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.

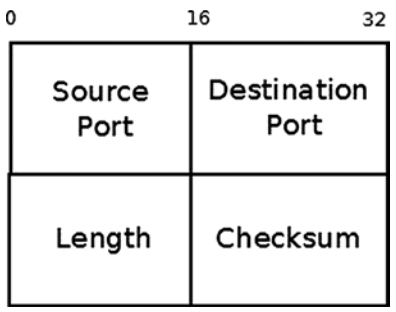

***Figure 11-1.*** *A UDP header (IPv4)*

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)
```

```
static int __init inet_init(void)
{
        . . .
        if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
                pr_crit("%s: Cannot add UDP protocol\n", __func__);
        . . .
}
(net/ipv4/af_inet.c)
```

We further define a `udp_prot` object and register it by calling the `proto_register()` method. This object contains mostly callbacks; these callbacks are invoked when opening a UDP socket in userspace and using the socket API. For example, calling the `setsockopt()` system call on a UDP socket will invoke the `udp_setsockopt()` callback.

```
struct proto udp_prot = {
         .name              = "UDP",
         .owner             = THIS_MODULE,
         .close             = udp_lib_close,
         .connect           = ip4_datagram_connect,
         .disconnect        = udp_disconnect,
         .ioctl             = udp_ioctl,
         . . .
         .setsockopt        = udp_setsockopt,
         .getsockopt        = udp_getsockopt,
         .sendmsg           = udp_sendmsg,
         .recvmsg           = udp_recvmsg,
         .sendpage          = udp_sendpage,
         . . .
};

(net/ipv4/udp.c)
int __init inet_init(void)
{
    int rc = -EINVAL;
    . . .
    rc = proto_register(&udp_prot, 1);
    . . .

}
(net/ipv4/af_inet.c)
```

■ **Note** The UDP protocol, along with other core protocols, is initialized via the `inet_init()` method at boot-time.

Now that you know about UDP initialization and its callback for sending packets, which is the `udp_sendmsg()` callback of the `udp_prot` object that was shown in this section, it is time to learn how packets are sent by UDP in IPV4.

## Sending Packets with UDP

Sending data from a UDP userspace socket can be done by several system calls: send(), sendto(), sendmsg(), and write(); eventually all of them are handled by the udp_sendmsg() method in the kernel. The userspace application builds a msghdr object that contains the data blocks and passes this msghdr object to the kernel. Let's take a look at this method:

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

In general, UDP packets are sent immediately. This behavior can be changed with the UDP_CORK socket option (introduced in kernel 2.5.44), which causes packet data passed to the udp_sendmsg() method to be accumulated until the final packet is released by unsetting the option. The same result can be achieved by setting the MSG_MORE flag:

```
        int corkreq = up->corkflag || msg->msg_flags&MSG_MORE;
        struct inet_sock *inet = inet_sk(sk);
              . . .
```

First we make some sanity checks. The specified len, for example, cannot be greater than 65535 (remember that the len field in the UDP header is 16 bits):

```
        if (len > 0xFFFF)
                 return -EMSGSIZE;
```

We need to know the destination address and the destination port in order to build a flowi4 object, which is needed for sending the SKB with the udp_send_skb() method or with the ip_append_data() method. The destination port should not be 0. There are two cases here: the destination is specified in the msg_name of the msghdr, or the socket is connected and its state is TCP_ESTABLISHED. Note that UDP (in contrast to TCP) is almost a fully stateless protocol. The notion of TCP_ESTABLISHED in UDP mostly means that the socket has passed some sanity checks.

```
        if (msg->msg_name) {
                struct sockaddr_in *usin = (struct sockaddr_in *)msg->msg_name;
                if (msg->msg_namelen < sizeof(*usin))
                        return -EINVAL;
                if (usin->sin_family != AF_INET) {
                        if (usin->sin_family != AF_UNSPEC)
                                return -EAFNOSUPPORT;
                }

                daddr = usin->sin_addr.s_addr;
                dport = usin->sin_port;
```

Linux code honors the fact that zero UDP/TCP ports are reserved by the IANA. The reservation of port 0 in TCP and UDP dates back to RFC 1010, "Assigned Numbers" (1987), and it was still present in RFC 1700, which was obsoleted by the online database (see RFC 3232), where they are still present. See www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml.

```
                if (dport == 0)
                        return -EINVAL;
        } else {
                if (sk->sk_state != TCP_ESTABLISHED)
                        return -EDESTADDRREQ;
                daddr = inet->inet_daddr;
                dport = inet->inet_dport;
                /* Open fast path for connected socket.
                   Route will not be used, if at least one option is set.
                 */
                connected = 1;
}

               . . .
```

A userspace application can send control information (also known as *ancillary data*) by setting `msg_control` and `msg_controllen` in the `msghdr` object. Ancillary data is, in fact, a sequence of `cmsghdr` objects with appended data. (For more details, see `man 3 cmsg`.) You can send and receive ancillary data by calling the `sendmsg()` and `recvmsg()` methods, respectively. For example, you can create an IP_PKTINFO ancillary message to set a source route to an unconnected UDP socket. (See `man 7 ip`.) When `msg_controllen` is not 0, this is a control information message, which is handled by the `ip_cmsg_send()` method. The `ip_cmsg_send()` method builds an `ipcm_cookie` (IP Control Message Cookie) object by parsing the specified `msghdr` object. The `ipcm_cookie` structure includes information that is used further when processing the packet. For example, when using an IP_PKTINFO ancillary message, you can set the source address by setting an address field in the control messages, which eventually sets the `addr` in the `ipcm_cookie` object. The `ipcm_cookie` is a short structure:

