CHAPTER 1

Introduction

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

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

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

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

The Linux Kernel Networking stack is a very important subsystem of the Linux kernel. It is quite difficult to find a Linux-based system, whether it is a desktop, a server, a mobile device or any other embedded device, that does not use any kind of networking. Even in the rare case when a machine doesn't have any hardware network devices, you will still be using networking (maybe unconsciously) when you use X-Windows, as X-Windows itself is based upon client-server networking. A wide range of projects are related to the Linux Networking stack, from core routers to small embedded devices. Some of these projects deal with adding vendor-specific features. For example, some hardware vendors implement Generic Segmentation Offload (GSO) in some network devices. GSO is a networking feature of the kernel network stack that divides a large packet into smaller ones in the Tx path. Many hardware vendors implement checksumming in hardware in their network devices. *Checksum* is a mechanism to verify that a packet was not

damaged on transit by calculating some hash from the packet and attaching it to the packet. Many projects provide some security enhancements for Linux. Sometimes these enhancements require some changes in the networking subsystem, as you will see, for example, in Chapter 3, when discussing the Openwall GNU/*/Linux project. In the embedded device arena there are, for example, many wireless routers that are Linux based; one example is the WRT54GL Linksys router, which runs Linux. There is also an open source, Linux-based operating system that can run on this device (and on some other devices), named OpenWrt, with a large and active community of developers (see https://openwrt.org/). Learning about how the various protocols are implemented by the Linux Kernel Networking stack and becoming familiar with the main data structures and the main paths of a packet in it are essential to understanding it better.

The Linux Network Stack

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

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

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

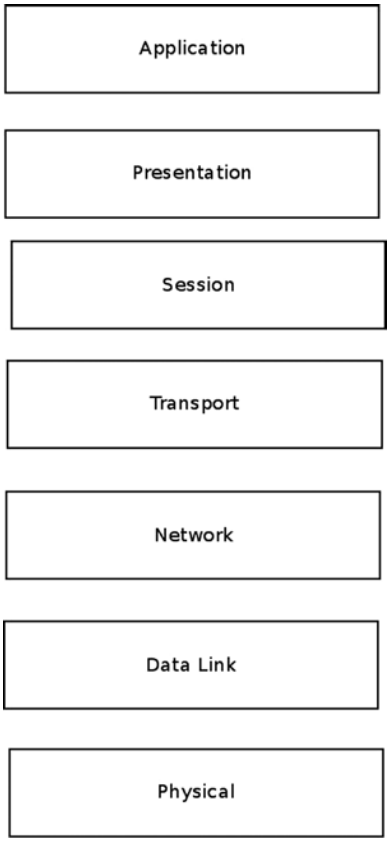

Figure 1-1. *The OSI seven-layer model*

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

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

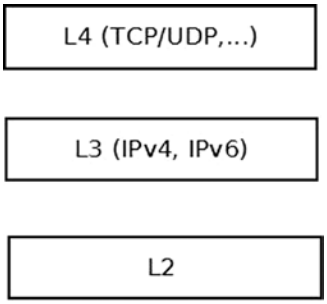

Figure 1-2. *The Linux Kernel Networking layers*

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

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

The Network Device

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

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

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

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

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

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

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

New API (NAPI) in Network Devices

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

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

Receiving and Transmitting Packets

The main tasks of the network device driver are these:

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

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

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

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

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

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

The Socket Buffer

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

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

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

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

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

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

Figure 1-3. *An IPv4 packet*

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

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

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

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

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

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

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

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

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

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

Virtualization solutions are also becoming popular, especially due to projects like Xen or KVM. Also hardware improvements, like VT-x for Intel processors or AMD-V for AMD processors, have made virtualization more efficient. There is another form of virtualization, which may be less known but has its own advantages. This virtualization is based on a different approach: process virtualization. It is implemented in Linux by namespaces. There is currently support for six namespaces in Linux, and there could be more in the future. The namespaces feature is already used by projects like Linux Containers (`http://lxc.sourceforge.net/`) and Checkpoint/Restore In Userspace (CRIU). In order to support namespaces, two system calls were added to the kernel: `unshare()` and `setns()`; and six new flags were added to the CLONE_* flags, one for each type of namespace. I discuss namespaces and network namespaces in particular in Chapter 14. Chapter 14 also deals with the Bluetooth subsystem and gives a brief overview about the PCI subsystem, because many network device drivers are PCI devices. I do not delve into the PCI subsystem internals, because that is out of the scope of this book. Another interesting subsystem discussed in Chapter 14 is the IEEE 8012.15.4, which is for low-power and low-cost devices. These devices are sometimes mentioned in conjunction with the *Internet of Things* (IoT) concept, which involves connecting IP-enabled embedded devices

to IP networks. It turns out that using IPv6 for these devices might be a good idea. This solution is termed IPv6 over Low Power Wireless Personal Area Networks (6LoWPAN). It has its own challenges, such as expanding the IPv6 Neighbour Discovery protocol to be suitable for such devices, which occasionally enter sleep mode (as opposed to ordinary IPv6 networks). These changes to the IPv6 Neighbour Discovery protocol have not been implemented yet, but it is interesting to consider the theory behind these changes. Apart from this, in Chapter 14 there are sections about other advanced topics like NFC, cgroups, Android, and more.

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

The Linux Kernel Networking Development Model

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

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

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

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

And I recommended using the following PERL scripts:

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

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

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

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

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

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

Summary

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