do not fragment), IP_CE (for congestion), or IP_OFFSET (offset part).

  The reason behind this is that there are cases when you do not want to allow IP fragmentation. For example, in Path MTU Discovery (PMTUD), you set the DF (don't fragment) flag of the IP header. Thus, you don't fragment the outgoing packets. Any network device along the path whose MTU is smaller than the packet will drop it and send back an ICMP packet ("Fragmentation Needed"). Getting these ICMP "Fragmentation Needed" packets is required in order to determine the Path MTU. See more in Chapter 3. From userspace, setting IP_PMTUDISC_DO is done, for example, thus (the following code snippet is taken from the source code of the `tracepath` utility from the `iputils` package; the `tracepath` utility finds the path MTU):

```
. . .
int on = IP_PMTUDISC_DO;
setsockopt(fd, SOL_IP, IP_MTU_DISCOVER, &on, sizeof(on));
. . .
```

- `__u8 cloned:1`

  When the packet is cloned with the `__skb_clone()` method, this field is set to 1 in both the cloned packet and the primary packet. Cloning SKB means creating a private copy of the `sk_buff struct`; the data block is shared between the clone and the primary SKB.

- `__u8 ip_summed:2`

  Indicator of IP (Layer 3) checksum; can be one of these values:

  - CHECKSUM_NONE: When the device driver does not support hardware checksumming, it sets the `ip_summed` field to be CHECKSUM_NONE. This is an indication that checksumming should be done in software.
  - CHECKSUM_UNNECESSARY: No need for any checksumming.

  - CHECKSUM_COMPLETE: Calculation of the checksum was completed by the hardware, for incoming packets.
  - CHECKSUM_PARTIAL: A partial checksum was computed for outgoing packets; the hardware should complete the checksum calculation. CHECKSUM_COMPLETE and CHECKSUM_PARTIAL replace the CHECKSUM_HW flag, which is now deprecated.

- `__u8 nohdr:1`

  Payload reference only, must not modify header. There are cases when the owner of the SKB no longer needs to access the header at all. In such cases, you can call the `skb_header_release()` method, which sets the `nohdr` field of the SKB; this indicates that the header of this SKB should not be modified.

- `__u8 nfctinfo:3`

  Connection Tracking info. Connection Tracking allows the kernel to keep track of all logical network connections or sessions. NAT relies on Connection Tracking information for its translations. The value of the `nfctinfo` field corresponds to the `ip_conntrack_info enum` values. So, for example, when a new connection is starting to be tracked, the value of `nfctinfo` is IP_CT_NEW. When the connection is established, the value of `nfctinfo` is IP_CT_ESTABLISHED. The value of `nfctinfo` can change to IP_CT_RELATED when the packet is related to an existing connection—for example, when the traffic is part of some FTP session or SIP session, and so on. For a full list of `ip_conntrack_info enum` values see `include/uapi/linux/netfilter/nf_conntrack_common.h`. The `nfctinfo` field of the SKB is set in the `resolve_normal_ct()` method, `net/netfilter/nf_conntrack_core.c`. This method performs a Connection Tracking lookup, and if there is a miss, it creates a new Connection Tracking entry. Connection Tracking is discussed in depth in Chapter 9, which deals with the netfilter subsystem.

- `__u8 pkt_type:3`

  For Ethernet, the packet type depends on the destination MAC address in the ethernet header, and is determined by the `eth_type_trans()` method:

  - PACKET_BROADCAST for broadcast
  - PACKET_MULTICAST for multicast
  - PACKET_HOST if the destination MAC address is the MAC address of the device which was passed as a parameter
  - PACKET_OTHERHOST if these conditions are not met

  See the definition of the packet types in `include/uapi/linux/if_packet.h`.

- `__u8 ipvs_property:1`

  This flag indicates whether the SKB is owned by `ipvs` (IP Virtual Server), which is a kernel-based transport layer load-balancing solution. This field is set to 1 in the transmit methods of `ipvs` (`net/netfilter/ipvs/ip_vs_xmit.c`).

- `__u8 peeked:1`

  This packet has been already seen, so stats have been done for it—so don't do them again.

- `__u8 nf_trace:1`

  The netfilter packet trace flag. This flag is set by the packet flow tracing the netfilter module, xt_TRACE module, which is used to mark packets for tracing (`net/netfilter/xt_TRACE.c`).

  Helper method:

  - `nf_reset_trace(struct sk_buff *skb)`: Sets the `nf_trace` of the specified `skb` to 0.

- `__be16 protocol`

  The protocol field is initialized in the Rx path by the `eth_type_trans()` method to be ETH_P_IP when working with Ethernet and IP.

- `void (*destructor)(struct sk_buff *skb)`

  A callback that is invoked when freeing the SKB by calling the `kfree_skb()` method.

- `struct nf_conntrack *nfct`

  The associated Connection Tracking object, if it exists. The `nfct` field, like the `nfctinfo` field, is set in the `resolve_normal_ct()` method. The Connection Tracking layer is discussed in depth in Chapter 9, which deals with the netfilter subsystem.

- `int skb_iif`

  The `ifindex` of the network device on which the packet arrived.

- `__u32 rxhash`

  The `rxhash` of the SKB is calculated in the receive path, according to the source and destination address of the IP header and the ports from the transport header. A value of zero indicates that the hash is not valid. The `rxhash` is used to ensure that packets with the same flow will be handled by the same CPU when working with Symmetrical Multiprocessing (SMP). This decreases the number of cache misses and improves network performance. The `rxhash` is part of the Receive Packet Steering (RPS) feature, which was contributed by Google developers (Tom Herbert and others). The RPS feature gives performance improvement in SMP environments. See more in `Documentation/networking/scaling.txt`.

- `__be16 vlan_proto`

  The VLAN protocol used—usually it is the `802.1q` protocol. Recently support for the `802.1ad` protocol (also known as Stacked VLAN) was added.

  The following is an example of creating `802.1q` and `802.1ad` VLAN devices in userspace using the `ip` command of the `iproute2` package:

```
ip link add link eth0 eth0.1000 type vlan proto 802.1ad id 1000
ip link add link eth0.1000 eth0.1000.1000 type vlan proto 802.1q id 100
```

  Note: this feature is supported in kernel 3.10 and higher.

- `__u16 vlan_tci`

  The VLAN tag control information (2 bytes), composed of ID and priority.

Helper method:

- `vlan_tx_tag_present(__skb)`: This macro checks whether the VLAN_TAG_PRESENT flag is set in the `vlan_tci` field of the specified `__skb`.

- `__u16 queue_mapping`

  Queue mapping for multiqueue devices.

  Helper methods:

  - `skb_set_queue_mapping (struct sk_buff *skb, u16 queue_mapping)`: Sets the specified `queue_mapping` for the specified `skb`.
  - `skb_get_queue_mapping(const struct sk_buff *skb)`: Returns the `queue_mapping` of the specified `skb`.

- `__u8 pfmemalloc`

  Allocate the SKB from PFMEMALLOC reserves.

  Helper method:

  - `skb_pfmemalloc()`: Returns `true` if the SKB was allocated from PFMEMALLOC reserves.

- `__u8 ooo_okay:1`

  The `ooo_okay` flag is set to avoid ooo (out of order) packets.

- `__u8 l4_rxhash:1`

  A flag that is set when a canonical 4-tuple hash over transport ports is used.

  See the `__skb_get_rxhash()` method in `net/core/flow_dissector.c`.

- `__u8 no_fcs:1`

  A flag that is set when you request the NIC to treat the last 4 bytes as Ethernet Frame Check Sequence (FCS).

- `__u8 encapsulation:1`

  The encapsulation field denotes that the SKB is used for encapsulation. It is used, for example, in the VXLAN driver. VXLAN is a standard protocol to transfer Layer 2 Ethernet packets over a UDP kernel socket. It can be used as a solution when there are firewalls that block tunnels and allow, for example, only TCP or UDP traffic. The VXLAN driver uses UDP encapsulation and sets the SKB encapsulation to 1 in the `vxlan_init_net()` method. Also the `ip_gre` module and the `ipip` tunnel module use encapsulation and set the SKB encapsulation to 1.

- `__u32 secmark`

  Security mark field. The `secmark` field is set by an `iptables` SECMARK target, which labels packets with any valid security context. For example:

  ```
  iptables -t mangle -A INPUT -p tcp --dport 80 -j SECMARK --selctx
  system_u:object_r:httpd_packet_t:s0
  iptables -t mangle -A OUTPUT -p tcp --sport 80 -j SECMARK --selctx
  system_u:object_r:httpd_packet_t:s0
  ```

  In the preceding rule, you are statically labeling packets arriving at and leaving from port 80 as `httpd_packet_t`. See: `netfilter/xt_SECMARK.c`.

Helper methods:

- `void skb_copy_secmark(struct sk_buff *to, const struct sk_buff *from)`: Sets the value of the `secmark` field of the first specified SKB (`to`) to be equal to the value of the `secmark` field of the second specified SKB (`from`).
- `void skb_init_secmark(struct sk_buff *skb)`: Initializes the `secmark` of the specified `skb` to be 0.

The next three fields: `mark`, `dropcount`, and `reserved_tailroom` appear in a union.

- `__u32 mark`

  This field enables identifying the SKB by marking it.

  You can set the `mark` field of the SKB, for example, with the `iptables` MARK target in an `iptables` PREROUTING rule with the mangle table.

- `iptables -A PREROUTING -t mangle -i eth1 -j  MARK  --set-mark   0x1234`

  This rule will assign the value of 0x1234 to every SKB `mark` field for incoming traffic on `eth1` before performing a routing lookup. You can also run an `iptables` rule which will check the `mark` field of every SKB to match a specified value and act upon it. Netfilter targets and `iptables` are discussed in Chapter 9, which deals with the netfilter subsystem.

- `__u32 dropcount`

  The `dropcount` counter represents the number of dropped packets (`sk_drops`) of the `sk_receive_queue` of the assigned `sock` object (`sk`). See the `sock_queue_rcv_skb()` method in `net/core/sock.c`.

- `_u32 reserved_tailroom`: Used in the `sk_stream_alloc_skb()` method.
- `sk_buff_data_t transport_header`

  The transport layer (L4) header.

  Helper methods:

  - `skb_transport_header(const struct sk_buff *skb)`: Returns the transport header of the specified `skb`.
  - `skb_transport_header_was_set(const struct sk_buff *skb)`: Returns 1 if the `transport_header` of the specified `skb` is set.

- `sk_buff_data_t network_header`

  The network layer (L3) header.

  Helper method:

  - `skb_network_header(const struct sk_buff *skb)`: Returns the network header of the specified `skb`.

- `sk_buff_data_t mac_header`

  The link layer (L2) header.

Helper methods:

- `skb_mac_header(const struct sk_buff *skb)`: Returns the MAC header of the specified `skb`.
- `skb_mac_header_was_set(const struct sk_buff *skb)`: Returns 1 if the `mac_header` of the specified `skb` was set.

- `sk_buff_data_t tail`

  The tail of the data.

- `sk_buff_data_t end`

  The end of the buffer. The `tail` cannot exceed `end`.

- `unsigned char head`

  The head of the buffer.

- `unsigned char data`

  The data head. The data block is allocated separately from the `sk_buff` allocation.

  See, in `_alloc_skb()`, `net/core/skbuff.c`:

  ```
  data = kmalloc_reserve(size, gfp_mask, node, &pfmemalloc);
  ```

  Helper methods:

  - `skb_headroom(const struct sk_buff *skb)`: This method returns the headroom, which is the number of bytes of free space at the head of the specified skb (`skb->data - skb->head`). See Figure A-1.
  - `skb_tailroom(const struct sk_buff *skb)`: This method returns the tailroom, which is the number of bytes of free space at the tail of the specified `skb` (`skb->end - skb->tail`). See Figure A-1.

Figure A-1 shows the headroom and the tailroom of an SKB.

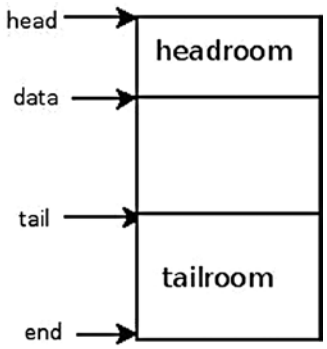

Figure A-1. *Headroom and tailroom of an SKB*

The following are some methods for handling buffers:

- `skb_put(struct sk_buff *skb, unsigned int len)`: Adds data to a buffer: this method adds `len` bytes to the buffer of the specified `skb` and increments the length of the specified `skb` by the specified `len`.
- `skb_push(struct sk_buff *skb, unsigned int len)`: Adds data to the start of a buffer; this method decrements the data pointer of the specified `skb` by the specified `len` and increments the length of the specified `skb` by the specified `len`.
- `skb_pull(struct sk_buff *skb, unsigned int len)`: Removes data from the start of a buffer; this method increments the data pointer of the specified `skb` by the specified `len` and decrements the length of the specified `skb` by the specified `len`.
- `skb_reserve(struct sk_buff *skb, int len)`: Increases the headroom of an empty `skb` by reducing the tail.

After describing some methods for handling buffers, I continue with listing the members of the `sk_buff` structure:

- `unsigned int truesize`

  The total memory allocated for the SKB (including the SKB structure itself and the size of the allocated data block).

- `atomic_t users`

  A reference counter, initialized to 1; incremented by the `skb_get()` method and decremented by the `kfree_skb()` method or by the `consume_skb()` method; the `kfree_skb()` method decrements the usage counter; if it reached 0, the method will free the SKB—otherwise, the method will return without freeing it.

  Helper methods:

  - `skb_get(struct sk_buff *skb)`: Increments the `users` reference counter by 1.
  - `skb_shared(const struct sk_buff *skb)`: Returns `true` if the number of `users` is not 1.
  - `skb_share_check(struct sk_buff *skb, gfp_t pri)`: If the buffer is not shared, the original buffer is returned. If the buffer is shared, the buffer is cloned, and the old copy drops a reference. A new clone with a single reference is returned. When being called from interrupt context or with spinlocks held, the `pri` parameter (priority) must be GFP_ATOMIC. If memory allocation fails, NULL is returned.
  - `consume_skb(struct sk_buff *skb)`: Decrements the `users` reference counter and frees the SKB if the `users` reference counter is zero.

struct skb_shared_info

The `skb_shared_info struct` is located at the end of the data block (`skb_end_pointer(SKB)`). It consists of only a few fields. Let's take a look at it:

```
struct skb_shared_info {
    unsigned char         nr_frags;
    __u8                  tx_flags;
    unsigned short        gso_size;
    unsigned short        gso_segs;
    unsigned short        gso_type;
```

```
    struct sk_buff        *frag_list;
    struct skb_shared_hwtstamps hwtstamps;
    __be32                ip6_frag_id;
    atomic_t              dataref;
    void *                destructor_arg;
    skb_frag_t            frags[MAX_SKB_FRAGS];
};
```

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

- `nr_frags`: Represents the number of elements in the `frags` array.
- `tx_flags` can be:
  - SKBTX_HW_TSTAMP: Generate a hardware time stamp.
  - SKBTX_SW_TSTAMP: Generate a software time stamp.
  - SKBTX_IN_PROGRESS: Device driver is going to provide a hardware timestamp.
  - SKBTX_DEV_ZEROCOPY: Device driver supports Tx zero-copy buffers.
  - SKBTX_WIFI_STATUS: Generate WiFi status information.
  - SKBTX_SHARED_FRAG: Indication that at least one fragment might be overwritten.
- When working with fragmentation, there are cases when you work with a list of `sk_buffs` (`frag_list`), and there are cases when you work with the `frags` array. It depends mostly on whether the Scatter/Gather mode is set.

  Helper methods:

  - `skb_is_gso(const struct sk_buff *skb)`: Returns `true` if the `gso_size` of the `skb_shared_info` associated with the specified `skb` is not 0.
  - `skb_is_gso_v6(const struct sk_buff *skb)`: Returns `true` if the `gso_type` of the `skb_shared_info` associated with the `skb` is SKB_GSO_TCPV6.
  - `skb_shinfo(skb)`: A macro that returns the `skb_shinfo` associated with the specified `skb`.
  - `skb_has_frag_list(const struct sk_buff *skb)`: Returns `true` if the `frag_list` of the `skb_shared_info` of the specified `skb` is not NULL.
  - `dataref`: A reference counter of the `skb_shared_info struct`. It is set to 1 in the method, which allocates the `skb` and initializes `skb_shared_info` (The `__alloc_skb()` method).

The net_device structure

The `net_device struct` represents the network device. It can be a physical device, like an Ethernet device, or it can be a software device, like a bridge device or a VLAN device. As with the `sk_buff` structure, I will list its important members. The `net_device struct` is defined in `include/linux/netdevice.h`:

- `char name[IFNAMSIZ]`

  The name of the network device. This is the name that you see with `ifconfig` or `ip` commands (for example `eth0`, `eth1`, and so on). The maximum length of the interface name is 16 characters. In newer distributions with `biosdevname` support, the naming scheme corresponds to the physical location of the network device. So PCI network

devices are named p<slot>p<port>, according to the chassis labels, and embedded ports (on motherboard interfaces) are named em<port>—for example, em1, em2, and so on. There is a special suffix for SR-IOV devices and Network Partitioning (NPAR)–enabled devices. Biosdevname is developed by Dell: http://linux.dell.com/biosdevname. See also this white paper: http://linux.dell.com/files/whitepapers/consistent_network_device_naming_in_linux.pdf.

Helper method:

- dev_valid_name(const char *name): Checks the validity of the specified network device name. A network device name must obey certain restrictions in order to enable creating corresponding sysfs entries. For example, it cannot be " . " or " .. "; its length should not exceed 16 characters. Changing the interface name can be done like this, for example: ip link set <oldDeviceName> p2p1 <newDeviceName>. So, for example, ip link set p2p1 name a12345678901234567 will fail with this message: Error: argument "a12345678901234567" is wrong: "name" too long. The reason is that you tried to set a device name that is longer than 16 characters. And running ip link set p2p1 name. will fail with RTNETLINK answers: Invalid argument, since you tried to set the device name to be "", which is an invalid value. See dev_valid_name() in net/core/dev.c.

- struct hlist_node name_hlist

  This is a hash table of network devices, indexed by the network device name. A lookup in this hash table is performed by dev_get_by_name(). Insertion into this hash table is performed by the list_netdevice() method, and removal from this hash table is done with the unlist_netdevice() method.

- char *ifalias

  SNMP alias interface name. Its length can be up to 256 (IFALIASZ).

  You can create an alias to a network device using this command line:

  ```
  ip link set <devName> alias myalias
  ```

  The ifalias name is exported via sysfs by /sys/class/net/<devName>/ifalias.

  Helper method:

  - dev_set_alias(struct net_device *dev, const char *alias, size_t len): Sets the specified alias to the specified network device. The specified len parameter is the number of bytes of specified alias to be copied; if the specified len is greater than 256 (IFALIASZ), the method will fail with -EINVAL.

- unsigned int irq

  The Interrupt Request (IRQ) number of the device. The network driver should call request_irq() to register itself with this IRQ number. Typically this is done in the probe() callback of the network device driver. The prototype of the request_irq() method is: int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev). The first argument is the IRQ number. The sepcified handler is the Interrupt Service Routine (ISR). The network driver should call the free_irq() method when it no longer uses this irq. In many cases, this irq is shared (the request_irq() method is called with the IRQF_SHARED flag). You can view the number of interrupts that occurred on each core by running cat /proc/interrupts. You can set the SMP affinity of the irq by echo irqMask > /proc/irq/<irqNumber>/smp_affinity.

In an SMP machine, setting the SMP affinity of interrupts means setting which cores are allowed to handle the interrupt. Some PCI network interfaces use Message Signaled Interrupts (MSIs). PCI MSI interrupts are never shared, so the IRQF_SHARED flag is not set when calling the `request_irq()` method in these network drivers. See more info in `Documentation/PCI/MSI-HOWTO.txt`.

- `unsigned long state`

  A flag that can be one of these values:

  - __LINK_STATE_START: This flag is set when the device is brought up, by the `dev_open()` method, and is cleared when the device is brought down.
  - __LINK_STATE_PRESENT: This flag is set in device registration, by the `register_netdevice()` method, and is cleared in the `netif_device_detach()` method.
  - __LINK_STATE_NOCARRIER: This flag shows whether the device detected loss of carrier. It is set by the `netif_carrier_off()` method and cleared by the `netif_carrier_on()` method. It is exported by `sysfs` via `/sys/class/net/<devName>/carrier`.
  - __LINK_STATE_LINKWATCH_PENDING: This flag is set by the `linkwatch_fire_event()` method and cleared by the `linkwatch_do_dev()` method.
  - __LINK_STATE_DORMANT: The dormant state indicates that the interface is not able to pass packets (that is, it is not "up"); however, this is a "pending" state, waiting for some external event. See section 3.1.12, "New states for IfOperStatus" in RFC 2863, "The Interfaces Group MIB."

  The `state` flag can be set with the generic `set_bit()` method.

  Helper methods:

  - `netif_running(const struct net_device *dev)`: Returns `true` if the __LINK_STATE_START flag of the state field of the specified device is set.
  - `netif_device_present(struct net_device *dev)`: Returns `true` if the __LINK_STATE_PRESENT flag of the state field of the specified device is set.
  - `netif_carrier_ok (const struct net_device *dev)`: Returns `true` if the __LINK_STATE_NOCARRIER flag of the state field of the specified device is not set.

  These three methods are defined in `include/linux/netdevice.h`.

- `netdev_features_t features`

  The set of currently active device features. These features should be changed only by the network core or in error paths of the `ndo_set_features()` callback. Network driver developers are responsible for setting the initial set of the device features. Sometimes they can use a wrong combination of features. The network core fixes this by removing an offending feature in the `netdev_fix_features()` method, which is invoked when the network interface is registered (in the `register_netdevice()` method); a proper message is also written to the kernel log.

I will mention some net_device features here and discuss them. For the full list of net_device features, look in include/linux/netdev_features.h.

- NETIF_F_IP_CSUM means that the network device can checksum L4 IPv4 TCP/UDP packets.
- NETIF_F_IPV6_CSUM means that the network device can checksum L4 IPv6 TCP/UDP packets.
- NETIF_F_HW_CSUM means that the device can checksum in hardware all L4 packets. You cannot activate NETIF_F_HW_CSUM together with NETIF_F_IP_CSUM, or together with NETIF_F_IPV6_CSUM, because that will cause duplicate checksumming.

If the driver features set includes both NETIF_F_HW_CSUM and NETIF_F_IP_CSUM features, then you will get a kernel message saying "mixed HW and IP checksum settings." In such a case, the netdev_fix_features() method removes the NETIF_F_IP_CSUM feature. If the driver features set includes both NETIF_F_HW_CSUM and NETIF_F_IPV6_CSUM features, you get again the same message as in the previous case. This time, the NETIF_F_IPV6_CSUM feature is the one which is being removed by the netdev_fix_features() method. In order for a device to support TSO (TCP Segmentation Offload), it needs also to support Scatter/Gather and TCP checksum; this means that both NETIF_F_SG and NETIF_F_IP_CSUM features must be set. If the driver features set does not include the NETIF_F_SG feature, then you will get a kernel message saying "Dropping TSO features since no SG feature," and the NETIF_F_ALL_TSO feature will be removed. If the driver features set does not include the NETIF_F_IP_CSUM feature and does not include NETIF_F_HW_CSUM, then you will get a kernel message saying "Dropping TSO features since no CSUM feature," and the NETIF_F_TSO will be removed.

■ **Note** In recent kernels, if CONFIG_DYNAMIC_DEBUG kernel config item is set, you might need to explicitly enable printing of some messages, via <debugfs>/dynamic_debug/control interface. See Documentation/dynamic-debug-howto.txt.

- NETIF_F_LLTX is the LockLess TX flag and is considered deprecated. When it is set, you don't use the generic Tx lock (This is why it is called LockLess TX). See the following macro (HARD_TX_LOCK) from net/core/dev.c:

```
#define HARD_TX_LOCK(dev, txq, cpu) { \ if ((dev->features & NETIF_F_LLTX) == 0) { \
    __netif_tx_lock(txq, cpu); \
  } \
   }
```

NETIF_F_LLTX is used in tunnel drivers like VXLAN, VETH, and in IP over IP (IPIP) tunneling driver. For example, in the IPIP tunnel module, you set the NETIF_F_LLTX flag in the ipip_tunnel_setup() method (net/ipv4/ipip.c).

The NETIF_F_LLTX flag is also used in a few drivers that have implemented their own Tx lock, like the cxgb network driver.

In drivers/net/ethernet/chelsio/cxgb/cxgb2.c, you have:

```
static int __devinit init_one(struct pci_dev *pdev,
const struct pci_device_id *ent)
{
    . . .
    netdev->features |= NETIF_F_SG | NETIF_F_IP_CSUM |
                        NETIF_F_RXCSUM | NETIF_F_LLTX;
   . . .
}
```

- NETIF_F_GRO is used to indicate that the device supports GRO (Generic Receive Offload). With GRO, incoming packets are merged at reception time. The GRO feature improves network performance. GRO replaced LRO (Large Receive Offload), which was limited to TCP/IPv4. This flag is checked in the beginning of the dev_gro_receive() method; devices that do not have this flag set will not perform the GRO handling part in this method. A driver that wants to use GRO should call the napi_gro_receive() method in the Rx path of the driver. You can enable/disable GRO with ethtool, by ethtool -K <deviceName> gro on/ ethtool -K <deviceName> gro off, respectively. You can check whether GRO is set by running ethtool –k <deviceName> and looking at the gro field.
- NETIF_F_GSO is set to indicate that the device supports Generic Segmentation Offload (GSO). GSO is a generalization of a previous solution called TSO (TCP segmentation offload), which dealt only with TCP in IPv4. GSO can handle also IPv6, UDP, and other protocols. GSO is a performance optimization, based on traversing the networking stack once instead of many times, for big packets. So the idea is to avoid segmentation in Layer 4 and defer segmentation as much as possible. The sysadmin can enable/disable GSO with ethtool, by ethtool -K <driverName> gso on/ethtool -K <driverName> gso off, respectively. You can check whether GSO is set by running ethtool –k <deviceName> and looking at the gso field. To work with GSO, you should work in Scatter/Gather mode. The NETIF_F_SG flag must be set.
- NETIF_F_NETNS_LOCAL is set for network namespace local devices. These are network devices that are not allowed to move between network namespaces. The loopback, VXLAN, and PPP network devices are examples of namespace local devices. All these devices have the NETIF_F_NETNS_LOCAL flag set. A sysadmin can check whether an interface has the NETIF_F_NETNS_LOCAL flag set or not by ethtool -k <deviceName>. This feature is fixed and cannot be changed by ethtool. Trying to move a network device of this type to a different namespace results in an error (-EINVAL). For details, look in the dev_change_net_namespace() method (net/core/dev.c). When deleting a network namespace, devices that do not have the NETIF_F_NETNS_LOCAL flag set are moved to the default initial network namespace (init_net). Network namespace local devices that have the NETIF_F_NETNS_LOCAL flag set are not moved to the default initial network namespace (init_net), but are deleted.
- NETIF_F_HW_VLAN_CTAG_RX is for use by devices which support VLAN Rx hardware acceleration. It was formerly called NETIF_F_HW_VLAN_RX and was renamed in kernel 3.10, when support for 802.1ad was added. “CTAG” was added to indicate that this device differ from “STAG” device (Service provider tagging). A device driver that sets the NETIF_F_HW_VLAN_RX feature must also define the ndo_vlan_rx_add_vid() and ndo_vlan_rx_kill_vid() callbacks. Failure to do so will avoid device registration and result in a “Buggy VLAN acceleration in driver” kernel error message.

- NETIF_F_HW_VLAN_CTAG_TX is for use by devices that support VLAN Tx hardware acceleration. It was formerly called NETIF_F_HW_VLAN_TX and was renamed in kernel 3.10 when support for 802.1ad was added.
- NETIF_F_VLAN_CHALLENGED is set for devices that can't handle VLAN packets. Setting this feature avoids registration of a VLAN device. Let's take a look at the VLAN registration method:

```
static int register_vlan_device(struct net_device *real_dev, u16 vlan_id) {
    int err;
    . . .
    err = vlan_check_real_dev(real_dev, vlan_id);
```

The first thing the vlan_check_real_dev() method does is to check the network device features and return an error if the NETIF_F_VLAN_CHALLENGED feature is set:

```
int vlan_check_real_dev(struct net_device *real_dev, u16 vlan_id)
{
        const char *name = real_dev->name;

        if (real_dev->features & NETIF_F_VLAN_CHALLENGED) {
                pr_info("VLANs not supported on %s\n", name);
                return -EOPNOTSUPP;
        }
                . . .
}
```

For example, some types of Intel e100 network device drivers set the NETIF_F_VLAN_CHALLENGED feature (see e100_probe() in drivers/net/ethernet/intel/e100.c).

You can check whether the NETIF_F_VLAN_CHALLENGED is set by running ethtool -k <deviceName> and looking at the vlan-challenged field. This is a fixed value that you cannot change with the ethtool command.