```
struct ipcm_cookie {
        __be32                  addr;
        int                     oif;
        struct ip_options_rcu   *opt;
        __u8                    tx_flags;
};
(include/net/ip.h)
```

Let's continue our discussion of the udp_sendmsg() method:

```
         if (msg->msg_controllen) {
                 err = ip_cmsg_send(sock_net(sk), msg, &ipc);
                 if (err)
                         return err;
                 if (ipc.opt)
                         free = 1;
                 connected = 0;
         }
         . . .
         if (connected)
                 rt = (struct rtable *)sk_dst_check(sk, 0);
         . . .
```

If the routing entry is NULL, a routing lookup should be performed:

```
        if (rt == NULL) {
                struct net *net = sock_net(sk);

                fl4 = &fl4_stack;
                flowi4_init_output(fl4, ipc.oif, sk->sk_mark, tos,
                                   RT_SCOPE_UNIVERSE, sk->sk_protocol,
                                   inet_sk_flowi_flags(sk)|FLOWI_FLAG_CAN_SLEEP,
                                   faddr, saddr, dport, inet->inet_sport);
                security_sk_classify_flow(sk, flowi4_to_flowi(fl4));
                rt = ip_route_output_flow(net, fl4, sk);
                if (IS_ERR(rt)) {
                        err = PTR_ERR(rt);
                        rt = NULL;
                        if (err == -ENETUNREACH)
                                IP_INC_STATS_BH(net, IPSTATS_MIB_OUTNOROUTES);
                        goto out;
                }

        . . .
```

In kernel 2.6.39, a lockless transmit fast path was added. This means that when the corking feature is not set, we do not hold the socket lock and we call the `udp_send_skb()` method, and when the corking feature is set, we hold the socket lock by calling the `lock_sock()` method and then send the packet:

```
        /* Lockless fast path for the non-corking case. */
        if (!corkreq) {
                skb = ip_make_skb(sk, fl4, getfrag, msg->msg_iov, ulen,
                                  sizeof(struct udphdr), &ipc, &rt,
                                  msg->msg_flags);
                err = PTR_ERR(skb);
                if (!IS_ERR_OR_NULL(skb))
                         err = udp_send_skb(skb, fl4);
                 goto out;
        }
```

Now we handle the case when the corking feature is set:

```
       lock_sock(sk);
do_append_data:
        up->len += ulen;
```

The `ip_append_data()` method buffers the data for transmission but does not transmit it yet. Subsequently calling the `udp_push_pending_frames()` method will actually perform the transmission. Note that the `udp_push_pending_frames()` method also handles fragmentation by the specified `getfrag` callback:

```
        err = ip_append_data(sk, fl4, getfrag, msg->msg_iov, ulen,
                             sizeof(struct udphdr), &ipc, &rt,
                             corkreq ? msg->msg_flags|MSG_MORE : msg->msg_flags);
```

If the method failed, we should flush all pending SKBs. This is achieved by calling the `udp_flush_pending_frames()` method, which will free all the SKBs in the write queue of the socket (`sk_write_queue`) by the `ip_flush_pending_frames()` method:

```
        if (err)
                udp_flush_pending_frames(sk);
        else if (!corkreq)
                err = udp_push_pending_frames(sk);
        else if (unlikely(skb_queue_empty(&sk->sk_write_queue)))
                up->pending = 0;
        release_sock(sk);
```

You learned in this section about sending packets with UDP. Now, to complete our discussion about UDP in IPv4, it's time to learn about how packets from the network layer (L3) are received with UDP in IPv4.

## Receiving Packets from the Network Layer (L3) with UDP

The main handler for receiving UDP packets from the network layer (L3) is the `udp_rcv()` method. All it does is invoke the `__udp4_lib_rcv()` method (`net/ipv4/udp.c`):

```
int udp_rcv(struct sk_buff *skb)
{
        return __udp4_lib_rcv(skb, &udp_table, IPPROTO_UDP);
}
```

Let's take a look at the `__udp4_lib_rcv()` method:

```
int __udp4_lib_rcv(struct sk_buff *skb, struct udp_table *udptable,
                   int proto)
{
        struct sock *sk;
        struct udphdr *uh;
        unsigned short ulen;
        struct rtable *rt = skb_rtable(skb);
        __be32 saddr, daddr;
        struct net *net = dev_net(skb->dev);
        . . .
```

We fetch the UDP header, header length, and source and destination addresses from the SKB:

```
        uh   = udp_hdr(skb);
        ulen = ntohs(uh->len);
        saddr = ip_hdr(skb)->saddr;
        daddr = ip_hdr(skb)->daddr;
```

We will skip some sanity checks that are being performed, like making sure that the UDP header length is not greater than the length of the packet and that the specified `proto` is the UDP protocol identifier (IPPROTO_UDP). If the packet is a broadcast or a multicast packet, it will be handled by the `__udp4_lib_mcast_deliver()` method:

```
        if (rt->rt_flags & (RTCF_BROADCAST|RTCF_MULTICAST))
             return __udp4_lib_mcast_deliver(net, skb, uh,
                                                    saddr, daddr, udptable);
```

Next we perform a lookup in the UDP sockets hash table:

```
        sk = __udp4_lib_lookup_skb(skb, uh->source, uh->dest, udptable);
              if (sk != NULL) {
```

We arrive here because the lookup we performed found a matching socket. So process the SKB further by calling the `udp_queue_rcv_skb()` method, which invokes the generic `sock_queue_rcv_skb()` method, which in turn adds the specified SKB to the tail of `sk->sk_receive_queue` (by calling the `__skb_queue_tail()` method):

```
int ret = udp_queue_rcv_skb(sk, skb);
sock_put(sk);

/* a return value > 0 means to resubmit the input, but
* it wants the return to be -protocol, or 0
*/
if (ret > 0)
     return -ret;
```

Everything is fine; return 0 to denote success:

```
    return 0;
}
. . .
```

We arrived here because the lookup for a socket failed. This means that we should not handle the packet. This can occur, for example, when there is no listening UDP socket on the destination port. If the checksum is incorrect, we should drop the packet silently. If it is correct, we should send an ICMP reply back to the sender. This should be an ICMP message of “Destination Unreachable” with code of “Port Unreachable.” Further on, we should free the packet and update an SNMP MIB counter:

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

The next command increments the UDP_MIB_NOPORTS (`NoPorts`) MIB counter. Note that you can query various UDP MIB counters by `cat /proc/net/snmp` or by `netstat -s`.

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

/*
* Hmm.  We got an UDP packet to a port to which we
* don't wanna listen.  Ignore it.
*/
kfree_skb(skb);
return 0;
```

Figure 11-2 illustrates our discussion in this section about receiving UDP packets.

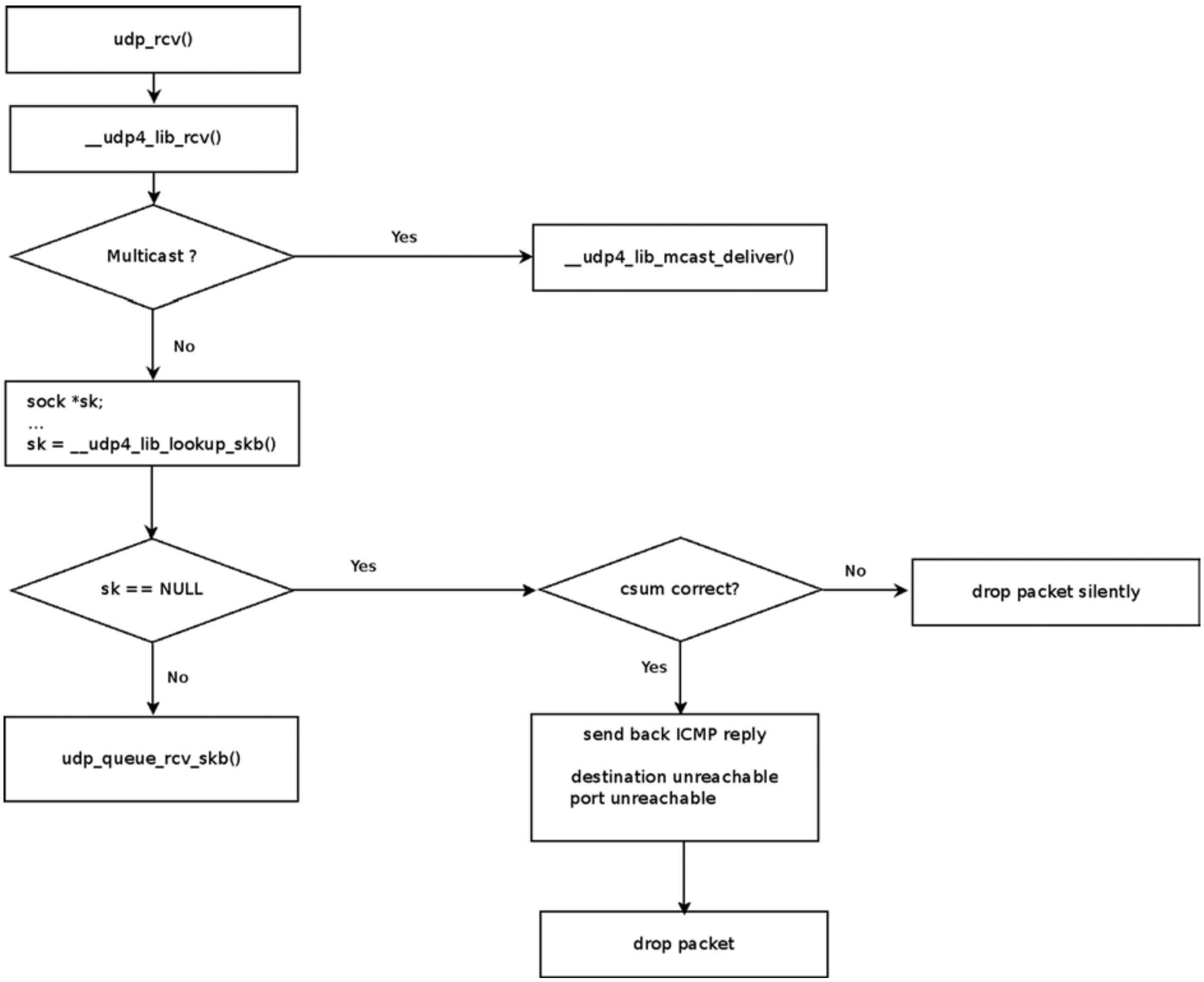

***Figure 11-2.*** *Receiving UDP packets*

Our discussion about UDP is now finished. The next section describes the TCP protocol, which is the most complex among the protocols discussed in this chapter.

