CHAPTER 3

Internet Control Message Protocol (ICMP)

Chapter 2 discusses the netlink sockets implementation and how netlink sockets are used as a communication channel between the kernel and userspace. This chapter deals with the ICMP protocol, which is a Layer 4 protocol. Userspace applications can use the ICMP protocol (to send and receive ICMP packets) by using the sockets API (the best-known example is probably the `ping` utility). This chapter discusses how these ICMP packets are handled in the kernel and gives some examples.

The ICMP protocol is used primarily as a mandatory mechanism for sending error and control messages about the network layer (L3). The protocol enables getting feedback about problems in the communication environment by sending ICMP messages. These messages provide error handling and diagnostics. The ICMP protocol is relatively simple but is very important for assuring correct system behavior. The basic definition of ICMPv4 is in RFC 792, "Internet Control Message Protocol." This RFC defines the goals of the ICMPv4 protocol and the format of various ICMPv4 messages. I also mention in this chapter RFC 1122 ("Requirements for Internet Hosts—Communication Layers") which defines some requirements about several ICMP messages; RFC 4443, which defines the ICMPv6 protocol; and RFC 1812, which defines requirements for routers. I also describe which types of ICMPv4 and ICMPv6 messages exist, how they are sent, and how they are processed. I cover ICMP sockets, including why they were added and how they are used. Keep in mind that the ICMP protocol is also used for various security attacks; for example, the Smurf Attack is a denial-of-service attack in which large numbers of ICMP packets with the intended victim's spoofed source IP are sent as broadcasts to a computer network using an IP broadcast address.

ICMPv4

ICMPv4 messages can be classified into two categories: error messages and information messages (they are termed "query messages" in RFC 1812). The ICMPv4 protocol is used in diagnostic tools like `ping` and `traceroute`. The famous `ping` utility is in fact a userspace application (from the `iputils` package) which opens a raw socket and sends an ICMP_ECHO message and should get back an ICMP_REPLY message as a response. `Traceroute` is a utility to find the path between a host and a given destination IP address. The `traceroute` utility is based on setting varying values to the Time To Live (TTL), which is a field in the IP header representing the hop count. The `traceroute` utility takes advantage of the fact that a forwarding machine will send back an ICMP_TIME_EXCEED message when the TTL of the packet reaches 0. The `traceroute` utility starts by sending messages with a TTL of 1, and with each received ICMP_DEST_UNREACH with code ICMP_TIME_EXCEED as a reply, it increases the TTL by 1 and sends again to the same destination. It uses the returned ICMP "Time Exceeded" messages to build a list of the routers that the packets traverse, until the destination is reached and returns an ICMP "Echo Reply" message. Traceroute uses the UDP protocol by default. The ICMPv4 module is `net/ipv4/icmp.c`. Note that ICMPv4 cannot be built as a kernel module.

ICMPv4 Initialization

ICMPv4 initialization is done in the `inet_init()` method, which is invoked in boot phase. The `inet_init()` method invokes the `icmp_init()` method, which in turn calls the `icmp_sk_init()` method to create a kernel ICMP socket for sending ICMP messages and to initialize some ICMP `procfs` variables to their default values. (You will encounter some of these `procfs` variables later in this chapter.)

Registration of the ICMPv4 protocol, like registration of other IPv4 protocols, is done in `inet_init()`:

```
static const struct net_protocol icmp_protocol = {
    .handler        =  icmp_rcv,
    .err_handler    =  icmp_err,
    .no_policy      =  1,
    .netns_ok       =  1,
};
```

(`net/ipv4/af_inet.c`)

- `icmp_rcv`: The `handler` callback. This means that for incoming packets whose protocol field in the IP header equals IPPROTO_ICMP (0x1), `icmp_rcv()` will be invoked.
- `no_policy`: This flag is set to 1, which implies that there is no need to perform IPsec policy checks; for example, the `xfrm4_policy_check()` method is not called in `ip_local_deliver_finish()` because the `no_policy` flag is set.
- `netns_ok`: This flag is set to 1, which indicates that the protocol is aware of network namespaces. Network namespaces are described in Appendix A, in the `net_device` section. The `inet_add_protocol()` method will fail for protocols whose `netns_ok` field is 0 with an error of `-EINVAL`.

```
static int __init inet_init(void) {
. . .
    if (inet_add_protocol(&icmp_protocol, IPPROTO_ICMP) < 0)
        pr_crit("%s: Cannot add ICMP protocol\n", __func__);
. . .

int __net_init icmp_sk_init(struct net *net)
{
    . . .
    for_each_possible_cpu(i) {
        struct sock *sk;

        err = inet_ctl_sock_create(&sk, PF_INET,
                       SOCK_RAW, IPPROTO_ICMP, net);
        if (err < 0)
            goto fail;

                net->ipv4.icmp_sk[i] = sk;
             . . .
                sock_set_flag(sk, SOCK_USE_WRITE_QUEUE);
        inet_sk(sk)->pmtudisc = IP_PMTUDISC_DONT;
    }
    . . .

}
```

In the `icmp_sk_init()` method, a raw ICMPv4 socket is created for each CPU and is kept in an array. The current `sk` can be accessed with the `icmp_sk(struct net *net)` method. These sockets are used in the `icmp_push_reply()` method. The ICMPv4 `procfs` entries are initialized in the `icmp_sk_init()` method; I mention them in this chapter and summarize them in the "Quick Reference" section at the end of this chapter. Every ICMP packet starts with an ICMPv4 header. Before discussing how ICMPv4 messages are received and transmitted, the following section describes the ICMPv4 header, so that you better understand how ICMPv4 messages are built.

ICMPv4 Header

The ICMPv4 header consists of type (8 bits), code (8 bits), and checksum (16 bits), and a 32 bits variable part member (its content varies based on the ICMPv4 type and code), as you can see in Figure 3-1. After the ICMPv4 header comes the payload, which should include the IPv4 header of the originating packet and a part of its payload. According to RFC 1812, it should contain as much of the original datagram as possible without the length of the ICMPv4 datagram exceeding 576 bytes. This size is in accordance to RFC 791, which specifies that "All hosts must be prepared to accept datagrams of up to 576 octets."

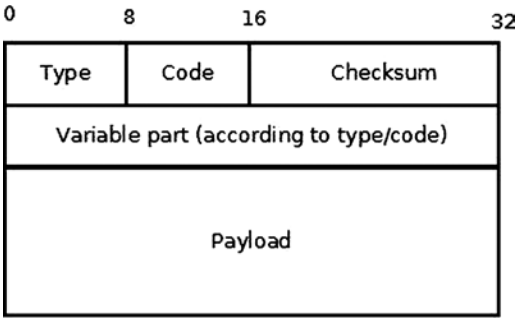

Figure 3-1. *The ICMPv4 header*

The ICMPv4 header is represented by `struct icmphdr`:

```
struct icmphdr {
  __u8        type;
  __u8        code;
  __sum16    checksum;
  union {
    struct {
        __be16    id;
        __be16    sequence;
    } echo;
    __be32    gateway;
    struct {
        __be16    __unused;
        __be16    mtu;
    } frag;
  } un;
};

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

You'll find the current complete list of assigned ICMPv4 message type numbers and codes at www.iana.org/assignments/icmp-parameters/icmp-parameters.xml.

The ICMPv4 module defines an array of `icmp_control` objects, named `icmp_pointers`, which is indexed by ICMPv4 message type. Let's take a look at the `icmp_control` structure definition and at the `icmp_pointers` array:

```
struct icmp_control {
    void (*handler)(struct sk_buff *skb);
    short error;        /* This ICMP is classed as an error message */
};

static const struct icmp_control icmp_pointers[NR_ICMP_TYPES+1];
```

NR_ICMP_TYPES is the highest ICMPv4 type, which is 18.

(include/uapi/linux/icmp.h)

The error field of the `icmp_control` objects of this array is 1 only for error message types, like the "Destination Unreachable" message (ICMP_DEST_UNREACH), and it is 0 (implicitly) for information messages, like echo (ICMP_ECHO). Some handlers are assigned to more than one type. Next I discuss handlers and the ICMPv4 message types they manage.

`ping_rcv()` handles receiving a ping reply (ICMP_ECHOREPLY). The `ping_rcv()` method is implemented in the ICMP sockets code, `net/ipv4/ping.c`. In kernels prior to 3.0, in order to send ping, you had to create a raw socket in userspace. When receiving a reply to a ping (ICMP_ECHOREPLY message), the raw socket that sent the ping processed it. In order to understand how this is implemented, let's take a look in `ip_local_deliver_finish()`, which is the method which handles incoming IPv4 packets and passes them to the sockets which should process them:

```
static int ip_local_deliver_finish(struct sk_buff *skb)
{
    . . .
        int protocol = ip_hdr(skb)->protocol;
        const struct net_protocol *ipprot;
        int raw;

    resubmit:
        raw = raw_local_deliver(skb, protocol);
        ipprot = rcu_dereference(inet_protos[protocol]);
            if (ipprot != NULL) {
                    int ret;
                    . . .
                    ret = ipprot->handler(skb);
                    . . .
```

(net/ipv4/ip_input.c)

When the `ip_local_deliver_finish()` method receives an ICMP_ECHOREPLY packet, it first tries to deliver it to a listening raw socket, which will process it. Because a raw socket that was opened in userspace handles the ICMP_ECHOREPLY message, there is no need to do anything further with it. So when the `ip_local_deliver_finish()` method receives ICMP_ECHOREPLY, the `raw_local_deliver()` method is invoked first to process it by a raw socket, and afterwards the `ipprot->handler(skb)` is invoked (this is the `icmp_rcv()` callback in the case of ICMPv4 packet). And because the packet was already processed by a raw socket, there is nothing more to do with it. So the packet is discarded silently by calling the `icmp_discard()` method, which is the handler for ICMP_ECHOREPLY messages.

When the ICMP sockets ("ping sockets") were integrated into the Linux kernel in kernel 3.0, this was changed. Ping sockets are discussed in the "ICMP Sockets ("Ping Sockets")" section later in this chapter. In this context I should

note that with ICMP sockets, the sender of ping can be also *not a raw socket*. For example, you can create a socket like this: socket (PF_INET, SOCK_DGRAM, PROT_ICMP) and use it to send ping packets. This socket is not a raw socket. As a result, the echo reply is not delivered to any raw socket, since there is no corresponding raw socket which listens. To avoid this problem, the ICMPv4 module handles receiving ICMP_ECHOREPLY messages with the ping_rcv() callback. The ping module is located in the IPv4 layer (net/ipv4/ping.c). Nevertheless, most of the code in net/ipv4/ping.c is a dual-stack code (intended for both IPv4 and IPv6). As a result, the ping_rcv() method also handles ICMPV6_ECHO_REPLY messages for IPv6 (see icmpv6_rcv() in net/ipv6/icmp.c). I talk more about ICMP sockets later in this chapter.

icmp_discard() is an empty handler used for nonexistent message types (message types whose numbers are without corresponding declarations in the header file) and for some messages that do not need any handling, for example ICMP_TIMESTAMPREPLY. The ICMP_TIMESTAMP and the ICMP_TIMESTAMPREPLY messages are used for time synchronization; the sender sends the originate timestamp in an ICMP_TIMESTAMP request; the receiver sends ICMP_TIMESTAMPREPLY with three timestamps: the originating timestamp which was sent by the sender of the timestamp request, as well as a receive timestamp and a transmit timestamp. There are more commonly used protocols for time synchronization than ICMPv4 timestamp messages, like the Network Time Protocol (NTP). I should also mention the Address Mask request (ICMP_ADDRESS), which is normally sent by a host to a router in order to obtain an appropriate subnet mask. Recipients should reply to this message with an address mask reply message. The ICMP_ADDRESS and the ICMP_ADDRESSREPLY messages, which were handled in the past by the icmp_address() method and by the icmp_address_reply() method, are now handled also by icmp_discard(). The reason is that there are other ways to get the subnet masks, such as with DHCP.

icmp_unreach() handles ICMP_DEST_UNREACH, ICMP_TIME_EXCEED, ICMP_PARAMETERPROB, and ICMP_QUENCH message types.

An ICMP_DEST_UNREACH message can be sent under various conditions. Some of these conditions are described in the "Sending ICMPv4 Messages: Destination Unreachable" section in this chapter.

An ICMP_TIME_EXCEEDED message is sent in two cases:

In ip_forward(), each packet decrements its TTL. According to RFC 1700, the recommended TTL for the IPv4 protocol is 64. If the TTL reaches 0, this is indication that the packet should be dropped because probably there was some loop. So, if the TTL reaches 0 in ip_forward(), the icmp_send() method is invoked:

```
icmp_send(skb, ICMP_TIME_EXCEEDED, ICMP_EXC_TTL, 0);
```

(net/ipv4/ip_forward.c)

In such a case, an ICMP_TIME_EXCEEDED message with code ICMP_EXC_TTL is sent, the SKB is freed, the InHdrErrors SNMP counter (IPSTATS_MIB_INHDRERRORS) is incremented, and the method returns NET_RX_DROP.

In ip_expire(), the following occurs when a timeout of a fragment exists:

```
icmp_send(head, ICMP_TIME_EXCEEDED, ICMP_EXC_FRAGTIME, 0);
```

(net/ipv4/ip_fragment.c)

An ICMP_PARAMETERPROB message is sent when parsing the options of an IPv4 header fails, in the ip_options_compile() method or in the ip_options_rcv_srr() method (net/ipv4/ip_options.c). The options are an optional, variable length field (up to 40 bytes) of the IPv4 header. IP options are discussed in Chapter 4.

An ICMP_QUENCH message type is in fact deprecated. According to RFC 1812, section 4.3.3.3 (Source Quench): "A router SHOULD NOT originate ICMP Source Quench messages", and also, "A router MAY ignore any ICMP Source Quench messages it receives." The ICMP_QUENCH message was intended to reduce congestion, but it turned out that this is an ineffective solution.

icmp_redirect() handles ICMP_REDIRECT messages; according to RFC 1122, section 3.2.2.2, hosts should not send an ICMP redirect message; redirects are to be sent only by gateways. icmp_redirect() handles ICMP_REDIRECT messages. In the past, icmp_redirect() called ip_rt_redirect(), but an ip_rt_redirect()

invocation is not needed anymore as the protocol handlers now all properly propagate the redirect back into the routing code. In fact, in kernel 3.6, the `ip_rt_redirect()` method was removed. So the `icmp_redirect()` method first performs sanity checks and then calls `icmp_socket_deliver()`, which delivers the packet to the raw sockets and invokes the protocol error handler (in case it exists). Chapter 6 discusses ICMP_REDIRECT messages in more depth.

`icmp_echo()` handles echo ("ping") requests (ICMP_ECHO) by sending echo replies (ICMP_ECHOREPLY) with `icmp_reply()`. If case `net->ipv4.sysctl_icmp_echo_ignore_all` is set, a reply will not be sent. For configuring ICMPv4 `procfs` entries, see the "Quick Reference" section at the end of this chapter, and also `Documentation/networking/ip-sysctl.txt`.

`icmp_timestamp()` handles ICMP Timestamp requests (ICMP_TIMESTAMP) by sending ICMP_TIMESTAMPREPLY with `icmp_reply()`.

Before discussing sending ICMP messages by the `icmp_reply()` method and by the `icmp_send()` method, I should describe the `icmp_bxm` ("ICMP build xmit message") structure, which is used in both methods:

```
struct icmp_bxm {
    struct sk_buff *skb;
    int offset;
    int data_len;