- NETIF_F_SG is set when the network interface supports Scatter/Gather IO. You can enable and disable Scatter/Gather with ethtool, by ethtool -K <deviceName> sg on/ ethtool -K <deviceName> sg off, respectively. You can check whether Scatter/Gather is set by running ethtool -k <deviceName> and looking at the sg field.
- NETIF_F_HIGHDMA is set if the device can perform access by DMA to high memory. The practical implication of setting this feature is that the ndo_start_xmit() callback of the net_device_ops object can manage SKBs, which have frags elements in high memory. You can check whether the NETIF_F_HIGHDMA is set by running ethtool -k <deviceName> and looking at the highdma field. This is a fixed value that you cannot change with the ethtool command.
- netdev_features_t hw_features

  The set of features that are changeable features. This means that their state may possibly be changed (enabled or disabled) for a particular device by a user's request. This set should be initialized in the ndo_init() callback and not changed later.

- `netdev_features_t wanted_features`

  The set of features that were requested by the user. A user may request to change various offloading features—for example, by running `ethtool -K eth1 rx on`. This generates a feature change event notification (NETDEV_FEAT_CHANGE) to be sent by the `netdev_features_change()` method.

- `netdev_features_t vlan_features`

  The set of features whose state is inherited by child VLAN devices. For example, let's look at the `rtl_init_one()` method, which is the `probe` callback of the `r8169` network device driver (see Chapter 14):

```
int rtl_init_one(struct pci_dev *pdev, const struct pci_device_id *ent)

{
   . . .
   dev->vlan_features=NETIF_F_SG|NETIF_F_IP_CSUM|NETIF_F_TSO|   NETIF_F_HIGHDMA;
   . . .
}
```

  (`drivers/net/ethernet/realtek/r8169.c`)

  This initialization means that all child VLAN devices will have these features. For example, let's say that your `eth0` device is an `r8169` device, and you add a VLAN device thus: `vconfig add eth0 100`. Then, in the initialization in the VLAN module, there is this code related to `vlan_features`:

```
static int vlan_dev_init(struct net_device *dev)
{
    . . .
    dev->features |= real_dev->vlan_features | NETIF_F_LLTX;
    . . .
}
```

  (`net/8021q/vlan_dev.c`)

  This means that it sets the features of the VLAN child device to be the `vlan_features` of the real device (which is `eth0` in this case), which were set according to what you saw earlier in the `rtl_init_one()` method.

- `netdev_features_t hw_enc_features`

  The mask of features inherited by encapsulating devices. This field indicates what encapsulation offloads the hardware is capable of doing, and drivers will need to set them appropriately. For more info about the network device features, see `Documentation/networking/netdev-features.txt`.

- `ifindex`

  The `ifindex` (Interface index) is a unique device identifier. This index is incremented by 1 each time you create a new network device, by the `dev_new_index()` method. The first network device you create, which is almost always the loopback device, has `ifindex` of 1. Cyclic integer overflow is handled by the method that handles assignment of the `ifindex` number. The `ifindex` is exported by `sysfs` via `/sys/class/net/<devName>/ifindex`.

- `struct net_device_stats stats`

  The statistics `struct`, which was left as a legacy, includes fields like the number of `rx_packets` or the number of `tx_packets`. New device drivers use the `rtnl_link_stats64 struct` (defined in `include/uapi/linux/if_link.h`) instead of the `net_device_stats struct`. Most of the network drivers implement the `ndo_get_stats64()` callback of `net_device_ops` (or the `ndo_get_stats()` callback of `net_device_ops`, when working with the older API).

  The statistics are exported via `/sys/class/net/<deviceName>/statistics`.

  Some drivers implement the `get_ethtool_stats()` callback. These drivers show statistics by `ethtool -S <deviceName>`

  See, for example, the `rtl8169_get_ethtool_stats()` method in `drivers/net/ethernet/realtek/r8169.c`.

- `atomic_long_t rx_dropped`

  A counter of the number of packets that were dropped in the RX path by the core network stack. This counter should not be used by drivers. Do not confuse the `rx_dropped` field of the `sk_buff` with the `dropped` field of the `softnet_data struct`. The `softnet_data struct` represents a per-CPU object. They are not equivalent because the `rx_dropped` of the `sk_buff` might be incremented in several methods, whereas the `dropped` counter of `softnet_data` is incremented only by the `enqueue_to_backlog()` method (`net/core/dev.c`). The dropped counter of `softnet_data` is exported by `/proc/net/softnet_stat`. In `/proc/net/softnet_stat` you have one line per CPU. The first column is the total packets counter, and the second one is the dropped packets counter.

For example:

```
cat /proc/net/softnet_stat
00000076 00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000005 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

You see here one line per CPU (you have two CPUs); for the first CPU, you see 118 total packets (hex 0x76), where one packet is dropped. For the second CPU, you see 5 total packets and 0 dropped.

- `struct net_device_ops *netdev_ops`

  The `netdev_ops` structure includes pointers for several callback methods that you want to define if you want to override the default behavior. Here are some callbacks of `netdev_ops`:

  - The `ndo_init()` callback is called when network device is registered.
  - The `ndo_uninit()` callback is called when the network device is unregistered or when the registration fails.
  - The `ndo_open()` callback handles change of device state, when a network device state is being changed from down state to up state.
  - The `ndo_stop()` callback is called when a network device state is being changed to be down.
  - The `ndo_validate_addr()` callback is called to check whether the MAC is valid. Many network drivers set the generic `eth_validate_addr()` method to be the `ndo_validate_addr()` callback. The generic `eth_validate_addr()` method returns `true` if the MAC address is not a multicast address and is not all zeroes.

- The `ndo_set_mac_address()` callback sets the MAC address. Many network drivers set the generic `eth_mac_addr()` method to be the `ndo_set_mac_address()` callback of `struct net_device_ops` for setting their MAC address. For example, the VETH driver (`drivers/net/veth.c`) or the VXLAN driver (`drivers/nets/vxlan.c`).
- The `ndo_start_xmit()` callback handles packet transmission. It cannot be NULL.
- The `ndo_select_queue()` callback is used to select a Tx queue, when working with multiqueues. If the `ndo_select_queue()` callback is not set, then the `__netdev_pick_tx()` is called. See the implementaion of the `netdev_pick_tx()` method in `net/core/flow_dissector.c`.
- The `ndo_change_mtu()` callback handles modifying the MTU. It should check that the specified MTU is not less than 68, which is the minimum MTU. In many cases, network drivers set the `ndo_change_mtu()` callback to be the generic `eth_change_mtu()` method. The `eth_change_mtu()` method should be overridden if jumbo frames are supported.
- The `ndo_do_ioctl()` callback is called when getting an IOCTL request which is not handled by the generic interface code.
- The `ndo_tx_timeout()` callback is called when the transmitter was idle for a quite a while (for watchdog usage).
- The `ndo_add_slave()` callback is called to set a specified network device as a slave to a specified netowrk device. It is used, for example, in the team network driver and in the bonding network driver.
- The `ndo_del_slave()` callback is called to remove a previously enslaved network device.
- The `ndo_set_features()` callback is called to update the configuration of a network device with new features.
- The `ndo_vlan_rx_add_vid()` callback is called when registering a VLAN id if the network device supports VLAN filtering (the NETIF_F_HW_VLAN_FILTER flag is set in the device features).
- The `ndo_vlan_rx_kill_vid()` callback is called when unregistering a VLAN id if the network device supports VLAN filtering (the NETIF_F_HW_VLAN_FILTER flag is set in the device features).

■ **Note** From kernel 3.10, the NETIF_F_HW_VLAN_FILTER flag was renamed to NETIF_F_HW_VLAN_CTAG_FILTER.

- There are also several callbacks for handling SR-IOV devices, for example, `ndo_set_vf_mac()` and `ndo_set_vf_vlan()`.

Before kernel 2.6.29, there was a callback named `set_multicast_list()` for addition of multicast addresses, which was replaced by the `dev_set_rx_mode()` method. The `dev_set_rx_mode()` callback is called primarily whenever the unicast or multicast address lists or the network interface flags are updated.

- `struct ethtool_ops *ethtool_ops`

  The `ethtool_ops` structure includes pointers for several callbacks for handling offloads, getting and setting various device settings, reading registers, getting statistics, reading RX flow hash indirection table, WakeOnLAN parameters, and many more. If the network driver does not initialize the `ethtool_ops` object, the networking core provides a default

empty `ethtool_ops` object named `default_ethtool_ops`. The management of `ethtool_ops` is done in `net/core/ethtool.c`.

Helper method:

- SET_ETHTOOL_OPS (netdev,ops): A macro which sets the specified `ethtool_ops` for the specified `net_device`.

You can view the offload parameters of a network interface device by running `ethtool -k <deviceName>`. You can set some offload parameters of a network interface device by running `ethtool -K <deviceName> offloadParameter off/on`. See `man 8 ethtool`.

- `const struct header_ops *header_ops`

  The `header_ops struct` include callbacks for creating the Layer 2 header, parsing it, rebuilding it, and more. For Ethernet it is `eth_header_ops`, defined in `net/ethernet/eth.c`.

- `unsigned int flags`

  The interface flags of the network device that you can see from userspace. Here are some flags (for a full list see `include/uapi/linux/if.h`):

  - IFF_UP flag is set when the interface state is changed from down to up.
  - IFF_PROMISC is set when the interface is in promiscuous mode (receives all packets). When running sniffers like `wireshark` or `tcpdump`, the network interface is in promiscuous mode.
  - IFF_LOOPBACK is set for the loopback device.
  - IFF_NOARP is set for devices which do not use the ARP protocol. IFF_NOARP is set, for example, in tunnel devices (see for example, in the `ipip_tunnel_setup()` method, `net/ipv4/ipip.c`).
  - IFF_POINTOPOINT is set for PPP devices. See for example, the `ppp_setup()` method, `drivers/net/ppp/ppp_generic.c`.
  - IFF_MASTER is set for master devices. See, for example, for bonding devices, the `bond_setup()` method in `drivers/net/bonding/bond_main.c`.
  - IFF_LIVE_ADDR_CHANGE flag indicates that the device supports hardware address modification when it's running. See the `eth_mac_addr()` method in `net/ethernet/eth.c`.
  - IFF_UNICAST_FLT flag is set when the network driver handles unicast address filtering.
  - IFF_BONDING is set for a bonding master device or bonding slave device. The bonding driver provides a method for aggregating multiple network interfaces into a single logical interface.
  - IFF_TEAM_PORT is set for a device used as a team port. The teaming driver is a load-balancing network software driver intended to replace the bonding driver.
  - IFF_MACVLAN_PORT is set for a device used as a macvlan port.
  - IFF_EBRIDGE is set for an Ethernet bridging device.

  The `flags` field is exported by `sysfs` via `/sys/class/net/<devName>/flags`.

Some of these flags can be set by userspace tools. For example, `ifconfig <deviceName> -arp` will set the IFF_NOARP network interface flag, and `ifconfig <deviceName> arp` will clear the IFF_NOARP flag. Note that you can do the same with the `iproute2 ip` command: `ip link set dev <deviceName> arp on` and `ip link set dev <deviceName> arp off`.

- `unsigned int priv_flags`

  The interface flags, which are invisible from userspace. For example, IFF_EBRIDGE for a bridge interface or IFF_BONDING for a bonding interface, or IFF_SUPP_NOFCS for an interface support sending custom FCS.

  Helper methods:

  - `netif_supports_nofcs()`: Returns `true` if the IFF_SUPP_NOFCS is set in the `priv_flags` of the specified device.
  - `is_vlan_dev(struct net_device *dev)`: Returns 1 if the IFF_802_1Q_VLAN flag is set in the `priv_flags` of the specified network device.

- `unsigned short gflags`

  Global flags (kept as legacy).

- `unsigned short padded`

  How much padding is added by the `alloc_netdev()` method.

- `unsigned char operstate`

  RFC 2863 operstate.

- `unsigned char link_mode`

  Mapping policy to operstate.

- `unsigned int mtu`

  The network interface MTU (Maximum Transmission Unit) value. The maximum size of frame the device can handle. RFC 791 sets 68 as a minimum MTU. Each protocol has MTU of its own. The default MTU for Ethernet is 1,500 bytes. It is set in the `ether_setup()` method, `net/ethernet/eth.c`. Ethernet packets with sizes higher than 1,500 bytes, up to 9,000 bytes, are called Jumbo frames. The network interface MTU is exported by `sysfs` via `/sys/class/net/<devName>/mtu.`

  Helper method:

  - `dev_set_mtu(struct net_device *dev, int new_mtu)`: Changes the MTU of the specified device to a new value, specified by the `mtu` parameter.

  The sysadmin can change the MTU of a network interface to 1,400, for example, in one of the following ways:

```
ifconfig <netDevice> mtu 1400
ip link set <netDevice> mtu 1400
echo 1400 > /sys/class/net/<netDevice>/mtu
```

  Many drivers implement the `ndo_change_mtu()` callback to change the MTU to perform driver-specific needed actions (like resetting the network card).

- `unsigned short type`

  The network interface hardware type. For example, for Ethernet it is ARPHRD_ETHER and is set in `ether_setup()` in `net/ethernet/eth.c`. For PPP interface, it is ARPHRD_PPP, and is set in the `ppp_setup()` method in `drivers/net/ppp/ppp_generic.c`. The type is exported by `sysfs` via `/sys/class/net/<devName>/type`.

- `unsigned short hard_header_len`

  The hardware header length. Ethernet headers, for example, consist of MAC source address, MAC destination address, and a type. The MAC source and destination addresses are 6 bytes each, and the type is 2 bytes. So the Ethernet header length is 14 bytes. The Ethernet header length is set to 14 (ETH_HLEN) in the `ether_setup()` method, `net/ethernet/eth.c`. The `ether_setup()` method is responsible for initializing some Ethernet device defaults, like the hard header len, Tx queue len, MTU, type, and more.

- `unsigned char perm_addr[MAX_ADDR_LEN]`

  The permanent hardware address (MAC address) of the device.

- `unsigned char addr_assign_type`

  Hardware address assignment type, can be one of the following:

  - NET_ADDR_PERM
  - NET_ADDR_RANDOM
  - NET_ADDR_STOLEN
  - NET_ADDR_SET

    By default, the MAC address is permanent (NET_ADDR_PERM). If the MAC address was generated with a helper method named `eth_hw_addr_random()`, the type of the MAC address is NET_ADD_RANDOM. The type of the MAC address is stored in the `addr_assign_type` member of the `net_device`. Also when changing the MAC address of the device, with `eth_mac_addr()`, you reset the `addr_assign_type` with ~NET_ADDR_RANDOM (if it was marked as NET_ADDR_RANDOM before). When a network device is registered (by the `register_netdevice()` method), if the `addr_assign_type` equals NET_ADDR_PERM, `dev->perm_addr` is set to be `dev->dev_addr`. When you set a MAC address, you set the `addr_assign_type` to be NET_ADDR_SET. This indicates that the MAC address of a device has been set by the `dev_set_mac_address()` method. The `addr_assign_type` is exported by `sysfs` via `/sys/class/net/<devName>/addr_assign_type`.

- `unsigned char addr_len`

  The hardware address length in octets. For Ethernet addresses, it is 6 (ETH_ALEN) bytes and is set in the `ether_setup()` method. The `addr_len` is exported by `sysfs` via `/sys/class/net/<deviceName>/addr_len`.

- `unsigned char neigh_priv_len`

  Used in the `neigh_alloc()` method, `net/core/neighbour.c`; `neigh_priv_len` is initialized only in the ATM code (`atm/clip.c`).

- `struct netdev_hw_addr_list uc`

  Unicast MAC addresses list, initialized by the `dev_uc_init()` method. There are three types of packets in Ethernet: unicast, multicast, and broadcast. Unicast is destined for one machine, multicast is destined for a group of machines, and broadcast is destined for all the machines in the LAN.

  Helper methods:

  - `netdev_uc_empty(dev)`: Returns 1 if the unicast list of the specified device is empty (its `count` field is 0).
  - `dev_uc_flush(struct net_device *dev)`: Flushes the unicast addresses of the specified network device and zeroes `count`.

- `struct netdev_hw_addr_list mc`

  Multicast MAC addresses list, initialized by the `dev_mc_init()` method.

  Helper methods:

  - `netdev_mc_empty(dev)`: Returns 1 if the multicast list of the specified device is empty (its `count` field is 0).
  - `dev_mc_flush(struct net_device *dev)`: Flushes the multicast addresses of the specified network device and zeroes the count field.

- `unsigned int promiscuity`

  A counter of the times a network interface card is told to work in promiscuous mode. With promiscuous mode, packets with MAC destination address which is different than the interface MAC address are not rejected. The `promiscuity` counter is used, for example, to enable more than one sniffing client; so when opening some sniffing clients (like `wireshark`), this counter is incremented by 1 for each client you open, and closing that client will decrement the promiscuity counter. When the last instance of the sniffing client is closed, `promiscuity` will be set to 0, and the device will exit from working in promiscuous mode. It is used also in the bridging subsystem, as the bridge interface needs to work in promiscuous mode. So when adding a bridge interface, the network interface card is set to work in promiscuous mode. See the call to the `dev_set_promiscuity()` method in `br_add_if()`, `net/bridge/br_if.c`.

  Helper method:

  - `dev_set_promiscuity(struct net_device *dev, int inc)`: Increments/decrements the `promiscuity` counter of the specified network device according to the specified increment. The `dev_set_promiscuity()` method can get a positive increment or a negative increment parameter. As long as the promiscuity counter remains above zero, the interface remains in promiscuous mode. Once it reaches zero, the device reverts back to normal filtering operation. Because promiscuity is an integer, the `dev_set_promiscuity()` method takes into account cyclic overflow of integer, which means it handles the case when the `promiscuity` counter is incremented when it reaches the maximum positive value an unsigned integer can reach.

- `unsigned int allmulti`

  The `allmulti` counter of the network device enables or disables the allmulticast mode. When selected, all multicast packets on the network will be received by the interface. You can set a network device to work in allmulticast mode by `ifconfig eth0 allmulti`. You disable the `allmulti` flag by `ifconfig eth0 -allmulti`.

Enabling/disabling the allmulticast mode can also be performed with the `ip` command:

```
ip link set p2p1 allmulticast on
ip link set p2p1 allmulticast off
```

You can also see the allmulticast state by inspecting the flags that are shown by the `ip` command:

```
ip addr show
flags=4610<BROADCAST,ALLMULTI,MULTICAST>  mtu 1500
```

Helper method:

- `dev_set_allmulti(struct net_device *dev, int inc)`: Increments/decrements the `allmulti` counter of the specified network device according to the specified increment (which can be a positive or a negative integer). The `dev_set_allmulti()` method also sets the IFF_ALLMULTI flag of the network device when setting the allmulticast mode and removes this flag when disabling the allmulticast mode.

The next three fields are protocol-specific pointers:

- `struct in_device __rcu *ip_ptr`

  This pointer is assigned to a pointer to `struct in_device`, which represents IPv4 specific data, in `inetdev_init()`, `net/ipv4/devinet.c`.

- `struct inet6_dev __rcu *ip6_ptr`

  This pointer is assigned to a pointer to `struct inet6_dev`, which represents IPv6 specific data, in `ipv6_add_dev()`, `net/ipv6/addrconf.c`.

- `struct wireless_dev *ieee80211_ptr`

  This is a pointer for the wireless device, assigned in the `ieee80211_if_add()` method, `net/mac80211/iface.c`.

- `unsigned long last_rx`

  Time of last Rx. It should not be set by network device drivers, unless really needed. Used, for example, in the bonding driver code.

- `struct list_head dev_list`

  The global list of network devices. Insertion to the list is done with the `list_netdevice()` method, when the network device is registered. Removal from the list is done with the `unlist_netdevice()` method, when the network device is unregistered.

- `struct list_head napi_list`

  NAPI stands for New API, a technique by which the network driver works in polling mode, and not in interrupt-driven mode, when it is under high traffic. Using NAPI under high traffic has been proven to improve performance. When working with NAPI, instead of getting an interrupt for each received packet, the network stack buffers the packets and from time to time triggers the poll method the driver registered with the `netif_napi_add()` method. When working with polling mode, the driver starts to work in interrupt-driven mode. When there is an interrupt for the first received packet, you reach the interrupt service routine (ISR), which is the method that was registered with `request_irq()`. Then the driver disables interrupts and notifies NAPI to take control,

usually by calling the `__napi_schedule()` method from the ISR. See, for example, the `cpsw_interrupt()` method in `drivers/net/ethernet/ti/cpsw`.

When the traffic is low, the network driver switches to work in interrupt-driven mode. Nowadays, most network drivers work with NAPI. The `napi_list` object is the list of `napi_struct` objects; The `netif_napi_add()` method adds `napi_struct` objects to this list, and the `netif_napi_del()` method deletes `napi_struct` objects from this list. When calling the `netif_napi_add()` method, the driver should specify its polling method and a weight parameter. The weight is a limit on the number of packets the driver will pass to the stack in each polling cycle. It is recommended to use a weight of 64. If a driver attempts to call `netif_napi_add()` with weight higher than 64 (NAPI_POLL_WEIGHT), there is a kernel error message. NAPI_POLL_WEIGHT is defined in `include/linux/netdevice.h`.

The network driver should call `napi_enable()` to enable NAPI scheduling. Usually this is done in the `ndo_open()` callback of the `net_device_ops` object. The network driver should call `napi_disable()` to disable NAPI scheduling. Usually this is done in the `ndo_stop()` callback of `net_device_ops`. NAPI is implemented using `softirqs`. This `softirq` handler is the `net_rx_action()` method and is registered by calling `open_softirq(NET_RX_SOFTIRQ, net_rx_action)` by the `net_dev_init()` method in `net/core/dev.c`. The `net_rx_action()` method invokes the poll method of the network driver which was registered with NAPI. The maximum number of packets (taken from all interfaces which are registered to polling) in one polling cycle (NAPI poll) is by default 300. It is the `netdev_budget` variable, defined in `net/core/dev.c`, and can be modified via a `procfs` entry, `/proc/sys/net/core/netdev_budget`. In the past, you could change the weight per device by writing values to a `procfs` entry, but currently, the `/sys/class/net/<device>/weight sysfs` entry is removed. See `Documentation/sysctl/net.txt`. I should also mention that the `napi_complete()` method removes a device from the polling list. When a network driver wants to return to work in interrupt-driven mode, it should call the `napi_complete()` method to remove itself from the polling list.

- `struct list_head unreg_list`

  The list of unregistered network devices. Devices are added to this list when they are unregistered.

- `unsigned char *dev_addr`

  The MAC address of the network interface. Sometimes you want to assign a random MAC address. You do that by calling the `eth_hw_addr_random()` method, which also sets the `addr_assign_type` to be NET_ADDR_RANDOM.

  The `dev_addr` field is exported by `sysfs` via `/sys/class/net/<devName>/address`.

  You can change `dev_addr` with userspace tools like `ifconfig` or `ip` of `iproute2`.

  Helper methods: Many times you invoke the following helper methods on Ethernet addresses in general and on `dev_addr` field of a network device in particular:

  - `is_zero_ether_addr(const u8 *addr)`: Returns `true` if the address is all zeroes.
  - `is_multicast_ether_addr(const u8 *addr)`: Returns `true` if the address is a multicast address. By definition the broadcast address is also a multicast address.
  - `is_valid_ether_addr (const u8 *addr)`: Returns `true` if the specified MAC address is not 00:00:00:00:00:00, is not a multicast address, and is not a broadcast address (FF:FF:FF:FF:FF:FF).

- `struct netdev_hw_addr_list dev_addrs`

  The list of device hardware addresses.

- `unsigned char broadcast[MAX_ADDR_LEN]`

  The hardware broadcast address. For Ethernet devices, the broadcast address is initialized to 0XFFFFFF in the `ether_setup()` method, `net/ethernet/eth.c`. The broadcast address is exported by `sysfs` via `/sys/class/net/<devName>/broadcast`.

- `struct kset *queues_kset`

  A `kset` is a group of `kobjects` of a specific type, belonging to a specific subsystem.

  The `kobject` structure is the basic type of the device model. A Tx queue is represented by struct `netdev_queue`, and the Rx queue is represented by `struct netdev_rx_queue`. Each of them holds a `kobject` pointer. The `queues_kset` object is a group of all `kobjects` of the Tx queues and Rx queues. Each Rx queue has the `sysfs` entry `/sys/class/net/<deviceName>/queues/<rx-queueNumber>`, and each Tx queue has the `sysfs` entry `/sys/class/net/<deviceName>/queues/<tx-queueNumber>`. These entries are added with the `rx_queue_add_kobject()` method and the `netdev_queue_add_kobject()` method respectively, in `net/core/net-sysfs.c`. For more information about the `kobject` and the device model, see `Documentation/kobject.txt`.

- `struct netdev_rx_queue *_rx`

  An array of Rx queues (`netdev_rx_queue` objects), initialized by the `netif_alloc_rx_queues()` method. The Rx queue to be used is determined in the `get_rps_cpu()` method. See more info about RPS in the description of the `rxhash` field in the previous `sk_buff` section.

- `unsigned int num_rx_queues`

  The number of Rx queues allocated in the `register_netdev()` method.

- `unsigned int real_num_rx_queues`
  Number of Rx queues currently active in the device.

  Helper method:

  - `netif_set_real_num_rx_queues (struct net_device *dev, unsigned int rxq)`: Sets the actual number of Rx queues used for the specified device according to the specified number of Rx queues. The relevant `sysfs` entries (`/sys/class/net/<devName>/queues/*`) are updated (only in the case that the state of the device is NETREG_REGISTERED or NETREG_UNREGISTERING). Note that `alloc_netdev_mq()` initializes `num_rx_queues`, `real_num_rx_queues`, `num_tx_queues` and `real_num_tx_queues` to the same value. One can set the number of Tx queues and Rx queues by using `ip link` when adding a device. For example, if you want to create a VLAN device with 6 Tx queues and 7 Rx queues, you can run this command:

    ```
    ip link add link p2p1 name p2p1.1 numtxqueues 6 numrxqueues 7 type vlan id 8
    ```

  - `rx_handler_func_t __rcu *rx_handler`

Helper methods:

- `netdev_rx_handler_register(struct net_device *dev, rx_handler_func_t *rx_handler  void *rx_handler_data)`

The `rx_handler` callback is set by calling the `netdev_rx_handler_register()` method. It is used, for example, in bonding, team, openvswitch, macvlan, and bridge devices.

- `netdev_rx_handler_unregister(struct net_device *dev)`: Unregisters a receive handler for the specified network device.

- `void __rcu *rx_handler_data`

  The `rx_handler_data` field is also set by the `netdev_rx_handler_register()` method when a non-NULL value is passed to the `netdev_rx_handler_register()` method.

- `struct netdev_queue __rcu *ingress_queue`

  Helper method:

  - `struct netdev_queue *dev_ingress_queue(struct net_device *dev)`: Returns the `ingress_queue` of the specified `net_device` (`include/linux/rtnetlink.h`).

- `struct netdev_queue *_tx`

  An array of Tx queues (`netdev_queue` objects), initialized by the `netif_alloc_netdev_queues()` method.

  Helper method:

  - `netdev_get_tx_queue(const struct net_device *dev,unsigned int index)`: Returns the Tx queue (`netdev_queue` object), an element of the `_tx` array of the specified network device at the specified `index`.

- `unsigned int num_tx_queues`

  Number of Tx queues, allocated by the `alloc_netdev_mq()` method.

- `unsigned int real_num_tx_queues`

  Number of Tx queues currently active in the device.

  Helper method:

  - `netif_set_real_num_tx_queues(struct net_device *dev, unsigned int txq)`: Sets the actual number of Tx queues used.

- `struct Qdisc *qdisc`

  Each device maintains a queue of packets to be transmitted named `qdisc`. The Qdisc (Queuing Disciplines) layer implements the Linux kernel traffic management. The default `qdisc` is `pfifo_fast`. You can set a different `qdisc` using `tc`, the traffic control tool of the `iproute2` package. You can view the `qdisc` of your network device by the using the `ip` command:

  ```
  ip addr show <deviceName>
  ```

For example, running

```
ip addr show eth1
```

can give:

```
2: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
link/ether 00:e0:4c:53:44:58 brd ff:ff:ff:ff:ff:ff
inet 192.168.2.200/24 brd 192.168.2.255 scope global eth1
inet6 fe80::2e0:4cff:fe53:4458/64 scope link
valid_lft forever preferred_lft forever
```

In this example, you can see that a `qdisc` of `pfifo_fast` is used, which is the default.

- `unsigned long tx_queue_len`

  The maximum number of allowed packets per queue. Each hardware layer has its own `tx_queue_len` default. For Ethernet devices, `tx_queue_len` is set to 1,000 by default (see the `ether_setup()` method). For FDDI, `tx_queue_len` is set to 100 by default (see the `fddi_setup()` method in `net/802/fddi.c`).

  The `tx_queue_len` field is set to 0 for virtual devices, such as the VLAN device, because the actual transmission of packets is done by the real device on which these virtual devices are based. You can set the Tx queue length of a device by using the command `ifconfig` (this option is called `txqueuelen`) or by using the command `ip link show` (it is called `qlen`), in this way, for example:

  ```
  ifconfig  p2p1 txqueuelen 900
  ip link set txqueuelen 950 dev p2p1
  ```

  The Tx queue length is exported via the following `sysfs` entry: `/sys/class/net/<deviceName>/tx_queue_len`.

- `unsigned long trans_start`

  The time (in `jiffies`) of the last transmission.

- `int watchdog_timeo`

  The watchdog is a timer that will invoke a callback when the network interface was idle and did not perform transmission in some specified timeout interval. Usually the driver defines a watchdog callback which will reset the network interface in such a case. The `ndo_tx_timeout()` callback of `net_device_ops` serves as the watchdog callback. The `watchdog_timeo` field represents the timeout that is used by the watchdog. See the `dev_watchdog()` method, `net/sched/sch_generic.c`.

- `int __percpu *pcpu_refcnt`

  Per CPU network device reference counter.

  Helper methods:

  - `dev_put(struct net_device *dev)`: Decrements the reference count.
  - `dev_hold(struct net_device *dev)`: Increments the reference count.

- `struct hlist_node index_hlist`

  This is a hash table of network devices, indexed by the network device index (the `ifindex` field). A lookup in this table is performed by the `dev_get_by_index()` method. Insertion into this table is performed by the `list_netdevice()` method, and removal from this list is done with the `unlist_netdevice()` method.

- `enum {...} reg_state`

  An `enum` that represents the various registration states of the network device.

  Possible values:

  - NETREG_UNINITIALIZED: When the device memory is allocated, in the `alloc_netdev_mqs()` method.
  - NETREG_REGISTERED: When the `net_device` is registered, in the `register_netdevice()` method.
  - NETREG_UNREGISTERING: When unregistering a device, in the `rollback_registered_many()` method.
  - NETREG_UNREGISTERED: The network device is unregistered but it is not freed yet.
  - NETREG_RELEASED: The network device is in the last stage of freeing the allocated memory of the network device, in the `free_netdev()` method.

- NETREG_DUMMY: Used in the `dummy` device, in the `init_dummy_netdev()` method. See `drivers/net/dummy.c`.

- `bool dismantle`

  A Boolean flag that shows that the device is in dismantle phase, which means that it is going to be freed.

- `enum {...} rtnl_link_state`

  This is an `enum` that can have two values that represent the two phases of creating a new link:

  - RTNL_LINK_INITIALIZE: The ongoing state, when creating the link is still not finished.
  - RTNL_LINK_INITIALIZING: The final state, when work is finished.

  See the `rtnl_newlink()` method in `net/core/rtnetlink.c`.

- `void (*destructor)(struct net_device *dev)`

  This destructor callback is called when unregistering a network device, in the `netdev_run_todo()` method. It enables network devices to perform additional tasks that need to be done for unregistering. For example, the loopback device destructor callback, `loopback_dev_free()`, calls `free_percpu()` for freeing its statistics object and `free_netdev()`. Likewise the team device destructor callback, `team_destructor()`, also calls `free_percpu()` for freeing its statistics object and `free_netdev()`. And there are many other network device drivers that define a `destructor` callback.

- `struct net *nd_net`

  The network namespace this network device is inside. Network namespaces support was added in the 2.6.29 kernel. These features provide process virtualization, which is considered lightweight in comparison to other virtualization solutions like KVM and Xen. There is currently support for six namespaces in the Linux kernel. In order to support network namespaces, a structure called `net` was added. This structure represents a network namespace. The process descriptor (`task_struct`) handles the network namespace and other namespaces via a new member which was added for namespaces support, named `nsproxy`. This `nsproxy` includes a network namespace object called `net_ns`, and also four other namespace objects of the following namespaces: pid namespace, mount namespace, uts namespace, and ipc namespace; the sixth namespace, the user namespace, is kept in struct `cred` (the credentials object) which is a member of the process descriptor, `task_struct`).

  Network namespaces provide a partitioning and isolation mechanism which enables one process or a group of processes to have a private view of a full network stack of their own. By default, after boot all network interfaces belong to the default network namespace, `init_net`. You can create a network namespace with userspace tools using the `ip` command from `iproute2` package or with the `unshare` command of `util-linux`—or by writing your own userspace application and invoking the `unshare()` or the `clone()` system calls with the CLONE_NEWNET flag. Moreover, you can also change the network namespace of a process by invoking the `setns()` system call. This `setns()` system call and the `unshare()` system call were added specially to support namespaces. The `setns()` system call can attach to the calling process an existing namespace of any type (network namespace, pid namespace, mount namespace, and so on). You need CAP_SYS_ADMIN privilege to call `set_ns()` for all namespaces, except the user namespace. See `man 2 setns`.

  A network device belongs to exactly one network namespace at a given moment. And a network socket belongs to exactly one network namespace at a given moment. Namespaces do not have names, but they do have a unique inode which identifies them. This unique inode is generated when the namespace is created and can be read by reading a `procfs` entry (the command `ls -al /proc/<pid>/ns/` shows all the unique inode numbers symbolic links of a process—you can also read these symbolic links with the `readlink` command).

  For example, using the `ip` command, creating a new namespace called `ns1` is done thus:

  ```
  ip netns add myns1
  ```

  Each newly created network namespace includes only the loopback device and includes no sockets. Each device (like a bridge device or a VLAN device) that is created from a process that runs in that namespace (like a shell) belongs to that namespace.

Removing a namespace is done using the following command:

```
ip netns del myns1
```

■ **Note** After deleting a namespace, all its physical network devices are moved to the default network namespace. Local devices (namespace local devices that have the NETIF_F_NETNS_LOCAL flag set, like PPP device or VXLAN device) are not moved to the default network namespace but are deleted.

Showing the list of all network namespaces on the system is done with this command:

```
ip netns list
```

Assigning the p2p1 interface to the `myns1` network namespace is done by the command:

```
ip link set p2p1 netns myns1
```

Opening a shell in `myns1` is done thus:

```
ip netns exec myns1 bash
```

With the `unshare` utility, creating a new namespace and starting a bash shell inside is done thus:

```
unshare --net bash
```

Two network namespaces can communicate by using a special virtual Ethernet driver, `veth.` (`drivers/net/veth.c`).

Helper methods:

- `dev_change_net_namespace(struct net_device *dev, struct net *net, const char *pat)`: Moves the network device to a different network namespace, specified by the `net` parameter. Local devices (devices in which the NETIF_F_NETNS_LOCAL feature is set) are not allowed to change their namespace. This method returns -EINVAL for this type of device. The `pat` parameter, when it is not NULL, is the name pattern to try if the current device name is already taken in the destination network namespace. The method also sends a KOBJ_REMOVE uevent for removing the old namespace entries from `sysfs`, and a KOBJ_ADD uevent to add the `sysfs` entries to the new namespace. This is done by invoking the `kobject_uevent()` method specifying the corresponding uevent.
- `dev_net(const struct net_device *dev)`: Returns the network namespace of the specified network device.
- `dev_net_set(struct net_device *dev, struct net *net)`: Decrements the reference count of the `nd_net` (namespace object) of the specified device and assigns the specified network namespace to it.

The following four fields are members in a union:

- `struct pcpu_lstats __percpu *lstats`

  The loopback network device statistics.

- `struct pcpu_tstats __percpu *tstats`

  The tunnel statistics.

- `struct pcpu_dstats __percpu *dstats`

  The dummy network device statistics.

- `struct pcpu_vstats __percpu *vstats`

  The VETH (Virtual Ethernet) statistics.

- `struct device dev`

The `device` object associated with the network device. Every device in the Linux kernel is associated with a device object, which is an instance of the `device` structure. For more information about the `device` structure, I suggest you read the "Devices" section in Chapter 14 of *Linux Device Drivers*, 3rd Edition (O'Reilly, 2005) and `Documentation/driver-model/overview.txt`.

Helper methods:

- `to_net_dev(d)`: Returns the `net_device` object that contains the specified device as its device object.
- SET_NETDEV_DEV (net, pdev): Sets the parent of the `dev` member of the specified network device to be that specified device (the second argument, `pdev`).

  With virtual devices, you do not call the SET_NETDEV_DEV() macro. As a result, entries for these virtual devices are created under `/sys/devices/virtual/net`.

  The SET_NETDEV_DEV() macro should be called before calling the `register_netdev()` method.

- SET_NETDEV_DEVTYPE(net, devtype): Sets the type of the `dev` member of the specified network device to be the specified type. The type is a `device_type` object.

  SET_NETDEV_DEVTYPE() is used, for example, in the `br_dev_setup()` method, in `net/bridge/br_device.c`:

```
static struct device_type br_type = {
.name = "bridge",
};

void br_dev_setup(struct net_device *dev)
{
    . . .
    SET_NETDEV_DEVTYPE(dev, &br_type);
    . . .

}
```

With the udevadm tool (udev management tool), you can find the device type, for example, for a bridge device named mybr:

```
udevadm info -q all -p /sys/devices/virtual/net/mybr

P: /devices/virtual/net/mybr

E: DEVPATH=/devices/virtual/net/mybr

E: DEVTYPE=bridge

E: ID_MM_CANDIDATE=1

E: IFINDEX=7

E: INTERFACE=mybr

E: SUBSYSTEM=net
```

- `const struct attribute_group *sysfs_groups[4]`

  Used by networking sysfs.

- `struct rtnl_link_ops *rtnl_link_ops`

  The rtnetlink link operations object. It consists of various callbacks for handling network devices, for example:

  - `newlink()` for configuring and registering a new device.
  - `changelink()` for changing parameters of an existing device.
  - `dellink()` for removing a device.
  - `get_num_tx_queues()` for getting the number of Tx queues.
  - `get_num_rx_queues()` for getting the number of Rx queues.

  Registration and unregistration of `rtnl_link_ops` object is done with the `rtnl_link_register()` method and the `rtnl_link_unregister()` method, respectively.

- `unsigned int gso_max_size`

  Helper method:

  - `netif_set_gso_max_size(struct net_device *dev, unsigned int size)`: Sets the specified `gso_max_size` for the specified network device.

- `u8 num_tc`

  The number of traffic classes in the net device.

Helper method:

- `netdev_set_num_tc(struct net_device *dev, u8 num_tc)`: Sets the `num_tc` of the specified network device (the maximum value of `num_tc` can be TC_MAX_QUEUE, which is 16).
- `int netdev_get_num_tc(struct net_device *dev)`: Returns the `num_tc` value of the specified network device.

- `struct netdev_tc_txq tc_to_txq[TC_MAX_QUEUE]`
- `u8 prio_tc_map[TC_BITMASK + 1];`
- `struct netprio_map __rcu *priomap`

The network priority `cgroup` module provides an interface to set the priority of network traffic. The cgroups layer is a Linux kernel layer that enables process resource management and process isolation. It enables assigning one task or several tasks to a system resource, like a networking resource, memory resource, CPU resource, and so on. The cgroups layer implements a Virtual File System (VFS) and is managed by filesystem operations like mounting/unmounting, creating files and directories, writing to cgroup VFS control files, and so forth. The cgroup project was started in 2005 by developers from Google (Paul Manage, Rohit Seth, and others). Some projects are based on cgroups usage, like `systemd` and `lxc` (Linux containers). Google has its own implementation of containers, based on cgroups. There is no relation between the cgroup implementation and the namespaces implementation. In the past, there was a namespace controller in cgroups but it was removed. No new system calls were added for cgroups implementations, and the cgroup code additions are not critical in terms of performance. There are two networking cgroups modules: `net_prio` and `net_cls`. These two `cgroup` modules are relatively short and simple.

Setting the priority of network traffic with the `netprio cgroup` module is done by writing an entry to a `cgroup` control file, `/sys/fs/cgroup/net_prio/<group>/net_prio.ifpriomap`. The entry is in the form "deviceName priority." It is true that an application can set the priority of its traffic via the `setsockopt()` system call with SO_PRIORITY, but this is not always possible. Sometimes you cannot change the code of certain applications. Moreover, you want to let the system administrator decide on priority according to site-specific setup. The `netprio` kernel module is a solution when using the `setsockopt()` system call with SO_PRIORITY is not feasible. The `netprio` module also exports another `/sys/fs/cgroup/netprio` entry, `net_prio.prioidx`. The `net_prio.prioidx` entry is a read-only file and contains a unique integer value that the kernel uses as an internal representation of this cgroup.

`netprio` is implemented in `net/core/netprio_cgroup.c`.

`net_cls` is implemented in `net/sched/cls_cgroup.c`.

The network classifier cgroup provides an interface to tag network packets with a class identifier (`classid`). Creating a `net_cls` cgroups instance creates a `net_cls.classid` control file. This `net_cls.classid` value is initialized to 0. You can set up rules for this classid with `tc`, the traffic control command of `iproute2`.

For more information, see `Documentation/cgroups/net_cls.txt`.

- `struct phy_device *phydev`

  The associated PHY device. The `phy_device` is the Layer 1 (the physical layer) device. It is defined in `include/linux/phy.h`. For many devices, PHY flow control parameters like autonegotiation, speed, or duplex can be configured via the PHY device with `ethtool` commands. See `man 8 ethtool` for more info.

- `int group`

  The group that the network device belongs to. It is initialized with INIT_NETDEV_GROUP (0) by default. The group is exported by `sysfs` via `/sys/class/net/<devName>/netdev_group`. The network device group filters are used for example in netfilter, in `net/netfilter/xt_devgroup.c`.

  Helper method:

  - `void dev_set_group(struct net_device *dev, int new_group)`: Changes the group of the specified device to be the specified group.

- `struct pm_qos_request pm_qos_req`

  Power Management Quality Of Service request object, defined in `include/linux/pm_qos.h`.

  For more details about PM QoS, see `Documentation/power/pm_qos_interface.txt`.

Next I will describe the netdev_priv() method and the alloc_netdev() macro, which are used a lot in network drivers.

The `netdev_priv(struct net_device *netdev)` method returns a pointer to the end of the `net_device`. This area is used by drivers, which define a private network interface structure in order to store private data. For example, in `drivers/net/ethernet/intel/e1000e/netdev.c`:

```
static int e1000_open(struct net_device *netdev)
{
    struct e1000_adapter *adapter = netdev_priv(netdev);
    . . .
}
```

The `netdev_priv()` method is used also for software devices, like the VLAN device. So you have:

```
static inline struct vlan_dev_priv *vlan_dev_priv(const struct net_device *dev)
{
    return netdev_priv(dev);
}
```

(net/8021q/vlan.h)

- The `alloc_netdev(sizeof_priv, name, setup)` macro is for allocation and initialization of a network device. It is in fact a wrapper around `alloc_netdev_mqs()`, with one Tx queue and one Rx queue. `sizeof_priv` is the size of private data to allocate space for. The `setup` method is a callback to initialize the network device. For Ethernet devices, it is usually `ether_setup()`.

  For Ethernet devices, you can use the `alloc_etherdev()` or `alloc_etherdev_mq()` macros, which eventually invoke `alloc_etherdev_mqs()`; `alloc_etherdev_mqs()` is also a wrapper around `alloc_netdev_mqs()`, with the `ether_setup()` as the setup callback method.

- Software devices usually define a setup method of their own. So, in PPP you have the `ppp_setup()` method in `drivers/net/ppp/ppp_generic.c`, and for VLAN you have `vlan_setup(struct net_device *dev)` in `net/8021q/vlan.h`.

RDMA (Remote DMA)

The following sections describe the RDMA API for the following data structures:

- RDMA device
- Protection Domain (PD)
- eXtended Reliable Connected (XRC)
- Shared Receive Queue (SRQ)
- Address Handle (AH)
- Multicast Groups
- Completion Queue (CQ)
- Queue Pair (QP)
- Memory Window (MW)
- Memory Region (MR)

RDMA Device

The following methods are related to the RDMA device.

The ib_register_client() Method

The `ib_register_client()` method registers a kernel client that wants to use the RDMA stack. The specified callbacks will be called for every RDMA device that currently exists in the system and for every new device that will be detected or removed by the system (using hot-plug). It will return 0 on success or the errno value with the reason for the failure.

```
int ib_register_client(struct ib_client *client);
```

- `client`: A structure that describes the attributes of the registration.

The ib_client Struct:

The device registration attributes are represented by `struct ib_client`:

```
struct ib_client {
        char  *name;
        void (*add)   (struct ib_device *);
        void (*remove)(struct ib_device *);

        struct list_head list;
};
```

- `name`: The name of the kernel module to be registered.
- `add`: A callback to be called for each RDMA device that exists in the system and for every new RDMA device that will be detected by the kernel.
- `remove`: A callback to be called for each RDMA device being removed by the kernel.

The ib_unregister_client() Method

The `ib_unregister_client()` method unregisters a kernel module that wants to stop using the RDMA stack.

```
void ib_unregister_client(struct ib_client *client);
```

- `device`: A structure that describes the attributes of the unregistration.
- `client`: Should be the same object that was used when `ib_register_client()` was called.

The ib_get_client_data() Method

The `ib_get_client_data()` method returns the client context which was associated with the RDMA device using the `ib_set_client_data()` method.

```
void *ib_get_client_data(struct ib_device *device, struct ib_client *client);
```

- `device`: The RDMA device to get the client context from.
- `client`: The object that describes the attributes of the registration/unregistration.

The ib_set_client_data() Method

The `ib_set_client_data()` method sets a client context to be associated with the RDMA device.

```
void  ib_set_client_data(struct ib_device *device, struct ib_client *client,
              void *data);
```

- `device`: The RDMA device to set the client context with.
- `client`: The object that describes the attributes of the registration/unregistration.
- `data`: The client context to associate.

The INIT_IB_EVENT_HANDLER macro

The INIT_IB_EVENT_HANDLER macro initializes an event handler for the asynchronous events that may occur to the RDMA device. This macro should be used before calling the `ib_register_event_handler()` method:

```
#define INIT_IB_EVENT_HANDLER(_ptr, _device, _handler)        \
    do {                                \
        (_ptr)->device  = _device;            \
        (_ptr)->handler = _handler;            \
        INIT_LIST_HEAD(&(_ptr)->list);            \
    } while (0)
```

- `_ptr`: A pointer to the event handler that will be provided to the `ib_register_event_handler()` method.
- `_device`: The RDMA device context; upon its events the callback will be called.
- `_handler`: The callback that will be called with every asynchronous event.

The ib_register_event_handler() Method

The `ib_register_event_handler()` method registers an RDMA event to be called with every handler asynchronous event. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_register_event_handler  (struct ib_event_handler *event_handler);
```

- `event_handler`: The event handler that was initialized with the macro INIT_IB_EVENT_HANDLER. This callback may occur in interrupt context.

The ib_event_handler struct:

The RDMA event handler is represented by `struct ib_event_handler`:

```
struct ib_event_handler {
    struct ib_device *device;
    void            (*handler)(struct ib_event_handler *, struct ib_event *);
    struct list_head  list;
};
```

The ib_event Struct

The event callback is being called with the new event that happens to the RDMA device. This event is represented by `struct ib_event`.

```
struct ib_event {
    struct ib_device    *device;
    union {
        struct ib_cq    *cq;
        struct ib_qp    *qp;
```

```
        struct ib_srq    *srq;
        u8        port_num;
    } element;
    enum ib_event_type    event;
};
```

- `device`: The RDMA device to which the asynchronous event occurred.
- `element.cq`: If this is a CQ event, the CQ on which the asynchronous event occurred.
- `element.qp`: If this is a QP event, the QP on which the asynchronous event occurred.
- `element.srq`: If this is an SRQ event, the SRQ on which the asynchronous event occurred.
- `element.port_num`: If this is a port event, the port number on which the asynchronous event occurred.
- `event`: The type of the asynchronous event that was occurred. It can be:
  - IB_EVENT_CQ_ERR: CQ event. An error occurred to the CQ and no more Work Completions will be generated to it.
  - IB_EVENT_QP_FATAL: QP event. An error occurred to the QP that prevents it from reporting an error through a Work Completion.
  - IB_EVENT_QP_REQ_ERR: QP event. An incoming RDMA request caused a transport error violation in the targeted QP.
  - IB_EVENT_QP_ACCESS_ERR: QP event. An incoming RDMA request caused a requested error violation in the targeted QP.
  - IB_EVENT_COMM_EST: QP event. A communication established event occurred. An incoming message was received by a QP when it was in the RTR state.
  - IB_EVENT_SQ_DRAINED: QP event. Send Queue drain event. The QP's Send Queue was drained.
  - IB_EVENT_PATH_MIG: QP event. Path migration was completed successfully and the primary was changed.
  - IB_EVENT_PATH_MIG_ERR: QP event. There was an error when trying to perform path migration.
  - IB_EVENT_DEVICE_FATAL: Device event. There was an error with the RDMA device.
  - IB_EVENT_PORT_ACTIVE: Port event. The port state has become active.
  - IB_EVENT_PORT_ERR: Port event. The port state was active and it is no longer active.
  - IB_EVENT_LID_CHANGE: Port event. The LID of the port was changed.
  - IB_EVENT_PKEY_CHANGE: Port event. A P_Key entry was changed in the port's P_Key table.
  - IB_EVENT_SM_CHANGE: Port event. The Subnet Manager that manages this port was change.
  - IB_EVENT_SRQ_ERR: SRQ event. An error occurred to the SRQ.
  - IB_EVENT_SRQ_LIMIT_REACHED: SRQ event/SRQ limit event. The number of Receive Requests in the SRQ dropped below the requested watermark.

- IB_EVENT_QP_LAST_WQE_REACHED: QP event. Last Receive Request reached from the SRQ, and it won't consume any more Receive Requests from it.
- IB_EVENT_CLIENT_REREGISTER: Port event. The client should reregister to all services from the Subnet Administrator.
- IB_EVENT_GID_CHANGE: Port event. A GID entry was changed in the port's GID table.

The ib_unregister_event_handler() Method

The `ib_unregister_event_handler()` method unregisters an RDMA event handler. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_unregister_event_handler(struct ib_event_handler *event_handler);
```

- `event_handler`: The event handler to be unregistered. It should be the same object that was registered with `ib_register_event_handler()`.

The ib_query_device() Method

The `ib_query_device()` method queries the RDMA device for its attributes. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_query_device(struct ib_device *device,
        struct ib_device_attr *device_attr);
```

- `device`: The RDMA device to be queried.
- `device_attr`: Pointer to a structure of an RDMA device attributes that will be filled.

The ib_device_attr struct:

The RDMA device attributes are represented by `struct ib_device_attr`:

```
struct ib_device_attr {
    u64            fw_ver;
    __be64         sys_image_guid;
    u64            max_mr_size;
    u64            page_size_cap;
    u32            vendor_id;
    u32            vendor_part_id;
    u32            hw_ver;
    int            max_qp;
    int            max_qp_wr;
    int            device_cap_flags;
    int            max_sge;
    int            max_sge_rd;
    int            max_cq;
    int            max_cqe;
    int            max_mr;
    int            max_pd;
```

```
    int             max_qp_rd_atom;
    int             max_ee_rd_atom;
    int             max_res_rd_atom;
    int             max_qp_init_rd_atom;
    int             max_ee_init_rd_atom;
    enum ib_atomic_cap    atomic_cap;
    enum ib_atomic_cap    masked_atomic_cap;
    int             max_ee;
    int             max_rdd;
    int             max_mw;
    int             max_raw_ipv6_qp;
    int             max_raw_ethy_qp;
    int             max_mcast_grp;
    int             max_mcast_qp_attach;
    int             max_total_mcast_qp_attach;
    int             max_ah;
    int             max_fmr;
    int             max_map_per_fmr;
    int             max_srq;
    int             max_srq_wr;
    int             max_srq_sge;
    unsigned int    max_fast_reg_page_list_len;
    u16             max_pkeys;
    u8              local_ca_ack_delay;
};
```

- `fw_ver`: A number which represents the FW version of the RDMA device. It can be evaluated as ZZZZYYXX: Zs are the major number, Ys are the minor number, and Xs are the build number.
- `sys_image_guid`: The system image GUID: Has a unique value for each system.
- `max_mr_size`: The maximum supported MR size.
- `page_size_cap`: Bitwise OR for all of supported memory page shifts.
- `vendor_id`: The IEEE vendor ID.
- `vendor_part_id`: Device's part ID, as supplied by the vendor.
- `hw_ver`: Device's HW version, as supplied by the vendor.
- `max_qp`: Maximum supported number of QPs.
- `max_qp_wr`: Maximum supported number of Work Requests in each non-RD QP.
- `device_cap_flags`: Supported capabilities of the RDMA device. It is a bitwise OR of the masks:
  - IB_DEVICE_RESIZE_MAX_WR: The RDMA device supports resize of the number of Work Requests in a QP.
  - IB_DEVICE_BAD_PKEY_CNTR: The RDMA device supports the ability to count the number of bad P_Keys.
  - IB_DEVICE_BAD_QKEY_CNTR: The RDMA device supports the ability to count the number of bad Q_Keys.

- IB_DEVICE_RAW_MULTI: The RDMA device supports raw packet multicast.
- IB_DEVICE_AUTO_PATH_MIG: The RDMA device supports Automatic Path Migration.
- IB_DEVICE_CHANGE_PHY_PORT: The RDMA device supports changing the QP's primary Port number.
- IB_DEVICE_UD_AV_PORT_ENFORCE: The RDMA device supports enforcements of the port number of UD QP and Address Handle.
- IB_DEVICE_CURR_QP_STATE_MOD: The RDMA device supports the current QP modifier when calling `ib_modify_qp()`.
- IB_DEVICE_SHUTDOWN_PORT: The RDMA device supports port shutdown.
- IB_DEVICE_INIT_TYPE: The RDMA device supports setting InitType and InitTypeReply.
- IB_DEVICE_PORT_ACTIVE_EVENT: The RDMA device supports the generation of the port active asynchronous event.
- IB_DEVICE_SYS_IMAGE_GUID: The RDMA device supports system image GUID.
- IB_DEVICE_RC_RNR_NAK_GEN: The RDMA device supports RNR-NAK generation for RC QPs.
- IB_DEVICE_SRQ_RESIZE: The RDMA device supports resize of a SRQ.
- IB_DEVICE_N_NOTIFY_CQ: The RDMA device supports notification when N Work Completions exists in the CQ.
- IB_DEVICE_LOCAL_DMA_LKEY: The RDMA device supports Zero Stag (in iWARP) and reserved LKey (in InfiniBand).
- IB_DEVICE_RESERVED: Reserved bit.
- IB_DEVICE_MEM_WINDOW: The RDMA device supports Memory Windows.
- IB_DEVICE_UD_IP_CSUM: The RDMA device supports insertion of UDP and TCP checksum on outgoing UD IPoIB messages and can verify the validity of those checksum for incoming messages.
- IB_DEVICE_UD_TSO: The RDMA device supports TCP Segmentation Offload.
- IB_DEVICE_XRC: The RDMA device supports the eXtended Reliable Connected transport.
- IB_DEVICE_MEM_MGT_EXTENSIONS: The RDMA device supports memory management extensions support.
- IB_DEVICE_BLOCK_MULTICAST_LOOPBACK: The RDMA device supports blocking multicast loopback.
- IB_DEVICE_MEM_WINDOW_TYPE_2A: The RDMA device supports Memory Windows type 2A: association with a QP number.
- IB_DEVICE_MEM_WINDOW_TYPE_2B: The RDMA device supports Memory Windows type 2B: association with a QP number and a PD.

- max_sge: Maximum supported number of scatter/gather elements per Work Request in a non-RD QP.
- max_sge_rd: Maximum supported number of scatter/gather elements per Work Request in an RD QP.
- max_cq: Maximum supported number of CQs.
- max_cqe: Maximum supported number of entries in each CQ.
- max_mr: Maximum supported number of MRs.
- max_pd: Maximum supported number of PDs.
- max_qp_rd_atom: Maximum number of RDMA Read and Atomic operations that can be sent to a QP as the target of the operation.
- max_ee_rd_atom: Maximum number of RDMA Read and Atomic operations that can be sent to an EE context as the target of the operation.
- max_res_rd_atom: Maximum number of for incoming RDMA Read and Atomic operations that can be sent to this RDMA device as the target of the operation.
- max_qp_init_rd_atom: Maximum number of RDMA Read and Atomic operations that can be sent from a QP as the initiator of the operation.
- max_ee_init_rd_atom: Maximum number of RDMA Read and Atomic operations that can be sent from an EE context as the initiator of the operation.
- atomic_cap: Ability of the device to support atomic operations. Can be:
  - IB_ATOMIC_NONE: The RDMA device doesn't guarantee any atomicity at all.
  - IB_ATOMIC_HCA: The RDMA device guarantees atomicity between QPs in the same device.
  - IB_ATOMIC_GLOB: The RDMA device guarantees atomicity between this device and any other component.
- masked_atomic_cap: The ability of the device to support masked atomic operations. Possible values as described in atomic_cap earlier.
- max_ee: Maximum supported number of EE contexts.
- max_rdd: Maximum supported number of RDDs.
- max_mw: Maximum supported number of MWs.
- max_raw_ipv6_qp: Maximum supported number of Raw IPv6 Datagram QPs.
- max_raw_ethy_qp: Maximum supported number of Raw Ethertype Datagram QPs.
- max_mcast_grp: Maximum supported number of multicast groups.
- max_mcast_qp_attach: Maximum supported number of QPs that can be attached to each multicast group.
- max_total_mcast_qp_attach: Maximum number of total QPs that can be attached to any multicast group.
- max_ah: Maximum supported number of AHs.
- max_fmr: Maximum supported number of FMRs.

- max_map_per_fmr: Maximum supported number of map operations which are allowed per FMR.
- max_srq: Maximum supported number of SRQs.
- max_srq_wr: Maximum supported number of Work Requests in each SRQ.
- max_srq_sge: Maximum supported number of scatter/gather elements per Work Request in an SRQ.
- max_fast_reg_page_list_len: Maximum number of page list that can be used when registering an FMR using a Work Request.
- max_pkeys: Maximum supported number of P_Keys.
- local_ca_ack_delay: Local CA ack delay. This value specifies the maximum expected time interval between the local device receiving a message and transmitting the associated ACK or NAK.