## TCP (Transmission Control Protocol)

The TCP protocol is described in RFC 793 from 1981. During the years since then, there have been many updates, variations, and additions to the base TCP protocol. Some additions were for specific types of networks (high-speed, satellite), whereas others were for performance improvements.

The TCP protocol is the most commonly used transport protocol on the Internet today. Many well-known protocols are based upon TCP. The most well-known protocol is probably HTTP, and we should also mention here some other well-known protocols such as `ftp`, `ssh`, `telnet`, `smtp`, and `ssl`. The TCP protocol provides a reliable and connection-oriented transport, as opposed to UDP. Transmission is made reliable by using sequence numbers and acknowledgments.

TCP is a very complex protocol; we will not discuss all the details, optimizations, and nuances of the TCP implementation in this chapter, as this requires a separate book in itself. TCP functionality consists of two ingredients: management of connections, and transmitting and receiving data. We will focus in this section on TCP initialization and TCP connection setup, which pertains to the first ingredient, connections management, and on receiving and sending packets, which pertains to the second ingredient. These are the important basics that enable further delving into the TCP protocol implementation. We should note that the TCP protocol self-regulates the byte-stream flow via congestion control. Many different congestion-control algorithms have been specified, and Linux provides a pluggable and configurable architecture to support a wide variety of algorithms. Delving into the details of the individual congestion-control algorithms is beyond the scope of this book.

Every TCP packet starts with a TCP header. You must learn about the TCP header in order to understand the operation of TCP. The next section describes the IPv4 TCP header.

## TCP Header

The TCP header length is 20 bytes, but it is scalable up to 60 bytes when using TCP options:

```
struct tcphdr {
        __be16  source;
        __be16  dest;
        __be32  seq;
        __be32  ack_seq;
#if defined(__LITTLE_ENDIAN_BITFIELD)
        __u16   res1:4,
                doff:4,
                fin:1,
                syn:1,
                rst:1,
                psh:1,
                ack:1,
                urg:1,
                ece:1,
                cwr:1;
#elif defined(__BIG_ENDIAN_BITFIELD)
        __u16   doff:4,
                res1:4,
                cwr:1,
                ece:1,
                urg:1,
                ack:1,
                psh:1,
                rst:1,
                syn:1,
                fin:1;
#else
#error  "Adjust your <asm/byteorder.h> defines"
#endif
        __be16  window;
        __sum16 check;
        __be16  urg_ptr;
};

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

The following is a description of the members of the `tcphdr` structure:

- `source`: The source port (16 bit), in the range 1-65535.
- `dest`: The destination port (16 bit), in the range 1-65535.
- `seq`: The Sequence number (32 bits).
- `ack_seq`: Acknowledgment number (32 bits). If the ACK flag is set, the value of this field is the next sequence number that the receiver is expecting.
- `res1`: Reserved for future use (4 bits). It should always be set to 0.
- `doff`: Data offset (4 bits). The size of the TCP header in multiplies of 4 bytes; the minimum is 5 (20 bytes) and the maximum is 15 (60 bytes).

The following are the TCP flags; each is 1 bit:

- `fin`: No more data from sender (when one of the endpoints wants to close the connection).
- `syn`: The SYN flag is initially sent when establishing the 3-way handshake between two endpoints.
- `rst`: The Reset flag is used when a segment that is not intended for the current connection arrives.
- `psh`: The data should be passed to userspace as soon as possible.
- `ack`: Signifies that the acknowledgment number (`ack_seq`) value in the TCP header is meaningful.
- `urg`: Signifies that the urgent pointer is meaningful.
- `ece`: ECN - Echo flag. *ECN* stands for "Explicit Congestion Notification." ECN provides a mechanism that sends end-to-end notification about network congestion without dropping packets. It was added by RFC 3168, "The Addition of Explicit Congestion Notification (ECN) to IP," from 2001.
- `cwr`: Congestion Window Reduced flag.
- `window`: TCP receive window size in bytes (16 bit).
- `check`: Checksum of the TCP header and TCP data.
- `urg_ptr`: Has significance only when the `urg` flag is set. It represents an offset from the sequence number indicating the last urgent data byte (16 bit).

Figure 11-3 shows a diagram of a TCP header.

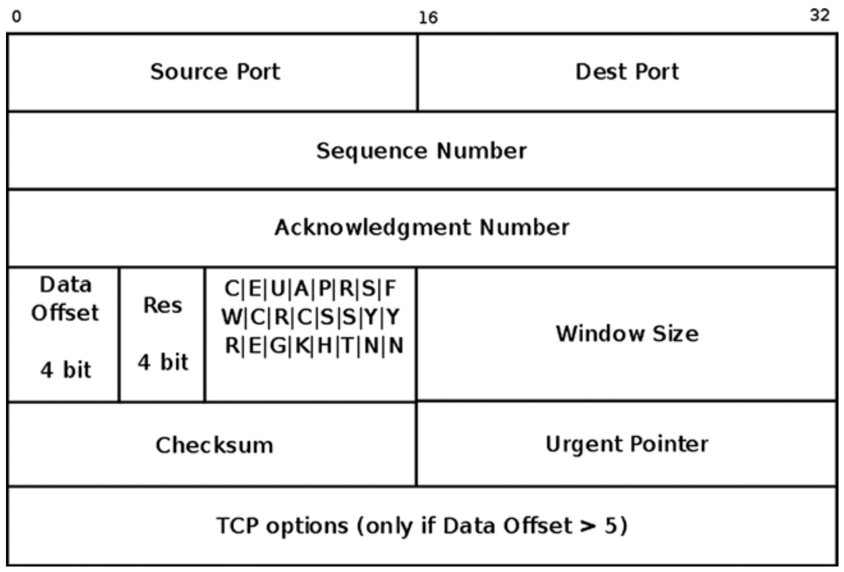

***Figure 11-3.*** *TCP header (IPv4)*

In this section, I described the IPv4 TCP header and its members. You saw that, as opposed to the UDP header, which has only 4 members, the TCP header has a lot more members, since TCP is a much more complex protocol. In the following section, I will describe how TCP initialization is done so that you will learn how and where the initialization of the callbacks for receiving and sending TCP packets takes place.

## TCP Initialization

We define the `tcp_protocol` object (`net_protocol` object) and add it with the `inet_add_protocol()` method:

```
static const struct net_protocol tcp_protocol = {
        .early_demux    =       tcp_v4_early_demux,
        .handler        =       tcp_v4_rcv,
        .err_handler    =       tcp_v4_err,
        .no_policy      =       1,
        .netns_ok       =       1,
};

(net/ipv4/af_inet.c)

static int __init inet_init(void)
  {
        . . .
        if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)
            pr_crit("%s: Cannot add TCP protocol\n", __func__);
        . . .
}

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

We further define a `tcp_prot` object and register it by calling the `proto_register()` method, like what we did with UDP:

```
struct proto tcp_prot = {
        .name                   = "TCP",
        .owner                  = THIS_MODULE,
        .close                  = tcp_close,
        .connect                = tcp_v4_connect,
        .disconnect             = tcp_disconnect,
        .accept                 = inet_csk_accept,
        .ioctl                  = tcp_ioctl,
        .init                   = tcp_v4_init_sock,
        . . .
};

(net/ipv4/tcp_ipv4.c)

static int __init inet_init(void)
{
        int rc;
        . . .
        rc = proto_register(&tcp_prot, 1);
        . . .
}

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

Note that in the `tcp_prot` definition, the `init` function pointer is defined to be the `tcp_v4_init_sock()` callback, which performs various initializations, like setting the timers by calling the `tcp_init_xmit_timers()` method, setting the socket state, and more. Conversely, in UDP, which is a much simpler protocol, the `init` function pointer was not defined at all because there are no special initializations to perform in UDP. We will discuss the `tcp_v4_init_sock()` callback later in this section.

In the next section, I will describe briefly the timers used by the TCP protocol.

## TCP Timers

TCP timers are handled in `net/ipv4/tcp_timer.c`. There are four timers used by TCP:

- **Retransmit timer:** Responsible for resending packets that were not acknowledged in a specified time interval. This can happen when a packet gets lost or corrupted. This timer is started after each segment is sent; if an ACK arrives before the timer expires, the timer is canceled.
- **Delayed ACK timer:** Delays sending ACK packets. It is set when TCP receives data that must be acknowledged but does not need to be acknowledged immediately.
- **Keep Alive timer:** Checks whether the connection is down. There are cases when sessions are idle for a long time and one side goes down. The Keep Alive timer detects such cases and calls the `tcp_send_active_reset()` method to reset the connection.
- **Zero window probe timer (also known as the *persistent timer*):** When the receive buffer is full, the receiver advertises a zero window and the sender stops sending. Now, when a receiver sends a segment with a new window size and this segment is lost, the sender will keep waiting forever. The solution is this: when the sender gets a zero window, it uses a persistent timer to probe the receiver for its window size; when getting a non-zero window size, the persistent timer is stopped.

## TCP Socket Initialization

To use a TCP socket, a userspace application should create a SOCK_STREAM socket and call the `socket()` system call. This is handled in the kernel by the `tcp_v4_init_sock()` callback, which invokes the `tcp_init_sock()` method to do the real work. Note that the `tcp_init_sock()` method performs address-family independent initializations, and it is invoked also from the `tcp_v6_init_sock()` method. The important tasks of the `tcp_init_sock()` method are the following:

- Set the state of the socket to be TCP_CLOSE.
- Initialize TCP timers by calling the `tcp_init_xmit_timers()` method.
- Initialize the socket send buffer (`sk_sndbuf`) and receive buffer (`sk_rcvbuf`); `sk_sndbuf` is set to be to `sysctl_tcp_wmem[1]`, which is by default 16384 bytes, and `sk_rcvbuf` is set to be `sysctl_tcp_rmem[1]`, which is by default 87380 bytes. These default values are set in the `tcp_init()` method; the `sysctl_tcp_wmem` and `sysctl_tcp_rmem` arrays default values can be overridden by writing to `/proc/sys/net/ipv4/tcp_wmem` and to `/proc/sys/net/ipv4/tcp_rmem`, respectively. See the "TCP Variables" section in `Documentation/networking/ip-sysctl.txt`.
- Initialize the out-of-order queue and the `prequeue`.
- Initialize various parameters. For example, the TCP initial congestion window is initialized to 10 segments (TCP_INIT_CWND), according to RFC 6928, "Increasing TCP's Initial Window," from 2013.

Now that you have learned how a TCP socket is initialized, I will discuss how to set up a TCP connection.

## TCP Connection Setup

TCP connection setup and teardown and TCP connection properties are described as transitions in a state machine. At each given moment, a TCP socket can be in one specified state; for example, the socket enters the TCP_LISTEN state when the `listen()` system call is invoked. The state of the `sock` object is represented by its `sk_state` member. For a list of all available states, refer to `include/net/tcp_states.h`.

A three way handshake is used to set up a TCP connection between a TCP client and a TCP server:

- First, the client sends a SYN request to the server. Its state changes to TCP_SYN_SENT.
- The server socket, which is listening (its state is TCP_LISTEN), creates a request socket to represent the new connection in the TCP_SYN_RECV state and sends back a SYN ACK.
- The client that receives the SYN ACK changes its state to TCP_ESTABLISHED and sends an ACK to the server.
- The server receives the ACK and changes the request socket into a child socket in the TCP_ESTABLISHED state, as the connection is now established and data can be sent.

■ **Note** to further look into the TCP state machine details, refer to the `tcp_rcv_state_process()` method (`net/ipv4/tcp_input.c`), which is the state machine engine, both for IPv4 and for IPv6. (It is called both from the `tcp_v4_do_rcv()` method and from the `tcp_v6_do_rcv()` method.)

The next section describes how packets are received from the network layer (L3) with TCP in IPv4.

## Receiving Packets from the Network Layer (L3) with TCP

The main handler for receiving TCP packets from the network layer (L3) is the `tcp_v4_rcv()` method (`net/ipv4/tcp_ipv4.c`). Let's take a look at this function:

```
int tcp_v4_rcv(struct sk_buff *skb)
{
       struct sock *sk;
       . . .
```

First we make some sanity checks (for example, checking to see if the packet type is not PACKET_HOST or if the packet size is shorter than the TCP header) and discard the packet if there are any problems; then some initializations are made and also a lookup for a corresponding socket is performed by calling the `__inet_lookup_skb()` method, which first performs a lookup in the established sockets hash table by calling the `__inet_lookup_established()` method. In the case of a lookup miss, it performs a lookup in the listening sockets hash table by calling the `__inet_lookup_listener()` method. If no socket is found, the packet is discarded at this stage.

```
        sk = __inet_lookup_skb(&tcp_hashinfo, skb, th->source, th->dest);
        . . .
        if (!sk)
                goto no_tcp_socket;
```

Now we check whether the socket is owned by some application. The `sock_owned_by_user()` macro returns 1 when there is currently an application that owns the socket, and it returns a value of 0 when there is no application that owns the socket:

```
        if (!sock_owned_by_user(sk)) {
        . . .
                {
```

We arrive here if no application owns the socket, so it can accept packets. First we try to put the packet in the `prequeue` by calling the `tcp_prequeue()` method, as packets in the `prequeue` are processed more efficiently. The `tcp_prequeue()` will return `false` if processing in the `prequeue` is not possible (for example, when the queue has no space); in such a case, we will call the `tcp_v4_do_rcv()` method, which we will discuss shortly:

```
                if (!tcp_prequeue(sk, skb))
                        ret = tcp_v4_do_rcv(sk, skb);
        }
```

When an application owns the socket, it means that it is in a locked state, so it cannot accept packets. In such a case, we add the packet to the backlog by calling the `sk_add_backlog()` method:

```
                } else if (unlikely(sk_add_backlog(sk, skb,
                                                   sk->sk_rcvbuf + sk->sk_sndbuf))) {
                        bh_unlock_sock(sk);
                        NET_INC_STATS_BH(net, LINUX_MIB_TCPBACKLOGDROP);
                        goto discard_and_relse;
                }
        }
```

Let's take a look at the `tcp_v4_do_rcv()` method:

```
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
{
```

If the socket is in the TCP_ESTABLISHED state, we call the `tcp_rcv_established()` method:

```
        if (sk->sk_state == TCP_ESTABLISHED) { /* Fast path */
        . . .
                if (tcp_rcv_established(sk, skb, tcp_hdr(skb), skb->len)) {
                        rsk = sk;
                        goto reset;
                }
                return 0;
```

If the socket is in the TCP_LISTEN state, we call the `tcp_v4_hnd_req()` method:

```
        if (sk->sk_state == TCP_LISTEN) {
                struct sock *nsk = tcp_v4_hnd_req(sk, skb);

        }
```

If we are not in the TCP_LISTEN state, we invoke the `tcp_rcv_state_process()` method:

```
        if (tcp_rcv_state_process(sk, skb, tcp_hdr(skb), skb->len)) {
                rsk = sk;
                goto reset;
        }
        return 0;

reset:
        tcp_v4_send_reset(rsk, skb);

}
```

In this section, you learned about the reception of a TCP packet. In the next section, we conclude the TCP part of this chapter by describing how packets are sent with TCP in IPv4.

## Sending Packets with TCP

As with UDP, sending packets from TCP sockets that were created in userspace can be done by several system calls: `send()`, `sendto()`, `sendmsg()`, and `write()`. Eventually all of them are handled by the `tcp_sendmsg()` method (`net/ipv4/tcp.c`). This method copies the payload from the userspace to the kernel and sends it as TCP segments. It is much more complicated than the `udp_sendmsg()` method.