    struct {
        struct icmphdr icmph;
        __be32         times[3];
    } data;
    int head_len;
    struct ip_options_data replyopts;
};
```

- `skb`: For the `icmp_reply()` method, this `skb` is the request packet; the `icmp_param` object (instance of `icmp_bxm`) is built from it (in the `icmp_echo()` method and in the `icmp_timestamp()` method). For the `icmp_send()` method, this `skb` is the one that triggered sending an ICMPv4 message due to some conditions; you will see several examples of such messages in this section.
- `offset`: Difference (offset) between `skb_network_header(skb)` and `skb->data`.
- `data_len`: ICMPv4 packet payload size.
- `icmph`: The ICMP v4 header.
- `times[3]`: Array of three timestamps, filled in `icmp_timestamp()`.
- `head_len`: Size of the ICMPv4 header (in case of `icmp_timestamp()`, there are additional 12 bytes for the timestamps).
- `replyopts`: An `ip_options` data object. IP options are optional fields after the IP header, up to 40 bytes. They enable advanced features like strict routing/loose routing, record routing, time stamping, and more. They are initialized with the `ip_options_echo()` method. Chapter 4 discusses IP options.

Receiving ICMPv4 Messages

The `ip_local_deliver_finish()` method handles packets for the local machine. When getting an ICMP packet, the method delivers the packet to the raw sockets that had performed registration of ICMPv4 protocol. In the `icmp_rcv()` method, first the `InMsgs` SNMP counter (ICMP_MIB_INMSGS) is incremented. Subsequently, the

checksum correctness is verified. If the checksum is not correct, two SNMP counters are incremented, `InCsumErrors` and `InErrors` (ICMP_MIB_CSUMERRORS and ICMP_MIB_INERRORS, respectively), the SKB is freed, and the method returns 0. The `icmp_rcv()` method does not return an error in this case. In fact, the `icmp_rcv()` method always returns 0; the reason for returning 0 in case of checksum error is that no special thing should be done when receiving an erroneous ICMP message except to discard it; when a protocol handler returns a negative error, another attempt to process the packet is performed, and it is not needed in this case. For more details, refer to the implementation of the `ip_local_deliver_finish()` method. Then the ICMP header is examined in order to find its type; the corresponding `procfs` message type counter is incremented (each ICMP message type has a `procfs` counter), and a sanity check is performed to verify that it is not higher than the highest permitted value (NR_ICMP_TYPES). According to section 3.2.2 of RFC 1122, if an ICMP message of unknown type is received, it must be silently discarded. So if the message type is out of range, the `InErrors` SNMP counter (ICMP_MIB_INERRORS) is incremented, and the SKB is freed.

In case the packet is a broadcast or a multicast, and it is an ICMP_ECHO message or an ICMP_TIMESTAMP message, there is a check whether broadcast/multicast echo requests are permitted by reading the variable `net->ipv4.sysctl_icmp_echo_ignore_broadcasts`. This variable can be configured via `procfs` by writing to `/proc/sys/net/ipv4/icmp_echo_ignore_broadcasts`, and by default its value is 1. If this variable is set, the packet is dropped silently. This is done according to section 3.2.2.6 of RFC 1122: "An ICMP Echo Request destined to an IP broadcast or IP multicast address MAY be silently discarded." And according to section 3.2.2.8 of this RFC, "An ICMP Timestamp Request message to an IP broadcast or IP multicast address MAY be silently discarded." Then a check is performed to detect whether the type is allowed for broadcast/multicast (ICMP_ECHO, ICMP_TIMESTAMP, ICMP_ADDRESS, and ICMP_ADDRESSREPLY). If it is not one of these message types, the packet is dropped and 0 is returned. Then according to its type, the corresponding entry in the `icmp_pointers` array is fetched and the appropriate handler is called. Let's take a look in the ICMP_ECHO entry in the `icmp_control` dispatch table:

```
static const struct icmp_control icmp_pointers[NR_ICMP_TYPES + 1] = {
...
  [ICMP_ECHO] = {
        .handler = icmp_echo,
    },
...
}
```

So when receiving a ping (the type of the message is "Echo Request," ICMP_ECHO), it is handled by the `icmp_echo()` method. The `icmp_echo()` method changes the type in the ICMP header to be ICMP_ECHOREPLY and sends a reply by calling the `icmp_reply()` method. Apart from `ping`, the only other ICMP message which requires a response is the timestamp message (ICMP_TIMESTAMP); it is handled by the `icmp_timestamp()` method, which, much like in the ICMP_ECHO case, changes the type to ICMP_TIMESTAMPREPLY and sends a reply by calling the `icmp_reply()` method. Sending is done by `ip_append_data()` and by `ip_push_pending_frames()`. Receiving a ping reply (ICMP_ECHOREPLY) is handled by the `ping_rcv()` method.

You can disable replying to pings with the following:

```
echo 1 >  /proc/sys/net/ipv4/icmp_echo_ignore_all
```

There are some callbacks that handle more than one ICMP type. The `icmp_discard()` callback, for example, handles ICMPv4 packets whose type is not handled by the Linux ICMPv4 implementation, and messages like ICMP_TIMESTAMPREPLY, ICMP_INFO_REQUEST , ICMP_ADDRESSREPLY, and more.

Sending ICMPv4 Messages: "Destination Unreachable"

There are two methods for sending an ICMPv4 message: the first is the `icmp_reply()` method, which is sent as a response for two types of ICMP requests, ICMP_ECHO and ICMP_TIMESTAMP. The second one is the `icmp_send()` method, where the local machine initiates sending an ICMPv4 message under certain conditions (described in this section). Both these methods eventually invoke `icmp_push_reply()` for actually sending the packet. The `icmp_reply()` method is called as a response to an ICMP_ECHO message from the `icmp_echo()` method, and as a response to an ICMP_TIMESTAMP message from the `icmp_timestamp()` method. The `icmp_send()` method is invoked from many places in the IPv4 network stack—for example, from netfilter, from the forwarding code (`ip_forward.c`), from tunnels like `ipip` and `ip_gre`, and more.

This section looks into some of the cases when a "Destination Unreachable" message is sent (the type is ICMP_DEST_UNREACH).

Code 2: ICMP_PROT_UNREACH (Protocol Unreachable)

When the protocol of the IP header (which is an 8-bit field) is a nonexistent protocol, an ICMP_DEST_UNREACH/ICMP_PROT_UNREACH is sent back to the sender because there is no protocol handler for such a protocol (the protocol handler array is indexed by the protocol number, so for nonexistent protocols there will be no handler). By *nonexistent* protocol I mean either that because of some error indeed the protocol number of the IPv4 header does not appear in the protocol number list (which you can find in `include/uapi/linux/in.h`, for IPv4), or that the kernel was built without support for that protocol, and, as a result, this protocol is not registered and there is no entry for it in the protocol handlers array. Because such a packet can't be handled, an ICMPv4 message of "Destination Unreachable" should be replied back to the sender; the ICMP_PROT_UNREACH code in the ICMPv4 reply signifies the cause of the error, "protocol is unreachable." See the following:

```
static int ip_local_deliver_finish(struct sk_buff *skb)
  {
    ...
    int protocol = ip_hdr(skb)->protocol;
    const struct net_protocol *ipprot;
    int raw;

resubmit:
    raw = raw_local_deliver(skb, protocol);

    ipprot = rcu_dereference(inet_protos[protocol]);
    if (ipprot != NULL) {
        ...
    } else {
    if (!raw) {
    if (xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
            IP_INC_STATS_BH(net, IPSTATS_MIB_INUNKNOWNPROTOS);
            icmp_send(skb, ICMP_DEST_UNREACH,ICMP_PROT_UNREACH, 0);
              }
        ...
  }
```

(`net/ipv4/ip_input.c`)

In this example, a lookup in the `inet_protos` array by protocol is performed; and because no entry was found, this means that the protocol is not registered in the kernel.

Code 3: ICMP_PORT_UNREACH ("Port Unreachable")

When receiving UDPv4 packets, a matching UDP socket is searched for. If no matching socket is found, the checksum correctness is verified. If it is wrong, the packet is dropped silently. If it is correct, the statistics are updated and a "Destination Unreachable"/"Port Unreachable" ICMP message is sent back:

```
int __udp4_lib_rcv(struct sk_buff *skb, struct udp_table *udptable, int proto)
{
        struct sock *sk;
        ...
        sk = __udp4_lib_lookup_skb(skb, uh->source, uh->dest, udptable)
        ...
        if (sk != NULL) {
        ...
        }

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

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

}
```

(net/ipv4/udp.c)

A lookup is being performed by the `__udp4_lib_lookup_skb()` method, and if there is no socket, the statistics are updated and an ICMP_DEST_UNREACH message with ICMP_PORT_UNREACH code is sent back.

Code 4: ICMP_FRAG_NEEDED

When forwarding a packet with a length larger than the MTU of the outgoing link, if the don't fragment (DF) bit in the IPv4 header (IP_DF) is set, the packet is discarded and an ICMP_DEST_UNREACH message with ICMP_FRAG_NEEDED code is sent back to the sender:

```
int ip_forward(struct sk_buff *skb)
{
        ...
        struct rtable *rt;      /* Route we use */
        ...
        if (unlikely(skb->len > dst_mtu(&rt->dst) && !skb_is_gso(skb) &&
                     (ip_hdr(skb)->frag_off & htons(IP_DF))) && !skb->local_df) {
                IP_INC_STATS(dev_net(rt->dst.dev), IPSTATS_MIB_FRAGFAILS);
```