The ib_query_port() Method

The ib_query_port() method queries the RDMA device port's attributes. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_query_port(struct ib_device *device,
         u8 port_num, struct ib_port_attr *port_attr);
```

- device: The RDMA device to be queried.
- port_num: The port number to be queried.
- port_attr: A pointer to a structure of an RDMA port attributes which will be filled.

The ib_port_attr Struct

The RDMA port attributes are represented by struct ib_port_attr:

```
struct ib_port_attr {
    enum ib_port_state    state;
    enum ib_mtu   max_mtu;
    enum ib_mtu   active_mtu;
    int           gid_tbl_len;
    u32           port_cap_flags;
    u32           max_msg_sz;
    u32           bad_pkey_cntr;
    u32           qkey_viol_cntr;
    u16           pkey_tbl_len;
    u16           lid;
    u16           sm_lid;
    u8            lmc;
    u8            max_vl_num;
    u8            sm_sl;
    u8            subnet_timeout;
```

```
    u8              init_type_reply;
    u8              active_width;
    u8              active_speed;
    u8              phys_state;
};
```

- `state`: The logical port state. Can be:
  - IB_PORT_NOP: Reserved value.
  - IB_PORT_DOWN: Logical link is down.
  - IB_PORT_INIT: Logical link is initialized. The physical link is up but the Subnet Manager hasn't started to configure the port.
  - IB_PORT_ARMED: Logical link is armed. The physical link is up but the Subnet Manager started, and did not yet complete, configuring the port.
  - IB_PORT_ACTIVE: Logical link is active.
  - IB_PORT_ACTIVE_DEFER: Logical link is active but the physical link is down. The link tries to recover from this state.
- `max_mtu`: The maximum MTU supported by this port. Can be:
  - IB_MTU_256: 256 bytes.
  - IB_MTU_512: 512 bytes.
  - IB_MTU_1024: 1,024 bytes.
  - IB_MTU_2048: 2,048 bytes.
  - IB_MTU_4096: 4,096 bytes.
- `active_mtu`: The actual MTU that this port is configured with. Can be as `max_mtu`, mentioned earlier.
- `gid_tbl_len`: The number of entries in the port's GID table.
- `port_cap_flags`: The port supported capabilities. It is a bitwise OR of the masks:
  - IB_PORT_SM: An indication that the SM that manages the subnet is sending packets from this port.
  - IB_PORT_NOTICE_SUP: An indication that this port supports notices.
  - IB_PORT_TRAP_SUP: An indication that this port supports traps.
  - IB_PORT_OPT_IPD_SUP: An indication that this port supports Inter Packet Delay optional values.
  - IB_PORT_AUTO_MIGR_SUP: An indication that this port supports Automatic Path Migration.
  - IB_PORT_SL_MAP_SUP: An indication that this port supports SL 2 VL mapping table.
  - IB_PORT_MKEY_NVRAM: An indication that this port supports saving the M_Key attributes in Non Volatile RAM.

  - IB_PORT_PKEY_NVRAM: An indication that this port supports saving the P_Key table in Non Volatile RAM.
  - IB_PORT_LED_INFO_SUP: An indication that this port supports turning on and off the LED using management packets.
  - IB_PORT_SM_DISABLED: An indication that there is an SM which isn't active in this port.
  - IB_PORT_SYS_IMAGE_GUID_SUP: An indication that the port supports system image GUID.
  - IB_PORT_PKEY_SW_EXT_PORT_TRAP_SUP: An indication that the SMA on the switch management port will monitor P_Key mismatches on each switch external port.
  - IB_PORT_EXTENDED_SPEEDS_SUP: An indication that the port supports extended speeds (FDR and EDR).
  - IB_PORT_CM_SUP: An indication that this port supports CM.
  - IB_PORT_SNMP_TUNNEL_SUP: An indication that an SNMP tunneling agent is listening on this port.
  - IB_PORT_REINIT_SUP: An indication that this port supports reinitialization of the node.
  - IB_PORT_DEVICE_MGMT_SUP: An indication that this port supports device management.
  - IB_PORT_VENDOR_CLASS_SUP: An indication that a vendor-specific agent is listening on this port.
  - IB_PORT_DR_NOTICE_SUP: An indication that this port supports Direct Route notices.
  - IB_PORT_CAP_MASK_NOTICE_SUP: An indication that this port supports sending a notice if the port's `port_cap_flags` is changed.
  - IB_PORT_BOOT_MGMT_SUP: An indication that a boot manager agent is listening on this port.
  - IB_PORT_LINK_LATENCY_SUP: An indication that this port supports link round trip latency measurement.
  - IB_PORT_CLIENT_REG_SUP: An indication that this port is capable of generating the IB_EVENT_CLIENT_REREGISTER asynchronous event.
- `max_msg_sz:` The maximum supported message size by this port.
- `bad_pkey_cntr:` A counter for the number of bad P_Key from messages that this port received.
- `qkey_viol_cntr:` A counter for the number of Q_Key violations from messages that this port received.
- `pkey_tbl_len`: The number of entries in the port's P_Key table.
- `lid`: The port's Local Identifier (LID), as assigned by the SM.
- `sm_lid`: The LID of the SM.
- `lmc`: LID mask of this port.

- `max_vl_num`: Maximum number of Virtual Lanes supported by this port. Can be:
  - 1: 1 VL is supported: VL0
  - 2: 2 VLs are supported: VL0–VL1
  - 3: 4 VLs are supported: VL0–VL3
  - 4: 8 VLs are supported: VL0–VL7
  - 5: 15 VLs are supported: VL0–VL14
- `sm_sl`: The SL to be used when sending messages to the SM.
- `subnet_timeout`: The maximum expected subnet propagation delay. This duration of time calculation is 4.094*2^subnet_timeout.
- `init_type_reply`: The value that the SM configures before moving the port state to IB_PORT_ARMED or IB_PORT_ACTIVE to specify the type of the initialization performed.
- `active_width`: The port's active width. Can be:
  - IB_WIDTH_1X: Multiple of 1.
  - IB_WIDTH_4X: Multiple of 4.
  - IB_WIDTH_8X: Multiple of 8.
  - IB_WIDTH_12X: Multiple of 12.
- `active_speed`: The port's active speed. Can be:
  - IB_SPEED_SDR: Single Data Rate (SDR): 2.5 Gb/sec, 8/10 bit encoding.
  - IB_SPEED_DDR: Double Data Rate (DDR): 5 Gb/sec, 8/10 bit encoding.
  - IB_SPEED_QDR: Quad Data Rate (DDR): 10 Gb/sec, 8/10 bit encoding.
  - IB_SPEED_FDR10: Fourteen10 Data Rate (FDR10): 10.3125 Gb/sec, 64/66 bit encoding.
  - IB_SPEED_FDR: Fourteen Data Rate (FDR): 14.0625 Gb/sec, 64/66 bit encoding.
  - IB_SPEED_EDR: Enhanced Data Rate (EDR): 25.78125 Gb/sec.
- `phys_state`: The physical port state. There isn't any enumeration for this value.

The rdma_port_get_link_layer() Method

The `rdma_port_get_link_layer()` method returns the link layer of the RDMA device port. It will return the following values:

- IB_LINK_LAYER_UNSPECIFIED: Unspecified value, usually legacy value that indicates that this is an InfiniBand link layer.
- IB_LINK_LAYER_INFINIBAND: Link layer is InfiniBand.
- IB_LINK_LAYER_ETHERNET: Link layer is Ethernet. This indicates that the port supports RDMA Over Converged Ethernet (RoCE).

```
enum rdma_link_layer rdma_port_get_link_layer(struct ib_device *device, u8 port_num);
```

- `device`: The RDMA device to be queried.
- `port_num`: The port number to be queried.

The ib_query_gid() Method

The `ib_query_gid()` method queries the RDMA device port's GID table. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_query_gid(struct ib_device *device, u8 port_num, int index, union ib_gid *gid);
```

- `device`: The RDMA device to be queried.
- `port_num`: The port number to be queried.
- `index`: The index in the GID table to be queried.
- `gid`: A pointer to the GID union to be filled.

The ib_query_pkey() Method

The `ib_query_pkey()` method queries the RDMA device port's P_Key table. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_query_pkey(struct ib_device *device,
        u8 port_num, u16 index, u16 *pkey);
```

- `device`: The RDMA device to be queried.
- `port_num`: The port number to be queried.
- `index`: The index in the P_Key table to be queried.
- `pkey`: A pointer to the P_Key to be filled.

The ib_modify_device() Method

The `ib_modify_device()` method modifies the RDMA device attributes. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_modify_device(struct ib_device *device,
            int device_modify_mask,
            struct ib_device_modify *device_modify);
```

- `device`: The RDMA device to be modified.
- `device_modify_mask`: The device attributes to be changed. It is a bitwise OR of the masks:
  - IB_DEVICE_MODIFY_SYS_IMAGE_GUID: Modifies the system image GUID.
  - IB_DEVICE_MODIFY_NODE_DESC: Modifies the node description.
- `device_modify`: The RDMA attributes to be modified, as described immediately.

The ib_device_modify Struct

The RDMA device attributes are represented by `struct ib_device_modify`:

```
struct ib_device_modify {
    u64    sys_image_guid;
    char    node_desc[64];
};
```

- `sys_image_guid`: A 64-bit value of the system image GUID.
- `node_desc`: A NULL terminated string that describes the node description.

The ib_modify_port() Method

The `ib_modify_port()` method modifies the RDMA device port's attributes. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_modify_port(struct ib_device *device,
           u8 port_num, int port_modify_mask,
           struct ib_port_modify *port_modify);
```

- `device`: The RDMA device to be modified.
- `port_num`: The port number to be modified.
- `port_modify_mask`: The port's attributes to be changed. It is a bitwise OR of the masks:
  - IB_PORT_SHUTDOWN: Moves the port state to IB_PORT_DOWN.
  - IB_PORT_INIT_TYPE: Sets the port InitType value.
  - IB_PORT_RESET_QKEY_CNTR: Resets the port's Q_Key violation counter.
- `port_modify`: The port attributes to be modified, as described in the next section.

The ib_port_modify struct:

The RDMA device attributes are represented by `struct ib_port_modify:`

```
struct ib_port_modify {
    u32    set_port_cap_mask;
    u32    clr_port_cap_mask;
    u8    init_type;
};
```

- `set_port_cap_mask`: The port capabilities bits to be set.
- `clr_port_cap_mask`: The port capabilities bits to be cleared.
- `init_type`: The InitType value to be set.

The ib_find_gid() Method

The `ib_find_gid()` method finds the port number and the index where a specific GID value exists in the GID table. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_find_gid(struct ib_device *device, union ib_gid *gid,
        u8 *port_num, u16 *index);
```

- `device`: The RDMA device to be queried.
- `gid`: A pointer of the GID to search for.
- `port_num`: Will be filled with the port number that this GID exists in.
- `index`: Will be filled with the index in the GID table that this GID exists in.

The ib_find_pkey() Method

The `ib_find_pkey()` method finds the index where a specific P_Key value exists in the P_Key table in a specific port number. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_find_pkey(struct ib_device *device,
         u8 port_num, u16 pkey, u16 *index);
```

- `device`: The RDMA device to be queried.
- `port_num`: The port number to search the P_Key in.
- `pkey`: The P_Key value to search for.
- `index`: The index in the P_Key table that this P_Key exists in.

The rdma_node_get_transport() Method

The `rdma_node_get_transport()` method returns the RDMA transport type of a specific node type. The available transport types can be:

- RDMA_TRANSPORT_IB: Transport is InfiniBand.
- RDMA_TRANSPORT_IWARP: Transport is iWARP.

The rdma_node_get_transport() Method

```
enum rdma_transport_type
rdma_node_get_transport(enum rdma_node_type node_type) __attribute_const__;
```

- `node_type`: The node type. Can be:RDMA_NODE_IB_CA: Node type is an InfiniBand Channel Adapter.
- RDMA_NODE_IB_SWITCH: Node type is an InfiniBand Switch.
- RDMA_NODE_IB_ROUTER: Node type is an InfiniBand Router.
- RDMA_NODE_RNIC: Node type is an RDMA NIC.

The ib_mtu_to_int() Method

The `ib_mtu_to_int()` method returns the number of bytes, as an integer, for MTU enumerations. It will return a positive value on success or –1 on a failure.

```
static inline int ib_mtu_enum_to_int(enum ib_mtu mtu);
```

- `mtu`: Can be an MTU enumeration, as described earlier.

The ib_width_enum_to_int() Method

The `ib_width_enum_to_int()` method returns the number of width multiple, as an integer, for an IB port enumerations. It will return a positive value on success or –1 on a failure.

```
static inline int ib_width_enum_to_int(enum ib_port_width width);
```

- `width`: Can be a port width enumeration, as described earlier.

The ib_rate_to_mult() Method

The `ib_rate_to_mult()` method returns the number of multiple of the base rate of 2.5 Gbit/sec, as an integer, for an IB rate enumerations. It will return a positive value on success or –1 on a failure.

```
int ib_rate_to_mult(enum ib_rate rate) __attribute_const__;
```

- rate: The rate enumeration to be converted. Can be:
  - IB_RATE_PORT_CURRENT: Current port's rate.
  - IB_RATE_2_5_GBPS: Rate of 2.5 Gbit/sec.
  - IB_RATE_5_GBPS: Rate of 5 Gbit/sec.
  - IB_RATE_10_GBPS: Rate of 10 Gbit/sec.
  - IB_RATE_20_GBPS: Rate of 20 Gbit/sec.
  - IB_RATE_30_GBPS: Rate of 30 Gbit/sec.
  - IB_RATE_40_GBPS: Rate of 40 Gbit/sec.
  - IB_RATE_60_GBPS: Rate of 60 Gbit/sec.
  - IB_RATE_80_GBPS: Rate of 80 Gbit/sec.
  - IB_RATE_120_GBPS: Rate of 120 Gbit/sec.
  - IB_RATE_14_GBPS: Rate of 14 Gbit/sec.
  - IB_RATE_56_GBPS: Rate of 56 Gbit/sec.
  - IB_RATE_112_GBPS: Rate of 112 Gbit/sec.
  - IB_RATE_168_GBPS: Rate of 168 Gbit/sec.
  - IB_RATE_25_GBPS: Rate of 25 Gbit/sec.

- IB_RATE_100_GBPS: Rate of 100 Gbit/sec.
- IB_RATE_200_GBPS: Rate of 200 Gbit/sec.
- IB_RATE_300_GBPS: Rate of 300 Gbit/sec.

The ib_rate_to_mbps() Method

The `ib_rate_to_mbps()` method returns the number of Mbit/sec, as an integer, for an IB rate enumerations. It will return a positive value on success or –1 on a failure.

```
int ib_rate_to_mbps(enum ib_rate rate) __attribute_const__;
```

- `rate`: The rate enumeration to be converted, as described earlier.

The ib_rate_to_mbps() Method

The `ib_rate_to_mbps()` method returns the IB rate enumerations for a multiple of the base rate of 2.5 Gbit/sec. It will return a positive value on success or –1 on a failure.

```
enum ib_rate mult_to_ib_rate(int mult) __attribute_const__;
```

- `mult`: The rate multiple to be converted, as described earlier.

Protection Domain (PD)

PD is an RDMA resource that associates QPs and SRQs with MRs and AHs with QPs. One can look at PD as a color, for example: red MR can work with a red QP, and red AH can work with a red QP. Working with green AH with a red QP will result in an error.

The ib_alloc_pd() Method

The `ib_alloc_pd()` method allocates a PD. It will return a pointer to the newly allocated PD on success or an ERR_PTR() which specifies the reason for the failure.

```
struct ib_pd *ib_alloc_pd(struct ib_device *device);
```

- `device`: The RDMA device that the PD will be associated with.

The ib_dealloc_pd() Method

The `ib_dealloc_pd()` method deallocates a PD. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_dealloc_pd(struct ib_pd *pd);
```

- `pd`: The PD to be deallocated.

eXtended Reliable Connected (XRC)

XRC is an IB transport extension that provides better scalability, in the sender side, for Reliable Connected QPs than the original Reliable Transport can provide. Using XRC will decrease the number of QPs between two specific cores: when using RC QPs, for each core, in each machine, there is a QP. When using XRC, there will be one XRC QP in each host. When sending a message, the sender needs to specify the remote SRQ number that will receive the message.

The ib_alloc_xrcd() Method

The `ib_alloc_xrcd()` method allocates an XRC domain. It will return a pointer to the newly created XRC domain on success or an ERR_PTR() which specifies the reason for the failure.

```
struct ib_xrcd *ib_alloc_xrcd(struct ib_device *device);
```

- `device`: The RDMA device that this XRC domain will be allocated on.

The ib_dealloc_xrcd_cq() Method

The `ib_dealloc_xrcd_cq()` method deallocates an XRC domain. It will return 0 on success or the errno value with the reason for the failure:

```
int ib_dealloc_xrcd(struct ib_xrcd *xrcd);
```

- `xrcd`: The XRC domain to be deallocated.

Shared Receive Queue (SRQ)

SRQ is a resource that helps RDMA to be more scalable. Instead of managing the Receive Requests in the Receive Queues of many QPs, it is possible to manage them in a single Receive Queue, which all of them share. This will eliminate starvation in RC QPs or packet drops in unreliable transport types and will help to reduce the total posted Receive Requests, thus reducing the consumed memory. Furthermore, unlike a QP, an SRQ can have a watermark to allow a notification if the number of RRs in the SRQ dropped below a specify value.

The ib_srq_attr Struct

The SRQ attributes are represented by `struct ib_srq_attr`:

```
struct ib_srq_attr {
    u32    max_wr;
    u32    max_sge;
    u32    srq_limit;
};
```

- `max_wr`: The maximum number of outstanding RRs that this SRQ can hold.
- `max_sge`: The maximum number of scatter/gather elements that each RR in the SRQ can hold.
- `srq_limit`: The watermark limit that creates an asynchronous event if the number of RRs in the SRQ dropped below this value.

The ib_create_srq() Method

The `ib_create_srq()` method creates an SRQ. It will return a pointer to the newly created SRQ on success or an ERR_PTR() which specifies the reason for the failure:

```
struct ib_srq *ib_create_srq(struct ib_pd *pd, struct ib_srq_init_attr *srq_init_attr);
```

- `pd`: The PD that this SRQ is being associated with.
- `srq_init_attr`: The attributes that this SRQ will be created with.

The ib_srq_init_attr Struct

The created SRQ attributes are represented by `struct ib_srq_init_attr`:

```
struct ib_srq_init_attr {
    void              (*event_handler)(struct ib_event *, void *);
    void               *srq_context;
    struct ib_srq_attr    attr;
    enum ib_srq_type    srq_type;

    union {
        struct {
            struct ib_xrcd *xrcd;
            struct ib_cq   *cq;
        } xrc;
    } ext;
};
```

- `event_handler`: A pointer to a callback that will be called in case of an affiliated asynchronous event to the SRQ.
- `srq_context`: User-defined context that can be associated with the SRQ.
- `attr`: The SRQ attributes, as described earlier.
- `srq_type`: The type of the SRQ. Can be:
  - IB_SRQT_BASIC: For regular SRQ.
  - IB_SRQT_XRC: For XRC SRQ.
- `ext`: If `srq_type` is IB_SRQT_XRC, specifies the XRC domain or the CQ that this SRQ is associated with.

The ib_modify_srq() Method

The `ib_modify_srq()` method modifies the attributes of the SRQ. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_modify_srq(struct ib_srq *srq, struct ib_srq_attr *srq_attr, enum ib_srq_attr_mask srq_attr_mask);
```

- srq: The SRQ to be modified.
- srq_attr: The SRQ attributes, as described earlier.
- srq_attr_mask: The SRQ attributes to be changed. It is a bitwise OR of the masks:
  - IB_SRQ_MAX_WR: Modify the number of RRs in the SRQ (that is, resize the SRQ). This can be done only if the device supports SRQ resize—that is, the IB_DEVICE_SRQ_RESIZE is set in the device flags.
  - IB_SRQ_LIMIT: Set the value of the SRQ watermark limit.

The ib_query_srq() Method

The ib_query_srq() method queries for the current SRQ attributes. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_query_srq(struct ib_srq *srq, struct ib_srq_attr *srq_attr);
```

- srq: The SRQ to be queried.
- srq_attr: The SRQ attributes, as described earlier.

The ib_destory_srq() Method

The ib_destory_srq() method destroys an SRQ. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_destroy_srq(struct ib_srq *srq);
```

- srq: The SRQ to be destroyed.

The ib_post_srq_recv() Method

The ib_post_srq_recv() method takes a linked list of Receive Requests and adds them to the SRQ for future processing. Every Receive Request is considered outstanding until a Work Completion is generated after its processing. It will return 0 on success or the errno value with the reason for the failure.

```
static inline int ib_post_srq_recv(struct ib_srq *srq, struct ib_recv_wr *recv_wr,
struct ib_recv_wr **bad_recv_wr);
```

- srq: The SRQ that the Receive Requests will be posted to.
- recv_wr: A linked list of Receive Request to be posted.
- bad_recv_wr: If there was an error with the handling of the Receive Requests, this pointer will be filled with the address of the Receive Request that caused this error.

The ib_recv_wr Struct

The Receive Request is represented by `struct ib_recv_wr`:

```
struct ib_recv_wr {
    struct ib_recv_wr          *next;
    u64            wr_id;
    struct ib_sge        *sg_list;
    int            num_sge;
};
```

- `next`: A pointer to the next Receive Request in the list or NULL, if this is the last Receive Request.
- `wr_id`: A 64-bit value that is associated with this Receive Request and will be available in the corresponding Work Completion.
- `sg_list`: The array of the scatter/gather elements, as described in the next section.
- `num_sge`: The number of entries in `sg_list`. The value zero means that the message size that can be saved has zero bytes.

The ib_sge Struct

The scatter/gather element is represented by `struct ib_sge`:

```
struct ib_sge {
    u64    addr;
    u32    length;
    u32    lkey;
};
```

- `addr`: The address of the buffer to access.
- `length`: The length of the address to access.
- `lkey`: The Local Key of the Memory Region that this buffer was registered with.

Address Handle (AH)

AH is an RDMA resource that describes the path from the local port to the remote port of the destination. It is being used for a UD QP.

The ib_ah_attr Struct

The AH attributes are represented by `struct ib_ah_attr`:

```
struct ib_ah_attr {
    struct ib_global_route    grh;
    u16                dlid;
    u8                sl;
    u8                src_path_bits;
    u8                static_rate;
    u8                ah_flags;
    u8                port_num;
};
```

- `grh`: The Global Routing Header attributes that are used for sending messages to another subnet or to a multicast group in the local or remote subnet.
- `dlid`: The destination LID.
- `sl`: The Service Level that this message will use.
- `src_path_bits`: The used source path bits. Relevant if LMC is used in this port.
- `static_rate`: The level of delay that should be done between sending the messages. It is used when sending a message to a remote node that supports a slower message rate than the local node.
- `ah_flags`: The AH flags. It is a bitwise OR of the masks:
  - IB_AH_GRH: GRH is used in this AH.
- `port_num`: The local port number that messages will be sent from.

The ib_create_ah() Method

The `ib_create_ah()` method creates an AH. It will return a pointer to the newly created AH on success or an ERR_PTR() which specifies the reason for the failure.

```
struct ib_ah *ib_create_ah(struct ib_pd *pd, struct ib_ah_attr *ah_attr);
```

- `pd`: The PD that this AH is being associated with.
- `ah_attr`: The attributes that this AH will be created with.

The ib_init_ah_from_wc() Method

The `ib_init_ah_from_wc()` method initializes an AH attribute structure from a Work Completion and a GRH structure. This is being done in order to return a message back for an incoming message of an UD QP. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_init_ah_from_wc(struct ib_device *device, u8 port_num, struct ib_wc *wc,
        struct ib_grh *grh, struct ib_ah_attr *ah_attr);
```

- `device`: The RDMA device that the Work Completion came from and the AH to be created on.
- `port_num`: The port number that the Work Completion came from and the AH will be associated with.
- `wc`: The Work Completion of the incoming message.
- `grh`: The GRH buffer of the incoming message.
- `ah_attr`: The attributes of this AH to be filled.

The ib_create_ah_from_wc() Method

The `ib_create_ah_from_wc()` method creates an AH from a Work Completion and a GRH structure. This is done in order to return a message back for an incoming message of a UD QP. It will return a pointer to the newly created AH on success or an ERR_PTR() which specifies the reason for the failure.

```
struct ib_ah *ib_create_ah_from_wc(struct ib_pd *pd, struct ib_wc *wc, struct ib_grh *grh, u8 port_num);
```

- `pd`: The PD that this AH is being associated with.
- `wc`: The Work Completion of the incoming message.
- `grh`: The GRH buffer of the incoming message.
- `port_num`: The port number that the Work Completion came from and the AH will be associated with.

The ib_modify_ah() Method

The `ib_modify_ah()` method modifies the attributes of the AH. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_modify_ah(struct ib_ah *ah, struct ib_ah_attr *ah_attr);
```

- `ah`: The AH to be modified.
- `ah_attr`: The AH attributes, as described earlier.

The ib_query_ah() Method

The `ib_query_ah()` method queries for the current AH attributes. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_query_ah(struct ib_ah *ah, struct ib_ah_attr *ah_attr);
```

- `ah`: The AH to be queried
- `ah_attr`: The AH attributes, as described earlier.

The ib_destory_ah() Method

The `ib_destory_ah()` method destroys an AH. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_destroy_ah(struct ib_ah *ah);
```

- `ah`: The AH to be destroyed.

Multicast Groups

Multicast groups are means to send a message from one UD QP to many UD QPs. Every UD QP that wants to get this message needs to be attached to a multicast group.

The ib_attach_mcast() Method

The `ib_attach_mcast()` method attaches a UD QP to a multicast group within an RDMA device. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_attach_mcast(struct ib_qp *qp, union ib_gid *gid, u16 lid);
```

- `qp`: A handler of a UD QP to be attached to the multicast group.
- `gid`: The GID of the multicast group that the QP will be added to.
- `lid`: The LID of the multicast group that the QP will be added to.

The ib_detach_mcast() method

The `ib_detach_mcast()` method detaches a UD QP from a multicast group within an RDMA device. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_detach_mcast(struct ib_qp *qp, union ib_gid *gid, u16 lid);
```

- `qp`: A handler of a UD QP to be detached from the multicast group.
- `gid`: The GID of the multicast group that the QP will be removed from.
- `lid`: The LID of the multicast group that the QP will be removed from.

Completion Queue (CQ)

A Work Completion specifies that a corresponding Work Request was completed and provides some information. about it: its status, the used opcode, its size, and so on. A CQ is an object that consists of Work Completions.

The ib_create_cq() Method

The `ib_create_cq()` method creates a CQ. It will return a pointer to the newly created CQ on success or an ERR_PTR() which specifies the reason for the failure.

```
struct ib_cq *ib_create_cq(struct ib_device *device, ib_comp_handler comp_handler,
void (*event_handler)(struct ib_event *, void *), void *cq_context, int cqe, int comp_vector);
```

- `device`: The RDMA device that this CQ is being associated with.
- `comp_handler`: A pointer to a callback that will be called when a completion event occur to the CQ.
- `event_handler`: A pointer to a callback that will be called in case of an affiliated asynchronous event to the CQ.

- `cq_context`: A user-defined context that can be associated with the CQ.
- `cqe`: The requested number of Work Completions that this CQ can hold.
- `comp_vector`: The index of the RDMA device's completion vector to work on. If the IRQ affinity masks of these interrupts are spread across the cores, this value can be used to spread the completion workload over all of the cores.

The ib_resize_cq() Method

The `ib_resize_cq()` method changes the size of the CQ to hold at least the new size, either by increasing the CQ size or decreasing it. Even if the user asks to resize a CQ, its size may not be resized.

```
int ib_resize_cq(struct ib_cq *cq, int cqe);
```

- `cq`: The CQ to be resized. This value cannot be lower than the number of Work Completions that exists in the CQ.
- `cqe`: The requested number of Work Completions that this CQ can hold.

The ib_modify_cq() Method

The `ib_modify_cq()` method changes the moderation parameter for a CQ. A Completion event will be generated if at least a specific number of Work Completion will enter the CQ or a timeout will expire. Using it may help to reduce the number of interrupts that happen to the RDMA device. It will return 0 on success or the -errno value with the reason for the failure.

```
int ib_modify_cq(structib_cq *cq, u16 cq_count, u16 cq_period);
```

- `cq`: The CQ to be modified.
- `cq_count`: The number of Work Completions that will be added to the CQ, since the last Completion event, that will trigger a CQ event.
- `cq_period`: The number of microseconds that will pass, since the last Completion event, that will trigger a CQ event.

The ib_peek_cq() Method

The `ib_peek_cq()` method returns the number of available Work Completions in the CQ. If the number of Work Completions in the CQ is equal to or greater than `wc_cnt`, it will return `wc_cnt`. Otherwise it will return the actual number of the Work Completions in the CQ. If an error occurred, it will return the errno value with the reason for the failure.

```
int ib_peek_cq(structib_cq *cq, intwc_cnt);
```

- `cq`: The CQ to peek.
- `cq_count`: The number of Work Completions that will added to the CQ, since the last Completion event, that will trigger a CQ event.

The ib_req_notify_cq() Method

The `ib_req_notify_cq()` method requests that a Completion event notification be created. Its return value can be:

- 0: This means that the notification was requested successfully. If IB_CQ_REPORT_MISSED_EVENTS was used, then a return value of 0 means that there aren't any missed events.
- Positive value is returned only when IB_CQ_REPORT_MISSED_EVENTS is used and there are missed events. The user should call the `ib_poll_cq()` method in order to read the Work Completions that exist in the CQ.
- Negative value is returned when an error occurred. The –errno value is returned, specifying the reason for the failure.