```
int tcp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
                size_t size)
{
        struct iovec *iov;
        struct tcp_sock *tp = tcp_sk(sk);
        struct sk_buff *skb;
        int iovlen, flags, err, copied = 0;
        int mss_now = 0, size_goal, copied_syn = 0, offset = 0;
        bool sg;
        long timeo;
        . . .
```

I will not delve into all the details of copying the data from the userspace to the SKB in this method. Once the SKB is built, it is sent with the `tcp_push_one()` method that calls the `tcp_write_xmit()` method, which in turn invokes the `tcp_transmit_skb()` method:

```
static int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb, int clone_it,
                            gfp_t gfp_mask)
{
```

The `icsk_af_ops` object (INET Connection Socket ops) is an address-family specific object. In the case of IPv4 TCP, it is set to be an `inet_connection_sock_af_ops` object named `ipv4_specific` in the `tcp_v4_init_sock()` method. The `queue_xmit()` callback is set to be the generic `ip_queue_xmit()` method. See `net/ipv4/tcp_ipv4.c`.

```
    . . .
    err = icsk->icsk_af_ops->queue_xmit(skb, &inet->cork.fl);
    . . .
}
(net/ipv4/tcp_output.c)
```

Now that you learned about TCP and UDP, you are ready to proceed to the next section which deals with the SCTP (Stream Control Transmission Protocol) protocol. The SCTP protocol combines features of both UDP and TCP, and it is newer than both of them.

## SCTP (Stream Control Transmission Protocol)

The SCTP protocol is specified in RFC 4960 from 2007. It was first specified in 2000. It is designed for Public Switched Telephone Network (PSTN) signaling over IP networks, but it can be used with other applications. The IETF SIGTRAN (Signaling Transport) working group originally developed the SCTP protocol and later handed the protocol over to the Transport Area working group (TSVWG) for the continued evolvement of SCTP as a general-purpose transport protocol. LTE (Long Term Evolution) uses SCTP; one of the main reasons for this is that the SCTP protocol is able to detect when a link goes down or when packets are dropped very quickly, whereas TCP does not have this feature. SCTP flow-control and congestion-control algorithms are very similar in TCP and SCTP. The SCTP protocol uses a variable for the advertised receiver window size (`a_rwnd`); this variable represents the current available space in the receiver buffer. The sender cannot send any new data if the receiver indicates that `a_rwnd` is 0 (no receive space available). The important features of SCTP are the following ones:

- SCTP combines the features of TCP and UDP. It is a reliable transport protocol with congestion control like TCP; it is a message-oriented protocol like UDP, whereas TCP is stream-oriented.
- The SCTP protocol provides improved security with its 4-way handshake (compared to the TCP 3-way handshake) to protect against SYN flooding attacks. I will discuss the 4-way handshake later in this chapter in the “Setting Up an SCTP Association” section.
- SCTP supports multihoming—that is, multiple IP addresses on both endpoints. This provides a network-level, fault-tolerance capability. I will discuss SCTP chunks later in this section.

- SCTP supports multistreaming, which means that it can send in parallel streams of data chunks. This can reduce the latency of streaming multimedia in some environments. I will discuss SCTP chunks later in this section.
- SCTP uses a heartbeat mechanism to detect idle/unreachable peers in the case of multihoming. I will discuss the SCTP heartbeat mechanism later in this chapter.

After this short description of the SCTP protocol, we will now discuss how SCTP initialization is done. The `sctp_init()` method allocates memory for various structures, initializes some `sysctl` variables, and registers the SCTP protocol in IPv4 and in IPv6:

```
int sctp_init(void)
{
       int status = -EINVAL;
        . . .
        status = sctp_v4_add_protocol();

       if (status)
               goto err_add_protocol;

       /* Register SCTP with inet6 layer.  */
       status = sctp_v6_add_protocol();
       if (status)
               goto err_v6_add_protocol;
       . . .
}
```

```
(net/sctp/protocol.c)
```

The registration of the SCTP protocol is done by defining an instance of `net_protocol` (named `sctp_protocol` for IPv4 and `sctpv6_protocol` for IPv6) and calling the `inet_add_protocol()` method, quite similarly to what you saw in other transport protocols, like the UDP protocol. We also call the `register_inetaddr_notifier()` to receive notifications about adding or deleting a network address. These events will be handled by the `sctp_inetaddr_event()` method, which will update the SCTP global address list (`sctp_local_addr_list`) accordingly.

```
 static const struct net_protocol sctp_protocol = {
        .handler     = sctp_rcv,
        .err_handler = sctp_v4_err,
        .no_policy   = 1,
};
```

```
(net/sctp/protocol.c)
```

```
static int sctp_v4_add_protocol(void)
{
        /* Register notifier for inet address additions/deletions. */
        register_inetaddr_notifier(&sctp_inetaddr_notifier);

        /* Register SCTP with inet layer.  */
        if (inet_add_protocol(&sctp_protocol, IPPROTO_SCTP) < 0)
                return -EAGAIN;

        return 0;
}
```

```
(net/sctp/protocol.c)
```

■ **Note** The `sctp_v6_add_protocol()` method (`net/sctp/ipv6.c`) is very similar, so we will not show it here.

Each SCTP packet starts with an SCTP header. I will now describe the structure of an SCTP header. I will start the discussion with SCTP chunks in the next section.

## SCTP Packets and Chunks

Each SCTP packet has an SCTP common header, which is followed by one or more chunks. Each chunk can contain either data or SCTP control information. Several chunks can be bundled into one SCTP packet (except for three chunks that are used when establishing and terminating a connection: INIT, INIT_ACK, and SHUTDOWN_COMPLETE). These chunks use the Type-Length-Value (TLV) format that you first encountered in Chapter 2.

### SCTP Common Header