```
                icmp_send(skb, ICMP_DEST_UNREACH, ICMP_FRAG_NEEDED,
                          htonl(dst_mtu(&rt->dst)));
                goto drop;
        }
        ...
}
```

(net/ipv4/ip_forward.c)

Code 5: ICMP_SR_FAILED

When forwarding a packet with the strict routing option and gatewaying set, a "Destination Unreachable" message with ICMP_SR_FAILED code is sent back, and the packet is dropped:

```
int ip_forward(struct sk_buff *skb)
 {
         struct ip_options *opt  = &(IPCB(skb)->opt);
         ...
         if (opt->is_strictroute && rt->rt_uses_gateway)
                 goto sr_failed;
         ...
sr_failed:
          icmp_send(skb, ICMP_DEST_UNREACH, ICMP_SR_FAILED, 0);
          goto drop;
}
```

(net/ipv4/ip_forward.c)

For a full list of all IPv4 "Destination Unreachable" codes, see Table 3-1 in the "Quick Reference" section at the end of this chapter. Note that a user can configure some rules with the `iptables` REJECT target and the `--reject-with` qualifier, which can send "Destination Unreachable" messages according to the selection; more in the "Quick Reference" section at the end of this chapter.

Both the `icmp_reply()` and the `icmp_send()` methods support rate limiting; they call `icmpv4_xrlim_allow()`, and if the rate limiting check allows sending the packet (the `icmpv4_xrlim_allow()` returns `true`), they send the packet. It should be mentioned here that rate limiting is not performed automatically on all types of traffic. Here are the conditions under which rate limiting check will not be performed:

- The message type is unknown.
- The packet is of PMTU discovery.
- The device is a loopback device.
- The ICMP type is not enabled in the rate mask.

If all these conditions are not matched, rate limiting is performed by calling the `inet_peer_xrlim_allow()` method. You'll find more info about rate mask in the "Quick Reference" section at the end of this chapter.

Let's look inside the `icmp_send()` method. First, this is its prototype:

```
void icmp_send(struct sk_buff *skb_in, int type, int code, __be32 info)
```

skb_in is the SKB which caused the invocation of the icmp_send() method, type and code are the ICMPv4 message type and code, respectively. The last parameter, info, is used in the following cases:

- For the ICMP_PARAMETERPROB message type it is the offset in the IPv4 header where the parsing problem occurred.
- For the ICMP_DEST_UNREACH message type with ICMP_FRAG_NEEDED code, it is the MTU.
- For the ICMP_REDIRECT message type with ICMP_REDIR_HOST code, it is the IP address of the destination address in the IPv4 header of the provoking SKB.

When further looking into the icmp_send() method, first there are some sanity checks. Then multicast/broadcast packets are rejected. A check of whether the packet is a fragment is performed by inspecting the frag_off field of the IPv4 header. If the packet is fragmented, an ICMPv4 message is sent, but only for the first fragment. According to section 4.3.2.7 of RFC 1812, an ICMP error message must not be sent as the result of receiving an ICMP error message. So first a check is performed to find out whether the ICMPv4 message to be sent is an error message, and if it is so, another check is performed to find out whether the provoking SKB contained an error ICMPv4 message, and if so, then the method returns without sending the ICMPv4 message. Also if the type is an unknown ICMPv4 type (higher than NR_ICMP_TYPES), the method returns without sending the ICMPv4 message, though this isn't specified explicitly by the RFC. Then the source address is determined according to the value of net->ipv4.sysctl_icmp_errors_use_inbound_ifaddr value (more details in the "Quick Reference" section at the end of this chapter). Then the ip_options_echo() method is invoked to copy the IP options of the IPv4 header of the invoking SKB. An icmp_bxm object (icmp_param) is being allocated and initialized, and a lookup in the routing subsystem is performed with the icmp_route_lookup() method. Then the icmp_push_reply() method is invoked.

Let's take a look at the icmp_push_reply() method, which actually sends the packet. The icmp_push_reply() first finds the socket on which the packet should be sent by calling:

```
sk = icmp_sk(dev_net((*rt)->dst.dev));
```

The dev_net() method returns the network namespace of the outgoing network device. (The dev_net() method and network namespaces are discussed in chapter 14 and in Appendix A.) Then, the icmp_sk() method fetches the socket (because in SMP there is a socket per CPU). Then the ip_append_data() method is called to move the packet to the IP layer. If the ip_append_data() method fails, the statistics are updated by incrementing the ICMP_MIB_OUTERRORS counter and the ip_flush_pending_frames() method is called to free the SKB. I discuss the ip_append_data() method and the ip_flush_pending_frames() method in Chapter 4.

Now that you know all about ICMPv4, it's time to move on to ICMPv6.

ICMPv6

ICMPv6 has many similarities to ICMPv4 when it comes to reporting errors in the network layer (L3). There are additional tasks for ICMPv6 which are not performed in ICMPv4. This section discusses the ICMPv6 protocol, its new features (which are not implemented in ICMPv4), and the features which are similar. ICMPv6 is defined in RFC 4443. If you delve into ICMPv6 code you will probably encounter, sooner or later, comments that mention RFC 1885. In fact, RFC 1885, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6)," is the base ICMPv6 RFC. It was obsoleted by RFC 2463, which was in turn obsoleted by RFC 4443. The ICMPv6 implementation is based upon IPv4, but it is more complicated; the changes and additions that were added are discussed in this section.

The ICMPv6 protocol has a next header value of 58, according to RFC 4443, section 1 (Chapter 8 discusses IPv6 next headers). ICMPv6 is an integral part of IPv6 and must be fully implemented by every IPv6 node. Apart from error handling and diagnostics, ICMPv6 is used for the Neighbour Discovery (ND) protocol in IPv6, which replaces and enhances functions of ARP in IPv4, and for the Multicast Listener Discovery (MLD) protocol, which is the counterpart of the IGMP protocol in IPv4, shown in Figure 3-2.

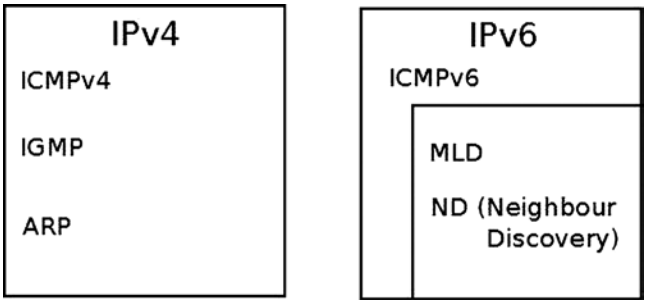

Figure 3-2. *ICMP in IPv4 and IPv6. The counterpart of the IGMP protocol in IPv6 is the MLD protocol, and the counterpart of the ARP protocol in IPv6 is the ND protocol*

This section covers the ICMPv6 implementation. As you will see, it has many things in common with the ICMPv4 implementation in the way messages are handled and sent. There are even cases when the same methods are called in ICMPv4 and in ICMPv6 (for example, `ping_rcv()` and `inet_peer_xrlim_allow()`). There are some differences, and some topics are unique to ICMPv6. The `ping6` and `traceroute6` utilities are based on ICMPv6 and are the counterparts of `ping` and `traceroute` utilities of IPv4 (mentioned in the ICMPv4 section in the beginning of this chapter). ICMPv6 is implemented in `net/ipv6/icmp.c` and in `net/ipv6/ip6_icmp.c`. As with ICMPv4, ICMPv6 cannot be built as a kernel module.

ICMPv6 Initialization

ICMPv6 initialization is done by the `icmpv6_init()` method and by the `icmpv6_sk_init()` method. Registration of the ICMPv6 protocol is done by `icmpv6_init()` (`net/ipv6/icmp.c`):

```
static const struct inet6_protocol icmpv6_protocol = {
        .handler        =       icmpv6_rcv,
        .err_handler    =       icmpv6_err,
        .flags          =       INET6_PROTO_NOPOLICY|INET6_PROTO_FINAL,
 };
```

The `handler` callback is `icmpv6_rcv()`; this means that for incoming packets whose protocol field equals IPPROTO_ICMPV6 (58), `icmpv6_rcv()` will be invoked.

When the INET6_PROTO_NOPOLICY flag is set, this implies that IPsec policy checks should not be performed; for example, the `xfrm6_policy_check()` method is not called in `ip6_input_finish()` because the INET6_PROTO_NOPOLICY flag is set:

```
int __init icmpv6_init(void)
 {
        int err;
        ...
        if (inet6_add_protocol(&icmpv6_protocol, IPPROTO_ICMPV6) < 0)
                goto fail;
        return 0;
 }
```

```
static int __net_init icmpv6_sk_init(struct net *net)
{
    struct sock *sk;
        ...
    for_each_possible_cpu(i) {
        err = inet_ctl_sock_create(&sk, PF_INET6,
                        SOCK_RAW, IPPROTO_ICMPV6, net);
        ...
        net->ipv6.icmp_sk[i] = sk;
        ...