```
static inline int ib_req_notify_cq(struct ib_cq *cq,
                    enum ib_cq_notify_flags flags);
```

- `cq`: The CQ that this Completion event will be generated for.
- `flags`: Information about the Work Completion that will cause the Completion event notification to be created. Can be one of:
  - IB_CQ_NEXT_COMP: The next Work Completion that will be added to the CQ, after calling this method, will trigger the CQ event.
  - IB_CQ_SOLICITED: The next Solicited Work Completion that will be added to the CQ, after calling this method, will trigger the CQ event.

Both of those values can be bitwise ORed with IB_CQ_REPORT_MISSED_EVENTS in order to request a hint about missed events (that is, when calling this method and there are already Work Completions in this CQ).

The ib_req_ncomp_notif() Method

The `ib_req_ncomp_notif()` method requests that a Completion event notification be created when the number of Work Completions in the CQ equals `wc_cnt`. It will return 0 on success, or the errno value with the reason for the failure.

```
static inline int ib_req_ncomp_notif(struct ib_cq *cq, int wc_cnt);
```

- `cq`: The CQ that this Completion event will be generated for.
- `wc_cnt`: The number of Work Completions that the CQ will hold before a Completion event notification is generated.

The ib_poll_cq() Method

The `ib_poll_cq()` method polls Work Completions from a CQ. It reads the Work Completion from the CQ and removes them. The Work Completions are read in the order they were added to the CQ. It will return 0 or a positive number to indicate the number of Work Completions that were read or the -errno value with the reason for the failure.

```
static inline int ib_poll_cq(struct ib_cq *cq, int num_entries,
              struct ib_wc *wc);
```

- `cq`: The CQ to be polled.

- num_entries: The maximum number of Work Completions to be polled.
- wc: An array that the number of polled Work Completions will be stored in.

The ib_wc Struct

Every Work Completion is represented by struct ib_wc:

```
struct ib_wc {
    u64             wr_id;
    enum ib_wc_status    status;
    enum ib_wc_opcode    opcode;
    u32             vendor_err;
    u32             byte_len;
    struct ib_qp           *qp;
    union {
        __be32        imm_data;
        u32           invalidate_rkey;
    } ex;
    u32             src_qp;
    int             wc_flags;
    u16             pkey_index;
    u16             slid;
    u8             sl;
    u8             dlid_path_bits;
    u8             port_num;
};
```

- wr_id: A 64-bit value that was associated with the corresponding Work Request.
- status: Status of the ended Work Request. Can be:
  - IB_WC_SUCCESS: Operation completed successfully.
  - IB_WC_LOC_LEN_ERR: Local length error. Either sent message is too big to be handled or incoming message is bigger than the available Receive Request.
  - IB_WC_LOC_QP_OP_ERR: Local QP operation error. An internal QP consistency error was detected while processing a Work Request.
  - IB_WC_LOC_EEC_OP_ERR: Local EE context operation error. Deprecated, since RD QPs aren't supported.
  - IB_WC_LOC_PROT_ERR: Local protection error. The protection of the Work Request buffers is invalid to the requested operation.
  - IB_WC_WR_FLUSH_ERR: Work Request flushed error. The Work Request was completed when the QP was in the Error state.
  - IB_WC_MW_BIND_ERR: Memory Windows bind error. The operation of the Memory Windows binding failed.
  - IB_WC_BAD_RESP_ERR: Bad response error. Unexpected transport layer opcode returned by the responder.

  - IB_WC_LOC_ACCESS_ERR: Local access error. A protection error occurred on local buffers during the processing of an RDMA Write With Immediate message.
  - IB_WC_REM_INV_REQ_ERR: Remove invalid request error. The incoming message is invalid.
  - IB_WC_REM_ACCESS_ERR: Remote access error. A protection error occurred to incoming RDMA operation.
  - IB_WC_REM_OP_ERR: Remote operation error. The incoming operation couldn't be completed successfully.
  - IB_WC_RETRY_EXC_ERR: Transport retry counter exceeded. The remote QP didn't send any Ack or Nack, and the timeout was expired after the message retransmission.
  - IB_WC_RNR_RETRY_EXC_ERR: RNR retry exceeded. The RNR NACK return count was exceeded.
  - IB_WC_LOC_RDD_VIOL_ERR: Local RDD violation error. Deprecated, since RD QPs aren't supported.
  - IB_WC_REM_INV_RD_REQ_ERR: Remove invalid RD request. Deprecated, since RD QPs aren't supported.
  - IB_WC_REM_ABORT_ERR: Remote aborted error. The responder aborted the operation.
  - IB_WC_INV_EECN_ERR: Invalid EE Context number. Deprecated, since RD QPs aren't supported.
  - IB_WC_INV_EEC_STATE_ERR: Invalid EE context state error. Deprecated, since RD QPs aren't supported.
  - IB_WC_FATAL_ERR: Fatal error.
  - IB_WC_RESP_TIMEOUT_ERR: Response timeout error.
  - IB_WC_GENERAL_ERR: General error. Other error which isn't covered by one of the earlier errors.
- `opcode`: The operation of the corresponding Work Request that was ended with this Work Completion. Can be:
  - IB_WC_SEND: Send operation was completed in the sender side.
  - IB_WC_RDMA_WRITE: RDMA Write operation was completed in the sender side.
  - IB_WC_RDMA_READ: RDMA Read operation was completed in the sender side.
  - IB_WC_COMP_SWAP: Compare and Swap operation was completed in the sender side.
  - IB_WC_FETCH_ADD: Fetch and Add operation was completed in the sender side.
  - IB_WC_BIND_MW: Memory bind operation was completed in the sender side.
  - IB_WC_LSO: Send operation with Large Send Offload (LSO) was completed in the sender side.
  - IB_WC_LOCAL_INV: Local invalidate operation was completed in the sender side.
  - IB_WC_FAST_REG_MR: Fast registration operation was completed in the sender side.
  - IB_WC_MASKED_COMP_SWAP: Masked Compare and Swap operation was completed in the sender side.

    - IB_WC_MASKED_FETCH_ADD: Masked Fetch and Add operation was completed in the sender side.
    - IB_WC_RECV: Receive Request of an incoming send operation was completed in the receiver side.
    - IB_WC_RECV_RDMA_WITH_IMM: Receive Request of an incoming RDMA Write with immediate operation was completed in the receiver side.
- `vendor_err`: A vendor-specific value that provides extra information about the reason for the error.
- `byte_len`: If this is a Work Completion that was created from the end of a Receive Request, the `byte_len` value indicates the number of bytes that were received.
- `qp`: Handle of the QP that got the Work Completion. It is useful when QPs are associated with an SRQ—this way you can know the handle associated with the QP, that its incoming message consumed the Receive Request from the SRQ.
- `ex.imm_data`: Out Of Band data (32 bits), in network order, that was sent with the message. It is available if IB_WC_WITH_IMM is set in `wc_flags`.
- `ex.invalidate_rkey`: The `rkey` that was invalidated. It is available if IB_WC_WITH_INVALIDATE is set in `wc_flags`.
- `src_qp`: Source QP number. The QP number that sent this message. Only relevant for UD QPs.
- `wc_flags`: Flags that provide information about the Work Completion. It is a bitwise OR of the masks:
    - IB_WC_GRH: Indicator that the message was received has a GRH and the first 40 bytes of the Receive Request buffers contains it. Only relevant for UD QPs.
    - IB_WC_WITH_IMM: Indicator that the received message has immediate data.
    - IB_WC_WITH_INVALIDATE: Indicator that a Send with Invalidate message was received.
    - IB_WC_IP_CSUM_OK: Indicator that the received message passed the IP checksum test done by the RDMA device. This is available only if the RDMA device supports IP checksum offload. It is available if IB_DEVICE_UD_IP_CSUM is set in the device flags.
- `pkey_index`: The P_Key index, relevant only for GSI QPs.
- `slid`: The source LID of the message. Only relevant for UD QPs.
- `sl`: The Service Level of the message. Only relevant for UD QPs.
- `dlid_path_bits`: The destination LID path bits. Only relevant for UD QPs.
- `port_num`: The port number from which the message came in. Only relevant for Direct Route SMPs on switches.

The ib_destory_cq() Method

The `ib_destory_cq()` method destroys a CQ. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_destroy_cq(struct ib_cq *cq);
```

- `cq`: The CQ to be destroyed.

Queue Pair (QP)

QP is a resource that combines two Work Queues together: the Send Queue and the Receive Queue. Each queue acts as a FIFO. WRs that are being posted to each Work Queue will be processed by the order of their arrival. However, there isn't any guarantee about the order between the Queues. This resource is the resource that sends and receives packets.

The ib_qp_cap Struct

The QP's Work Queues sizes are represented by `struct ib_qp_cap`:

```
struct ib_qp_cap {
    u32    max_send_wr;
    u32    max_recv_wr;
    u32    max_send_sge;
    u32    max_recv_sge;
    u32    max_inline_data;
};
```

- `max_send_wr`: The maximum number of outstanding Work Requests that this QP can hold in the Send Queue.
- `max_recv_wr`: The maximum number of outstanding Work Requests that this QP can hold in the Receive Queue. This value is ignored if the QP is associated with an SRQ.
- `max_send_sge`: The maximum number of scatter/gather elements that each Work Request in the Send Queue will be able to hold.
- `max_recv_sge`: The maximum number of scatter/gather elements that each Work Request in the Receive Queue will be able to hold.
- `max_inline_data`: The maximum message size that can be sent inline.

The ib_create_qp() Method

The `ib_create_qp()` method creates a QP. It will return a pointer to the newly created QP on success or an ERR_PTR() which specifies the reason for the failure.

```
struct ib_qp *ib_create_qp(struct ib_pd *pd,
        struct ib_qp_init_attr *qp_init_attr);
```

- `pd`: The PD that this QP is being associated with.
- `qp_init_attr`: The attributes that this QP will be created with.

The ib_qp_init_attr Struct

The created QP attributes are represented by `struct ib_qp_init_attr`:

```
struct ib_qp_init_attr {
    void                  (*event_handler)(struct ib_event *, void *);
    void               *qp_context;
    struct ib_cq           *send_cq;
    struct ib_cq           *recv_cq;
    struct ib_srq           *srq;
    struct ib_xrcd          *xrcd;     /* XRC TGT QPs only */
    struct ib_qp_cap       cap;
    enum ib_sig_type       sq_sig_type;
    enum ib_qp_type        qp_type;
    enum ib_qp_create_flags    create_flags;
    u8                port_num; /* special QP types only */
};
```

- `event_handler`: A pointer to a callback that will be called in case of an affiliated asynchronous event to the QP.
- `qp_context`: User-defined context that can be associated with the QP.
- `send_cq`: A CQ that is being associated with the Send Queue of this QP.
- `recv_cq`: A CQ that is being associated with the Receive Queue of this QP.
- `srq`: A SRQ that is being associated with the Receive Queue of this QP or NULL if the QP isn't associated with an SRQ.
- `xrcd`: An XRC domain that this QP will be associated with. Relevant only if qp_type is IB_QPT_XRC_TGT.
- `cap`: A structure that describes the size of the Send and Receive Queues. This structure is described earlier.
- `sq_sig_type`: The signaling type of the Send Queue. It can be:
  - IB_SIGNAL_ALL_WR: Every posted Send Request to the Send Queue will end with a Work Completion.
  - IB_SIGNAL_REQ_WR: Only posted Send Requests to the Send Queue with an explicit request, i.e. set the IB_SEND_SIGNALED flag—will end with a Work Completion. This is called *selective signaling*.
- `qp_type`: The QP transport type. Can be:
  - IB_QPT_SMI: A Subnet Management Interface QP.
  - IB_QPT_GSI: A General Service Interface QP.
  - IB_QPT_RC: A Reliable Connected QP.
  - IB_QPT_UC: An Unreliable Connected QP.
  - IB_QPT_UD: An Unreliable Datagram QP.
  - IB_QPT_RAW_IPV6: An IPv6 raw datagram QP.

- IB_QPT_RAW_ETHERTYPE: An EtherType raw datagram QP.
- IB_QPT_RAW_PACKET: A raw packet QP.
- IB_QPT_XRC_INI: An XRC-initiator QP.
- IB_QPT_XRC_TGT: An XRC-target QP.

- `create_flags`: QP attributes flags. It is a bitwise OR of the masks:
  - IB_QP_CREATE_IPOIB_UD_LSO: The QP will be used to send IPoIB LSO messages.
  - IB_QP_CREATE_BLOCK_MULTICAST_LOOPBACK: Block loopback multicast packets.
- `port_num`: The RDMA device port number that this QP is associated with. Only relevant when `qp_type` is IB_QPT_SMI or IB_QPT_GS.

The ib_modify_qp() Method

The `ib_modify_qp()` method modifies the attributes of the QP. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_modify_qp(struct ib_qp *qp,
    struct ib_qp_attr *qp_attr,
    int qp_attr_mask);
```

- `qp`: The QP to be modified.
- `qp_attr`: The QP attributes, as described earlier.
- `qp_attr_mask`: The QP attributes to be changed. Each mask specifies the attributes that will be modified in this QP transition, such as specifying which attributes in `qp_attr` will be used. It is a bitwise OR of the masks:
  - IB_QP_STATE: Modifies the QP state, specified in the `qp_state` field.
  - IB_QP_CUR_STATE: Modifies the assumed current QP state, specified in the `cur_qp_state` field.
  - IB_QP_EN_SQD_ASYNC_NOTIFY: Modifies the status of the request for notification when the QP state is SQD.drained, specified in the `en_sqd_async_notify` field.
  - IB_QP_ACCESS_FLAGS: Modifies the allowed incoming Remote operations, specified in the `qp_access_flags` field.
  - IB_QP_PKEY_INDEX: Modifies the index in the P_Key table that this QP is associated with in the primary path, specified in the `pkey_index` field.
  - IB_QP_PORT: Modifies the RDMA device's port number that QP's primary path is associated with, specified in the `port_num` field.
  - IB_QP_QKEY: Modifies the Q-Key of the QP, specified in the `qkey` field.
  - IB_QP_AV: Modifies the Address Vector attributes of the QP, specified in the `ah_attr` field.
  - IB_QP_PATH_MTU: Modifies the MTU of the path, specified in the `path_mtu` field.
  - IB_QP_TIMEOUT: Modifies the timeout to wait before retransmission, specified in the field timeout.

- IB_QP_RETRY_CNT: Modifies the number of retries of the QP for lack of Ack/Nack, specified in the `retry_cnt` field.
- IB_QP_RNR_RETRY: Modifies the number of RNR retry of the QP, specified in the `rq_psn` field.
- IB_QP_RQ_PSN: Modifies the start PSN of the received packets, specified in the `rnr_retry` field.
- IB_QP_MAX_QP_RD_ATOMIC: Modifies the number of RDMA Read and Atomic operations that this QP can process in parallel as an initiator, specified in the `max_rd_atomic` field.
- IB_QP_ALT_PATH: Modifies the alternate path of the QP, specified in the `alt_ah_attr`, `alt_pkey_index`, `alt_port_num`, and `alt_timeout` fields.
- IB_QP_MIN_RNR_TIMER: Modifies the minimum RNR timer that the QP will report to the remote side in the RNR Nak, specified in the `min_rnr_timer` field.
- IB_QP_SQ_PSN: Modifies the start PSN of the sent packets, specified in the `sq_psn` field.
- IB_QP_MAX_DEST_RD_ATOMIC: Modifies the number of RDMA Read and Atomic operations that this QP can process in parallel as an initiator, specified in the `max_dest_rd_atomic` field.
- IB_QP_PATH_MIG_STATE: Modifies the state of the path migration state machine, specified in the `path_mig_state` field.
- IB_QP_CAP: Modifies the size of the Work Queues in the QP (both Send and Receive Queues), specified in the `cap` field.
- IB_QP_DEST_QPN: Modifies the destination QP number, specified in the `dest_qp_num` field.

The ib_qp_attr Struct

The QP attributes are represented by `struct ib_qp_attr`:

```
struct ib_qp_attr {
    enum ib_qp_state    qp_state;
    enum ib_qp_state    cur_qp_state;
    enum ib_mtu        path_mtu;
    enum ib_mig_state    path_mig_state;
    u32            qkey;
    u32            rq_psn;
    u32            sq_psn;
    u32            dest_qp_num;
    int            qp_access_flags;
    struct ib_qp_cap    cap;
    struct ib_ah_attr    ah_attr;
    struct ib_ah_attr    alt_ah_attr;
    u16            pkey_index;
    u16            alt_pkey_index;
    u8            en_sqd_async_notify;
    u8            sq_draining;
    u8            max_rd_atomic;
```

```
    u8              max_dest_rd_atomic;
    u8              min_rnr_timer;
    u8              port_num;
    u8              timeout;
    u8              retry_cnt;
    u8              rnr_retry;
    u8              alt_port_num;
    u8              alt_timeout;
};
```

- `qp_state`: The state to move the QP to. Can be:
  - IB_QPS_RESET: Reset state.
  - IB_QPS_INIT: Initialized state.
  - IB_QPS_RTR: Ready To Receive state.
  - IB_QPS_RTS: Ready To Send state.
  - IB_QPS_SQD: Send Queue Drained state.
  - IB_QPS_SQE: Send Queue Error state.
  - IB_QPS_ERR: Error state.
- `cur_qp_state`: The assumed current state of the QP. Can be like `qp_state`.
- `path_mtu`: The size of the MTU in the path. Can be:
  - IB_MTU_256: 256 bytes.
  - IB_MTU_512: 512 bytes.
  - IB_MTU_1024: 1,024 bytes.
  - IB_MTU_2048: 2,048 bytes.
  - IB_MTU_4096: 4,096 bytes.
- `path_mig_state`: The path migration state machine, used in APM (Automatic Path Migration). Can be:
  - IB_MIG_MIGRATED: Migrated. The state machine of path migration is Migrated (initial state of migration was done).
  - IB_MIG_REARM: Rearm. The state machine of path migration is Rearm (attempt to try to coordinate the remote RC QP to move both local and remote QPs to Armed state).
  - IB_MIG_ARMED: Armed. The state machine of path migration is Armed (both local and remote QPs are ready to perform a path migration).
- `qkey`: The Q_Key of the QP.
- `rq_psn`: The expected PSN of the first packet in the Receive Queue. The value is 24 bits.
- `sq_psn`: The used PSN of the first packet in the Send Queue. The value is 24 bits.
- `dest_qp_num`: The QP number in the remote (destination) side. The value is 24 bits.

- `qp_access_flags`: The allowed incoming RDMA and Atomic operations. It is a bitwise OR of the masks:
  - IB_ACCESS_REMOTE_WRITE: Incoming RDMA Write operations are allowed.
  - IB_ACCESS_REMOTE_READ: Incoming RDMA Read operations are allowed.
  - IB_ACCESS_REMOTE_ATOMIC: Incoming Atomic operations are allowed.
- `cap`: The QP size. The number of Work Requests in the Receive and Send Queues. This can be done only if the device supports QP resize—that is, the IB_DEVICE_RESIZE_MAX_WR is set in the device flags. This structure is described earlier.
- `ah_attr`: Address vector of the primary path of the QP. This structure is described earlier.
- `alt_ah_attr`: Address vector of the alternate path of the QP. This structure is described earlier.
- `pkey_index`: The P_Key index of the primary path that this QP is associated with.
- `alt_pkey_index`: The P_Key index of the alternate path that this QP is associated with.
- `en_sqd_async_notify`: If value isn't zero, request that the asynchronous event callback will be called when the QP will moved to SQE.drained state.
- `sq_draining`: Relevant only for `ib_query_qp()`. If value isn't zero, the QP is in state SQD. drainning (and not SQD.drained).
- `max_rd_atomic`: The number of RDMA Read and Atomic operations that this QP can process in parallel as an initiator.
- `max_dest_rd_atomic`: The number of RDMA Read and Atomic operations that this QP can process in parallel as a destination.
- `min_rnr_timer`: The timeout to wait before resend the message again if the remote side responds with an RNR Nack.
- `port_num`: The RDMA device's Port number that this QP is associated with in the Primary path.
- `timeout`: The timeout to wait before resending the message again if the remote side didn't respond with any Ack or Nack in the primary path. The `timeout` is a 5-bit value, 0 is infinite time, and any other value means that the timeout will be 4.096 * 2 ^ `timeout` usec.
- `retry_cnt`: The number of times to (re)send the message if the remote side didn't respond with any Ack or Nack.
- `rnr_retry`: The number of times to (re)send the message if the remote side answered with an RNR Nack. 3 bits value, 7 means infinite retry. The value can be:
  - IB_RNR_TIMER_655_36: Delay of 655.36 milliseconds.
  - IB_RNR_TIMER_000_01: Delay of 0.01 milliseconds.
  - IB_RNR_TIMER_000_02: Delay of 0.02 milliseconds.
  - IB_RNR_TIMER_000_03: Delay of 0.03 milliseconds.
  - IB_RNR_TIMER_000_04: Delay of 0.04 milliseconds.
  - IB_RNR_TIMER_000_06: Delay of 0.06 milliseconds.
  - IB_RNR_TIMER_000_08: Delay of 0.08 milliseconds.
  - IB_RNR_TIMER_000_12: Delay of 0.12 milliseconds.

- IB_RNR_TIMER_000_16: Delay of 0.16 milliseconds.
- IB_RNR_TIMER_000_24: Delay of 0.24 milliseconds.
- IB_RNR_TIMER_000_32: Delay of 0.32 milliseconds.
- IB_RNR_TIMER_000_48: Delay of 0.48 milliseconds.
- IB_RNR_TIMER_000_64: Delay of 0.64 milliseconds.
- IB_RNR_TIMER_000_96: Delay of 0.96 milliseconds.
- IB_RNR_TIMER_001_28: Delay of 1.28 milliseconds.
- IB_RNR_TIMER_001_92: Delay of 1.92 milliseconds.
- IB_RNR_TIMER_002_56: Delay of 2.56 milliseconds.
- IB_RNR_TIMER_003_84: Delay of 3.84 milliseconds.
- IB_RNR_TIMER_005_12: Delay of 5.12 milliseconds.
- IB_RNR_TIMER_007_68: Delay of 7.68 milliseconds.
- IB_RNR_TIMER_010_24: Delay of 10.24 milliseconds.
- IB_RNR_TIMER_015_36: Delay of 15.36 milliseconds.
- IB_RNR_TIMER_020_48: Delay of 20.48 milliseconds.
- IB_RNR_TIMER_030_72: Delay of 30.72 milliseconds.
- IB_RNR_TIMER_040_96: Delay of 40.96 milliseconds.
- IB_RNR_TIMER_061_44: Delay of 61.44 milliseconds.
- IB_RNR_TIMER_081_92: Delay of 81.92 milliseconds.
- IB_RNR_TIMER_122_88: Delay of 122.88 milliseconds.
- IB_RNR_TIMER_163_84: Delay of 163.84 milliseconds.
- IB_RNR_TIMER_245_76: Delay of 245.76 milliseconds.
- IB_RNR_TIMER_327_68: Delay of 327.86 milliseconds.
- IB_RNR_TIMER_491_52: Delay of 391.52 milliseconds.

- `alt_port_num`: The RDMA device's Port number that this QP is associated with in the alternate path.
- `alt_timeout`: The timeout to wait before resend the message again if the remote side didn't respond with any Ack or Nack in the alternate path. 5-bit value, 0 is infinite time, and any other value means that the timeout will be 4.096 * 2 ^ `timeout` usec.

The ib_query_qp() Method

The `ib_query_qp()` method queries for the current QP attributes. Some of the attributes in `qp_attr` may change in subsequent calls to `ib_query_qp()` the state fields. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_query_qp(struct ib_qp *qp, struct ib_qp_attr *qp_attr, int qp_attr_mask,
struct ib_qp_init_attr *qp_init_attr);
```

- qp: The QP to be queried.
- qp_attr: The QP attributes, as described earlier.
- qp_attr_mask: The mask of the mandatory attributes to query. Low-level drivers can use it as a hint for the fields to be queried, but they may also ignore it as well and fill the whole structure.
- qp_init_attr: The QP init attributes, as described earlier.

The ib_destory_qp() method destroys a QP. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_destroy_qp(struct ib_qp *qp);
```

- qp: The QP to be destroyed.

The ib_open_qp() Method

The ib_open_qp() method obtains a reference to an existing sharable QP among multiple processes. The process that created the QP may exit, allowing transfer of the ownership of the QP to another process. It will return a pointer to the sharable QP on success or an ERR_PTR() which specifies the reason for the failure.

```
struct ib_qp *ib_open_qp(struct ib_xrcd *xrcd, struct ib_qp_open_attr *qp_open_attr);
```

- xrcd: The XRC domain that the QP will be associated with.
- qp_open_attr: The attributes of the existing QP to be opened.

The ib_qp_open_attr Struct

The shared QP attributes are represented by struct ib_qp_open_attr:

```
struct ib_qp_open_attr {
    void                (*event_handler)(struct ib_event *, void *);
    void               *qp_context;
    u32             qp_num;
    enum ib_qp_type    qp_type;
};
```

- event_handler: A pointer to a callback that will be called in case of an affiliated asynchronous event to the QP.
- qp_context: User-defined context that can be associated with the QP.
- qp_num: The QP number that this QP will open.
- qp_type: QP transport type. Only IB_QPT_XRC_TGT is supported.

The ib_close_qp() Method

The ib_close_qp() method releases an external reference to a QP. The underlying shared QP won't be destroyed until all internal references that were acquired by the ib_open_qp() method are released. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_close_qp(struct ib_qp *qp);
```

- qp: The QP to be closed.

The ib_post_recv() Method

The ib_post_recv() method takes a linked list of Receive Requests and adds them to the Receive Queue for future processing. Every Receive Request is considered outstanding until a Work Completion is generated after its processing. It will return 0 on success or the errno value with the reason for the failure.

```
static inline int ib_post_recv(struct ib_qp *qp, struct ib_recv_wr *recv_wr, struct ib_recv_wr
**bad_recv_wr);
```

- qp: The QP that the Receive Requests will be posted to.
- recv_wr: A linked list of Receive Request to be posted.
- bad_recv_wr: If there was an error with the handling of the Receive Requests, this pointer will be filled with the address of the Receive Request that caused this error.

The ib_post_send() Method

The ib_post_send() method takes a linked list of Send Requests as an argument and adds them to the Send Queue for future processing. Every Send Request is considered outstanding until a Work Completion is generated after its processing. It will return 0 on success or the errno value with the reason for the failure.

```
static inline int ib_post_send(struct ib_qp *qp, struct ib_send_wr *send_wr, struct ib_send_wr
**bad_send_wr);
```

- qp: The QP that the Send Requests will be posted to.
- send_wr: A linked list of Send Requests to be posted.
- bad_send_wr: If there was an error with the handling of the Send Requests, this pointer will be filled with the address of the Send Request that caused this error.

The ib_send_wr Struct

The Send Request is represented by struct ib_send_wr:

```
struct ib_send_wr {
    struct ib_send_wr          *next;
    u64            wr_id;
    struct ib_sge        *sg_list;
    int            num_sge;
    enum ib_wr_opcode    opcode;
    int            send_flags;
    union {
        __be32        imm_data;
        u32        invalidate_rkey;
    } ex;
    union {
```

```
        struct {
            u64    remote_addr;
            u32    rkey;
        } rdma;
        struct {
            u64    remote_addr;
            u64    compare_add;
            u64    swap;
            u64    compare_add_mask;
            u64    swap_mask;
            u32    rkey;
        } atomic;
        struct {
            struct ib_ah     *ah;
            void           *header;
            int             hlen;
            int             mss;
            u32        remote_qpn;
            u32        remote_qkey;
            u16        pkey_index; /* valid for GSI only */
            u8        port_num;   /* valid for DR SMPs on switch only */
        } ud;
        struct {
            u64                iova_start;
            struct ib_fast_reg_page_list       *page_list;
            unsigned int            page_shift;
            unsigned int            page_list_len;
            u32                length;
            int                access_flags;
            u32                rkey;
        } fast_reg;
        struct {
            struct ib_mw                    *mw;
            /* The new rkey for the memory window. */
            u32                              rkey;
            struct ib_mw_bind_info           bind_info;
        } bind_mw;
    } wr;
    u32            xrc_remote_srq_num;    /* XRC TGT QPs only */
};
```

- `next`: A pointer to the next Send Request in the list or NULL, if this is the last Send Request.
- `wr_id`: 64-bit value that is associated with this Send Request and will be available in the corresponding Work Completion.
- `sg_list`: The array of the scatter/gather elements. As described earlier.
- `num_sge`: The number of entries in `sg_list`. The value zero means that the message size is zero bytes.