```
typedef struct sctphdr {
        __be16 source;
        __be16 dest;
        __be32 vtag;
        __le32 checksum;
} __attribute__((packed)) sctp_sctphdr_t;
```

`(include/linux/sctp.h)`

Following is a description of the members of the `sctphdr` structure:

- `source`: SCTP source port.
- `dest`: SCTP destination port.
- `vtag`: Verification Tag, which is a 32 bit random value.
- `checksum`: Checksum of SCTP common header and all chunks.

## SCTP Chunk Header

The SCTP chunk header is represented by `struct sctp_chunkhdr`:

```
typedef struct sctp_chunkhdr {
        __u8 type;
        __u8 flags;
        __be16 length;
} __packed sctp_chunkhdr_t;
```

`(include/linux/sctp.h)`

The following is a description of the members of the `sctp_chunkhdr` structure:

- `type`: The SCTP type. For example, the type of data chunks is SCTP_CID_DATA. See Table 11-2, Chunk types, in the "Quick Reference" section at the end of this chapter, and also see the chunk ID enum definition (`sctp_cid_t`) in `include/linux/sctp.h`.

- `flags`: Usually, all 8 bits in it should be set to 0 by the sender and ignored by the receiver. There are cases when different values are used. For example, in ABORT chunk, we use the T bit (the LSB) thus: it is set to 0 if the sender filled in the Verification Tag, and it is set to 1 if the Verification Tag is reflected.
- `length`: The length of the SCTP chunk.

## SCTP Chunk

The SCTP chunk is represented by `struct sctp_chunk`. Each chunk object contains the source and destination address for this chunk and a subheader (member of the `subh` union) according to its type. For example, for data packets we have the `sctp_datahdr` subheader, and for the INIT type we have the `sctp_inithdr` subtype:

```
struct sctp_chunk {
        . . .
        atomic_t refcnt;

        union {
                __u8 *v;
                struct sctp_datahdr       *data_hdr;
                struct sctp_inithdr       *init_hdr;
                struct sctp_sackhdr       *sack_hdr;
                struct sctp_heartbeathdr   *hb_hdr;
                struct sctp_sender_hb_info *hbs_hdr;
                struct sctp_shutdownhdr    *shutdown_hdr;
                struct sctp_signed_cookie  *cookie_hdr;
                struct sctp_ecnehdr       *ecne_hdr;
                struct sctp_cwrhdr        *ecn_cwr_hdr;
                struct sctp_errhdr        *err_hdr;
                struct sctp_addiphdr      *addip_hdr;
                struct sctp_fwdtsn_hdr    *fwdtsn_hdr;
                struct sctp_authhdr       *auth_hdr;
        } subh;

        struct sctp_chunkhdr    *chunk_hdr;
        struct sctphdr          *sctp_hdr;

        struct sctp_association *asoc;

        /* What endpoint received this chunk? */
        struct sctp_ep_common   *rcvr;

        . . .

        /* What is the origin IP address for this chunk?  */
        union sctp_addr source;
        /* Destination address for this chunk. */
        union sctp_addr dest;

        . . .
        /* 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.

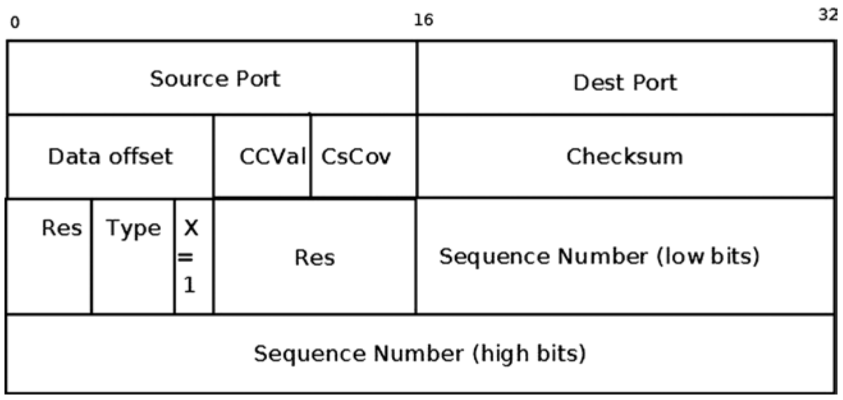

***Figure 11-4.*** *DCCP header (the Extended Sequence Numbers bit is set, dccph_x=1)*

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

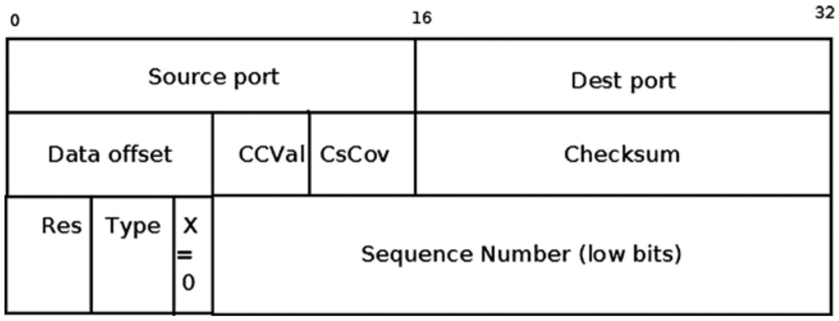

***Figure 11-5.*** *DCCP header (the Extended Sequence Numbers bit is not set, dccph_x=0)*

## 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. |