}
```

As in ICMPv4, a raw ICMPv6 socket is created for each CPU and is kept in an array. The current `sk` can be accessed by the `icmpv6_sk()` method.

ICMPv6 Header

The ICMPv6 header consists of `type` (8 bits), `code` (8 bits), and `checksum` (16 bits), as you can see in Figure 3-3.

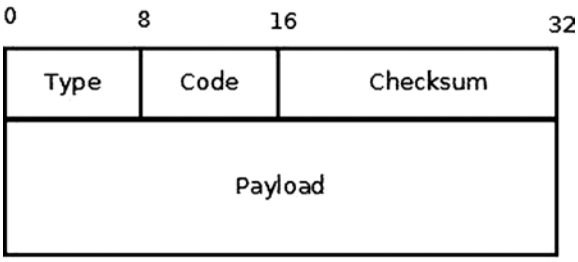

Figure 3-3. *ICMPv6 header*

The ICMPv6 header is represented by `struct icmp6hdr`:

```
struct icmp6hdr {
    __u8        icmp6_type;
    __u8        icmp6_code;
    __sum16     icmp6_cksum;
    ...
}
```

There is not enough room to show all the fields of `struct icmp6hdr` because it is too large (it is defined in `include/uapi/linux/icmpv6.h`). When the high-order bit of the type field is 0 (values in the range from 0 to 127), it indicates an error message; when the high-order bit is 1 (values in the range from 128 to 255), it indicates an information message. Table 3-1 shows the ICMPv6 message types by their number and kernel symbol.

Table 3-1. *ICMPv6 Messages*

Type	Kernel symbol	Error/Info	Description
1	ICMPV6_DEST_UNREACH	Error	Destination Unreachable
2	ICMPV6_PKT_TOOBIG	Error	Packet too big
3	ICMPV6_TIME_EXCEED	Error	Time Exceeded
4	ICMPV6_PARAMPROB	Error	Parameter problem
128	ICMPV6_ECHO_REQUEST	Info	Echo Request
129	ICMPV6_ECHO_REPLY	Info	Echo Reply
130	ICMPV6_MGM_QUERY	Info	Multicast group membership management query
131	ICMPV6_MGM_REPORT	Info	Multicast group membership management report
132	ICMPV6_MGM_REDUCTION	Info	Multicast group membership management reduction
133	NDISC_ROUTER_SOLICITATION	Info	Router solicitation
134	NDISC_ROUTER_ADVERTISEMENT	Info	Router advertisement
135	NDISC_NEIGHBOUR_SOLICITATION	Info	Neighbour solicitation
136	NDISC_NEIGHBOUR_ADVERTISEMENT	Info	Neighbour advertisement
137	NDISC_REDIRECT	Info	Neighbour redirect

The current complete list of assigned ICMPv6 types and codes can be found at www.iana.org/assignments/icmpv6-parameters/icmpv6-parameters.xml.

ICMPv6 performs some tasks that are not performed by ICMPv4. For example, Neighbour Discovery is done by ICMPv6, whereas in IPv4 it is done by the ARP/RARP protocols. Multicast group memberships are handled by ICMPv6 in conjunction with the MLD (Multicast Listener Discovery) protocol, whereas in IPv4 this is performed by IGMP (Internet Group Management Protocol). Some ICMPv6 messages are similar in meaning to ICMPv4 messages; for example, ICMPv6 has these messages: "Destination Unreachable," (ICMPV6_DEST_UNREACH), "Time Exceeded" (ICMPV6_TIME_EXCEED), "Parameter Problem" (ICMPV6_PARAMPROB), "Echo Request" (ICMPV6_ECHO_REQUEST), and more. On the other hand, some ICMPv6 messages are unique to IPv6, such as the NDISC_NEIGHBOUR_SOLICITATION message.

Receiving ICMPv6 Messages

When getting an ICMPv6 packet, it is delivered to the `icmpv6_rcv()` method, which gets only an SKB as a parameter. Figure 3-4 shows the `Rx` path of a received ICMPv6 message.

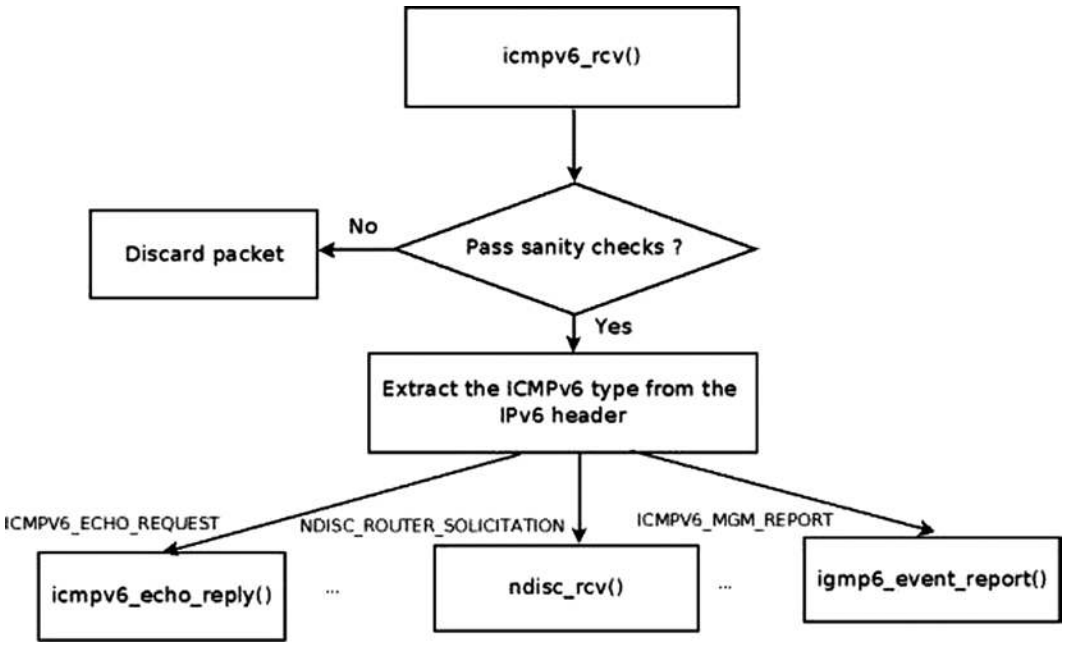

Figure 3-4. *Receive path of ICMPv6 message*

In the `icmpv6_rcv()` method, after some sanity checks, the `InMsgs` SNMP counter (ICMP6_MIB_INMSGS) is incremented. Subsequently, the checksum correctness is verified. If the checksum is not correct, the `InErrors` SNMP counter (ICMP6_MIB_INERRORS) is incremented, and the SKB is freed. The `icmpv6_rcv()` method does not return an error in this case (in fact it always returns 0, much like its IPv4 counterpart, `icmp_rcv()`).Then the ICMPv6 header is read in order to find its type; the corresponding `procfs` message type counter is incremented by the ICMP6MSGIN_INC_STATS_BH macro (each ICMPv6 message type has a `procfs` counter). For example, when receiving ICMPv6 ECHO requests ("pings"), the `/proc/net/snmp6/Icmp6InEchos` counter is incremented, and when receiving ICMPv6 Neighbour Solicitation requests, the `/proc/net/snmp6/Icmp6InNeighborSolicits` counter is incremented.

In ICMPv6, there is no dispatch table like the `icmp_pointers` table in ICMPv4. The handlers are invoked according to the ICMPv6 message type, in a long `switch(type)` command:

- "Echo Request" (ICMPV6_ECHO_REQUEST) is handled by the `icmpv6_echo_reply()` method.
- "Echo Reply" (ICMPV6_ECHO_REPLY) is handled by the `ping_rcv()` method. The `ping_rcv()` method is in the IPv4 `ping` module (`net/ipv4/ping.c`); this method is a dual-stack method (it handles both IPv4 and IPv6—discussed in the beginning of this chapter).
- Packet too big (ICMPV6_PKT_TOOBIG).
  - First a check is done to verify that the data block area (pointed to by `skb->data`) contains a block of data whose size is at least as big as an ICMP header. This is done by the `pskb_may_pull()` method. If this condition is not met, the packet is dropped.
  - Then the `icmpv6_notify()` method is invoked. This method eventually calls the `raw6_icmp_error()` method so that the registered raw sockets will handle the ICMP messages.

- "Destination Unreachable," "Time Exceeded," and "Parameter Problem" (ICMPV6_DEST_UNREACH, ICMPV6_TIME_EXCEED, and ICMPV6_PARAMPROB respectively) are also handled by `icmpv6_notify()`.
- Neighbour Discovery (ND) messages:
  - NDISC_ROUTER_SOLICITATION: Messages which are sent usually to the `all-routers` multicast address of FF02::2, and which are answered by router advertisements. (Special IPv6 multicast addresses are discussed in Chapter 8).
  - NDISC_ROUTER_ADVERTISEMENT: Messages which are sent periodically by routers or as an immediate response to router solicitation requests. Router advertisements contain prefixes that are used for on-link determination and/or address configuration, a suggested hop limit value, and so on.
  - NDISC_NEIGHBOUR_SOLICITATION: The counterpart of ARP request in IPv4.
  - NDISC_NEIGHBOUR_ADVERTISEMENT: The counterpart of ARP reply in IPv4.
  - NDISC_REDIRECT: Used by routers to inform hosts of a better first hop for a destination.
  - All the Neighbour Discovery (ND) messages are handled by the neighbour discovery method, `ndisc_rcv()` (`net/ipv6/ndisc.c`). The `ndisc_rcv()` method is discussed in Chapter 7.
- ICMPV6_MGM_QUERY (Multicast Listener Report) is handled by `igmp6_event_query()`.
- ICMPV6_MGM_REPORT (Multicast Listener Report) is handled by `igmp6_event_report()`. Note: Both ICMPV6_MGM_QUERY and ICMPV6_MGM_REPORT are discussed in more detail in Chapter 8.
- Messages of unknown type, and the following messages, are all handled by the `icmpv6_notify()` method:
  - ICMPV6_MGM_REDUCTION: When a host leaves a multicast group, it sends an MLDv2 ICMPV6_MGM_REDUCTION message; see the `igmp6_leave_group()` method in `net/ipv6/mcast.c`.
  - ICMPV6_MLD2_REPORT: MLDv2 Multicast Listener Report packet; usually sent with destination address of the all MLDv2-capable routers Multicast Group Address (FF02::16).
  - ICMPV6_NI_QUERY- ICMP: Node Information Query.
  - ICMPV6_NI_REPLY: ICMP Node Information Response.
  - ICMPV6_DHAAD_REQUEST: ICMP Home Agent Address Discovery Request Message; see section 6.5, RFC 6275, "Mobility Support in IPv6."
  - ICMPV6_DHAAD_REPLY: ICMP Home Agent Address Discovery Reply Message; See section 6.6, RFC 6275.
  - ICMPV6_MOBILE_PREFIX_SOL: ICMP Mobile Prefix Solicitation Message Format; see section 6.7, RFC 6275.
  - ICMPV6_MOBILE_PREFIX_ADV: ICMP Mobile Prefix Advertisement Message Format; see section 6.8, RFC 6275.

Notice that the `switch(type)` command ends like this:

```
    default:
        LIMIT_NETDEBUG(KERN_DEBUG "icmpv6: msg of unknown type\n");

        /* informational */
        if (type & ICMPV6_INFOMSG_MASK)
            break;

        /*
         * error of unknown type.
         * must pass to upper level
         */

        icmpv6_notify(skb, type, hdr->icmp6_code, hdr->icmp6_mtu);
    }
```

Informational messages fulfill the condition (`type & ICMPV6_INFOMSG_MASK`), so they are discarded, whereas the other messages which do not fulfill this condition (and therefore should be error messages) are passed to the upper layer. This is done in accordance with section 2.4 ("Message Processing Rules") of RFC 4443.

Sending ICMPv6 Messages

The main method for sending ICMPv6 messages is the `icmpv6_send()` method. The method is called when the local machine initiates sending an ICMPv6 message under conditions described in this section. There is also the `icmpv6_echo_reply()` method, which is called only as a response to an ICMPV6_ECHO_REQUEST ("ping") message. The `icmp6_send()` method is invoked from many places in the IPv6 network stack. This section looks at several examples.

Example: Sending "Hop Limit Time Exceeded" ICMPv6 Messages

When forwarding a packet, every machine decrements the Hop Limit Counter by 1. The Hop Limit Counter is a member of the IPv6 header—it is the IPv6 counterpart to Time To Live in IPv4. When the value of the Hop Limit Counter header reaches 0, an ICMPV6_TIME_EXCEED message is sent with ICMPV6_EXC_HOPLIMIT code by calling the `icmpv6_send()` method, then the statistics are updated and the packet is dropped:

```
int ip6_forward(struct sk_buff *skb)
{
    ...
        if (hdr->hop_limit <= 1) {
                /* Force OUTPUT device used as source address */
                skb->dev = dst->dev;
                icmpv6_send(skb, ICMPV6_TIME_EXCEED, ICMPV6_EXC_HOPLIMIT, 0);
                IP6_INC_STATS_BH(net,
                                 ip6_dst_idev(dst), IPSTATS_MIB_INHDRERRORS);

                kfree_skb(skb);
                return -ETIMEDOUT;
         }
    ...
}
```

(net/ipv6/ip6_output.c)

Example: Sending "Fragment Reassembly Time Exceeded" ICMPv6 Messages

When a timeout of a fragment occurs, an ICMPV6_TIME_EXCEED message with ICMPV6_EXC_FRAGTIME code is sent back, by calling the `icmpv6_send()` method:

```
void ip6_expire_frag_queue(struct net *net, struct frag_queue *fq,
                           struct inet_frags *frags)
 {
        ...
        icmpv6_send(fq->q.fragments, ICMPV6_TIME_EXCEED, ICMPV6_EXC_FRAGTIME, 0);
        ...
 }
```

(net/ipv6/reassembly.c)

Example: Sending "Destination Unreachable"/"Port Unreachable" ICMPv6 Messages

When receiving UDPv6 packets, a matching UDPv6 socket is searched for. If no matching socket is found, the checksum correctness is verified. If it is wrong, the packet is dropped silently. If it is correct, the statistics (UDP_MIB_NOPORTS MIB counter, which is exported to `procfs` by `/proc/net/snmp6/Udp6NoPorts`) is updated and a "Destination Unreachable"/"Port Unreachable" ICMPv6 message is sent back with `icmpv6_send()`:

```
int __udp6_lib_rcv(struct sk_buff *skb, struct udp_table *udptable, int proto)
{
        ...
       sk = __udp6_lib_lookup_skb(skb, uh->source, uh->dest, udptable);
        if (sk != NULL) {
        ...
        }
        ...
        if (udp_lib_checksum_complete(skb))
                goto discard;

        UDP6_INC_STATS_BH(net, UDP_MIB_NOPORTS, proto == IPPROTO_UDPLITE);
        icmpv6_send(skb, ICMPV6_DEST_UNREACH, ICMPV6_PORT_UNREACH, 0);
        ...

}
```

This case is very similar to the UDPv4 example given earlier in this chapter.

Example: Sending "Fragmentation Needed" ICMPv6 Messages

When forwarding a packet, if its size is larger than the MTU of the outgoing link, and the `local_df` bit in the SKB is not set, the packet is discarded and an ICMPV6_PKT_TOOBIG message is sent back to the sender. The information in this message is used as part of the Path MTU (PMTU) discovery process.

Note that as opposed to the parallel case in IPv4, where an ICMP_DEST_UNREACH message with ICMP_FRAG_NEEDED code is sent, in this case an ICMPV6_PKT_TOOBIG message is sent back, and not a "Destination Unreachable" (ICMPV6_DEST_UNREACH) message. The ICMPV6_PKT_TOOBIG message has a message type number of its own in ICMPv6:

```
int ip6_forward(struct sk_buff *skb)
{
...
         if ((!skb->local_df && skb->len > mtu && !skb_is_gso(skb)) ||
             (IP6CB(skb)->frag_max_size && IP6CB(skb)->frag_max_size > mtu)) {
                 /* Again, force OUTPUT device used as source address */
                 skb->dev = dst->dev;
                 icmpv6_send(skb, ICMPV6_PKT_TOOBIG, 0, mtu);
                 IP6_INC_STATS_BH(net,
                                  ip6_dst_idev(dst), IPSTATS_MIB_INTOOBIGERRORS);
                 IP6_INC_STATS_BH(net,
                                  ip6_dst_idev(dst), IPSTATS_MIB_FRAGFAILS);
                 kfree_skb(skb);
                 return -EMSGSIZE;
         }
...
}
```

(net/ipv6/ip6_output.c)

Example: Sending "Parameter Problem" ICMPv6 Messages

When encountering a problem in parsing extension headers, an ICMPV6_PARAMPROB message with ICMPV6_UNK_OPTION code is sent back:

```
static bool ip6_tlvopt_unknown(struct sk_buff *skb, int optoff) {
        switch ((skb_network_header(skb)[optoff] & 0xC0) >> 6) {
        ...
        case 2: /* send ICMP PARM PROB regardless and drop packet */
                 icmpv6_param_prob(skb, ICMPV6_UNK_OPTION, optoff);
                 return false;
         }
```

(net/ipv6/exthdrs.c)

The `icmpv6_send()` method supports rate limiting by calling `icmpv6_xrlim_allow()`. I should mention here that, as in ICMPv4, rate limiting is not performed automatically in ICMPv6 on all types of traffic. Here are the conditions under which rate limiting check will not be performed:

- Informational messages
- PMTU discovery
- Loopback device

If all these conditions are not matched, rate limiting is performed by calling the `inet_peer_xrlim_allow()` method, which is shared between ICMPv4 and ICMPv6. Note that unlike IPv4, you can't set a rate mask in IPv6. It is not forbidden by the ICMPv6 spec, RFC 4443, but it was never implemented.

Let's look inside the `icmp6_send()` method. First, this is its prototype:

```
static void icmp6_send(struct sk_buff *skb, u8 type, u8 code, __u32 info)
```

The parameters are similar to those of the `icmp_send()` method of IPv4, so I won't repeat the explanation here. When further looking into the `icmp6_send()` code, you find some sanity checks. Checking whether the provoking message is an ICMPv6 error message is done by calling the `is_ineligible()` method; if it is, the `icmp6_send()` method terminates. The length of the message should not exceed 1280, which is IPv6 minimum MTU (IPV6_MIN_MTU, defined in `include/linux/ipv6.h`). This is done in accordance with RFC 4443, section 2.4 (c), which says that every ICMPv6 error message must include as much of the IPv6 offending (invoking) packet (the packet that caused the error) as possible without making the error message packet exceed the minimum IPv6 MTU. Then the message is passed to the IPv6 layer, by the `ip6_append_data()` method and by the `icmpv6_push_pending_frame()` method, to free the SKB.

Now I'll turn to the `icmpv6_echo_reply()` method; as a reminder, this method is called as a response to an ICMPV6_ECHO message. The `icmpv6_echo_reply()` method gets only one parameter, the SKB. It builds an `icmpv6_msg` object and sets its type to ICMPV6_ECHO_REPLY. Then it passes the message to the IPv6 layer, by the `ip6_append_data()` method and by the `icmpv6_push_pending_frame()` method. If the `ip6_append_data()` method fails, an SNMP counter (ICMP6_MIB_OUTERRORS) is incremented, and `ip6_flush_pending_frames()` is invoked to free the SKB.

Chapters 7 and 8 also discuss ICMPv6. The next section introduces ICMP sockets and the purpose they serve.

ICMP Sockets ("Ping sockets")

A new type of sockets (IPPROTO_ICMP) was added by a patch from the Openwall GNU/*/Linux distribution (Owl), which provides security enhancements over other distributions. The ICMP sockets enable a `setuid-less` "ping." For Openwall GNU/*/Linux, it was the last step on the road to a `setuid-less` distribution. With this patch, a new ICMPv4 ping socket (which is not a raw socket) is created with:

```
socket(PF_INET, SOCK_DGRAM, IPPROTO_ICMP);
```

instead of with:

```
socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);
```

There is also support for IPPROTO_ICMPV6 sockets, which was added later, in `net/ipv6/icmp.c`. A new ICMPv6 ping socket is created with:

```
socket(PF_INET6, SOCK_DGRAM, IPPROTO_ICMPV6);
```

instead of with:

```
socket(PF_INET6, SOCK_RAW, IPPROTO_ICMP6);
```

Similar functionality (non-privileged ICMP) is implemented in Mac OS X; see: www.manpagez.com/man/4/icmp/.

Most of the code for ICMP sockets is in `net/ipv4/ping.c`; in fact, large parts of the code in `net/ipv4/ping.c` are dual-stack (IPv4 and IPv6). In `net/ipv6/ping.c` there are only few IPv6-specific bits. Using ICMP sockets is disabled by default. You can enable ICMP sockets by setting the following `procfs` entry: `/proc/sys/net/ipv4/ping_group_range`. It is "1 0" by default, meaning that nobody (not even root) may create ping sockets. So, if you want to allow a user with `uid` and `gid` of 1000 to use the ICMP socket, you should run this from the command line (with root privileges): `echo 1000 1000 > /proc/sys/net/ipv4/ping_group_range`, and then you can `ping` from this user

account using ICMP sockets. If you want to set privileges for a user in the system, you should run from the command line `echo 0  2147483647 > /proc/sys/net/ipv4/ping_group_range`. (2147483647 is the value of GID_T_MAX; see `include/net/ping.h`.) There are no separate security settings for IPv4 and IPv6; everything is controlled by `/proc/sys/net/ipv4/ping_group_range`. The ICMP sockets support only ICMP_ECHO for IPv4 or ICMPV6_ECHO_REQUEST for IPv6, and the code of the ICMP message must be 0 in both cases.

The `ping_supported()` helper method checks whether the parameters for building the ICMP message (both for IPv4 and IPv6) are valid. It is invoked from `ping_sendmsg()`:

```
static inline int ping_supported(int family, int type, int code)
{
    return (family == AF_INET && type == ICMP_ECHO && code == 0) ||
           (family == AF_INET6 && type == ICMPV6_ECHO_REQUEST && code == 0);
}
```

(`net/ipv4/ping.c`)

ICMP sockets export the following entries to procfs: `/proc/net/icmp` for IPv4 and `/proc/net/icmp6` for IPv6.

For more info about ICMP sockets see http://openwall.info/wiki/people/segoon/ping and http://lwn.net/Articles/420799/.

Summary

This chapter covered the implementation of ICMPv4 and ICMPv6. You learned about the ICMP header format of both protocols and about receiving and sending messages with both protocols. The new features of ICMPv6, which you will encounter in upcoming chapters, were also discussed. The Neighbouring Discovery protocol, which uses ICMPv6 messages, is discussed in Chapter 7, and the MLD protocol, which also uses ICMPv6 messages, is covered in Chapter 8. The next chapter, Chapter 4, talks about the implementation of the IPv4 network layer.

In the "Quick Reference" section that follows, I cover the top methods related to the topics discussed in this chapter, ordered by their context. Then two tables mentioned in the chapter, some important relevant `procfs` entries and a short section about ICMP messages usage in `iptables reject` rules are all covered.

Quick Reference

I conclude this chapter with a short list of important methods of ICMPv4 and ICMPv6, 6 tables, a section about `procfs` entries, and a short section about using a reject target in `iptables` and `ip6tables` to create ICMP "Destination Unreachable" messages.

Methods

The following methods were covered in this chapter.

int icmp_rcv(struct sk_buff *skb);

This method is the main handler for processing incoming ICMPv4 packets.

extern void icmp_send(struct sk_buff *skb_in, int type, int code, __be32 info);

This method sends an ICMPv4 message. The parameters are the provoking SKB, ICMPv4 message type, ICMPv4 message code, and `info` (which is dependent on type).

struct icmp6hdr *icmp6_hdr(const struct sk_buff *skb);

This method returns the ICMPv6 header, which the specified `skb` contains.

void icmpv6_send(struct sk_buff *skb, u8 type, u8 code, __u32 info);

This method sends an ICMPv6 message. The parameters are the provoking SKB, ICMPv6 message type, ICMPv6 message code, and `info` (which is dependent on type).

void icmpv6_param_prob(struct sk_buff *skb, u8 code, int pos);

This method is a convenient version of the `icmp6_send()` method, which all it does is call `icmp6_send()` with ICMPV6_PARAMPROB as a type, and with the other specified parameters, `skb`, `code` and `pos`, and frees the SKB afterwards.

Tables

The following tables were covered in this chapter.

Table 3-2. *ICMPv4 "Destination Unreachable" (ICMP_DEST_UNREACH) Codes*

Code	Kernel Symbol	Description
0	ICMP_NET_UNREACH	Network Unreachable
1	ICMP_HOST_UNREACH	Host Unreachable
2	ICMP_PROT_UNREACH	Protocol Unreachable
3	ICMP_PORT_UNREACH	Port Unreachable
4	ICMP_FRAG_NEEDED	Fragmentation Needed, but the DF flag is set.
5	ICMP_SR_FAILED	Source route failed
6	ICMP_NET_UNKNOWN	Destination network unknown
7	ICMP_HOST_UNKNOWN	Destination host unknown
8	ICMP_HOST_ISOLATED	Source host isolated
9	ICMP_NET_ANO	The destination network is administratively prohibited.
10	ICMP_HOST_ANO	The destination host is administratively prohibited.
11	ICMP_NET_UNR_TOS	The network is unreachable for Type Of Service.
12	ICMP_HOST_UNR_TOS	The host is unreachable for Type Of Service.
13	ICMP_PKT_FILTERED	Packet filtered
14	ICMP_PREC_VIOLATION	Precedence violation
15	ICMP_PREC_CUTOFF	Precedence cut off
16	NR_ICMP_UNREACH	Number of unreachable codes

Table 3-3. *ICMPv4 Redirect (ICMP_REDIRECT) Codes*

Code	Kernel Symbol	Description
0	ICMP_REDIR_NET	Redirect Net
1	ICMP_REDIR_HOST	Redirect Host
2	ICMP_REDIR_NETTOS	Redirect Net for TOS
3	ICMP_REDIR_HOSTTOS	Redirect Host for TOS

Table 3-4. *ICMPv4 Time Exceeded (ICMP_TIME_EXCEEDED) Codes*

Code	Kernel Symbol	Description
0	ICMP_EXC_TTL	TTL count exceeded
1	ICMP_EXC_FRAGTIME	Fragment Reassembly time exceeded

Table 3-5. *ICMPv6 "Destination Unreachable" (ICMPV6_DEST_UNREACH) Codes*

Code	Kernel Symbol	Description
0	ICMPV6_NOROUTE	No route to destination
1	ICMPV6_ADM_PROHIBITED	Communication with destination administratively prohibited
2	ICMPV6_NOT_NEIGHBOUR	Beyond scope of source address
3	ICMPV6_ADDR_UNREACH	Address Unreachable
4	ICMPV6_PORT_UNREACH	Port Unreachable

Note that ICMPV6_PKT_TOOBIG, which is the counterpart of IPv4 ICMP_DEST_UNREACH /ICMP_FRAG_NEEDED, is not a code of ICMPV6_DEST_UNREACH, but an ICMPv6 type in itself.

Table 3-6. *ICMPv6 Time Exceeded (ICMPV6_TIME_EXCEED) Codes*

Code	Kernel Symbol	Description
0	ICMPV6_EXC_HOPLIMIT	Hop limit exceeded in transit
1	ICMPV6_EXC_FRAGTIME	Fragment reassembly time exceeded

Table 3-7. *ICMPv6 Parameter Problem (ICMPV6_PARAMPROB) Codes*

Code	Kernel Symbol	Description
0	ICMPV6_HDR_FIELD	Erroneous header field encountered
1	ICMPV6_UNK_NEXTHDR	Unknown Next Header type encountered
2	ICMPV6_UNK_OPTION	Unknown IPv6 option encountered

procfs entries

The kernel provides a way of configuring various settings for various subsystems from the userspace by way of writing values to entries under `/proc`. These entries are referred to as `procfs` entries. All of the ICMPv4 `procfs` entries are represented by variables in the `netns_ipv4` structure (`include/net/netns/ipv4.h`), which is an object in the network namespace (`struct net`). Network namespaces and their implementation are discussed in Chapter 14. The following are the names of the `sysctl` variables that correspond to the ICMPv4 `netns_ipv4` elements, explanations about their usage, and the default values to which they are initialized, specifying also in which method the initialization takes place.

sysctl_icmp_echo_ignore_all

When `icmp_echo_ignore_all` is set, echo requests (ICMP_ECHO) will not be replied.

`procfs` entry: `/proc/sys/net/ipv4/icmp_echo_ignore_all`
Initialized to 0 in `icmp_sk_init()`

sysctl_icmp_echo_ignore_broadcasts

When receiving a broadcast or a multicast echo (ICMP_ECHO) message or a timestamp (ICMP_TIMESTAMP) message, you check whether broadcast/multicast requests are permitted by reading `sysctl_icmp_echo_ignore_broadcasts`. If this variable is set, you drop the packet and return 0.

`procfs` entry: `/proc/sys/net/ipv4/icmp_echo_ignore_broadcasts`
Initialized to 1 in `icmp_sk_init()`

sysctl_icmp_ignore_bogus_error_responses

Some routers violate RFC1122 by sending bogus responses to broadcast frames. In the `icmp_unreach()` method, you check this flag. If this flag is set to TRUE, the kernel will not log these warnings ("<IPv4Addr>sent an invalid ICMP type. . .").

`procfs` entry: `/proc/sys/net/ipv4/icmp_ignore_bogus_error_responses`
Initialized to 1 in `icmp_sk_init()`

sysctl_icmp_ratelimit

Limit the maximal rates for sending ICMP packets whose type matches the icmp ratemask (`icmp_ratemask`, see later in this section) to specific targets.

A value of 0 means disable any limiting; otherwise it is the minimal space between responses in milliseconds.
`procfs` entry: `/proc/sys/net/ipv4/icmp_ratelimit`
Initialized to 1 * HZ in `icmp_sk_init()`

sysctl_icmp_ratemask

Mask made of ICMP types for which rates are being limited. Each bit is an ICMPv4 type.

`procfs` entry: `/proc/sys/net/ipv4/icmp_ratemask`
Initialized to 0x1818 in `icmp_sk_init()`

sysctl_icmp_errors_use_inbound_ifaddr

The value of this variable is checked in `icmp_send()`. When it's not set, the ICMP error messages are sent with the primary address of the interface on which the packet will be sent. When it is set, the ICMP message will be sent with the primary address of the interface that received the packet that caused the icmp error.

`procfs` entry: `/proc/sys/net/ipv4/icmp_errors_use_inbound_ifaddr`
Initialized to 0 in `icmp_sk_init()`

■ **Note** See also more about the ICMP `sysctl` variables, their types and their default values in `Documentation/networking/ip-sysctl.txt`.

Creating "Destination Unreachable" Messages with iptables

The `iptables` userspace tool enables us to set rules which dictate what the kernel should do with traffic according to filters set by these rules. Handling `iptables` rules is done in the netfilter subsystem, and is discussed in Chapter 9. One of the `iptables` rules is the `reject` rule, which discards packets without further processing them. When setting an `iptables reject` target, the user can set a rule to send a "Destination Unreachable" ICMPv4 messages with various codes using the `-j REJECT` and `--reject-with` qualifiers. For example, the following `iptables` rule will discard any packet from any source with sending back an ICMP message of "ICMP Host Prohibited":

```
iptables -A INPUT -j REJECT --reject-with icmp-host-prohibited
```

These are the possible values to the `--reject-with` qualifier for setting an ICMPV4 message which will be sent in reply to the sending host:

```
icmp-net-unreachable   - ICMP_NET_UNREACH
icmp-host-unreachable  - ICMP_HOST_UNREACH
icmp-port-unreachable  - ICMP_PORT_UNREACH
icmp-proto-unreachable - ICMP_PROT_UNREACH
icmp-net-prohibited    - ICMP_NET_ANO
icmp-host-prohibited   - ICMP_HOST_ANO
icmp-admin-prohibited  - ICMP_PKT_FILTERED
```

You can also use `--reject-with tcp-reset` which will send a TCP RST packet in reply to the sending host.

(`net/ipv4/netfilter/ipt_REJECT.c`)

With `ip6tables` in IPv6, there is also a REJECT target. For example:

```
ip6tables -A INPUT -s 2001::/64 -p ICMPv6  -j REJECT --reject-with icmp6-adm-prohibited
```

These are the possible values to the `--reject-with` qualifier for setting an ICMPv6 message which will be sent in reply to the sending host:

```
no-route, icmp6-no-route              - ICMPV6_NOROUTE.
adm-prohibited, icmp6-adm-prohibited  - ICMPV6_ADM_PROHIBITED.
port-unreach, icmp6-port-unreachable  - ICMPV6_NOT_NEIGHBOUR.
addr-unreach, icmp6-addr-unreachable  - ICMPV6_ADDR_UNREACH.
```

(`net/ipv6/netfilter/ip6t_REJECT.c`)