- opcode: The operation to perform. This affects the way that data is being transferred, the direction of it, and whether a Receive Request will be consumed in the remote side and which fields in the Send Request (`send_wr`) will be used. Can be:
  - IB_WR_RDMA_WRITE: RDMA Write operation.
  - IB_WR_RDMA_WRITE_WITH_IMM: RDMA Write with immediate operation.
  - IB_WR_SEND: Send operation.
  - IB_WR_SEND_WITH_IMM: Send with immediate operation.
  - IB_WR_RDMA_READ: RDMA Read operation.
  - IB_WR_ATOMIC_CMP_AND_SWP: Compare and Swap operation.
  - IB_WR_ATOMIC_FETCH_AND_ADD: Fetch and Add operation.
- IB_WR_LSO: Send an IPoIB message with LSO (let the RDMA device fragment the big SKBs to multiple MSS-sized packets).LSO is an optimization feature which allows to use large packets by reducing CPU overhead.
  - IB_WR_SEND_WITH_INV: Send with invalidate operation.
  - IB_WR_RDMA_READ_WITH_INV: RDMA Read with invalidate operation.
  - IB_WR_LOCAL_INV: Local invalidate operation.
  - IB_WR_FAST_REG_MR: Fast MR registration operation.
  - IB_WR_MASKED_ATOMIC_CMP_AND_SWP: Masked Compare and Swap operation.
  - IB_WR_MASKED_ATOMIC_FETCH_AND_ADD: Masked Fetch and Add operation.
  - IB_WR_BIND_MW: Memory bind operation.
- `send_flags`: Extra attributes for the Send Request. It is a bitwise OR of the masks:
  - IB_SEND_FENCE: Before performing this operation, wait until the processing of prior Send Requests has ended.
  - IB_SEND_SIGNALED: If the QP was created with selective signaling, when the processing of this Send Request is ended, a Work Completion will be generated.
  - IB_SEND_SOLICITED: Mark that a Solicited event will be created in the remote side.
  - IB_SEND_INLINE: Post this Send Request as inline—that is, let the low-level driver read the memory buffers in if `sg_list` instead of the RDMA device; this may increase the latency.
  - IB_SEND_IP_CSUM: Send an IPoIB message and calculate the IP checksum in HW (checksum offload).
- `ex.imm_data`: The immediate data to send. This value is relevant if `opcode` is IB_WR_SEND_WITH_IMM or IB_WR_RDMA_WRITE_WITH_IMM.
- `ex.invalidate_rkey`: The `rkey` to be invalidated. This value is relevant if opcode is IB_WR_SEND_WITH_INV.

The following union is relevant if opcode is IB_WR_RDMA_WRITE, IB_WR_RDMA_WRITE_WITH_IMM, or IB_WR_RDMA_READ:

- `wr.rdma.remote_addr`: The remote address that this Send Request is going to access.
- `wr.rdma.rkey`: The Remote Key (`rkey`) of the MR that this Send Request is going to access.

The following union is relevant if `opcode` is IB_WR_ATOMIC_CMP_AND_SWP, IB_WR_ATOMIC_FETCH_AND_ADD,IB_WR_MASKED_ATOMIC_CMP_AND_SWP, or IB_WR_MASKED_ATOMIC_FETCH_AND_ADD:

- `wr.atomic.remote_addr`: The remote address that this Send Request is going to access.
- `wr.atomic.compare_add`: If opcode is IB_WR_ATOMIC_FETCH_AND_ADD*, this is the value to add to the content of `remote_addr`. Otherwise, this is the value to compare the content of `remote_addr` with.
- `wr.atomic.swap`: The value to place in `remote_addr` if the value in it is equal to `compare_add`. This value is relevant if opcode is IB_WR_ATOMIC_CMP_AND_SWP or IB_WR_MASKED_ATOMIC_CMP_AND_SWP.
- `wr.atomic.compare_add_mask`: If opcode is IB_WR_MASKED_ATOMIC_FETCH_AND_ADD, this is the mask of the values to change when adding the value of `compare_add` to the content of `remote_addr`. Otherwise, this is the mask to use on the content of `remote_addr` when comparing it with swap.
- `wr.atomic.swap_mask`: This is the mask of the value in the content of `remote_addr` to change. Relevant only if opcode is IB_WR_MASKED_ATOMIC_CMP_AND_SWP.
- `wr.atomic.rkey`: The `rkey` of the MR that this Send Request is going to access.

The following union is relevant if the QP type that this Send Request is being posted to is UD:

- `wr.ud.ah`: The address handle that describes the path to the target node(s).
- `wr.ud.header`: A pointer that contains the header. Relevant if opcode is IB_WR_LSO.
- `wr.ud.hlen`: The length of `wr.ud.header`. Relevant if opcode is IB_WR_LSO.
- `wr.ud.mss`: The Maximum Segment Size that the message will be fragmented to. Relevant if opcode is IB_WR_LSO.
- `wr.ud.remote_qpn`: The remote QP number to send the message to. The enumeration IB_MULTICAST_QPN should be used if sending this message to a multicast group.
- `wr.ud.remote_qkey`: The remote Q_Key value to use. If the MSB of this value is set, then the value of the Q_Key will be taken from the QP attributes.
- `wr.ud.pkey_index`: The P_Key index that the message will be sent with. Relevant if QP type is IB_QPT_GSI.
- `wr.ud.port_num`: The port number that the message will be sent from. Relevant for Direct Route SMP on a switch.

The following union is relevant if opcode is IB_WR_FAST_REG_MR:

- `wr.fast_reg.iova_start`: I/O Virtual Address of the newly created FMR.
- `wr.fast_reg.page_list`: List of pages to allocate to map in the FMR.
- `wr.fast_reg.page_shift`: Log 2 of size of “pages” to be mapped.
- `wr.fast_reg.page_list_len`: The number of pages in `page_list`.

- `wr.fast_reg.length`: The size, in bytes, of the FMR.
- `wr.fast_reg.access_flags`: The allowed operations on this FMR.
- `wr.fast_reg.rkey`: The value of the remote key to be assigned to the FMR.

The following union is relevant if opcode is IB_WR_BIND_MW:

- `wr.bind_mw.mw`: The MW to be bounded.
- `wr.bind_mw.rkey`: The value of the remote key to be assigned to the MW.
- `wr.bind_mw.bind_info`: The bind attributes, as explained in the next section.

The following member is relevant if the QP type that this Send Request is being posted to is XRCTGT:

- `xrc_remote_srq_num`: The remote SRQ that will receive the messages.

The ib_mw_bind_info Struct

The MW binding attributes for both MW type 1 and type 2 are represented by `struct ib_mw_bind_info`.

```
struct ib_mw_bind_info {
    struct ib_mr       *mr;
    u64        addr;
    u64        length;
    int        mw_access_flags;
};
```

- `mr`: A Memory Region that this Memory Window will be bounded to.
- `addr`: The address where the Memory Window will start from.
- `length`: The length, in bytes, of the Memory Window.
- `mw_access_flags`: The allowed incoming RDMA and Atomic operations. It is a bitwise OR of the masks:
  - IB_ACCESS_REMOTE_WRITE: Incoming RDMA Write operations are allowed.
  - IB_ACCESS_REMOTE_READ: Incoming RDMA Read operations are allowed.
  - IB_ACCESS_REMOTE_ATOMIC: Incoming Atomic operations are allowed.

Memory Windows (MW)

Memory Windows are used as a lightweight operation to change the allowed permission of incoming remote operations and invalidate them.

The ib_alloc_mw() Method

The `ib_alloc_mw()` method allocates a Memory Window. It will return a pointer to the newly allocated MW on success or an ERR_PTR() which specifies the reason for the failure.

```
struct ib_mw *ib_alloc_mw(struct ib_pd *pd, enum ib_mw_type type);
```

- pd: The PD that this MW is being associated with.
- type: The type of the Memory Window. Can be:
  - IB_MW_TYPE_1: MW that can be bounded using a verb and supports only association of a PD.
  - IB_MW_TYPE_2: MW that can be bounded using Work Request and supports association of a QP number only or a QP number and a PD.

The ib_bind_mw() Method

The ib_bind_mw() method binds a Memory Window to a specified Memory Region with a specific address, size, and remote permissions. If there isn't any immediate error, the rkey of the MW will be updated to the new value, but the bind operation may still fail asynchronously (and end with completion with error). It will return 0 on success or the errno value with the reason for the failure.

```
static inline int ib_bind_mw(struct ib_qp *qp, struct ib_mw *mw, struct ib_mw_bind *mw_bind);
```

- qp: The QP that the bind WR will be posted to.
- mw: The MW to bind.
- mw_bind: The bind attributes, as explained next.

The ib_mw_bind Struct

The MW binding attributes for type 1 MW are represented by struct ib_mw_bind.

```
struct ib_mw_bind {
    u64                    wr_id;
    int                    send_flags;
    struct ib_mw_bind_info bind_info;
};
```

- wr_id: A 64-bit value that is associated with this bind Send Request The value of Work Request id (wr_id) will be available in the corresponding Work Completion.
- send_flags: Extra attribute for the bind Send Request, as explained earlier. Only IB_SEND_FENCE and IB_SEND_SIGNALED are supported here.
- bind_info: More attributes for the bind operation. As explained earlier.

The ib_dealloc_mw() Method

The ib_dealloc_mw() method deallocates an MW. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_dealloc_mw(struct ib_mw *mw);
```

- mw: The MW to be deallocated.

Memory Region (MR)

Every memory buffer that is being accessed by the RDMA device needs to be registered. During the registration process, the memory will be pinned (prevented from being swapped out), and the memory translation information (from virtual addresses ➤ physical addresses) will be saved in the RDMA device. After the registration, every Memory Region has two keys: one for local access and one for remote access. Those keys will be used when specifying those memory buffers in Work Requests.

The ib_get_dma_mr() Method

The `ib_get_dma_mr()` method returns a Memory Region for system memory that is usable for DMA. Creating this MR isn't enough, and the `ib_dma_*()` methods below are needed in order to create or destroy addresses that the `lkey` and `rkey` of this MR will be used with. It will return a pointer to the newly allocated MR on success or an ERR_PTR() which specifies the reason for the failure.

```
struct ib_mr *ib_get_dma_mr(struct ib_pd *pd, int mr_access_flags);
```

- `pd`: The PD that this MR is being associated with.
- `mr_access_flags`: The allowed operations on this MR. Local Write is always supported in this MR. It is a bitwise OR of the masks:
  - IB_ACCESS_LOCAL_WRITE: Local write to this Memory Region is allowed.
  - IB_ACCESS_REMOTE_WRITE: Incoming RDMA Write operations to this Memory Region are allowed.
  - IB_ACCESS_REMOTE_READ: Incoming RDMA Read operations to this Memory Region are allowed.
  - IB_ACCESS_REMOTE_ATOMIC: Incoming Atomic operations to this Memory Region are allowed.
  - IB_ACCESS_MW_BIND: MW bind to this Memory Region is allowed.
  - IB_ZERO_BASED: Indication that the Virtual address is zero based.

The ib_dma_mapping_error() Method

The `ib_dma_mapping_error()` method checks if the DMA address that was returned from `ib_dma_*()` failed. It will return a non-zero value if there was any failure and zero if the operation finished successfully.

```
static inline int ib_dma_mapping_error(struct ib_device *dev, u64 dma_addr);
```

- `dev`: The RDMA device for which the DMA address was created by using an `ib_dma_*()` method.
- `dma_addr`: The DMA address to verify.

The ib_dma_map_single() Method

The `ib_dma_map_single()` method maps a kernel virtual address to a DMA address. It will return a DMA address that needed to be checked with the `ib_dma_mapping_error()` method for errors:

```
static inline u64 ib_dma_map_single(struct ib_device *dev, void *cpu_addr, size_t size, enum dma_
data_direction direction);
```

- dev: The RDMA device on which the DMA address will be created.
- cpu_addr: The kernel virtual address to map for DMA.
- size: The size, in bytes, of the region to map.
- direction: The direction of the DMA. Can be:
  - DMA_TO_DEVICE: DMA from the main memory to the device.
  - DMA_FROM_DEVICE: DMA from the device to main memory.
  - DMA_BIDIRECTIONAL: DMA from the main memory to the device or from the device to main memory.

The ib_dma_unmap_single() Method

The ib_dma_unmap_single() method unmaps a DMA mapping that was assigned using ib_dma_map_single():

```
static inline void ib_dma_unmap_single(struct ib_device *dev, u64 addr, size_t size, enum dma_data_
direction direction);
```

- dev: The RDMA device on which the DMA address was created.
- addr: The DMA address to unmap.
- size: The size, in bytes, of the region to unmap. This value must be the same value that was used in the ib_dma_map_single() method.
- direction: The direction of the DMA. This value must be the same value that was used in the ib_dma_map_single() method.

The ib_dma_map_single_attrs() Method

The ib_dma_map_single_attrs() method maps a kernel virtual address to a DMA address according to a DMA attributes. It will return a DMA address that is needed to be checked with the ib_dma_mapping_error() method for errors.

```
static inline u64 ib_dma_map_single_attrs(struct ib_device *dev, void *cpu_addr, size_t size, enum
dma_data_direction direction, struct dma_attrs *attrs);
```

- dev: The RDMA device on which the DMA address will be created.
- cpu_addr: The kernel virtual address to map for DMA.
- size: The size, in bytes, of the region to map.
- direction: The direction of the DMA. As described earlier.
- attrs: The DMA attributes for the mapping. If this value is NULL, this method behaves like the ib_dma_map_single() method.

The ib_dma_unmap_single_attrs() Method

The ib_dma_unmap_single_attrs() method unmaps a DMA mapping that was assigned using the ib_dma_map_single_attrs() method:

```
static inline void ib_dma_unmap_single_attrs(struct ib_device *dev, u64 addr, size_t size,
enum dma_data_direction direction, struct dma_attrs *attrs);
```

- dev: The RDMA device on which the DMA address was created.
- addr: The DMA address to unmap.
- size: The size, in bytes, of the region to unmap. This value must be the same value that was used in the ib_dma_map_single_attrs() method.
- direction: The direction of the DMA. This value must be the same value that was used in the ib_dma_map_single_attrs() method.
- attrs: The DMA attributes of the mapping. This value must be the same value that was used in the ib_dma_map_single_attrs() method. If this value is NULL, this method behaves like the ib_dma_unmap_single() method.

The ib_dma_map_page() Method

The ib_dma_map_page() method maps a physical page to a DMA address. It will return a DMA address that needs to be checked with the ib_dma_mapping_error() method for errors:

```
static inline u64 ib_dma_map_page(struct ib_device *dev, struct page *page, unsigned long offset,
size_t size, enum dma_data_direction direction);
```

- dev: The RDMA device on which the DMA address will be created.
- page: The physical page address to map for DMA.
- offset: The offset within the page that the registration will start from.
- size: The size, in bytes, of the region.
- direction: The direction of the DMA. As described earlier.

The ib_dma_unmap_page() Method

The ib_dma_unmap_page() method unmaps a DMA mapping that was assigned using the ib_dma_map_page() method:

```
static inline void ib_dma_unmap_page(struct ib_device *dev, u64 addr, size_t size, enum dma_data_
direction direction);
```

- dev: The RDMA device on which the DMA address was created.
- addr: The DMA address to unmap.
- size: The size, in bytes, of the region to unmap. This value must be the same value that was used in the ib_dma_map_page() method.
- direction: The direction of the DMA. This value must be the same value that was used in the ib_dma_map_page() method.

The ib_dma_map_sg() Method

The `ib_dma_map_sg()` method maps a scatter/gather list to a DMA address. It will return a non-zero value on success and 0 on a failure.

```
static inline int ib_dma_map_sg(struct ib_device *dev, struct scatterlist *sg, int nents, enum dma_
data_direction direction);
```

- `dev`: The RDMA device on which the DMA address will be created.
- `sg`: An array of the scatter/gather entries to map.
- `nents`: The number of scatter/gather entries in `sg`.
- `direction`: The direction of the DMA. As described earlier.

The ib_dma_unmap_sg() Method

The `ib_dma_unmap_sg()` method unmaps a DMA mapping that was assigned using the `ib_dma_map_sg()` method:

```
static inline void ib_dma_unmap_sg(struct ib_device *dev, struct scatterlist *sg, int nents, enum
dma_data_direction direction);
```

- `dev`: The RDMA device on which the DMA address was created.
- `sg`: An array of the scatter/gather entries to unmap. This value must be the same value that was used in the `ib_dma_map_sg()` method.
- `nents`: The number of scatter/gather entries in `sg`. This value must be the same value that was used in the `ib_dma_map_sg()` method.
- `direction`: The direction of the DMA. This value must be the same value that was used in the `ib_dma_map_sg()` method.

The ib_dma_map_sg_attr() Method

The `ib_dma_map_sg_attr()` method maps a scatter/gather list to a DMA address according to a DMA attributes. It will return a non-zero value on success and 0 on a failure.

```
static inline int ib_dma_map_sg_attrs(struct ib_device *dev, struct scatterlist *sg, int nents, enum
dma_data_direction direction, struct dma_attrs *attrs);
```

- `dev`: The RDMA device on which the DMA address will be created.
- `sg`: An array of the scatter/gather entries to map.
- `nents`: The number of scatter/gather entries in `sg`.
- `direction`: The direction of the DMA. As described earlier.
- `attrs`: The DMA attributes for the mapping. If this value is NULL, this method behaves like the `ib_dma_map_sg()` method.

The ib_dma_unmap_sg() Method

The ib_dma_unmap_sg() method unmaps a DMA mapping that was done using the ib_dma_map_sg() method:

```
static inline void ib_dma_unmap_sg_attrs(struct ib_device *dev, struct scatterlist *sg, int nents,
enum dma_data_direction direction, struct dma_attrs *attrs);
```

- dev: The RDMA device on which the DMA address was created.
- sg: An array of the scatter/gather entries to unmap. This value must be the same value that was used in the ib_dma_map_sg_attrs() method.
- nents: The number of scatter/gather entries in sg. This value must be the same value that was used in the ib_dma_map_sg_attrs() method.
- direction: The direction of the DMA. This value must be the same value that was used in the ib_dma_map_sg_attrs() method.
- attrs: The DMA attributes of the mapping. This value must be the same value that was used in the ib_dma_map_sg_attrs() method. If this value is NULL, this method behaves like the ib_dma_unmap_sg() method.

The ib_sg_dma_address() Method

The ib_sg_dma_address() method returns the DMA address from a scatter/gather entry.

```
static inline u64 ib_sg_dma_address(struct ib_device *dev, struct scatterlist *sg);
```

- dev: The RDMA device on which the DMA address was created.
- sg: A scatter/gather entry.

The ib_sg_dma_len() Method

The ib_sg_dma_len() method returns the DMA length from a scatter/gather entry.

```
static inline unsigned int ib_sg_dma_len(struct ib_device *dev, struct scatterlist *sg);
```

- dev: The RDMA device on which the DMA address was created.
- sg: A scatter/gather entry.

The ib_dma_sync_single_for_cpu() Method

The ib_dma_sync_single_for_cpu() method transfers a DMA region ownership to the CPU. This method must be called before the CPU accesses a DMA-mapped buffer in order to read or modify its content, and prevents the device from accessing it:

```
static inline void ib_dma_sync_single_for_cpu(struct ib_device *dev, u64 addr, size_t size,
enum dma_data_direction dir);
```

- dev: The RDMA device on which the DMA address was created.
- addr: The DMA address to sync.

- `size`: The size, in bytes, of the region.
- `direction`: The direction of the DMA. As described earlier.

The ib_dma_sync_single_for_device() Method

The `ib_dma_sync_single_for_device()` method transfers a DMA region ownership to the device. This method must be called before the device can access a DMA-mapped buffer again after the `ib_dma_sync_single_for_cpu()` method was called.

```
static inline void ib_dma_sync_single_for_device(struct ib_device *dev, u64 addr, size_t size, enum
dma_data_direction dir);
```

- `dev`: The RDMA device on which the DMA address was created.
- `addr`: The DMA address to sync.
- `size`: The size, in bytes, of the region.
- `direction`: The direction of the DMA. As described earlier.

The ib_dma_alloc_coherent() Method

The `ib_dma_alloc_coherent()` method allocates a memory block that can be accessible by the CPU and maps it for DMA. It will return the virtual address that the CPU can access on success or NULL in case of a failure:

```
static inline void *ib_dma_alloc_coherent(struct ib_device *dev, size_t size, u64 *dma_handle, gfp_t flag);
```

- `dev`: The RDMA device on which the DMA address will be created.
- `size`: The size, in bytes, of the memory to allocate and map.
- `direction`: The direction of the DMA. As described earlier.
- `dma_handle`: A pointer that will be filled with the DMA address of the region, if the allocation succeeds.
- `flag`: Memory allocation flags. Can be:
  - GFP_KERNEL: To allow blocking (not in interrupt, not holding SMP locks).
  - GFP_ATOMIC: Prevent blocking.

The ib_dma_free_coherent() method

The `ib_dma_free_coherent()` method frees a memory block that was allocated using the `ib_dma_alloc_coherent()` method:

```
static inline void ib_dma_free_coherent(struct ib_device *dev, size_t size, void *cpu_addr,
u64 dma_handle);
```

- `dev`: The RDMA device on which the DMA address was created.
- `size`: The size, in bytes, of the memory region. This value must be the same value that was used in the `ib_dma_alloc_coherent()` method.

- cpu_addr: The CPU memory address to free. This value must be the value that was returned by the ib_dma_alloc_coherent() method.
- dma_handle: The DMA address to free. This value must be the value that was returned by the ib_dma_alloc_coherent() method.

The ib_reg_phys_mr() Method

The ib_reg_phys_mr() method takes a set of physical pages, register them and prepare a virtual address that can be accessed by an RDMA device. It will return a pointer to the newly allocated MR on success or an ERR_PTR(), which specifies the reason for the failure.

```
struct ib_mr *ib_reg_phys_mr(struct ib_pd *pd, struct ib_phys_buf *phys_buf_array, int num_phys_buf,
int mr_access_flags, u64 *iova_start);
```

- pd: The PD that this MR is being associated with.
- phys_buf_array: An array of physical buffers to use in the Memory Region.
- num_phys_buf: The number of physical buffers in phys_buf_array.
- mr_access_flags: The allowed operations on this MR. As specified earlier.
- iova_start: A pointer to the requested I/O Virtual Address to be associated with the Region, which is allowed to begin anywhere within the first physical buffer. The RDMA device will set this value with the actual I/O virtual address of the Region. This value may be different from the requested one.

The ib_phys_buf Struct

The physical buffer is represented by struct ib_phys_buf.

```
struct ib_phys_buf {
    u64      addr;
    u64      size;
};
```

- addr: The physical address of the buffer.
- size: The size of the buffer.

The ib_rereg_phys_mr() Method

The ib_rereg_phys_mr() method modifies the attributes of an existing Memory Region. This method can be thought of as a call to the ib_dereg_mr() method, which was followed by a call to the ib_reg_phys_mr() method. Where possible, resources are reused instead of being deallocated and reallocated. It will return 0 on success or the errno value with the reason for the failure:

```
int ib_rereg_phys_mr(struct ib_mr *mr, int mr_rereg_mask, struct ib_pd *pd, struct ib_phys_buf
*phys_buf_array, int num_phys_buf, int mr_access_flags, u64 *iova_start);
```

- mr: The Memory Region to be reregistered.
- mr_rereg_mask: The Memory Region attributes to be changed. It is a bitwise OR of the masks:
  - IB_MR_REREG_TRANS: Modify the memory pages of this Memory Region.
  - IB_MR_REREG_PD: Modify the PD of this Memory Region.
  - IB_MR_REREG_ACCESS: Modify the allowed operations of this Memory Region.
- `pd`: The new Protection Domain that this Memory Region will be associated with.
- `phys_buf_array`: The new physical pages to be used.
- `num_phys_buf`: The number of physical pages to be used.
- `mr_access_flags`: The new allowed operations of this Memory Region.
- `iova_start`: The new I/O Virtual Address of this Memory Region.

The ib_query_mr() Method

The `ib_query_mr()` method retrieves the attributes of a specific MR. It will return 0 on success or the errno value with the reason for the failure.

```
int ib_query_mr(struct ib_mr *mr, struct ib_mr_attr *mr_attr);
```

- `mr`: The MR to be queried.
- `mr_attr`: The MR attributes as describe in the next section.

The MR attributes are represented by `struct ib_mr_attr`.

The ib_mr_attr Struct

```
struct ib_mr_attr {
    struct ib_pd    *pd;
    u64         device_virt_addr;
    u64         size;
    int         mr_access_flags;
    u32         lkey;
    u32         rkey;
};
```

- `pd`: The PD that the MR is associated with.
- `device_virt_addr`: The address of the virtual block that this MR covers.
- `size`: The size, in bytes, of the Memory Region.
- `mr_access_flags`: The access permissions of this Memory Region.
- `lkey`: The local key of this Memory Region.
- `rkey`: The remote key of this Memory Region.

The ib_dereg_mr() Method

The `ib_dereg_mr()` method deregisters an MR. This method may fail if a Memory Window is bounded to it. It will return 0 on success or the errno value with the reason for the failure:

```
int ib_dereg_mr(struct ib_mr *mr);
```

- `mr`: The MR to be deregistered.

APPENDIX B

Network Administration

This appendix reviews some of the most popular tools for network administration and debugging. These tools can help a lot in finding solutions to common problems and in developing, debugging, benchmarking, analyzing, troubleshooting, and researching network projects. Most of these tools have very good documentation resources, either with man pages or with wiki pages, and a lot of other information resources about them are on the Internet. Many of them have active mailing lists (for users and developers) and a bug reporting system. Some of the most commonly used tools are described here by specifying their purpose and relevant links, accompanied by several examples. The tools mentioned in this appendix appear in alphabetical order.

arp

This command is for ARP table management. Example of usage:

You can display the ARP table by running `arp` from the command-line. `arp -n` will display the ARP table without name resolution.

You can add static entries to the ARP table by:

```
arp -s 192.168.2.10 00:e0:4c:11:22:33
```

The `arp` utility belongs to the `net-tools` package. Website: http://net-tools.sourceforge.net.

arping

A utility to send ARP requests. The `-D` flag is for Duplicate Address Detection (DAD). The `arping` utility belongs to the `iputils` package. Website: http://www.skbuff.net/iputils/.

arptables

A userspace tool for configuring rules for a Linux-based ARP rules firewall. Website: http://ebtables.sourceforge.net/.

arpwatch

A userspace tool for monitoring ARP traffic. Website: http://ee.lbl.gov/.

ApacheBench (ab)

A command-line utility for measuring the performance of HTTP web servers. The ApacheBench tool is part of the Apache open source project. In many distributions (for example, Ubuntu) it is part of the `apache2-utils` package. Example of usage:

```
ab -n 100  http://www.google.com/
```

The `-n` option is the number of requests to perform for the benchmarking session.

brctl

A command-line utility for administration of Ethernet bridges, enabling the setup of a bridge configuration. The `brctl` utility belongs to the `bridge-utils` package. Examples for usage:

- `brctl addbr mybr`: Add a bridge named `mybr`.
- `brctl delbr mybr`: Delete the bridge named `mybr`.
- `brctl addif mybr eth1`: Add the `eth1` interface to the bridge.
- `brctl delif mybr eth1`: Delete the `eth1` interface from the bridge.
- `brctl show`: Show information about the bridge and its attached ports.

The maintainer of the `bridge-utils` package is Stephen Hemminger. Fetching the `git` repository can be done by:

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/shemminger/bridge-utils.git
```

Website: http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge.

conntrack-tools

A set of userspace tools for management of netfilter connection tracking. It consists of a userspace daemon, `conntrackd`, and a command-line tool, `conntrack`. Website: http://conntrack-tools.netfilter.org/.

crtools

A utility for checkpoint/restore of a process. Website: http://criu.org/Installation.

ebtables

A userspace tool for configuring rules for a Linux-based bridging firewall. Website: http://ebtables.sourceforge.net/.

ether-wake

A utility to send Wake-On-LAN Magic Packets. The `ether-wake` utility belongs to the `net-tools` package.

ethtool

The `ethtool` utility provides a way to query or control network driver and hardware settings, get statistics, get diagnostic information, and more. With `ethtool` you can control parameters of Ethernet devices, such as speed, duplex, auto-negotiation and flow control. Many features of `ethtool` require support in the network driver code.

Examples:

- Output of `ethtool eth0`:

```
Settings for eth0:
        Supported ports: [ TP MII ]
        Supported link modes:   10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
                                1000baseT/Half 1000baseT/Full
        Supported pause frame use: No
        Supports auto-negotiation: Yes
        Advertised link modes:  10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
                                1000baseT/Half 1000baseT/Full
        Advertised pause frame use: Symmetric Receive-only
        Advertised auto-negotiation: Yes
        Speed: 10Mb/s
        Duplex: Half
        Port: MII
        PHYAD: 0
        Transceiver: internal
        Auto-negotiation: on
        Supports Wake-on: pumbg
        Wake-on: g
        Current message level: 0x00000033 (51)
                               drv probe ifdown ifup
        Link detected: no
```

- Getting offload parameters is done by: `ethtool -k eth1`.
- Setting offload parameters is done by: `ethtool -K eth1 offLoadParamater`.
- Querying the network device for associated driver information is done by: `ethtool -i eth1`.
- Showing statistics is done by: `ethtool -S eth1` (note that not all the network device drivers implement this feature).
- Show permanent hardware (MAC) address: `ethtool -P eth0`.

The development of `ethtool` is done by sending patches to the `netdev` mailing list. The maintainer of `ethtool` as of this writing is Ben Hutchings. The `ethtool` project is developed over a `git` repository. It can be downloaded by: `git clone git://git.kernel.org/pub/scm/network/ethtool/ethtool.git`.

Website: www.kernel.org/pub/software/network/ethtool/.

git

A distributed version control system started by Linus Torvalds. Linux kernel development, as well as many Linux related projects, are managed by `git`. One can also use the `git send-email` command in order to send patches by mail. Website: http://git-scm.com/.

hciconfig

A command-line tool for configuring Bluetooth devices. With `hciconfig`, you can display information such as the Bluetooth interface type (BR/EDR or AMP), its Bluetooth address, its flags, and more. The `hciconfig` tool belongs to the `bluez` package. Example:

```
hciconfig
hci0:   Type: BR/EDR  Bus: USB
        BD Address: 00:02:72:AA:FB:94  ACL MTU: 1021:7  SCO MTU: 64:1
        UP RUNNING PSCAN
        RX bytes:964 acl:0 sco:0 events:41 errors:0
        TX bytes:903 acl:0 sco:0 commands:41 errors:0
```

Website: http://www.bluez.org/.

hcidump

A command-line utility for dumping raw HCI data coming from and going to a Bluetooth device. The `hcidump` utility belongs to the `bluez-hcidump` package. Website: http://www.bluez.org/.

hcitool

A command-line utility for configuring Bluetooth connections and for sending some special commands to Bluetooth devices. For example, you can scan for nearby Bluetooth devices by: `hcitool scan`. The `hcitool` utility belongs to the `bluez-hcidump` package.

ifconifg

The `ifconfig` command allows you to configure various network interface parameters, including the IP address of the device, the MTU, the MAC address, the Tx queue length (`txqueuelen`), flags, and more. The `ifconfig` tool belongs to the `net-tools` package, which is older than the `iproute2` package (discussed later in this appendix). Here are three examples of usage:

- `ifconfig eth0 mtu 1300`: Change the MTU to 1300.
- `ifconfig eth0 txqueuelen 1100`: Change the Tx Queue length to 1100.
- `ifconfig eth0 -arp`: Disable the ARP protocol on `eth0`.

Website: http://net-tools.sourceforge.net.

ifenslave

A utility for attaching and detaching slave network devices to a bonding device. *Bonding* is putting multiple physical Ethernet devices into a single logical one, what is often termed as Link aggregation/Trunking/Link bundling. The source file is in `Documentation/networking/ifenslave.c`. You can attach `eth0`, for example, to a bonding device `bond0` by:

```
ifenslave bond0 eth0
```

The `ifenslave` utility belongs to the `iputils` package, maintained by Yoshifuji Hideaki. Website: www.skbuff.net/iputils/.

iperf

The iperf project is an open source project that provides a benchmarking tool to measure TCP and UDP bandwidth performance. It allows you to tune various parameters. The iperf tool reports bandwidth, delay jitter, and datagram loss. It was originally developed by the Distributed Applications Support Team (DAST) at the National Laboratory for Applied Network Research (NLANR) in C++. It works in a client-server model. A new implementation from scratch, iperf3, which is not backwards compatible with the original iperf, is available from https://code.google.com/p/iperf/. The iperf3 is said to have a simpler code base. The iperf3 tool can report also the average CPU utilization of the client and the server.

Using iperf

Following is a simple example of using iperf for measuring TCP performance. On one device (which has an IP address of 192.168.2.104), run the next command, which starts the server side (by default, it is a TCP socket on port 5001):

```
iperf -s
```

On a second device, run the iperf TCP client to connect to the iperf server:

```
iperf -c 192.168.2.104
```

On the client side you will see the following:

```
------------------------------------------------------------
Client connecting to 192.168.2.104, TCP port 5001
TCP window size: 22.9 KByte (default)
------------------------------------------------------------
[  3] local 192.168.2.200 port 35146 connected with 192.168.2.104 port 5001
```

The default time interval is 10 seconds. After 10 seconds, the client will be disconnected, and you will see a message like this on the terminal:

```
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.3 sec  7.62 MBytes  6.20 Mbits/sec
```

You can tune many parameters of iperf, like these:

- -u: For using a UDP socket.
- -t: For using a different time interval in seconds instead of the default of 10 seconds.
- -T: Sets a TTL for multicast (the default is 1).
- -B: Bind to a host, an interface, or a multicast address.

See man iperf. Website: http://iperf.sourceforge.net/.

iproute2

The iproute2 package provides many tools for interaction between the userspace and the kernel networking subsystem. The most well-known is the ip command. It is based on netlink sockets (discussed in Chapter 2). With the

ip command, you can perform various operations in a wide range of networking areas, and it has numerous options; see man 8 ip. Here are several examples of using the ip command for various tasks:

- Configuration of a network device with ip addr:
  - ip addr add 192.168.0.10/24 dev eth0: Sets an IP address on eth0.
  - ip addr show: Displays the addresses of all network interfaces (both IPv4 and IPv6).

  See man ip address.
- Configuration of a network device with ip link:
  - ip link add mybr type bridge: Creates a bridge named mybr.
  - ip link add name myteam type team: Creates a teaming device named myteam. (The teaming device driver aggregates multiple physical Ethernet devices into one logical one and is in fact the new bonding device. The teaming driver is discussed in Chapter 14.)
  - ip link set eth1 mtu 1450: Sets the MTU of eth1 to be 1450.

  See man ip link.
- Management of ARP tables (IPv4) and NDISC (IPv6) tables:
  - ip neigh show: Shows both the IPv4 neighbouring table (ARP table) and the IPv6 neighbouring table.
  - ip -6 neigh show: Shows only the IPv6 neighbouring table.
  - ip neigh flush dev eth0: Removes all entries from the neighboring tables associated with eth0.
  - ip neigh add 192.168.2.20 dev eth2 lladdr 00:11:22:33:44:55 nud permanent: Adds a permanent neighbour entry (parallel to adding static entries in an ARP table).
  - ip neigh change 192.168.2.20 dev eth2 lladdr 55:44:33:22:11:00 nud permanent: Updates a neighbour entry.

  See man ip neighbour.
- Management of the parameters for the neighbour tables:
  - ip ntable show: Displays the neighbour tables parameters.
  - ip ntable change name arp_cache locktime 1200 dev eth0: Changes the locktime parameter for the IPv4 neighbouring table associated with eth0.

  See man ip ntable.
- Network namespaces management:
  - ip netns add myNamespace: Adds a network namespace named myNamespace.
  - ip netns del myNamespace: Deletes the network namespace named myNamespace.
  - ip netns list: Shows all network namespaces on the host.
  - ip netns monitor: Displays a line of screen for each network namespace that is added or removed by the ip netns command.

See `man ip netns`.

- Configuration of multicast addresses:
  - `ip maddr show`: Shows all multicast addresses on the host (both IPv4 and IPv6).
  - `ip maddr add 00:10:02:03:04:05 dev eth1`: Adds a multicast address on `eth1`.

  See `man ip maddress`.
- Monitor netlink messages. For example:
  - `ip monitor route` displays on the screen messages about various network events like adding or deleting a route.

  See `man ip monitor`.
- Management of routing tables:
  - `ip route show`: Shows the routing table.
  - `ip route flush dev eth1`: Removes routing entries associated with `eth1` from the routing table.
  - `ip route add default via 192.168.2.1`: Adds 192.168.2.1 as a default gateway.
  - `ip route get 192.168.2.10`: Gets the route to 192.168.2.10 and displays it.

  See `man ip route`.
- Management of rules in the RPDB (Routing Policy DataBase). For example:
  - `ip rule add tos 0x02 table 200`: Adds a rule that sets the routing subsystem to perform a lookup in routing table 252 for packets whose TOS value is 0x02 (TOS is a field in the IPv4 header).
  - `ip rule del tos 0x02 table 200`: Deletes a specified rule from the RPDB.
  - `ip rule show`: Displays the rules in the RPDB.

  See `man ip rule`.
- Management of TUN/TAP devices:
  - `ip tuntap add tun1 mode tun`: Creates a TUN device named `tun1`.
  - `ip tuntap del tun1 mode tun`: Deletes a TUN device named `tun1`.
  - `ip tuntap add tap1 mode tap`: Creates a TAP device named `tap1`.
  - `ip tuntap del tap1 mode tap`: Deletes a TAP device named `tap1`.
- Management of IPsec policies:
  - `ip xfrm policy show`: Shows IPsec policies.
  - `ip xfrm state show`: Shows IPsec states.

  See `man ip xfrm`.

The `ss` tool is used to dump socket statistics. For example, running

```
ss -t –a
```

will show all TCP sockets:

```
State       Recv-Q Send-Q          Local Address:Port              Peer Address:Port
LISTEN      0      32                          *:ftp                          *:*
LISTEN      0      128                         *:ssh                          *:*
LISTEN      0      128             127.0.0.1:ipp                          *:*
ESTAB       0      0               192.168.2.200:ssh             192.168.2.104:52089
ESTAB       0      52              192.168.2.200:ssh             192.168.2.104:51352
ESTAB       0      0               192.168.2.200:ssh             192.168.2.104:51523
ESTAB       0      0               192.168.2.200:59532          107.21.231.190:http
LISTEN      0      128                        :::ssh                         :::*
LISTEN      0      128                       ::1:ipp                         :::*
CLOSE-WAIT  1      0                         ::1:48723                      ::1:ipp
```

There are other tools of `iproute2`:

- `bridge`: Shows/manipulates bridge addresses and devices. For example:
  - `bridge fdb show`: Displays forwarding entries.

  See `man bridge`.
- `genl`: Gets information (like id, header size, max attributes, and more) about registered generic netlink families. For example, running `genl ctrl list` can have this as a result:

```
Name: nlctrl
        ID: 0x10  Version: 0x2  header size: 0  max attribs: 7
        commands supported:
                #1:  ID-0x3
                Capabilities (0xe):
                  can doit; can dumpit; has policy

        multicast groups:
                #1:  ID-0x10  name: notify
```

- `lnstat`: Displays Linux network statistics.
- `rtmon`: Monitors Rtnetlink sockets.
- `tc`: Shows/manipulates traffic control settings. For example:
  - `tc qdisc show`: Running this command shows which queueing discipline (`qdisc`) entries are installed, for example:

```
qdisc pfifo_fast 0: dev eth1 root refcnt 2 bands 3 priomap  1 2 . . .
```

- This shows that the `pfifo_fast qdisc` is associated with the `eth1` network device. The `pfifo_fast qdisc`, which is a classless queueing discipline, is the default `qdisc` in Linux.
  - `tc -s qdisc show dev eth1`: Shows statistics of the `qdisc` associated to `eth1`.

  See `man tc`.

  See: Linux Advanced Routing & Traffic Control HOWTO: www.lartc.org/howto/.

The development of `iproute2` is done by sending patches to the `netdev` mailing list. The maintainer of `ethtool` as of this writing is Stephen Hemminger. The `iproute2` is developed over a `git` repository, which can be downloaded by: `git clone git://git.kernel.org/pub/scm/linux/kernel/git/shemminger/iproute2.git`.

iptables and iptables6

The `iptables` and `iptables6` are administration tools for packet filtering and NAT management for IPv4 and IPv6, respectively. With `iptables/iptables6`, you can define lists of rules. Each such rule tells what should be done with the packet (for example, discard it or accept it). Each rule specifies some matching condition for a packet, for example, that it will be a UDP packet. Following are some examples for using the `iptables` command:

- `iptables -A INPUT -p tcp --dport=80 -j LOG --log-level 1`: The meaning of this rule is that incoming TCP packets with destination port 80 will be dumped to the syslog.
- `iptables –L`: Lists all rules in the filter table. (There is no table mentioned in the command, so it accesses the Filter table, which is the default table.)
- `iptables –t nat –L`: Lists all rules in the NAT table.
- `iptables –F`: Flushes the selected table.
- `iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE`: Sets a MASQUERADE rule.

Website: www.netfilter.org/.

ipvsadm

A tool for Linux Virtual Server administration. Website: www.linuxvirtualserver.org/software/ipvs.html.

iw

Shows/manipulates wireless devices and their configuration. The `iw` package is based on generic netlink sockets (see Chapter 2). For example, you can perform these operations:

- `iw dev wlan0 scan`: Scans for nearby wireless devices.
- `iw wlan0 station dump`: Displays statistics about a station.
- `iw list`: Gets information about a wireless device (such as band information and 802.11n information).
- `iw dev wlan0 get power_save` – get power save mode.
- `iw dev wlan0 set type ibss`: Changes the wireless interface mode to be ibss (Ad-Hoc).
- `iw dev wlan0 set type mesh`: Changes the wireless interface mode to be mesh mode.
- `iw dev wlan0 set type monitor`: Changes the wireless interface mode to be monitor mode.
- `iw dev wlan0 set type managed`: Changes the wireless interface mode to be managed mode.

See `man iw`.
Gitweb: http://git.kernel.org/cgit/linux/kernel/git/jberg/iw.git.
Website: http://wireless.kernel.org/en/users/Documentation/iw.

iwconfig

The old tool for administering wireless devices. The `iwconfig` belongs to the `wireless-tools` package and is based on IOCTLs. Website: www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html.

libreswan Project

An IPsec software solution which forked from openswan version 2.6.38. Website: http://libreswan.org/.

l2ping

A command-line utility for sending L2CAP echo requests and receiving answers over a Bluetooth device. The l2ping utility belongs to the bluez package. Website: www.bluez.org/.

lowpan-tools

A set of utilities to manage the Linux LoWPAN stack. Website: http://sourceforge.net/projects/linux-zigbee/files/linux-zigbee-sources/0.3/.

lshw

A utility that displays information about the hardware configuration of the machine. Website: http://ezix.org/project/wiki/HardwareLiSter.

lscpu

A utility for displaying information about the CPUs on the system. It is based on information from /proc/cpuinfo and sysfs. The lscpu belongs to the util-linux package.

lspci

A utility for displaying information about PCI buses in the system and devices connected to them. Sometimes you need to get some information about a PCI network device with the lspci command. The lspci utility belongs to the pciutils package. Website: http://mj.ucw.cz/sw/pciutils/.

mrouted

A multicast routing daemon, implementing the IPv4 Distance Vector Multicast Routing Protocol (DVMRP), which is specified in RFC 1075 from 1988. Website: http://troglobit.com/mrouted.html.

nc

A command-line utility that reads and writes data across networks. The nc belongs to the nmap-ncat package. Website: http://nmap.org/.

ngrep

A command-line tool, based on the well-known `grep` command, that allows you to specify extended expressions to match against data payloads of packets. It recognizes TCP, UDP, and ICMP across Ethernet, PPP, SLIP, FDDI, and null interfaces. Website: http://ngrep.sourceforge.net/.

netperf

Netperf is a networking benchmarking tool. Website: www.netperf.org/netperf/.

netsniff-ng

`netsniff-ng` is an open source project networking toolkit that, among other things, can help in analyzing network traffic, performing stress tests, generating packets at a very high speed, and more. It uses the PF_PACKET zero-copy RINGs (TX and RX). Among the tools it provides are the following:

- `netsniff-ng` is a fast zero-copy analyzer, `pcap` capturing and replaying tool. The `netsniff-ng` tool is Linux-specific and does not support other operating systems, unlike many of the tools mentioned in this appendix. Example: Running `netsniff-ng --in eth1 --out dump.pcap -s -b 0` creates a `pcap` file that can be read by `wireshark` or by `tcpdump`. The `-s` flag is for silence, and the `-b 0` is for binding to CPU 0. See `man netsniff-ng`.
- `trafgen` is a zero-copy high performance network packet traffic generator utility.
- `ifpps` is a small utility that periodically provides top-like networking and system statistics from the kernel. `ifpps` gathers its data directly from `procfs` files.
- `bpfc` is a small Berkeley Packet Filter assembler and compiler.

Fetching the `git` repository: `git clone git://github.com/borkmann/netsniff-ng.git`.
Website: http://netsniff-ng.org/.

netstat

The `netstat` tool enables you to print multicast memberships, routing tables, network connections, interface statistics, state of sockets, and more. The `netstat` tool belongs to the `net-tools` package. Useful flags:

- `netstat -s`: Displays summary statistics for each protocol.
- `netstat -g`: Displays multicast group membership information for IPv4 and IPv6.
- `netstat -r`: Shows the kernel IP routing table.
- `netstat -nl`: Shows the listening sockets (the `-n` flag is for showing numerical addresses instead of trying to determine symbolic host, port, or user names).
- `netstat -aw`: Shows all raw sockets.
- `netstat -ax`: Shows all Unix sockets.
- `netstat -at`: Shows all TCP sockets.
- `netstat -au`: Shows all UDP sockets.

Website: http://net-tools.sourceforge.net.

nmap (Network Mapper)

Nmap is an open source security project that provides a network exploration and probing tool and a security/port scanner. It has features like port scanning (detecting the open ports on target hosts), OS detection, detecting MAC addresses, and more. For example,

```
nmap www.google.com
```

can give output such as:

```
Starting Nmap 6.00 (http://nmap.org ) at 2013-09-26 16:37 IDT
Nmap scan report for www.google.com (212.179.154.227)
Host is up (0.013s latency).
Other addresses for www.google.com (not scanned): 212.179.154.221 212.179.154.251 212.179.154.232
212.179.154.237 212.179.154.216 212.179.154.231 212.179.154.241 212.179.154.247 212.179.154.222
212.179.154.226 212.179.154.236 212.179.154.246 212.179.154.212 212.179.154.217 212.179.154.242
Not shown: 998 filtered ports
PORT    STATE SERVICE
80/tcp  open  http
443/tcp open  https
Nmap done: 1 IP address (1 host up) scanned in 5.24 seconds
```

The `nping` utility of `nmap` can be used to generate raw packets for ARP poisoning, networking stress tests, and Denial of Service attacks, as well as to test connectivity like the ordinary `ping` utility. You can use the `nping` utility for setting IP options in generated traffic. See `http://nmap.org/book/nping-man-ip-options.html`. Website: `http://nmap.org/`.

openswan

An open source project implementing an IPsec-based VPN solution. It is based on the FreeS/WAN project. Website: `www.openswan.org/projects/openswan`.

OpenVPN

An open source project implementing VPN based on SSL/TLS. Website: `www.openvpn.net/`.

packeth

An Ethernet-based packet generator tool for Ethernet. The tool has both GUI and CLI. Website: `http://packeth.sourceforge.net/packeth/Home.html`.

ping

The well-known utility for testing connectivity by sending ICMP ECHO request messages. Here are four useful options that are also mentioned in this book:

- `-Q tos`: Enables setting Quality Of Service bits in an ICMP packet. Mentioned in this appendix in the explanation about `tshark` filters.
- `-R`: Sets the Record Route IP option (discussed in Chapter 4).

- `-T`: Sets the timestamp IP option (discussed in Chapter 4).
- `-f`: Flood ping.
- See `man ping` for more command-line options.

The `ping` utility belongs to the `iputils` package. Website: .`www.skbuff.net/iputils/`.

pimd

An open source lightweight stand-alone Protocol Independent Multicast - Sparse Mode (PIM-SM) v2 multicast daemon. Maintained by Joachim Nilsson. See `http://troglobit.com/pimd.html`. `git` repository: `https://github.com/troglobit/pimd/`.

poptop

PPTP Server for Linux. Website: `http://poptop.sourceforge.net/dox/`.

ppp

An open source PPP daemon. `git` repository: `git://ozlabs.org/~paulus/ppp.git`. Website: `http://ppp.samba.org/download.html`.

pktgen

The `pktgen` kernel module (`net/core/pktgen.c`) can generate packets at very high speed. Monitoring and controlling is done via writing to `/proc/net/pktgen` entries. For "HOWTO for the linux packet generator" see `Documentation/networking/pktgen.txt`.

radvd

This is a Router Advertisement Daemon for IPv6. It is an open source project maintained by Reuben Hawkins. It can be used for IPv6 stateless autoconfiguration and for renumbering. Website: `www.litech.org/radvd/`. `git` repository: `https://github.com/reubenhwk/radvd`.

route

A command-line tool for routing tables management. It belongs to the `net-tools` package, which is based on IOCTLs and which is older than the `iproute2` package. Examples:

- `route -n`: Shows the routing table without name resolving.
- `route add default gateway 192.168.1.1`: Adds 192.168.1.1 as a default gateway.
- `route -C`: Displays the routing cache (keep in mind that the IPv4 routing cache was removed in kernel 3.6; see the "IPv4 Routing Cache" section in chapter 5).

See `man route`.

RP-PPPoE

An open source PPP over Ethernet (PPPoE) client for Linux and Solaris systems. Website: www.roaringpenguin.com/products/pppoe.

sar

A command-line tool to collect and report statistics about system activity. It is part of the `sysstat` package. As an example, running the following command will display four times the CPU statistics with interval of 1 second and the average at the end:

```
sar 1 4
Linux 3.6.10-4.fc18.x86_64 (a)  10/22/2013      _x86_64_        (2 CPU)

07:47:10 PM     CPU     %user     %nice   %system   %iowait    %steal     %idle
07:47:11 PM     all      0.00      0.00      0.00      0.00      0.00    100.00
07:47:12 PM     all      0.00      0.00      0.00      0.00      0.00    100.00
07:47:13 PM     all      0.00      0.00      0.00      0.00      0.00    100.00
07:47:14 PM     all      0.00      0.00      0.50      0.00      0.00     99.50
Average:        all      0.00      0.00      0.13      0.00      0.00     99.87
```

Website: http://sebastien.godard.pagesperso-orange.fr/.

smcroute

A command-line tool for multicast routing manipulation. Website: www.cschill.de/smcroute/.

snort

An open source project that provides a network intrusion detection system (IDS) and a network intrusion prevention system (IPS). Website: www.snort.org/.

suricata

An open source project that provides an IDS/IPS and a network security monitoring engine. Website: http://suricata-ids.org/.

strongSwan

An open source project that implements IPsec solutions for Linux, Android, and other operating systems. Both IKEv1 and IKEv2 are implemented. The maintainer is Professor Andreas Steffen. Website: www.strongswan.org/.

sysctl

The `sysctl` utility displays kernel parameters (including network parameters) at runtime. It can also set kernel parameters. For example, `sysctl -a` shows all kernel parameters. The `sysctl` utility belongs to the `procps-ng` package.

taskset

A command-line utility for setting or retrieving a process's CPU affinity. The `taskset` utility is from the `util-linux` package.

tcpdump

`Tcpdump` is an open source command-line protocol analyzer, available from `www.tcpdump.org`. It is based on a C/C++ network traffic capture library called `libpcap`. Like `wireshark`, it can write its results to a file and read them from a file and it supports filtering. Unlike `wireshark`, it does not have a front end GUI. However, its output files can be read by `wireshark`. Example of sniffing with `tcpdump`:

```
tcpdump -i eth1
```

Website: `www.tcpdump.org`.

top

The `top` utility provides a real-time view of the system (parameters like memory usage, CPU usage, and more) and a system summary. This utility is part of the `procps-ng` package. Website: `https://gitorious.org/procps`.

tracepath

The `tracepath` command traces a path to a destination address, discovering the MTU along this path. For IPv6 destination addresses, you can use `tracepath6`. The `tracepath` utility belongs to the `iputils` package. Website: `www.skbuff.net/iputils/`.

traceroute

Print the path that packets traverse to some destination. The `traceroute` utility uses the IP protocol's Time To Live (TTL) field to cause hosts on the packet path to return an ICMP TIME EXCEEDED response. The `traceroute` utility is discussed in Chapter 3, which deals with the ICMP protocol. Website: `http://traceroute.sourceforge.net`.

tshark

The `tshark` utility provides a command-line packet analyzer. It is part of the `wireshark` package. It has many command-line options. For example, you can write the output to a file with the `-w` option. You can set various filters to the packet filtering with `tshark`, some of which can be complex filters (as you will soon see). Example of setting a filter for capturing only ICMPv4 packets:

```
tshark -R icmp
Capturing on eth1
17.609101 192.168.2.200 -> 81.218.16.241 ICMP 98 Echo (ping) request  id=0x0dc6, seq=1/256, ttl=64
17.617101 81.218.16.241 -> 192.168.2.200 ICMP 98 Echo (ping) reply    id=0x0dc6, seq=1/256, ttl=58
```

You can also set a filter on a value of a field in the IPv4 header. For example, the following command sets a filter on the DS field in the IPv4 header:

```
tshark -R "ip.dsfield==0x2"
```

If from a second terminal you send traffic with DS field as 0x2 in the IPv4 header (such traffic can be sent, for example, with `ping -Q 0x2 destinationAdderss`), it will be displayed onscreen by `tshark`.

Example for filtering by source MAC address:

```
tshark ether src host 00:e0:4c:11:22:33
```

Example for filtering for UDP packets whose ports are in the port range 6000–8000:

```
tshark -R udp portrange 6000-8000
```

Example for setting a filter for capturing traffic where the source IP address is 192.168.2.200 and the port is 80 (it does not have to be TCP traffic only because here there is no filter set on some specified protocol):

```
tshark -i eth1 -f "src host 192.168.2.200 and port 80"
```

tunctl

`tunctl` is an older tool for creating TUN/TAP devices. It is available from http://tunctl.sourceforge.net. Note that you can also create or remove a TUN/TAP device with the `ip` command (see the `iproute2` section earlier in this appendix) and with the `openvpn` command-line tool of the `openvpn` package:

```
openvpn --mktun --dev tun1
openvpn --rmtun --dev tun1
```

udevadm

You can get the network device type by running `udevadm` on its `sysfs` entry. For example, if the device has this entry under `sysfs`:

```
/sys/devices/virtual/net/eth1.100
```

then you can find that its DEVTYPE is VLAN:

```
udevadm info -q all -p  /sys/devices/virtual/net/eth1.100/

P: /devices/virtual/net/eth1.100
E: COMMENT=net device ()
E: DEVPATH=/devices/virtual/net/eth1.100
E: DEVTYPE=vlan
E: IFINDEX=4
E: INTERFACE=eth1.100
E: MATCHADDR=00:e0:4c:53:44:58
E: MATCHDEVID=0x0
```

```
E: MATCHIFTYPE=1
E: SUBSYSTEM=net
E: UDEV_LOG=3
E: USEC_INITIALIZED=28392625695
```

udevadm belongs to the udev package. Website: www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html.

unshare

The unshare utility enables you to create a namespace and run a program within that namespace that is unshared from its parent. The unsare utility belongs to the util-linux package. For various command-line options of the unshare utility, see man unshare, Example of usage:

```
unshare -u /bin/bash
```

This will create a UTS namespace.

```
unshare --net /bin/bash
```

This will create a new network namespace, in which a bash process will be started. Gitweb: http://git.kernel.org/cgit/utils/util-linux/util-linux.git. Website: http://userweb.kernel.org/~kzak/util-linux/.

vconfig

The vconfig utility enables you to configure VLAN (802.1q) interface. Examples of usage:

- vconfig add eth2 100: Adds a VLAN interface. This will create a VLAN interface, eth2.100.
- vconfig rem eth2.100: Remove the eth2.100 VLAN interface.
- Note that you can also add and delete VLAN interfaces with the ip command, for example, like this:
  - ip link add link eth0 name eth0.100 type vlan id 100
- vconfig set_egress_map eth2.100 0 4: Map SKB priority of 0 to VLAN priority 4, so that outgoing packets which their SKB priority is 0 will be tagged with 4 as VLAN priority. The default VLAN priority is 0.
- vconfig set_ingress_map eth2.100 1 5: Map VLAN priority 5 to SKB priority of 1, so that incoming packets with VLAN priority of 5 will be queued with SKB priority of 1. The default SKB priority is 0.

See man vconfig.

Note that if VLAN support is compiled as a kernel module, then you must load the VLAN kernel module before trying to add the VLAN interface, by modprobe 8021q. Website: www.candelatech.com/~greear/vlan.html.

wpa_supplicant

Open source software that provides a wireless supplicant for Linux and other OSs. It supports WPA and WPA2. Website: http://hostap.epitest.fi/wpa_supplicant/.

wireshark

The `wireshark` project provides a free and open source analyzer ("sniffer"). It has two flavors: a front-end GTK+ based GUI and a command-line, the `tshark` utility (mentioned earlier in this appendix). It is available on many operating systems and evolves dynamically: when new features are added to existing protocols and new protocols are added, new parsers ("dissectors") are modified or added. Wireshark has many features:

- Enables defining a wide range of filters (ports, destination or source address, protocol identifier, fields in headers, and more).
- Enables sorting the result according to various parameters (protocol type, time, and so on).
- Saves the sniffer output to a file/read a sniffer output from a file.
- Reads/writes many different capture file formats: `tcpdump` (`libpcap`), Pcap NG, and more.
- Capture Filters and Display Filters.

Activating the `wireshark` or `thsark` sniffer puts the network interface to be in promiscuous mode to enable it to handle packets that are not destined to the local host. A lot of information is available in the man pages: `man wireshark` and `man tshark`. You can find more than 75 sniff samples of different protocols in `http://wiki.wireshark.org/SampleCaptures`. Wireshark users mailing list: `www.wireshark.org/mailman/listinfo/wireshark-users`. Website: `www.wireshark.org`. Wiki: `http://wiki.wireshark.org/`.

XORP

An Open Source project, implementing various routing protocols, like BGP, IGMP, OLSR, OSPF, PIM, and RIP. The name XORP is derived from eXtensible Open Router Platform. Website: `www.xorp.org/`.

APPENDIX C

Glossary

The following list of glossary terms are covered in this book.

ACL—Asynchronous Connection-oriented Link. A Bluetooth protocol.

ADB — Android Debug Bridge.

AVDTP—Audio/Video Distribution Transport Protocol. A Bluetooth protocol.

AEAD—Authenticated Encryption with Associated Data.

AES-NI—AES instruction set.

AH—Authentication Header protocol. Used in IPsec, has a protocol number 51.

AID—Association ID. A unique number that a wireless client gets when it associates to an Access Point. It is assigned by the Access Point, and it is in the range 1–2007.

AMP—Alternate MAC/PHY.

AMPDU—Aggregated Mac Protocol Data Unit. A type of packet aggregation in IEEE 802.11n.

AMSDU—Aggregated Mac Service Data Unit. A type of packet aggregation in IEEE 802.11n.

AOSP—Android Open Source Project.

AP—Access Point. In wireless networks, a wireless device to which wireless clients associate and which enables them to connect to a wired network.

API—Application Programming Interface. A set of methods and data structures that define the interface to a software layer, such as an interface for a library.

ABRO—Authoritative Border Router Option. Added for Neighbour Discovery Optimization for IPv6. See RFC 6775.

ABS—Android Builders Summit.

ARO—Address Registration Option. Added for Neighbour Discovery Optimization for IPv6. See RFC 6775.

ARP—Address Resolution Protocol. A protocol used to find the mapping between a network address (such as IPv4 address) into a link layer address (like a 48-bit Ethernet address).

ARPD—ARP daemon. A userspace daemon that implements the ARP functionality.

Ashmem—Android shared memory.

ASM—Any-Source Multicast. In the any-source model, you do not specify interest in receiving multicast traffic from a single particular source address or from a set of addresses.

BA—Block Acknowledgement mechanism used in IEEE 802.11n.

BGP—Border Gateway Protocol. A core routing protocol.

BLE—Bluetooth Low Energy.

BNEP—Bluetooth Network Encapsulation Protocol.

BTH—Base Transport Header. An InfiniBand header of 12 bytes. It specifies the source and destination QPs, the operation, packet sequence number, and partition.

CM—Communication Manager in the InfiniBand stack.

CIDR—Classless Inter-Domain Routing. A way to allocate Internet addresses used in inter-domain routing.

CQ—Completion Queue (InfiniBand).

CRIU — Checkpoint/Restore In Userspace. CRIU is a software tool, mainly implemented in userspace, with which you can freeze a running process and checkpoint it to a filesystem as a collection of files. You can then use these files to restore and run the application from the point where it was frozen. See http://criu.org/Main_Page.

CSMA/CD—Carrier Sense Multiple Access/Collision Detection. A Media Access Control method used in Ethernet networks.

CSMA/CA—Carrier Sense Multiple Access/Collision Avoidance. A Media Access Control method used in wireless networks.

CT—Connection Tracking. A netfilter layer that is the basis for NAT.

DAD—Duplicate Address Detection. The DAD is a mechanism that helps to detect the existence of double L3 addresses on different hosts on a LAN.

DAC—Duplicate Address Confirmation. An ICMPv6 type which was added in RFC 6775, with numeric value of 158.

DAR—Duplicate Address Request. An ICMPv6 type which was added in RFC 6775, with numeric value of 157.

DCCP—Datagram Congestion Control Protocol. An unreliable, congestion-controlled transport layer protocol. The use of DCCP would make sense, for instance, in applications that require low delays and where a small degree of data loss is permitted, like in telephony and streaming media applications.

DHCP—Dynamic Host Configuration Protocol. A protocol for configuring network device parameters like an IP address, a default route, and one or more DNS server addresses.

DMA—Direct Memory Access.

DNAT—Destination NAT. A NAT that changes the destination address.

DNS—Domain Name System. A system for translating domain names to IP addresses.

DSCP—Differentiated Services Code Point. A classifying mechanism.

DVMRP—Distance Vector Multicast Routing Protocol. A protocol for routing multicast datagrams. Suitable for use within an autonomous system. Defined in RFC 1075 from 1988.

ECN—Explicit Congestion Notification. See RFC 3168, "The Addition of Explicit Congestion Notification (ECN) to IP."

EDR—Enhanced Data Rate.

EGP—Exterior Gateway Protocol. A routing protocol which is now considered obsolete. It was first formalized in RFC 827 in 1982.

ERTM—Enhanced Retransmission Mode. A reliable protocol with error and flow control, used in Bluetooth.

ESP—Encapsulating Security Payload. Used in IPsec, has protocol number 50.

ETH—Extended Transport Header: An InfiniBand header with size from 4 to 28 bytes. This header represents an extra family of headers that may be present depending on the class of the service and the used operation.

ETSI—European Telecommunications Standards Institute.

FCS—Frame Check Sequence

FIB—Forwarding Information Base. The database that contains the routing tables information.

FMR—Fast Memory Region (InfiniBand).

FSF—Free Software Foundation.

FTP—File Transfer Protocol. A protocol for transferring files between two hosts, based on TCP.

GCC—GNU Compiler Collection.

GID—Global Identifier.

GMP—Group Management Protocol. A term that refers to both IGMP and MLD. See RFC 4604, section 1.

GRE—Generic Routing Encapsulation. A tunneling protocol.

GRH—Global Routing Header. An InfiniBand header of 40 bytes. It describes the source and destination port using GIDs, and its format is identical to the IPv6 header.

GRO—Generic Receive Offload. A technique with which incoming packets are merged at reception time into a bigger packet to improve performance.

GSO—Generic Segmentation Offload. A technique with which outgoing packets are segmented not in the transport layer but as close as possible to the network driver or in the network driver itself.

GUID—Global Unique Identifier.

HAL—Hardware Abstraction Layer.

HCA—Host Channel Adapter.

HCI—Host Controller Interface. Used, for example, in Bluetooth, PCI and more.

HDP—Health Device Profile. Used by Bluetooth.

HFP—Hands-Free Profile. Used by Bluetooth.

HoL Blocking—Head-of-line blocking is a performance-limiting phenomenon that occurs when a line of packets is held up by the first packet, for example, in multiple requests in HTTP pipelining.

HPC—High Performance Computing. Management of computer resources in a way that gives high performance for heavy tasks such as solving large-scale problems in science, engineering, or economics.

HS—High Speed.

HTTP—Hypertext Transfer Protocol. The basic protocol for accessing the World Wide Web.

HWMP— Hybrid Wireless Mesh Protocol. A routing protocol used in wireless Mesh networks that consists of two types of routing: *on-demand* routing and *proactive* routing.

iWARP—Internet Wide Area RDMA Protocol.

iSER—iSCSI extension for RDMA.

IANA—Internet Assigned Numbers Authority. Responsible for IP addressing, global coordination of the DNS Root, and other IP-related symbols and numbers. Operated by the Internet Corporation for Assigned Names and Numbers (ICANN).

IBTA—InfiniBand Trade Association.

ICMP—Internet Control Message Protocol. An IP protocol for control and informational messages. The well-known `ping` utility is based on ICMP. The ICMP protocol is known to be used in various types of security DoS attacks, like the Smurf attack.

ICE—Interactive Connectivity Establishment. Specified in RFC 5245. A protocol for NAT traversal.

ICRC—Invariant CRC. An InfiniBand header of 4 bytes. Covers all fields, which should not be changed as the packet travels in the subnet.

IDS—Intrusion Detection System.

IoT—Internet of Things. Networking of everyday objects.

IEEE—Institute of Electrical and Electronics Engineers.

IGMP—Internet Group Management Protocol. Multicast group memberships protocol.

IKE—Internet Key Exchange. A protocol for setting an IPsec Security Association.

IOMMU—I/O Memory Management Unit.

IP—Internet Protocol. The primary addressing and routing protocol for the Internet. IPv4 was first specified in RFC 791 from 1981, and IPv6 was first specified in RFC 1883 from 1995.

IPoIB—IP over InfiniBand.

IPS—Intrusion Prevention System.

ISAKMP—Internet Security Association & Key Management Protocol.

IOCTL—Input/Output Control. A system call that provides access from userspace to kernel.

IPC—Inter Process Communication. There are many different mechanisms for IPC, such as shared memory semaphores, message queues, and more.

IPCOMP—IP Payload Compression Protocol. A compressing protocol intended to reduce the size of data sent over a slow network connection. Using IPComp increases the overall communication performance between two network nodes.

IPsec—IP security. A set of protocols developed by the IETF for secure exchange of packets over the IP protocol. IPsec is mandatory in IPv6 according to the IPv6 spec and optional in IPv4, though many operating systems implemented it also in IPv4. IPsec uses two encryption modes: Transport and Tunnel.

IPVS—IP Virtual Server. A Linux kernel load balancing infrastructure, supports IPv4 and IPv6. See http://www.linuxvirtualserver.org/software/ipvs.html.

ISR—Interrupt Service Routine. An interrupt handler that is invoked when an interrupt is received.

ISM—Industrial, scientific, and medical radio band.

jumbo frames—Packets with size up to 9K. Some network interfaces allow using an MTU of up to 9K. Using jumbo frames can improve the network performance in some cases, such as in bulk data transfers.

KVM—Kernel-based Virtual Machine. A Linux virtualization project.

LACP—Link Aggregation Control Protocol.

LAN—Local Area Network. A network that connects a limited area, such as an office building.

LID—Local Identifier. A 16-bit value assigned to every subnet port by the Subnet Manager (InfiniBand).

L2CAP—Logical Link Control and Adaptation Protocol. Used in Bluetooth.

L2TP—Layer 2 Tunneling Protocol used by VPNs. L2TPv3 is specified in RFC 3931 (RFC 5641 has some updates).

LKML—Linux Kernel Mailing List.

LLCP —Logical Link Control Protocol. Used by NFC.

LLN—Low-power and Lossy Network.

LoWPAN—Low-power Wireless Personal Area Network.

LMP—Link Management Protocol. Controls the radio link between two Bluetooth devices.

LPM—Longest Prefix Match. An algorithm used by the routing subsystem.

LRH—Local Routing Header. An InfiniBand header of 8 bytes. It identifies the local source and destination ports of the packet. It also specifies the requested QoS attributes (SL and VL) of the message.

LRO—Large Receive Offload.

LR-WPAN—Low-Rate Wireless Personal Area Network. Used in IEEE 802.15.4.

LSB—Least significant bit.

LSRR—Loose Source Record Route.

LTE—Long Term Evolution.

MAC—Media Access Control. A sublayer of the Data Link Layer (L2) of the OSI model.

MAD—Management Datagram (InfiniBand).

MFC—Multicast Forwarding Cache. A data structure in the kernel that consists of multicast forwarding entries.

MIB—Management Information Base.

MLD—Multicast Listener Discovery protocol. Enables each IPv6 router to discover the presence of multicast listeners. The MLD protocol is specified in RFC 3810, from 2004.

MLME—MAC Layer Management Entity. A component in the IEEE 802.11 management layer responsible for operations such as scanning, authentication, association, and reassociation.

MR—Memory Region (InfiniBand).

MSF—Multicast Source Filtering. This is the feature to set filters so that multicast traffic from sources other than the expected ones will be dropped.

MSI—Message Signaled Interrupts.

MSS—Maximum Segment Size. A parameter of the TCP protocol.

MTU—Maximum transmission unit. The size of the largest packet that a network protocol can transmit.

MW—Memory Window (InfiniBand).

NAP—Network Access Point.

NAPI—New API. A technique by which network drivers are not interrupt-driven, but use polling. NAPI is discussed in Chapter 1.

NAT—Network Address Translation. A layer responsible for modifying IP headers. In Linux, support for IPv6 NAT was merged in kernel 3.7.

NAT-T—NAT traversal.

NCI—NFC Controller Interface.

ND / NDISC—Neighbour Discovery Protocol. Used in IPv6. Among its tasks: discovering network nodes on the same link, autoconfiguration of addresses, finding the Link Layer addresses of other nodes, and maintaining reachability information about other nodes.

NFC—Near Field Communication.

NDEF—NFC Data Exchange Format.

NIC—Network Interface Card, also known as Network Interface Controller or Network Adapter. The hardware network device.

NUMA—Non-Uniform Memory Access.

NPP—NDEF Push Protocol.

NPAR—NIC Partitioning. A technology that enables you to split up network card (NIC) traffic in partitions.

NUD—Network Unreachability Detection. A mechanism responsible for determining whether a neighbour can be reached.

OBEX—Object Exchange. A protocol for exchange of binary objects between devices, used in Bluetooth.

OEM—Original Equipment Manufacturer.

OFA—OpenFabrics Alliance.

OCF—Open Cryptography Framework.

OHA—Open Handset Alliance.

OOTB—Out of the Blue packet (a term of the SCTP protocol). A packet is an OOTB packet if it is correctly formed (that is, no checksum error), but the receiver is not able to identify the SCTP association to which the packet belongs (see section 8.4 in RFC 4960).

OPP—Object Push Profile. Used by Bluetooth.

OSI Model—Open Systems Interconnection.

OSPF—Open Shortest Path First. Interior gateway routing protocol developed for IP networks.

PADI—PPPoE Active Discovery Initiation.

PADO—PPPoE Active Discovery Offer.

PADR—PPPoE Active Discovery Request.

PADS—PPPoE Active Discovery Session.

PADT—PPPoE Active Discovery Terminate.

PAN—Personal Area Networking. A profile used in Bluetooth.

PCI—Peripheral Component Interconnect. A bus for attaching devices. Many network interface cards are PCI devices.

PD—Protection Domain.

PHDC—Personal Health Device Communication. Used by NFC.

PID—Process Identifier.

PIM—Protocol Independent Multicast Protocol. A multicast routing protocol.

PIM-SM—Protocol Independent Multicast—Sparse Mode.

PLME—Physical Layer Management Entity in IEEE 802.11.

PM—Power Management.

PPP—Point To Point data link protocol. A protocol for direct communication between two hosts.

PPPoE—PPP over Ethernet. The PPPoE protocol is specified in RFC 2516 from 1999.

PERR—Path Error. A message that informs about some failure in a wireless Mesh network routing.

PREP—Path Reply. A unicast packet sent as a reply to a PREQ message in a wireless Mesh network.

PREQ—Path Request. A broadcast packet sent when looking for some address in a wireless Mesh network.

PSK—Preshared Key.

Qdisc—Queuing Disciplines.

QP—Queue Pair (InfinBand).

RA—Router Alert. One of the IPv4 options. It notifies transit routers to more closely examine the contents of an IP packet. It is used by many protocols, such as IGMP, MLD, and more.

RANN—Root Announcement. A broadcast packet sent periodically by a Root Mesh point in a wireless Mesh network.

RARP—Reverse Address Resolution Protocol. A protocol used to find the mapping between a link layer address (like a 48-bit Ethernet address) to a network address (like an IPv4 address).

RC—A QP transport type in InfiniBand.

RDMA—Remote Direct Memory Access. A direct memory access from one host to another.

RDS—Reliable Datagram Socket. A reliable connectionless protocol developed by Oracle.

RFC—Request For Comments. A document that specifies Internet specifications, communications protocols, procedures, and events. The standardization process of RFCs is documented at http://tools.ietf.org/html/rfc2026, "The Internet Standards Process."

RFID—Radio Frequency ID.

RFCOMM—Radio Frequency Communications protocol. Used in Bluetooth.

RFS—Receive Flow Steering.

RIP—Routing Information Protocol: A distance-vector routing protocol.

RoCE—RDMA over Converged Ethernet.

RP—Rendezvous Point.

RPL—IPv6 Routing Protocol for Low-Power and Lossy Networks. The RPL protocol is specified in RFC 6550.

RPDB—Routing Policy DataBase.

RPF—Reverse Path Filter. A technique intended to prevent source address spoofing.

RPC—Remote Procedure Call.

RPS—Receive Packet Steering.

RS—Router Solicitations.

RSA—A cryptography algorithm. RSA stands for Ron Rivest, Adi Shamir, and Leonard Adleman, the people who developed it.

RTP—Real-time Transport Protocol. A protocol for transmitting audio and video over IP networks.

RTR—Ready To Receive. A state in InfiniBand QP State Machine.

RTS—Ready To Send. A state in InfiniBand QP State Machine.

SA—Security Association. A logical relationship between two hosts that consists of various parameters, such as cryptographic key, cryptographic algorithm, SPI, and more.

SACK—Selective Acknowledgments. See RFC 2018, "TCP Selective Acknowledgment Options," from 1996.

SAD—Security Association Database.

SAR—Segmentation and Reassembly.

SBC—Session Border Controllers.

SCO—Synchronous Connection Oriented link. A Bluetooth protocol.

SDP—Service Discovery Protocol. Used in Bluetooth.

SCTP—Stream Control Transmission Protocol. A transport protocol that has features of both UDP and TCP.

SE—Security Element (NFC).

SIG—Special Interest Group.

SIP—Session Initiation Protocol. A signaling protocol for VoIP, intended for creating and modifying VoIP sessions.

SLAAC—Stateless Address autoconfiguration. Specified in RFC 4862.

SKB —Socket Buffer. A kernel data structure representing a network packet (implemented by the `sk_buff` structure, `include/linux/skbuff.h`).

SL—Service Level. The QoS in InfiniBand is implemented using the SL to VL mapping and the resources for each VL.

SLAAC—Stateless Address Autoconfiguration.

SM—Subnet Manager.

SMA—Subnet Management Agent.

SME—System Management Entity in IEEE 802.11.

SMP—Symmetrical Multiprocessing. An architecture where two or more identical processors are connected to a single shared main memory.

SNAT—Source NAT. A NAT that changes the source address.

SNEP—Simple NDEF Exchange Protocol (SNEP) for exchanging NDEF-formatted data.

SNMP—Simple Network Management Protocol.

SPI—Security Parameter Index. Used by IPsec.

SPD—Security Policy Database.

SQD—Send Queue Drained. A state in InfiniBand QP State Machine.

SQE—Send Queue Error. A state in InfiniBand QP State Machine.

SRP—SCSI RDMA protocol.

SR-IOV—Single Root I/O Virtualization. A specification that allows a PCIe device to appear to be multiple separate physical PCIe devices.

SRQ—Shared Receive Queue (InfiniBand).

SSM—Source Specific Multicast.

STUN —Session Traversal Utilities for NAT.

SSP—Secure Simple Pairing. A security feature required by Bluetooth v2.1.

TCP—Transmission Control Protocol. The TCP protocol is the most commonly used transport protocol on the Internet today. Many protocols run on top of TCP, including FTP, HTTP, and more. TCP is specified in RFC 793 from 1981, and during the years since then there have been many protocol updates, variations, and additions to the base TCP protocol.

TIPC—Transparent Inter-process Communication protocol.
See http://tipc.sourceforge.net/.

TOS —Type Of Service.

TSO—TCP Segmentation Offload.

TTL—Time To Live. A counter in the IPv4 header (its counterpart in IPv6 is called Hop Limit) that is decremented in each forwarding device. When this counter reaches 0, an ICMP of Time Exceeded is sent back, and the packet is discarded. Both the `ttl` member of the IPv4 header and the `hop_limit` member of the IPv6 header are 8-bit fields.

TURN—Traversal Using Relays around NAT.

UC—Unreliable Connected. A QP transport type in InfiniBand.

UD—Unreliable Datagram. A QP transport type in InfiniBand.

UDP—User Datagram Protocol. UDP is an unreliable protocol, as there is no guarantee that packets will be delivered for upper layer protocols. There is no handshaking phase in UDP, in contrast to TCP. The UDP header is simple and consists of only 4 fields: source port, destination port, checksum, and length.

USAGI—UniverSAl playGround for Ipv6. A project that developed IPv6 and IPsec (for both IPv4 and IPv6) stacks for the Linux kernel.

UTS—Unix Time-sharing System.

VCRC—Variant CRC. An InfiniBand header of 2 bytes. Covers all the fields of the packet.

VETH—Virtual Ethernet. A network driver which enables communication between two network devices in different network namespaces.

VoIP—Voice Over IP.

VFS—Virtual File System.

VL—Virtual Lanes. A mechanism for creating multiple virtual links over a single physical link.

VLAN—Virtual Local Area Network.

VPN—Virtual Private Network.

VXLAN—Virtual Extensible Local Area Network. VXLAN is a standard protocol to transfer Layer 2 Ethernet packets over UDP. VXLAN is needed because there are cases where firewalls block tunnels and allow, for example, only TCP/UDP traffic.

WDS—Wireless Distribution System.

WLAN—Wireless LAN.

WOL—Wake On LAN.

WSN—Wireless Sensor Networks.

XRC—eXtended Reliable Connected. A QP transport type in InfiniBand.

XFRM—IPsec Transformer. A Linux kernel framework for handling IPsec transformations. The two most fundamental data structures of the XFRM framework are the XFRM policy and the XFRM state.

Index

A

B

C

D

E

F

G

H

I, J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X, Y

Z

Linux Kernel Networking: Implementation and Theory

ISBN-13 (pbk): 978-1-4302-6196-4

ISBN-13 (electronic): 978-1-4302-6197-1

President and Publisher: Paul Manning
Lead Editor: Michelle Lowman
Technical Reviewer: Brendan Horan
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Louise Corrigan, James DeWolf, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Steve Weiss, Tom Welsh
Coordinating Editor: Kevin Shea
Copy Editor: Corbin Collins
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code.

To Dr. Joseph Shapira, Qualcomm Israel Founder and Ex-President, coauthor of "CDMA Radio with Repeaters"(Springer, 2007).

To Dr Ruth Shapira.

Iris & Dr. Shye Shapira, made of the stuff dreams are made of.

—Rami Rosen

About the Author

Rami Rosen is a software engineer, a computer science graduate of the Technion, Israel High Institute of Technology. In the last 17 years he has been a software developer for three innovative startups and a semiconductor company. Rami lives in Israel and he has participated in highly advanced Linux kernel projects, in particular those related to networking. He has published several articles and given lectures about Linux kernel networking and virtualization.

About the Technical Reviewer

Brendan Horan is a hardware fanatic, with a full high rack of all types of machine architectures in his home. He has more than ten years of experience working with large UNIX systems and tuning the underlying hardware for optimal performance and stability. Brendan's love for all forms of hardware has helped him throughout his IT career, from fixing laptops to tuning servers and their hardware in order to suit the needs of high-availability designs and ultra low-latency applications. Brendan takes pride in the open source movement and is happy to say that every computer in his house is powered by open source technology. He resides in Hong Kong with his wife, Vikki, who continues daily to teach him more Cantonese.

Acknowledgments

Thanks to my editors for giving me the honor of writing this book; to Michelle Lowman, the lead editor, for believing in this book while it was still just an idea; to Kevin Shea, the coordinating editor, who guided and supported me from the initial stages until the book was fully realized; to Brendan Horan, the technical reviewer, for his helpful comments that helped me to improve the book by a lot; to Troy Mott, the development editor, for his many suggestions and for his hard work; to Corbin Collins and Roger LeBlanc, the copy editors, for shaping up the text; and to Kumar Dhaneesh from the production team.

I would like to thank the Linux kernel networking maintainer, David Miller, for the great work he has done over all these years and all the developers who continue to participate and contribute to the networking subsystem. I would like also to say thanks to the Linux kernel networking community and all its members who helped me by reviewing my text: Julian Anastasov, Timo Teras, Steffen Klassert, Gerrit Renker, Javier Cardona, Gao feng, Vlad Yasevich, Cong Wang, Florian Westphal, Reuben Hawkins, Pekka Savola, Andreas Steffen, Daniel Borkmann, Joachim Nilsson, David Hauweele, Maxime Ripard, Alexandre Belloni, Benjamin Zores, and too many others to mention. Thanks to Donald Wood and Eliezer Tamir from Intel for their help with the "Busy Polling Sockets" section, and to Samuel Ortiz from Intel for his advice in preparing the NFC section. Thanks for Dotan Barak, an InfiniBand expert, for contributing Chapter 13, "InfiniBand."

Rami Rosen

Preface

This book takes you on a guided, in-depth tour of the current Linux kernel networking implementation and the theory behind it. For almost a decade, no new book about Linux networking has been written. A decade of dynamic and fast-paced Linux kernel development is quite a long time. There are important kernel networking subsystems that are not described in any other book; for example, IPv6, IPsec, Wireless (IEEE 802.11), IEEE 802.15.4, NFC, InfiniBand, and more. There is also very little information on the Web about the implementation details of these subsystems. For all these reasons, I have written this book.

About ten years ago I made my first steps in kernel programming. I was a developer in a startup taking part in a VoIP project for a Linux-based set-top box (STB). There were crashes in the USB stack with some USB cameras, and we had to delve into the code to try to find a solution, because the vendors of that STB did not want to spend time to solve the problem. In fact, it was not that they did not want to, they simply did not know how to. In these days, there was almost no documentation about the USB stack. The *Linux Device Drivers* book from O'Reilly in those days was only in its second edition (the USB chapter was added only in the third edition). Success in that project was crucial for us as a startup. I had learned much about kernel programming in the process of solving the USB crash. Later on we had a project where a NAT traversal solution was needed. The userspace solution was so heavy that the device quickly crashed. When I suggested a kernel solution, my managers were very skeptical, but they did let me try. The kernel solution proved to be very stable and took much less CPU than the userspace solution. Since then I have taken part in many kernel networking projects. This book is a result of my many years of development and research.

Who This Book Is For

This book is intended for computer professionals, including developers, software architects, designers, project managers, and CTOs, who are working on networking-related projects. These projects can be in a wide range of professional areas, such as communication, data centers, embedded devices, virtualization, security, and more. In addition, students and academy researchers and theorists who deal with networking projects or networking research or operating systems research will find a lot of help in this book.

How This Book Is Structured

In Chapter 1 you will find a general overview of the Linux kernel and the Linux network stack. Other topics in this chapter include the implementation of the network device, the socket buffer, and the Rx and Tx paths. Chapter 1 concludes with a section about the Linux Kernel Networking Development Model.

In chapter 2 you will learn about netlink sockets, which provide a mechanism for bidirectional communication between userspace and the kernel, and which are used by the networking subsystem as well as by other subsystems. You will also find a section in this chapter about generic netlink sockets, which can be perceived as advanced netlink sockets, and which you will encounter in Chapter 12 and while browsing the kernel networking source code.

In Chapter 3 you will learn about the ICMP protocol, which helps to keep the system behaving correctly by sending error and control messages about the network layer (L3). You will learn about the implementation of the ICMP protocol both in IPv4 and in IPv6.

Chapter 4 delves into the IPv4 protocol—the Internet and modern life cannot be described without it. You will learn about the structure of IPv4 header, about the Rx and Tx path, about IP options, about fragmentation and defragmentation and why they are needed, and about forwarding packets, which is one of the important tasks of IPv4.

Chapters 5 and 6 are devoted to the IPv4 Routing Subsystem. In chapter 5 you will learn how a lookup in the routing subsystem is performed, how the routing tables are organized, which optimizations are used in the IPv4 routing subsystem and about the removal of the IPv4 routing cache. Chapter 6 discusses advanced routing topics such as Multicast Routing, Policy Routing, and Multipath Routing.

Chapter 7 endeavors to explain the neighbouring subsystem. You will learn about the ARP protocol, which is used in IPv4, and about the the NDISC protocol used in IPv6, and about some of the differences between the two protocols. You will also learn about the Duplicate Address Detection (DAD) mechanism in IPv6.

Chapter 8 discusses the IPv6 protocol, which seems to be the inevitable solution to the shortage of IPv4 addresses. This chapter describes the implementation of IPv6 and discusses topics such as IPv6 addresses, the IPv6 header and extension headers, autoconfiguration in IPv6, Rx path, and forwarding. It also describes the MLD protocol.

Chapter 9 deals with the netfilter subsystem. You will learn about netfilter hooks and how they are registered, about Connection Tracking, about IP tables and Network Address Translation (NAT), and about callback used by Connection Tracking and NAT.

Chapter 10 deals with IPsec, one of the most complex networking subsystems. Topics like the IKE protocol (which is implemented in userspace) and cryptography aspects of IPsec are discussed briefly (full treatment is beyond the scope of the book). You will learn about the XFRM framework, which is the basis of the Linux IPsec subsystem, and about its two most important structures: XFRM policy and XFRM state. The ESP protocol is briefly described, as well as the IPsec Rx path and Tx path in transport mode. The chapter concludes with a section about XFRM lookup and a short section about NAT traversal.

Chapter 11 describes four Layer 4 protocols, starting with the most commonly used protocols, UDP and TCP, and concluding with two newer protocols, SCTP and DCCP.

Chapter 12 deals with wireless in Linux (IEEE 802.11). You will learn about the mac80211 subsystem and its implementation, about various wireless network topologies, about power save mode, and about IEEE 802.11n and packet aggregation. There is also a section devoted to Wireless Mesh networks in this chapter.

Chapter 13 delves into the InfiniBand subsystem, a technology enjoying a rising popularity in datacenters. You will learn about the RDMA stack organization, about addressing in InfiniBand, about the organization of InfiniBand packets, and about the RDMA API.

Chapter 14 concludes the book with a discussion of advanced topics such as Linux namespaces and network namespaces in particular, Busy Poll Sockets, the Bluetooth subsystem, the IEEE 802.15.4 subsystem, the Near Field Communication (NFC) subsystem, the PCI subsystem, and more.

Appendices A, “Linux API,” and C, “Glossary ,” provide complete reference information for many topics dicussed in the book. Appendix B, “Network Administration,” provides information about various tools which you will need while working with Linux kernel networking.

Conventions

Throughout the book, I’ve kept a consistent style. All code snippets, whether inside text paragraphs or on lines of their own, along with library paths, shell commands, URLs, and other code-related elements, are set in monospaced font, like this. New terms are set off in *italics*, and other emphasis may be given in **bold**.